Heavy Nodes in a Small Neighborhood: Algorithms and Applications

Huiping Chen* Grigorios Loukides* Robert Gwadera† 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* incident 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. As the size of $\mathcal{G}$ can be very large in practice, we also develop 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 optimal or near-optimal solutions, outperforming a natural baseline, and that they can detect money laundering much more effectively and efficiently than a state-of-the-art method.

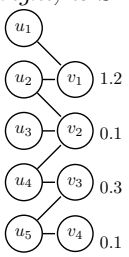
Keywords: Minimum k union, Dense subgraph discovery, Combinatorial optimization, Money laundering detection.

1 Introduction

The minimum k union problem [7, 8] 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 [8], 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 incident to the nodes in S). We introduce a weighted, unconstrained 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 $w : V \to \mathbb{R}_{\geq 0}$, find a set $S \subseteq V$ of nodes such that $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ is maximized, where $N(S) = \{u : \exists (u, v) \in E \wedge u \in U, v \in S\}$.*

Example 1. *The solution of* HNSN *on the graph below, where the weights of nodes in V appear on their right, is $S = \{v_1\}$, since $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ is maximized to $\frac{1.2}{2}$.*



(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 the following 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 [23], 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 the *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 [19].

A.2 *Team formation* [26]: 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 incident 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* [5, 6]: Smurfing is a key money laundering technique [14, 22, 17, 21]. It

*King's College London, huiping.chen@kcl.ac.uk, gloukides@acm.org
†Independent Researcher
‡CWI and Vrije Universiteit Amsterdam, solon.pissis@cwi.nl

starts with a criminal (detection target) who distributes their illicit money into peers. These peers deposit the money into their bank accounts and perform transactions with other peers who then transfer the money, or a large part of it, into the bank account of the criminal. 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. 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 4). Furthermore, a suspicious peer corresponding to v typically receives money from few peers corresponding to nodes in U [17, 14], since 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 both 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 [8], we show that HNSN can be solved *exactly in polynomial time*. In particular, we design a highly non-trivial Linear Programming (LP) algorithm for HNSN and prove that it solves HNSN exactly.

2. We design a greedy heuristic for HNSN and show that it can be implemented in near *linear time*.

3. We examine how smurfing attacks can be detected based on HNSN. Beyond the single-target attack in A3, 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.

4. We conducted experiments on 5 real datasets and synthetic ones showing that our algorithms find optimal or near-optimal solutions to HNSN, outperforming a natural baseline [20] on effectiveness and/or efficiency. We also show, using a real dataset, that our algorithms can detect money laundering substantially more effectively and efficiently than a state-of-the-art method [17].

Contributions 1, 2, 3, and 4 are presented in Sections 2, 3, 4, and 6. We defer related work to Section 5.

2 Exact LP Algorithm

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 incident 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:

$$\max \quad \Big(\sum_{i=1}^{|V|} w_i \cdot x_i\Big) \Big/ \sum_{j=1}^{|U|} y_j \tag{2.1a}$$

$$\text{s.t.} \quad \sum_{j=1}^{|U|} y_j > 0 \tag{2.1b}$$

$$x_i \le y_j, \quad (u_j, v_i) \in E \tag{2.1c}$$

$$x_i \in \{0,1\}, \quad i \in [1, |V|] \tag{2.1d}$$

$$y_j \in \{0,1\}, \quad j \in [1, |U|]. \tag{2.1e}$$

Constraint 2.1b 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 2.1c is added because, if a node v_i is contained in S, all its incident nodes are contained in $N(S)$ and, if v_i is not contained in S, its incident nodes may still be contained in $N(S)$. At this point, it is not obvious that we can solve the program in Eq. 2.1 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 [4]. That is, we introduce a variable $y_0 = \frac{1}{\sum_{j=1}^{|U|} y_j} > 0$, which can be defined due to Constraint 2.1b, and using y_0 we rewrite the program in Eq. 2.1 as:

$$\max \quad y_0 \cdot \sum_{i=1}^{|V|} w_i \cdot x_i \tag{2.2a}$$

$$\text{s.t.} \quad y_0 \cdot \sum_{j=1}^{|U|} y_j = 1 \tag{2.2b}$$

$$\sum_{j=1}^{|U|} y_j > 0 \tag{2.2c}$$

$$x_i \le y_j, \quad (u_j, v_i) \in E \tag{2.2d}$$

$$x_i \in \{0,1\}, \quad i \in [1, |V|] \tag{2.2e}$$

$$y_j \in \{0,1\}, \quad j \in [1, |U|] \tag{2.2f}$$

$$y_0 \in \mathbb{R}_{>0}. \tag{2.2g}$$

Second, we observe that Constraint 2.2c is redundant, since it is satisfied from the requirement $y_0 > 0$ (Constraint 2.2g) and from Constraint 2.2b. We also relax the integrality requirement for all variables x_i and y_j, writing the program in Eq. 2.2 as the following non-linear program:

$$\max \quad y_0 \cdot \sum_{i=1}^{|V|} w_i \cdot x_i \tag{2.3a}$$

$$\text{s.t.} \quad y_0 \cdot \sum_{j=1}^{|U|} y_j = 1 \tag{2.3b}$$

$$x_i \le y_j, \quad (u_j, v_i) \in E \tag{2.3c}$$

$$x_i \in [0,1], \quad i \in [1, |V|] \tag{2.3d}$$

$$y_j \in [0,1], \quad j \in [1, |U|] \tag{2.3e}$$

$$y_0 \in \mathbb{R}_{>0}. \tag{2.3f}$$

Last, by setting $z_i = y_0 \cdot x_i$ and $q_j = y_0 \cdot y_j$ in the program of Eq. 2.3, we get the Linear Program below. We obtain Constraint 2.4c by multiplying both parts of Constraint 2.3c by y_0 and substituting with z_i and q_j.

$$\text{(2.4a)} \qquad \max \quad \sum_{i=1}^{|V|} w_i \cdot z_i$$

$$\text{(2.4b)} \qquad \text{s.t.} \quad \sum_{j=1}^{|U|} q_j = 1$$

$$\text{(2.4c)} \qquad z_i \leq q_j, \quad (u_j, v_i) \in E$$

$$\text{(2.4d)} \qquad z_i \in [0,1], \quad i \in [1, |V|]$$

$$\text{(2.4e)} \qquad q_j \in [0,1], \quad j \in [1, |U|].$$

2.1 LP Optimality and Complexity. While LP relaxations do not generally lead to optimal solutions, we prove that the LP in Eq. 2.4 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 [3] but our problem and its LP formulation are quite different.

LEMMA 2.1. (LOWER BOUND) *For any set $S \subseteq V$, the value of the program in Eq. 2.4 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 $\bar{q}_j = 0$. Since $\sum_{j=1}^{|U|} \bar{q}_j = \sum_{j:u_j \in N(S)} \bar{q}_j = |N(S)| \cdot \frac{1}{|N(S)|} = 1$, Constraint 2.4b 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, it holds $\sum_{i=1}^{|V|} w_i \cdot \bar{z}_i = \sum_{i:v_i \in S} w_i \cdot \bar{z}_i = \frac{\sum_{v \in S} w(v)}{|N(S)|}$. 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 $\sum_{i=1}^{|V|} w_i \cdot \bar{z}_i = \frac{\sum_{v \in S} w(v)}{|N(S)|}$, we have $\sum_{i=1}^{|V|} w_i \cdot z_i \geq \frac{\sum_{v \in S} w(v)}{|N(S)|}$. ∎

LEMMA 2.2. (UPPER BOUND) *Given a feasible solution of the program in Eq. 2.4 with value τ, we can construct $S \subseteq V$ such that $\frac{\sum_{v \in S} w(u)}{|N(S)|} \geq \tau$.*

Proof. Consider a feasible solution $(\bar{z}, \bar{q})$ to the program in Eq. 2.4. Let $n^-(v_i) \subseteq U$ be the set of incident nodes to a node $v_i \in V$ and $n^+(u_j) \subseteq V$ be the set of incident 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 2.4c holds, for any $u_j \in U$ that is incident to v_i and $\bar{z}_i$ must be maximal due to Eq. 2.4a.

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 $N(r)$ is comprised of the indices of nodes that are incident to the nodes whose indices are in $S(r)$. 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 incident 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.

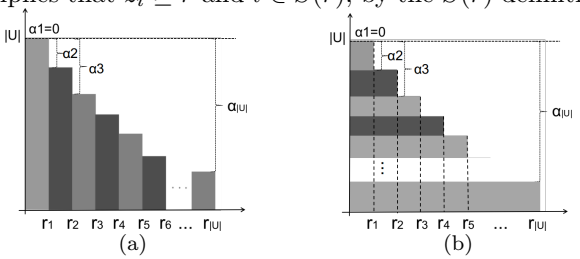


Figure 1: Illustration of (a) the first and (b) the second equality of Eq. 2.5.

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. 1a). 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, Eq. 2.5 holds (see also Fig. 1):

$$\text{(2.5)} \qquad \int_0^\infty |N(r)|\, dr = (|U| - \alpha_1) \cdot r_1 + (|U| - \alpha_2) \cdot (r_2 - r_1) + \ldots + (|U| - \alpha_{|U|}) \cdot (r_{|U|} - r_{|U|-1}) = r_1(\alpha_2 - \alpha_1) + r_2(\alpha_3 - \alpha_2) + \ldots + r_{|U|}(|U| - \alpha_{|U|}) = \sum_{j=1}^{|U|} \bar{q}_j.$$

The first equality holds due to the definition of $N(r)$ and r_j's, and the second one holds trivially (see also Fig. 1b). 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 $r_1(\alpha_2 - \alpha_1) + \ldots + r_{|U|}(|U| - \alpha_{|U|})$ 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, $r_1(\alpha_2 - \alpha_1) + \ldots + r_{|U|}(|U| - \alpha_{|U|}) = \sum_{j=1}^{|U|} \bar{q}_j$.

In addition, $\sum_{j=1}^{|U|} \bar{q}_j = 1$ due to Eq. 2.4b, 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 Eq. 2.4. 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 $\frac{\sum_{v \in S} w(v)}{|N(S)|} \geq \tau$. ∎

Putting Lemmas 2.1 and 2.2 together, we obtain:

THEOREM 2.1. (OPTIMALITY AND CONSTRUCTION) *Let OPT be the value of an optimal solution to the program in Eq. 2.4. Then, the following holds:*

$$\max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|} = OPT. \tag{2.6}$$

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 2.1, OPT$\geq \max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|}$ (consider the set $S^* = \arg\max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|}$). Due to Lemma 2.2, $\max_{S \subseteq V} \frac{\sum_{v \in S} w(v)}{|N(S)|} \geq$ OPT. Thus, Eq. 2.6 holds. The proof of Lemma 2.2 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 Eq. 2.4. ∎

Our algorithm solves the program in Eq. 2.4 and then constructs an optimal solution S. Since this program is linear, its solution can be obtained in $O(n^{2.5})$ time [25], where n is the total number of variables and constraints; see [12] for the state of the art. S can be constructed as in the proof of Lemma 2.2. However, it is also possible to construct S in $O(|V|)$ time given an optimal solution (z^*, q^*) of the program in Eq. 2.4. 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 Eq. 2.4, (x^*, y^*) is optimal for the program in Eq. 2.2, 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 Eq. 2.2, 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 Eq. 2.4c, 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 Eq. 2.2a. Thus, for the S we constructed, $y_0^* \cdot \sum_{i=1}^{|V|} w_i \cdot x_i^* = \frac{1}{\sum_{j:u_j \in N(S)} y_j^*} \cdot \sum_{i=1}^{|V|} w_i \cdot x_i^* = \frac{1}{|N(S)|} \cdot \sum_{i=1}^{|V|} w_i \cdot x_i^* = \frac{\sum_{i:v_i \in S} w_i}{|N(S)|}$. Since the constraints of the program in Eq. 2.2 are satisfied, Eq. 2.2c 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.

Algorithm 1 GREEDY$(\mathcal{G}, w)$

1: $i \leftarrow 0$; $R_i \leftarrow V$
2: **while** $R_i \neq \varnothing$ **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: **return** $S \leftarrow \arg\max_{R_j : j \in [0, |V|)} \frac{\sum_{v \in R_j} w(v)}{|N(R_j)|}$

3 Greedy Heuristic

GREEDY is a "peeling" algorithm for HNSN (see Algorithm 1), which: (I) iteratively removes from the 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; and (III) outputs a best set S of remaining nodes over all iterations.

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 $v \in V$ 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 incident 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). Last, it returns the set of remaining nodes with a largest objective value over all iterations (Line 9). This is performed because the objective function of HNSN is clearly non-monotone (i.e., it may increase or decrease after a node removal).

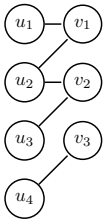
Efficient Implementation. A direct implementation of GREEDY takes $O(|E| \cdot |V|)$ time because computing all ratios requires traversing $\mathcal{G}$, which takes $O(|V| + |E|) = O(|E|)$ time per iteration. We next show a near linear-time implementation of GREEDY.

Line 3. We observe 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 $N(R_i)$. This is because $|N(R_i \setminus \{v\})| = |N(R_i)| - |N(\{v\})| + |N(R_i \setminus \{v\}) \cap N(\{v\})|$ and thus $|N(R_i)| - |N(R_i \setminus \{v\})|$ is equal to $|N(\{v\})| - |N(R_i \setminus \{v\}) \cap N(\{v\})|$, which is exactly the number of neighbors of v with degree 1. Thus, before Line 2, we build a perfect hashtable with an entry for each node $v \in V$ that has at least one neighbor in U with degree 1, in $O(|V|)$ total time. If v is removed in Line 7, we delete its entry from the hashtable in $O(1)$ time. We also check the degree of each neighbor $u \in U$ of v and if the degree of u has become 1, we add the *single* neighbor of u in the hashtable in $O(1)$ time. This takes $O(|E|)$ time over all iterations. We can thus perform the check in Line 3 in $O(1)$ time: if the hashtable is nonempty, the check succeeds. Therefore, the total time for Line 3 over all iterations is $O(|V| + |E|) = O(|E|)$.

Line 4. The key is to: (I) identify nodes whose ratio must be updated in an iteration; and (II) update their ratios and find a minimum ratio, instead of computing all ratios from scratch to find a minimum.

Regarding issue I, we noted above that, for a node v, the denominator in Line 4 in iteration i is equal to the number of neighbors of v with degree 1, in the graph induced by the nodes in R_i and those in $N(R_i)$. Let v' be the node that was removed in iteration $i-1$. We observe that the denominator for v changes during iteration i *only* when v and v' have at least one common neighbor whose degree *after the removal* is 1. This is because removing v' reduces the degree of each neighbor of v' by 1 and the neighbors in $N(\{v\}) \cap N(\{v'\})$ with degree 2 before the removal (and thus degree 1 after) are considered in the denominator for v. Thus, to identify every node v whose ratio must be updated in iteration i, we find the common neighbors between v' and v having degree 2 before the removal, for each $v \in R_i$.

EXAMPLE 2. *In the graph below, where the weights of nodes in V appear on their right, $R_0 = V = \{v_1, v_2, v_3\}$.* GREEDY *first removes v_1 from R_0; thus $R_1 = \{v_2, v_3\}$. The ratio for v_2 needs updating, as v_2 and v_1 have a common neighbor, u_2, which has degree 1 after the removal of v_1. In fact, $\frac{w(v_2)}{|N(R_0)|-|N(R_0 \setminus \{v_2\})|} = \frac{1}{1}$ and $\frac{w(v_2)}{|N(R_1)|-|N(R_1 \setminus \{v_2\})|} = \frac{1}{2}$. The ratio for v_3 does not need updating; $\frac{w(v_3)}{|N(R_0)|-|N(R_0 \setminus \{v_3\})|} = \frac{w(v_3)}{|N(R_1)|-|N(R_1 \setminus \{v_3\})|}$.*



Regarding issue II, during a preprocessing step, we identify every node $v \in V$ such that $|N(R_i)| - |N(R_i \setminus \{v\})| > 0$. For each such node v, we add: (1) its ratio $\frac{w(v)}{|N(R_i)|-|N(R_i \setminus \{v\})|}$ in a height-balanced binary search tree $\mathcal{T}$; and (2) an entry in a perfect hashtable $\mathcal{H}$ with key v and value a pointer to the ratio of v in $\mathcal{T}$. After removing a node v' in iteration $i-1$ (Line 7), we find the nodes whose ratios need updating, which we call *affected* nodes, by finding each neighbor $u \in U$ of v' that has degree 2 and then the other neighbor of u. Next, we search each affected node in $\mathcal{H}$. If an affected node is contained in $\mathcal{H}$, we update its ratio in $\mathcal{T}$, following the pointer; otherwise, we add entries for this affected node in $\mathcal{H}$ and in $\mathcal{T}$.

Note that: (1) Each removed node v' in iteration $i-1$ changes the ratio of $O(|N(\{v'\})|)$ nodes in R_i. Thus, there are $O(|E|)$ affected nodes over all iterations and finding them takes $O(|E|)$ time. (2) If an affected node v is found in $\mathcal{H}$, its ratio is updated in $O(\log(|V|))$ time: $O(1)$ time for finding v in $\mathcal{H}$; and $O(\log(|V|))$ for updating its ratio in $\mathcal{T}$. Otherwise, the entries for v in $\mathcal{H}$ and in $\mathcal{T}$ are added in $O(\log(|V|))$ total time. Thus, the total cost of Line 4 over all iterations is $O(|E| \cdot \log(|V|))$.

Line 6. During a preprocessing step, we identify every node $v \in V$ that has no neighbor in U with degree 1 in $O(|E|)$ time. For each such node v, we add: (1) an entry $\langle \frac{w(v)}{|N(\{v\})|}, v \rangle$ in a doubly linked list $\mathcal{L}$; and (2) an entry v in a perfect hashtable $\mathcal{H}_{\mathcal{L}}$ whose key is v and value is a pointer to the entry of v in $\mathcal{L}$. We sort $\mathcal{L}$ in increasing order in terms of ratio. To implement Line 6, we iterate over $\mathcal{L}$ and check if the node in the second element of the current entry of $\mathcal{L}$ has been selected in Line 4. In this case, we delete its entries from $\mathcal{L}$ and from $\mathcal{H}_{\mathcal{L}}$, and continue iterating. Otherwise, we select this node as v, delete its entries from $\mathcal{L}$ and from $\mathcal{H}_{\mathcal{L}}$ and stop iterating. Note that $\mathcal{L}$ remains sorted; as we do not delete nodes in U, the denominators of the ratios, and hence the ratios, do not change. Thus, the extra time required by Line 6 over all iterations is $O(|V| \cdot \log(|V|))$, the time to build and sort $\mathcal{L}$.

Line 7. We remove v from R_i. Then, we find and deal with the affected nodes, as discussed before.

Line 9. This line can be implemented in $O(|E|)$ time, by computing the numerator and denominator of $\frac{\sum_{v \in R_j} w(v)}{|N(R_j)|}$ decrementally as we remove nodes. Specifically, we maintain an array A storing the degree of each node in U. We also maintain the number P_A of elements of the array A with positive value. After removing v (Line 7), we update the degrees of the nodes in $N(\{v\})$ in A and also P_A, if needed. Since in every iteration i, it holds that $|N(R_i)| = P_A$, we compute $\frac{\sum_{v \in R_i} w(v)}{|N(R_i)|}$ as $\frac{\sum_{v \in R_i} w(v)}{P_A}$ and maintain the maximum over all iterations and its corresponding R_i. The latter R_i will be returned in Line 9. This decremental update takes $O(|E|)$ time amortized over all iterations because each node in U is accessed as many times as its degree, and

the total degree of all nodes in U is $|E|$.

Based on the above, we obtain the following result:

THEOREM 3.1. GREEDY *can be implemented in* $O(|E| \cdot \log(|V|))$ *time.*

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.

4 Applications to Money Laundering Detection

Money Laundering (ML) is the process of transforming crime profits into legitimate assets [5]. ML involves three steps [5]: (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 [17, 21, 14, 22] 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.

We focus on detecting two types of smurfing attacks considered in [17, 14, 21]: *single-target* and *multi-target*.

Single-target Attack. This attack has been discussed in Introduction 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 [14, 16].

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 incident 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 account holders may have a large account balance) [21]. We assume $i_v + b_v > 0$; e.g., banks require a non-zero balance in order to maintain an account.

If a solution $S \subseteq V$ 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 two facts: (I) Smurfs receive money from few others [17, 14]. 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 [17, 22].

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 $M(S) = \{w : \exists (v, w) \in E \wedge v \in S, w \in W\}$ is the subset of nodes in W that receive money from the nodes in S. A set $S \subseteq V$ 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 [21]). Our measure is fundamentally different from that of [17] and offers three benefits: (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 incident node in* U *and at least one incident node in* W*, and a weight function* $w : V \to \mathbb{R}_{\geq 0}$*, find a set* $S \subseteq V$ *of nodes such that* $\frac{\sum_{v \in S} w(v)}{|N(S)| + |M(S)|}$ *is maximized, where* $N(S) = \{u : \exists (u, v) \in E \wedge u \in U, v \in S\}$ *and* $M(S) = \{w : \exists (v, w) \in E \wedge v \in S, w \in W\}$.

Each $v \in V$ must have at least one incident node in U and at least one incident node in W, as we are interested in detecting money flows from nodes in U to target nodes 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, $w(v) = \frac{o_u}{i_u + b_u}$, 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 deferred to the full version of the paper.

5 Related Work

Related Problems. HNSN is a weighted and unconstrained variant of the minimum k union problem, which has been studied theoretically in [7, 8]. As discussed in

Introduction, 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 [8]. 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 [15], for a survey). Two well-known such problems are the densest subgraph problem [3], which asks for a subgraph with maximum average degree, and the densest k-subgraph problem [9], 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 [11] and the densest k-subhypergraph problem [8, 7]. Both are well studied [11, 2, 24, 8].

Related Applications. Money laundering detection [5] can be performed using rule-based or machine-learning approaches (see [6], for a survey). Our work is more relevant to approaches for detecting smurfing attacks from graph data [14, 21, 22, 17]. The approaches of [14] and [21] output all subgraphs that are isomorphic to certain graph patterns. Unlike [14] and [21], our approach outputs the most suspicious subgraph, without restricting its topology to that of specific patterns. The approach of [22] considers a streaming setting and tracks statistics for each account over time to identify suspicious accounts. The closest work to ours is [17]. It proposes a suspiciousness measure, which has some weaknesses, addressed by our measure (see Section 4), and FLowScope, a greedy algorithm that is less effective and efficient than our algorithms (see Section 6).

Profitable product set discovery falls into the area of utility mining [10]. 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 [19].

6 Experimental Evaluation

Datasets. For the profitable product set discovery application, we used 5 real, benchmark datasets, obtained from `https://bit.ly/3S5y0de`, and synthetic datasets (see Table 1). The real datasets contain utilities (profits) of items (nodes in U) which we summed up to obtain the transaction (node $v \in V$) utility, following [23]. The synthetic datasets model bipartite graphs with given $|U|$, $|V|$, and $|E|$, in which edges are formed between randomly selected nodes. The degrees of nodes in V are in $[1, 50]$ and sampled from a Zipfian distribution with parameter 0.1; such power-law degree distributions are often encountered in real graphs. The largest synthetic dataset is called SYN. For the money laundering detection application, we used the Czech Financial Dataset (CFD), which contains anonymous money transfers of a Czech bank. It has been used in [17, 22] 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 $\mathcal{G}(U, V, W, E)$.

Dataset	$\lvert U\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
Foodmart (FM)	1,559	4,141	18,319
E-commerce (EC)	3,468	14,975	174,354
Liquor (LI)	4,026	52,131	410,609
Fruithut (FR)	1,265	181,970	652,773
YooChoose (YC)	107,256	234,300	507,266
SYN	2,000	500,000	50,000,000
Czech Financial Dataset (CFD)	60,606	1,496	138,256

Table 1: Dataset characteristics.

Setup. We evaluated our algorithms, Exact LP (LP), Greedy (GR), and FastGreedy (FGR), by comparing them to GreedRatio [1, 20] (GRR), a state-of-the-art method for minimizing a ratio of non-negative monotone submodular functions. As in [1], we ensured that there is at least one $v \in V$ with $w(v) > 0$ and applied GRR 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 [13]); 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) [13]. GRR takes $O(|E| \cdot |V|)$ time and cannot be sped up by lazy evaluation [18], which is applicable for minimizing the ratio of a modular to a submodular function [1]. In addition, we compared our algorithms to FlowScope (FL) [17] in the context of money laundering detection. FL takes $O(|E| \cdot \log(|V|))$ time when applied to a tripartite graph as in our case. We did not compare to [21, 14, 22], as they deal with different problems and/or are applied to different settings (see Section 5).

All experiments ran on an AMD EPYC 7702 CPU with 256GB RAM. We implemented our algorithms in `C++` and used Gurobi 9.5.2 in LP. The implementation of FL was obtained from `https://bit.ly/3LHigea`. See `https://bitbucket.org/hnsn/sdm2023/` for our code and datasets.

Profitable Product Set Discovery. We first examined the effectiveness and efficiency of all methods on the 5 real datasets. Fig. 2a shows the scores for all methods in HNSN (i.e., $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ for a solution S of HNSN), and Fig. 2b shows the corresponding runtimes. LP performed the best, especially in the FR and YC datasets, as it is *exact*. Notably, LP was faster than the competitor GRR, in EC, FR, and YC (e.g., up to

two orders of magnitude faster in the largest dataset YC). Our GR and FGR algorithms performed similarly to GRR in terms of effectiveness but were *three orders of magnitude faster* on average, as they take $O(|E| \cdot \log(|V|))$ and $O(|E| + |V| \cdot \log(|V|))$ time, respectively, whereas GRR takes $O(|E| \cdot |V|)$ time.

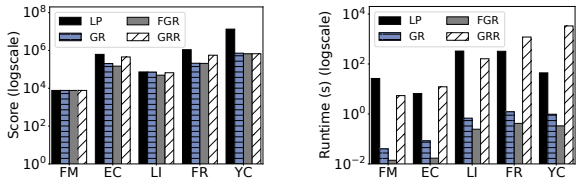


Figure 2: (a) Score $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ for all methods on real datasets. (b) Runtime (s) on real datasets.

We also used synthetic datasets. To thoroughly evaluate effectiveness, we applied all algorithms on 500 graphs, each constructed by sampling a number of records (transactions) uniformly at random from a dataset. We performed this process for all datasets, sampling 10,000 transactions from LI and YC, and 1,000 from all other datasets. The results in Fig. 3 are analogous to those in Fig. 2a. LP was the best, GR outperformed GRR in the FM and YC datasets (the difference was statistically significant at p value < 0.005) and performed similarly in the other datasets (the difference was not statistically significant at p value < 0.005). GR constructed solutions within 76.7% from the optimal, on average over all tested graphs. FGR was slightly less effective than GR.

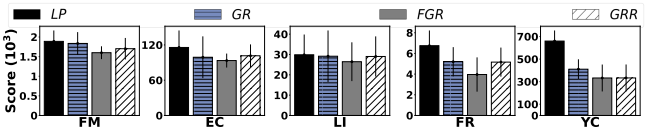


Figure 3: Score $\frac{\sum_{v \in S} w(v)}{|N(S)|}$ on random subgraphs, produced from all real datasets.

In addition, we measured runtime using subgraphs of SYN with varying $|E|$ or $|V|$ (see Fig. 4). Again, GR and FGR were faster than GRR by several orders of magnitude. For example, GR and FGR needed less than 2 minutes and 5 seconds, respectively, to process SYN, which contains 50 million edges, while GRR needed 2.6 hours. LP was faster than GRR by 2.1 times on average, albeit exact and much more effective. FGR scaled better than GR with $|E|$ (see Fig. 4a) and performed similarly when $|V|$ was close to $|E|$ (see the last point in Fig. 4b), since FGR needs $O(|E| + |V| \cdot \log(|V|))$ time and GR needs $O(|E| \cdot \log(|V|))$ time.

In sum, LP is exact and reasonably fast, GR is slightly less effective but substantially faster, and FGR trades some effectiveness for much higher efficiency. GRR is as effective as GR is but slower than LP and orders of magnitude slower than our other methods.

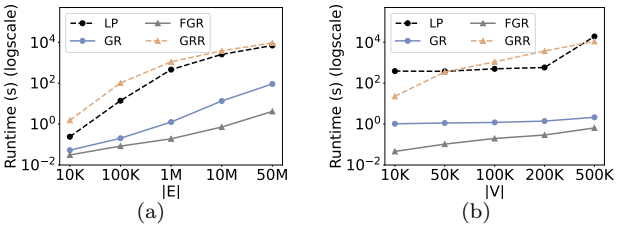


Figure 4: Runtime (s) on subgraphs of SYN with $|U| = 2000$ and varying: (a) $|E|$ for $|V| = 10^5$; and (b) $|V|$ for $|E| = 10^6$.

Money Laundering Detection. We first detected the most suspicious subgraph from the CFD dataset using each of our methods. All found the same solution; a subgraph with 21 nodes (8 source, 1 middle, and 12 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 [17, 21]. Also, this subgraph is 3.22 times more suspicious according to our measure.

We also compared our algorithms with FL, following the methodology of [22]. That is, we injected a fraudulent pattern in CFD and examined whether or not it can be detected. We injected two types of patterns from [21], modeling single and multi-target attacks; see Fig. 5. We generated over $1,000$ different patterns. To favor FL, we used balance $b_v = 0$, for each $v \in V$.

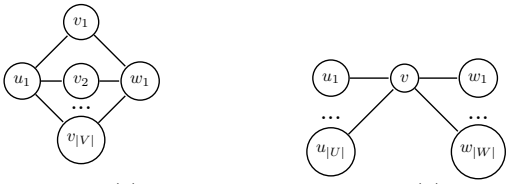


Figure 5: (a) Single-target pattern. (b) Multi-target pattern. The notation "..." denotes additional nodes and edges.

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 $|V| \in \{1, 2, 4, 6, ..., 18\}$ and $\mu \in \{0.5, 0.55, 0.6, ..., 1\}$. All our algorithms detected all injected patterns, whereas FL did not detect *any* injected pattern when $|V| \geq 8$.

For the multi-target patterns, we considered two configurations. In the first, we tried all combinations of $|U|, |V| \in \{1, 2, \ldots, 6\}$ and $w(v) \in \{0.5, 0.55, \ldots, 1\}$. 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 configuration, we sampled $w(v) \in (0, 1]$ from a normal distribution $\mathcal{N}(\mu, \sigma = 0.1)$ with $\mu \in \{0.7, 0.75, \ldots, 0.9\}$. For each mean value μ, we

constructed 100 patterns with $|U|$ and $|V|$, selected uniformly at random from $[1, 6]$. As can be seen in Table 2, all our methods detected *all* injected patterns, while FL detected between 18% and 66% of them.

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 2: Percentage of detected fraudulent patterns by our methods and by FL [17] for different mean values.

Our methods were much faster than FL; FGR was faster than GR and LP was slower than GR. For single-target patterns, FL needed on average 8.5 seconds, while LP 1.2 seconds; GR and FGR needed 0.14 and 0.12 seconds, respectively. For multi-target patterns, FL needed on average 6.3 seconds, while LP 1.5 seconds; GR and FGR needed 0.14 and 0.11 seconds, respectively.

In sum, our methods detected a more suspicious graph than FL from the real dataset CFD and also detected injected patterns more accurately and efficiently.

Acknowledgments

H.C. is supported by a CSC Scholarship. We would like to thank Hilde Verbeek for helpful discussions on Lemma 2.2.

References

[1] W. Bai et al. "Algorithms for Optimizing the Ratio of Submodular Functions". In: *ICML*. 2016, pp. 2751–2759.

[2] S. K. Bera et al. "A New Dynamic Algorithm for Densest Subhypergraphs". In: *WWW*. 2022, pp. 1093–1103.

[3] M. Charikar. "Greedy Approximation Algorithms for Finding Dense Components in a Graph". In: *APPROX*. 2000, pp. 84–95.

[4] A. Charnes and W. W. Cooper. "Programming with linear fractional functionals". *Naval Research Logistics Quarterly* 3-4 (1962), pp. 181–186.

[5] C. S. Chaudhari. *A Guide to Risk Based Internal Audit System in Banks*. Notion Press, Inc., 2017.

[6] Z. Chen et al. "Machine Learning Techniques for Anti-Money Laundering (AML) Solutions in Suspicious Transaction Detection: A Review". *Knowl. Inf. Syst.* 2 (2018), pp. 245–285.

[7] E. Chlamtác, M. Dinitz, and Y. Makarychev. "Minimizing the Union: Tight Approximations for Small Set Bipartite Vertex Expansion". In: *SODA*. 2017, pp. 881–899.

[8] E. Chlamtác et al. "The Densest k-Subhypergraph Problem". *SIAM Journal on Discrete Mathematics* 2 (2018), pp. 1458–1477.

[9] U. Feige, G. Kortsarz, and D. Peleg. "The Dense k-Subgraph Problem". *Algorithmica* 3 (2001), pp. 410–421.

[10] W. Gan et al. "A Survey of Utility-Oriented Pattern Mining". *TKDE* 4 (2021), pp. 1306–1327.

[11] D. J.-H. Huang and A. B. Kahng. "When clusters meet partitions: new density-based methods for circuit decomposition". In: *ED&TC*. 1995, pp. 60–64.

[12] S. Jiang et al. "A Faster Algorithm for Solving General LPs". In: *STOC*. 2021, pp. 823–832.

[13] A. Krause and D. Golovin. "Submodular function maximization". In: *Tractability*. 2013.

[14] M.-C. Lee et al. "AutoAudit: Mining Accounting and Time-Evolving Graphs". In: *BigData*. 2020, pp. 950–956.

[15] V. E. Lee et al. "A Survey of Algorithms for Dense Subgraph Discovery". In: *Managing and Mining Graph Data*. 2010, pp. 303–336.

[16] M. Levi and P. Reuter. "Money Laundering". *Crime and Justice* (2006), pp. 289–375.

[17] X. Li et al. "FlowScope: Spotting Money Laundering Based on Graphs". In: *AAAI*. 2020, pp. 4731–4738.

[18] M. Minoux. "Accelerated greedy algorithms for maximizing submodular set functions". In: *Optimization Techniques*. 1978, pp. 234–243.

[19] A. Payne et al. *Relationship marketing for competitive advantage*. Butterworth-Heinemann, 1995.

[20] C. Qian et al. "Optimizing Ratio of Monotone Set Functions". In: *IJCAI*. 2017, pp. 2606–2612.

[21] M. Starnini et al. "Smurf-Based Anti-money Laundering in Time-Evolving Transaction Networks". In: *ECML/PKDD*. 2021, pp. 171–186.

[22] X. Sun et al. "MonLAD: Money Laundering Agents Detection in Transaction Streams". In: *WSDM*. 2022, pp. 976–986.

[23] V. S. Tseng et al. "Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases". *TKDE* 8 (2013), pp. 1772–1786.

[24] C. E. Tsourakakis. "The K-clique Densest Subgraph Problem". In: *WWW*. 2015, pp. 1122–1132.

[25] P. M. Vaidya. "Speeding-Up Linear Programming Using Fast Matrix Multiplication (Extended Abstract)". In: *FOCS*. 1989, pp. 332–337.

[26] K. Wang et al. "Efficient and Effective Community Search on Large-scale Bipartite Graphs". In: *ICDE*. 2021, pp. 85–96.