Optimizing Network Resilience via Vertex Anchoring

Siyi Teng§
Guangzhou University
Chinese University of Hong Kong
tsyyxxxka@gmail.com

Jiadong Xie§
Chinese University of Hong Kong
jdxie@se.cuhk.edu.hk

Fan Zhang*
Guangzhou University
zhangf@gzhu.edu.cn

Can Lu
Guangzhou University
lucan@gzhu.edu.cn

Juntao Fang
Guangzhou University
fangjuntaocs@e.gzhu.edu.cn

Kai Wang
ACEM, Shanghai Jiao Tong University
w.kai@sjtu.edu.cn

ABSTRACT
Network resilience is a critical ability of a network to maintain its functionality against disturbances. A network is resilient/robust when a large portion of the nodes are to be better engaged in the network, i.e., they are less likely to leave given the changes on the network. Existing studies validate that the engagement of a node can be well captured by its coreness on network topology. Therefore, it is promising to maximize the number of nodes with increasing coreness values. In this paper, we propose and study the follower maximization problem: maximizing the resilience gain (the number of coreness-increased vertices) via anchoring a set of vertices within a given budget. We prove that the problem is NP-hard and \(W[2]\)-hard, and it is NP-hard to approximate within an \(O(n^{1-\varepsilon})\) factor. We first propose an advanced greedy approach, followed by a time-dependent framework designed to quickly find high-quality results. The framework is initialized by the advanced greedy algorithm and incorporates novel techniques for optimizing the search space. The effectiveness and efficiency of our solution are verified with extensive experiments on 8 real-life datasets. Our source codes are available at https://github.com/Tsyxxxka/Follower-Maximization.

CCS CONCEPTS
- Theory of computation → Graph algorithms analysis.

KEYWORDS
Network resilience; User engagement; Core decomposition

1 INTRODUCTION
Network resilience refers to a network’s ability to adapt and endure changes, where node/user engagement is a key issue [15]. Many real-life networks are susceptible to dynamics [14], e.g., in a social network, there are often natural failures (the random leave of users due to their individual situations) and artificial attacks on the network (user attraction strategies from competing networks). The departure of users may contagiously affect the engagement of other users [26], which may even lead to the collapse of a network [15, 34]. Correspondingly, the users with increasing engagement may encourage the participation of other users who are thus less likely to leave the network. A network is resilient/robust if few nodes will leave the network given the negative changes. Thus, in order to sustain the resilience of a network, it is critical to identify and enhance node engagement to the greatest extent, e.g., maximize the number of engagement-enhanced nodes.

Real-world networks are usually modeled as graphs in which different graph characteristics are considered for the resilience study, e.g., centrality, connectivity, and diameter [14]. Despite the various proposed metrics like node influence, as we will discuss in Section 7, most of the metrics do not consider the engagement dynamic of nodes which is closely correlated with network resilience.

The \(k\)-core is a widely studied cohesive subgraph model for user engagement analysis, which is defined as a maximal subgraph where each vertex has at least \(k\) neighbors [27, 32]. Every vertex in the graph has a unique coreness value, i.e., the largest \(k\) such that the \(k\)-core contains the vertex. The core decomposition can compute the coreness of every vertex by iteratively removing the vertices with the smallest degree in the remaining graph. This procedure well captures the engagement dynamic of users in the unraveling of a network, and thus the coreness is validated as the "best practice" for measuring user engagement on graph structure [26]. As shown in Figure 1, in Gowalla social network [23], there is a clear correlation between coreness \(c\) and node engagement (represented by the average number of check-ins for all nodes with coreness \(c\)). Note that the correlation is also validated in other networks [39] and the outliers are due to the sparsity of the vertices with the same coreness.

Vertex anchoring is a common practice in recent studies to optimize the engagement of targeted users and improve the engagement of other users through contagious user interactions [5, 11, 24, 25]. We may provide incentives to key users s.t. they will be first “anchored” regarding user engagement and thus contagiously enhance the overall engagement of all the users. The degree of each anchored vertex can be considered as positive infinity, while its connections
Challenges and Contributions. To best of our knowledge, we are the first to study the problem of optimizing network resilience via vertex anchoring. We prove the problem is NP-hard and W[2]-hard parameterized by the budget $b$, and it is NP-hard to approximate the problem within an $O(n^{1-\epsilon})$ factor (Section 3). Although GAC proposed in [24] can be migrated to solve the FM problem by modifying the computation of coreness gain to resilience gain, denoted as GAC–FM, it is time-consuming in practice due to large search space and unspecific design of techniques especially on large datasets, e.g., it takes more than three days on the LiveJournal when $b = 100$.

To efficiently solve the FM problem, we propose a series of novel computing techniques: (a) We first propose AdvGreedy, an advanced greedy approach with high efficiency (Section 4). The key idea of AdvGreedy is to compute the followers of each candidate vertex based on shell component, which is more fine-grained than the core component tree used in GAC–FM; (b) For the time-consuming follower computation, we propose a novel explore-and-retract strategy in local core decomposition, which can scan as few candidate followers as possible for accelerating the algorithm; (c) In addition, we propose a tight upper bound to reduce the number of vertices that require exact follower computation; and (d) We further refine the upper bound of follower number by combining it with the reuse technique. Although our proposed AdvGreedy significantly outperforms GAC–FM in both running time and resilience gain (shown in Section 6), it does not have any approximate guarantees (Corollary 2). To bridge this gap, we propose a time-dependent framework equipped with AdvGreedy, which applies branch and bound searching to obtain a high-quality solution quickly and then continues exploring promising spaces to produce better answers (Section 5).

Extensive experiments conducted show that (i) our AdvGreedy is more effective than other heuristics including GAC–FM in improving resilience gain; (ii) AdvGreedy is faster than GAC–FM by more than 1 order of magnitude; and (iii) the time-dependent framework continues to produce better results over time (Section 6).

2 PRELIMINARIES

We consider a simple, undirected and unweighted graph $G = (V, E)$, where $V(G)$ (resp. $E(G)$) represents the set of vertices (resp. edges) in $G$. We denote $n = |V(G)|$, $m = |E(G)|$ and assume $m > n$. Let $G[V]$ denote the induced graph by the vertex set $V$. Given a vertex $v$ in a subgraph $S$ of $G$, $N(v, S)$ denotes the neighbor set of $v$ in $S$, i.e., $N(v, S) = \{u \mid (u, v) \in E(S)\}$. The degree of $v$ in subgraph $S$, i.e., $|N(v, S)|$, is denoted by $d(v, S)$.

Definition 1 ($k$-core $C_k$). Given a graph $G$ and an integer $k$, a subgraph $S$ is a $k$-core of $G$, if (i) each vertex $v \in S$ has at least $k$ neighbors in $S$, i.e., $d(v, S) \geq k$; and (ii) $S$ is maximal, i.e., any supergraph of $S$ is not a $k$-core except $S$ itself.

Note that $k$-core in this paper is not required to be connected as in [27, 32], we use $k$-core to represent all the subgraphs satisfying Definition 1. According to the definition of $k$-core, every vertex in the graph has a unique coreness value.

Definition 2 (coreness $c(\cdot)$). Given a graph $G$, the coreness of a vertex $u \in V(G)$, denoted by $c(u, G)$, is the largest $k$ such that $C_k(G)$ contains $u$, i.e., $c(u, G) = \arg\max_{k \in \mathbb{N}} u \in C_k(G)$.

A graph can be decomposed into a hierarchy where the vertices are distinguished and arranged by their coreness values.

Definition 3 (core decomposition). Given a graph $G$, core decomposition is to compute the coreness of every vertex in $V(G)$.

The core decomposition can be computed in $O(m)$ time by recursively removing the vertex with the smallest degree in the remaining graph, and updating the degrees of its neighbors by bin sort [2].

In this paper, once a vertex $x$ in graph $G$ is anchored, its degree is considered as positive infinity while its neighbor set is not changed. Every anchored vertex is called an anchor or an anchor vertex. An anchor will never be removed in core decomposition of $G$, and core decomposition with anchors can still be computed in $O(m)$ time.

The existence of anchors may raise the corenesses of other vertices in the core decomposition with anchors. Let $c^A(u)$ denote
the coreness of each vertex $u$ with the anchor set $A$. For each new anchor $x$, the coreness-increased vertices whose corenesses are not changed by previous anchors are the followers of $x$, i.e., the follower set of $x$ is $\{u \mid c^A(u) > c^A(u) \land c^A(u) = c(u)\}$, where $A$ is the anchor set before anchoring $x$. We assume the coreness of each anchor $x$ is increased by the anchoring, i.e., $c^A(x) > c^A(x)$.

**Definition 4 (resilience gain $g(\cdot)$).** Given a graph $G$ and the anchor set $A$, the resilience gain of $G$ regarding $A$, denoted by $g(A, G)$, is the number of followers by anchoring $A$, i.e., the number of vertices with coreness increased after anchoring $A$. We have $g(A, G) = |\{v \in V(G) \mid v \in A \land c^A(v) > c(v)\}|$.

**Definition 5 (follower maximization problem).** Given a graph $G$ and a budget $b$, the follower maximization (FM) problem aims to find a set $A$ of at most $b$ vertices in $G$, such that the resilience gain regarding $A$, i.e., $g(A, G)$, is maximized.

The state-of-the-art solution for the FM problem is to adapt the GAC algorithm [24]. The main idea is to replace the coreness gain with the resilience gain in greedy anchor selection and upper bound pruning. More details are given in the appendix.

## 3 PROBLEM ANALYSIS

In this section, we first prove the FM problem is NP-hard and hard to approximate in general graphs, i.e., there is no polynomial time algorithm to approximate the optimal solution within a factor of $O(n^{1-\epsilon})$, for every positive constant $\epsilon > 0$.

**Lemma 1.** It is NP-hard to distinguish between instances of the FM problem where the optimal solution has value $\Omega(n)$ versus when the optimal solution has value $O(b)$.

**Proof Sketch.** We prove the lemma through a reduction from the set cover decision (SCD) problem [21] to our FM problem, by constructing corresponding FM instances for any general SCD instances. The main idea of designing new FM instances is to construct graphs, which can get $O(b)$ resilience gain when the corresponding SCD instance is a no-instance, and can get $O(n)$ resilience gain when the corresponding SCD instance is a yes-instance. Since the SCD problem is NP-complete, Lemma 1 can be proved. □

**Corollary 1.** Given a graph $G$, the FM problem is NP-hard.

Lemma 1 further immediately indicates that there does not exist any $O(n^{1-\epsilon})$ approximate solution for the FM problem.

**Corollary 2. For any $\epsilon > 0$, it is NP-hard to approximate the FM problem on general graphs within an $O(n^{1-\epsilon})$ factor.**

From a parameterized perspective, we prove that the FM problem is W[2]-hard with respect to the budget parameter $b$.

**Theorem 3.** The FM problem is W[2]-hard parameterized by $b$.

## 4 AN ADVANCED GREEDY APPROACH

GAC [24] finds the followers of each anchor vertex $x$ in those $k$-core components which contain at least one neighbor of $x$ with the same or higher coreness as $x$. Better still, it develops reuse techniques and upper bound based on the $k$-core component. However, $k$-core component is not the atomic unit in finding the followers. We can find that if a vertex $u$ is a follower of an anchor $x$, then there exists at least one path from $x$ to $u$ (Lemma 2) s.t. all the vertices in the path except $x$ share the same coreness. Motivated by this, we propose the concept of shell components, which are connected subgraphs of $k$-core components with the same $k$ and contain all the followers. Therefore, we can efficiently find all the followers of an anchor in the smaller shell components rather than in the larger $k$-core components. Based on the shell component structure, we propose our approach AdvGreedy. Intuitively, AdvGreedy outperforms GAC-FM by following reasons: (1) Follower computation. Since each shell component is a subgraph of a $k$-core component, the search space is reduced significantly. Moreover, we propose an explore-and-retract strategy to further reduce the number of scanned vertices, which utilizes a multi-queue data structure to maintain the scan order. (2) Reuse results. The shell component is more fine-grained than the $k$-core component. Consider a vertex $u$ whose shell component remains unchanged while $k$-core component has changed, thus $u$’s follower results can be reused in AdvGreedy while needs re-computation in GAC-FM. (3) Upper bound computation. We utilize a shell component to tighten the upper bound of the follower numbers, significantly improving the pruning effect of non-candidates.

In what follows, we first introduce the shell component structure (Section 4.1), and combine it with the explore-and-retract strategy to propose the followers computation method (Section 4.2). We then detail the mechanism to reuse the intermediate results across greedy interactions (Section 4.3) and the design of the upper bound pruning method (Section 4.4). Finally, we put the above techniques together and propose our AdvGreedy algorithm (Section 4.5).

### 4.1 Shell Component Structure

**Definition 6 ($k$-shell).** Given a graph $G$ and a positive integer $k$, the $k$-shell, denoted by $H_k(G)$, is the set of vertices in $G$ with their corenesses exactly equal to $k$, i.e., $H_k(G) = V(C_k(G)) \setminus V(C_{k+1}(G))$.

**Definition 7 (shell component).** Given a graph $G$ and the $k$-shell $H_k(G)$, a subgraph $S$ is a shell component of $H_k(G)$, if $S$ is a maximal connected component of the induced subgraph $G[H_k(G)]$.

Different from Definition 9, where vertices in the same $k$-core component can be connected through other vertices whose coreness is larger than $k$, vertices in the same shell component must be connected through vertices that share the same coreness $k$. A $k$-shell is formed by the vertices in a series of non-overlapping shell components, and each vertex is contained in exactly one shell component. Note that in core decomposition, the deletion sequence of the shell components of $H_k(G)$ can be arbitrary. For a shell component $S$ of $H_k(G)$, we denote $S.V, S.E$ and $S.c$ as the vertex set, edge set and coreness of any vertex in $S$ respectively, i.e., $S.V = V(S), S.E = E(S)$ and $S.c = k$. We use structure $SC$ to index the shell components for all the vertices. For each $v \in V(G), SC[v]$ is the only shell component that $v \in SC[v].V$.

Vertices in each shell component can be further divided into different vertex sets, named layers, according to their deletion sequence in core decomposition [2]. We use $l(u)$ to denote the layer number of vertex $u$, and use $H^l$ to denote the $l$-layer vertex set in the $k$-shell $H_k(G)$, i.e., the set of vertices that are deleted in the $l$-th batch of $H_k(G)$ in core decomposition. Formally, $H^l = \{u \mid d(u, G_l) < k + 1 \land u \in H_k(G)\}$, where $G_1 = C_k(G)$, and for
must be caused by the increased number of \( v \)'s supporters. Denoted by \( u, v \)'s supporters can be divided into three sets: (1) \( u \in A \), the anchors can always support its neighbors; (2) \( c^A(u) > c^A(v) \), since the coreness of \( v \) increases at most 1, \( u \) is still a supporter of \( v \) after anchoring \( x \); (3) \( c^A(u) = c^A(v) \), \( u \) will remain as a supporter of \( v \) if \( c(u) \) also increases after anchoring \( x \).

Since case (1) is easy to identify, we focus on the latter two. For a vertex \( v \), if its coreness increases after anchoring \( x \), it is likely that new supporters of \( v \) come from its neighbors with the same coreness before anchoring \( x \). In this case, to determine whether a vertex \( v \) is a follower of \( x \), we only need to focus on \( v \)'s neighbors who are in \( SC[v] \) before anchoring \( x \). As a result, we can compute \( x \)'s followers on each of its candidate shell components \( s \in CS(x) \) independently.

**Explore-and-Retract Strategy.** To compute followers of anchor \( x \), we employ the explore-and-retract strategy to check if the corenesses of vertices who lie on any upstair path from \( x \) will increase. Specifically, we continue to explore the higher-layer neighbors (due to Lemma 2) of the vertex which we suppose its coreness will probably increase, and immediately retract when meeting a vertex whose coreness cannot increase, to check whether this "impossible" vertex will cause its lower-layer neighbors to also become "impossible".

As shown in Algorithm 2, we first find candidate shell components \( CS(x) \) based on Theorem 4 (Lines 1-2). For each \( s \in CS(x) \), if \( x \)'s followers in \( S \) remain the same as the last iteration, we reuse the results (Line 4, detailed in Section 4.3). Otherwise, we find its
followers in each component $S$ independently (Lines 5-25), which are maintained in $F[x][S]$ (Line 5). To apply the explore-and-retract strategy, we scan vertices in ascending order of their coreness-layer pairs, and decide whether the coreness of a vertex will increase by checking its supporter number. To organize candidate followers in linear time, we construct multiple queues for different layers. More specifically, for each $S$, we use a sequence of queues $\{Q_1, Q_2, \ldots, Q_{\text{max~layer}}\}$ to maintain the traverse order, where $\text{max~layer}$ denotes the maximum layer number in $S$ (Lines 6-7). We first push $x$’s successive neighbors in $S$ into the queues (Line 8), then traverse each element $v$ in $Q_1$ in ascending order of $i$ (Lines 9-11).

For each vertex $v$, we denote its supporter number as $\text{sup}(v)$ and divide its neighbors $u \in SC[v]$ into three categories to compute $\text{sup}(v)$: (i) unexplored and $I(u) \nleq I(v)$: We first assume that $u$ is a supporter of $v$. If $\text{sup}(u) \geq c(u) + 1$, we regard $v$ as a potential coreness-increased vertex and will explore $u$ later. If we later find that $u$’s coreness cannot increase (i.e., $u$ is actually not a supporter of $v$), we perform retract to check if $v$’s coreness will increase. (ii) unexplored and $I(u) > I(v)$: In this case, $u$ cannot be a supporter of $v$. As we scan vertices in layer order, $u$ will never be explored. (iii) explored: We have temporarily decided whether the coreness of $u$ will increase. If so, $u$ can be a supporter of $v$. Thus $\text{sup}(v)$ includes (i) $\text{sup}(v)$ contains (i) $\text{Q}_1 \cup SN(u)$, (ii) $F[x][S]$ and $x$ (Lines 12-13). We then check if $v$’s coreness can increase (Line 14). If so, we temporarily assume $v$ is the follower of $x$, put it into $F[x][S]$ (Line 15) and continue to explore its higher-layer neighbors (Line 16). Otherwise, we ensure that $v$’s coreness will not increase, then recursively retract to check if other vertices will remain in $F[x][S]$. Vertex that does not satisfy the coreness-increasing requirement will be removed from $F[x][S]$ (Line 21). As the removed vertex may have been regarded as a supporter of its neighbors before, we recursively retract to check its neighbors’ supporter numbers (Lines 22-24). Therefore, the final remaining vertices in $F[x][S]$ are the true followers of $x$ (Line 25).

The time complexity of Algorithm 2 is $O(m)$, because each edge is accessed at most two times: explore/retract when meeting/failing the coreness-increasing requirement. In practice, the number of scanned vertices in Algorithm 2 is much smaller, as the explore-and-retract strategy will make local decomposition early terminate.

**Example 1.** We give an example of computing the follower set of $x = v_1$ in Figure 3 by Algorithm 2. We first push $x$’s neighbors $v_2, v_6$ into $Q_2$. As $v_1 \in \{x\}$, $v_5 \in SN(v_2)$ and $v_6 \in Q_2$, we have $\text{sup}(v_2) = 3 \geq c(v_2) + 1$. Thus we push $v_2$ into $F[x][S]$ and push $v_5$ into $Q_3$. Since $I(v_5) < I(v_6)$, we have $\text{sup}(v_6) = 2 < c(v_6) + 1$, which triggers the retract strategy. It makes $\text{sup}(v_6)$ decreases and returns to check the supporter number of $v_2$. We find that $\text{sup}(v_2) = 2 < c(v_2) + 1$, which means $v_2$ is actually not a follower of $x$ and is removed from $F[x][S]$. Then we enumerate the elements in $Q_3$. As $\text{sup}(v_5) = 2 < c(v_5) + 1$ and there is no more element in the queues, the Algorithm 2 terminates and we know $F[x][S] = \emptyset$.

### 4.3 Reuse Follower Computation Results

The greedy algorithm contains $b$ iterations, and we apply the reuse technique in order to avoid redundant computations. For each vertex $v \in V(G) \setminus (A \cup \{x\})$ and each shell component $S$, we decide whether the computed follower $F[v][S]$ will remain the same after anchoring $x$, thus can be reused in the next selection iteration.

```plaintext
Algorithm 3: Reuse(x, G, A, SC)

Input : $x$ : the anchor, $G$ : the graph, $A$ : the anchor set, $SC$ : the shell components of $G$
Output : $\text{Reuse}(v)$ for each non-anchor vertex $v$, where $F[v][S]$ can be reused for each $v \in \text{Reuse}(v)$

1 for each $v \in V(G) \setminus A$ do
2 \text{Reuse}(v) $\leftarrow$ $\text{CS}(v)$;
3 Remove $v$ from $\text{Reuse}(v)$ if $F[v][S]$ is not computed;
4 $V^* \leftarrow \bigcup_{S \in SC(v)} S.\cup V$;
5 Compute $c'()$ through core decomposition [2];
6 $SC' \leftarrow \text{ShellDecomp}(G)$;
7 $S^* \leftarrow \bigcup_{S \in VS.\cup V^*} SC'[v]$; $V^* \leftarrow \bigcup_{S \in S.\cup V^*} S.\cup V$;
8 Remove $S \in SC$ from all $\text{Reuse}(v)$ if $V^* \setminus S.\cup V \neq \emptyset$;
9 return $\text{Reuse}(v)$ for each $v \in V(G) \setminus A$;
```

Algorithm 3 finds all the candidate anchors and shell components in which the follower number can be reused. For each $v \in V(G) \setminus A$, $\text{Reuse}(v)$, initialized as $\text{CS}(v)$, contains all $v$’s candidate followers (Lines 1-2, Theorem 4). For each $S \in \text{Reuse}(v)$, $F[v][S]$ must have been computed before (Line 3). Let $V^*$ denote the vertex set of all shell components in $\text{CS}(v)$ (Line 4), we update the coreness after anchoring $x$ and construct new shell components (Lines 5-6). Let $V^*$ denote the vertex set of all new shell components containing some vertex in $V'$ (Line 7). Original shell components which contain some vertex in $V^*$ can not be reused, hence are removed from $\text{Reuse}(\cdot)$ (Line 8). Algorithm 3 runs in $O(m)$ time complexity as we will scan each edge at most once to get $\text{Reuse}(\cdot)$ initially, and core decomposition and Algorithm 1 both needs $O(m)$ time.

**Theorem 5. (Correctness).** After anchoring $x$, for every non-anchor vertex $v$, we have $F[v][S]$ remaining the same if $S \in \text{Reuse}(v)$.

#### 4.4 A Tighter Upper Bound

We first review the upper bound pruning used in GAC. Based on Lemma 2, Linghu et al. propose the upper bound of follower number of any non-anchor vertex $x$, i.e., $UB_{G'}(x) = 1 + \sum_{u \in SN(x)} UB_{G}(u)$, where $UB_{G}(x) = 1 + \sum_{u \in SN(x) \cap \{v \mid c(v) < c(x)} UB_{G}(u)$. However, the following example shows this bound has much overlap.

**Example 2.** Consider computing the upper bound of $v_5$ in the graph of Figure 3. We have $UB_{G}(v_5) = UB_{G}(v_9) = 1$ and then $UB_{G}(v_5) = UB_{G}(v_9) = 1 + UB_{G}(v_4) + UB_{G}(v_9) = 3$. Thus $UB_{G}(v_5) = 1 + UB_{G}(v_9) + UB_{G}(v_9) = 7$, which double counts $v_4$ and $v_9$.

Worse still, we experimentally find that a large ratio of the upper bounds computed in this way exceeds $n$ (shown in Table 3). To refine the technique, according to Lemma 2, we first consider the size of the upstair DAG as the direct upper bound, i.e., the number of vertices that can be reached from $x$ through any upstair path. However, there exists no linear algorithm which can compute the exact size of the reachable DAG for each vertex $[9]$. We thus refine the upper bound based on the shell components. Specifically, for an anchor candidate $x$, we first get the upper bound of its followers of each shell component $S$, making it no more than the number of vertices with larger layers than $x$ in $S$, i.e., $UB(x, S) = \min \{|S.\cup V \cap \{x\}|, \sum_{u \in SN(x) \cap S.\cup V} UB(u, S)\}$, where $U(x) = \{v \mid P(x) < P(v)\}$. If $x$’s coreness has never changed
before, we set \( UB(x, SC[x]) = UB(x, SC[x]) + 1 \). Then the upper bound of \( x \)'s followers is \( UB(x) = \sum_{S \in SC(x)} UB(x, S) \). Furthermore, applying the reuse technique, if the follower result \( F[x][S] \) can be reused in current iteration, we can use it directly as it is exactly the number of \( x \)'s followers, i.e., the tightest upper bound.

**Theorem 6.** Given a graph \( G \), a current anchor set \( A \) and a vertex \( x \in V(G) \setminus A \), we have \( g(A \cup \{x\}, G) - g(A) \leq UB(x) \).

As we can compute the upper bounds of all the candidate anchors in a reverse order of topological sorting of their coreness-layer pairs, the time complexity of the upper bound computation is \( O(m) \).

### 4.5 An Advanced Greedy Approach

Algorithm 4 shows the details of our final \( \text{AdvGreedy} \) algorithm which combines all the techniques proposed in the last 4 Sections. We first compute the coreness of each vertex and construct the shell which can be reused in the next iteration for each vertex (Line 14). By applying the reuse technique, if the follower result \( F[x][S] \) can be reused in the current iteration, we can use it directly as it is exactly the number of \( x \)'s followers, i.e., the tightest upper bound.

**Algorithm 4.** \( \text{AdvGreedy}(G, b) \)

**Input**: \( G \): the graph, \( b \): number of anchors  
**Output**: \( A \): the set of anchored vertices

1. Compute \( c[\cdot] \) through core decomposition [2];  
2. \( SC \leftarrow \text{ShellDecomp}(G) \);  
3. \( g \leftarrow \emptyset; A \leftarrow \emptyset \);  
4. for \( i \leftarrow 1 \) to \( b \) do  
   5. \( x \leftarrow \text{null}; \Delta \leftarrow 0 \);  
   6. Compute upper bounds \( UB[u] \) for each \( u \in V(G) \setminus A \);  
   7. for each \( u \in V(G) \setminus A \) with decreasing order \( UB(u) \) do  
      8. if \( UB(u) > \Delta \) then  
         9. \( F \leftarrow \text{FindFollowers}(u, G, SC) \);  
         10. if \( |F| > \Delta \) then  
              11. \( \Delta \leftarrow |F| \); \( x \leftarrow u \);  
         else Break;  
      12. \( A \leftarrow A \cup \{x\}; d(x) \leftarrow +\infty; g \leftarrow g \cup F \);  
   13. \( \text{Reuse}(x, G, A, SC) \);  
15. return \( A \);

Specifically, we design an exact algorithm paradigm for the budget minimization problem and then consider returning to solve the FM problem. The exact algorithm paradigm needs to explore all possible \( 2^b \) solutions, which are encoded by a solution tree \( T \), i.e., \( T \) is a perfect binary tree with \( 2^b \) leaves. Every node in \( T \) has two children. Its left child means adding a new vertex \( x \) into the anchor set \( A \), while its right child means \( x \) will not be considered as an anchor. We use \( T(A, A_{-}) \) to denote each tree node, where \( A \) is the anchor set of the current node, and \( A_{-} \) is the set of disregarded vertices up to now. For each tree node, the "to be decided vertex" \( x \) is chosen by the greedy approach. Specifically, for a tree node \( T(A, A_{-}) \), the next vertex we choose to add into \( A \) or \( A_{-} \) is \( x = \arg \max_{u \in V(G) \setminus (A \cup A_{-})} g(A \cup \{x\}) - g(A) \), i.e., the left child node is \( T(A \cup \{x\}, A_{-}) \) and the right child node is \( T(A, A_{-} \cup \{x\}) \). We apply a DFS to search for solutions in \( T \), thus the first solution we can find is the result from the greedy method, which satisfies output a good solution quickly. Then we will explore more vertices according to their follower numbers, which means the vertices that can lead to larger resilience gain will be explored first, this follows search for better solutions as quickly as possible.

**Reuse Intermediate Results.** As DFS has two main actions, continuing to search the child nodes and backtrack to the father nodes, we design a linear space implementation for reusing the intermediate results. Specifically, in the subtree rooted at \( T(A, A_{-}) \), we greedily add vertices into \( A \) in the child nodes, store the follower upper bound and the reusable shell components for each vertex \( v \) in \( UB(A)[v] \) and \( SC(A)[v] \). Then we push the follower results of vertices into \( H[A]\). Now, \( H[\cdot] \) is a max heap and is ordered by the follower number of each vertex. Thus we only need to compute \( UB(A)[\cdot] \) and \( SC(A)[\cdot] \) once in the subtree, and continue computing the followers of each vertex based on \( H[A] \).

**Return to FM Problem.** The above search process deals with the budget minimization problem, we then introduce how to use the search results to further improve the solution of the FM problem. As the first result returned is the same as the greedy approach, we use the given budget to get the first result and use it as the target.
resilience gain. Then we continue to apply the paradigm to search for smaller budgets, and once we get a smaller budget, we naturally have more extra budgets to improve the resilience gain, i.e., we apply the greedy method to select \((b - b_{\min})\) more anchors.

A detailed description of the paradigm can be found in Appendix. **Pruning Strategies.** To speed up the search of \(T\), we apply some effective pruning strategies. Recall that we turn to the budget minimization problem to further solve the FM problem. Let \(b_{\min}\) denote the current best solution, and \(g_t\) denote the target resilience gain, we apply the following strategies to accelerate the search process:

1. If \(g(A) \geq g_t\), the subtree can be pruned, for other solutions in it can not have smaller budgets.
2. If \(g(A) < g_t\) and \(|A| \geq b_{\min} - 1\), the subtree can be pruned, because the best possible solution in its subtree is \(b_{\min}\).
3. If \(g(V(G) \setminus A_v) < g_t\), the subtree can be pruned, since no solutions in the subtree can reach the target gain due to the monotonicity of \(g(\cdot)\) (proof of monotonicity is in Appendix).

**Bounded-death Heuristic.** As the solution space is still large even with the above pruning techniques, to limit the search to a relatively better region in the solution space, we apply the bounded-death heuristic [43] in our framework. Specifically, we further prune the subtree rooted at tree node \(T(A, A_v)\) if \(|A_v| > \lambda\), where \(\lambda \geq 0\) is a given constant integer. In our paradigm, if \(\lambda = 0\), the result is exactly what the greedy approach AdvGreedy finds.

**Example 3.** Consider the graph in Figure 3, we construct its corresponding search tree in Figure 4 with budget \(b = 2\) and \(\lambda = 1\). In each tree node, we mark the current \(A\) and \(A_v\), and for each edge, we use \(+v_1\) and \(-v_1\) to denote adding \(v_1\) into \(A\) or \(A_v\). For the root node, \(A\) and \(A_v\) are both originally \(\emptyset\) and the greedy approach selects \(v_7\) as the first anchor. For the left child of the root, it adds \(v_7\) into \(A\), and greedily selects the next anchor as \(v_1\). For its left child, we further add \(v_1\) into \(A\) and get the first solution \(A = \{v_1, v_7\}\). Then we find that we can prune the subtree rooted at the right child of node \((\{v_7\}, \emptyset)\), because \(g(A) = 5 < g_t\) and \(|A| = 1 \geq b_{\min} - 1\) (Pruning 2). For the right child of the root, it adds \(v_7\) into \(A_v\), thus due to \(\lambda = 1\), the subtree rooted in its right child will be pruned. For the node \((\{v_2\}, \{v_7\})\), its left child’s subtree is pruned due to Pruning 2, and its right child’s subtree is pruned because of \(\lambda = 1\).

### 6 Experimental Evaluation

**Datasets.** The experiments are conducted on 8 public datasets from KONECT [22] and SNAP [23]. The statistics of datasets are included in the appendix, where the largest dataset in our experiments contains 3, 072, 441 vertices and 117, 185, 083 edges.

**Environments.** Experiments are performed on a CentOS Linux server (Release 7.5.1804) with Quad-Core Intel Xeon CPU (E5-2640 v4 @ 2.20GHz) and 128G memory. All algorithms are implemented in C++17. Source code is compiled by GCC under -O3 optimization.

#### 6.1 Compared Methods

Towards effectiveness, we compare greedy method (AdvGreedy / GAC-FM) with exact ones and other 7 heuristics. We survey heuristics proposed in related works and adapt them to solve our problem.

**Vertex Attribute.** The basic heuristics are the attributes of vertices. Degree (Deg), Deg anchors \(b\) vertices with the highest degree. Coreness (Core), Core anchors \(b\) vertices with the highest coreness.

**Bound of Resilience Gain.** We can use the estimated bounds of resilience gain as another type of heuristic to select anchors. Upper Bound (UB). UB chooses \(b\) vertices with the largest budget, i.e., \(UB(x)\) for each vertex \(x\) (details in Section 4.4).

**Upstair DAG Size (UD).** UD chooses \(b\) vertices with the largest upstair DAG size, i.e., the number of vertices that can be reached from each vertex through its upstair paths. It is the tighter version of UB, but it is time-consuming since there is no linear algorithm.

**Successive Degree (SD).** Experiments in [24] compare with GAC by the successive degree, that is, choose \(b\) anchors with the highest successive degree, i.e., \([SN(\cdot)]\). It can be regarded as a lower bound of the upper bound (1-hop of the upstair DAG).

**Score Function.** Applying scoring functions to evaluate the candidate vertices’ quality is a common method in related works. The details of the following two algorithms are included in the appendix. Shapley Value (SV), Shapley Value is a concept in cooperative game theory. Motivated by [28], we design a Shapley Value to capture the importance of a vertex inside a vertex set.

**Combinational Score (CS).** Motivated by the score function from [25], we consider the combinational effect of anchors and design a new heuristic for our problem.

#### 6.2 Experimental Results

**Exp 1: Comparison with Other Heuristics.** We compare the resilience gain of AdvGreedy with other 7 heuristics (details in Section 6.1) when the budget is 100. Note that UD and SV do not return results within three days in three larger datasets, we mark them by “OOT” in the figure. As shown in Figure 5, AdvGreedy always performs the best among all the heuristics. CS and SV perform relatively well as they both consider the income of anchor combination. They may fail on some datasets, e.g., in Stanford the performance gap between CS and AdvGreedy is huge. The efficiency of SV is much worse than AdvGreedy even when reducing the samples. Among three bound heuristics, UD performs the best as it equips with a tighter bound. For vertex attributes, the performance of Deg is better than Core, but it is still much worse than AdvGreedy. Core performs...
the worst, since vertices with larger corenesses are originally others’ supporters thus anchoring them will not provide additional support. **Exp 2: Overall Efficiency.** Figure 6a shows the total running time of GAC-FM and AdvGreedy on all datasets when \( b = 100 \). GAC-FM cannot return results on Orkut within one week, thus we mark it as “OOT”. In all 8 datasets, AdvGreedy always outperforms GAC-FM by almost 1 order of magnitude and up to 2 orders. Besides, the gap becomes larger with the scale of the datasets increasing.

**Exp 3: Varying the Budget.** Figures 6b and 6c present the running time on Gowalla and Youtube when budgets vary from 1 to 1000. As GAC-FM do not return results within 24 hours when \( b \geq 487 \), we do not report its running time in Figure 6c. In both two figures, the slope of the curve decreases as the budget increases, indicating that AdvGreedy has excellent scalability when the budget is large. Besides, AdvGreedy is always faster than GAC-FM by more than 1 and 2 orders of magnitude in Gowalla and Youtube, respectively. We can also find that the gap between them is huge when the budget is relatively small because of the refined upper bound.

**Exp 4: Performance of Time-Dependent Framework.** Figures 7a and 7b show the performance of the time-dependent search framework on Gowalla and Youtube when \( b = 100 \) respectively. GreedySearch first finds a resilience gain of 4026 on Gowalla in 188s and 4578 on Youtube in 846s, similar to AdvGreedy. The framework continues to search for a better solution with parameter \( \lambda \) varying from 1 to 3. The framework can always discover better solutions as the search time increases. The performance is the best when \( \lambda = 2 \) on both datasets, as smaller \( \lambda \) may result in excessive pruning, and bigger \( \lambda \) may be time-consuming, e.g., GreedySearch can not terminate within \( 10^6 \) seconds, when we set \( \lambda = 3 \) on Gowalla.

**7 RELATED WORKS**

To measure the ability of a network to withstand and sustain disturbances, extensive studies are conducted on network resilience or robustness. As surveyed in [13, 14, 29], there are various resilience measures based on different graph characteristics, e.g., adjacency, connectivity, and distance. The intuition of adjacency approaches is that nodes with many connections are more critical to the overall graph structure [17, 36]. Connectivity metrics measure the robustness of connecting/disconnecting the graph with key nodes [3, 7, 20]. Distance metrics consider the path length between node pairs [1, 4]. Besides, the problems based on influence propagation [16, 19, 35, 37, 38] find and utilize key nodes s.t. the influence spread can be optimized, where node activation probability should be estimated and influence spread computation is costly. However, the focuses of the above studies are different from our model, e.g., the centrality measures essentially consider the resilience of information flow [36] while our FM problem is built on the true engagement dynamics of all the vertices.

As the coreness metric is the "best practice" for measuring the vertex engagement with network topology [26], many previous works consider measuring user engagement (network stability) by monitoring k-core structure, e.g., maximizing the size of k-core via vertex anchoring [5, 8, 25, 41], minimizing the size of k-core via vertex removal [42, 44], and edge manipulations [28, 45]. As the focus on the k-core with a fixed k value is relatively a local view on user engagement, existing studies tend to consider the overall coreness dynamic of all the nodes, i.e., the overall coreness gain/loss [24, 40]. However, as shown in the introduction, the engagement enhancement is biased on certain nodes and the optimization of network resilience aims to enhance as many nodes as possible. Dey et al. [12] propose TMVC problem to maximize the number of coreness-changed vertices by deleting at most \( b \) vertices. This is to consider the protection of tender nodes which is different from the enhancement of vertex anchoring in our FM problem.

**8 CONCLUSION AND FUTURE WORK**

In this paper, we propose and study the following maximization problem, aiming to maximize coreness-increased vertices by finding an anchor set. We prove the problem is NP-hard, and NP-hard to approximate within a factor of \( O(n^{1-\epsilon}) \). The problem is also \( W[2]\)-hard parameterized by budget. Given such hardness, we develop an efficient greedy method AdvGreedy based on shell components and pruning techniques. Extensive experiments on eight real-life networks demonstrate the effectiveness of AdvGreedy, especially on massive graphs. To bridge the gap between theory and practice, a time-dependent framework is proposed, producing a solution quickly and continuing to search for better solutions if time permits. In future work, it is promising to design more powerful heuristics which can achieve similar effectiveness while more efficient, then the extended generic framework may beat the greedy approach on both sides.

**ACKNOWLEDGMENTS**

This work is partially supported by the National Natural Science Foundation of China (62002073, 62002236, 62302294, U2241211) and the Guangdong Basic and Applied Basic Research Foundation (2023A1515012603).
REFERENCES


A APPENDICES

State-of-the-Art from Existing Method. Linghu et al. [24] propose a greedy algorithm GAC for the anchored coreness problem. In a nutshell, GAC starts from an empty anchor set A = ∅, and then iteratively finds one best anchor u with the highest coreness gain to add into A in each of the b iterations, i.e., u = arg max A∈A cG(A) \ A cG(A∪{u}, G) = cG(A, G) , where cG(A, G) is the coreness gain of anchor set A in G. The computation of cG(⋅) in GAC is mainly based on the following lemma.

**Lemma 3** ([24]). If a vertex x is anchored in G, the coreness of any u ∈ V(G) \ {x} will either not decrease or increase by at most 1.

We can apply a core decomposition to compute the coreness gain for each candidate vertex, which requires O(m) time. The greedy method is conceptually simple, while it is computationally expensive. GAC speeds up its efficiency by utilizing the core component tree, widely used in related works [24, 27, 30, 31, 40], which organizes V(G) based on the k-core components in G with different k.

**Definition 9** (k-core component). Given a graph G and the k-core Ck(G), a subgraph S is a k-core component if S is a connected component of Ck(G).

Linghu et al. find that the followers of each vertex x are constrained to the core components that contain at least one neighbor of x with the same or higher coreness as x, and these components are denoted by T(C(x)). Therefore, GAC finds followers of each vertex through partial core decomposition, i.e., applies core decomposition independently in components in T(C(x)). In addition, they propose two pruning strategies to further improve the efficiency: the reuse technique and the upper bound. The reuse technique avoids redundant computation in each iteration by only computing the followers
Algorithm 5: GreedySearch(G, b, A, A−, g, t, bmin)

Input : G: the graph, b : budget, A : current anchor set, A− : current disregarded vertex set, g : target resilience gain, 
bmin : current minimal budget for the target resilience gain
1 if |A| ≥ bmin then return;
2 if |A| = b or g(A) ≥ g then return;
3 if g = ∞ then gi ← g(A);
4 bmin ← |A|;
5 A′ ← AdvGreedy(G, b− bmin) with A′ ∩ (A ∪ A−) = ∅;
6 Print the current solution A′ ∪ A;
7 return;
8 if A ≠ Af, where Af is the anchor set of the father tree node then
9 Compute UB(|A||u|) for each u ∈ V(G) \ (A ∪ A−);
10 H[|A|] ← ∅;
11 for each u ∈ V(G) \ (A ∪ A−) with decreasing order UB(|A||u|) do
12 Continue;
13 if (u, w) ∈ H[|A|] then
14 F ← FindFollowers(u, G, SC);
15 Push (F, u) into H[|A|];
16 else Break;
18 (−, x) ← H[|A|].top(); H[|A|].pop(); (d(x) ← −∞;
19 ReuseSC[|A| + 1] = ReuseG, A, SC);
20 GreedySearch(G, b, A ∪ \{x\}, A−, g, t, bmin);
21 d(x) ← |N(x)|;
22 GreedySearch(G, b, A, A− ∪ \{x\}, g, t, bmin);

Shapley Value is a concept in cooperative game theory. Motivated by [28], we design a Shapley Value to capture the importance of a vertex inside a vertex set. Given a vertex v and a subset A ⊆ V(G) \ {v}, the marginal contribution of v to A is g(A ∪ {v}) − g(A, G). Let P be the set of all |V(G)|! permutations of all the vertices in V(G) and P(π, v) be the set of vertices that appear before v in a permutation π. The Shapley Value of v is the average of its marginal contribution to the vertex set that appears before v in the permutations, i.e., SV(v) = 1 |P| ∑π∈P g(P(π, v)) U (v). Since computing the exact Shapley Value requires O(|V(G)|!) time, we estimate the value via sampling. Combinational Score (CS). Motivated by the score function from [25], we consider the combinatorial effect of anchors and design a new heuristic for our problem. For each vertex v in G with an anchor set A, V (v) = cA(v) + 1 − |{u ∈ N(v) ∩ cA(v) > cA(\{v\})}| measures the extra supporters needed to increase v’s coreness by 1. Although anchoring v may not increase the coreness of vertex u, it may provide more support for u, i.e., V (u) − V (A∪\{v\}) (u) > 0. Hence, we define CS considering whether the coreness of v is increased or not separately, i.e., CS(v) = scoreup(v) + scoreup(v), where scoreup(v) = g(A ∪ \{v\}, G) − g(A, G), scoreup(v) = \sum\{w∈V(G)\} cA(w) u∈N(v) cA(w) u∈N(v) (u) cA(w) u∈N(v) (u) and we can find that for a vertex u, if V (u) − V (A∪\{v\}) (u) changes after anchoring v, u must be a neighbor of v or v’s followers. Therefore, we can use our AdvGreedy to compute the followers of each anchor vertex and compute the value of CS.

Additional Experiment Results.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>dmax</th>
<th>davg</th>
<th>kmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv</td>
<td>34,546</td>
<td>421,578</td>
<td>846</td>
<td>24.4</td>
<td>30</td>
</tr>
<tr>
<td>Gowalla</td>
<td>196,591</td>
<td>456,830</td>
<td>14,730</td>
<td>9.2</td>
<td>51</td>
</tr>
<tr>
<td>NotreDame</td>
<td>325,729</td>
<td>1,090,108</td>
<td>10,721</td>
<td>6.5</td>
<td>155</td>
</tr>
<tr>
<td>Stanford</td>
<td>281,903</td>
<td>1,992,636</td>
<td>38,625</td>
<td>16.4</td>
<td>71</td>
</tr>
<tr>
<td>Youtube</td>
<td>1,134,890</td>
<td>2,987,624</td>
<td>28,754</td>
<td>5.3</td>
<td>51</td>
</tr>
<tr>
<td>Wiki</td>
<td>557,677</td>
<td>19,177,218</td>
<td>93,188</td>
<td>51.6</td>
<td>814</td>
</tr>
<tr>
<td>Livejournal</td>
<td>5,997,962</td>
<td>34,681,189</td>
<td>14,815</td>
<td>17.4</td>
<td>360</td>
</tr>
<tr>
<td>Orkut</td>
<td>5,072,441</td>
<td>117,185,083</td>
<td>33,313</td>
<td>76.3</td>
<td>253</td>
</tr>
</tbody>
</table>

Statistics of Datasets. Table 1 shows the statistics of the datasets, ordered by the number of edges, where dmax is the maximum vertex degree, davg is the average vertex degree and kmax is the maximum coreness of vertices in the graph.
Given an arbitrary instance, we compare the upper bounds and report the average resilience gain in Table 2. The resilience gain of AdvGreedy is at least 82% of exact algorithm, and we find that the resilience gain ratio of AdvGreedy over the exact algorithm may increase with a larger budget $b$. The running time of exact algorithm is also reported in the table, while weomit that of AdvGreedy since it takes less than 1ms on all budgets. We can find that AdvGreedy is faster than the exact algorithm by up to 7 orders of magnitude.

Exp 7: Performance on Budget Minimization Problem. The heuristics comparisons for the budget minimization problem can also be shown in Figures 5 and 8 by swapping the x and y axis, we can find that the greedy approach obviously performs best.

Figure 9 presents the results of time-dependent framework on budget minimization problem of FM with $\lambda$ varying from 1 to 3 on Gowalla and Youtube respectively. We set the target resilience gains as the results AdvGreedy when $b = 100$ on both datasets. The results show that the budget continues to decrease with the running time increasing, and the minimized budgets can decrease from 100 to 90 on Gowalla and to 98 on Youtube within $10^8$ seconds.

Exp 8: Core component tree v.s. Shell component. We compare the size and number of the basic units of core component tree and shell component on Gowalla and Youtube, shown in Figure 10. For vertices share the same coreness, shell component can divide them into more and smaller units compared with core component tree, especially when coreness is less than 40. As Table 3 shows, the largest component size and average component size of core component tree are both much worse than shell components.

Exp 6: Comparison with Exact Solution. We compare AdvGreedy with exact algorithm which identifies the optimal $b$ anchors by enumerating all possible combinations. Due to the enormous time cost, we extract small datasets by iteratively extracting a vertex and all its neighbors, until the number of extracted vertices reaches 100. For both Gowalla and Youtube, we extract 5 subgraphs and report the average resilience gain in Table 2. The resilience gain of AdvGreedy is at least 82% of exact algorithm, and we find that the resilience gain ratio of AdvGreedy over the exact algorithm may increase with a larger budget $b$. The running time of exact algorithm is also reported in the table, while we omit that of AdvGreedy since it takes less than 1ms on all budgets. We can find that AdvGreedy is faster than the exact algorithm by up to 7 orders of magnitude.

Exp 7: Performance on Budget Minimization Problem. The heuristics comparisons for the budget minimization problem can also be shown in Figures 5 and 8 by swapping the x and y axis, we can find that the greedy approach obviously performs best.

Figure 9 presents the results of time-dependent framework on budget minimization problem of FM with $\lambda$ varying from 1 to 3 on Gowalla and Youtube respectively. We set the target resilience gains as the results AdvGreedy when $b = 100$ on both datasets. The results show that the budget continues to decrease with the running time increasing, and the minimized budgets can decrease from 100 to 90 on Gowalla and to 98 on Youtube within $10^8$ seconds.

Exp 8: Core component tree v.s. Shell component. We compare the size and number of the basic units of core component tree and shell component on Gowalla and Youtube, shown in Figure 10. For vertices share the same coreness, shell component can divide them into more and smaller units compared with core component tree, especially when coreness is less than 40. As Table 3 shows, the largest component size and average component size of core component tree are both much worse than shell components.

Exp 6: Comparison with Exact Solution. We compare AdvGreedy with exact algorithm which identifies the optimal $b$ anchors by enumerating all possible combinations. Due to the enormous time cost, we extract small datasets by iteratively extracting a vertex and all its neighbors, until the number of extracted vertices reaches 100. For both Gowalla and Youtube, we extract 5 subgraphs and report the average resilience gain in Table 2. The resilience gain of AdvGreedy is at least 82% of exact algorithm, and we find that the resilience gain ratio of AdvGreedy over the exact algorithm may increase with a larger budget $b$. The running time of exact algorithm is also reported in the table, while we omit that of AdvGreedy since it takes less than 1ms on all budgets. We can find that AdvGreedy is faster than the exact algorithm by up to 7 orders of magnitude.
According to Lemma 1, for each instance \( S \! : \! \{ \cdot \} \):

- \( \) each \( w_i \) corresponds to elements \( u_i \) in \( U \) in the SCD instance. Each position of matrix \( M \) contains a \( d \)-clique initially. For each clique in \( M \), we arbitrarily select three vertices \( x_{i,j}, y_{i,j} \) and \( z_{i,j} \), and then modify \( M \) as follows: (i) remove edges \( (x_{i,j}, y_{i,j}) \) and \( (x_{i,j}, z_{i,j}) \) from each \( d \)-clique; (ii) for each \( i \in [1, p] \) and \( j \in [1, N] \), add edges \( (x_{i,j}, x_{i,j+1}) \); (iii) for each \( j \in [1, N] \), add edges \( (y_{i,j}, z_{i,j+1}) \) for each \( i \in [1, p] \), where \( f(i, j) = (i + (j - 1) \mod (p - 1)) \mod p + 1 \) (making the connection between rows cycle by cycle \( (p - 1) \)-th); (iv) add edges from \( w_k \) to \( x_{i,1} \) if \( u_i \in S_k \); (v) add edges from each \( x_{i,N} \) to \( u_l \) for each \( i \in [1, p] \).

We can prove that the coreness of each \( w_i \) is \( |S_i| \), and the coreness of each vertex \( v \) in \( M \) is \( d - 2 \). We then show that \( G \) has the following two properties corresponding to the instance of the SCD problem:

(i) If the instance \((U, S, r)\) is a yes-instance, then there exists an \( r \)-size anchor set \( A \) such that \( g(A, G) = r + Npd \). Consider anchoring all the \( b \) vertices on \( w_i, w_{i+1}, \ldots, w_b \), which correspond to the solution of SCD problem, then the coreness of every vertex in matrix \( M \) will increase from \((d - 2)\) to \((d - 1)\). Let \( N > (d + 1 + q)/(pd) \), we have \( g(A, G) = r + Npd > Npd + 2/2 + (d + 1 + q)/2 = (Npd + d + 1 + q)/2 = n/2 \). Therefore, \( g(A, G) = \Omega(n) \).

(ii) If \((U, S, r)\) is a no-instance, then there exists at least an \( i \)-th row in \( M \), in which the corenesses of all the vertices will not increase. Therefore, these vertices will be removed in core decomposition when \( k = d - 1 \). Note that \( N \) is a multiple of \((p - 1)\), for each row in \( M \) we denote positions \((i - 1) \cdot (p - 1) + i \cdot (p - 1) \) by \( patch_i \). Then for each \( j \)-th row where \( j \neq i \), there exists at least one vertex in each \( patch_i \) which is adjacently accessible to a vertex in the \( i \)-th row, i.e. if no anchor is placed in each \( patch_i \), this patch will also be removed when \( k = d - 1 \) via the core decomposition. Thus \( r \) anchors can obtain at most \( r(p - 1)d + r ) \) resilience gain. Since \( d > 2 \), we have \( r(p - 1)d + r < rpd \). As \( r \) is corresponding with budget \( b \) in the FM instance, i.e., \( r, b \), we can ensure that \( g(A, G) = O(b) \).

Proof of Corollary 1. According to Lemma 1, for each instance \((U, S, r)\) of SCD problem, it is a yes-instance iff there is a \( r \)-size anchor set \( S \) such that the resilience gain is \( r + Npd \) in the corresponding FM instance. If there is a PTIME solution for the FM problem, then we can determine in PTIME whether the optimal resilience gain exceeds \( r + Npd \), and subsequently solve SCD problem in PTIME. \( Q.E.D. \)

Proof of Theorem 3. We prove this theorem by an FPT-reduction from the W\[2\]-hard SCD problem parameterized by the size of set cover \([6]\). Consider an arbitrary instance \((U, S, r)\) of the SCD problem, we construct a corresponding instance of the FM problem on a graph \( G \). For each \( S_i \in S \), we create a vertex \( w_i \) in \( G \). For each \( u_i \in U \), we create a vertex \( m_i \) with \( p \) cliques connected to it, where each clique is a \((p + 2)\)-clique. Finally, we add edges between \( m_i \) and \( w_j \) if \( u_i \notin S_j \). Note that the budget \( b \) in the FM instance corresponds to the size \( r \) of the set cover in the SCD instance.

We next prove that the SCD instance \((U, S, r)\) is a yes-instance iff there exists an anchor set \( A \subseteq V(G) \) with \(|A| \leq b \) that the corresponding resilience gain \( g(A, G) \geq b + p \).

In one direction, we assume that the SCD instance is a yes-instance. In graph \( G \), we know that the coreness of every \( w_i \) is \(|S_i| \), and the coreness of each \( m_j \) is \( p \). According to the solution \( \{S_1, \ldots, S_L\} \) of the SCD instance, we can anchor the corresponding vertices \( w_{i_1}, \ldots, w_{i_L} \) in \( G \), thus the resilience gain is \( b + p \). Hence we can conclude that the resilience gain is at least \( b + p \).

For the other direction, we prove by contradiction, i.e., assume that the SCD instance is a no-instance. Given that in \( G \), anchoring each \( m_i \) or vertex in cliques can obtain only 1 resilience gain, we consider placing anchors in \( w_i \), which can get extra gain in \( m_i \) if edge \((w_i, m_j) \) exists. As there exists no set cover of size \( r \), we can obtain at most \( b + p - 1 \) resilience gain after anchoring \( A \) with \(|A| < b \), where we place \( b \) anchors on \( w_{i_1}, w_{i_2}, \ldots, w_{i_b} \). Hence there exists a contradiction to that there exists an anchor set \( A \) with \(|A| \leq b \) that \( g(A, G) \geq b + p \).

Proof of Theorem 4. From Lemma 2 and the definition of upstair path, if a vertex \( u \) is a follower of anchor \( x \), it must be in set \( SN(x) \) or connected to a vertex \( v \in SN(x) \) through a path where each vertex has the same coreness \( c(v) \). By the definition of shell component, \( u \) and \( v \) are in the same shell component, thus shell components in CS(x) contain all the followers of vertex \( x \).

Proof of Theorem 5. Recall the analysis in Section 4.2, \( F[x] \subseteq S \) will not change after anchoring \( x \) if the supporters of vertices in \( S \) do not change. As \( S \subseteq \text{ReuseSC}(x) \) does not contain any vertex in \( V \prime \prime \), \( S \) will remain the same and the anchor \( x \) is not a supporter of any vertex in \( S \). Besides, we consider the supporters of vertices in \( S \) by considering the shell components \( S' \) with at least one edge between \( S' \) and \( S \); (i) \( S' \cdot c > S \cdot c \), as \( S' \cdot c \) will not decrease, the vertices in \( S' \) who are supporters of vertices in \( S \) before anchoring \( x \) will still be supporters of them after anchoring \( x \); (ii) \( S' \cdot c < S \cdot c \), since \( S \) does not change after anchoring \( x \), \( S' \cdot c \) is at most \( S \cdot c - 1 \), thus the vertices in \( S' \) are still not the supporters of vertices in \( S \); (iii) no \( S' \) with \( S' \cdot c = S \cdot c \), otherwise \( S' \) and \( S \) are the same component.

Proof of Theorem 6. By Lemma 2, we know that a follower of vertex \( x \) must be included in its upstair DAG. In a shell component \( S \), every vertex in \( x \)’s upstair DAG is counted at least once in \( U(B(x, S)) = \sum_{i \in S \cup U(x)} |V(S_i) \cup U(B(u, S))| \). Therefore we have \( |F[x]| \subseteq U(B(x, S)) \), thus \( g(A \cup \{x\}, G) = g(A) = \sum_{i \in S \cup \{x\}} |F[x]| |S_i| \leq \sum_{i \in S \cup \{x\}} U(B(x, S)) = U(B(x)). \)

Proof of monotonicity of \( g(\cdot) \). Suppose there exists an anchor vertex set \( A \) and a vertex \( x \notin A \), anchoring new vertex \( x \) cannot decrease other vertices’ corenesses. Thus we have \( g(A) \leq g(A \cup \{x\}) \), which means the function \( g(\cdot) \) is monotonic.

![Figure 11: Construction example of Lemma 1](attachment:image.png)