Preventing Unraveling in Social Networks: The Anchored k-Core Problem

Kshipra Bhawalkar[1], Jon Kleinberg[2], Kevin Lewi[1], Tim Roughgarden[1], and Aneesh Sharma[3]

[1] Stanford University, Stanford, CA, USA
[2] Cornell University, Ithaca, NY, USA
[3] Twitter, Inc.

Abstract. We consider a model of user engagement in social networks, where each player incurs a cost to remain engaged but derives a benefit proportional to the number of engaged neighbors. The natural equilibrium of this model corresponds to the k-*core* of the social network — the maximal induced subgraph with minimum degree at least k.

We introduce the problem of "anchoring" a small number of vertices to maximize the size of the corresponding anchored k-core — the maximal induced subgraph in which every non-anchored vertex has degree at least k. This problem corresponds to preventing "unraveling" — a cascade of iterated withdrawals — and it identifies the individuals whose participation is most crucial to the overall health of a social network.

We classify the computational complexity of this problem as a function of k and of the graph structure. We provide polynomial-time algorithms for general graphs with $k = 2$, and for bounded-treewidth graphs with arbitrary k. We prove strong inapproximability results for general graphs and $k \geq 3$.

1 Introduction

A defining property of social networks — where nodes represent individuals, and edges represent friendships — is that the behavior of an individual is influenced by that of his or her friends. In particular, they often exhibit positive "network effects", where the utility of an individual is increasing in the number of friends that behave in a certain way. For example, empirical work by Burke et al. [7] found support for "social learning" — that individuals are more likely to contribute useful content to a social network if their friends do. Increasingly, empirical studies suggest that the influence of interactions in social network extends to behavior outside of these networks, as well (see Ellison et al. [12], for example). These phenomena motivate the following question:

> *How should a social network be designed or modified to maximize the participation and engagement of its users?*

Indeed, this general question was recently studied by Farzan et al .[13] from a system-building perspective.

For concreteness, consider scenarios where each individual of a social network has two strategies, to "engage" or to "drop out". Being engaged could mean contributing to a public good (like network content), signing up for a new social network feature, adopting one technology instead of another, and so on. We assume that a player is more likely to be engaged if many friends are. For this Introduction, we focus on our most basic model. We first describe our model via a process of cascading withdrawals, and then formulate it using a simultaneous-move game.

Assume that all individuals are initially engaged, and for a parameter k, a node remains engaged if and only if at least k friends are engaged. For example, engagement could represent active participation in the social network, which is worthwhile to an individual if and only if at least k friends are also actively participating. Or, dropping out could represent the abandonment of an incumbent product in favor of a newly

arrived competitor; when the number of one's friends using the old product falls below k, one switches to the new product.

In this basic model, it is clear that all individuals with less than k friends will drop out. These initial withdrawals can be contagious, spreading to individuals with many more than k friends. See Figure 1 for an example of this phenomenon. In general, after such iterated withdrawals die out, the remaining engaged individuals correspond to a well-known concept in graph theory — the *k-core* of the original social network, which by definition is the (unique) maximal induced subgraph with minimum degree at least k. Alternatively, the k-core is the (unique) result of iteratively deleting nodes that have degree less than k, in any order.

Schelling [23, P.214] describes this type of "unraveling" in typically picturesque language, by contrasting the cycle with the line (with $k = 2$). He imagines people sitting with reading lamps, each of whom can get additional partial illumination from the lamps of their neighbor(s):

> *In some cases the arrangement matters. If everybody needs 100 watts to read by and a neighbor's bulb is equivalent to half one's own, and everybody has a 60-watt bulb, everybody can read as long as he and both his neighbors have their lights on. Arranged in a circle, everybody will keep his light on if everybody else does (and nobody will if his neighbors do not); arranged in a line, the people at the ends cannot read anyway and the whole thing unravels.*

A Game-Theoretic Formulation. The k-core can be seen as the maximal equilibrium in a natural game-theoretic model; it has been studied previously in this guise in the social sciences literature [8,9,22]. Concretely, imagine that each node in a social network G is considering whether to remain engaged in a social activity. We suppose that each node v in G incurs an (integer) cost of $k > 0$ for the effort it must spend to remain engaged. Node v also obtains a benefit of 1 from each neighbor w who is engaged; this reflects the idea that the benefit from participation in the activity comes from interaction with neighbors in the social network.

If each node makes its decision simultaneously, we can model the situation as a simultaneous-move game in which the nodes are the players, and v's possible strategies are to remain engaged or to drop out. For a choice of strategies σ by each player, let S_σ be the set of players who choose to remain engaged. The payoff of v is 0 if it drops out, and otherwise it is v's degree in the induced subgraph $G[S_\sigma]$ minus k. Note that we can talk about sets of engaged nodes and strategy profiles interchangeably.

There is a natural structure to the set of pure Nash equilibria in this game: σ is an equilibrium if and only if $G[S_\sigma]$ has minimum degree k (so that no engaged player wants to drop out), and no node in $V - S_\sigma$ has k or more neighbors in S_σ (so that no player who has dropped out wants to remain engaged). There will generally be multiple equilibria — for example, if G has minimum degree at least k, then $S_\sigma = \emptyset$ and $S_\sigma = V$ define two of possibly many equilibria. There is always a unique maximal equilibrium σ^*, in the sense that S_{σ^*} contains S_σ for all other equilibria σ. This maximal equilibrium is easily seen to correspond to the k-core of G — the unique maximal set S^* of minimum internal degree at least k in G.

Chwe [8,9] and Saaskilahti [22] argue that it is reasonable to assume that this maximal equilibrium will be selected in an actual play of the game, since it optimizes the welfare of all the players simultaneously (as well as the provider of the service, whose goal is to attract a large audience). That is, all incentives are aligned to coordinate on this equilibrium.

The Anchored k-Core Problem. The unraveling described in Schelling's line example is often undesirable, and could represent the end of a social network, a product, or a public good. When and how can such unraveling be prevented? For instance, in Schelling's example, the solution is clear: giving the two readers at the ends an extra lamp yields persistent illumination for all.

We formalize this problem as the *anchored k-core* problem. In the most basic version of the problem, the input is an undirected graph and two parameters $k, b \in \{1, 2, \ldots, n\}$, where b denotes a budget. Solutions

(a) The original graph G

(b) The 3-core of G

Fig. 1. k-core for $k = 3$ on an example graph G

correspond to subsets of at most b vertices, which are said to be *anchored*. Anchored vertices remain engaged no matter what their friends do — for example, due to internal properties (loyalty, constraints, etc.) or external incentives like rewards for participation, or rebates for using a product. The anchored k-core, corresponding to the final subgraph of engaged individuals, is computed like the k-core, except that anchored vertices are never deleted. That is, unanchored vertices with degree less than k are deleted iteratively, in any order. In Schelling's line example, anchoring the two endpoints causes the anchored 2-core to be equal to the entire network. Another example, with $b = 2$ and $k = 3$, is displayed in Figure 2.

(a) An assignment of 2 anchors to G

(b) The vertices saved in an anchored 3-core (with budget 2) of G

Fig. 2. Anchored 3-core with budget 2 on G

Summarizing, we have seen that cascades of withdrawals can cause an unraveling of engagement, but that such cascades can sometimes be prevented by anchoring (i.e., rendering non-strategic) a small number of individuals. The goal of this paper is to study systematically the optimization problem of anchoring a given number of individuals to maximize the amount of engagement in a social network — to maximize the size of the resulting anchored k-core. Solving this problem identifies the individuals whose participation is most crucial to the overall health of a community. Once identified, a designer can focus attention and resources on retaining the participation of these critical individuals.

Our Results. We first study general graphs, where we identify a "phase transition" in the computational complexity of the anchored k-core problem with respect to the parameter k. (The problem is interesting for all $k \geq 2$.) Here, we prove the following.

1. The anchored 2-core problem is solvable in polynomial time in arbitrary graphs. Our algorithm is greedy and natural, and we show that a suitable implementation of it runs in linear time with modest constant factors. Establishing the correctness of this greedy algorithm requires a careful argument.

2. For every $k \geq 3$, we prove that the anchored k-core problem admits no non-trivial polynomial-time approximation algorithm. Precisely, we prove that it is NP-hard to distinguish between instances in which $\Omega(n)$ vertices are in the optimal anchored k-core, and those in which the optimal anchored k-core has size only $O(b)$. This inapproximability result holds even for a natural resource augmentation version of the problem. We also prove, for every $k \geq 3$, that the problem is $W[2]$-hard with respect to the budget parameter b.

Our negative results motivate studying the anchored k-core problem in restricted classes of graphs, and here we provide positive results.

3. For arbitrary k, we prove that the anchored k-core problem can be solved exactly in polynomial time in graphs with bounded treewidth. Our polynomial-time algorithm extends to many natural variations of the problem: for directed graphs, for non-uniform anchoring costs, for vertex-specific values of k, and others.

Further Related Work. Our mathematical model of user engagement in social networks appears to be new, although it is related to a number of previous works in the social sciences literature. Saaskilahti's model [22] is the closest to ours — the payoff structure in [22] includes ours as a special case, though only a few special network topologies are considered (the complete graph, the cycle, and the star). Earlier economic models that capture positive network effects of participation but consider only the complete graph are given in Arthur [4] and Katz and Shapiro [16]. Blume [5], Ellison [11], and Morris [20] analyze economic models with general network topolgies, but these works focus on competing behaviors rather than on positive network effects, resulting in models different from ours.

The papers cited above focus on equilibrium analysis and do not consider algorithms for optimizing an equilibrium, as we do here. The problem of identifying influential nodes of a social network in order to incite cascades, introduced by Kempe et al. [17] and studied further in [18,21], shares some of the spirit of the optimization problem studied in the present work. This work on influence maximization also inspired several models on revenue maximization in social networks with positive network externalities [15,19,3,2,1], where a seller must trade off the benefits of low prices (wider adoption in the long run) and high prices (greater immediate revenue). These works focus on diffusion processes resulting from sequential and myopic decision-making by users, as opposed to the focus on maximal Nash equilibria taken here.

2 Anchored 2-Core

In this section, we give a linear-time algorithm that solves the anchored k-core problem when $k = 2$. We begin by making a necessary generalization of the anchored k-core problem, which we call anchored k-core with thresholds.

Definition 1 (Anchored k-Core with Thresholds). *Let $G = (V, E)$ be an undirected graph with $|V(G)| = n$, $b \in \mathbb{Z}$, and a threshold function $\mathsf{threshold} : V(G) \to \mathbb{Z}$. The* anchored k-core with thresholds problem *asks to identify a subset S of size at most b that maximizes the size of the largest induced subgraph H of G such that every vertex $v \in V(H)$ either has degree at least $\mathsf{threshold}(v)$ or is a member of S.*

The anchored k-core problem described in the introduction is equivalent to the anchored k-core problem with all thresholds equal to k.

2.1 Preprocessing: The Algorithm RemoveCore(G)

We next define a preprocessing step that make use of the special structure of 2-cores. Conceptually, it reduces the anchored 2-core problem on general graphs to the problem on forests.

Specifically, the subroutine $\mathsf{RemoveCore}(G)$ works as follows on the input graph G. It first computes the set U of vertices of the 2-core of G, and contracts all vertices in U down to a single vertex r. (If G is initially disconnected, we also identify the different components of the 2-core into a single vertex r.) This is easy to do in linear time. What remains is a single tree with vertex r, and zero or more other trees. We denote this output by $(r, \mathcal{R}, \mathcal{S})$, where $\mathcal{R}$ is the tree that contains r and $\mathcal{S}$ is the rest of the trees. The vertex r denotes the vertices of G that are always in the anchored 2-core, and we set its threshold to 0. All other vertices have a threshold of 2. To simplify the presentation, we assume that $\mathcal{R}$ and r exist; this can be enforced, if needed, by adding an isolated node with threshold 0.

Proposition 1. *Let G' be the output of $\mathsf{RemoveCore}(G)$. Every optimal solution to the anchored k-core problem with threshold on G' is an optimal solution the anchored k-core problem on G.*

Intuitively, each vertex in the 2-core would remain in the graph without the assistance of any anchors. So, we can think of these vertices as already being anchored, and the structure within the 2-core no longer affects the anchored 2-core of G. Thus, it is enough to devise a solution for the graph $\mathsf{RemoveCore}(G)$ in order to obtain a solution for the anchored 2-core problem on G.

2.2 An Efficient Exact Algorithm

We now show how to solve the anchored 2-core problem. We describe this algorithm intuitively and present it more explicitly as Algorithm 2.1 below. For the initial description of the algorithm, we solves instances with $b \leq 4$ by trying all $\binom{n}{b} = O(n^4)$ possible solutions; we later discuss a linear-time implementation. Let $(r, \mathcal{R}, \mathcal{S})$ denote the output of $\mathsf{RemoveCore}(G)$.

Our algorithm is greedy. We find two vertices $v_1, v_2 \in V(\mathcal{R})$ such that placing an anchor at v_1 maximizes the number of vertices saved across all placements of a single anchor in $V(\mathcal{R})$, and v_2 does the same assuming that v_1 has already been placed. In other words, v_1 will be a vertex furthest from r, and v_2 will be the next vertex furthest from r, after v_1 has already been selected and all vertices on the r-v_1 path contracted. Next, we find $v_3, v_4 \in V(\mathcal{S})$ such that placing anchors at v_3 and v_4 maximizes the number of vertices saved across all placements of two anchors in $V(\mathcal{S})$. In other words, v_3 and v_4 are on the endpoints of a longest path, maximized over all trees within $\mathcal{S}$.

Let $c_{\mathcal{R}}(v_1)$ and $c_{\mathcal{R}}(v_2)$ denote the number of vertices saved by placing anchors at v_1 and v_2, respectively. Similarly, let $c_{\mathcal{S}}(v_3, v_4)$ be the number of vertices saved by placing anchors together at v_3 and v_4. If $c_{\mathcal{R}}(v_1) + c_{\mathcal{R}}(v_2) > c_{\mathcal{S}}(v_3, v_4)$, then we place an anchor at v_1 and decrease b by 1. Otherwise, we place anchors at v_3 and v_4 and decrease b by 2. After the anchor placement, we set $(r', \mathcal{R}', \mathcal{S}')$ to be the output of $\mathsf{RemoveCore}(\mathcal{R} \sqcup \mathcal{S})$ and repeat the process of determining $\{v_1, v_2, v_3, v_4\}$ until $b < 5$, using $(r', \mathcal{R}', \mathcal{S}')$ in place of $(r, \mathcal{R}, \mathcal{S})$. Finally, when $b < 5$, we simply try all $\binom{n}{b}$ possible ways to place the remaining anchors, taking the set which maximizes the resulting k-core.

In the next section, we prove the following theorem.

Theorem 1. *Algorithm 2.1 yields an anchored 2-core of maximum size.*

2.3 Proof of Theorem 1

Before we move onto the proof of the main theorem, we warm up with the following lemma.

Lemma 1. *Let T be a tree in $\mathcal{S}$. Let P be a longest path with endpoints at u and v. within T. Then, assuming $b \geq 2$, there exists an optimal assignment on T which anchors u and v.*

Algorithm 2.1 A polynomial-time, exact algorithm for anchored 2-core

$(r, \mathcal{R}, \mathcal{S}) \leftarrow \mathsf{RemoveCore}(G)$ // $\mathcal{S}$ is a forest
$S \leftarrow \emptyset$
while $b \geq 5$ **do**
  $v_1 \leftarrow$ a vertex furthest from r
  $v_2 \leftarrow$ a vertex r after the r-v_1 is contracted
  $(v_3, v_4) \leftarrow$ a pair of vertices on the endpoints of a longest path across trees in $\mathcal{S}$
  if $c_{\mathcal{R}}(v_1) + c_{\mathcal{R}}(v_2) > c_{\mathcal{S}}(v_3, v_4)$ **then**
    $S \leftarrow S \cup \{v_1\}, b \leftarrow b - 1$ // Place an anchor on v_1
  else
    $S \leftarrow S \cup \{v_3, v_4\}, b \leftarrow b - 2$ // Place anchors on v_3 and v_4
  $(r, \mathcal{R}, \mathcal{S}) \leftarrow \mathsf{RemoveCore}(\mathcal{R} \sqcup \mathcal{S})$ // G modified due to newly anchored vertices
$U \leftarrow$ best placement of the remaining b anchors // Obtained by trying all $\binom{n}{b}$ possible placements of b anchors
$S \leftarrow S \cup U, b \leftarrow 0$

Proof. Let S be an optimal assignment of $b \geq 2$ anchors. Let $x \in S$ be the vertex such that $d(u, L(x, P))$ is minimized over all choices of $x \in S$, where $L(x, P)$ denote the vertex of P that is closest to x. Define y such that it minimizes $d(v, L(y, P))$. See also Figure 2.3. We claim that moving x and y to u and v only improves the solution. To see this, note that if we move the anchors on x and y to $L(x, P)$ and $L(y, P)$, the assignment loses value at most $d(x, L(x, P)) + d(y, L(y, P))$. Then, by moving these two anchors to u and v, the assignment gains value at least $d(u, L(x, P)) + d(v, L(y, P))$. Since u and v are endpoints of a longest path of T, the lemma follows. □

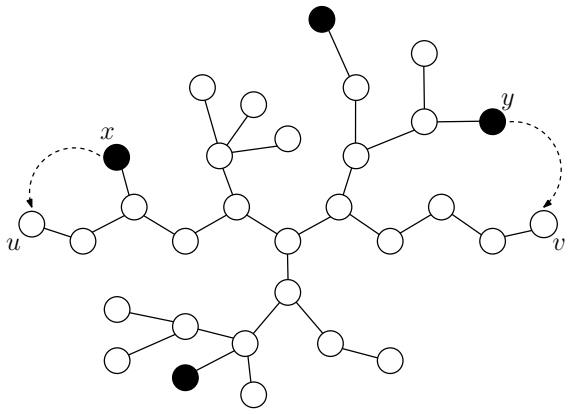
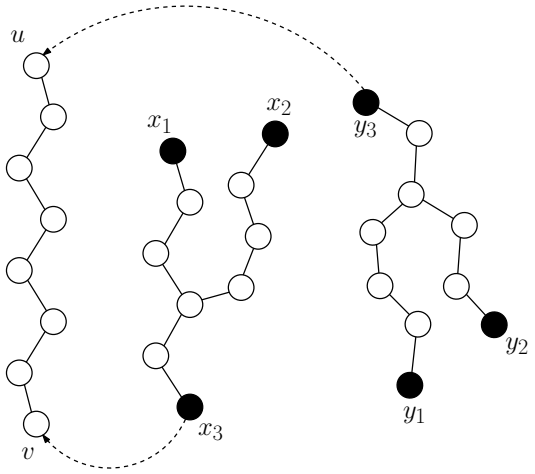


Fig. 3. Illustration for Lemma 1.

Case Analysis. We prove correctness of the algorithm by an inductive argument. At each iteration of the algorithm, we show that, if the algorithm places an anchor at some vertex v, then there must exist an optimal assignment of anchors that also places an anchor at v. It then follows that the algorithm terminates with an optimal assignment.

Although the algorithm is greedy in nature and rather straightforward, the proof of correctness is more involved. Assume that the algorithm has made a partial assignment $\mathcal{A}$ of anchors, and has b anchors left to place. By the inductive assumption, there exists an optimal assignment which also places anchors at the vertices of $\mathcal{A}$. Call this assignment `OPT`. For two vertices u, v belonging to some tree T, let $P(u, v)$ be the



v

Fig. 4. A diagram for Case 7.

set of vertices that lie on the (unique) path between u and v, including its endpoints. For a vertex v and set $S \subset V(G)$, let $L(v, S)$ be a vertex $u \in S$ that is closest to v amongst all members of S. We proceed with a case-by-case analysis. The first three cases and the remaining cases justify iterations in which our algorithm places one anchor in $\mathcal{R}$ and two anchors in $\mathcal{S}$, respectively.

Case 1: Suppose the algorithm places an anchor at a vertex $v \in V(\mathcal{R})$. Suppose OPT places at least one anchor in $V(\mathcal{R})$. Let $S \subseteq V(\mathcal{R})$ be the set of anchors that the optimal assignment places in $V(\mathcal{R})$. Let $u = \text{argmax}_{w \in S} |P(w, r) \cap P(v, r)|$. In other words, u is the vertex anchored by the optimal assignment whose path to the root of the tree intersects the most with the path to the root of the tree from v. Let a be the vertex with $P(a) = P(u, r) \cap P(v, r)$ — the least common ancestor of u and v. By removing u from the optimal assignment, the assignment loses at most $d(a, u)$ vertices, and by placing the anchor at v, the assignment gains $d(a, v)$ vertices. By the greedy criterion used to choose v, this swap must produce another optimal assignment.

Case 2: Suppose the algorithm places an anchor at $v \in V(\mathcal{R})$, but OPT does not place any anchors in $V(\mathcal{R})$, and only places anchors within $\mathcal{S}$. Also, suppose that an optimal assignment places exactly two anchors x and y in some tree T within $\mathcal{S}$. Since our algorithm places a single anchor in $V(\mathcal{R})$, it must be the case that $c_{\mathcal{R}}(v_1) + c_{\mathcal{R}}(v_2) > c_{\mathcal{S}}(v_3, v_4)$, where $v_1 = v$, v_2 is another vertex in $V(\mathcal{R})$, and the pair (v_3, v_4) represent a pair of vertices of maximum distance in $V(\mathcal{S})$. Since this optimal assignment did not place any anchors in $V(\mathcal{R})$, we can move the anchors from x and y to v_1 and v_2 to form a superior assignment.

Case 3: Suppose the algorithm places an anchor at $v \in V(\mathcal{R})$, but OPT does not place any anchors in $V(\mathcal{R})$, and only places anchors within $\mathcal{S}$. Also, suppose that an optimal assignment places at least three anchors in some tree T within $\mathcal{S}$, and let x be one of these anchors. (We can assume that OPT places zero or at least two anchors in each tree of $\mathcal{S}$.) Since our algorithm places a single anchor in $V(\mathcal{R})$, it must be the case that $c_{\mathcal{R}}(v_1) + c_{\mathcal{R}}(v_2) > c_{\mathcal{S}}(v_3, v_4)$, where $v_1 = v$, v_2 is another vertex in $V(\mathcal{R})$, and the pair (v_3, v_4) represent a pair of vertices of maximum distance in $V(\mathcal{S})$. We now analyze the cost of removing

the anchor from x. We claim that the value lost from removing x cannot exceed $\frac{1}{2}d(y,z)$, where the pair y, z are anchors used by the optimal assignment within T that achieve the maximum distance $d(y,z)$ across all pairs of anchors used by the optimal assignment within T. To see this, let w be the vertex on the path between y and z that is closest to x. First, $d(w,x)$ is an upper bound on the number of anchors lost when x is removed from the assignment. Second, since $d(y,z) \geq \max\{d(x,y), d(x,z)\}$, $d(x,w) \leq \min\{d(w,y), d(w,z)\} \leq \frac{1}{2}d(y,z)$. On the other hand, $d(y,z) \leq c_{\mathcal{S}}(v_3,v_4) \leq 2c_{\mathcal{R}}(v_1)$, since $c_{\mathcal{R}}(v_1) \geq c_{\mathcal{R}}(v_2)$. Thus, moving an anchor from x to v can only improve the assignment.

Case 4: Suppose the algorithm places two anchors $u, v \in V(\mathcal{S})$, both in some tree T within $\mathcal{S}$, and suppose that an optimal assignment also places at least two anchors in T. Lemma 1 immediately implies that we obtain an optimal assignment from `OPT` by moving two of its anchors to u and v.

Case 5: Suppose the algorithm places two anchors $u, v \in V(\mathcal{S})$, both in some tree T within $\mathcal{S}$, and `OPT` places exactly two anchors x, y in some tree T' within $\mathcal{S}$ where $T' \neq T$, and no anchors in T. Since $d(u,v) \geq d(x,y)$, we can remove the anchors from x and y, causing us to lose at most $d(x,y)+1$ value, and place them at u and v. Placing these anchors at u and v will cause us to gain exactly $d(u,v)+1$ value, since `OPT` did not place any anchors in T. Thus, by our greedy choice of u and v, we have another optimal assignment which uses u and v.

Case 6: Suppose the algorithm places two anchors $u, v \in V(\mathcal{S})$, both in some tree T within $\mathcal{S}$. Suppose, there exists a tree $T' \neq T$ within $\mathcal{S}$ where `OPT` places at least four anchors in T', and `OPT` does not place any anchors in T. Let $S \subseteq V(T)$ be the anchors that the assignment uses within T'. We now show the existence of a pair $x, y \in S$ such that removing x and y from the assignment will cause the value to decrease by at most $d(x,y)+1$; the rest of the argument is the same as Case 5. To see this, pick an arbitrary quadruple of anchors $q, s, r, w \in S$. If it is not already the case that removing q and s causes the value of the assignment to decrease by at most $d(q,s)+1$, then it must be the case that removing q and r causes the value of the assignment to decrease by at most $d(q,r)+1$.

Case 7: Suppose the algorithm places two anchors $u, v \in V(\mathcal{S})$, both in some tree T within $\mathcal{S}$, and an optimal assignment only places anchors within trees of $\mathcal{S}$, three anchors at a time for each tree, and does not place any anchors in T. Suppose the optimal assignment also uses at least two such trees T_1 and T_2 within $\mathcal{S}$. Let x_1, x_2, x_3 be the three vertices used by the assignment in T_1, labeled such that $d(x_1,x_2) \geq d(x_2,x_3)$ and $d(x_1,x_2) \geq d(x_1,x_3)$. Label y_1, y_2, y_3 for T_2 analogously. We claim that moving x_3 and y_3 to u and v results in another optimal assignment. As seen in Lemma 1, the value lost by removing x_3 cannot exceed $\frac{1}{2}d(x_1,x_2)$, and the value lost by removing y_3 cannot exceed $\frac{1}{2}d(y_1,y_2)$. Since $d(u,v) \geq \max(d(x_1,x_2), d(y_1,y_2)) \geq \frac{1}{2}d(x_1,x_2) + \frac{1}{2}d(y_1,y_2)$, the claim follows. See also Figure 2.3.

Case 8: Suppose the algorithm places two anchors $u, v \in V(\mathcal{S})$, both in some tree T within $\mathcal{S}$, and `OPT` does not place any anchors in T, and places at least two anchors in $\mathcal{R}$. By our algorithm's greedy criterion, moving these two anchors from $\mathcal{R}$ to u and v can only improve the solution.

Case 9: Suppose the algorithm places two anchors $u, v \in V(\mathcal{S})$, both in some tree T within $\mathcal{S}$, and an optimal assignment only places anchors within trees of $\mathcal{S}$, three anchors at a time for each tree. Suppose the optimal assignment uses exactly one tree T' within $\mathcal{S}$, and places at most one anchor in $V(\mathcal{R})$. Then, $b \leq 4$, and our algorithm computes an optimal anchor assignment by brute force.

2.4 A Linear-Time Implementation

We now proceed to our linear-time implementation of Algorithm 2.1. First let's understand the possible optimal solutions when $b = 3, 4$. We use the following notation. Let r_1, r_2, r_3, r_4 denote the four best options to anchor from within $V(\mathcal{R})$, where as usual r_i is defined with respect to previous choices, and assuming the corresponding paths have been contracted. Let (s_1, s_2) and (s_3, s_4) represent the two best options to anchor from within $V(\mathcal{S})$, using two separate trees. Let (t_1, t_2, t_3) represent the best triple of

options to anchor from within $V(\mathcal{S})$. Let (u_1, u_2, u_3, u_4) represent the best quadruple of nodes to anchor from within the same tree of $\mathcal{S}$. We show later how to keep track of all the relevant options in linear time.

$b = 3$. Compare between $c_{\mathcal{R}}(r_1) + c_{\mathcal{R}}(r_2) + c_{\mathcal{R}}(r_3)$, $c_{\mathcal{S}}(s_1, s_2) + c_{\mathcal{R}}(r_1)$, and $c_{\mathcal{S}}(t_1, t_2, t_3)$. Take the set of three anchors which maximizes across these options. To see that this achieves an optimal assignment for $b = 3$, fix some optimal assignment $\texttt{OPT}$. Since $b = 3$, $\texttt{OPT}$ must either use 0 trees in $\mathcal{S}$ (hence, placing all anchors within $V(\mathcal{R})$), or use 1 tree in $\mathcal{S}$, for which it either places exactly 2 anchors in (and the last anchor within $V(\mathcal{R})$) or exactly 3 anchors in $\mathcal{S}$. One can see that since we take the maximum of these three options, we must achieve an optimal assignment.

$b = 4$. Compare between $c_{\mathcal{R}}(r_1) + c_{\mathcal{R}}(r_2) + c_{\mathcal{R}}(r_3) + c_{\mathcal{R}}(r_4)$, $c_{\mathcal{S}}(s_1, s_2) + c_{\mathcal{R}}(r_1) + c_{\mathcal{R}}(r_2)$, $c_{\mathcal{S}}(s_1, s_2) + c_{\mathcal{S}}(s_3, s_4)$, $c_{\mathcal{S}}(t_1, t_2, t_3)$ + $c_{\mathcal{R}}(r_1)$, and $c_{\mathcal{S}}(u_1, u_2, u_3, u_4)$. Take the set of four anchors which maximizes across these options. To see that this achieves an optimal assignment for $b = 4$, fix some optimal assignment $\texttt{OPT}$. Since $b = 4$, $\texttt{OPT}$ must either use 0 trees in $\mathcal{S}$ (hence, placing all anchors within $V(\mathcal{R})$), use 1 tree in $\mathcal{S}$, for which it either places exactly 2 anchors in (and two anchors within $V(\mathcal{R})$), exactly 3 anchors in (and the last anchor in $V(\mathcal{R})$), or exactly four anchors in. Finally, $\texttt{OPT}$ could use two separate trees in $\mathcal{S}$, each placing exactly two anchors in. One can see that since we take the maximum of these four options, we must achieve an optimal assignment.

We continue with the full explanation of the linear-time implementation of Algorithm 2.1. In this section, we prove the following theorem.

Theorem 2. *Algorithm 2.1 can be implemented to run in time $O(m + n)$.*

Define a *successive path ordering* of the leaves of $\mathcal{R}$ to be a permutation $\pi = \pi_1, \ldots, \pi_t$ of the leaves such that π_1 is the leaf at maximum distance from r, π_2 is the leaf at maximum distance from r after the r-π_1 path has been contracted, and so on.

Lemma 2. *Let T be a tree with n nodes, and let r be the root vertex. A successive path ordering can be computed in time $O(n)$.*

Proof. We construct a successive path ordering π iteratively, adding vertices one at a time in order. We first show how to construct a dynamic programming table $\mathsf{Table}\colon V(T) \to V(T) \times [n]$. Let r be the root of T. For each vertex u in T, set $\mathsf{Table}(u) = (\mathsf{furthest}(u), \mathsf{height}(u))$. This table is constructed in a bottom-up manner—for each leaf ℓ of each tree, $\mathsf{Table}(\ell) := (\ell, 0)$, and then for each non-leaf vertex u,

$$\mathsf{Table}(u) := (\mathsf{furthest}(\operatorname*{argmax}_{v \in \mathsf{Children}(u)} \mathsf{height}(v)), \max_{v \in \mathsf{Children}(u)} \mathsf{height}(v) + 1).$$

In other words, the entry $\mathsf{Table}(u)$ is obtained by using the entries of its children, since the furthest vertex from u within $T(u)$ is the vertex that is furthest from the child node with the greatest height, and the height of u is simply one more than the maximum height across u's children. We initially populate $\mathsf{Table}(u)$ for all $u \in T$ in a single pass, in time $O(n)$. The semantics of a table entry (v, h) for a vertex u are: "if none of u's proper descendants have been saved yet, then the optimal placement for an anchor in u's subtree is at v, which will save h vertices (not counting u itself).

We maintain a bucket system consisting of n buckets, denoted by $B_1, \ldots, B_n$. Now, for the root vertex r, let $(v, h) := \mathsf{Table}(r)$. We place the entry (v, h) into the bucket B_h. We will simultaneously perform an array scan over the n buckets while updating its members. We keep a scanning pointer p, an integer which will always be equal to the maximum $z \in [n]$ such that B_z is non-empty (initially h). We then consider an element (x, z) from B_z, appending x to the successive path ordering π. Then, we remove all edges on the

path from x to r. As we traverse the path towards r, for each vertex u that we encounter on the path, we also remove u and all of its incident edges. Conceptually, we are contracting the vertex neighborhood of u, $N(u)$, with the root vertex r. For each vertex w that was adjacent to a member of the path but did not lie on the path — conceptually, these are the new children of the root, after the latest path contraction — we find the entry $\mathsf{Table}(w) = (v, h)$ and add this entry to bucket B_{h+1}.

We repeat this process of scanning for the maximum element until the buckets are all empty and the resulting graph has become the trivial graph after contractions. The entire process runs in linear time. □

We process the tree $\mathcal{R}$ in the above manner. The procedure for processing trees within $\mathcal{S}$ is slightly different. To preprocess a tree T within $\mathcal{S}$, we first find a maximum path within T (in linear time), storing this value into the bucket system for $\mathcal{S}$. Then, we contract the vertices along this maximum path, and treat the contracted vertex r to be of threshold 0. We then run the same preprocessing step from Lemma 2 on this modified tree.

Linear-Time Implementation. After the preprocessing step, the graph G can be completely ignored, and we are left with the bucket systems $B^{\mathcal{R}}$, $B^{\mathcal{S}}$ (corresponding to maximum paths of trees of $\mathcal{S}$), and B^{T_i} for each $T_i \in \mathcal{S}$ (corresponding to rooted trees, after maximum paths have been contracted), where each B^{T_i} has an index m_i precomputed such that the bucket indexed at m_i is the maximum non-empty bucket within B^{T_i}.

We now show how to simulate the original polynomial time algorithm, Algorithm 2.1, with a linear time implementation using these bucket systems. We will show how to recover the optimal solution in a linear-time pass for when $b \geq 5$, first. Then, for $b \leq 4$, we can perform a second linear-time pass to recover the optimal solution.

We keep two pointers p_1 and p_2 pointing to the buckets containing the top two entries within $B^{\mathcal{R}}$. Thus, we define $B^{\mathcal{R}}_{p_1} = \max_{B_i \in B^{\mathcal{R}}, B_i \neq \emptyset} B_i$, and $B^{\mathcal{R}}_{p_2} = B^{\mathcal{R}}_{p_1}$ if $|B^{\mathcal{R}}_{p_1}| \geq 2$, and $B^{\mathcal{R}}_{p_2} = \max_{B_i \in B^{\mathcal{R}}, B_i \neq B^{\mathcal{R}}_{p_1}, B_i \neq \emptyset} B_i$ otherwise. Similarly, we will set q_1 to point to the bucket containing the top entry within $B^{\mathcal{S}}$, so that $B^{\mathcal{S}}_{q_1} = \max_{B_i \in B^{\mathcal{S}}} B_i \neq \emptyset$. If we are able to maintain p_1, p_2, q_1 such that they always point to the respective top buckets of the bucket systems, then this is enough to simulate the exact algorithm. Therefore, it remains to prove that we can update these three pointers with only linear time overhead.

To initialize, we start from $B^{\mathcal{R}}_n$ and repeatedly lower the index until we find a non-empty bucket, allowing us to set p_1 and p_2. Similarly, we start from $B^{\mathcal{S}}_n$ and scan downwards until we find a non-empty bucket, which determines q_1. Now, we can inductively assume that the three pointers have been set correctly after the i^{th} anchor has been placed. Then, the algorithm either takes the element corresponding to p_1, or the element corresponding to q_1. If the algorithm takes the element corresponding to p_1, then we can update p_1 and p_2 by scanning downwards, starting from $B^{\mathcal{R}}_{p_1}$ and $B^{\mathcal{R}}_{p_2}$, until the next non-empty buckets are found. Note that q_1 remains unchanged, since the set $\mathcal{S}$ has not been modified. If, however, the algorithm chooses the element corresponding to q_1, then more work is needed. We can update q_1 in a similar manner by starting at $B^{\mathcal{S}}_{q_1}$ and scanning downwards until the next non-empty bucket is found. However, recall that when the algorithm selects some $T_i \in \mathcal{S}$, this tree now becomes rooted and must be accounted for in $\mathcal{R}$. To do this, we must merge the elements from B^{T_i} into $B^{\mathcal{R}}$. Now, recall that m_i represents the maximum index for which the bucket is non-empty within B^{T_i}. To update p_1 and p_2, we set $p_1 = \max(p_1, m_i)$ and then scan downwards from p_1 to get p_2. This concludes the process for updating the three pointers. Two observations explain why this process takes linear time over the course of the algorithm. First, every downward scan after the initialization can be uniquely charged to vertices that were saved in the previous iteration. Second, only $O(n)$ vertices are saved over the course of the algorithm.

We conclude with the cases where $b < 5$. When $b \leq 2$, the process is the same. When $b = 3, 4$, we need to construct a larger set of pointers. For $b = 3$, we can identify the elements p_1, p_2, p_3, s_1, s_2 in the same manner as before with a linear scan over the buckets of $B^{\mathcal{R}}$ and $B^{\mathcal{S}}$. However, to determine t_1, t_2, t_3,

we must do something different. For each element in $B^{\mathcal{S}}$ (which corresponds to some tree $T_i \in \mathcal{S}$), we add the cost of the element with the cost of the maximal element of B^{T_i}. Then, we take the maximum over these sums. The entire process for determining the t_1, t_2, t_3 can be done through a scan over $B^{\mathcal{S}}$ and identifying the buckets corresponding to the indices m_i for each B^{T_i}, which is done in linear time. Note that by Lemma 1, a maximal assignment of 3 anchors within a tree is such that two of the anchors are on a maximal path within the tree. Thus, we can correctly identify all elements needed for $b = 3$ in linear time.

For $b = 4$, we can identify the elements $p_1, p_2, p_3, s_1, s_2, s_3, s_4$ in the same manner as before with a linear scan over the buckets of $B^{\mathcal{R}}$ and $B^{\mathcal{S}}$. Also, we can determine t_1, t_2, t_3 using the same process for the $b = 3$ case. It remains to show how to determine u_1, u_2, u_3, u_4. Again, for each element in $B^{\mathcal{S}}$ (which corresponds to some tree $T_i \in \mathcal{S}$), we add the cost of the element with the cost of the maximal element of B^{T_i} along with the second maximal element of B^{T_i}. To determine the second maximal element, we can perform a linear scan of B^{T_i} starting from index m_i until the first non-empty bucket is reached. Then, we take the maximum over these sums. The entire process for determining the u_1, u_2, u_3, u_4 can be done through a scan over $B^{\mathcal{S}}$ and identifying the buckets corresponding to the indices m_i for each B^{T_i}, along with a scan through each B^{T_i} starting from index m_i, which is done in linear overall time. Note that by Lemma 1, a maximal assignment of 4 anchors within a tree is such that two of the anchors are on a maximal path within the tree. Thus, we can correctly identify all elements needed for $b = 4$.

This concludes the proof of Theorem 2.

3 Inapproximability of $k \geq 3$

The natural next step is to determine the complexity of the anchored k-core problem for $k \geq 3$. Note that every solution to the anchored k-core problem has objective function value in the range $[b, n]$. In this section we show that for $k \geq 3$, it is NP-hard to approximate the optimal anchored k-core to within a factor of $O(n^{1-\epsilon})$, for every positive constant $\epsilon > 0$.

3.1 A Preliminary Construction

Our reduction is from the Set Cover problem. Fix an instance I of set cover with n sets $S_1, \ldots, S_n$ and m elements $\{e_1, \ldots, e_m\} = \bigcup_{i=1}^{n} S_i$. We first give the construction only for instances of set cover such that for all i, $|S_i| \leq k - 1$. Then, we show how to lift this restriction while obtaining the same results.

We now define a corresponding anchored k-core instance. Let H be an arbitrarily large graph where every vertex has degree k except for a single vertex with degree $k-1$—call this vertex $t(H)$. Now, consider the graph consisting of a set of nodes $\{v_1, \ldots, v_n\}$ associated with sets $S_1, \ldots, S_n$ and a set $B = \{H_1, \ldots, H_m\}$ consisting of m disjoint copies H_j of H, where each copy of H is associated with an element e_j. There is an edge between v_i and $t(H_j)$ if and only if $e_j \in S_i$.

Lemma 3. *Fix $k \geq 3$. For every instance of set cover with maximum set size at most $k - 1$, there is a set cover of size z if and only if there exists an assignment in the corresponding anchored k-core instance using only z anchors such that all vertices in B are saved.*

Proof. Notice that the H_j's are designed such that if there exists some i such that v_i is adjacent to $t(H_j)$, then all vertices in H_j will be saved. Thus, if there is a set cover $\mathcal{C}$ of size z, then one can place the z anchors at v_i for all i such that $S_i \in \mathcal{C}$ and hence save all vertices in B. For the converse, we see that it is enough to restrict attention to assignments with anchors placed on v_i's. Since we are assuming that $|S_i| < k$ for all sets, each v_i will not be saved unless anchored. Thus, we must anchor some vertex adjacent to $t(H_j)$ for each copy H_j, which corresponds precisely to a set cover of size z.

Now, define $\mathsf{tree}(d, y)$ to be a perfect d-ary tree (each node has exactly d children) with exactly y leaves. To lift the restriction on the maximum set size, we replace each instance of v_i with $\mathsf{tree}(k-1, |S_i|)$, and if $y_1, \ldots, y_{|S_i|}$ are the leaves. (We can add dummy elements as needed to ensure that each $|S_i|$ has size compatible with such a tree.) For each $e_\ell \in S_i$, we contract the pairs of vertices (y_ℓ, e_ℓ). Extending the previous proof (with the tree roots serving the roles of the $t(H_j)$'s) yields the following.

Lemma 4. *Fix $k \geq 3$. For every instance of set cover, there is a set cover of size z if and only if there exists an assignment in the corresponding anchored k-core instance using only z anchors such that all vertices in B are saved.*

3.2 The Reduction from Set Cover

At this point, we have already shown that obtaining an optimal solution for the anchored k-core problem for $k \geq 3$ is NP-hard. Now, there exists a way to arrange the edges between each copy of H such that if there exists some vertex of B which is not saved, then *the majority* of the vertices will not be saved, either. Since this construction is somewhat complicated, we first show how to complete the reduction given such a construction. The details for the construction are thus deferred to the next section. From here on, we refer to the full construction as the graph $G(c, I)$, where I is an instance of Set Cover and c is an arbitrarily large constant.

Recall the decision problem for (unweighted) Set Cover: Given a collection of sets $\mathcal{C}$ which contain elements from a universe of size m, does there exist a set cover of size at most ℓ? We are able to show that $G(c, I)$ has the following two properties:

1. If I is a yes-instance, then there exists an assignment of ℓ anchors such that at least $km^{c+1}\ell$ vertices are saved.
2. If I is a no-instance, then no assignment of ℓ anchors can save more than $km\ell$ anchors.

Since c can be arbitrarily large, we can ensure that $km^{c+1}\ell = \Omega(n)$, where n is the number of vertices in the anchored k-core instance. We can therefore conclude with the following theorem and corollary.

Theorem 3. *It is NP-hard to distinguish between instances of anchored k-core where the optimal solution has value $\Omega(n)$ versus when the optimal solution has value $O(b)$.*

Corollary 1. *It is NP-hard to approximate the anchored k-core problem on general graphs within an $O(n^{1-\epsilon})$ factor.*

3.3 The Full Construction

We now describe how to construct such a graph $G(c, I)$ with the desired properties. The reader may wish to refer to Figure 3.3, which shows the full construction used in the reduction for the specific case of $k = 4$. We first consider the graph which we will call $\mathsf{column}(N, d)$, obtained by adjoining N instances of the clique on d vertices, K_d, by a single edge between each consecutive clique. Define $\mathsf{matrix}(N, m, d)$, which will contain m instances of $\mathsf{column}(N, d)$ (along with some additional edges between the columns to be defined). We require now that $d \geq 4$, although we will later consider when $d = 3$. Arrange the m columns vertically shown in Figure 3.3, along with a sink vertex $\perp$.

The organization of the edges within $\mathsf{matrix}(N, m, d)$ can be described as follows: Intuitively, we want the columns to be connected in such a way that the failure to save a single vertex in some column would "unravel" all other columns (and hence all other vertices) from the k-core in the matrix.

More formally, we impose an ordering on the columns in the matrix, so that we can now refer to the ith column for $i \in [1, m]$. Furthermore, impose an ordering on the cliques in each column so that the jth clique

in column i is the jth closest clique to the bottom of the column. Now, let $s_{i,j}$ denote the set of vertices of degree $d-1$ in the jth clique of the ith column. Partition $s_{i,j}$ into two nonempty sets $p_{i,j}$ and $q_{i,j}$. As a sanity check, note that $|s_{i,j}| = d-2$. Also, since $d-2 \geq 2$, such a partitioning is always possible.

Let $f(i,j) = (i+1) + (j-1 \pmod{m-1})$. We will pair up the vertices in $p_{i,j}$ and $q_{f(i,j),j}$, and add an edge between the vertices in each pair. If $|p_{i,j}| > |q_{f(i,j),j}|$, then we link the extra vertices in $p_{i,j}$ to the sink vertex $\perp$ (they are not important). After adding these edges, note now that each column has only one degree $d-1$ vertex: the vertex furthest from the bottom. We will call these the tips of each column. This completes the definition of the matrix.

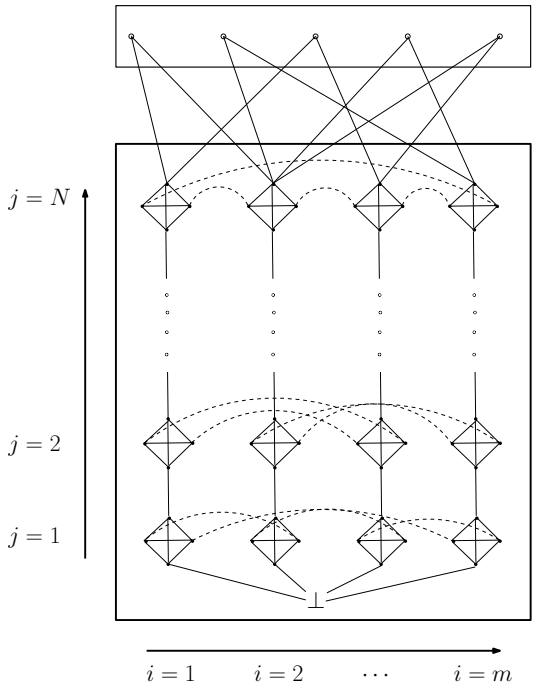


Fig. 5. Given an instance of Set Cover, we construct a graph where each vertex in the top row represents a set, and each column in the bottom grid represents an element (which we call $\mathsf{matrix}(N, m, d)$). We connect each of the set vertices to the top of the column of elements that they cover. The edges between the columns are construed in a way such that if one element (column) is not covered by a set, then all columns will be affected in an adversarial manner.

Consider $\mathsf{matrix}(N, m, d)$ where N is extremely large (to be set later) and m is the number of elements in the Set Cover instance. For each set S of the instance I, create an instance of $\mathsf{tree}(k-1, |S|)$, call it S_T. Now, merge the leaves of S_T with the tips of the columns that represent the elements that belong to S. Any extra leaves are merged with the sink, since they do not matter. Call this new graph $G(c, I)$, where $N \geq m^c \ell$. Note that c can be set to be arbitrarily large.

3.4 Proof of Correctness for Reduction

Proposition 2. *If there is a set cover using all of the sets, then the only vertices in $G(c, I)$ of degree less than k are the roots of the trees.*

We would like to reduce this problem to finding an approximate solution to anchored k-core on $G(c, I)$, with budget ℓ. Concretely, we will prove the following lemmas.

Lemma 5. *If there is a set cover for I of size at most ℓ, then the optimal anchored k-core for $G(c, I)$ with budget ℓ will save at least Nmk vertices.*

Proof. This follows easily from Proposition 2, counting only the $\geq Nmk$ vertices in the matrix. □

For the next proof we will need some additional notation to make things more convenient. Assume that N is a multiple of $m - 1$. For each column, call the set of the first $m - 1$ cliques the first patch P_1, the set of the second $m - 1$ cliques the second patch P_2, etc. The following proposition follows the edges between columns that we add in our construction.

Proposition 3. *Fix a column C. For all columns D, for each patch P_i, there exists a vertex in P_i of C that is adjacent to a vertex in P_i of D.*

Now, we have enough notation defined in order to show that if there is no set cover of small enough size, then there is no optimal solution that uses a small number of anchors and saves most of the vertices in the graph. Set $\lambda(I) = km\ell$.

Lemma 6. *If there is no set cover for I of size at most ℓ, then the optimal anchored k-core for $G(c, I)$ with budget ℓ is at most $\lambda(I)$.*

Proof. Consider an arbitrary assignment of ℓ anchors, and let A be the set of vertices where these anchors are. We first show the existence of a column C such that there does not exist an $a \in A$ that either lies in C or is in a tree that intersects with the tip of C. For, if this were not the case, then every column would be such that there is an anchor in the column or there is an anchor in a tree S_T that covers the element represented by the column. Then, we could simply move all anchors within columns to the trees that are incident to the columns that they are associated with and get a valid set covering, which is a contradiction. Note now that no vertex in C will remain in the anchored k-core. This is because the tip of C has only $k - 1$ neighbors, since no tree that intersects with the tip of C has an anchor placed in it. This will also cause all other vertices in C to be deleted, since there are no anchors in C to stop the iterative process. Now, Proposition 3 implies that for each column $D \neq C$, for each patch P_i of D, if no anchor is placed in P_i, then all vertices in P_i will disappear. Thus, the maximum number of patches that A can save is ℓ. Each patch consists of $(m-1)k$ vertices, and so A saves at most $\lambda(I) = \ell mk$ vertices. □

Additionally, it is important to mention the following corollary, as it shows that being able to save more than $\lambda(I)$ vertices would imply directly that the Set Cover instance can be solved efficiently.

Corollary 2. *If an algorithm, given budget ℓ, outputs an assignment A that saves more than $\lambda(I)$ vertices, then A can be used to find a set cover for I.*

Proof. Similar to the proof of Lemma 6, we know that if there exists a column that is not saved, then the assignment will save at most $\lambda(I)$ anchors. The contrapositive can be applied here, implying that all columns are saved. Thus, we can just push the anchors up to a tree incident to each column, and that is a valid set cover for I that uses only ℓ sets. □

Recall that we have set $\lambda(I) = km\ell$. Thus, by Lemma 5, Lemma 6, and Corollary 2, we have the necessary construction $G(c, I)$ used to conclude Theorem 3. We now show how to extend this construction to the $k = 3$ case.

Extending to 3-Core. To show that Theorem 3 holds for $k = 3$, we slightly modify the construction of $\mathsf{column}(N, d)$. Recall that $\mathsf{column}(N, 3)$ is currently defined to adjoin N instances of K_3 together. Each K_3 thus has a vertex of degree 2. Call this vertex v. We will add an additional K_3 and connect one of the vertices of the new K_3 to v. This graph is sometimes called the "bowtie" graph. Thus, in our new construction of $\mathsf{column}(N, 3)$, we have N instances of the bowtie graph adjoined, and now note that there are 2 vertices of degree 2 in each instance of the bowtie. Thus, we can place one of these vertices in $p_{i,j}$ and the other in $q_{i,j}$ as done for $k \geq 4$, and the rest of the proof remains the same.

3.5 Resource Augmentation Extensions

Suppose we are interested in comparing the performance of an algorithm given a budget of $O(b \cdot \alpha)$ anchors against the optimal assignment given only b anchors, for some $\alpha > 1$. Here, we augment the original reduction above to yield the following result.

Corollary 3. *For $\alpha = o(\log n)$, unless P = NP, there does not exist a polynomial time algorithm which, given $O(b \cdot \alpha)$ anchors, finds a solution within an $O(n^{1-\epsilon})$ multiplicative factor of the optimal solution with b anchors.*

Here, we use the inapproximability (not just NP-hardness) of the Set Cover problem. Specifically, Set Cover is not approximable by an $o(\log n)$ factor unless P = NP [14]. We now show how to prove Corollary 3. The intuitive idea behind this extension is to use instances $\mathsf{matrix}(N, d)$ to represent the sets of the Set Cover problem rather than instances of $\mathsf{tree}(x, y)$. For a fixed h, we can think of our construction of having h "layers", where the original construction lies at layer 0, and n^i copies of the matrix gadget lie at layer i, in general. The edges that cross between layer i an $i - 1$ will correspond to n^{i-1} instances of a fixed set cover instance. More specifically, the instance of $\mathsf{matrix}(N, d)$ in layer i will represent a set that covers the elements in an instance in layer $i - 1$, and at the same time each column in the matrix will represent an element of a set in layer $i + 1$.

The 2-Layered Version. Fix a set cover instance I. We will think of the instance as consisting of pairs (i, j) such that the pair (i, j) is in the instance if and only if the set S_i contains element e_j. For now, we only describe the example with $h = 2$, as the extension to general h requires extra detail but is not too much more difficult. The construction consists of 3 layers of clusters of vertices.

In Layer 0, we will have a single matrix M^*. In Layer 1, we will have n instances of matrices $M_1, \cdots, M_n$. For each column c of matrix M_i in Layer 1, we will require that c is *strongly connected* (see definition below) to column c_j from M^* in Layer 0 if and only if the pair (i, j) is in the set cover instance I. Finally, in Layer 2, we will have n^2 trees, $T_{\ell,i}$ for $\ell, i \in [1, n]$. Then, for each M_ℓ in Layer 1, we will require that $T_{\ell,i}$ is connected to column c_j of M_ℓ if and only if the pair (i, j) is in the set cover instance I.

Let the n copies of the set cover instance between Layer 1 and Layer 2 be denoted $J_1, \ldots, J_n$. Let $M(J_i)$ be the matrix in Layer 1 corresponding to set cover instance J_i, and let $\mathcal{T}(J_i)$ be the set of trees associated with J_i. The organization of edges imply the following two properties.

Property 1. Suppose some tree $T \in \mathcal{T}(J_i)$ (from Layer 2) for a fixed J_i is not anchored at the root. If less than b anchors are placed within $M(J_i) \cup \mathcal{T}(J_i)$, then $M(J_i)$ is not fully saved.

Property 2. Suppose some matrix M (from Layer 1) is not fully saved. Then, M^* is not fully saved.

Definition 2 (Strongly Connected). *We now define what it means for two columns c_i and c_j (possibly from different layers) to be strongly connected. Let patch P_q refer to the qth set of cliques in a column c, of size $m+nm$. We require that there exists an edge between c_i and c_j within each patch P_q across the two columns.*

As before, we require that the columns within each matrix (M^* and each M_ℓ) be strongly connected within the matrix.

The Extended Reduction Algorithm. The reduction algorithm is as follows:

1. Run the approximation algorithm on the multi-layered gadget, and look at the assignment of b^2 anchors.
2. Extract the solution from M^* by identifying the set of indices S^* such that

$$S^* = \{i \mid \text{at least } b \text{ anchors placed in } M_i \text{ or a tree connected to } M_i\},$$

   and let S^* be the subset of the collection of sets to select in the set cover instance. Verify if this solution satisfies the set cover instance.
3. For each $\ell \in [1, n]$, look at M_ℓ for each M_ℓ that has at least b anchors in its system. We extract a solution S_ℓ by first moving all anchors within M_ℓ up to arbitrary trees that are connected to the columns containing anchors, and then selecting the roots of the trees that have been anchored. This defines the solution S_ℓ, which we then verify satisfies the set cover instance.
4. If none of the verifications affirmed a solution for the set cover instance, then output that it is a NO-instance. Otherwise, we can output that it is a YES-instance.

Proof (Sketch) of Correctness. First, we note that there exists a way to anchor the graph such that all vertices within M^* are saved. To do this, let C, a subset of the collection of sets of the set cover instance, be the optimal solution to the instance, where $|C| = b$. Then, we anchor the roots of each tree $T_{\ell,i}$, for all i such that $S_i \in C$, and for all ℓ such that $S_\ell \in C$. This requires only b^2 anchors. By the organization of the edges between Layer 1 and Layer 2, This will save all of the vertices in each matrix M_ℓ of Layer 1 such that $S_\ell \in C$. Them, by the organization of the edges between Layer 0 and Layer 1, this will save all vertices within M^* at Layer 0.

Recall that in the original construction, we organized the edges so that the columns are connected in such a way that if one vertex of one column is not saved by a solution, then the unraveling effect takes place and results in none of the columns being saved. The augmentation result we show here builds upon this idea by ensuring that if the columns corresponding to the sets chosen in layer $i + 1$ are not saved, then there can't exist a way to save the columns of layer i. Thus, by creating recursive copies of the construction in this layered manner, we can ensure that, for a fixed set cover instance with parameter ℓ, there exists a solution that uses only ℓ^h anchors and saves the majority of the vertices in the graph. However, unless P = NP, we must require $(\ell \log n)^h$ anchors in order to achieve a solution comparable to the optimal on ℓ^h anchors. This gap in approximation allows us to conclude Corollary 3.

W[2]-hardness with Respect to Budget. We can also show that the anchored k-core problem is not in FPT with respect to the budget parameter b. We establish this result via a reduction from the Dominating Set problem. Recall that the decision version of Dominating Set is as follows: given a budget ℓ, determine if there a subset S of vertices such that $|S| \leq \ell$ and each vertex in $V \setminus S$ is adjacent to a vertex in S. As shown in [10], this problem is W[2]-hard. Here, we establish that if the anchored k-core problem can be solved in time $O(f(b) \cdot \mathsf{poly}(n))$ for any $k \geq 3$, then Dominating Set is in FPT with respect to ℓ, and hence FPT = W[2].

Theorem 4. *For every $k \geq 3$, the anchored k-core problem is W[2]-hard with respect to the parameter b.*

Proof. Given a graph $G(V, E)$ with m edges and n vertices, we construct a graph G' as follows: Start with an instance of K_{k+1}, the clique on $k + 1$ vertices. For each $v_i \in V$, create a node u_i and attach it two $k - 1$ arbitrary vertices of K_{k+1}. Then, create $T_i = \mathsf{tree}(k - 1, |N(v_i) \cup \{v_i\}|)$, with $y_1, \ldots, y_\ell$ the leaves, and (pairwise) contract each leaf y_j with a u_j such that $v_j \in N(v_i) \cup \{v_i\}$. (Note here that we use the assumption that $k \geq 3$, for if $k = 2$, then $\mathsf{tree}(1, n)$ would not be possible to construct.)

We next show how to map an optimal ℓ-anchored k-core solution in G' to a dominating set in the original graph G, and vice versa. Consider an optimal ℓ-anchored k-core in the new graph G'. The $k+1$ vertices of clique form a k-core of the graph and hence are always included in the optimal solution. Any u_i has $k-1$ neighbors in the clique, and hence needs only one more neighbor to be included in the solution. If, for any solution to the anchored k-core, some u_i is anchored, then this anchor can always be moved to the root node of T_i — this modification does not decrease the value of the solution. Likewise, if one of the internal nodes of T_i is anchored, the anchor can be moved to the root node T_i, which can only increase the value of the solution. Thus, we may assume without loss of generality that there exists some optimal solution which places all anchors on the roots of the trees.

The root nodes of each T_i have degree $k-1$, and so they need to be anchored to be included in the ℓ-anchored k-core. Each time we anchor a root node of the tree gadget corresponding to node v, we include all of the nodes in the tree gadget along with copies of neighbors of v in V_2. Thus any ℓ-anchored k-core includes $\ell \cdot h + k + 1 + T$ nodes where T is the number of vertices in V_2 that are included in the ℓ-anchored k-core.

This solution can then be mapped to a Dominating Set solution in the original graph. The set of vertices corresponding to the tree gadgets whose roots are anchored will belong to the dominating set and set of vertices that correspond to vertices in V_2 that belong to the solution will be the ones that are dominated. Similarly given a solution to the Dominating Set problem, we can use the root nodes of the tree gadgets corresponding to the vertices in the dominating set as anchors. All the tree gadget nodes corresponding to the anchored roots along with nodes in V_2 corresponding to nodes that are dominated will included in the ℓ-anchored k-core.

Thus given an algorithm to exactly compute the ℓ-anchored k-core we can decide if there is dominating set of size ℓ. All we have to check is whether the optimal ℓ-anchored k-core has $\ell \cdot h + n + k + 1$ vertices. Thus, we can find an ℓ-anchored k-core in time $O(f(l) \cdot \mathsf{poly}(n,k))$ then we can decide if there is a dominating set of size ℓ in time $O(f(l) \cdot \mathsf{poly}(n))$.

4 Graphs with Bounded Treewidth

Although we see that the anchored k-core problem is hopelessly inapproximable on general graphs, we next give polynomial time exact algorithms for graphs with bounded treewidth. The treewidth of a graph is defined as the minimum width over all tree decompositions of the graph, where the width of a tree decomposition is one more than the size of the largest node in the tree decomposition (see [6] for a tutorial and survey on treewidth). In this section, we present an algorithm that runs in time $O(f(k,w) \cdot b^2) \cdot \mathsf{poly}(n)$, where $f(k,w) = (3(k+1)^2)^w$, using $w-1$ as the graph's treewidth. To distinguish the vertices of a tree decomposition from the vertices of the original graph, we will call the elements of a tree decomposition nodes, and the elements of the original graph will remain as vertices. We will use the concept of nice tree decompositions for graphs, defined in [6] — the idea that a tree decomposition can be converted into another tree decomposition (a "nice" one) of the same treewidth and $O(n)$ nodes, but with the special property that each node comes in one of four types:

- Leaf Node: Only one vertex is associated with this node
- Introduce Node: The node has a single child, and if X is the set of vertices associated with this node and Y is the child, then $X = Y \cup \{v\}$ for some v.
- Forget Node: The node has a single child, and if X is the set of vertices associated with this node and Y is the child, then $X = Y \setminus \{v\}$ for some v.
- Join Node: The node has two children, and if X is the set of vertices associated with this node, Y and Z are its children, then $X = Y = Z$.

We show how to solve a generalization of the anchored k-core problem. Let $\mathsf{threshold}(v)$ represent the *threshold* of v, which is the minimum number of neighbors that v requires in order to remain in the k-core (assuming v is not anchored). Traditionally, in k-core, we use $\mathsf{threshold}(v) = k$ for all $v \in V(G)$. Since we are now considering a situation where the threshold function varies across vertices, we will instead use k to denote the maximum threshold across all vertices.

For a fixed assignment of anchors, a vertex is either *anchored*, *not saved*, or *indirectly saved* (not anchored, but saved). We show that the categorization of vertices into these three types is enough to capture the complexity of the problem on graphs with bounded treewidth. Define a *fixture* f of a tree decomposition T to be an assignment of these three types to a subset of $G[r(T)]$, the vertices of G that are associated with the root node of T. We say that an assignment A of anchors to vertices *satisfies* a fixture f if under the assignment A, the type of each vertex designated by the fixture agrees with the type induced by the assignment A. Define a threshold alteration m (which we will simply call an *alteration*) to be a setting of the thresholds of some subset $S \subseteq V(G)$ so that for each $v \in S$, m reduces the threshold of v by some integer in the interval $[0, \mathsf{threshold}(v)]$. We use the notation $m(T)$ to denote the tree obtained by lowering the thresholds of all vertices as prescribed by m.

4.1 The Algorithm

Algorithm 4.1 $\mathsf{Solve}(T)$: The main subroutine used in the exact algorithm for graphs with bounded treewidth.

$(\mathsf{Solutions}_{T_1}, \mathsf{Solutions}_{T_2}) \leftarrow (\mathsf{Solve}(T_1), \mathsf{Solve}(T_2))$
for all fixtures f, alterations m, and budgets b **do**
  if $r(T)$ is a leaf node **then**
    $\mathsf{Solutions}_T[f][m][b] \leftarrow$ the result dictated by the fixture f
  if $r(T)$ is a forget node **then**
    $S \leftarrow \{\text{fixtures } f' \text{ of } T_1 : \forall v \in G[r(T)], f(v) = f'(v)\}$
    $\mathsf{Solutions}_T[f][m][b] \leftarrow \max_{f' \in S} \mathsf{Solutions}_{T_1}[f'][m][b]$
  if $r(T)$ is an introduce node **then**
    Set f' to be such that $\forall v \in G[rT_1)], f'(v) = f(v)$
    Let v be the sole element of $G[r(T)] \setminus G[r(T_1)]$
    Set m' so that $\forall u \in N(v), m'(u) = m(u) - 1$ and $\forall u \notin N(v), m'(u) = m(u)$
    if $f(v)$ is anchored **then**
      $\mathsf{Solutions}_T[f][m][b] \leftarrow \mathsf{Solutions}_{T_1}[f'][m'][b-1]$
    if $f(v)$ is not saved **then**
      $\mathsf{Solutions}_T[f][m][b] \leftarrow \mathsf{Solutions}_{T_1}[f'][m][b]$
    if $f(v)$ is indirectly saved **then**
      $\mathsf{Solutions}_T[f][m][b] \leftarrow \mathsf{Solutions}_{T_1}[f'][m'][b]$
  if $r(T)$ is a join node **then**
    $\hat{b} \leftarrow b - |\{v : f(v) \text{ is anchored}\}|$
    $t \leftarrow \max_{i \in [0,\hat{b}], \hat{m} \in [0,k]^w} (\mathsf{Solutions}_{T_1}[f][\hat{m}][i] + \mathsf{Solutions}_{T_2}[f][m - \hat{m}][\hat{b} - i])$
    $\mathsf{Solutions}_T[f][m][b] \leftarrow t$
return $\mathsf{Solutions}_T$

We first define a subroutine: $\mathsf{Solve}(T)$ for a tree decomposition T, also outlined in Algorithm 4.1. The output of $\mathsf{Solve}(T)$ will be, for all fixtures f and all alterations m (within $r(T)$), and $\hat{b} \in [1, b]$, the table $\mathsf{Solutions}(m(T), f, \hat{b})$. Each entry of this table of solutions, $\mathsf{Solutions}_T[f][m][b]$, will describe an optimal assignment of b anchors which satisfies the fixture f on the graph $G[T]$, the vertices of G that are associated with the nodes of T, under alteration m. Note that if no such assignment that satisfies the stated restrictions can exist, then the output of the entry is $\perp$. (This could occur if, for example, the fixture f requires 3 nodes to be anchored yet $b < 3$.)

Thus, $\mathsf{Solve}(T)$ will output at most $(3(k+1))^w \cdot b$ solutions. The 3^w term is due to the fact that there are 3^w possible fixtures, and the $(k+1)^w$ term comes from the fact that each vertex has threshold at most k, and so there are at most $k+1$ choices for lowering the thresholds of each vertex in the root node (within the range $[0, k]$). We now show that given the outputs of $\mathsf{Solve}(T_i)$ for each child subtree T_i, we can compute $\mathsf{Solve}(T)$. This will be done through a case analysis on the node type of the root node, denoted by $r(T)$.

If $r(T)$ is a leaf node, then $\mathsf{Solutions}(m(T), f, b)$ is trivial to determine for all alterations m, fixtures f, and budgets b, as there is only one vertex associated with $r(T)$ and there are no other nodes in T (and so, an anchor is placed on this vertex if it is anchored under f and $b \geq 1$). Otherwise, let T_1 and T_2 denote the child subtrees of $r(T)$. If $r(T)$ is a forget node, then let v be the vertex that is associated with the child node but not in $G[r(T)]$. To find $\mathsf{Solutions}(m(T), f, b)$ for all f, m, and b, we simply compute the maximum solution over possible choices of the fixture type of v and the threshold alteration induced by the partial fixture of the other vertices in $G[r(T)]$.

If $r(T)$ is an introduce node, then let v be the vertex in $G[r(T)]$ that is not associated with the child node. For fixtures f such that v is anchored, we simply subtract one from the budget for T_1 and retrieve the optimal solution there under the induced partial fixture. This is obtained from the output of $\mathsf{Solutions}(m'(T), f, b)$, where $m'(u) = m(u) - 1$ if $u \in N(v)$ and $m'(u) = m(u)$ otherwise. If v is indirectly saved, we do not change the budget but obtain the optimal solution under the induced partial fixture for $\mathsf{Solutions}(m'(T), f, b)$. If v is not saved, we use the optimal solution on T_1 under the induced fixture f. Finally, if $r(T)$ is a join node, then let $i \in [1, b]$. Also, for a fixture f, let S be the set of all vertices in the root node that are indirectly saved. Note that $|S| \leq w$. We iterate over all of the at most $(k+1)^w$ possibilities of dividing up the thresholds for each $v \in R$ between T_1 and T_2. We then iterate over all pairs of solutions $\mathtt{OPT}(m(T_1), f, i)$ and $\mathtt{OPT}(m(T_2), f, b-i)$ and take the maximum over such i.

This approach is repeated in a bottom-up manner until we have covered the entire tree. Finally, we simply take the output of $\mathsf{Solve}(T)$ (the original tree decomposition T of the entire graph) and find the optimal solution corresponding to the tree on b anchors by taking the maximum value over all fixtures.

4.2 Proof of Correctness

In order to show that this algorithm correctly outputs the optimal solution, it is only needed to show that $\mathsf{Solve}(T)$'s output is correct given the outputs of $\mathsf{Solve}(T_i)$ for the child subtrees of T. Let $\mathsf{OptSol}(T, f, b)$ represent the optimal solution to T under a fixture f given b anchors. We thus perform a case analysis on the type of the root node $r(T)$. Recall that a nice tree decomposition is a tree decomposition such that the nodes come in four types:

- Leaf Node: Only one vertex is associated with this node
- Introduce Node: The node has a single child, and if X is the set of vertices associated with this node and Y for the child, then $X = Y \cup \{v\}$ for some v.
- Forget Node: The node has a single child, and if X is the set of vertices associated with this node and Y for the child, then $X = Y \setminus \{v\}$ for some v.
- Join Node: The node has two children, and if X is the set of vertices associated with this node, Y and Z for its children, then $X = Y = Z$.

For $r(T)$ being a leaf node, $\mathsf{Solve}(T)$ clearly outputs the optimal solution for all fixtures, alterations, and budget, as there is only one node in the graph. If $r(T)$ is a forget node, let v be the vertex that is associated with the child node but not in $G[r(T)]$. We want to show that $\mathsf{Solve}(T)$ outputs $\mathsf{OptSol}(m(T), f, b)$ for all fixtures f, alterations m, and budget b. Note that $G[T_1] = G[T]$ — in other words, although we are considering a new node $r(T)$ in the tree decomposition, no new vertices are being added to the subgraph under consideration. Thus, the set of optimal solutions for fixtures under $G[r(T)]$ can be obtained by taking a maximum over the solutions covered by the partial induced fixture by f of $G[r(T_1)]$.

If $r(T)$ is an introduce node, let v be the only vertex such that $v \in G[r(T)]$ but $v \notin G[r(T_1)]$. For any alteration m and each fixture f, we look at the type that v is assigned to. There are three cases to consider on the assignment of v by f. If v is not saved, then any solution on the graph without v would have the same objective function value when including v in the graph, and so it follows that $\mathsf{OptSol}(m(T_1), f, b) = \mathsf{OptSol}(m(T), f, b)$. Recall now that we define $m'(T)$ to be such that if $u \in N(v)$, $m'(u) = m(u) - 1$ and $m'(u) = m(u)$ otherwise. If v is anchored, then we can use $\mathsf{OptSol}(m'(T_1), f, b-1)$ as $\mathsf{OptSol}(m(T), f, b)$, since any solution on $m'(T_1)$ under fixture f and with budget $b-1$ will have increased its objective function by exactly 1 when mapped to $m(T)$ under f with budget b, but one anchor is placed on v. Similarly, if v is indirectly saved, then we use the same argument as in the previous case, except that no anchor is placed on v, so the value of any solution on $m'(T_1)$ under f with budget b is exactly one minus the solution on $m(T)$ under f with budget b with the guarantee that v is indirectly saved. Therefore, we use $\mathsf{OptSol}(m'(T_1), f, b)$ as $\mathsf{OptSol}(m(T), f, b)$).

Finally, if $r(T)$ is a join node, then again note that we are not introducing any new vertices into the subgraph under consideration. We want to show that $\mathsf{Solve}(T)$ will output $\mathsf{OptSol}(m(T), f, b)$ for all fixtures f, alteration m, and budget b. The main issue in this case, intuitively, is how to split the budget b and threshold restriction m across the outputs of $\mathsf{Solve}(T_1)$ and $\mathsf{Solve}(T_2)$.

Fix some $\mathsf{OptSol}(m(T), f, b)$ (under the current setting m, f, and b). First, let S_1 denote the set of all anchored vertices under f, and S_2 the set of all indirectly saved vertices under f. Furthermore, let i be the number of anchors placed in $G[T_1] \setminus G[r(T)]$ and j the number of anchors placed in $G[T_2] \setminus G[r(T)]$. Note that $b = i + j + |S_1|$ by our definitions. Also, for each $v \in S_2$, let $z(v) = \mathsf{threshold}(v) - |N(v) \cap S_1 \cap S_2|$. Intuitively, $z(v)$ is the number of vertices that must be saved within $G[T] \setminus G[r(T)]$ in $\mathsf{OptSol}(m(T), f, b)$ (since it must satisfy f). Thus, define $h_1(v)$ and $h_2(v)$ to be the number of vertices that are saved in $N(v) \cap (G[T_1] \setminus G[r(T)])$ and $N(v) \cap (G[T_2] \setminus G[r(T)])$, respectively. By our definitions, we have that $h_1(v) + h_2(v) \geq z(v)$. Thus, there exist a pair of integers $(g_1(v), g_2(v))$ such that $g_1(v) + g_2(v) = z(v)$ and $h_1(v) \geq g_1(v)$, and $h_2(v) \geq g_2(v)$. We now have enough information to show how our algorithm retrieves $\mathsf{OptSol}(m(T), f, b)$.

Note that our algorithm takes the maximum valued solution over such partitions of the budget and the thresholds on each vertex in S_2. Thus, the only way our algorithm will fail to output a solution at least as good as $\mathsf{OptSol}(m(T), f, b)$ is if all solutions outputted over all possible partitions of the budget and threshold across the two subtrees save less vertices than $\mathsf{OptSol}(m(T), f, b)$. In the previous paragraph, we have shown that there exists an i such that the optimal solution uses i anchors in T_1 and $b - i$ anchors in T_2 (excluding the root nodes), and there exists a threshold setting $g_1(v)$ for each $v \in S_2$ such that the optimal solution is achieved when setting $\mathsf{threshold}(v) = g_1(v)$ in T_1 and $\mathsf{threshold}(v) = z(v) - g_1(v)$ in T_2. Furthermore, $i \in [1, b]$ and $g_1(v) \in [0, k]$ and there are at most $|S_2| \leq w$ members of S_2. Thus, our algorithm considers the partition that achieves an optimal solution, and so it must yield a solution at least as good as the optimal.

This concludes the case analysis, and so we have thus shown that $\mathsf{Solve}(T)$ correctly outputs $\mathsf{OptSol}(m(T), f, b)$ for all fixtures f, alterations m, and budget b. Then, of course, the solution to the original tree T (the true decomposition of the graph) is obtained by considering the maximum over all fixtures f, using the true budget b, and under the alteration m such that $m(T) = T$.

Run-time Analysis. The size of the output of $\mathsf{Solve}(T)$ is at most $b \cdot (3(k+1))^w$, and note that if $r(T)$ is a join node, the operation takes time $b \cdot (k+1)^w$ to compute due to the number of possible partitionings of the budget and thresholds across T_1 and T_2. This process is repeated on $O(n)$ nodes of the tree, and so our total running time, given an oracle for a tree decomposition of the graph, is $O(n \cdot b^2 \cdot (3(k+1)^2)^w)$.

Generalizations. This dynamic programming approach allows for several generalizations to the anchored k-core problem on graphs with bounded treewidth, all of which can be solved exactly and run in time $O(f(k,w) \cdot \mathsf{poly}(n,b))$. For example, one can assign weights to vertices. Also, as we have already seen, the vertex thresholds can be non-uniform. Furthermore, the edges of the graph can be directed, and each arc $a = (u,v)$ can have a weight $w(a)$ such that ensuring that u is in the graph contributes a value of $w(a)$ to the threshold of v.

5 Concluding Remarks

There remain several attractive open problems related to the anchored k-core problem, especially on restricted classes of graphs. Is there a PTAS for planar graphs? What can be said about the problem on random graphs? Can the running time of our polynomial-time algorithm for bounded treewidth graphs be improved? Is there a linear-time algorithm for the anchored 2-core problem in general graphs?

References

1. H. Akhlaghpour, M. Ghodsi, N. Haghpanah, V. Mirrokni, H. Mahini, and A. Nikzad. Optimal iterative pricing over social networks. In *Proceedings of the 6th Workshop on Internet and Network Economics*, pages 415–423, 2010.
2. N. Anari, S. Ehsani, M. Ghodsi, N. Haghpanah, N. Immorlica, H. Mahini, and V. Mirrokni. Equilibrium pricing with positive externalities. In *Proceedings of the 6th Workshop on Internet and Network Economics*, pages 424–431, 2010.
3. David Arthur, Rajeev Motwani, Aneesh Sharma, and Ying Xu. Pricing strategies for viral marketing on social networks. In *Proceedings of the 5th Workshop on Internet and Network Economics*, 2009.
4. W. Brian Arthur. Competing technologies, increasing returns, and lock-in by historical events. *The Economic Journal*, 99(394):116–131, March 1989.
5. Larry Blume. The statistical mechanics of strategic interaction. *Games and Economic Behavior*, 5:387–424, 1993.
6. Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, pages 255–269, 2008.
7. Moira Burke, Cameron Marlow, and Thomas Lento. Feed me: motivating newcomer contribution in social network sites. In *CHI*, 2009.
8. Michael Suk-Young Chwe. Structure and strategy in collective action. *American Journal of Sociology*, 105(1):128–156, July 1999.
9. Michael Suk-Young Chwe. Communication and coordination in social networks. *Review of Economic Studies*, 67:1–16, 2000.
10. Rodney G. Downey, Michael R. Fellows, Catherine McCartin, and Frances Rosamond. Parameterized approximation of dominating set problems. *Information Processing Letters*, 109:68–70, 2008.
11. Glenn Ellison. Learning, local interaction, and coordination. *Econometrica*, 61:1047–1071, 1993.
12. Nicole B. Ellison, Charles Steinfield, and Cliff Lampe. The Benefits of Facebook Friends: Social Capital and College Students' Use of Online Social Network Sites. *Journal of Computer-Mediated Communication*, 2007.
13. Rosta Farzan, Laura A. Dabbish, Robert E. Kraut, and Tom Postmes. Increasing commitment to online communities by designing for social presence. In *CSCW*, 2011.
14. U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.
15. Jason Hartline, Vahab Mirrokni, and Mukund Sundararajan. Optimal Marketing Strategies over Social Networks. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.

16. Michael L. Katz and Carl Shapiro. Network externalities, competition, and compatibility. *American Economic Review*, 75(3):424–440, June 1985.
17. David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.
18. David Kempe, Jon Kleinberg, and Éva Tardos. Influential nodes in a diffusion model for social networks. In *ICALP*, pages 1127–1138, 2005.
19. Hamid Mahini, Vahab Mirrokni, Nima Haghpanah, Mohammad Ghodsi, Hessameddin Akhlaghpour, and Afshin Nikzad. Optimal Iterative Pricing over Social Networks. In *Proceedings of the Fifth Workshop on Ad Auctions*, 2009.
20. Stephen Morris. Contagion. *Review of Economic Studies*, 67:57–78, 2000.
21. Elchanan Mossel and Sébastien Roch. On the submodularity of influence in social networks. In *STOC*, pages 128–134, 2007.
22. Pekka Saaskilahti. Monopoly pricing of social goods. Technical report, University Library of Munich, Germany, 2007.
23. Thomas C Schelling. *Micromotives and Macrobehavior*. Norton, 1978.