# Best of Both Worlds: Practical and Theoretically Optimal Submodular Maximization in Parallel

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

For the problem of maximizing a monotone, submodular function with respect to a cardinality constraint $k$ on ground set of size $n$, we provide an algorithm that achieves the state-of-the-art in both its empirical performance and its theoretical properties, in terms of adaptive complexity, query complexity, and approximation ratio. Previously introduced algorithms with sublinear adaptive complexity are either theoretically nearly optimal but highly impractical or excellent in practice but far from theoretically optimal. Our main algorithm is assembled from two components which are of independent interest. The first component of our algorithm is the novel algorithm LINEARSEQ which achieves approximation ratio $\approx 1/4$ with high probability in $O(n)$ expected query complexity and $O(\log n)$ adaptivity – LINEARSEQ is independently useful as a preprocessing algorithm to improve the query complexity of many algorithms. Moreover, a variant of LINEARSEQ is shown to have adaptive complexity of $O(\log(n/k))$ which is smaller than that of any previous algorithm in the literature. The second component is a parallelizable thresholding procedure THRESHOLDSEQ for adding elements with gain above a constant threshold – THRESHOLDSEQ improves over several previous algorithms developed for the same task. Finally, we demonstrate that our main algorithm empirically outperforms, in terms of runtime, total queries, and objective value, the previous state-of-the-art algorithm FAST for this problem in a comprehensive evaluation with six submodular objective functions.

## 1 Introduction

The cardinality-constrained optimization of a monotone, submodular function $f : 2^{\mathcal{N}} \to \mathbb{R}^+$, defined on subsets of a ground set $\mathcal{N}$ of size $n$, is a general problem formulation that is ubiquitous in wide-ranging applications, *e.g.* video or image summarization [26], network monitoring [23], information gathering [20], and MAP Inference for Determinantal Point Processes [17], among many others. The function $f : 2^{\mathcal{N}} \to \mathbb{R}^+$ is submodular iff for all $S \subseteq T \subseteq \mathcal{N}$, $x \notin T$, $\Delta(x \mid T) \leq \Delta(x \mid S)$[1]; and the function $f$ is monotone if $f(S) \leq f(T)$ for all $S \subseteq T$. In this paper, we study the following submodular maximization problem (SM)

$$\text{maximize} f(S), \text{ subject to } |S| \leq k, \tag{SM}$$

where $f$ is a monotone, submodular function; SM is an $NP$-hard problem. Therefore, there has been extensive effort into the design of approximation algorithms for SM over the course of more than 45 years, *e.g.* [27, 11, 9, 18, 21]. For SM, the optimal ratio has been shown to be $1 - 1/e \approx 0.63$ [27].

As instance sizes have grown very large, there has been much effort into the design of efficient, parallelizable algorithms for SM. Since queries to the objective function can be very expensive, the

[1] $\Delta(x \mid S)$ denotes the *marginal gain* of $x$ to $S$: $f(S \cup \{x\}) - f(S)$.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

Table 1: Theoretical comparison to algorithms that achieve nearly optimal adaptivity, ratio, and query complexity: the algorithm of Ene and Nguyen [13], RANDOMIZED-PARALLEL-GREEDY (RPG) of Chekuri and Quanrud [10], BINARY-SEARCH-MAXIMIZATION (BSM) and SUBSAMPLE-MAXIMIZATION (SM) of Fahrbach et al. [15]. The symbol † indicates the result holds with constant probability or in expectation, while no symbol indicates the result holds with probability greater than $1 - O(1/n)$. Observe that our algorithm LS+PGB dominates in at least one category when compared head-to-head with any one of the previous algorithms.

| Reference | Ratio | Adaptivity | Queries |
|---|---|---|---|
| [13] | $1 - 1/e - \varepsilon$ | $O\left(\frac{1}{\varepsilon^2}\log(n)\right)$ | $O\left(n\text{poly}(\log n, 1/\varepsilon)\right)$ |
| RPG [10] | $1 - 1/e - \varepsilon$ | $O\left(\frac{1}{\varepsilon^2}\log(n)\right)$ † | $O\left(\frac{n}{\varepsilon^4}\log(n)\right)$ † |
| BSM [15] | $1 - 1/e - \varepsilon$ † | $O\left(\frac{1}{\varepsilon^2}\log(n)\right)$ | $O\left(\frac{n}{\varepsilon^3}\log\log k\right)$ † |
| SM [15] | $1 - 1/e - \varepsilon$ † | $O\left(\frac{1}{\varepsilon^2}\log(n)\right)$ | $O\left(\frac{n}{\varepsilon^3}\log(1/\varepsilon)\right)$ † |
| LS+PGB [Theorem 3] | $1 - 1/e - \varepsilon$ | $O\left(\frac{1}{\varepsilon^2}\log(n/\varepsilon)\right)$ | $O\left(\frac{n}{\varepsilon^2}\right)$ † |

overall efficiency of an algorithm for SM is typically measured by the *query complexity*, or number of calls made to the objective function $f$ [2, 8]. The degree of parallelizability can be measured by the *adaptive complexity* of an algorithm, which is the minimum number of rounds into which the queries to $f$ may be organized, such that within each round, the queries are independent and hence may be arbitrariliy parallelized. Observe that the lower the adaptivity complexity, the more parallelizable an algorithm is. To obtain a constant approximation factor, a lower bound of $\Omega(n)$ has been shown on the query complexity [21] and a lower bound of $\Omega(\log(n)/\log\log(n))$ has been shown on the adaptive complexity [3].

Several algorithms have been developed recently that are nearly optimal in terms of query and adaptive complexities [13, 10, 15, 4]; that is, these algorithms achieve $O(\log n)$ adaptivity and $O(n\text{polylog}(n))$ query complexity (see Table 1). However, these algorithms use sampling techniques that result in very large constant factors that make these algorithms impractical. This fact is discussed in detail in Breuer et al. [8]; as an illustration, to obtain ratio $1 - 1/e - 0.1$ with 95% confidence, all of these algorithms require more than $10^6$ queries of sets of size $k/\log(n)$ in every adaptive round [8]; moreover, even if these algorithms are run as heuristics using a single sample, other inefficiencies preclude these algorithms of running even on moderately sized instances [8]. For this reason, the FAST algorithm of Breuer et al. [8] has been recently proposed, which uses an entirely different sampling technique called adaptive sequencing. Adaptive sequencing was originally introduced in Balkanski et al. [5], but the original version has quadratic query complexity in the size of the ground set and hence is still impractical. To speed it up, the FAST algorithm sacrifices theoretical guarantees to yield an algorithm that parallelizes well and is faster than all previous algorithms for SM in an extensive experimental evaluation. The theoretical sacrifices of FAST include: the adaptivity of FAST is $\Omega(\log(n)\log^2(\log n))$, which is higher than the state-of-the-art, and most significantly, the algorithm obtains no approximation ratio for $k < 850$[2]; since many applications require small choices for $k$, this limits the practical utility of FAST. A natural question is thus: *is it possible to design an algorithm that is both practical and theoretically optimal in terms of adaptivity, ratio, and total queries?*

### 1.1 Contributions

Our results are organized into three main contributions. The first contribution is the algorithm LINEARSEQ (LS, Section 2) that achieves with probability $1 - 1/n$ a constant factor $(4 + O(\varepsilon))^{-1}$ in expected linear query complexity and with $O(\log n)$ adaptive rounds (Theorem 1). Although the ratio of $\approx 0.25$ is smaller than the optimal $1 - 1/e \approx 0.63$, this algorithm can be used to improve the query complexity of many extant algorithms, as we decribe in the Related Work section below. Interestingly, LINEARSEQ can be modified to have adaptivity $O(\log(n/k))$ at a small cost to its ratio as discussed in Appendix F. This version of LINEARSEQ is a constant-factor algorithm for SM with

[2]The approximation ratio $1 - 1/e - 4\varepsilon$ of FAST holds with probability $1 - \delta$ for $k \geq \theta(\varepsilon, \delta, k) = 2\log(2\delta^{-1}\log(\frac{1}{\varepsilon}\log(k)))/\varepsilon^2(1 - 5\varepsilon)$.

Table 2: Empirical comparison of our algorithm with FAST, with each algorithm using 75 CPU threads on a broad range of $k$ values and six applications; details provided in Section 4. Parameter settings are favorable to FAST, which is run without theoretical guarantees while LS+PGB enforces ratio $\approx 0.53$ with high probability. For each application, we report the arithmetic mean of each metric over all instances. Observe that LS+PGB outperforms FAST in runtime on all applications.

| | Runtime (s) | | Objective Value | | Queries | |
|---|---|---|---|---|---|---|
| Application | FAST | LS+PGB | FAST | LS+PGB | FAST | LS+PGB |
| TrafficMonitor | $3.7 \times 10^{-1}$ | $\mathbf{2.1 \times 10^{-1}}$ | $4.7 \times 10^{8}$ | $\mathbf{5.0 \times 10^{8}}$ | $3.5 \times 10^{3}$ | $\mathbf{2.4 \times 10^{3}}$ |
| InfluenceMax | $4.4 \times 10^{0}$ | $\mathbf{2.3 \times 10^{0}}$ | $1.1 \times 10^{3}$ | $1.1 \times 10^{3}$ | $7.7 \times 10^{4}$ | $\mathbf{4.0 \times 10^{4}}$ |
| TwitterSumm | $1.6 \times 10^{1}$ | $\mathbf{3.5 \times 10^{0}}$ | $3.8 \times 10^{5}$ | $3.8 \times 10^{5}$ | $1.5 \times 10^{5}$ | $\mathbf{6.2 \times 10^{4}}$ |
| RevenueMax | $3.9 \times 10^{2}$ | $\mathbf{5.4 \times 10^{1}}$ | $1.4 \times 10^{4}$ | $1.4 \times 10^{4}$ | $7.6 \times 10^{4}$ | $\mathbf{2.7 \times 10^{4}}$ |
| MaxCover (BA) | $3.7 \times 10^{2}$ | $\mathbf{7.6 \times 10^{1}}$ | $6.0 \times 10^{4}$ | $\mathbf{6.3 \times 10^{4}}$ | $5.8 \times 10^{5}$ | $\mathbf{1.8 \times 10^{5}}$ |
| ImageSumm | $1.6 \times 10^{1}$ | $\mathbf{8.1 \times 10^{-1}}$ | $9.1 \times 10^{3}$ | $9.1 \times 10^{3}$ | $1.3 \times 10^{5}$ | $\mathbf{4.8 \times 10^{4}}$ |

smaller adaptivity than any previous algorithm in the literature, especially for values of $k$ that are large relative to $n$.

Our second contribution is an improved parallelizable thresholding procedure THRESHOLDSEQ (TS, Section 3) for a commonly recurring task in submodular optimization: namely, add all elements that have a gain of a specified threshold $\tau$ to the solution. This subproblem arises not only in SM, but also *e.g.* in submodular cover [15] and non-monotone submodular maximization [22]. Our TS accomplishes this task with probability $1 - 1/n$ in $O(\log n)$ adaptive rounds and expected $O(n)$ query complexity (Theorem 2), while previous procedures for this task only add elements with an expected gain of $\tau$ and use highly impractical sampling techniques [15]; have $\Omega(\log^2 n)$ adaptivity [19]; or have $\Omega(kn)$ query complexity [5].

Finally, we present in Section 3 the parallelized greedy algorithm PARALLELGREEDYBOOST (PGB), which is used in conjunction with LINEARSEQ and THRESHOLDSEQ to yield the final algorithm LS+PGB, which answers the above question affirmatively: LS+PGB obtains nearly the optimal $1 - 1/e$ ratio with probability $1 - 2/n$ in $O(\log n)$ adaptive rounds and $O(n)$ queries in expectation; moreover, LS+PGB is faster than FAST in an extensive empirical evaluation (see Table 2). In addition, LS+PGB improves theoretically on the previous algorithms in query complexity while obtaining nearly optimal adaptivity (see Table 1).

### 1.2 Additional Related Work

**Adaptive Sequencing.** The main inefficiency of the adaptive sequencing method of Balkanski et al. [5] (which causes the quadratic query complexity) is an explicit check that a constant fraction of elements will be filtered from the ground set. In this work, we adopt a similar sampling technique to adaptive sequencing, except that we design the algorithm to filter a constant fraction of elements *with only constant probability*. This method allows us to reduce the quadratic query complexity of adaptive sequencing to linear query complexity while only increasing the adaptive complexity by a small constant factor; arguably, this is the right way to conduct adaptive sequencing. In contrast, FAST of Breuer et al. [8] speeds up adaptive sequencing by increasing the adaptive complexity of the algorithm through adaptive binary search procedures, which, in addition to the increasing the adaptivity by logarithmic factors, place restrictions on the $k$ values for which the ratio can hold. This improved adaptive sequencing technique is the core of our THRESHOLDSEQ procedure, which has the additional benefit of being relatively simple to analyze.

**Algorithms with Linear Query Complexity.** Our LINEARSEQ algorithm also uses the improved adaptive sequencing technique, but in addition this algorithm integrates ideas from the $\Omega(n)$-adaptive linear-time streaming algorithm of Kuhnle [21] to achieve a constant-factor algorithm with low adaptivity in expected linear time. Integration of the improved adaptive sequencing with the ideas of Kuhnle [21] is non-trivial, and ultimately this integration enables the theoretical improvement in query complexity over previous algorithms with sublinear adaptivity that obtain a constant ratio with high probability (see Table 1). In Fahrbach et al. [15], a linear-time procedure SUBSAMPLEPRE-

PROCESSING is described; this procedure is to the best of our knowledge the only algorithm in the literature that obtains a constant ratio with sublinear adaptive rounds and linear query complexity and hence is comparable to LINEARSEQ. However, SUBSAMPLEPREPROCESSING uses entirely different ideas from our LINEARSEQ and has much weaker theoretical guarantees: for input $0 < \delta < 1$, it obtains ratio $\frac{\delta^2}{2\times 10^6}$ with probability $1-\delta$ in $O(\log(n)/\delta)$ adaptive rounds and $O(n)$ queries in expectation – the small ratio renders SUBSAMPLEPREPROCESSING impractical; also, its ratio holds only with constant probability. By contrast, with $\varepsilon = 0.1$, our LINEARSEQ obtains ratio $\approx 0.196$ with probability $1 - 1/n$ in $O(\log(n))$ adaptive rounds and $O(n)$ queries in expectation.

**Using LS for Preprocessing: Guesses of OPT.** Many algorithms for SM, including FAST and all of the algorithms listed in Table 1 except for SM and our algorithm, use a strategy of guessing logarithmically many values of OPT. Our LINEARSEQ algorithm reduces the interval containing OPT from size $k$ to a small constant size in expected linear time. Thus, LINEARSEQ could be used for preprocessing prior to running FAST or one of the other algorithms in Table 1, which would improve their query complexity without compromising their adaptive complexity or ratio; this illustrates the general utility of LINEARSEQ. For example, with this change, the theoretical adaptivity of FAST improves, although it remains worse than LS+PGB: the adaptive complexity of FAST becomes $O\left(\frac{1}{\varepsilon^2}\log(n)\log\left(\frac{1}{\varepsilon}\log(k)\right)\right)$ in contrast to the $O\left(\frac{1}{\varepsilon^2}\log\left(n/\varepsilon\right)\right)$ of LS+PGB. Although SUBSAMPLEPREPROCESSING may be used for the same purpose, its ratio only holds with constant probability which would then limit the probability of success of any following algorithm.

**MapReduce Framework.** Another line of work studying parallelizable algorithms for SM has focused on the MapReduce framework [12] in a distributed setting, *e.g.* [6, 7, 14, 24]. These algorithms divide the dataset over a large number of machines and are intended for a setting in which the data does not fit on a single machine. None of these algorithms has sublinear adaptivity and hence all have potentially large numbers of sequential function queries on each machine. In this work, our empirical evaluation is on a single machine with a large number of CPU cores; we do not evaluate our algorithms in a distributed setting.

**Organization.** The constant-factor algorithm LINEARSEQ is described and analyzed in Section 2; the details of the analysis are presented in Appendix C. The variant of LINEARSEQ with lower adaptivity is described in Appendix F. The algorithms THRESHOLDSEQ and PARALLELGREEDYBOOST are discussed at a high level in Section 3, with detailed descriptions of these algorithms and theoretical analysis presented in Appendices D and E. Our empirical evaluation is summarized in Section 4 with more results and discussion in Appendix H.

## 2 A Parallelizable Algorithm with Linear Query Complexity: LINEARSEQ

In this section, we describe the algorithm LINEARSEQ for SM (Alg. 1) that obtains ratio $(4+O(\varepsilon))^{-1}$ in $O\left(\frac{1}{\varepsilon^3}\log(n)\right)$ adaptive rounds and expected $O\left(\frac{n}{\varepsilon^3}\right)$ queries. If $\varepsilon \leq 0.21$, the ratio of LINEARSEQ is lower-bounded by $(4+16\varepsilon)^{-1} \geq 0.135$, which shows that a relatively large constant ratio is obtained even at large values of $\varepsilon$. An initial run of this algorithm is required for our main algorithm LS+PGB.

**Description of LS.** The work of LS is done within iterations of a sequential outer **for** loop (Line 5); this loop iterates at most $O(\log(n))$ times, and each iteration requires two adaptive rounds; thus, the adaptive complexity of the algorithm is $O(\log(n))$. Each iteration adds more elements to the set $A$, which is initially empty. Within each iteration, there are four high-level steps: 1) filter elements from $V$ that have gain less than $f(A)/k$ (Line 6); 2) randomly permute $V$ (Line 8); 3) compute in parallel the marginal gain of adding blocks of the sequence of remaining elements in $V$ to $A$ (**for** loop on Line 11); 4) select a prefix of the sequence $V = \left(v_1, v_2, \ldots, v_{|V|}\right)$ to add to $A$ (Line 14). The selection of the prefix to add is carefully chosen to approximately satisfy, on average, Condition 1 for elements added; and also to ensure that, with constant probability, a constant fraction of elements of $V$ are filtered on the next iteration.

The following theorem states the theoretical results for LINEARSEQ. The remainder of this section proves this theorem, with intuition and discussion of the proof. The omitted proofs for all lemmata are provided in Appendix C.

**Theorem 1.** *Let* $(f, k)$ *be an instance of* SM. *For any constant* $0 < \varepsilon < 1/2$, *the algorithm* LINEARSEQ *has adaptive complexity* $O\left(\log(n)/\varepsilon^3\right)$ *and outputs* $A' \subseteq \mathcal{N}$ *with* $|A'| \leq k$ *such*

**Algorithm 1** The algorithm that obtains ratio $(4 + O(\varepsilon))^{-1}$ in $O\left(\log(n)/\varepsilon^3\right)$ adaptive rounds and expected $O\left(n/\varepsilon^3\right)$ queries.

1: **procedure** LINEARSEQ($f, \mathcal{N}, k, \varepsilon$)
2: **Input:** evaluation oracle $f : 2^{\mathcal{N}} \to \mathbb{R}^+$, constraint $k$, error $\epsilon$
3: $a = \arg\max_{u \in \mathcal{N}} f(\{u\})$
4: Initialize $A \leftarrow \{a\}$ , $V \leftarrow \mathcal{N}$, $\ell = \lceil 4(1 + 1/(\beta\varepsilon)) \log(n) \rceil$, $\beta = \varepsilon/(16 \log(8/(1 - e^{-\varepsilon/2})))$
5: **for** $j \leftarrow 1$ to $\ell$ **do**
6: Update $V \leftarrow \{x \in V : \Delta(x \mid A) \geq f(A)/k\}$ and filter out the rest
7: **if** $|V| = 0$ **then break**
8: $V = \{v_1, v_2, \ldots, v_{|V|}\}$ ←**random-permutation**($V$)
9: $\Lambda \leftarrow \{\lfloor (1+\varepsilon)^u \rfloor : 1 \leq \lfloor (1+\varepsilon)^u \rfloor \leq k, u \in \mathbb{N}\}$
$\cup \{\lfloor k + u\varepsilon k \rfloor : \lfloor k + u\varepsilon k \rfloor \leq |V|, u \in \mathbb{N}\} \cup \{|V|\}$
10: $B[\lambda_i] =$ **false**, for $\lambda_i \in \Lambda$
11: **for** $\lambda_i \in \Lambda$ in parallel **do**
12: $T_{\lambda_{i-1}} \leftarrow \{v_1, v_2, \ldots, v_{\lambda_{i-1}}\}$ ; $T_{\lambda_i} \leftarrow \{v_1, v_2, \ldots, v_{\lambda_i}\}$ ; $T'_{\lambda_i} \leftarrow T_{\lambda_i} \backslash T_{\lambda_{i-1}}$
13: **if** $\Delta\left(T'_{\lambda_i} \mid A \cup T_{\lambda_{i-1}}\right)/|T'_{\lambda_i}| \geq (1-\varepsilon) f(A \cup T_{\lambda_{i-1}})/k$ **then** $B[\lambda_i] \leftarrow$ **true**
14: $\lambda^* \leftarrow \max\{\lambda_i \in \Lambda : B[\lambda_i] =$ **false** and $((\lambda_i \leq k$ and $B[1]$ to $B[\lambda_{i-1}]$ are all **true**) or $(\lambda_i > k$ and $\exists m \geq 1$ s.t. $|\bigcup_{u=m}^{i-1} T'_{\lambda_u}| \geq k$ and $B[\lambda_m]$ to $B[\lambda_{i-1}]$ are all **true**))$\}$
15: $A \leftarrow A \cup T_{\lambda^*}$
16: **if** $|V| > 0$ **then return** *failure*
17: **return** $A' \leftarrow$ last $k$ elements added to $A$

*that the following properties hold: 1) The algorithm succeeds with probability at least* $1 - 1/n$. *2) There are* $O\left((1/(\varepsilon k) + 1)n/\varepsilon^3\right)$ *oracle queries in expectation. 3) If the algorithm succeeds,* $\left[4 + \frac{4(2-\varepsilon)}{(1-\varepsilon)(1-2\varepsilon)}\varepsilon\right] f(A') \geq f(O)$, *where* $O$ *is an optimal solution to the instance* $(f, k)$.

**Overview.** The goal of this section is to produce a constant factor, parallelizable algorithm with linear query complexity. As a starting point, consider an algorithm[1] that takes one pass through the ground set, adding each element $e$ to candidate set $A$ iff

$$\Delta(e \mid A) \geq f(A)/k. \tag{1}$$

Condition 1 ensures two properties: 1) the last $k$ elements in $A$ contain a constant fraction of the value $f(A)$; and 2) $f(A)$ is within a constant fraction of OPT. By these two properties, the last $k$ elements of $A$ are a constant factor approximation to SM with exactly one query of the objective function per element of the ground set. For completeness, we give a pseudocode (Alg. 2) and proof in Appendix B. However, each query depends on all of the previous ones and thus there are $n$ adaptive rounds. Therefore, the challenge is to approximately simulate Alg. 2 in a lowly adaptive (highly parallelizable) manner, which is what LINEARSEQ accomplishes.

### 2.1 Approximately Satisfying Condition 1

**Discarding Elements.** In one adaptive round during each iteration $j$ of the outer **for** loop, all elements with gain to $A$ of less than $f(A)/k$ are discarded from $V$ (Line 6). Since the size of $A$ increases as the algorithm runs, by submodularity, the gain of these elements can only decrease and hence these elements cannot satisfy Condition 1 and can be safely discarded from consideration. The process of filtering thereby ensures the following lemma at termination.

**Lemma 1.** *At successful termination of* LINEARSEQ, $f(O) \leq 2f(A)$, *where* $O \subseteq \mathcal{N}$ *is the optimal solution of size* $k$.

**Addition of Elements.** Next, we describe the details of how elements are added to the set $A$. The random permutation of remaining elements on Line 8 constructs a sequence $\left(v_1, v_2, \ldots, v_{|V|}\right)$ such that each element is uniformly randomly sampled from the remaining elements. By testing the marginal gains along the sequence in parallel, it is possible to determine a good prefix of the sequence

[1] This algorithm is a simplified version of the streaming algorithm of Kuhnle [21].

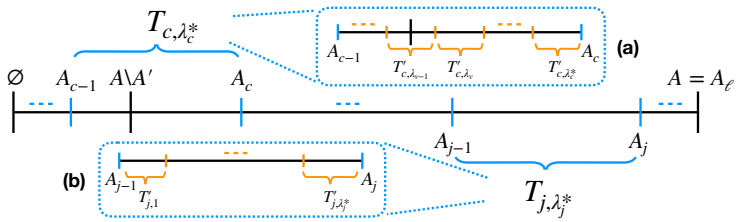


Figure 1: This figure depicts the way elements are added to $A$, with $A_i$ denoting the value of $A$ after iteration $i$ of the outer **for** loop; the prefix added at iteration $i$ is denoted by $T_{i,\lambda_i^*}$. The orange lines mark the blocks comprising each prefix. Also, the set $A'$ of the last $k$ elements added to $A$ is shown. Parts **(a)** and **(b)** depict two separate cases in the proof of Claim 1.

$(v_i)$ to add to $A$ to ensure the following: 1) Condition 1 is approximately satisfied; and 2) We will discard a constant fraction of $V$ in the next iteration with constant probability. Condition 1 is important for the approximation ratio and discarding a constant fraction of $V$ is important for the adaptivity and query complexity. Below, we discuss how to choose the prefix such that both are achieved. To speed up the algorithm, we do not test the marginal gain at each point in the sequence $(v_i)$, but rather test blocks of elements at once as determined by the index set $\Lambda$ defined in the pseudocode.

**Prefix Selection.** Say a block is *bad* if this block does not satisfy the condition checked on Line 13 (which is an approximate, average form of Condition 1); otherwise, the block is *good*. At the end of an iteration, we select the largest block index $\lambda^*$, where this block is bad and the previous consecutive blocks which together have at least $k$ elements are all good; or this block is bad and all the previous blocks are good blocks. Then, we add the prefix $T_{\lambda^*} = (v_1, v_2, \ldots, v_{\lambda^*})$ into $A$. Now, the relevance of Condition 1 for the approximation ratio is that it implies $f(A) \geq 2f(A \setminus A')$, where $A'$ are the last $k$ elements added to $A$. Lemma 2 shows that the conditions required on the marginal gains of blocks added imply an approximate form of this fact is satisfied by LINEARSEQ. Indeed, the proof of Lemma 2 informs the choice $\Lambda$ of blocks evaluated and the computation of $\lambda^*$.

**Lemma 2.** *Suppose* LINEARSEQ *terminates successfully. Then* $f(A) \geq \frac{2(1-\varepsilon+\varepsilon^2)}{1+\varepsilon} f(A \backslash A')$.

*Proof.* If $|A| \leq k$, the lemma is immediate, so assume $|A| > k$. For iteration $j$, let $T_{j,\lambda_j^*}$ denote the set added to $A$ during iteration $j$; and let $T_{j,\lambda_j^*} = \emptyset$ if the algorithm terminates before iteration $j$. Let $A_j$ denote the value of $A$ after iteration $j$. Define $c = \max\{c \in \mathbb{N} : A' \subseteq (\cup_{j=c}^{\ell} T_{j,\lambda_j^*})\}$. Then, $|T_{c,\lambda_c^*}| > 0$; and for any $j > c$, $|T_{j,\lambda_j^*}| < k$. It holds that $(\cup_{j=c+1}^{\ell} T_{j,\lambda_j^*}) \subset A' \subseteq (\cup_{j=c}^{\ell} T_{j,\lambda_j^*})$. Figure 1 shows how $A$ is composed of these sets $T_{j,\lambda_j^*}$ and how each set is composed of blocks. The following claim is proven in Appendix C.3.

**Claim 1.** It holds that $\Delta\left(T_{c,\lambda_c^*} \mid A \backslash A'\right) \geq (1-\varepsilon)\max\{0, |T_{c,\lambda_c^*} \cap A'| - 2\varepsilon k\} \cdot f(A \backslash A')/k$. For $j > c$, it holds that $\Delta\left(T_{j,\lambda_j^*} \mid A_{j-1}\right) \geq \frac{1-\varepsilon}{1+\varepsilon} |T_{j,\lambda_j^*}| \cdot f(A \backslash A')/k$.

From Claim 1,

$$f(A) - f(A \backslash A') = \Delta\left(T_{c,\lambda_c^*} \mid A \backslash A'\right) + \sum_{j=c+1}^{\ell} \Delta\left(T_{j,\lambda_j^*} \mid A_{j-1}\right) \geq \frac{(1-\varepsilon)(1-2\varepsilon)}{1+\varepsilon} \cdot f(A \backslash A'), \quad (2)$$

where Inequality 2 is proven in Appendix C.4. □

In the remainder of this subsection, we will show that a $(\beta\varepsilon)$-fraction of $V$ is discarded at each iteration $j$ of the outer **for** loop with probability at least $1/2$, where $\beta$ is a constant in terms of $\varepsilon$ as defined on Line 4 in the pseudocode. The remainder of the proofs in this section are implicitly conditioned on the behavior of the algorithm prior to iteration $j$. The next lemma describes the behavior of the number of elements that will be filtered at iteration $j+1$. Observe that the set $S_i$ defined in the next lemma is the set of elements that would be filtered at the next iteration if prefix $T_i$ is added to $A$.

**Lemma 3.** *Let* $S_i = \{x \in V : \Delta(x \mid A \cup T_i) < f(A \cup T_i)/k\}$. *It holds that* $|S_0| = 0$, $|S_{|V|}| = |V|$, *and* $|S_i| \leq |S_{i+1}|$.

By Lemma 3, we know the number of elements in $S_i$ increases from 0 to $|V|$ with $i$. Therefore, there exists a $t$ such that $t = \min\{i \in \mathbb{N} : |S_i| \geq \beta\varepsilon|V|\}$. If $\lambda^* \geq t$, $|S_{\lambda^*}| \geq \beta\varepsilon|V|$, and we will successfully filter out more than $(\beta\varepsilon)$-fraction of $V$ at the next iteration. In this case, we say that the iteration $j$ *succeeds*. Otherwise, if $\lambda^* < t$, the iteration may fail. The remainder of the proof bounds the probability that $\lambda^* < t$, which is an upper bound on the probability that iteration $j$ fails. Let $\lambda_t = \max\{\lambda \in \Lambda : \lambda < t\}$, and let $\lambda'_t = \max(\{\lambda' \in \Lambda : \sum_{\lambda \in \Lambda, \lambda' \leq \lambda \leq \lambda_t} |T_\lambda| \geq k\} \cup \{1\})$. If $\lambda^* < t$, there must be at least one index $\lambda$ between $\lambda'_t$ and $\lambda_t$ such that the block $T'_\lambda$ is bad. The next lemma bounds the probability that any block $T'_\lambda$, with $\lambda < \lambda_t$, is bad.

**Lemma 4.** *Let* $t = \min\{i \in \mathbb{N} : |S_i| \geq \beta\varepsilon|V|\}$; $\lambda_t = \max\{\lambda \in \Lambda : \lambda < t\}$; *and* $\alpha \in \Lambda$. *Then for any* $\lambda < \alpha$, $Pr\left(B[\lambda] = \textbf{\textit{false}} \mid \lambda_t = \alpha\right) \leq \min\{\beta, e^{-\frac{(1-\beta)^2}{1+\beta}\varepsilon|T'_\lambda|}\}$.

Finally, we bound the probability that an iteration $j$ of the outer **for** loop fails, conditioned on the event $\lambda_t = \alpha$ for fixed $\alpha \in \Lambda$. Let $B_1 = \{\lambda \in \Lambda : \lambda \leq k \text{ and } \lambda \leq \lambda_t\}$, $B_2 = \{\lambda \in \Lambda : |\Lambda \cap [\lambda, \lambda_t]| \leq \lceil 1/\varepsilon \rceil\}$; observe that $B_1$, $B_2$ are fixed sets if $\lambda_t = \alpha$. Then

$$Pr\left(\text{iteration } j \text{ fails} \mid \lambda_t = \alpha\right) \leq Pr\left(\exists \lambda \in B_1 \cup B_2 \text{ with } B[\lambda] = \textbf{false} \mid \lambda_t = \alpha\right) \leq 1/2, \quad (3)$$

where the proof of Inequality 3 is in Appendix C.7. By the Law of Total Probability, $Pr(\text{iteration } j \text{ fails}) \leq 1/2$ after unfixing the event $\lambda_t = \alpha$ and the behavior of the algorithm prior to iteration $j$.

### 2.2 Proof of Theorem 1

From Section 2.1, the probability at any iteration of the outer **for** loop of successful filtering of an $(\beta\varepsilon)$-fraction of $V$ is at least $1/2$. We can model the success of the iterations as a sequence of dependent Bernoulli random variables, with success probability that depends on the results of previous trials but is always at least $1/2$.

**Success Probability of LinearSeq.** If there are at least $m = \lceil \log_{1-\beta\varepsilon}(1/n) \rceil$ successful iterations, the algorithm LinearSeq will succeed. The number of successful iterations $X_\ell$ up to and including the $\ell$-th iteration is a sum of dependent Bernoulli random variables. With some work (Lemma 6 in Appendix A), the Chernoff bounds can be applied to ensure the algorithm succeeds with probability at least $1 - 1/n$, as shown in Appendix C.2.

**Adaptivity and Query Complexity.** Oracle queries are made on Lines 6 and 13 of LinearSeq. The filtering on Line 6 is in one adaptive round, and the inner **for** loop is also in one adaptive round. Thus, the adaptivity is proportional to the number of iterations of the outer **for** loop, $O(\ell) = O\left(\log(n)/\varepsilon^3\right)$. For the query complexity, let $Y_i$ be the number of iterations between the $(i-1)$-th and $i$-th successful iterations of the outer **for** loop. By Lemma 6 in Appendix A, $\mathbb{E}[Y_i] \leq 2$. From here, we show in Appendix C.8 that there are at most $O\left(n/\varepsilon^3\right)$ queries in expectation.

**Approximation Ratio.** Suppose LinearSeq terminates successfully. We have the approximation ratio as follows:

$$f(A') \overset{(a)}{\geq} f(A) - f(A \backslash A') \overset{(b)}{\geq} f(A) - \frac{1+\varepsilon}{2(1-\varepsilon+\varepsilon^2)} f(A) \overset{(c)}{\geq} \frac{1}{4 + \frac{4(2-\varepsilon)}{(1-\varepsilon)(1-2\varepsilon)} \cdot \varepsilon} f(O),$$

where Inequality (a) is from submodularity of $f$, Inequality (b) is from Lemma 2, and Inequality (c) is from Lemma 1.

## 3 Improved Thresholding Procedure and the Main Algorithm

In this section, we describe how to obtain the nearly optimal ratio in nearly optimal query and adaptive complexities (Section 3.2). First, in Section 3.1, we describe ThresholdSeq, a parallelizable procedure to add all elements with gain above a constant threshold to the solution. In Section 3.2, we describe ParallelGreedyBoost and finally the main algorithm LS+PGB. Because of space constraints, the algorithms are described in the main text at a high level only, with detailed descriptions and all proofs deferred to Appendices D and E.

### 3.1 The THRESHOLDSEQ Procedure

In this section, we discuss a the algorithm THRESHOLDSEQ, which adds all elements with gain above an input threshold $\tau$ up to accuracy $\varepsilon$ in $O(\log n)$ adaptive rounds and $O(n)$ queries in expectation. Pseudocode is given in Alg. 3 in Appendix D.

**Overview.** The goal of this algorithm is, given an input threshold $\tau$ and size constraint $k$, to produce a set of size at most $k$ such that the average gain of elements added is at least $\tau$. As discussed in Section 1, this task is an important subroutine of many algorithms for submodular optimization (including our final algorithm), although by itself it does not produce any approximation ratio for SM. As a starting point, consider an algorithm that takes one pass through the ground set, adding each element $e$ to candidate set $A$ iff

$$\Delta\left(e \mid A\right) \geq \tau \text{ and } |A \cup \{e\}| \leq k. \tag{4}$$

Clearly, such an algorithm requires at most $n$ queries and meets the desired requirements; unfortunately, the adaptivity of this approach is also $n$ rounds. Therefore, the challenge is to parallelize this algorithm.

Observe that Conditions 1 and 4 share a common form, although the threshold $\tau$ in Condition 4 is fixed, while the threshold in Condition 1 is the variable $f(A)/k$ which depends on $A$. Therefore, the overall strategy of our parallelizable algorithm THRESHOLDSEQ is analagous to that of LINEARSEQ, although THRESHOLDSEQ is considerably simpler to analyze. The detailed algorithm description and analysis are deferred to Appendix D. The following theorem summarizes the theoretical guarantees of THRESHOLDSEQ and is proven in Appendix D.

**Theorem 2.** *Suppose* THRESHOLDSEQ *is run with input* $(f, k, \varepsilon, \delta, \tau)$*. Then, the algorithm has adaptive complexity* $O(\log(n/\delta)/\varepsilon)$ *and outputs* $A \subseteq \mathcal{N}$ *with* $|A| \leq k$ *such that the following properties hold: 1) The algorithm succeeds with probability at least* $1 - \delta/n$*. 2) There are* $O(n/\varepsilon)$ *oracle queries in expectation. 3) It holds that* $f(A)/|A| \geq (1-\varepsilon)\tau/(1+\varepsilon)$*. 4) If* $|A| < k$*, then* $\Delta\left(x \mid A\right) < \tau$ *for all* $x \in \mathcal{N}$*.*

### 3.2 PARALLELGREEDYBOOST and the Main Algorithm

In this section, we describe the greedy algorithm PARALLELGREEDYBOOST (PGB, Alg. 4) that uses multiple calls to THRESHOLDSEQ with descending thresholds. Next, our state-of-the-art algorithm LS+PGB is specified.

**Description of PARALLELGREEDYBOOST.** This procedure takes as input the results from running an $\alpha$-approximation algorithm on the instance $(f, k)$ of SM; thus, PARALLELGREEDYBOOST is not meant to be used as a standalone algorithm. Namely, PARALLELGREEDYBOOST takes as input $\Gamma$, the solution value of an $\alpha$-approximation algorithm for SM; this solution value $\Gamma$ is then boosted to ensure the ratio $1 - 1/e - \varepsilon$ on the instance. The values of $\Gamma$ and $\alpha$ are used to produce an initial threshold value $\tau$ for THRESHOLDSEQ. Then, the threshold value is iteratively decreased by a factor of $(1-\varepsilon)$ and the call to THRESHOLDSEQ is iteratively repeated to build up a solution, until a minimum value for the threshold of $\Gamma/(3k)$ is reached. Therefore, THRESHOLDSEQ is called at most $O\left(\log(1/\alpha)/\varepsilon\right)$ times. We remark that $\alpha$ is not required to be a constant approximation ratio.

Pseudocode specifying the exact parameter values for the THRESHOLDSEQ calls is provided in Appendix E. The following theorem summarizes the theoretical guarantees of PARALLELGREEDYBOOST and is proven in Appendix E.

**Theorem 3.** *Let* $(f, k)$ *be an instance of* SM*. Suppose an* $\alpha$*- approximation algorithm for* SM *is used to obtain* $\Gamma, \alpha$*, where the approximation ratio* $\alpha$ *holds with probability* $1 - p_\alpha$*. For any constant* $\varepsilon > 0$*, the algorithm* PARALLELGREEDYBOOST *has adaptive complexity* $O\left(\frac{\log \alpha^{-1}}{\varepsilon^2} \log\left(\frac{n \log\left(\alpha^{-1}\right)}{\varepsilon}\right)\right)$ *and outputs* $A \in \mathcal{N}$ *with* $|A| \leq k$ *such that the following properties hold: 1) The algorithm succeeds with probability at least* $1 - 1/n - p_\alpha$*. 2) If the algorithm succeeds, there are* $O\left(n \log\left(\alpha^{-1}\right)/\varepsilon^2\right)$ *oracle queries in expectation. 3) If the algorithm succeeds,* $f(A) \geq (1 - 1/e - \varepsilon) f(O)$*, where* $O$ *is an optimal solution to the instance* $(f, k)$*.*

**Main Algorithm: LS+PGB.** To obtain the main algorithm of this paper (and its nearly optimal theoretical guarantees), we use PARALLELGREEDYBOOST with the solution value and ratio of LS for $\Gamma$ and $\alpha$, respectively. Because this choice requires an initial run of LS, we denote this algorithm by LS+PGB.

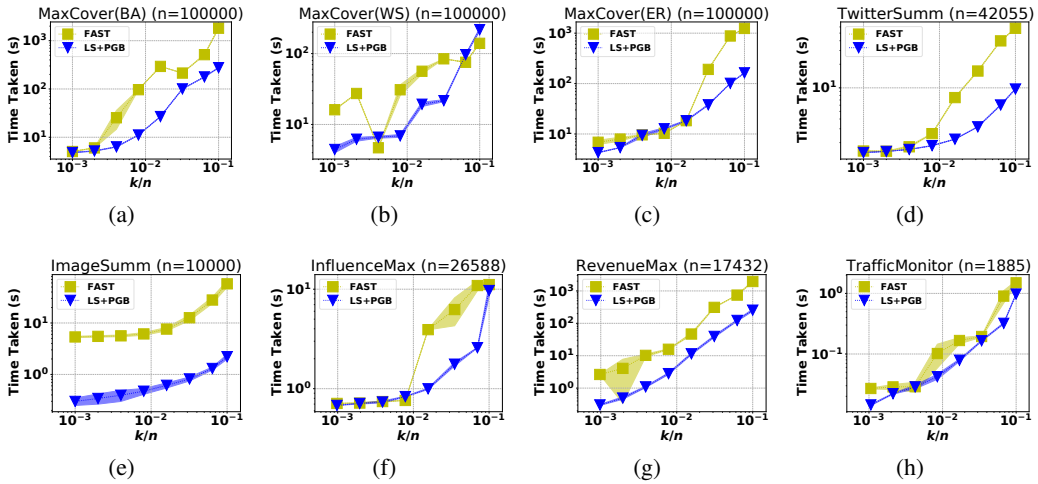


Figure 2: Runtime (s) vs. $k/n$. Both axes are log-scaled.

## 4 Empirical Evaluation

In this section, we demonstrate that the empirical performance of LS+PGB outperforms that of FAST for the metrics of total time, total queries, and objective value across six applications of SM: maximum cover on random graphs (MaxCover), twitter feed summarization (TweetSumm), image summarization (ImageSumm), influence maximization (Influence), revenue maximization (RevMax), and Traffic Speeding Sensor Placement (Traffic). See Appendix H.2 for the definition of the objectives. The sizes $n$ of the ground sets range from $n = 1885$ to $100000$.

**Implementation and Environment.** We evaluate the same implementation of FAST used in Breuer et al. [8]. Our implementation of LS+PGB is parallelized using MPI within the same Python codebase as FAST (see the Supplementary Material for source code). Practical optimizations to LINEARSEQ are made, which do not compromise the theoretical guarantees, which are discussed in Appendix G. The algorithms are evaluated using up to 75 threads on 40 processor cores. The time is measured using a rigorous notion of parallel (wall clock) runtime; we also report the objective value and total queries to the submodular function. On each instance, the algorithms are repeated independently for five repetitions, and the mean and standard deviation of each metric is plotted for each instance.

**Parameters.** The parameters $\varepsilon, \delta$ of FAST are set to enforce the nominal ratio of $1-1/e-0.1 \approx 0.53$ with probability $0.975$; these are the same parameter settings for FAST as in the Breuer et al. [8] evaluation. The $\varepsilon$ parameter of LS+PGB is set to enforce the same ratio with probability $1-2/n$. With these parameters, FAST ensures its ratio only if $k \geq \theta(\varepsilon, \delta, k) = 2\log(2\delta^{-1}\log(\frac{1}{\varepsilon}\log(k)))/\varepsilon^2(1-5\varepsilon) \geq 22500$. Since $k < 22500$ on all of our instances, FAST is evaluated in our experiments (and also in Breuer et al. [8]) as a theoretically motivated heuristic. In contrast, the ratio of LS+PGB holds on all instances evaluated. We use exponentially increasing $k$ values from $n/1000$ to $n/10$ for each application to explore the behavior of each algorithm across a broad range of instance sizes.

**Overview of Results.** In Fig. 2, the mean runtimes of FAST and LS+PGB with 75 threads available are shown on all evaluated instances. On many instances, LS+PGB is faster by more than an order of magnitude. Both algorithms exhibit linear scaling of runtime with $k$ – this is because the time required to evaluate the objective $f(S)$ typically scales linearly with $|S|$. Mean objective values and total queries are shown in Figs. 3,4, respectively. The improvement of LS+PGB over FAST is summarized in Table 2. Finally, FAST and LS+PGB exhibit very similar scaling behavior with the number of processor threads employed: as shown in Fig. 5, each algorithm exhibits nearly linear speedup with the number of threads.

# References


[1] CalTrans. Pems: California performance measuring system. http://pems.dot.ca.gov/. [Online; accessed 1-May-2018].

[2] Ashwinkumar Badanidiyuru and Jan Vondrák. Fast algorithms for maximizing submodular functions. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2014.

[3] Eric Balkanski and Yaron Singer. The adaptive complexity of maximizing a submodular function. In *ACM SIGACT Symposium on Theory of Computing (STOC)*, 2018.

[4] Eric Balkanski, Aviad Rubinstein, and Yaron Singer. An Exponential Speedup in Parallel Running Time for Submodular Maximization without Loss in Approximation. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.

[5] Eric Balkanski, Aviad Rubinstein, and Yaron Singer. An optimal approximation for submodular maximization under a matroid constraint in the adaptive complexity model. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 66–77, nov 2019.

[6] Rafael Barbosa, Alina Ene, Huy Le Nguyen, and Justin Ward. The Power of Randomization: Distributed Submodular Maximization on Massive Datasets. In *International Conference on Machine Learning (ICML)*, 2015.

[7] Rafael Da Ponte Barbosa, Alina Ene, Huy L. Nguyen, and Justin Ward. A New Framework for Distributed Submodular Maximization. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2016.

[8] Adam Breuer, Eric Balkanski, and Yaron Singer. The FAST Algorithm for Submodular Maximization. In *International Conference on Machine Learning (ICML)*, 2019.

[9] Gruia Calinescu, Chandra Chekuri, Martin Pál, and Jan Vondrák. Maximizing a Submodular Set Function subject to a Matroid Constraint. In *Integer Programming and Combinatorial Optimization (IPCO)*, pages 182–196, 2007.

[10] Chandra Chekuri and Kent Quanrud. Submodular function maximization in parallel via the multilinear relaxation. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 303–322, jul 2019.

[11] Michele Conforti and Gérard Cornuéjols. Submodular set functions, matroids and the greedy algorithm: Tight worst-case bounds and some generalizations of the Rado-Edmonds theorem. *Discrete Applied Mathematics*, 7(3):251–274, 1984.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Alina Ene and Huy L Nguyen. Submodular Maximization with Nearly-optimal Approximation and Adaptivity in Nearly-linear Time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.

[14] Alessandro Epasto, Vahab Mirrokni, and Morteza Zadimoghaddam. Bicriteria Distributed Submodular Maximization in a Few Rounds. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[15] Matthew Fahrbach, Vahab Mirrokni, and Morteza Zadimoghaddam. Submodular Maximization with Nearly Optimal Approximation, Adaptivity, and Query Complexity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 255–273, 2019.

[16] Matthew Fahrbach, Vahab Mirrokni, and Morteza Zadimoghaddam. Non-monotone submodular maximization with nearly optimal adaptivity and query complexity. In *International Conference on Machine Learning*, pages 1833–1842. PMLR, 2019.

[17] Jennifer Gillenwater, Alex Kulesza, and Ben Taskar. Near-Optimal MAP Inference for Determinantal Point Processes. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[18] Thibaut Horel and Yaron Singer. Maximization of Approximately Submodular Functions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[19] Ehsan Kazemi, Marko Mitrovic, Morteza Zadimoghaddam, Silvio Lattanzi, and Amin Karbasi. Submodular Streaming in All its Glory: Tight Approximation, Minimum Memory and Low Adaptive Complexity. In *International Conference on Machine Learning (ICML)*, 2019.

[20] Andreas Krause and Carlos Guestrin. Near-optimal observation selection using submodular functions. *AAAI Conference on Artificial Intelligence*, 2007.

[21] Alan Kuhnle. Quick Streaming Algorithms for Maximization of Monotone Submodular Functions in Linear Time. In *Artificial Intelligence and Statistics (AISTATS)*, 2021.

[22] Alan Kuhnle. Nearly Linear-Time, Parallelizable Algorithms for Non-Monotone Submodular Maximization. In *AAAI Conference on Artificial Intelligence*, 2021.

[23] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective Outbreak Detection in Networks. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2007.

[24] Vahab Mirrokni and Morteza Zadimoghaddam. Randomized Composable Core-Sets for Distributed Submodular Maximization. In *ACM Symposium on Theory of Computing (STOC)*, 2015.

[25] Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, and Amin Karbasi. Fast constrained submodular maximization: Personalized data summarization. In *International Conference on Machine Learning*, pages 1358–1367. PMLR, 2016.

[26] Baharan Mirzasoleiman, Stefanie Jegelka, and Andreas Krause. Streaming Non-Monotone Submodular Maximization: Personalized Video Summarization on the Fly. In *AAAI Conference on Artificial Intelligence*, 2018.

[27] G L Nemhauser and L A Wolsey. Best Algorithms for Approximating the Maximum of a Submodular Set Function. *Mathematics of Operations Research*, 3(3):177–188, 1978.

[28] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

# Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [No]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] Complete proofs are provided in the Supplementary Material.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]