The FAST Algorithm for Submodular Maximization

Adam Breuer [1] **Eric Balkanski** [1] **Yaron Singer** [1]

Abstract

In this paper we describe a new parallel algorithm called Fast Adaptive Sequencing Technique (FAST) for maximizing a monotone submodular function under a cardinality constraint k. This algorithm achieves the optimal $1 - 1/e$ approximation guarantee and is orders of magnitude faster than the state-of-the-art on a variety of experiments over real-world data sets. Following recent work by Balkanski & Singer (2018a), there has been a great deal of research on algorithms whose theoretical parallel runtime is exponentially faster than algorithms used for submodular maximization over the past 40 years. However, while these new algorithms are fast in terms of asymptotic worst-case guarantees, it is computationally infeasible to use them in practice even on small data sets because the number of rounds and queries they require depend on large constants and high-degree polynomials in terms of precision and confidence. The design principles behind the FAST algorithm we present here are a significant departure from those of recent theoretically fast algorithms. Rather than optimize for asymptotic theoretical guarantees, the design of FAST introduces several new techniques that achieve remarkable practical and theoretical parallel runtimes. The approximation guarantee obtained by FAST is arbitrarily close to $1 - 1/e$, and its asymptotic parallel runtime (adaptivity) is $\mathcal{O}(\log(n)\log^2(\log k))$ using $\mathcal{O}(n \log\log(k))$ total queries. We show that FAST is orders of magnitude faster than any algorithm for submodular maximization we are aware of, including hyper-optimized parallel versions of state-of-the-art serial algorithms, by running experiments on large data sets.

[1]Harvard University, Cambridge, MA. Correspondence to: Adam Breuer <breuer@g.harvard.edu>, Eric Balkanski <ebalkans@gmail.com>, Yaron Singer <yaron@seas.harvard.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

1. Introduction

In this paper we describe a fast parallel algorithm for submodular maximization.[1] Informally, a function is submodular if it exhibits a natural diminishing returns property. For the canonical problem of maximizing a monotone submodular function under a cardinality constraint, it is well known that the greedy algorithm, which iteratively adds elements whose marginal contribution is largest to the solution, obtains a $1 - 1/e$ approximation guarantee (Nemhauser et al., 1978) which is optimal for polynomial-time algorithms (Nemhauser & Wolsey, 1978). The greedy algorithm and other submodular maximization techniques are heavily used in machine learning and data mining as many fundamental objectives such as entropy, mutual information, graphs cuts, diversity, and set cover are submodular.

In recent years there has been a great deal of progress on fast algorithms for submodular maximization designed to accelerate computation on large data sets. The first line of work considers *serial* algorithms where queries can be evaluated on a single processor (Leskovec et al., 2007; Badanidiyuru & Vondrák, 2014; Mirzasoleiman et al., 2015; 2016; Ene & Nguyen, 2019b;c). For serial algorithms the state-of-the-art for maximization under a cardinality constraint is the *lazier-than-lazy-greedy* (LTLG) algorithm which returns a solution that is in expectation arbitrarily close to the optimal $1-1/e$ and does so in a linear number of queries (Mirzasoleiman et al., 2015). This algorithm is a stochastic greedy algorithm coupled with lazy updates, which not only performs well in terms of the quality of the solution it returns, but is also very fast in practice.

Accelerating computation beyond linear runtime requires *parallelization*. The parallel runtime of blackbox optimization is measured by *adaptivity*, which is the number of sequential rounds an algorithm requires when polynomially-many queries can be executed in parallel in every round. For maximizing a submodular function defined over a ground set of n elements under a cardinality constraint k, the adaptivity of the naive greedy algorithm is $\mathcal{O}(k)$, which in the worst case is $\mathcal{O}(n)$. Until recently no algorithm was known to have better parallel runtime than that of greedy.

A very recent line of work initiated by Balkanski &

[1]Code is available from www.adambreuer.com/code.

Algorithm	*rounds*	*queries*	*time (sec)*
AMORTIZED-FILTERING (Balkanski et al., 2019a)	961	2124351	20.29
BINARY-SEARCH-MAXIMIZATION (Fahrbach et al., 2019a)	8744	2552028	24.64
RANDOMIZED-PARALLEL-GREEDY (Chekuri & Quanrud, 2019b)	92	148642	4.11
PARALLEL-LTLG (Mirzasoleiman et al., 2015)	200	856	0.15
FAST	**9**	**1598**	**0.033**

Singer (2018a) develops techniques for designing constant approximation algorithms for submodular maximization whose parallel runtime is *logarithmic* (Balkanski & Singer, 2018b; Balkanski et al., 2018; Ene & Nguyen, 2019a; Fahrbach et al., 2019a;b; Kazemi et al., 2019; Chekuri & Quanrud, 2019a;b; Balkanski et al., 2019a;b; Ene et al., 2019; Chen et al., 2019; Esfandiari et al., 2019; Qian & Singer, 2019). In particular, Balkanski & Singer (2018a) describe a technique called *adaptive sampling* that obtains in $\mathcal{O}(\log n)$ rounds a $1/3$ approximation for maximizing a monotone submodular function under a cardinality constraint. This technique can be used to obtain an approximation arbitrarily close to the optimal $1-1/e$ in $\mathcal{O}(\log n)$ rounds (Balkanski et al., 2019a; Ene & Nguyen, 2019a).

1.1. From theory to practice

The focus of the work on adaptivity described above has largely been on conceptual and theoretical contributions: achieving strong approximation guarantees under various constraints with runtimes that are exponentially faster under worst case theoretical analysis. From a practitioner's perspective however, even the state-of-the-art algorithms in this genre are infeasible for large data sets. The logarithmic parallel runtime of these algorithms carries extremely large constants and polynomial dependencies on precision and confidence parameters that are hidden in their asymptotic analysis. In terms of sample complexity alone, obtaining (for example) a $1-1/e-0.1$ approximation with 95% confidence for maximizing a submodular function under cardinality constraint k requires evaluating at least 10^8 (Balkanski et al., 2019a) or 10^6 (Fahrbach et al., 2019a; Chekuri & Quanrud, 2019b) samples of sets of size approximately $\frac{k}{\log n}$ in every round. Even if one heuristically uses a single sample in every round, other sources of inefficiencies that we discuss throughout the paper prevent these algorithms from being applied even on moderate-sized data sets. The question is then whether the plethora of breakthrough techniques in this line of work of exponentially faster algorithms for submodular maximization can lead to algorithms that are fast in practice for large problem instances.

1.2. Our contribution

In this paper we design a new algorithm called Fast Adaptive Sequencing Technique (FAST) for maximizing a monotone submodular function under a cardinality constraint k. FAST has an approximation ratio that is arbitrarily close to $1-1/e$, is $\mathcal{O}(\log(n)\log^2(\log k))$ adaptive, and uses a total of $\mathcal{O}(n\log\log(k))$ queries. The main contribution is not in the algorithm's asymptotic guarantees, but in its design that is extremely efficient both in terms of its non-asymptotic worst case query complexity and number of rounds, and in terms of its practical runtime. In terms of *actual* query complexity and practical runtime, this algorithm outperforms all algorithms for submodular maximization we are aware of, including hyper-optimized versions of LTLG. To be more concrete, we give a brief experimental comparison in the table above for a movie recommendation objective on $n = 500$ movies against optimized implementations of algorithms with the same adaptivity and approximation (experiment details in Section 4).[2]

FAST achieves its speedup by thoughtful design that results in frugal worst case query complexity as well as several heuristics used for practical speedups. From a purely analytical perspective, FAST improves the ε dependency in the linear term of the query complexity of at least $\tilde{\mathcal{O}}(\varepsilon^{-5}n)$ in Balkanski et al. (2019a) and Ene & Nguyen (2019a) and $\tilde{\mathcal{O}}(\varepsilon^{-3}n)$ in Fahrbach et al. (2019a) to $\tilde{\mathcal{O}}(\varepsilon^{-2}n)$. We provide the first non-asymptotic bounds on the query and adaptive complexity of an algorithm with sublinear adaptivity, showing dependency on small constants. Our algorithm uses adaptive sequencing (Balkanski et al., 2019b) and multiple optimizations to improve the query complexity and runtime.

1.3. Paper organization

We introduce the main ideas and decisions behind the design of FAST in Section 2. We describe and analyze guarantees in Section 3. We discuss experiments in Section 4.

[2] To obtain these values, we set all algorithms to guarantee a $1-1/e-0.1$ approximation with probability 0.95 except LTLG, which has this guarantee in expectation, and we set $k = 200$.

2. FAST Overview

Before describing the algorithm, we give an overview of the major ideas and discuss how they circumvent the bottlenecks for practical implementation of existing logarithmic adaptivity algorithms.

Adaptive sequencing vs. adaptive sampling. The large majority of low-adaptivity algorithms use *adaptive sampling* (Balkanski & Singer, 2018a; Ene & Nguyen, 2019a; Fahrbach et al., 2019a;b; Balkanski et al., 2018; 2019a; Kazemi et al., 2019), a technique introduced in Balkanski & Singer (2018a). These algorithms sample a large number of sets of elements at every iteration to estimate (1) the expected marginal contribution of a random set R to the current solution S and (2) the expected marginal contributions of each element a to $R \cup S$. These estimates, which rely on concentration arguments, are then used to either add a random set R to S or discard elements with low expected marginal contribution to $R \cup S$.

In contrast, the *adaptive sequencing* technique which was recently introduced in Balkanski et al. (2019b) generates at every iteration a *single* random sequence $(a_1, \ldots, a_{|X|})$ of the elements X not yet discarded. A prefix $A_{i^\star} = (a_1, \ldots, a_{i^\star})$ of the sequence is then added to the solution S, where $i^\star$ is the largest position i such that a large fraction of the elements in X have high marginal contribution to $S \cup A_{i-1}$. Elements with low marginal contribution to the new solution S are then discarded from X.

The first choice we made was to use an adaptive sequencing technique rather than adaptive sampling.

- **Dependence on large polynomials in ε.** Adaptive sampling algorithms crucially rely on sampling, and as a result their query complexity has high polynomial dependency on ε (e.g. at least $\mathcal{O}(\varepsilon^{-5}n)$ in Balkanski et al. (2019a) and Ene & Nguyen (2019a)). Due to these ε dependencies, the query complexity blows up with any reasonable value for ε. In contrast, adaptive *sequencing* generates a *single* random sequence at every iteration. Therefore, in the term that is linear in n we can obtain an ε dependence that is only $\tilde{\mathcal{O}}(\varepsilon^{-2})$.
- **Dependence on large constants.** The asymptotic query complexity of previous algorithms depends on very large constants (e.g. at least 60000 in Balkanski et al. (2019a) and Ene & Nguyen (2019a)) making them impractical. As we tried to optimize constants for adaptive sampling, we found that due to the sampling and the requirement to maintain strong theoretical guarantees, the constants cascade and grow through multiple parts of the analysis. In principle, adaptive *sequencing* does not rely on sampling, which dramatically reduces its dependency on constants.

Negotiating the adaptive complexity with the query complexity. The vanilla version of our algorithm, whose description and analysis are in Appendix A, has at most $\varepsilon^{-2}\log n$ adaptive rounds and uses a total of $\varepsilon^{-2}nk$ queries to obtain a $1-1/e-\frac{3}{2}\varepsilon$ approximation, without additional dependence on constants or lower order terms. In our actual algorithm, we trade a small factor in adaptive complexity for a substantial improvement in query complexity. We do this in the following manner:

- **Search for estimates of `OPT`.** All algorithms with logarithmic adaptivity require a good estimate of `OPT`, which can be obtained by running $\varepsilon^{-1}\log k$ instances of the algorithms with different guesses of `OPT` in parallel, so that one guess is guaranteed to be a good approximation to `OPT`.[3] We accelerate this search by binary searching over the guesses of `OPT`. A main difficulty when using this binary search is that the approximation guarantee of the solution obtained with each guess of `OPT` needs to hold with high probability, instead of in expectation, to obtain any guarantee for the global solution.

  Even though the guarantees on the marginal contributions obtained from each element added to the solution only hold in expectation for adaptive sequencing, we obtain high probability guarantees for the global solution by generalizing the robust guarantees obtained in Hassidim & Singer (2017) so that they also apply to adaptive sequencing. In the practical speedups below, we discuss how we often only need a single iteration of this binary search in practice;

- **Search for position $i^\star$.** To find the position $i^\star$, which is the largest position $i \in [k]$ in the sequence such that a large fraction of not-yet-discarded elements have high marginal contribution to $S \cup A_{i-1}$, the vanilla adaptive sequencing technique queries the marginal contribution of all elements in X at each of the k positions. This search for $i^\star$ causes the $\mathcal{O}(nk)$ query complexity.

  Instead, similarly to the search of `OPT`, we binary search over a set of $\varepsilon^{-1}\log k$ geometrically increasing values i that correspond to guesses of $i^\star$. This improves the $\mathcal{O}(nk)$ dependency on n and k in the query complexity to $\mathcal{O}(n\log(\log k))$. Then, at any step of the binary search over a position i, instead of evaluating the marginal contribution of all elements in X to $S \cup A_{i-1}$, we only evaluate a small sample of elements. In the practical speedups below, we discuss how we can often skip this binary search for $i^\star$ in practice.

[3] Fahrbach et al. (2019a) do some preprocessing to estimate `OPT`, but it is estimated within some very large constant.

Practical speedups. We include several ideas which result in considerable speedups in practice without sacrificing approximation, adaptivity, or query complexity guarantees:

- **Preprocessing the sequence.** At the outset of each iteration of the algorithm, before searching for a prefix $A_{i^\star}$ to add to the solution S, we first use a preprocessing step that adds high value elements from the sequence to S. Specifically, we add to the solution S all sequence elements a_i that have high marginal contribution to $S \cup A_{i-1}$.

  After adding these high-value elements, we discard surviving elements in X that have low marginal contribution to the new solution S. In the case where this step discards a large fraction of surviving elements from X, we can also skip this iteration's binary search for $i^\star$ and continue to the next iteration without adding a prefix to S;

- **Number of elements added per iteration.** An adaptive sampling algorithm which samples sets of size s adds at most s elements to the current solution at each iteration. In contrast, adaptive sequencing and the preprocessing step described above often allow our algorithm to add a very large number of elements to the current solution at each iteration in practice;

- **Single iteration of the binary search for `OPT`.** Even with binary search, running multiple instances of the algorithm with different guesses of OPT is undesirable. We describe a technique that often needs only a single guess of OPT. This guess is the sum $v = \max_{|S|\leq k} \sum_{a\in S} f(a)$ of the k highest valued singletons, which is an upper bound on OPT. If the solution S obtained with that guess v has value $f(S) \geq (1-1/e)v$, then, since $v \geq \texttt{OPT}$, S is guaranteed to obtain a $1-1/e$ approximation and the algorithm does not need to continue the binary search. Note that with a single guess of OPT, the robust guarantees for the binary search are not needed, which improves the sample complexity to $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)} \log(2\delta^{-1})$;

- **Lazy updates.** There are many situations where lazy evaluations of marginal contributions can be performed (Minoux, 1978; Mirzasoleiman et al., 2015). Since we never discard elements from the solution S, the marginal contributions of elements a to S are non-increasing at every iteration by submodularity. Elements with low marginal contribution c to the current solution at some iteration are ignored until the threshold t is lowered to $t \leq c$. Lazy updates also accelerate the binary search over $i^\star$.

3. The Algorithm

We describe the FAST-FULL algorithm (Algorithm 1). The main part of the algorithm is the FAST subroutine (Algorithm 2), which is instantiated with different guesses of OPT. These guesses $v \in V$ of OPT are geometrically increasing from $\max_{a\in N} f(a)$ to $\max_{|S|\leq k} \sum_{a\in S} f(a)$ by a $(1-\varepsilon)^{-1}$ factor, so V contains a value that is a $1-\varepsilon$ approximation to OPT. The algorithm binary searches over guesses for the largest guess v that obtains a solution S that is a $1-1/e$ approximation to v.

Algorithm 1 FAST-FULL: the full algorithm

input function f, cardinality constraint k, parameter ε
  $V \leftarrow \text{GEOM}(\max_a f(a), \max_{|S|\leq k} \sum_{a\in S} f(a), 1-\varepsilon)$
  $v^\star \leftarrow \text{B-SEARCH}(\max\{v \in V : f(S_v) \geq (1-1/e)v\})$
    where $S_v \leftarrow \text{FAST}(v)$
  return $S_{v^\star}$

FAST generates at every iteration a uniformly random sequence $a_1, \ldots, a_{|X|}$ of the elements X not yet discarded. After the preprocessing step which adds to S elements guaranteed to have high marginal contribution, the algorithm identifies a position $i^\star$ in this sequence which determines the prefix $A_{i^\star-1}$ that is added to the current solution S. Position $i^\star$ is defined as the largest position such that there is a large fraction of elements in X with high contribution to $S \cup A_{i^\star-1}$. To find $i^\star$, we binary search over geometrically increasing positions $i \in I \subseteq [k]$. At each position i, we only evaluate the contributions of elements $a \in R$, where R is a uniformly random subset of X of size m, instead of all elements X.

Algorithm 2 FAST: the Fast Adaptive Sequencing Technique algorithm

input f, constraint k, guess v for OPT, parameter ε
  $S \leftarrow \emptyset$
  while $|S| < k$ and number of iterations $< \varepsilon^{-1}$ **do**
    $X \leftarrow N, t \leftarrow (1-\varepsilon)(v - f(S))/k$
    while $X \neq \emptyset$ and $|S| < k$ **do**
      $a_1, \ldots, a_{|X|} \leftarrow \text{SEQUENCE}(X, |X|)$
      $A_i \leftarrow a_1, \ldots, a_i$
      $S \leftarrow S \cup \{a_i : f_{S\cup A_{i-1}}(a_i) \geq t\}$
      $X_0 \leftarrow \{a \in X : f_S(a) \geq t\}$
      if $|X_0| \leq (1-\varepsilon)|X|$ **then**
        $X \leftarrow X_0$ and continue to next iteration
      $R \leftarrow \text{SAMPLE}(X, m)$,
      $I \leftarrow \text{GEOM}(1, k-|S|, 1-\varepsilon)$
      $R_i \leftarrow \left\{a \in R : f_{S\cup A_{i-1}}(a) \geq t\right\}$, for $i \in I$
      $i^\star \leftarrow \text{B-SEARCH}(\max\{i : |R_i| \geq (1-2\varepsilon)|R|\})$
      $S \leftarrow S \cup A_{i^\star}$
  return S

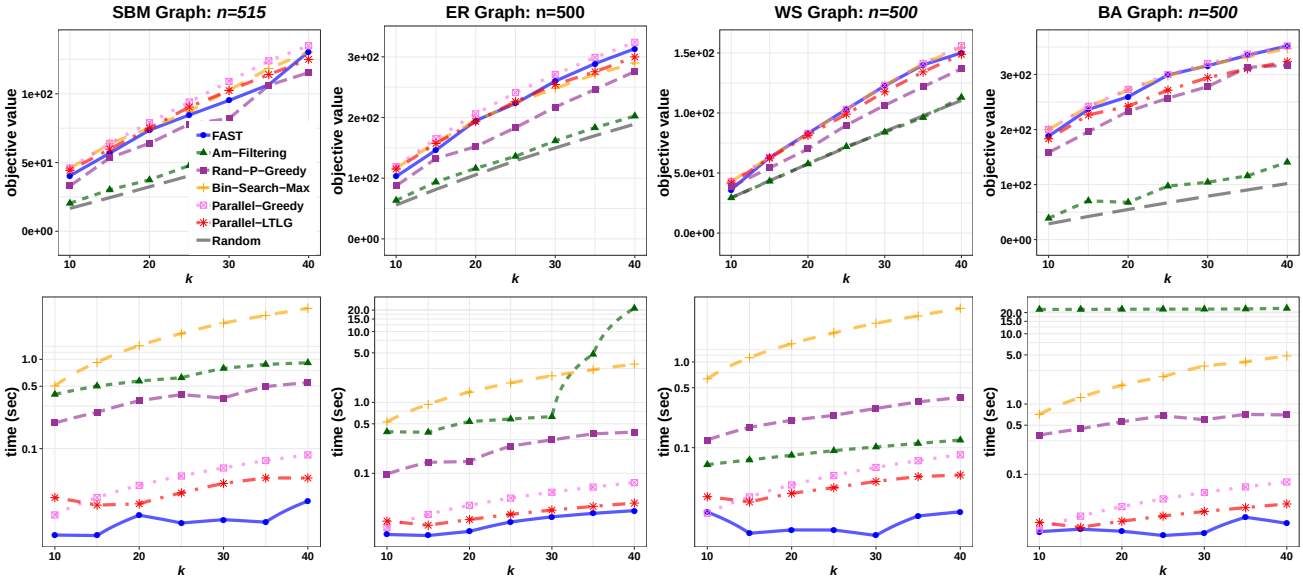

Figure 1. Experiment Set 1.a: FAST *(blue) vs. low-adaptivity algorithms and* PARALLEL-LTLG *on graphs (time axis* ***log-scaled****).*

3.1. Analysis

We show that FAST obtains a $1 - 1/e - \varepsilon$ approximation w.p. $1-\delta$ and that it has $\tilde{O}(\varepsilon^{-2} \log n)$ adaptive complexity and $\tilde{O}(\varepsilon^{-2}n + \varepsilon^{-4} \log(n) \log(\delta^{-1}))$ query complexity.

Theorem 1. *Assume* $k \geq \frac{2\log(2\delta^{-1}\ell)}{\varepsilon^2(1-5\varepsilon)}$ *and* $\varepsilon \in (0, 0.1)$, *where* $\ell = \log(\frac{\log k}{\varepsilon})$. *Then,* FAST *with sample complexity* $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)} \log(\frac{4\ell \log n}{\delta\varepsilon^2})$ *has at most* $\varepsilon^{-2} \log(n)\ell^2$ *adaptive rounds,* $2\varepsilon^{-2}\ell n + \varepsilon^{-4} \log(n)\ell^2 m$ *queries, and achieves a* $1 - \frac{1}{e} - 4\varepsilon$ *approximation with probability* $1 - \delta$.

We defer the analysis to Appendix B. The main part of it is for the approximation guarantee, which consists of two cases depending on the condition which breaks the outer-loop. Lemma 3 shows that when there are ε^{-1} iterations of the outer-loop, the set of elements added to S at every iteration of the outer-loop contributes $\varepsilon^{-1}(\texttt{OPT} - f(S))$. Lemma 5 shows that for the case where $|S| = k$, the expected contribution of each element a_i added to S is arbitrarily close to $(\texttt{OPT} - f(S))/k$. For each solution S_v, we need the approximation guarantee to hold with high probability instead of in expectation to be able to binary search over guesses for $\texttt{OPT}$, which we obtain in Lemma 7 by generalizing the robust guarantees of Hassidim & Singer (2017) in Lemma 6. The main observation to obtain the adaptive complexity (Lemma 6) is that, by definition of $i^\star$, at least an ε fraction of the surviving elements in X are discarded at every iteration with high probability.[4] For the query complexity (Lemma 7), we note that there are $|X| + m\ell$ function evaluations per iteration.

4. Experiments

Our goal in this section is to show that in practice, FAST finds solutions whose value meets or exceeds alternatives in less parallel runtime than both state-of-the-art low-adaptivity algorithms and LAZIER-THAN-LAZY-GREEDY (LTLG). To accomplish this, we build optimized parallel MPI implementations of FAST, other low-adaptivity algorithms, and LTLG, which is widely regarded as the fastest algorithm for submodular maximization in practice. We then use 95 Intel Skylake-SP 3.1 GHz processors on AWS to compare the algorithms' runtime over a variety of objectives defined on 8 real and synthetic datasets. We measure runtime using a rigorous measure of parallel time (see Appendix C.8). Appendices C.1, C.4, C.9, and C.6 contain detailed descriptions of the benchmarks, objectives, implementations, hardware, and experimental setup on AWS.[5]

We conduct two sets of experiments. The first set compares FAST to previous low-adaptivity algorithms on 8 objectives. Since previous algorithms all have practically intractable sample complexity, we grossly reduce their sample complexity to only 95 samples per iteration so that each processor performs a single function evaluation per iteration. This reduction, which we discuss in detail below, gives these algorithms a large runtime advantage over FAST, which computes its full theoretical sample complexity in all experiments. This is practically feasible for FAST

[4] To obtain the adaptivity r with probability 1 and the approximation guarantee w.p. $1-\delta$, the algorithm declares failure after r rounds and accounts for this failure probability in δ.

[5] Code is available from www.adambreuer.com/code.

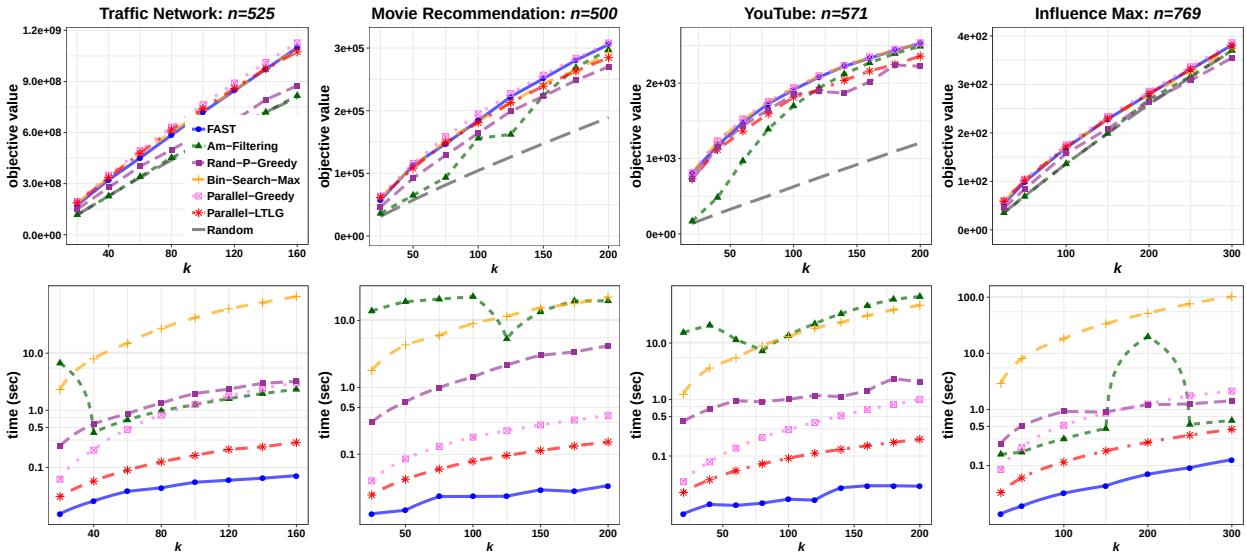

Figure 2. Experiment Set 1.b: FAST *(blue) vs. low-adaptivity algorithms and* PARALLEL-LTLG *on real data (time axis* ***log-scaled****).*

because FAST samples *elements*, not *sets* of elements like previous algorithms. Despite the large advantage this setup gives to the previous low-adaptivity algorithms, FAST is consistently one to three orders of magnitude faster.

The second set of experiments compares FAST to PARALLEL-LAZIER-THAN-LAZY-GREEDY (PARALLEL-LTLG) on large-scale data sets. We scale up the 8 objectives to be defined on synthetic data with $n = 100000$ and real data with up to $n = 26000$ and various k ranging from $k = 25$ to $k = 25000$. We find that FAST is consistently 1.5 to 27 times faster than PARALLEL-LTLG, and its runtime advantage increases in k. These fast relative runtimes are a loose lower bound on FAST's performance advantage, as FAST can reap additional speedups by adding up to n processors, whereas PARALLEL-LTLG performs at most $n \log(\varepsilon^{-1})/k$ function evaluations per iteration, so using over 95 processors often does not help. In Section 4.1, we show that on many objectives FAST is faster even with only a single processor.

4.1. Experiments set 1: FAST vs. low-adaptivity algorithms

Our first set of experiments compares FAST to state-of-the-art low-adaptivity algorithms. To accomplish this, we built optimized parallel MPI versions of each of the following algorithms: RANDOMIZED-PARALLEL-GREEDY (Chekuri & Quanrud, 2019b), BINARY-SEARCH-MAXIMIZATION (Fahrbach et al., 2019a), and AMORTIZED-FILTERING (Balkanski et al., 2019a). For any given $\varepsilon > 0$ all these algorithms achieve a $1 - 1/e - \varepsilon$ approximation in $\mathcal{O}(\mathrm{poly}(\varepsilon^{-1}) \log n)$ rounds.

We also compare these low-adaptivity algorithms to an optimized parallel MPI implementation of LAZIER-THAN-LAZY-GREEDY (LTLG) (Mirzasoleiman et al., 2015) (see Appendix C.11). LTLG is widely regarded as the fastest algorithm for submodular maximization in practice, and it has a $(1-1/e-\varepsilon)$ approximation guarantee in expectation.

For calibration, we also ran (1) PARALLEL-GREEDY, a parallel version of the standard GREEDY algorithm, as a heuristic upper bound for the objective value, as well as (2) RANDOM, an algorithm that simply selects k elements uniformly at random.

A fair comparison of the low-adaptivity algorithms' parallel runtimes and solution values is to run each algorithm with parameters that yield the same guarantees, for example a $1-1/e-\varepsilon$ approximation w.p. $1-\delta$ with $\varepsilon = 0.1$ and $\delta = 0.05$. However, this is infeasible since the other low-adaptivity algorithms all require a practically intractable number of queries to achieve any reasonable guarantees, e.g. every round of AMORTIZED-FILTERING would require at least 10^8 samples, even with $n = 500$.

Dealing with benchmarks' practically intractable query complexity. To run other low-adaptivity algorithms despite their huge sample complexity we made two major modifications:

1. **Accelerating subroutines.** We optimize each of the three other low-adaptivity benchmarks by implementing parallel binary search to replace brute-force search and several other modifications that reduce unnecessary queries (for a full description of these fast implementations, see Appendix C.10). These optimizations result in

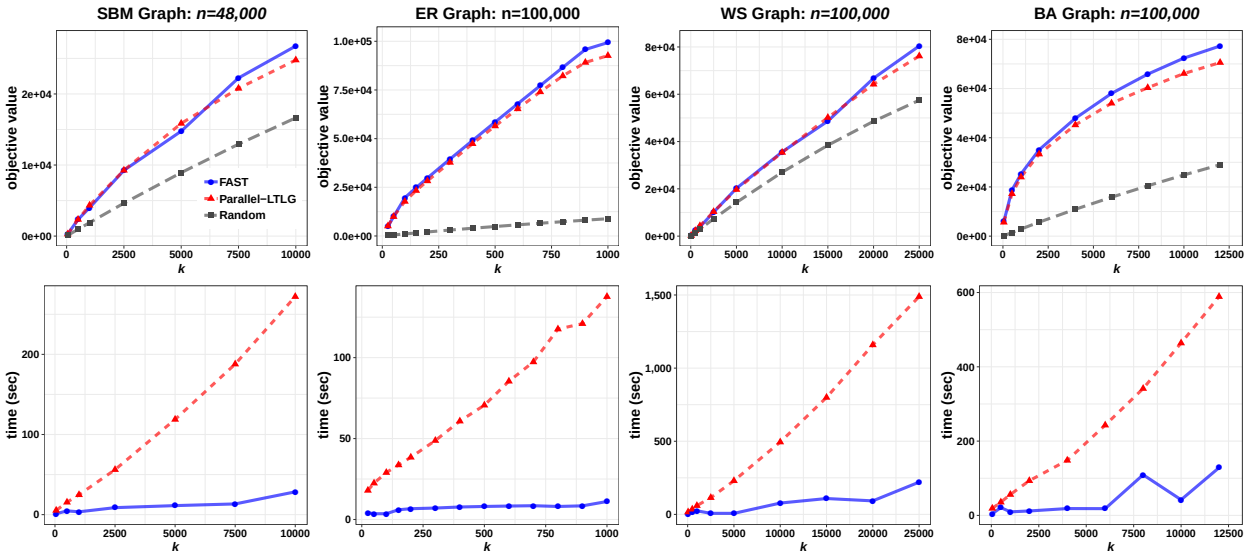

Figure 3. Experiment Set 2.a: FAST *(blue) vs.* PARALLEL-LTLG *(red) on graphs.*

speedups that reduce their runtimes by an order of magnitude in practice, and our implementations are publicly available in our code base. Despite this, it remains practically infeasible to compute these algorithms' high number of samples in practice even on small problems (e.g. $n = 500$ elements);

2. **Using a single query per processor.** Since our interest is in comparing runtime and not quality of approximation, we dramatically lowered the number of queries the three benchmark algorithms require to achieve their guarantees. Specifically, we set the parameters ε and δ for both FAST and the three low-adaptivity benchmarks such that all algorithms guarantee the same $1 - 1/e - 0.1$ approximation with probability 0.95 (see Appendix C.3). However, for the low-adaptivity benchmarks, we reduce their theoretical sample complexity in each round to have exactly **one** sample per processor (instead of their large sample complexity, e.g. 10^8 samples needed for AMORTIZED-FILTERING).

   This reduction in the number of samples per round allows the benchmarks to have each processor perform a single function evaluation per round instead of e.g. $10^8/95$ functions evaluations per processor per round, which 'unfairly' accelerates their runtimes at the expense of their approximations. However, we do *not* perform this reduction for FAST. Instead, we require FAST to compute the *full count* of samples for its guarantees. This is feasible since FAST samples elements rather than sets.

Data sets. Even with these modifications, for tractability we could only use small data sets:

- **Experiments 1.a: synthetic data sets ($n \approx 500$).** To compare the algorithms' runtimes under a range of conditions, we solve max cover on synthetic graphs generated via four different well-studied graph models: *Stochastic Block Model* (SBM); *Erdős Rényi* (ER); *Watts-Strogatz* (WS); and *Barbási-Albert* (BA). See Appendix C.4.1 for additional details;

- **Experiments 1.b: real data sets ($n \approx 500$).** To compare the algorithms' runtimes on real data, we optimize *Sensor Placement* on California roadway traffic data; *Movie Recommendation* on MovieLens data; *Revenue Maximization* on YouTube Network data; and *Influence Maximization* on Facebook Network data. See Appendix C.4.3 for additional details.

Results of experiment set 1. Figures 1 and 2 plot all algorithms' solution values and parallel runtimes for various k on synthetic and real data (each point is the mean of 5 trials with the corresponding k). In terms of solution values, across all experiments, values obtained by FAST are nearly indistinguishable from values obtained by GREEDY—the heuristic upper bound. From this comparison, it is clear that FAST does not compromise on the values of its solutions. In terms of runtime, FAST is 36 to 1600 times faster than BINARY-SEARCH-MAXIMIZATION; 7 to 120 times faster than RANDOMIZED-PARALLEL-GREEDY; 4 to 2200 times faster than AMORTIZED-FILTERING; and 1.1 to 7 times faster than PARALLEL-LTLG on the 8 objectives and various k (the time axes of Figures 1 and 2 are *log-scaled*). Appendix C.12 shows that FAST continues to outperform all benchmarks even (1) when we turn off its lazy updates, and (2) when we run all low-adaptivity benchmarks on just

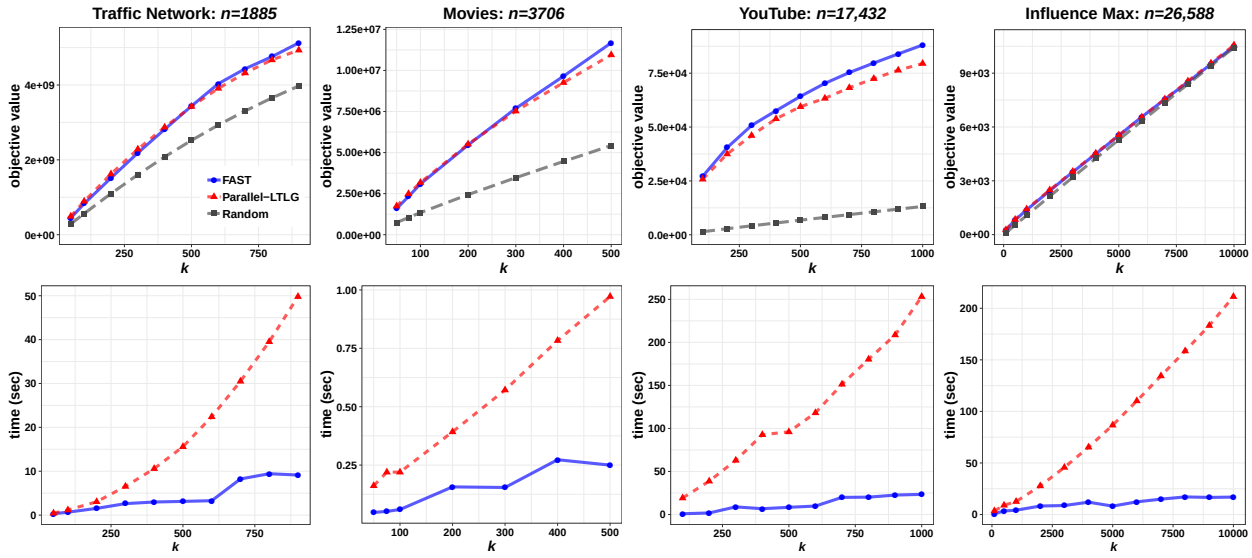

Figure 4. Experiment Set 2.b: FAST *(blue) vs.* PARALLEL-LTLG *(red) on real data.*

a single 'good' guess for OPT. We emphasize that FAST's faster runtimes were obtained despite the fact that the three other low-adaptivity algorithms were run with only a single sample per processor each iteration, rather than the 10^8 or 10^6 samples required for their respective guarantees.

4.2. Experiment set 2: FAST vs. Parallel-Lazier-than-Lazy-Greedy

Our second set of experiments compares FAST to the optimized parallel version of LAZIER-THAN-LAZY-GREEDY (LTLG) (Mirzasoleiman et al., 2015) on large data sets. Specifically, our optimized parallel MPI implementation of LTLG allows us to scale up to random graphs with $n \approx 100000$, large real data with n up to 26000, and various k from 25 to 25000 (see Appendix C.11). For these large experiments, running the parallel GREEDY algorithm is impractical. LTLG has a $(1 - 1/e - \varepsilon)$ approximation guarantee in expectation, so we likewise set both algorithms' parameters ε to guarantee a $(1 - 1/e - 0.1)$ approximation in expectation (see Appendix C.3).

Results of experiment set 2. Figures 3 and 4 plot solution values and runtimes for various k on large experiments with synthetic and real data (each point is the mean of 5 trials). In terms of solution values, while the two algorithms achieved similar solution values across all 8 experiments, FAST obtained slightly higher solution values than PARALLEL-LTLG on most objectives and values of k.

In terms of runtime, FAST was 1.5 to 32 times faster than PARALLEL-LTLG on each of the 8 objectives and all k we tried from $k = 25$ to 25000. More importantly, runtime disparities between FAST and PARALLEL-LTLG increase in larger k, so larger problems exhibit even greater runtime advantages for FAST.

Furthermore, we emphasize that due to the fact that the sample complexity of PARALLEL-LTLG is less than 95 for many experiments, it cannot achieve better runtimes by using more processors, whereas FAST can leverage up to n processors to achieve additional speedups. Therefore, FAST's fast relative runtimes are a loose lower bound for what can be obtained on larger-scale hardware and problems. Figure 5 plots FAST's parallel speedups versus the number of processors we use.

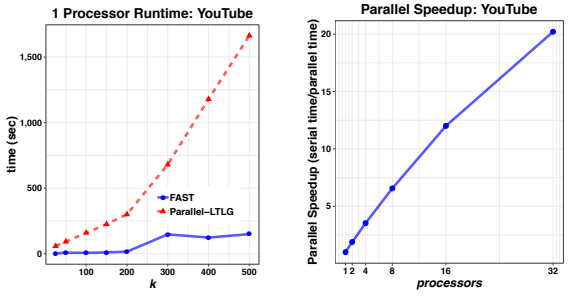

Figure 5. Single processor runtimes for FAST *and* PARALLEL-LTLG*, and parallel speedups vs. number of processors for* FAST *for the YouTube experiment. See Appendix C.14 for details.*

Finally, we note that *even on a single processor*, FAST is faster than LTLG for reasonable values of k on 7 of the 8 objectives due to the fact that FAST often uses fewer queries (see Appendix C.13). For example, Figure 5 plots single processor runtimes for the YouTube experiment.

5. Acknowledgements

This research was supported by a Google PhD Fellowship, NSF grant CAREER CCF-1452961, BSF grant 2014389, NSF USICCS proposal 1540428, NSF Grant 164732, a Google research award, and a Facebook research award.

References

Badanidiyuru, A. and Vondrák, J. Fast algorithms for maximizing submodular functions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pp. 1497–1514, 2014. doi: 10.1137/1.9781611973402.110. URL https://doi.org/10.1137/1.9781611973402.110.

Balkanski, E. and Singer, Y. The adaptive complexity of maximizing a submodular function. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 1138–1151. ACM, 2018a.

Balkanski, E. and Singer, Y. Approximation guarantees for adaptive sampling. In *International Conference on Machine Learning*, pp. 393–402, 2018b.

Balkanski, E., Breuer, A., and Singer, Y. Non-monotone submodular maximization in exponentially fewer iterations. *NIPS*, 2018.

Balkanski, E., Rubinstein, A., and Singer, Y. An exponential speedup in parallel running time for submodular maximization without loss in approximation. *SODA*, 2019a.

Balkanski, E., Rubinstein, A., and Singer, Y. An optimal approximation for submodular maximization under a matroid constraint in the adaptive complexity model. *STOC*, 2019b.

CalTrans. Pems: California performance measuring system. http://pems.dot.ca.gov/ [accessed: August 1, 2019].

Chekuri, C. and Quanrud, K. Parallelizing greedy for submodular set function maximization in matroids and beyond. *STOC*, 2019a.

Chekuri, C. and Quanrud, K. Submodular function maximization in parallel via the multilinear relaxation. *SODA*, 2019b.

Chen, L., Feldman, M., and Karbasi, A. Unconstrained submodular maximization with constant adaptive complexity. *STOC*, 2019.

Ene, A. and Nguyen, H. L. Submodular maximization with nearly-optimal approximation and adaptivity in nearly-linear time. *SODA*, 2019a.

Ene, A. and Nguyen, H. L. A nearly-linear time algorithm for submodular maximization with a knapsack constraint. *ICALP*, 2019b.

Ene, A. and Nguyen, H. L. Towards nearly-linear time algorithms for submodular maximization with a matroid constraint. *ICALP*, 2019c.

Ene, A., Nguyen, H. L., and Vladu, A. Submodular maximization with matroid and packing constraints in parallel. *STOC*, 2019.

Esfandiari, H., Karbasi, A., and Mirrokni, V. Adaptivity in adaptive submodularity. *arXiv preprint arXiv:1911.03620*, 2019.

Fahrbach, M., Mirrokni, V., and Zadimoghaddam, M. Submodular maximization with optimal approximation, adaptivity and query complexity. *SODA*, 2019a.

Fahrbach, M., Mirrokni, V. S., and Zadimoghaddam, M. Non-monotone submodular maximization with nearly optimal adaptivity and query complexity. *ICML*, 2019b.

Feldman, M., Harshaw, C., and Karbasi, A. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems 42, 1 (2015), 33 pages.*, 2015.

Harper, F. M. and Konstan., J. A. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages.*, 2015. doi: http://dx.doi.org/10.1145/2827872.

Hassidim, A. and Singer, Y. Robust guarantees of stochastic greedy algorithms. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1424–1432. JMLR. org, 2017.

Kazemi, E., Mitrovic, M., Zadimoghaddam, M., Lattanzi, S., and Karbasi, A. Submodular streaming in all its glory: Tight approximation, minimum memory and low adaptive complexity. *arXiv preprint arXiv:1905.00948*, 2019.

Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J. M., and Glance, N. S. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*, pp. 420–429, 2007. doi: 10.1145/1281192.1281239. URL https://doi.org/10.1145/1281192.1281239.

Minoux, M. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization techniques*, pp. 234–243. Springer, 1978.

Mirzasoleiman, B., Badanidiyuru, A., Karbasi, A., Vondrák, J., and Krause, A. Lazier than lazy greedy. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

Mirzasoleiman, B., Badanidiyuru, A., and Karbasi, A. Fast constrained submodular maximization: Personalized data summarization. In *ICML*, pp. 1358–1367, 2016.

Nemhauser, G. L. and Wolsey, L. A. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of operations research*, 3(3):177–188, 1978.

Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1): 265–294, 1978.

Qian, S. and Singer, Y. Fast parallel algorithms for statistical subset selection problems. In *Advances in Neural Information Processing Systems*, pp. 5073–5082, 2019.

Rossi, R. A. and Ahmed, N. K. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL http://networkrepository.com.

Traud, A. L., Mucha, P. J., and Porter, M. A. Social structure of Facebook networks. *Phys. A*, 391(16):4165–4180, Aug 2012.

Troyanskaya, O., Cantor, M., Sherlock, G., Brown, P., Hastie, T., Tibshirani, R., Botstein, D., and Altman, R. B. Missing value estimation methods for dna microarrays. *Bioinformatics*, 17(6):520–525, 2001.

Appendix

A. Vanilla ADAPTIVE-SEQUENCING

We begin by describing a simplified version of the algorithm. This algorithm is $\varepsilon^{-2}\log n$ adaptive (without additional dependence on constants), uses a total of $\varepsilon^{-2}nk$ queries (again, no additional constants), and obtains a $1-\frac{1}{e}-\frac{3}{2}\varepsilon$ approximation in expectation. Importantly, it assumes the value of the optimal solution $\texttt{OPT}$ is known. The full algorithm is an optimized version which does not assume $\texttt{OPT}$ is known and improves the query complexity.

A.1. Description of ADAPTIVE-SEQUENCING

ADAPTIVE-SEQUENCING, formally described below as Algorithm 3, generates at every iteration a random sequence $a_1,\ldots,a_k$ of elements that is used to both add elements to the current solution and discard elements from further consideration. More precisely, each element a_i, for $i\in[k]$, in $\text{SEQUENCE}(X,k)$ is a uniformly random element from the set of surviving elements X, which initially contains all elements. The algorithm identifies a position $i^\star$ in this sequence which determines the elements $a_1,\ldots,a_{i^\star-1}$ that are added to the current solution S, as well as the elements $a\in X$ with low contribution to $S\cup\{a_1,\ldots,a_{i^\star-1}\}$ that are discarded from X. This position $i^\star$ is defined to be the smallest position such that there is at least an ε fraction of elements in X with low contribution to $S\cup\{a_1,\ldots,a_{i^\star-1}\}$. By the minimality of $i^\star$, we simultaneously obtain that (1) the elements added to S, which are the elements before position $i^\star$, are likely to contribute high value to the solution and (2) at least an ε fraction of the surviving elements have low contribution to $S\cup\{a_1,\ldots,a_{i^\star-1}\}$ and are discarded.

The algorithm iterates until there are no surviving elements left in X. It then lowers the threshold t between high and low contribution and reinitializes the surviving elements X to be all elements. The algorithm lowers the threshold at most ε^{-1} times and then returns the solution S obtained.

Algorithm 3 ADAPTIVE-SEQUENCING

input function f, cardinality constraint k, parameter ε, value of optimal solution $\texttt{OPT}$
  $S\leftarrow\emptyset$
  while $|S|<k$ and number of iterations $<\varepsilon^{-1}$ **do**
    $X\leftarrow N, t\leftarrow(1-\varepsilon)(\texttt{OPT}-f(S))/k$
    while $X\neq\emptyset$ and $|S|<k$ **do**
      $a_1,\ldots,a_k\leftarrow\text{SEQUENCE}(X,k)$
      $i^\star\leftarrow\min\{i\in\{1,\ldots,k\}:|X_i|\leq(1-\varepsilon)|X|\}$
        with $X_i\leftarrow\left\{a\in X:f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a)\geq t\right\}$
      $S\leftarrow S\cup\{a_1,\ldots,a_{i^\star-1}\}$
      $X\leftarrow X_{i^\star}$
  return S

A.2. Analysis of ADAPTIVE-SEQUENCING

A.2.1. THE ADAPTIVE COMPLEXITY AND QUERY COMPLEXITY

The main observation to bound the number of iterations of the algorithm is that, by definition of $i^\star$, at least an ε fraction of the surviving elements in X are discarded at every iteration. Since the queries at every iteration of the inner-loop can be evaluated in parallel, the adaptive complexity is the total number of iterations of this inner-loop. For the query complexity, we note that there are $|X|k$ function evaluations per iteration.

Lemma 2. *The adaptive complexity of* ADAPTIVE-SEQUENCING *is at most* $\varepsilon^{-2}\log n$. *Its query complexity is at most* $\varepsilon^{-2}nk$.

Proof. We first analyze the adaptive complexity and then the query complexity.

The adaptive complexity. The algorithm consists of an outer-loop and an inner-loop. We first argue that at any iteration of the outer-loop, there are at most $\varepsilon^{-1}\log n$ iterations of the inner loop. By definition of $i^\star$, we have that $|X_{i^\star}|\leq(1-\varepsilon)|X|$. Thus there is at least an ε fraction of the elements in X that are discarded at every iteration. The inner-loop

terminates when $|X| = 0$, which occurs at the latest at iteration i where $(1-\varepsilon)^i n < 1$. This implies that there are at most $\varepsilon^{-1}\log n$ iterations of the inner loop. The function evaluations inside an iteration of the inner-loop are non-adaptive and can be performed in parallel in one round. These are the only function evaluations performed by the algorithm.[6] Since there are at most ε^{-1} iterations of the outer-loop, there are at most $\varepsilon^{-2}\log n$ rounds of parallel function evaluations.

The query complexity. In the inner-loop, the algorithm evaluates the marginal contribution of each element $a \in X$ to $S \cup \{a_1, \ldots, a_i\}$ for all $i \in \{0, \ldots, k-1\}$, so a total of $k|X|$ function evaluations. Similarly as for Lemma 2, at any iteration of the outer-loop, there are at most $\varepsilon^{-1}\log n$ iterations of the inner-loop and we have $|X| \leq (1-\varepsilon)^j n$ at iteration j. We conclude that the query complexity is $\frac{1}{\varepsilon}\sum_{j=1}^{\frac{\log n}{\varepsilon}} k(1-\varepsilon)^j n < \frac{nk}{\varepsilon^2}$. □

A.2.2. THE APPROXIMATION GUARANTEE

There are two cases depending on the condition which breaks the outer-loop. The main lemma for the case where there are ε^{-1} iterations of the outer-loop is that at every iteration, the elements added to S contribute an ε fraction of the remaining value $\texttt{OPT} - f(S)$.

Lemma 3. *Let S_i be the current solution S at the start of iteration i of the outer-loop of* ADAPTIVE-SEQUENCING. *For any i, if $|S_{i+1}| < k$, then $f_{S_i}(S_{i+1}) \geq \varepsilon(\texttt{OPT} - f(S_i))$.*

Proof. Since $|S_{i+1}| < k$, $X = \emptyset$ at the end of iteration i. This implies that for all elements $a \in N$, a is discarded from X by the algorithm at some iteration where $f_{S\cup\{a_1,\ldots,a_{i^\star-1}\}}(a) < (1-\varepsilon)(\texttt{OPT} - f(S_i))/k$ for some $S_i \subseteq S \subseteq S_{i+1}$. By submodularity, for any element $a \in N$, we get $f_{S_{i+1}}(a) \leq (1-\varepsilon)(\texttt{OPT} - f(S_i))/k$. Next, by monotonicity and submodularity, $\texttt{OPT} - f(S_{i+1}) \leq f_{S_{i+1}}(O) \leq \sum_{o\in O} f_{S_{i+1}}(o)$. Combining the two previous inequalities, we obtain

$$\texttt{OPT} - f(S_{i+1}) \leq \sum_{o\in O} f_{S_{i+1}}(o) \leq \sum_{o\in O}(1-\varepsilon)(\texttt{OPT} - f(S_i))/k = (1-\varepsilon)(\texttt{OPT} - f(S_i)).$$

By rearranging the terms, we get the desired result. □

The main lemma for the case where $|S| = k$ is that the expected contribution of each element a_i added to the current solution S is arbitrarily close to a $1/k$ fraction of the remaining value $\texttt{OPT} - f(S)$.

Lemma 4. *At any iteration of the inner-loop of* ADAPTIVE-SEQUENCING, *for all $i < i^\star$, we have $\mathbb{E}_{a_i}\left[f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a_i)\right] \geq (1-\varepsilon)^2(\texttt{OPT} - f(S))/k$.*

Proof. Since a_i is a uniformly random element from X and $|X_i| \geq (1-\varepsilon)|X|$ for $i < i^\star$, we have

$$\mathop{\mathbb{E}}_{a_i}\left[f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a_i)\right] \geq \Pr_{a_i}\left[f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a_i) \geq t\right]\cdot t \geq (1-\varepsilon)\cdot(1-\varepsilon)(\texttt{OPT} - f(S))/k. \quad \square$$

By standard greedy analysis, Lemmas 3 and 4 imply that the algorithm obtains a $1 - 1/e - \mathcal{O}(\varepsilon)$ approximation in each case. We emphasize the low constants and dependencies on ε in this result compared to previous results in the adaptive complexity model.

Theorem 5. ADAPTIVE-SEQUENCING *is an algorithm with at most $\varepsilon^{-2}\log n$ adaptive rounds and $\varepsilon^{-2}nk$ queries that achieves a $1 - 1/e - \frac{3\varepsilon}{2}\varepsilon$ approximation in expectation.*

Proof. We first consider the case where there are ε^{-1} iterations of the outer-loop. Let $S_1, \ldots, S_{\varepsilon^{-1}}$ be the set S at each of the ε^{-1} iterations of ADAPTIVE-SEQUENCING. The algorithm increases the value of the solution S by at least $\varepsilon\left(\texttt{OPT} - f(S)\right)$ at every iteration by Lemma 3. Thus,

$$f(S_i) \geq f(S_{i-1}) + \varepsilon\left(\texttt{OPT} - f(S_{i-1})\right).$$

Next, we show by induction on i that

$$f(S_i) \geq \left(1 - (1-\varepsilon)^i\right)\texttt{OPT}.$$

[6] The value of $f(S)$ needed to compute t can be obtained using $f_{S\cup\{a_1,\ldots,a_{i^\star-1}\}}(a_{i^\star})$ that was computed in the previous iteration.

Observe that

$$\begin{aligned} f(S_i) &\geq f(S_{i-1}) + \varepsilon\left(\texttt{OPT} - f(S_{i-1})\right) \\ &= \varepsilon\texttt{OPT} + (1-\varepsilon)\, f(S_{i-1}) \\ &\geq \varepsilon\texttt{OPT} + (1-\varepsilon)\left(1-(1-\varepsilon)^{i-1}\right)\texttt{OPT} \\ &= \left(1-(1-\varepsilon)^{i}\right)\texttt{OPT} \end{aligned}$$

Since $1-x \leq e^{-x}$, we get

$$f(S_{\varepsilon^{-1}}) \geq \left(1-e^{-1}\right)\texttt{OPT}.$$

Similarly, for the case where the solution S returned is such that $|S| = k$, by Lemma 4 and by induction we get that

$$f(S) \geq \left(1-e^{-(1-\varepsilon)^2}\right)\texttt{OPT} \geq \left(1-e^{-(1-2\varepsilon)}\right)\texttt{OPT} \geq \left(1-e^{-1}(1-4\varepsilon)\right)\texttt{OPT} \geq (1-e^{-1}-\frac{3}{2}\varepsilon)\texttt{OPT}.$$

□

B. Analysis of the Main Algorithm

We define $\ell = \log(\varepsilon^{-1}\log k)$ and $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)}\log(4\ell\varepsilon^{-2}\log(n)\delta^{-1})$

B.1. Adaptive Complexity and Query Complexity

The adaptivity of the main algorithm is slightly worse than for ADAPTIVE-SEQUENCING due to the binary searches over V and I. To obtain the adaptive complexity with probability 1, if at any iteration of the outer while-loop there are at least $\varepsilon^{-1}\log n$ iterations of the inner-loop, we declare failure. In Lemma 2, we show this happens with low probability.

Lemma 6. *The adaptive complexity of* FAST *is at most* $\varepsilon^{-2}\log(n)\cdot\log^2(\varepsilon^{-1}\log k)$.

Proof. The algorithm consists of four nested loops: a binary search over V, an outer while-loop, an inner while-loop, and a binary search over I. For the binary searches, we have $|V| \leq \varepsilon^{-1}\log k$ and $|I| \leq \varepsilon^{-1}\log k$. Thus, there are at most ℓ iterations for each binary search.

Due to the termination condition of the while-loops, there are at most ε^{-1} and $\varepsilon^{-1}\log n$ iterations of each while-loop. The function evaluations inside an iteration of the last nested loop are non-adaptive and can be performed in parallel in one round. Thus the adaptive complexity of FAST is a most

$$\varepsilon^{-2}\log(n)\cdot\log^2(\varepsilon^{-1}\log k).$$

□

Thanks to the binary search over I and the subsampling of R from X, the query complexity is improved from $\mathcal{O}(\varepsilon^{-2}nk)$ to $\tilde{O}(\varepsilon^{-2}n + \varepsilon^{-4}\log(n)\log(\delta^{-1}))$

Lemma 7. *The query complexity of* FAST *is at most* $2\varepsilon^{-2}\ell n + \ell^2\varepsilon^{-2}\log(n)m$.

Proof. There are n queries, $f(a)$ for all a, needed to compute V. At each iteration of the binary search over I, there are m queries needed for R_i to evaluate $f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a)$ for $a \in R$. There are at most $\ell\varepsilon^{-2}\log n$ instances of the binary search over I, each with at most ℓ iterations. The total number of queries for this binary search is at most

$$\ell^2\varepsilon^{-2}\log(n)m.$$

At each iteration i of the inner-while loop, there are at most $|X| \leq (1-\varepsilon)^i n$ queries to update X and at most $|X|$ queries to add elements a_i to S. There are at most $\ell\varepsilon^{-1}$ instances of the inner while-loop each with at most $\varepsilon^{-1}\log n$ iterations. The total number of queries for updating X and S is

$$\ell\varepsilon^{-1}\sum_{i=1}^{\varepsilon^{-1}\log n} 2(1-\varepsilon)^i n \leq 2\varepsilon^{-2}\ell n.$$

By combining the queries needed to compute V, R_i, X and S, we get the desired bound on the query complexity. □

B.2. The Approximation

B.2.1. FINDING $i^\star$

Similarly as for ADAPTIVE-SEQUENCING, we denote $X_i = \{a \in X : f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a) \geq t\}$ and $R_i = \{a \in R : f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a) \geq t\}$.

Lemma 1. *Assume that $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)}\log(4\ell\varepsilon^{-2}\log(n)\delta^{-1})$, then, with probability $1-\delta/2$, for all iterations of the inner while-loop, we have that $|X_{(1-\varepsilon)i^\star}| \geq (1-3\varepsilon)|X|$ and $|X_{i^\star}| \leq (1-\varepsilon)|X|$.*

Proof. By the definition of $i^\star$ and I, we have that $|R_{(1-\varepsilon)i^\star}| \geq (1-2\varepsilon)|R|$ and $|R_{i^\star}| \leq (1-2\varepsilon)|R|$. We show by contrapositive that, with probability $1-\delta/4$ if $|X_{(1-\varepsilon)i^\star}| \leq (1-3\varepsilon)|X|$ then $|R_{(1-\varepsilon)i^\star}| \leq (1-2\varepsilon)|R|$ and that if $|X_{i^\star}| \geq (1-\varepsilon)|X|$ then $|R_{i^\star}| \geq (1-2\varepsilon)|R|$.

Note that for all $a \in R$ and $i \in [k]$, we have

$$\Pr_a\left[f_{S\cup\{a_1,\ldots,a_{i-1}\}}(a) \geq t\right] = \frac{|X_i|}{|X|}.$$

First, assume that $|X_{i^\star}| > (1-\varepsilon)|X|$. Then by the Chernoff bound, with $\mu = m \cdot \frac{|X_{i^\star}|}{|X|} \geq (1-\varepsilon)m$,

$$\Pr\left[|R_{i^\star}| \leq (1-2\varepsilon)|R|\right] \leq \Pr\left[|R_{i^\star}| \leq (1-\varepsilon)^2 m\right] \leq e^{-\varepsilon^2(1-\varepsilon)m/(2+\varepsilon)} \leq \frac{\delta}{4\ell\varepsilon^{-2}\log n}.$$

Next, assume that $|X_{i^\star}| < (1-3\varepsilon)|X|$. By the Chernoff bound with $\mu \leq (1-3\varepsilon)m$,

$$\Pr\left[|R_{i^\star}| \geq (1-2\varepsilon)|R|\right] \leq \Pr\left[|R_{i^\star}| \geq (1+\varepsilon)(1-3\varepsilon)m\right] \leq \frac{\delta}{4\ell\varepsilon^{-2}\log n}.$$

Thus, with $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)}\log(4\ell\varepsilon^{-2}\log(n)\delta^{-1})$ and by contrapositive, we have that $|X_{(1-\varepsilon)i^\star}| \geq (1-3\varepsilon)|X|$ and $|X_{i^\star}| \leq (1-\varepsilon)|X|$ each with probability $1-\delta/(4\ell\varepsilon^{-2}\log n)$. By a union bound, these both hold with probability $1-\delta/2$ for all $\ell\varepsilon^{-2}\log n$ iterations of the inner while-loop. □

Corollary 1. *With probability $1-\delta/2$, for all iterations of the inner while-loop, we have*

- *$|X_i| \geq (1-3\varepsilon)|X|$ for all $i < (1-\varepsilon)i^\star$, and*
- *$|X_i| \leq (1-\varepsilon)|X|$ for all $i \geq i^\star$*

Proof. Consider an iteration of the inner while-loop. We first note that, by submodularity, $|R_i|$ is monotonically decreasing as i increases. Thus we can perform a binary search over I to find $i^\star$. By Lemma 1, we have that with probability $1-\delta/2$, we have that $|X_{(1-\varepsilon)i^\star}| \geq (1-3\varepsilon)|X|$ and $|X_{i^\star}| \leq (1-\varepsilon)|X|$. We conclude the proof by noting that by submodularity, $|X_i|$ is also monotonically decreasing as i increases. □

Lemma 2. *With probability $1-\delta/2$, at every iteration of the outer while-loop, $X = \emptyset$ after at most $\varepsilon^{-1}\log n$ iterations of the inner while-loop.*

Proof. By Lemma 1, with probability $1-\delta/2$, at every iteration of the inner while-loop, there is at least an ε fraction of the elements in X that are discarded. We assume this is the case. After $\varepsilon^{-1}\log n$ iterations of discarding an ε fraction of the elements in X, we have $X = \emptyset$.

□

B.2.2. IF NUMBER OF ITERATIONS OF OUTER WHILE-LOOP IS ε^{-1}

The analysis defers depending on whether the number of iterations or the size of the solution caused the algorithm to terminate. We first analyze the case where ADAPTIVE-SEQUENCING returned S s.t. $|S| < k$ because the number of iterations reached ε^{-1}. The main lemma for this case is that at every iteration of ADAPTIVE-SEQUENCING, if $v \leq \texttt{OPT}$, the set T added to the current solution S contributes at least an ε fraction of the remaining value $v - f(S)$.

Lemma 3. *Assume that $v \leq \texttt{OPT}$ and let S_i be the set S at the start of iteration i of the outer while-loop of* FAST. *With probability $1 - \delta/2$, for all $v \in V$ and all $i \leq \varepsilon^{-1}$, we have that if $|S_{i+1}| < k$, then $f_{S_i}(S_{i+1}) \geq \varepsilon(v - f(S_i))$.*

Proof. By Lemma 2, with probability $1 - \delta/2$, at every iteration of the outer while-loop, $X = \emptyset$ after at most $\varepsilon^{-1} \log n$ iterations of the inner while-loop. We assume this holds for the remaining of this proof.

Since $|S_{i+1}| < k$, $X = \emptyset$ at the end of iteration i of the outer while-loop. This implies that for all elements $a \in N$, a is discarded from X by the algorithm at some iteration where

$$f_{S \cup \{a_1, \ldots, a_{i^\star - 1}\}}(a) < (1 - \varepsilon)(v - f(S_i))/k$$

for some $S_i \subseteq S \subseteq S_{i+1}$. By submodularity, for any element $a \in N$, we get

$$f_{S_{i+1}}(a) \leq (1 - \varepsilon)(v - f(S_i))/k.$$

Next, since $v \leq \texttt{OPT}$, by monotonicity, and by submodularity,

$$v - f(S_{i+1}) \leq \texttt{OPT} - f(S_{i+1}) \leq f_{S_{i+1}}(O) \leq \sum_{o \in O} f_{S_{i+1}}(o).$$

Combining the previous inequalities, we obtain

$$v - f(S_{i+1}) \leq \sum_{o \in O} f_{S_{i+1}}(o) \leq \sum_{o \in O} (1 - \varepsilon)(v - f(S_i))/k = (1 - \varepsilon)(v - f(S_i)).$$

By rearranging the terms, we get the desired result. □

By standard greedy analysis, similarly as for the proof of Theorem 5 we obtain that $f(S) \geq (1 - 1/e)v$.

Lemma 4. *With probability $1 - \delta/2$, for all $v \leq \texttt{OPT}$, after ε^{-1} iterations of the outer while-loop of* FAST, *$f(S) \geq (1 - 1/e)v$.*

B.2.3. IF $|S| = k$

Next, we analyze the case where the outer-loop terminated because $|S| = k$. We show that each element added to S is, in expectation, a good approximation to t_1.

Lemma 5. *With probability $1 - \delta/2$, at every iteration of the inner while-loop, we have that independently for each $i \leq (1 - \varepsilon)i^\star$, with probability at least $1 - 3\varepsilon$,*

$$f_{S \cup \{a_1, \ldots, a_{i-1}\}}(a_i) \geq (1 - \varepsilon)(v - f(S))/k.$$

Proof. By Corollary 1, we have that with probability $1 - \delta/2$, for all iterations of the inner while-loop,

$$\left|\{a \in X : f_{S \cup \{a_1, \ldots, a_{i-1}\}}(a) \geq (1 - \varepsilon)(v - f(S))/k\}\right| = |X_i| \geq (1 - 3\varepsilon)|X|$$

for all $i \leq (1 - \varepsilon)i^\star$. We assume this is the case and consider an iteration of the inner while-loop. Since each a_i is a uniformly random element from X, we have that independently for each $i \leq (1 - \varepsilon)i^\star$,

$$\Pr_{a_i}\left[f_{S \cup \{a_1, \ldots, a_{i-1}\}}(a_i) \geq (1 - \varepsilon)(v - f(S))/k\right] \geq 1 - 3\varepsilon.$$

□

B.2.4. GUESSING OPT

Lemma 6 (Extends (Hassidim & Singer, 2017))**.** *Consider a set $S = \{a_1, \ldots, a_{|S|}\}$ and let $S_i = \{a_1, \ldots, a_i\}$. Assume that, independently for each $i \in [|S|]$, we have that with probability at least $1 - \delta$,*

$$f_{S_{i-1}}(a_i) \geq \mu \cdot \frac{1}{k}(v - f(S_{i-1})),$$

then, for any $\varepsilon \in (0,1)$ *such that* $|S| \geq \frac{1}{\varepsilon^2(1-\delta)\mu}$,

$$f(S) \geq \left(1 - e^{-\frac{|S|}{k}(1-\delta)\mu(1-\varepsilon)}\right) v$$

with probability at least $1 - e^{-|S|(1-\delta)\mu\varepsilon^2/2}$.

Proof. The analysis is similar as in (Hassidim & Singer, 2017). Assume that $f_{S_{i-1}}(a_i) = \xi_i \cdot \frac{1}{k}(v - f(S_{i-1}))$ and let $\hat{\mu} = \frac{1}{k}\sum_{i=1}^{|S|} \xi_i$. We first argue that $f(S) \geq \left(1 - e^{-\hat{\mu}}\right) v$. By induction, we have that

$$f(S_i) \geq \left(1 - \prod_{j=1}^{i}\left(1 - \frac{\xi_j}{k}\right)\right) v.$$

Since $1 - x \leq e^{-x}$, we obtain

$$f(S) \geq \left(1 - \prod_{j=1}^{|S|}\left(1 - \frac{\xi_j}{k}\right)\right) v \geq \left(1 - e^{-\sum_{j=1}^{|S|}\frac{\xi_j}{k}}\right) v \left(1 - e^{-\hat{\mu}}\right) v.$$

Let $S' = \{a_i \in S : f_{S_{i-1}}(a_i) \geq \mu \cdot \frac{1}{k}(v - f(S_{i-1}))$. By the Chernoff bound,

$$\Pr\left[|S'| < (1-\varepsilon)(1-\delta)|S|\right] \leq e^{-\frac{\varepsilon^2(1-\delta)|S|}{2}}.$$

Thus, with probability at least $1 - e^{-\frac{\varepsilon^2(1-\delta)|S|}{2}}$, we get

$$\hat{\mu} \geq \frac{1}{k}|S'|\mu \geq \frac{|S|}{k}(1-\varepsilon)(1-\delta)\mu.$$

□

Lemma 7. *Assume* $k \geq \frac{2\log(2\delta^{-1}\ell)}{\varepsilon^2(1-5\varepsilon)}$. *With probability at least* $1-\delta$, *we have that for all* v *at some iteration of the binary search over* V *such that* $v \leq \texttt{OPT}$,

$$f(S) \geq \left(1 - e^{-(1-6\varepsilon)}\right) v.$$

Proof. With probability $1 - \delta/2$, Corollary 1, and consequently Lemma 4 and Lemma 5, hold for all iterations of the inner while-loop and we assume this is the case for the remainder of this proof.

Consider v at some iteration of the binary search over V. If the outer while-loop terminated after ε^{-1} iterations, then by Lemma 4, we have $f(S) \geq (1 - 1/e)\, v$.

Otherwise, the outer while-loop terminated with $|S| = k$. The algorithm adds elements to S that are of two types: those added before the if condition and those in $A_{i^\star}$ added after. Let $T \subseteq S$ be the set obtained by discarding from S the elements $a \in A_{i^\star}$ that, at the iteration of the inner while-loop where a was added, had position i in the sequence $a_1, \ldots, a_{k-|S|}$ such that $(1-\varepsilon)i^\star \leq i \leq i^\star$. This set T is such that $|T| \geq (1-\varepsilon)|S| = (1-\varepsilon)k$.

Consider $a_i \in S$ added before the if condition and let S_{i-1} be the set of elements in S added to S before a_i. Since a_i was added to S, by submodularity, we have $f_{S_{i-1}}(a_i) \geq (1-\varepsilon)\frac{1}{k}(v - f_{S_{i-1}})$ with probability 1. Consider an elements $a_i \in T$. By Lemma 5 and by definition of T, we have that independently for each $a_i \in T$, with probability at least $1 - 3\varepsilon$, $f_{S_{i-1}}(a_i) \geq (1-\varepsilon)\frac{1}{k}(v - f_{S_{i-1}})$ where S_{i-1} is the set of elements in T added to T before a_i.

By Lemma 6 and since $|T| \geq (1-\varepsilon)k$, with $\delta = 3\varepsilon$, $\mu = 1-\varepsilon$, and $\varepsilon = \varepsilon$, we have that $f(T) \geq \left(1 - e^{-(1-\varepsilon)(1-\varepsilon)(1-3\varepsilon)(1-\varepsilon)}\right) v \geq\geq \left(1 - e^{-(1-6\varepsilon)}\right) v$ with probability at least $1 - \delta/(2\ell)$ if $k \geq \frac{2\log(2\delta^{-1}\ell)}{\varepsilon^2(1-\varepsilon)(1-\varepsilon)(1-3\varepsilon)} \geq \frac{2\log(2\delta^{-1}\ell)}{\varepsilon^2(1-5\varepsilon)}$. By monotonicity, $f(S) \geq f(T)$.

By a union bound over all ℓ iterations of the binary search over V, for all v considered during this binary search, we have that $f(S) \geq \left(1 - e^{-(1-6\varepsilon)}\right) v$ with probability at least $1 - \delta/2$. □

Theorem 1. *Assume* $k \geq \frac{2\log(2\delta^{-1}\ell)}{\varepsilon^2(1-5\varepsilon)}$ *and* $\varepsilon \in (0, 0.1)$*, where* $\ell = \log(\frac{\log k}{\varepsilon})$*. Then,* FAST *with sample complexity* $m = \frac{2+\varepsilon}{\varepsilon^2(1-3\varepsilon)}\log(\frac{4\ell\log n}{\delta\varepsilon^2})$ *has at most* $\varepsilon^{-2}\log(n)\ell^2$ *adaptive rounds,* $2\varepsilon^{-2}\ell n + \varepsilon^{-4}\log(n)\ell^2 m$ *queries, and achieves a* $1 - \frac{1}{e} - 4\varepsilon$ *approximation with probability* $1 - \delta$*.*

Proof. By the definition of V and since $\max_{a\in N} f(a) \leq \texttt{OPT} \leq \max_{|S|\leq k}\sum_{a\in S} f(a)$, there exists $v' \in V$ such that $v' \in [(1-\varepsilon)\texttt{OPT}, \texttt{OPT}]$.

By Lemma 7, with probability at least $1-\delta$, we have that for all v at some iteration of the binary search over V such that $v \leq \texttt{OPT}$, $f(S) \geq \left(1 - e^{-(1-6\varepsilon)}\right) v$. Since $v' \leq \texttt{OPT}$, it must be the case that $v^\star \geq v'$ and we get

$$\left(1 - e^{-(1-6\varepsilon)}\right)^{-1} f(S_{v^\star}) \geq\ v^\star \geq v' \geq (1-\varepsilon)\texttt{OPT}.$$

For $\varepsilon \in (0, 0.1)$, we have $e^{6\varepsilon} \leq 1 + 9\varepsilon$. We get

$$(1-\varepsilon)(1 - e^{-(1-6\varepsilon)} \geq (1-\varepsilon)(1 - e^{-1}(1+9\varepsilon)) \geq 1 - e^{-1} - 4\varepsilon.$$

□

C. Additional Information for Experiments

C.1. Benchmark algorithms

Our first set of experiments compares FAST's performance to three state-of-the-art low-adaptivity algorithms:

- **Amortized-Filtering (Balkanski et al., 2019a).** Given a guess for OPT, AMORTIZED-FILTERING proceeds as follows: At each round, AMORTIZED-FILTERING sets an adaptive value threshold based on the value of its current solution. It uses this threshold to filter remaining elements into high-value and low-value groups. It then adds a randomly chosen set of high-value elements to the solution and updates the threshold for the next round. AMORTIZED-FILTERING achieves a $(1 - 1/e - \varepsilon)$ approximation in $O(\log(n)\varepsilon^{-3})$ rounds.

- **Randomized-Parallel-Greedy (Chekuri & Quanrud, 2019b).** Given a guess for OPT, RANDOMIZED-PARALLEL-GREEDY proceeds as follows: At each round, RANDOMIZED-PARALLEL-GREEDY partitions elements into high-value and low-value groups based on their marginal contributions. It then uses the multilinear extension to estimate the maximum probability ('step size') with which it can randomly add elements from the high value group to the solution while maintaining theoretical guarantees. This algorithm achieves a $(1 - 1/e - \varepsilon)$-approximation in $O(\log(n)\varepsilon^{-2})$ parallel rounds.

- **Binary-Search-Maximization (Fahrbach et al., 2019a).** Given a guess for OPT, BINARY-SEARCH-MAXIMIZATION begins by fixing a value threshold. It then iteratively partitions remaining elements into high-value and low-value groups by determining whether each element's average marginal contribution to a random set exceeds the value threshold. It draws elements from the high-value group to form a candidate solution S. Finally, it lowers the value threshold and repeats the process for several rounds, keeping track of the candidate solution with the highest value. BINARY-SEARCH-MAXIMIZATION achieves a $(1-1/e-\varepsilon)$-approximation in $O(\log(n)\varepsilon^{-2})$ adaptive rounds.

We also compare FAST to a parallel version of LAZIER-THAN-LAZY-GREEDY (LTLG):

- **Parallel-Lazier-than-Lazy-Greedy (Parallel-LTLG) (Mirzasoleiman et al., 2015).** LTLG is widely regarded as the fastest algorithm for submodular maximization in practice. At each round, it draws a small random sample of elements. It then attempts a lazy update via a single query by testing whether the element in the sample with the highest previously-computed marginal value has a current marginal value that exceeds the second-highest previously-computed marginal value among the samples. If this is the case, it adds this best element to the solution. Otherwise, it computes marginal contribution of all samples and adds the best element in the sample set to the solution. It achieves a $(1 - 1/e - \varepsilon)$ approximation in k adaptive rounds.

For calibration, we also ran (1) a parallel version of the standard GREEDY algorithm and (2) an algorithm that estimates the value of a random solution:

- **Parallel-Greedy.** GREEDY iteratively adds the element with the highest marginal value to the solution set S for each of k rounds. GREEDY achieves a $1 - 1/e$ approximation in k adaptive rounds, and its solution values are widely regarded as an heuristic upper bound.
- **Random.** RANDOM returns the average value of a randomly chosen set S of k elements.

C.2. Choosing and optimizing benchmarks' guesses for OPT

Low-adaptivity benchmarks also require guesses for OPT. We introduce optimizations to these guesses to accelerate the benchmarks as described here. Note that in Appendix C.12, we also rerun the low-adaptivity experiments giving each low-adaptivity benchmark just a single 'good' guess for OPT. We also introduce several other optimizations to the benchmark algorithms—see Appendix C.9.

- **Amortized-Filtering (Balkanski et al., 2019a).** Recall that when OPT is unknown, the standard approach described in Balkanski et al. (2019a) is to run AMORTIZED-FILTERING once for each guess of OPT. This requires ~ 60 unique runs of AMORTIZED-FILTERING to maintain the approximation guarantee even for relatively small k and $\varepsilon = 0.1$. Therefore, to optimize this algorithm, we (1) implement binary search over these guesses of OPT. We also (2) introduce the same stopping condition that we describe for FAST, such that whenever a run of AMORTIZED-FILTERING with a particular guess of OPT finds a solution S with $f(S) \geq (1 - 1/e - 0.1)v$, where v is an upper-bound guess for OPT, then we return this solution. Finally, we (3) set the upper-bound guess v to $v = \max_{|S|\leq k} \sum_{a\in S} f(a)$ of the k highest valued singletons (as in FAST), rather than the looser upper bound guess described in Balkanski et al. (2019a). These optimizations reduce the value of AMORTIZED-FILTERING's solutions in practice, but they dramatically accelerate its runtimes to provide a more stringent runtime benchmark for FAST.
- **Randomized-Parallel-Greedy (Chekuri & Quanrud, 2019b).** The analysis in Chekuri & Quanrud (2019b) shows that we can either use multiple guesses for OPT, or we can use a single guess that is an upper bound for OPT. We use the latter option, as using a single guess is the fastest approach, so this choice is consistent with our goal of providing the most difficult speed benchmarks for FAST. Specifically, we guess OPT to be the sum $v = \max_{|S|\leq k} \sum_{a\in S} f(a)$ of the k highest valued singletons, which is an upper bound on OPT (and the same guess used by FAST). We note that this is a tighter upper bound on the value of the true OPT than commonly used alternatives (e.g. k times the value of the top singleton), so by choosing this guess, we further accelerate our runs of RANDOMIZED-PARALLEL-GREEDY. We also note that using a single guess for OPT achieves this greater speed by sacrificing some solution value, which is why in our experiments RANDOMIZED-PARALLEL-GREEDY sometimes finds solutions that have lower values than other benchmarks.
- **Binary-Search-Maximization (Fahrbach et al., 2019a).** BINARY-SEARCH-MAXIMIZATION already includes a clever processing scheme that efficiently searches for tighter upper-bound and lower-bound guesses for OPT, so we implement this approach exactly as described in Fahrbach et al. (2019a).

C.3. Choosing parameters ε and δ

For *Experiment set 1*, we choose all algorithms' δ and ε such that each guarantees a $(1-1/e-0.1)$ approximation with probability 0.95. We therefore choose $\delta = 0.95$ for all algorithms and set ε to 0.025 for FAST; 0.1 for AMORTIZED-FILTERING; 0.1 for for BINARY-SEARCH-MAXIMIZATION; and 0.048 for RANDOMIZED-PARALLEL-GREEDY. For *Experiment set 2*, the $(1 - 1/e - \varepsilon)$ approximation guarantee of LTLG holds in expectation, so we set $\varepsilon = 0.1$ for PARALLEL-LTLG and $\varepsilon = 0.025$ for FAST, which gives the same $(1 - 1/e - 0.1)$ approximation in expectation (see Theorem 1).

C.4. Objective functions and data sets

C.4.1. MAX COVER ON RANDOM GRAPHS

Recall the max cover objective: given a graph G, the cover function $f(S)$ measures the count of nodes with at least one neighbor in S. This is a canonical monotone submodular function. To compare the algorithms' runtimes under a range of conditions, we solve max cover on synthetic graphs generated via four different well-studied graph models:

- **Erdős Rényi.** We generate $G(n, p)$ graphs with a $p = 0.01$ probability of each edge. Since many nodes have similar degree in this model and each node's edges are spread randomly across the graph, a random set of nodes often achieves good coverage.
- **Stochastic block model.** We generate SBM graphs with a $p = 0.1$ probability of an edge between each pair of nodes in the same cluster. Here, we expect that a good solution will cover nodes in all clusters.
- **Watts-Strogatz.** We generate WS graphs initialized as ring lattices with 2 edges per node and a $p = 0.1$ probability of rewiring edges. In these 'small-world' graphs, many nodes have identical degree, so good solutions contain nodes chosen to minimize coverage overlaps.
- **Barbási-Albert.** We generate BA graphs with $m = 1$ edges added per iteration. BA graphs exhibit scale-free structure and tend to have a small set of high-degree nodes. Therefore, it is often possible to obtain high coverage in these graphs by choosing the highest degree nodes.

C.4.2. RANDOM GRAPHS: EXPERIMENT SIZE

For ER, WS, and BA graphs, we set $n = 500$ in our small experiments and $n = 100,000$ in our large experiments. For SBM graphs, we fix parameters to approximately match these sizes in expectation, as the actual size of an SBM graph is a draw from a random process. Specifically, for small SBM experiments we draw 10 clusters of 10 to 100 nodes each, and for large experiments we draw 50 clusters of 100 to 5000 nodes each.

C.4.3. REAL DATA

- **Traffic speeding sensor placement.** In this application, we select a set of locations to install traffic speeding sensors on a highway network, and our objective is to choose locations to maximize the traffic that the sensors observe. Similarly to (Balkanski et al., 2018), we conduct this experiment using data from the CalTrans PeMS system (CalTrans), which allows us to reconstruct the directed network where nodes are locations on each California highway ($40,000$ locations) and directed edges are the total count of vehicles that passed between adjacent locations in April, 2018. We use the directed, weighted max cover function to measure the total count of traffic observed at a set of sensor locations. For a given set S of sensor locations, this objective function returns the sum of edge weights (traffic counts along roadway sections) for which at least one endpoint is in S. For our small experiments, we follow (Balkanski et al., 2018) and restrict the network to the 521 locations within a 10 miles of the Los Angeles city center. For our large experiments, we expand this to all of the ~ 2000 locations in the region.
- **Movie recommendation.** In movie recommendation, the objective is to recommend a small, diverse, and highly-rated set of movies based on a data set of users' movie ratings. We use the objective function and dataset from (Balkanski & Singer, 2018b), which sums the ratings of movies in the set S and includes a diversity term that captures how well the chosen set of movies covers the set of movie genres in the data. A good set of movies includes movies that have high overall ratings, but it also should appeal to users' different tastes by including at least one film that each user rates very highly. Therefore, we also include a diversity term that counts the number of users who would give a high rating to at least one film in the set of movies S. We obtain the following objective:

$$f(S) = \sum_{i \in U} \sum_{j \in S} r_{i,j} + \alpha C(S) + \beta D(S) \tag{1}$$

where U is the set of users i, $r_{i,j}$ is user i's predicted rating of movie j; $C(S)$ is a coverage function that counts the number of different genres covered by S; $D(S)$ is a coverage function that counts the number of users with at least one highly rated film in S; and parameters $\alpha \geq 0$ and $\beta \geq 0$ control the relative weight that the objective function places on highly rated movies versus diversity. Note that eqn. 1 is a monotone submodular function. As in (Balkanski & Singer, 2018b), we predict missing ratings for the user-movie ratings matrix using the standard approach of low-rank matrix completion via the iterative low-rank SVD decomposition algorithm SVDIMPUTE analyzed in (Troyanskaya et al., 2001). We set $\alpha = 0.5 \max_j(\sum_i r_j)$ and $\beta = 1$, and we define a high rating as $r_j > 4.5$ (which corresponds to 1% of the ratings). For our small experiments, we randomly select 500 movies and users from the MovieLens $1m$ data set of 6000 users' ratings of 4000 movies (Harper & Konstan., 2015). For our larger experiments, we use the entire data set.

- **Revenue maximization on YouTube.** In the revenue maximization experiment, we choose a set of YouTube users who will each advertise a different product to their network neighbors, and the objective is to maximize product revenue. We adopt an objective function and dataset based on (Mirzasoleiman et al., 2016). Specifically, the expected revenue from each user is a function $V(S)$ of the sum of influences (edge weights) of her neighbors who are in S:

$$f(S) = \sum_{i \in X} V\Big(\sum_{j \in S} w_{i,j}\Big) \tag{2}$$

$$V(y) = y^{\alpha} \tag{3}$$

  where X is the set of all users (nodes) in the network and $w_{i,j}$ is the network edge weight between users i and j, and $\alpha : 0 < \alpha < 1$ is a parameter that determines the rate of diminishing returns on increased cover. Note that eqn. 2 is a monotone submodular function. We conduct our small experiments on the social network of 50 randomly selected communities (~ 500 nodes) from the 5000 largest communities in the YouTube social network (Feldman et al., 2015). For our larger experiments, we increase this to 2000 communities (~ 18000 nodes). We set $\alpha = 0.9$ for all experiments, and we draw the weights of edges in this network from the uniform distribution $U(1, 2)$.

- **Influence maximization on a social network.** In this application, we select a set of social network 'influencers' to post about a topic we wish to promote, and our objective is to select the set that achieves the greatest aggregate influence. We adopt the following random cover function: an arbitrary social network user has a small independent probability of being influenced by each influencer to whom she is connected, so we maximize the expected count of users who will be influenced by at least one influencer. The probability that a single user i will be influenced is:

$$\begin{cases} f_i(S) & = 1 \quad \text{for } i \in S \\ f_i(S) & = 1 - (1-p)^{|N_S(i)|} \quad \text{for } i \notin S \end{cases}$$

  where $|N_S(i)|$ is node i's count of neighbors who are in S. We set $p = 0.01$. We conduct our small experiments on the CalTech Facebook Network data set (Traud et al., 2012) of 769 Facebook users & 17000 edges, and we conduct our larger experiments on the Epinions data set of 27000 users and 100000 edges (Rossi & Ahmed, 2015).

C.5. Parallelization

We parallelize all algorithms via Message Passing Interface (MPI). We make this selection both because it is the industry standard, and also because it allows precise control over the architecture of parallel communication between processors as well as exactly what information is communicated between processors. This allows us to build efficient parallel architectures that minimize communication, such that our implementations are CPU-bound (i.e. query-bound). This property both permits fast implementations also aligns with the theoretical view of adaptive sampling algorithms, which assume that computation time is a function of rounds of queries rather than communication, data copying, etc. In contrast, simpler-to-use parallel libraries (e.g. $JobLib$) often communicate copies of all data to all processors at each communicative step, which may render implementations based on these simpler libraries both slower and also communication-bound.

C.6. AWS Hardware

While our MPI implementations of the algorithms are scalable to thousands of cores, we conduct all experiments on an *m5d.24xlarge* instance with 96 cores—the largest single instance currently available on AWS (computing on more cores requires launching an AWS cluster). This instance features Intel Xeon Platinum 8000 series (Skylake-SP) processors with sustained all core Turbo CPU clock speed of up to 3.1 GHz.

We select this hardware to ensure both the internal validity and external validity of our experiments. Specifically, with regard to internal validity, we note that if we instead had scaled up to a cluster of multiple instances, then communication times between cores in the same instance vs. across instances may differ to a greater extent. Because different algorithms require different amounts and structures of communication between processors, this might bias runtimes in unpredictable ways.

Second and more importantly, our goal in this paper is *not* to show that FAST is the fastest practical algorithm only on large scale state-of-the-art hardware (though larger-scale hardware would further improve the runtime advantage of FAST

over alternatives—see Section 4.1). Instead, our goal is to show that FAST is faster than alternatives even with modest hardware that is widely accessible to researchers, and these fast runtimes can be further improved on larger scale hardware. Specifically, we note that this *m5d.24xlarge* instance has far fewer cores than the n processors necessary to unlock the full speed potential of FAST, whereas adding more processors cannot accelerate PARALLEL-LTLG for most experiments due to the fact that PARALLEL-LTLG has sample complexity less than 95 for many values of k we tried.

C.7. Instance setup

We initialize the *m5d.24xlarge* instance with Amazon's Deep Learning AMI (Amazon Linux) Version 24.0. We install the *Open-MPI* MPI library on our AWS instance and run all experiments via ssh using $mpirun$ to launch and execute the experiments.

C.8. Measuring parallel runtimes.

We measure true parallel time in the following manner. First, before we start the runtime clock, all processors are initialized with a copy of the objective function and dataset for the experiment, which is followed by a call to a blocking parallel barrier ($comm.barrier()$). This forces the condition that no processor begins computations—and the clock does not start—until all processors are initialized. Runtime is then measured via MPI's parallel clock, $MPI.Wtime()$, from the moment this barrier is completed and the algorithm function is called. Upon the algorithm's completion, we use a blocking parallel barrier ($comm.barrier()$) call to all processors followed by a call to the parallel clock $MPI.Wtime()$. This ensures that a case where one processor finishes its part of the computations early does not result in an erroneously reported lower runtime.

We run all experiments on 95 cores of the 96-core instance. We deliberately leave one core free during all experiments so that the 95 cores conducting the experiment are not simultaneously scheduled to run a background task, which may result in a slowdown that would endanger the integrity of measured runtimes.

C.9. Overview of fast parallel implementations

For all algorithms, we implement several generally applicable and algorithm-specific optimizations. The intuition behind our generally applicable optimizations is to ensure that (1) implementations are optimized and vectorized such that all operations besides queries take negligible time (such that the algorithms are effectively query-bound); (2) communicative architectures between processors are designed to avoid superfluous communication; (3) we implement parallel reduces where possible to leverage these parallel architectures to further reduce computation; and (4) no algorithm ever queries the marginal value of an element when this marginal value is known to be 0, i.e. $f_T(x), x \in T$. Algorithm-specific optimizations are discussed below.

C.10. Fast parallel implementation of low-adaptivity algorithms

- **Amortized-Filtering**. The key optimizations we introduce to AMORTIZED-FILTERING are the binary search over `OPT` guesses, the tighter upper-bound guess for `OPT`, and the stopping condition described in Appendix C.2 above.

- **Binary-Search-Maximization**. BINARY-SEARCH-MAXIMIZATION tends to run significantly more loop iterations than other benchmarks, so optimizations that reduce these loop iterations result in significant speedup. We note that two loops of this algorithm loop over indices, where indices are calculated as elements of a geometric sequence then rounded to integers. This process results in the algorithm looping over numerous redundant indices, as many unique floating point numbers from these sequences round to the same whole numbers. We therefore achieve large speedups by precomputing these sequences and looping only over unique indices. In addition to this optimization, we also note that the REDUCED-MEAN subroutine (which is responsible for the vast majority of computation time) requires the processors to parallel-compute a fraction of elements that exceed a threshold. A naive approach would be to computing marginal values in parallel, $gather$ or $allgather$ all of these values (i.e. each processor communicates its share of values to all other processors), and then compute this fraction locally. However, our optimized approach uses a fast parallel reduction where each processor computes its local fraction, then a fast parallel reduce using $MPI.Sum()$ such that (1) processors need only communicate this fraction (float) instead of the entire vector of elements, and (2) the global fraction is then rapidly computed via a parallel reduce. More advanced MPI architectures such as these result in meaningfully lower runtimes in practice, particularly when using relatively fast-to-compute objective functions.

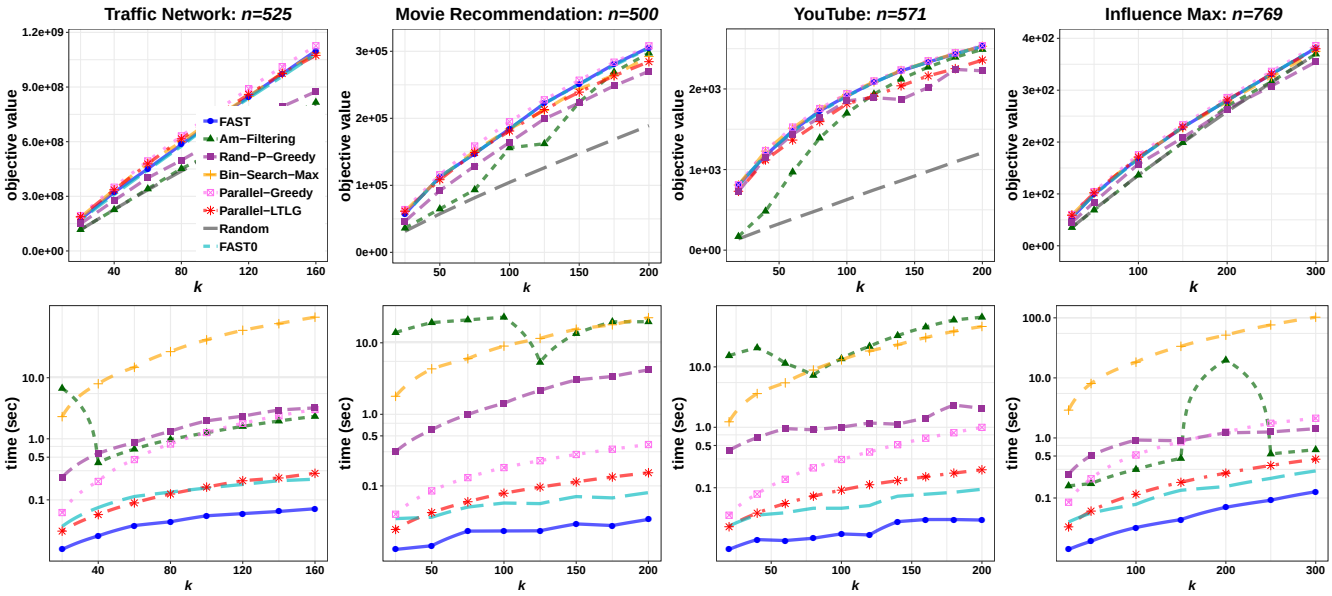

Figure 6. FAST0 *(light blue) vs. low-adaptivity algorithms.*

- **Randomized-Parallel-Greedy**. The key decision when implementing RANDOMIZED-PARALLEL-GREEDY is how to choose guesses for the step size δ, which is the probability with which we randomly add high-value elements to the solution. This is important because the majority of RANDOMIZED-PARALLEL-GREEDY's runtime is taken up by calls to the multilinear extension that are made in order to choose δ. Specifically, recall that in each iteration, RANDOMIZED-PARALLEL-GREEDY calls the multilinear extension to search for the maximum δ that obeys certain conditions. Here, if we implement RANDOMIZED-PARALLEL-GREEDY such that it tests more guesses (i.e. more closely spaced) guesses for δ, then we may find a δ that is closer to the true maximum δ, but this uses additional calls to the multilinear extension that slow runtimes. We therefore precompute just n guesses for δ as $[1/n, 2/n, \ldots]$, iterate over each of these, and set δ to the rightmost value that does not violate the conditions. Based on this choice, the minimum value we attempt for δ will increase $|S|$ by one element in expectation when all elements are high-valued. By using this relatively large $1/n$ stepsize between subsequent guesses for δ, we reduce solution values in practice, but we accelerate the algorithm (thus providing a more difficult runtime benchmark for FAST). Before making this choice, we also experimented with geometrically-spaced guesses for δ, but found that this resulted in a significant further reduction in performance and also that it caused the algorithm to attempt many very small options for δ that were unlikely to result in adding a single element.

C.11. Fast parallel implementation of LAZIER THAN LAZY GREEDY

- **Parallel-LTLG.** Designing a fast implementation of PARALLEL-LTLG is nontrivial because at each iteration, we want to attempt a lazy update (which requires a single query), but in the event that this lazy update fails, we do not want to complete the iteration any slower than if we had not attempted the lazy update. Put differently, the lazy update attempted at each iteration should result in a speedup when it succeeds, but never in a slowdown when it fails, such that PARALLEL-LTLG is strictly faster than STOCHASTIC-GREEDY.

  To accomplish this, we adopt the following optimized parallel architecture for PARALLEL-LTLG. At each of k rounds, the root processor draws a set R of sample elements from remaining elements in the ground set $X \backslash S$. The root process broadcasts these sample elements to the other processors. Then, all processors simultaneously make a single query: the root process queries the marginal value of the best element according to previous (lazy) marginal values, and remaining processors each query a single other element from the sample. If the root process succeeds in finding a lazy update with its single query, it communicates this to all processors, and all processors add this element to S and move to the next iteration. If the lazy update fails, then the c processors have *already* completed c marginal value queries of the samples (so no time is lost). They then simply each compute $1/c$ of the $(|R| - c)$ *remaining* samples' marginal

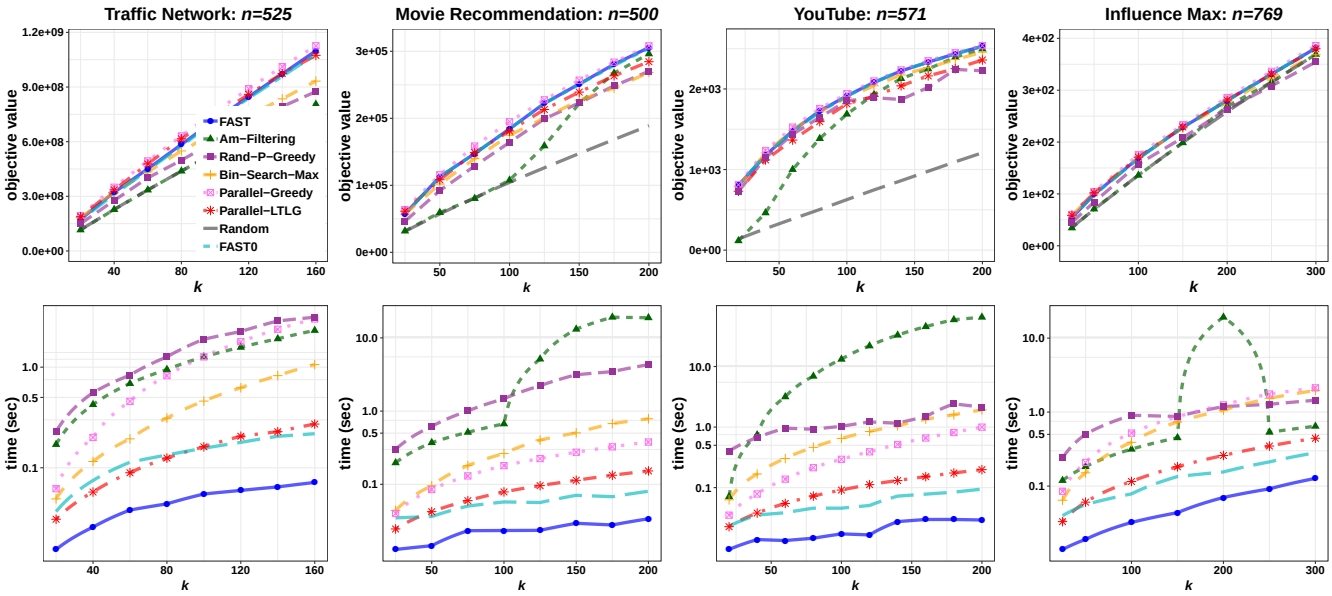

Figure 7. FAST *(blue) &* FAST0 *(light blue) vs. low-adaptivity algorithms run on a single guess for* OPT.

values, communicate them, add the best element to S, and move to the next iteration.

C.12. Experiment set 1 results: additional discussion

Our goal in this section is to show that FAST still achieves faster runtimes than benchmarks (1) even when we turn off its lazy updates and (2) even when we run each low-adaptivity benchmark on just a single 'good' guess for OPT. Figure 6 plots all real data experiments from Section 4.1, but with an added line for FAST0 (*light blue*). FAST0 is identical to FAST, but without lazy updates. Across all of these experiments, FAST0 was 50 to 500 times faster than BINARY-SEARCH-MAXIMIZATION, 5 to 50 times faster than RANDOMIZED-PARALLEL-GREEDY, and 2 to 700 times faster than AMORTIZED-FILTERING.

We note that across these experiments, FAST0 (without lazy updates) also achieved lower runtimes than PARALLEL-LTLG on nearly all objectives and values of k.

We also compare FAST and FAST0 to low-adaptivity benchmarks when each of the latter is run on just a single guess for OPT. Specifically, we again rerun all real data experiments from Section 4.1, but for each low-adaptivity benchmark, we set a single guess of OPT to $v = \max_{|S|\leq k} \sum_{a\in S} f(a)$ of the k highest valued singletons (the upper-bound guess used in FAST). Figure 7 plots solution values and runtimes for these experiments. FAST was 3 to 66 times faster than BINARY-SEARCH-MAXIMIZATION, 11 to 127 times faster than RANDOMIZED-PARALLEL-GREEDY, and 5 to 2200 times faster than AMORTIZED-FILTERING.

Finally, we note that across all of the low-adaptivity experiments (Experiments Set 1), the non-linearities in the time plots of benchmark algorithms are due to the thresholding techniques used by the algorithms as k increases, and not to the variance of the randomized algorithms. While for each objective and each value of k, the figures report the mean objective value and runtime of 5 independent trials of each algorithm, we note that algorithms' runtimes tend to be quite consistent over multiple trials for a given value of k.

C.13. Experiment set 2 results: additional discussion

Figure 8 plots queries used by FAST and PARALLEL-LTLG for the 8 objectives in *Experiment set 2* (each point is the mean of 5 trials). When counting queries for PARALLEL-LTLG, we count rounds where a lazy update succeeded as single query rounds despite the fact that in this case PARALLEL-LTLG uses $\max[c, s]$ queries where c is the number of processors and s is its sample complexity per round. This choice allows us to compare the queries used by FAST vs. PARALLEL-LTLG for any k and determine whether FAST used fewer queries than *serial* $LTLG$ would use. Note that this occurs for various k in

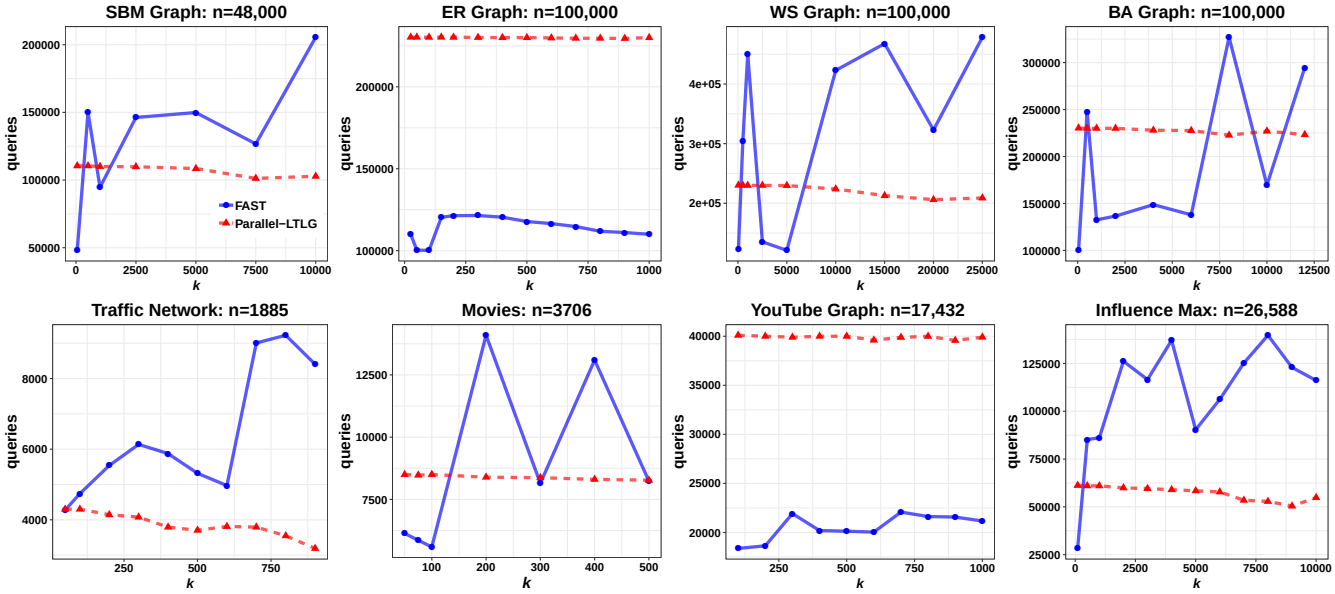

Figure 8. Experiment Set 2: Queries used by FAST *(blue) vs.* PARALLEL-LTLG *(red).*

7 of the 8 experiments. We also note that in practice, serial LTLG is often slower than FAST even when both perform the same number of queries due to the fact that FAST performs more queries at a time (i.e. more per round for fewer rounds), which is often computationally faster.

C.14. Additional discussion of parallel runtime & speedup plots (Fig. 5

We note that in Fig. 5 *left* (1 Processor Runtime of LTLG vs. FAST), the runtime of LTLG increases linearly with k. The reason is that, for certain objectives, the runtime of computing $f(S)$ increases in $|S|$. For this objective, $f(S)$ entails summing over $|S|$ rows in a network adjacency matrix and this causes the runtime of the algorithms to increase in k. For some other objectives we could avoid this time increase with optimizations. An additional advantage of FAST in objectives with this time increase is that, even with lazy updates, LTLG's queries-per-round are (roughly) constant, but FAST typically does most of its queries in early rounds when $|S|$ is still small, then avoids later queries with its speedups, so it can be faster than LTLG on 1 processor even when it uses more queries. For example, in Fig. 5 *left*, FAST used 75% of its total queries by round 2 of 11.