Fast Greedy Algorithms in MapReduce and Streaming

Ravi Kumar*
Google
Mountain View, CA
ravi.k53@gmail.com

Benjamin Moseley*†
Toyota Technological Institute
Chicago, IL
moseley@ttic.edu

Sergei Vassilvitskii*
Google
Mountain View, CA
sergeiv@google.com

Andrea Vattani*
University of California
San Diego, CA
avattani@cs.ucsd.edu

ABSTRACT

Greedy algorithms are practitioners' best friends—they are intuitive, simple to implement, and often lead to very good solutions. However, implementing greedy algorithms in a distributed setting is challenging since the greedy choice is inherently sequential, and it is not clear how to take advantage of the extra processing power.

Our main result is a powerful sampling technique that aids in parallelization of sequential algorithms. We then show how to use this primitive to adapt a broad class of greedy algorithms to the MapReduce paradigm; this class includes maximum cover and submodular maximization subject to p-system constraints. Our method yields efficient algorithms that run in a logarithmic number of rounds, while obtaining solutions that are arbitrarily close to those produced by the standard sequential greedy algorithm. We begin with algorithms for modular maximization subject to a matroid constraint, and then extend this approach to obtain approximation algorithms for submodular maximization subject to knapsack or p-system constraints. Finally, we empirically validate our algorithms, and show that they achieve the same quality of the solution as standard greedy algorithms but run in a substantially fewer number of rounds.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

Distributed computing, Algorithm analysis, Approximation algorithms, Greedy algorithms, Map-reduce, Submodular function

*Part of this work was done while the author was at Yahoo! Research.

†Partially supported by NSF grant CCF-1016684.

SPAA'13, June 23–25, 2013, Montreal, Quebec, Canada.

1. INTRODUCTION

Greedy algorithms have been very successful in practice. For a wide range of applications they provide good solutions, are computationally efficient, and are easy to implement. A typical greedy algorithm repeatedly chooses an action that maximizes the objective given the previous decisions that it has made. A common application of greedy algorithms is for (sub)modular maximization problems, such as the MAXCOVER[1] problem. For this rich class of problems, greedy algorithms are a panacea, giving near-optimal solutions.

Submodular maximization. Submodular maximization has received a significant amount of attention in optimization; see [8] and [38] for pointers to relevant work. Examples of its numerous applications include model-driven optimization [21], skyline representation [36], search result diversification [1, 9], string transformations [2], social networks analysis [11, 12, 23, 28, 31], the generalized assignment problem [8], and auction theory [3]. In submodular maximization, we are given a submodular function f and a universe U, with the goal of selecting a subset $S \subseteq U$ such that $f(S)$ is maximized. Typically, S must satisfy additional feasibility constraints such as cardinality, knapsack, matroid, or p-systems constraints (see Section 2).

Maximizing a submodular function subject to these types of constraints generalizes many well-known problems such as the maximum spanning tree problem (modular maximization subject to a single matroid constraint), the maximum weighted matching problem in general graphs (modular maximization subject to a 2-system), and MAXCOVER (submodular maximization subject to a cardinality constraint). For this class of problems there is a natural greedy algorithm: iteratively add the best feasible element to the current solution. This simple algorithm turns out to be a panacea for submodular maximization. It is optimal for modular function maximization subject to one matroid constraint [17] and achieves a $(1/p)$-approximation for p matroid constraints [26]. For the submodular coverage problem, it achieves a $(1 - 1/e)$-approximation for a single uniform matroid constraint [33], and a $(1/(p+1))$-approximation for p matroid constraints [8, 20]; for these two cases, it is known that it is NP-hard to achieve an approximation factor better than $(1 - 1/e)$ and $\Omega(\frac{\log p}{p})$, respectively [19, 25].

The case of big data. Submodular maximization arises in data management, online advertising, software services, and online inventory control domains [7, 13, 14, 30, 35], where the data sizes are

[1]In the MAXCOVER problem we are given a universe U and a family of sets $\mathcal{S}$. The goal is to find a set $S' \subset \mathcal{S}$ of size k that maximizes the total union, $|\cup_{X \in S'} X|$.

routinely measured in tens of terabytes. For these problem sizes, the MapReduce paradigm [16] is standard for large-scale parallel computation. An instantiation of the BSP model [37] for parallel computing, a MapReduce computation begins with data randomly partitioned across a set of machines. The computation then proceeds in rounds, with communication between machines taking place only between successive rounds. A formal model of computation for this paradigm was described in [27]. It requires both the number of machines and the memory per machine to be sublinear in the size of the input, and looks to reduce the total number of rounds necessary.

The greedy algorithm for submodular maximization does not appear to be useful in this setting due to its inherently sequential nature. The greedy choice made at each step critically depends on the its previous actions, hence a naive MapReduce implementation would perform one action in each round, and gain little advantage due to parallelism. A natural question arises: *is it possible to realize an efficient version of the greedy algorithm in a distributed setting*? The challenge comes from the fact that to reduce the number of rounds, one is forced to add a large number of elements to the solution in every round, even though the function guiding the greedy selection can change dramatically with every selection. Moreover, the selection must be done in parallel, without any communication between the machines. An affirmative answer to this question would open up the possibility of using these algorithms for large-scale applications. In fact, the broader question looms: which greedy algorithms are amenable to MapReduce-style parallelization?

There have been some recent efforts to find efficient algorithms for submodular maximization problems in the parallel setting. For the MAXCOVER problem Chierichetti et al. [13] obtained a MapReduce algorithm that achieves a factor of $1-1/e-\epsilon$ by adapting an algorithm for parallel set cover of Berger et al. [6]; the latter was improved recently by Blelloch et al. [7]. Cormode et al. [14] also consider improving the running time of the natural greedy algorithm for large datasets in the sequential setting. For the densest subgraph problem, Bahmani et al. [4] obtained a streaming and MapReduce algorithm by adapting a greedy algorithm of Charikar [10]. Lattanzi et al. [30] considered the vertex cover and maximal matching problems in MapReduce. The recent work Ene et al. [18] adapted the well-known greedy algorithm for the k-center problem and the local search algorithm for the k-median problem to MapReduce. Beyond these, not much is known about maximizing general submodular functions under matroid/knapsack constraints—in particular the possibility of adapting the greedy algorithm—in the MapReduce model.

1.1 Main contributions

We show that a large class of greedy algorithms for non-negative submodular maximization with hereditary constraints can be efficiently realized in MapReduce. (We note that all of our results can be extended to the streaming setting, but defer the complete proofs to a full version of this paper.) We do this by defining an approximately greedy approach that selects a feasible element with benefit at least $1/(1+\epsilon)$ of the maximum benefit. This ϵ-greedy algorithm works almost as well as the standard algorithm for many important problems (Theorem 5), but gives the algorithm designer some flexibility in selecting which element to add to the solution.

We then show how to simulate the ϵ-greedy algorithm in a distributed setting. If Δ is a bound on the maximum increase in the objective any element can offer and k is the size of the optimal solution, then the MapReduce algorithm runs in $O(\frac{1}{\epsilon\delta}\log\Delta)$ rounds when the memory per machine is $O(kn^\delta)$ for any $\delta > 0$. In fact, all of the algorithms we propose display a smooth tradeoff between the maximum memory per machine and the total number of rounds before the computation finishes. Specifically, when each machine has roughly $O(kn^\delta)$ memory, the computation takes $O(1/\delta)$ rounds. We present a summary of the results in Table 1. Recall that any simulation of PRAM algorithms requires $\Omega(\log n)$ rounds, we improve on this bound whenever the memory per machine is $\omega(k)$.

At the heart of the simulation lies the main technical contribution of this paper. We describe a sampling technique called SAMPLE&PRUNE that is quite powerful for the MapReduce model. The high level idea is to find a candidate solution to the problem on a sample of the input and then use this intermediate solution to repeatedly prune the elements that can no longer materially contribute to the solution. By repeating this step we can quickly home in on a nearly optimal (or in some cases optimal) solution. This *filtering* approach was used previously in [30] for the maximal matching problem, where it was argued that a matching on a sample of edges can be used to drop most of the edges from consideration. Here we abstract and extend the idea and apply to a large class of greedy algorithms. Unlike the maximal matching case, where it is trivial to decide when an element can be discarded (it is adjacent to an already selected edge), this task is far less obvious for other greedy algorithms such as for set cover. We believe that the technique is useful for scaling other algorithms in the MapReduce model and is one of the first *rules of thumb* for this setting.

Improvements on previous work. While we study parallelizations of greedy algorithms and give the first results for general submodular function maximization, for some specific problems, we improve over previous MapReduce algorithms provided that the memory per machine is polynomial. In [13] an $O(\text{poly}\log(n,\Delta))$-round $(1-1/e-\epsilon)$-approximate algorithm was given for the MAXCOVER problem. In this work, the approximate greedy algorithm reduces the number of rounds to $O(\log\Delta)$ and achieves the same approximation guarantees. Further, the algorithm for submodular maximization subject to 1-knapsack constraint implies an even faster *constant*-round algorithm that achieves a slightly worse approximation ratio of $1/2-\epsilon$. For the maximum weighted matching problem previously a $1/8$-approximate constant-round algorithm was known [30]. Knowing that maximum matching can be represented as a 2-system, our algorithm for p-systems implies a constant-round algorithm for the maximum matching problem that achieves an improved approximation of $\frac{1}{3+\epsilon}$.

Organization. We begin in Section 2 by introducing necessary definitions of the problems and the MapReduce model. In Section 3 we introduce the SAMPLE&PRUNE technique and bound its guarantees. Then we apply this technique to simulate a broad class of greedy algorithms in Section 4. In Section 5 we introduce an $O(1)$ round algorithm for maximizing a modular function subject to a matroid constraint. In Section 6 we consider maximizing a submodular function subject to knapsack or matroid constraints. We present an experimental evaluation of our algorithm in Section 8.

2. PRELIMINARIES

Let U be a universe of $n = |U|$ elements, let $f : 2^U \to \mathbb{R}^+$ be a function, and let $\mathcal{I} \subseteq 2^U$ be a given family of feasible solutions. We are interested in solving the following optimization problem:

$$\max\{f(S) \mid S \in \mathcal{I}\}. \tag{1}$$

We will focus on the case when f and $\mathcal{I}$ satisfy some nice structural properties. For simplicity, we use the notation $f'_S(u)$ to denote the incremental value in f of adding u to S, i.e., $f'_S(u) = f(S\cup\{u\}) - f(S)$.

Problem		Approximation	Rounds	Reference
Objective	Constraint			
Modular	1-matroid	1	$O(1/\delta)$	Section 5
Submodular	1-matroid	$1/(2+\epsilon)$	$O(\frac{1}{\epsilon\delta}\log\Delta)$	Section 4
	1-knapsack	$1/2-\epsilon$	$O(1/\delta)$	Section 6
	d-knapsacks	$\Omega(1/d)$		
	p-system	$\frac{1}{p+1}\lceil 1/\delta\rceil^{-1}$		
		$\frac{1}{p+1+\epsilon}$	$O(\frac{1}{\epsilon\delta}\log\Delta)$	Section 4

Table 1: A summary of the results. Here, n denotes the total number of elements and Δ, the maximum change of the function under consideration. Our algorithms use $O(n/\mu)$ machines with μ, the memory per machine, $\mu = O(kn^{\delta}\log n)$, where k is the cardinality of the optimum solution.

DEFINITION 1 (MODULAR AND SUBMODULAR FUNCTIONS). *A function $f : 2^U \to \mathbb{R}^+$ is said to be* submodular *if for every $S' \subseteq S \subseteq U$ and $u \in U \setminus S$, we have $f'_{S'}(u) \geq f'_S(u)$; it is said to be* modular *if $f'_{S'}(u) = f'_S(u)$.*

A function f is said to be *monotone* if $S' \subseteq S \implies f(S) \geq f(S')$. A family $\mathcal{I}$ is *hereditary* if it is downward closed, i.e., $A \in \mathcal{I} \wedge B \subseteq A \implies B \in \mathcal{I}$. The greedy simulation framework we introduce will only be restricted to submodular functions for a hereditary feasible family. We now define more specific types on constraints on the feasible family.

DEFINITION 2 (MATROID). *The pair $\mathcal{M} = (U, \mathcal{I})$ is a* matroid *if $\mathcal{I}$ is hereditary and satisfies the following* augmentation *property: $\forall A, B \in \mathcal{I}, |A| < |B| \implies \exists u \in B \setminus A$ such that $A \cup \{u\} \in \mathcal{I}$.*

Given a matroid $(U, \mathcal{I})$, a set $A \subseteq U$ is called *independent* if and only if $A \in \mathcal{I}$. Throughout the paper, we will often use the terms feasible and independent interchangeably, even for more general families $\mathcal{I}$ of feasible solutions. The *rank* of a matroid is the number of elements in a maximal independent set.

We next recall the notion of p-systems, which is a generalization of intersection of p matroids (e.g., see [8, 26, 29]). Given $U' \subseteq U$ and a hereditary family $\mathcal{I}$, the maximal independent sets of $\mathcal{I}$ contained in U' are given by $\mathcal{B}(U') = \{A \in \mathcal{I} \mid A \subseteq U' \wedge \nexists A' \in \mathcal{I}, A \subsetneq A' \subseteq U'\}$.

DEFINITION 3 (p-SYSTEM). *The pair $\mathcal{P} = (U, \mathcal{I})$ is a* p-system *if $\mathcal{I}$ is hereditary and for all $U' \subseteq U$, we have $\max_{A \in \mathcal{B}(U')} |A| \leq p \cdot \min_{A \in \mathcal{B}(U')} |A|$.*

Given $\mathcal{F} = (U, \mathcal{I})$, where $\mathcal{I}$ is hereditary, we will use the notation $\mathcal{F}[X]$, for $X \subseteq U$, to denote the pair $(X, \mathcal{I}[X])$, where $\mathcal{I}[X]$ is the restriction of $\mathcal{I}$ to the sets over elements in X. We observe that if $\mathcal{F}$ is a matroid (resp., p-system), then we have that $\mathcal{F}[X]$ is a matroid (resp., p-system) as well.

2.1 Greedy algorithms

A natural way to solve (1) is to use the following sequential *greedy* algorithm: start with an empty set S and grow S by iteratively adding the element $u \in U$ such that greedily maximizes the benefit:

$$u = \arg\max_{\substack{u' \in U \\ S \cup \{u'\} \in \mathcal{I}}} f'_S(u').$$

It is known that this greedy algorithm obtains a $(1-1/e)$ approximation for the maximum coverage problem, a $(1/p)$-approximation for maximizing a modular function subject to p-system constraints [26] and a $(1/(p+1))$-approximation for maximizing a submodular function [8, 20]. Recently, for $p = 1$, Calinescu et al. [8] obtained a $(1-1/e)$-approximation algorithm for maximizing a submodular function; this result generalizes the maximum coverage result.

We now define an approximate greedy algorithm, which is a modification of the standard greedy algorithm. Let $0 \leq \epsilon \leq 1$ be a fixed constant.

DEFINITION 4 (ϵ-GREEDY ALGORITHM). *Repeatedly, add an element u to the current solution S such that $S \cup \{u\} \in \mathcal{I}$ and $f'_S(u) \geq \frac{1}{1+\epsilon}\left(\max_{\substack{u' \in U \\ S \cup \{u'\} \in \mathcal{I}}} f'_S(u')\right)$, with ties broken in an arbitrary but consistent way.*

The usefulness of this definition is evident from the following.

THEOREM 5. *The ϵ-greedy algorithm achieves a: (i) $\frac{1}{2+\epsilon}$ approximation for maximizing a submodular function subject to a matroid constraint; (ii) $\frac{1-1/e}{1+\epsilon}$ approximation for maximizing a submodular function subject to choosing at most k elements; and (iii) $\frac{1}{p+1+\epsilon}$ (resp., $\frac{1}{p+\epsilon}$) approximation for maximizing submodular (modular) function subject to a p-system.*

The proof follows by definition and from the analysis of Calinescu et al. [8]. Finally, without loss of generality, we assume that for every $S \subseteq U$ and $u \in U$ such that $f'_S(u) \neq 0$, we have $1 \leq f'_S(u) \leq \Delta$, i.e., Δ represents the "spread" of the non-zero incremental values.

2.2 MapReduce

We describe a high level overview of the MapReduce computational model; for more details see [16, 18, 27, 30]. In this setting all of the data is represented by $\langle$`key`; `value`$\rangle$ pairs. For each pair, the `key` can be seen as the logical address of the machine that contains the `value`, with pairs sharing the same `key` being stored on the same machine.

The computation itself proceeds in rounds, which each round consisting of a *map*, *shuffle*, and *reduce* phases. Semantically, the map and shuffle phases distribute the data, and the reduce phase performs the computation. In the map phase, the algorithm designer specifies the desired location of each value by potentially changing its key. The system then routes all of the values with the same key to the same machine in the shuffle phase. The algorithm designer specifies a reduce function that takes as input all $\langle$`key`; `value`$\rangle$ pairs with the same key and outputs either the final solution or a set of $\langle$`key`; `value`$\rangle$ pairs to be mapped in a subsequent MapReduce round.

Karloff et al. [27] introduced a formal model, designed to capture the real world restrictions of MapReduce as faithfully as possible. Specifically, their model insists that the total number of machines available for a computation of size n is at most $n^{1-\epsilon}$, with each

machine having $n^{1-\epsilon}$ amount of memory, for some constant $\epsilon > 0$. The overall goal is to reduce the number of rounds necessary for the computation to complete. Later refinements on the model [18, 30, 34] allow for a specific tradeoff between the memory per machine the number of machines and the total number of rounds.

The MapReduce model of computation is incomparable to the traditional PRAM model where an algorithm can use a polynomial number of processors that share an unlimited amount of memory. The authors of [27] have shown how to simulate T-step EREW PRAM algorithms in $O(T)$ rounds of MapReduce. This result has subsequently been extended by [22] to CRCW algorithms with an additional overhead of $\log_\mu M$, where μ is the memory per machine and M is the aggregate memory used by the PRAM algorithm. We note that only a few special cases of the problems we consider have efficient PRAM algorithms and, the MapReduce simulation of these cases requires at least logarithmic factor larger running time than our algorithms.

3. SAMPLE&PRUNE

In this section we describe a primitive that will be used in our algorithms to progressively reduce the size of the input. A similar *filtering* method was used in [30] for maximal matchings and in [18] for clustering. Our contribution is to abstract this method and significantly expand the scope of its applications.

At a high level, the idea is to identify a large subset of the data that can be safely discarded without changing the value of the optimum solution. As a concrete example, consider the MAXCOVER problem. We are given a universe of elements U, and a family of subsets $\mathcal{S}$. Given an integer k, the goal is to choose at most k subsets from $\mathcal{S}$ such that the largest number of elements are covered. Say that the sets in U are $S_1 = \{a, b, c\}, S_2 = \{a\}, S_3 = \{a, c\}$, and $S_4 = \{b, d\}$. If S_1 has been selected as part of the solution, then both S_2 and S_3 are redundant and can be clearly removed thus reducing the problem size. We refer to S_1 above as a *seed* solution. We show for a large family of optimization functions that a seed solution computed on a *sample* of all of the elements can be used to reduce the problem size by a factor proportional to the size of the sample.

Formally, consider a universe U and a function $\mathcal{G}_k$ such that for $A \subseteq U$, the function $\mathcal{G}_k(A)$ returns a subset of A of size at most k and $\mathcal{G}_k(A) \subseteq \mathcal{G}_k(B)$, for $A \subseteq B$. The algorithm SAMPLE&PRUNE begins by running $\mathcal{G}_k$ on a sample of the data to obtain a seed solution S. It then examines every element $u \in U$ and removes it if it appears to be redundant under $\mathcal{G}_k$ given S.

SAMPLE&PRUNE$(U, \mathcal{G}_k, \ell)$
1: $X \leftarrow$ sample each point in U with probability $\min\{1, \frac{\ell}{|U|}\}$
2: $S \leftarrow \mathcal{G}_k(X)$
3: $M_S \leftarrow \{u \in U \setminus S \mid u \in \mathcal{G}_k(S \cup \{u\})\}$
4: **return** (S, M_S)

The algorithm is sequential, but can be easily implemented in the MapReduce setting, since both lines (1) and (3) are trivially parallelizable. Further, the set X at line (2) has size at most $O(\ell \log n)$ with high probability. Therefore, for any $\ell > 0$ the algorithm can be implemented in two rounds of MapReduce using $O(n/\mu)$ machines each with $\mu = O(\ell \log n)$ memory.

Finally, we show that the number $|M_S|$ of non-redundant elements is smaller than $|U|$ by a factor of ℓ/k. Thus, a repeated call to SAMPLE&PRUNE terminates after $O(\log_{\ell/k} n)$ iterations, as shown in Corollary 7.

LEMMA 6. *After one round of* SAMPLE&PRUNE *with* $\ell = kn^\delta \log n$, *we have* $\Pr\left[|M_S| \le 2n^{1-\delta}\right] \ge 1 - 2n^{-k}$ *when* $\mathcal{G}_k(A)$ *is a function that returns a subset of* A *of size at most* k *and* $\mathcal{G}_k(A) \subseteq \mathcal{G}_k(B)$, *for* $A \subseteq B$.

PROOF. First, observe that when $n \le \ell$, we have $M_S = \emptyset$ and hence we are done. Therefore, assume $n > \ell$. Call the set S obtained in step 2 a *seed* set. Fix a set S and let $\mathcal{E}_S$ be the event that S was selected as the seed set. Note that when $\mathcal{E}_S$ happens, by the hereditary property of $\mathcal{G}_k$, none of the elements in M_S is part of the sample X, since any element $x \in X$ discarded in step 2 would be discarded in step 3 as well. Therefore, for each S such that $|M_S| \ge m = (2k/\ell)n \log n$, we have $\Pr[\mathcal{E}_S] \le \left(1 - \frac{\ell}{n}\right)^{|M_S|} \le \left(1 - \frac{\ell}{n}\right)^m \le n^{-2k}$.

Since each seed set S has at most k elements, there are at most $2n^k$ such sets, and hence $\sum_{S:|M_S|\ge m} n^{-2k} \le 2n^k \cdot n^{-2k} = 2n^{-k}$ □

COROLLARY 7. *Suppose we run* SAMPLE&PRUNE *repeatedly for* T *rounds, that is: for* $1 \le i \le T$, *let* $(S_i, M_i) =$ SAMPLE&PRUNE$(M_{i-1}, \mathcal{G}_k, \ell)$, *where* $M_0 = U$. *If* $\ell = kn^\delta \log n$ *and* $T > 1/\delta$, *then we have* $M_T = \emptyset$, *with probability at least* $1 - 2Tn^{-k}$.

PROOF. By Lemma 6, assuming that $M_i \le n(2n^{-\delta})^i$ we can prove that $M_{i+1} \le n\left(2n^\delta\right)^{i+1}$ with probability at least $1 - 2n^{-k}$. The claim follows from a union bound. □

We now illustrate how Corollary 7 generalizes some prior results [18, 30]. Note that the choice of the function $\mathcal{G}_k$ affects both the running time and the approximate ratio of the algorithm. Since Corollary 7 guarantees convergence, the algorithm designer can focus on judiciously selecting a $\mathcal{G}_k$ that offers approximation guarantees for the problem at hand. For finding maximal matchings, Lattanzi et al. [30] set $\mathcal{G}_k(X, S)$ to be the maximal matching obtained by streaming all of the edges in X, given that the edges in S have already been selected. Setting $k = n/2$, as at most k edges can ever be in a maximum matching, and the initial edge size to m, we see that using nm^δ memory, the algorithm converges after $O(\frac{1}{\delta})$ rounds. For clustering, Ene et al. [18] first compute the value v of the distance of the (roughly) n^δth furthest point in X to a predetermined set S and the function $\mathcal{G}_k(X, S)$ returns all of the points that have distance to S further than v. Since $\mathcal{G}_k$ returns at most $k = n^\delta$ points, using $n^{2\delta}$ memory, the algorithm converges after $O(\frac{1}{\delta})$ rounds.

In addition to these two examples, we will show below how to use the power of SAMPLE&PRUNE to parallelize a large class of greedy algorithms.

4. GREEDY ALGORITHMS

In this section we consider a generic submodular maximization problem with hereditary constraints. We first introduce a simple scaling approach that simulates the ϵ-greedy algorithm. The scaling algorithm proceeds in phases: in phase $i \in \{1, \ldots, \log_{1+\epsilon} \Delta\}$, only the elements that improve the current solution by an amount in $[\frac{\Delta}{(1+\epsilon)^{i+1}}, \frac{\Delta}{(1+\epsilon)^i})$ are considered (recall that no element can improve the solution by more than Δ). These elements are added one at a time to the solution as the algorithm scans the input. We will show that the algorithm terminates after $O(\frac{\log \Delta}{\epsilon})$ iterations, and effectively simulates the ϵ-greedy algorithm. We note that somewhat similar scaling ideas have proven to be useful for submodular problems such as in [5, 24]. We first describe the sequential algorithm (GREEDYSCALING) and then show how it can be parallelized.

GREEDYSCALING
1: $S \leftarrow \emptyset$
2: **for** $i = 1, \ldots, \log_{1+\epsilon} \Delta$ **do**
3: **for all** $u \in U$ **do**
4: **if** $S \cup \{u\} \in \mathcal{I}$ **and** $f'_S(u) \geq \frac{\Delta}{(1+\epsilon)^i}$ **then**
5: $S \leftarrow S \cup \{u\}$
6: **end if**
7: **end for**
8: **end for**

LEMMA 8. *For the submodular maximization problem with hereditary constraints,* GREEDYSCALING *implements the ϵ-greedy method.*

PROOF. Given the current solution, call an element u *feasible* if $S \cup \{u\} \in \mathcal{I}$. We use induction to prove the following claim: at the beginning of phase i, the marginal benefit of any feasible element is at most $\frac{\Delta}{(1+\epsilon)^{i-1}}$. The base case is clear. Suppose the claim is true at the beginning of phase j. At the end of this phase, no feasible element with a marginal benefit of more than $\frac{\Delta}{(1+\epsilon)^j}$ to the solution remains. By the hereditary property of $\mathcal{I}$, the set of feasible elements at the end of the phase is a subset of feasible elements at the beginning and, by submodularity of f, the marginal benefit of any element could only have decreased during the phase. Therefore, the marginal benefit of any element added by GREEDYSCALING is within $(1+\epsilon)$ of that of the best element, completing the proof. □

For realizing in MapReduce, we use SAMPLE&PRUNE in every phase of the outer **for** loop to find all elements with high marginal values. Recall that SAMPLE&PRUNE takes a function $\mathcal{G}_k : 2^U \rightarrow U^k$ as input, and returns a pair of sets (S, M). Note that k is a bound on the solution size. We will show how to define $\mathcal{G}_k$ in each phase so that we can emulate the behavior of the scaling algorithm. Our goal is to ensure that every element not returned by SAMPLE&PRUNE was rightfully discarded and does not need to be considered in this round.

Let $\tau_i = \frac{\Delta}{(1+\epsilon)^i}$ be the threshold used in the ith round. Let $\mathcal{G}_{S,i} : 2^U \rightarrow U^k$ be the function implemented in lines (3) through (7) of GREEDYSCALING during the ith iteration of the outer loop. In other words, the function $\mathcal{G}_{S,i}(A)$ maintains a solution S', and for every element $a \in A$ adds it to the solution if $f'_{S \cup S'}(a) \geq \tau_i$ (the marginal gain is above the threshold), and $S \cup S' \cup \{a\} \in \mathcal{I}$ (the resulting solution is feasible). Note that the procedure of $\mathcal{G}_{S,i}$ can change in each iteration of the loop since S can change and $\mathcal{G}_{S,i}$ may return a set of size less than k, but never larger than k. Consider the algorithm GREEDYSCALINGMR that repeatedly calls SAMPLE&PRUNE while adding the sampled elements into the solution set.

GREEDYSCALINGMR(Phase i)
1: $S \leftarrow \emptyset, M \leftarrow U$
2: **while** $M \neq \emptyset$ **do**
3: $(S', M) \leftarrow$ SAMPLE&PRUNE$(M, \mathcal{G}_{S,i}, \ell)$
4: $S \leftarrow S \cup S'$
5: **end while**
6: **return** S

Observe that every element added to S has a marginal value of at least τ_i by definition of $\mathcal{G}_{S,i}$. We show the converse below.

LEMMA 9. *Let S be the result of the ith phase of* GREEDYSCALINGMR *on U. For any element $u \notin S$ either $S \cup \{u\} \notin \mathcal{I}$ or $f'_S(u) \leq \tau_i$.*

PROOF. Suppose not, and consider the iteration of the algorithm immediately before u is removed from M. We have sets S, M with $u \in M$, but $u \notin S' \cup M$ after an additional iteration of SAMPLE&PRUNE. If u were removed, then $u \notin \mathcal{G}_{S,i}(S' \cup \{u\})$ and therefore the addition of S' to the solution either resulted in u being infeasible (i.e., $S \cup S' \cup \{u\} \notin \mathcal{I}$) or its marginal benefit becoming inadequate (i.e., $f'_{S \cup S'}(u) \leq \tau_i$). Since the constraints are hereditary, once u becomes infeasible it stays infeasible, and the marginal benefit of u can only decrease. Therefore, at the end of the iteration, either $S \cup \{u\} \notin \mathcal{I}$ or $f'_S(u) \leq \tau_i$, a contradiction. □

The following is then immediate given Lemma 9, Corollary 7, and that SAMPLE&PRUNE can be realized in two rounds of MapReduce.

THEOREM 10. *For any hereditary family $\mathcal{I}$ and submodular function f,* GREEDYSCALING *emulates the ϵ-greedy algorithm on $(U, f, \mathcal{I})$ and can be realized in $O(\frac{1}{\epsilon\delta} \log \Delta)$ rounds of MapReduce using $O(n/\mu \log n)$ machines with $\mu = O(kn^\delta \log n)$ memory, with high probability.*

To put the above statement in perspective, consider the classical greedy algorithm for the maximum k-coverage problem. If the largest set covers $\Delta = O(\text{poly} \log n)$ elements, then by setting $\ell = \sqrt{n}$ and $k = O(\text{poly} \log n)$, we obtain an $1 - 1/e - \epsilon$ approximate algorithm that uses $O(\sqrt{n})$ machines each with $\sqrt{n} \log n$ memory in $O(\frac{\log \log n}{\epsilon})$ rounds, improving on the comparable PRAM algorithms [6, 7]. Further, in [13] an $O(\text{poly} \log(n, \Delta))$ round MapReduce algorithm for the maximum k-coverage problem was given and our algorithm achieves the same guarantees while reducing the number of rounds to be $O(\log \Delta)$ when δ and ϵ are constants.

5. MODULAR MAXIMIZATION WITH A MATROID CONSTRAINT

In this section we consider the problem of finding the maximum weight independent set of a matroid $\mathcal{M} = (U, \mathcal{I})$ of rank k with respect to a modular function $f : U \rightarrow \mathbf{R}^+$. We assume that k is given as input to the algorithm. Note that when the function is modular, it is sufficient to define the function only on the elements, as the value of a solution S is $\sum_{u \in S} f(u)$. It is a well-known fact that the greedy procedure that adds the maximum weight feasible element at each step solves the problem optimally. Indeed, the set obtained at the end of the ith step is an independent set that is maximum among those of size i. Henceforth, we will assume that f is injective; note that this is without loss of generality, as we can make f injective by applying small perturbations to the values of f. We observe that when the weights are distinct, the greedy algorithm has only one optimal choice at each step.

The following claim directly follows by the greedy property.

FACT 11. *Let $\mathcal{M} = (U, \mathcal{I})$ be a matroid of rank k and $f : U \rightarrow \mathbf{R}^+$ be an injective weight function. Let $S^* = \{s_1^*, \ldots, s_k^*\}$, where $f(s_1^*) > \cdots > f(s_k^*)$ is the greedy solution. Then, for every $1 \leq i < k$ and any $u \in U \setminus \{s_1^*, \ldots, s_{i+1}^*\}$, we have that either $\{s_1^*, \ldots, s_i^*, u\} \notin \mathcal{I}$ or $f(u) < f(s_{i+1}^*)$.*

Our analysis will revolve around the following technical result, which imposes additional structure on the greedy solution. Fix a function f and consider a matroid, $\mathcal{M} = (U, \mathcal{I})$. Let S be the greedy solution under f and consider an element $u \in S$. For any $X \subseteq U$ it is the case that u is in the greedy solution for $\mathcal{M}[X \cup \{u\}]$ under f. In other words, no element u in the (optimal) greedy solution can be blocked by other elements in a submatroid. This property of the greedy algorithm is known in the literature and, we give a proof here for completeness.

LEMMA 12. *Let $\mathcal{M} = (U, \mathcal{I})$ be a matroid of rank k and $f : U \to \mathbf{R}^+$ be an injective weight function. For any $X \subseteq U$, let $\mathcal{G}(X)$ be the greedy solution for $\mathcal{M}[X]$ under f. Then, for any $X \subseteq U$ and $u \in \mathcal{G}(U)$, it holds that $u \in \mathcal{G}(\mathcal{G}(X) \cup \{u\})$.*

PROOF. Let $u \in \mathcal{G}(U)$ and let X be any subset of U such that $u \in X$. Let A contain an element $u' \in U$ such that $f(u') > f(u)$ and let $A' = A \cap X$. We define $r(S)$ to be the rank of the matroid $M[S]$ for any $S \subseteq U$ and let $r(S, u') = r(S \cup \{u\}) - r(S)$ for any $u' \in U$. It can be easily verified that $r(S)$ is a submodular function. Notice that $r(A, u) = 1$ because u is in $\mathcal{G}(U)$. Knowing that r is submodular and $A' \subseteq A$ we have that $r(A', u) \geq r(A, u) = 1$. Thus, it must be the case that $u \in \mathcal{G}(X \cup \{u\})$ □

To find the maximum weight independent set in MapReduce we again turn to the SAMPLE&PRUNE procedure. As in the case of the greedy scaling solution, our goal will be to cull the elements not in the global greedy solution, based on the greedy solution for the sub-matroid induced by a set of sampled elements. Intuitively, Lemma 12 guarantees that no optimal element will be removed during this procedure.

MATROIDMR$(U, \mathcal{I}, f, \ell)$
1: $S \leftarrow \emptyset$, $M \leftarrow U$
2: **while** $M \neq \emptyset$ **do**
3: $\quad (S, M) \leftarrow$ SAMPLE&PRUNE$(S \cup M, \mathcal{G}, \ell)$
4: **end while**
5: **return** S

Algorithm MATROIDMR contains the formal description. Given the matroid $\mathcal{M} = (U, \mathcal{I})$, the weight function f, and a memory parameter ℓ, the algorithm repeatedly (a) computes the greedy solution S on a submatroid obtained from $\mathcal{M}$ by restricting it to a small sample of elements that includes the solution computed in the previous iteration and (b) prunes away all the elements that cannot individually improve the newly computed solution S. We remark that, unlike in the algorithm for the greedy framework of Section 4, the procedure $\mathcal{G}$ passed to SAMPLE&PRUNE is invariant throughout the course of the whole algorithm and corresponds to the classical greedy algorithm for matroids. Note that $\mathcal{G}$ always returns a solution of size at most k, as no feasible set has size more than the rank.

The correctness of the algorithm again follows from the fact that no element in the global greedy solution is ever removed from consideration.

LEMMA 13. *Upon termination, the algorithm* MATROIDMR *returns the optimal greedy solution S^*.*

PROOF. It is enough to show inductively that after each call of SAMPLE&PRUNE, we have $S^* \subseteq S \cup M$. Indeed, this implies that after each call, the greedy solution of $S \cup M$ is exactly S^*. Let $s^* \in S^* \subseteq S \cup M$, and suppose by contradiction that after a call of SAMPLE&PRUNE$(S \cup M, \mathcal{G}, \ell)$, we have $s^* \notin S' \cup M'$, where (S', M') is the new pair returned by the call. By definition of SAMPLE&PRUNE and $\mathcal{G}$, it must be that s^* was pruned since it was not a part of the greedy solution on $S' \cup \{s^*\}$. But then Lemma 12 guarantees that s^* is not part of the greedy solution of $S \cup M$. By induction, $s^* \notin S^*$, a contradiction. □

For the running time of the algorithm, we bound the number of iterations of the **while** loop using a slight variant of Corollary 7. The reason Corollary 7 is not directly applicable is that S is added to M after each iteration of the while loop in MATROIDMR.

LEMMA 14. *Let $\ell = kn^\delta \log n$. Then, the* **while** *loop of the algorithm* MATROIDMR *is executed at most $T = 1 + 1/\delta$ times, with probability at least $1 - 2Tn^{-k}$.*

PROOF. Let (S_i, M_i) be the pair returned by SAMPLE&PRUNE in the ith iteration of the **while** loop, where $S_0 = \emptyset$ and $M_0 = U$. Also, let $n_i = |S_i \cup M_i|$ and $\gamma = n^{-\delta}$. Note that $n_0 = |U| = n$. We want to show that the sequence of n_i decreases rapidly. Assuming $n_i \leq \gamma^i n + k(1 + \gamma + \cdots + \gamma^{i-1})$, Lemma 6 implies that, with probability at least $1 - 2n^{-k}$,

$$n_{i+1} \leq |M_i| + |S_i| \leq \gamma n_i + |S_i| \leq \gamma n_i + r \leq \gamma^{i+1} n + k \sum_{j=0}^{i} \gamma^j.$$

Therefore, for every $i \geq 1$, we have $n_i \leq \gamma^i n + k\frac{1-\gamma^i}{1-\gamma} < \gamma^i n + \frac{k}{1-\gamma^i}$, with probability at least $1 - 2in^{-k}$. For $i = T - 1 = \frac{1}{\delta}$, we have $n_{T-1} \leq \frac{k}{1-\gamma}$. Therefore, at iteration T, SAMPLE&PRUNE will sample with probability one all elements in $S_{T-1} \cup M_{T-1}$, and hence $S_T = \mathcal{G}(S_{T-1} \cup M_{T-1})$. By Lemma 13, $S_T = S^*$ and therefore $M_T = \emptyset$. □

Given that SAMPLE&PRUNE is realizable in MapReduce the following is immediate from Lemma 14.

COROLLARY 15. MATROIDMR *can be implemented to run in $O(1/\delta)$ MapReduce rounds using $O(n/\mu \log n)$ machines with $\mu = O(kn^\delta \log n)$ memory with high probability.*

6. MONOTONE SUBMODULAR MAXIMIZATION

In this section we consider the problem of maximizing a monotone submodular function under a cardinality constraint, knapsack constraints and p-system constraints. We first consider the special case of a single cardinality constraint, i.e., 1-knapsack constraint with unit weights. In this case, we provide a $(1/2 - \epsilon)$-approximation using a simple threshold argument. Then, we observe that the same ideas can be applied to achieve a $(1/d)$-approximation for d knapsack constraints. This is essentially the best achievable as Dean et al. [15] show a $(1/d^{(1-\epsilon)})$-hardness.

6.1 Cardinality constraint

We consider maximizing a submodular function subject to a cardinality constraint. The general techniques have the same flavor as the greedy algorithm described in Section 4. At a high level, we observe that for these problems a *single* threshold setting leads to an approximate solution and hence we do not need to iterate over the $\log_{1+\epsilon} \Delta$ thresholds. We will show the following theorem.

THEOREM 16. *For the problem of maximizing a submodular function under a k-cardinality constraint, there is a MapReduce algorithm that produces a $(\frac{1}{2} - \epsilon)$-approximation in $O(\frac{1}{\delta})$ rounds using $O(n/\mu \log n)$ machines with $\mu = O(kn^\delta \log n)$ memory, with high probability.*

Given a monotone submodular function f and an integer $k \geq 1$, we would like to find a set $S \subset U$ of size at most k that maximizes f. We show how to achieve a $1/2 - \epsilon$ approximation in a constant number of MapReduce rounds. Also our techniques will lead to a one-pass streaming algorithms that use $\tilde{O}(k)$ space. Details of the streaming algorithm are omitted in this version of the paper. The analysis is based on the following lemma.

LEMMA 17. *Fix any $\gamma > 0$. Let* OPT *be the value of an optimal solution S^* and $\tau =$ OPT $\cdot \frac{\gamma}{2k}$. Consider any $S = \{s_1, \ldots, s_t\} \subseteq U, |S| = t \leq k$, with the following properties.*

(i) There exists an ordering of elements $s_1, \ldots, s_t$ such that for all $0 \leq i < t$, $f'_{\{s_1,\ldots,s_i\}}(s_{i+1}) \geq \tau$.

(ii) If $t < k$, then $f'_S(u) \leq \tau$, for all $u \in U$.

Then, $f(S) \geq \text{OPT} \cdot \min\left\{\frac{\gamma}{2}, 1 - \frac{\gamma}{2}\right\}$.

PROOF. If $t = k$, then $f(S) = \sum_{i=0}^{k-1} f'_{S_i}(s_{i+1}) \geq \sum_{i=0}^{k-1} \tau = k\tau = \text{OPT}\gamma/2$. If $t < k$, we know that $f'_S(u^*) \leq \tau$, for any element $u^* \in S^*$. Furthermore, since f is submodular, $f(S \cup S^*) - f(S) \leq \tau \cdot |S^* \setminus S| \leq \gamma\text{OPT}/2$. We have $f(S) \geq f(S \cup (S^* \setminus S)) - \frac{\gamma\text{OPT}}{2} \geq \text{OPT} - \frac{\gamma\text{OPT}}{2} = \left(1 - \frac{\gamma}{2}\right)\text{OPT}$, where the third step follows from the monotonicity of f. □

We now show how to adapt the algorithm to MapReduce. We will again make use of the SAMPLE&PRUNE procedure. As in the case of the greedy scaling solution, our goal will be to prune the elements with marginal gain below τ with respect to a current solution S. Moreover, only elements with marginal gain above τ will be added to S, so that Lemma 17 can be applied.

The algorithm THRESHOLDMR shares similar ideas with the MapReduce algorithm for greedy framework. In particular, the procedure $\mathcal{G}_{S,\tau,k}$ passed to SAMPLE&PRUNE changes throughout different calls, and is defined as follows: $\mathcal{G}_{S,\tau,k}(X)$ maintains a solution S' and for every element $u \in X$ adds it to the solution S' if $f'_{S \cup S'}(u) \geq \tau$ (the marginal gain is above the threshold) and $|S \cup S'| < k$ (the solution is feasible). The output of $\mathcal{G}_{S,\tau,k}(X)$ is the resulting set $S \cup S'$.

Algorithm 1 THRESHOLDMR(U, f, k, τ, ℓ)

```
S ← ∅, M ← U
while M ≠ ∅ do
    (S', M) ← SAMPLE&PRUNE(M, G_{S,τ,k}, ℓ)
    S ← S ∪ S'
end while
return S
```

Observe that all of the elements that were added to S had their marginal value at least τ by definition of $\mathcal{G}_{S,\tau,k}$. The following claim guarantees the applicability of Lemma 17. The proof is substantially a replica of that of Lemma 9 and is omitted.

LEMMA 18. *Let S be the solution returned by the algorithm* THRESHOLD. *Then, either $|S| = k$, or $f'_S(u) \leq \tau$, for all $u \in U$.*

Finally, observe that an upper bound on Δ (and therefore on OPT) can be computed in one MapReduce round. Hence, we can easily run the algorithm in parallel on different thresholds τ. This observation along with Lemma 18, Corollary 7 and the fact that SAMPLE&PRUNE can be implemented in two rounds of MapReduce, yield the following corollary.

COROLLARY 19. THRESHOLD *produces $(\frac{1}{2} - \epsilon)$-approximation and can be implemented to run in $O(\frac{1}{\delta})$ rounds using $O(n \log n/\mu)$ machines with $\mu = O(kn^\delta \log n)$ memory with high probability.*

6.2 Knapsack constraints

For the case of d knapsack constraints we are given a submodular function f and would like to find a set $S \subseteq U$ of maximum value with respect to f with the property that its characteristic vector $\mathbf{x}_S$ satisfies $C\mathbf{x}_S \leq \mathbf{b}$. Here, $C_{i,j}$ is the "weight" of the element u_j with respect to the ith knapsack, and b_i is the capacity of the ith knapsack, for $1 \leq i \leq d$. Our goal is to show the following theorem. Let k be a given upper bound on the size of the smallest optimal solution.

Algorithm 2 THRESHOLDSTREAM(f, k, τ)

```
S ← ∅
for all u ∈ U do
    if |S| < k and f'_S(u) ≥ τ then
        S ← S ∪ {u}
    end if
end for
return S
```

THEOREM 20. *For the problem of maximizing a submodular function under d knapsack constraints, there exists a MapReduce algorithm that produces a $\Omega(1/d)$-approximation in $O(\frac{1}{\delta})$ rounds using $O(n/\mu \log n)$ machines with $\mu = O(kn^\delta \log n)$ memory with high probability.*

Without loss of generality, we will assume that there is no element that individually violates a knapsack constraint, i.e., there is no u_j such that $c_{i,j} > b_i$, for every $1 \leq i \leq d$. We begin by providing an analog of Lemma 17.

LEMMA 21. *Fix any $\gamma > 0$. Let* OPT *be the value of an optimal solution S^* and $\tau_i = \frac{\gamma\text{OPT}}{2b_i d}$, for $1 \leq i \leq d$. Consider any feasible solution $S \subseteq U$ with the following properties:*

(i) There exists an ordering of its elements $a_{j_1}, \ldots, a_{j_t}$ such that for all $0 \leq m < t$ and $1 \leq i \leq d$, it holds that $f'_{S_m}(a_{j_{m+1}}) \geq \tau_i c_{i,j_{m+1}}$, where $S_m = \{a_{j_1}, \ldots, a_{j_m}\}$.

(ii) Assuming that for all $1 \leq i \leq d$, $\sum_{m=1}^{t} c_{i,j_m} \leq \frac{b_i}{2}$, then for all $a_j \in U$ there exists an index $1 \leq h(j) \leq d$ such that $f'_S(a_j) \leq \tau_{h(j)} c_{h(j),j}$.

Then, $f(S) \geq \text{OPT} \cdot \min\left\{\frac{\gamma}{4(d+1)}, 1 - \frac{\gamma}{2}\right\}$.

PROOF. Consider first the case that the solution S fills at least half of a knapsack, i.e., for some $1 \leq i \leq d$, $\sum_{m=1}^{t} c_{i,j_m} \geq \frac{b_i}{2}$; since each element $a_{j_m} \in S$ added a marginal value of at least $\tau_i c_{i,j_m}$, we have $f(S) \geq \tau_i \sum_{m=1}^{t} c_{i,j_m} = \frac{\gamma\text{OPT}}{4d}$.

Now assume that the solution S does not fill half of any knapsack. Then, using (ii), define S_i^* to be the set of elements $a_j \in S^* \setminus S$ such that $h(j) = i$, for $1 \leq i \leq d$. Because of the knapsack constraints, submodularity, and the fact that $f'_S(a_j) \leq \tau_{h(j)} c_{h(j),j}$, we must have that $f(S \cup S_i^*) - f(S) \leq \frac{\gamma\text{OPT}}{2d}$, for every $1 \leq i \leq d$. Therefore, we have $f(S \cup (S^* \setminus S)) - f(S) \leq \frac{\gamma\text{OPT}}{2}$. Therefore, by monotonicity of f, we have $f(S) \geq f(S^*) - \frac{\gamma\text{OPT}}{2} = (1 - \frac{\gamma}{2})\text{OPT}$. □

Now we show how the algorithm can be implemented in MapReduce. We assume the algorithm is given an estimate γOPT to compute all τ_i's for some γ and an upper bound k on the size of the smallest optimal solution. The algorithm will simulate the presence of an extra knapsack constraint stating that no more than k elements can be picked. (Note that the value of the optimum value for this new system is not changed, as S^* is feasible in this new system.)

We say that an element $a_j \in U$ is *heavy profitable* if $c_{i,j} \geq \frac{b_i}{2}$ and $f(\{a_j\}) \geq \tau_i c_{i,j}$ for some i. The algorithm works as follows: if there exists a heavy profitable element (which can be detected in a single MapReduce round), the algorithm returns it as solution. Otherwise, the algorithm is identical to THRESHOLD, except for the procedure $\mathcal{G}_{S,\tau,k}$ which is now defined as follows: $\mathcal{G}_{S,\tau,k}(X)$ maintains a solution S' and for every element $a_j \in X$ adds it to S' if, for all $1 \leq i \leq d$, $f'_{S \cup S'}(a_j) \geq \tau_i c_{i,j}$, and $S \cup S' \cup \{a_j\}$ is

feasible w.r.t. the knapsack constraints; the output of $\mathcal{G}_{S,\tau,k}(X)$ is the resulting set $S \cup S'$.

We now analyze the quality of the solution returned by the algorithm. In the presence of a heavy profitable element, then the returned solution has value at least $\frac{\gamma \text{OPT}}{4d}$. Otherwise, we argue that S satisfies the properties of Lemma 17: the first property is satisfied trivially by definition of the algorithm and the second property is valid by submodularity and the fact that there is no heavy profitable element.

Similarly to the algorithm THRESHOLD, we can run the algorithm in parallel on different values of the estimate γOPT. We can conclude the following.

THEOREM 22. *There exists a MapReduce algorithm that produces a $\Omega(1/d)$ approximation in $O(\frac{1}{\delta})$ rounds using $O(n \log n/\mu)$ machines with $\mu = O(kn^{\delta} \log n)$ memory with high probability.*

6.3 p-system constraints

Finally, consider the problem of maximizing a monotone submodular function subject to a p-system constraint. This generalizes the question of maximizing a monotone submodular function subject to p matroid constraints. Although one could try to extend the threshold algorithm for submodular functions to this setting, simple examples show that the algorithm can result in a solution whose value is arbitrarily smaller than the optimal due to the p-system constraint. The algorithms in this section are more closely related to the algorithm for the modular matroid; however, in this case, we will need to store all of the intermediate solutions returned by SAMPLE&PRUNE and choose the one with the largest value. Our goal is to show the following theorem.

THEOREM 23. SUBMODULAR-P-SYSTEM *can be implemented to run in $O(\frac{1}{\delta})$ rounds of MapReduce using $O(n/\mu \log n)$ machines with memory $\mu = O(kn^{\delta} \log n)$ with high probability, and produces a $(\frac{1}{p+1}\lceil \frac{1}{\delta} \rceil^{-1})$-approximation.*

The algorithm SUBMODULAR-P-SYSTEM is similar to the algorithm for the modular matroid as the procedure $\mathcal{G}$ passed to SAMPLE&PRUNE is simply the greedy algorithm. However, unlike MATROIDMR, all intermediate solutions are kept and the largest one is chosen as final solution. The following lemma provides a bound on the quality of the returned solution.

LEMMA 24. *If T is the number of iterations of the* **while** *loop in* SUBMODULAR-P-SYSTEM*, then* SUBMODULAR-P-SYSTEM *gives a $\frac{1}{(1+p)T}$-approximation.*

PROOF. Let R_i be the elements of U that are removed during the ith iteration of the **while** loop of SUBMODULAR-P-SYSTEM. Let O denote the optimal solution and $O_i := O \cap R_i$. Let S_i be the set returned by SAMPLE&PRUNE during the ith iteration. Note that the algorithm $\mathcal{G}$ used in SUBMODULAR-P-SYSTEM is known to be a $1/(p+1)$-approximation for maximizing a monotone submodular function subject to a p-system if the input to $\mathcal{G}$ is the entire universe [8].

Let $\mathcal{I}_i$ denote the p-system induced on only the elements in $R_i \cup S_i$. Note that by definition of p-systems $\mathcal{I}_i$ exists. By definition of $\mathcal{G}$ and SAMPLE&PRUNE, an element e is in the set R_i because the algorithm $\mathcal{G}$ with input $\mathcal{I}_i$ and f returns S_i and $e \notin S_i$. This is because for each element $e \in R_i$, SAMPLE&PRUNE runs the algorithm $\mathcal{G}$ on the set $(S_i \cup \{e\})$ and e was not chosen to be in the solution. Thus if we run $\mathcal{G}$ on $S \cup R_i$ still no element in R_i will be chosen to be in the output solution. Therefore, $f(S_i) \geq \frac{1}{p+1} f(O_i)$ since $\mathcal{G}$ is a $1/(p+1)$ approximation algorithm for the instance consisting of $\mathcal{I}_i$ and f. By submodularity we have that $\sum_{i=1}^{t} f(S_i) \geq \frac{1}{p+1} \sum_{i=1}^{t} f(O_i) \geq \frac{1}{p+1} f(O)$. Since there are only T such sets S_i if we return the set $\arg\max_{S \in \mathcal{F}} f(S)$ then this must give a $\frac{1}{(1+p)T}$-approximation. Further, by definition of $\mathcal{G}$, we know that for all i the set S_i is in $\mathcal{I}$, so the set returned must be a feasible solution. □

Algorithm 3 SUBMODULAR-P-SYSTEM$(U, \mathcal{I}, k, f, \ell)$

1: For $X \subseteq U$, let $\mathcal{G}(X, k)$ be the greedy procedure running on the subset X of elements: that is, start with $A := \emptyset$, and greedily add to A the element $a \in X \setminus A$, with $A \cup \{a\} \in \mathcal{I}$, maximizing $f(A \cup \{a\}) - f(A)$. The output of $\mathcal{G}(X, k)$ is the resulting set A.
2: $\mathcal{F} = \emptyset$
3: **while** $U \neq \emptyset$ **do**
4: $\quad (S, M_S) \leftarrow$ SAMPLE&PRUNE$(U, \mathcal{G}, \ell)$
5: $\quad \mathcal{F} \leftarrow \mathcal{F} \cup \{S\}$
6: $\quad U \leftarrow M_S$
7: **end while**
8: **return** $\arg\max_{S \in \mathcal{F}} f(S)$

7. ADAPTATION TO STREAMING

In this section we discuss how the algorithms given in this paper extend to the streaming setting. In the data stream model (cf. [32]), we assume that the elements of U arrive in an arbitrary order, with both f and membership in $\mathcal{I}$ available via oracle accesses. The algorithm makes one or more passes over the input and has a limited amount of memory. The goal in this setting is to minimize the number of passes over the input and to use as little memory as possible.

ϵ-greedy. We can realize each phase of GREEDYSCALING with a pass over the input in which an element is added to the current solution if the marginal value of adding the element is above the threshold for the phase. If k is the desired solution size, the algorithm will require $O(k)$ space and will terminate after $O(\frac{\log \Delta}{\epsilon})$ phases. The approximation guarantees are given in Theorem 5. For the case of modular function maximization subject to a p-system constraint, the approximation ratio achieved by GREEDYSCALING is $\frac{1}{p+\epsilon}$; we show that the approximation ratio achieved in the streaming setting cannot be improved without drastically increasing either the memory or the number of passes. The proof is omitted from this version of the paper.

THEOREM 25. *Fix any $p \geq 2$ and $\epsilon > 0$. Then any ℓ-pass streaming algorithm achieving a $\frac{1}{p-\epsilon}$ approximation for maximizing a modular function subject to a p-system constraint requires $\Omega(n/(\ell p^2 \log p))$ space.*

Matroid optimization. Lemma 12 leads to a very simple algorithm in the streaming setting. The algorithm keeps an independent set and, when a new element comes, updates it by running the greedy algorithm on the current solution and the new element. Observe that the algorithm uses only $k + 1$ space. Lemma 12 implies that no element in the global greedy solution will be discarded when examined. Therefore, we have:

THEOREM 26. *Algorithm* MATROIDSTREAM *finds an optimal solution in a single pass using $k + 1$ memory.*

Submodular maximization. The algorithms and analysis for these settings are omitted from this version of the paper.

THEOREM 27. *There exists a one-pass streaming algorithm that uses $O(k/\epsilon \log(n\Delta))$ (resp. $O(k \log(n\Delta))$ memory and produces a $\frac{1}{2} - \epsilon$ (resp. $(1/d)$) approximation for the problem of maximizing a submodular function subject to a k-cardinality constraint (resp. d knapsack constraints).*

THEOREM 28. *There exists a streaming algorithm that uses $O(kn^{\delta} \log n)$ memory and produces a $\frac{1}{p+1}\lceil \frac{1}{\delta} \rceil^{-1}$ approximation in $O(\frac{1}{\delta})$ passes, where k is the size of the largest independent set.*

8. EXPERIMENTS

In this section we describe the experimental results for our algorithm outlined in section 4 for the MAXCOVER problem. Recall the MAXCOVER problem: given a family $\mathcal{S}$ of subsets and a budget k, pick at most k elements from $\mathcal{S}$ to maximize the size of their union. The standard (sequential) greedy algorithm gives a $(1-1/e)$-approximation to this problem. We measure the performance of our algorithm, focusing on two aspects: the quality of approximation with respect to the greedy algorithm and the number of rounds; note that the latter is k for the straightforward implementation of the greedy algorithm. On real-world datasets, our algorithms perform on par with the greedy algorithm in terms of approximation, while obtaining significant savings in the number of rounds.

For our experiments, we use two publicly available datasets from `http://fimi.ua.ac.be/data/`: ACCIDENTS and KOSARAK. For ACCIDENTS, $|\mathcal{S}| = 340,183$ and for KOSARAK, $|\mathcal{S}| = 990,002$.

Figure 1 shows the performance of our algorithm for $\epsilon = \delta = 0.5$ for various values of k, on these two datasets. It is easy to see that the approximation factor is essentially same as that of Greedy once k grows beyond 40 and we obtain a significant factor savings in the number of rounds (more than an order of magnitude savings for KOSARAK.)

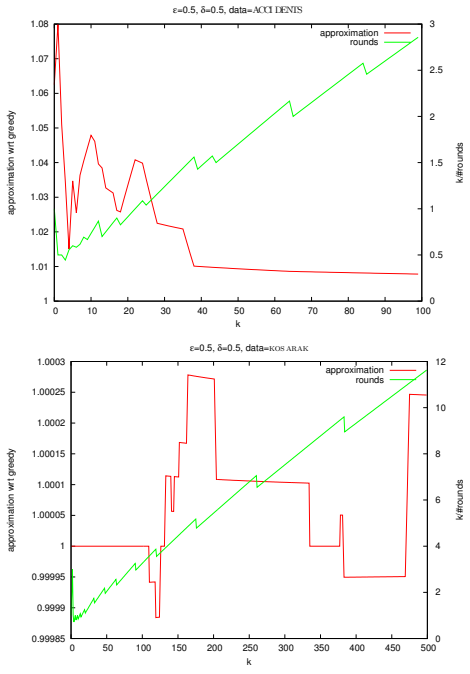

Figure 1: Approximation and number of rounds with respect to Greedy for KOSARAK and ACCIDENTS datasets as a function of k.

The first pane of Figure 2 shows the role of ϵ (Theorem 5). Even for large values of ϵ, our algorithm performs almost on par with greedy in terms of approximation (note the scale on the y-axis), while the number of rounds is significantly less. Though Theorem 5 only provides a weak guarantee of $(1-1/e)/(1+\epsilon)$, these results show that one can use $\epsilon \gg 1$ in practice without sacrificing much in approximation, while gaining significantly in the number of rounds.

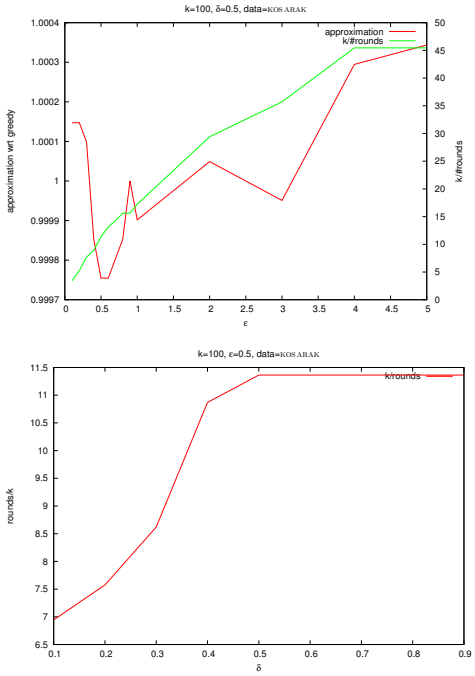

Figure 2: Approximation and number of rounds with respect to Greedy for KOSARAK datasets as a function of ϵ and δ.

Finally, the lower pane of Figure 2 shows the role of δ on the number of rounds. Smaller values of δ require more iterations of SAMPLE&PRUNE to achieve the memory requirement, thus requiring a larger number of rounds overall. However even for $\delta = 1/2$, with memory requirement only $k\sqrt{n}\log n$, is already enough to achieve the best possible speedup. Once again, a higher value of δ results in a larger savings in the number of rounds. The approximation factor is unchanged.

9. CONCLUSIONS

In this paper we presented algorithms for large-scale submodular optimization problems in the MapReduce and streaming models. We showed how to realize the classical and inherently sequential greedy algorithm in these models and allow algorithms to select more than one element at a time in parallel without sacrificing the quality of the solution. We validated our algorithms on real world datasets for the maximum coverage problem and showed that they yield an order of magnitude improvement in reducing the number of rounds, while producing solutions of the same quality. Our work opens up the possibility of solving submodular optimization problems at web-scale.

Many interesting questions remain. Not all greedy algorithms fall under the framework presented in this paper. For example the augmenting-paths algorithm for maximum flow, or the celebrated Gale-Shapley matching algorithm cannot be phrased as submodular function optimization. Giving efficient MapReduce or streaming algorithms for those problems is an interesting open question.

More generally, understanding what classes of algorithms can and cannot be efficiently implemented in the MapReduce setting is a challenging open problem.

Acknowledgments. We thank Chandra Chekuri for helpful discussions and pointers to relevant work. We thank Sariel Har-Peled for advice on the presentation of this paper.

10. REFERENCES

[1] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14, 2009.

[2] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

[3] M. Babaioff, N. Immorlica, and R. Kleinberg. Matroids, secretary problems, and online mechanisms. In *SODA*, pages 434–443, 2007.

[4] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and MapReduce. *PVLDB*, 5(1), 2012.

[5] M. Bateni, M. Hajiaghayi, and M. Zadimoghaddam. Submodular secretary problem and extensions. In *APPROX-RANDOM*, 2010.

[6] Berger, Rompel, and Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *JCSS*, 49:454–477, 1994.

[7] G. E. Blelloch, R. Peng, and K. Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *SPAA*, pages 23–32, 2011.

[8] G. Calinescu, C. Chekuri, M. Pal, and J. Vondrak. Maximizing a submodular set function subject to a matroid constraint. *SICOMP*, To appear.

[9] G. Capannini, F. M. Nardini, R. Perego, and F. Silvestri. Efficient diversification of web search results. *PVLDB*, 4(7):451–459, 2011.

[10] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*, pages 84–95, 2000.

[11] W. Chen, C. Wang, and Y. Wang. Scalable inïňĆuence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, pages 1029–1038, 2010.

[12] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–âĂŞ208, 2009.

[13] F. Chierichetti, R. Kumar, and A. Tomkins. Max-cover in Map-reduce. In *WWW*, pages 231–240, 2010.

[14] G. Cormode, H. J. Karloff, and A. Wirth. Set cover algorithms for very large datasets. In *CIKM*, pages 479–488, 2010.

[15] B. C. Dean, M. X. Goemans, and J. Vondrák. Adaptivity and approximation for stochastic packing problems. In *SODA*, pages 395–404, 2005.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, page 10, 2004.

[17] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[18] A. Ene, S. Im, and B. Moseley. Fast clustering using MapReduce. In *KDD*, pages 85–94, 2011.

[19] U. Feige. A threshold of $\ln n$ for approximating set cover. *JACM*, 45(4):634–652, 1998.

[20] M. L. Fisher, G. L. Nemhauser, and L. A. Wolsey. An analysis of approximation for maximizing submodular set functions II. *Math. Prog. Study*, 8:73–87, 1978.

[21] A. Goel, S. Guha, and K. Munagala. Asking the right questions: model-driven optimization using probes. In *PODS*, pages 203–212, 2006.

[22] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. In *ISAAC*, pages 374–383, 2011.

[23] A. Goyal, F. Bonchi, and L. V. S. Lakshmana. A data-based approach to social influence maximization. *PVLDB*, 5(1), 2012.

[24] A. Gupta, A. Roth, G. Schoenebeck, and K. Talwar. Constrained non-monotone submodular maximization: Offline and secretary algorithms. In *WINE*, pages 246–257, 2010.

[25] E. Hazan, S. Safra, and O. Schwartz. On the complexity of approximating k-set packing. *Computational Complexity*, 15(1):20–39, 2006.

[26] T. A. Jenkyns. The efficacy of the "greedy" algorithm. In *Proceedings of 7th South Eastern Conference on Combinatorics, Graph Theory and Computing*, pages 341–350, 1976.

[27] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, pages 938–948, 2010.

[28] D. Kempe, J. M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

[29] B. Korte and D. Hausmann. An analysis of the greedy heuristic for independence systems. *Annals of Discrete Math.*, 2:65–74, 1978.

[30] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *SPAA*, pages 85–94, 2011.

[31] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.

[32] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[33] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximation for maximizing submodular set functions I. *Math. Prog.*, 14:265–294, 1978.

[34] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for mapreduce computations. *CoRR*, abs/1111.2228, 2011.

[35] B. Saha and L. Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *SDM*, pages 697–708, 2009.

[36] A. D. Sarma, A. Lall, D. Nanongkai, R. J. Lipton, and J. J. Xu. Representative skylines using threshold-based preference distributions. In *ICDE*, pages 387–398, 2011.

[37] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[38] J. Vondrak. *Submodularity in Combinatorial Optimization*. PhD thesis, Charles University, Prague, 2007.