John Miller
May 4, 2014
CS167 Presentation Write-up

A Tight Linear Time (1/2)-Approximation for Unconstrained Submodular Maximization [6]

1 Introduction

Consider a manager hiring workers to staff a new factory. Moving from zero employees to a thousand employees exponentially increases the factory's production. Recognizing a pattern, the manager again hires a thousand employees. This increase in workers also raises factory production because machines are now used more efficiently, but the effect is not quite as dramatic as opening the factory from scratch. Emboldened by his previous success, the manager decides to hire another thousand workers. However, this time the additional workers leads to overcrowding, and the factory's production barely increases.

This phenomenon, the *principle of diminishing returns*, is a foundational concept in economics. Intuitively, adding an additional unit of a good to a small set has a much larger impact compared to adding a single unit of a good to a larger set. This notion is rigorously captured by the theory of submodular functions, which we define as follows.

Definition 1. Let N be a finite set. A function $f : 2^N \to \mathbb{R}^+$ is called *submodular* if for every $A \subseteq B \subseteq N$ and element $u \in N$, $f(A \cup \{u\}) - f(A) \geq f(B \cup \{u\}) - f(B)$.

Interestingly, submodular functions frequently arise in a variety of mathematical contexts. We give two demonstrative examples.

Example 1. *Graph Cuts* Let $G = (V, E)$ be an undirected or directed graph. Consider $S \subseteq V$. Let $\delta(S)$ denote the number of edges connecting S to $V \setminus S$. Formally, $\delta(S) = |\{uv \in E \mid u \in S \text{ and } v \in V - S\}|$. A basic counting argument shows that δ is submodular. Furthermore, we can extend δ to the weighted case. Given nonnegative edge weights $w : E \to R^+$, we use a modified function

$$w(\delta(S)) = \sum_{i \in S j \in V - S} w_{ij}$$

. As before, this function is submodular.

Example 2. *Set Cover* Let U be a finite ground set with n subsets $S_1, \ldots, S_n \subseteq U$. The *coverage* function, defined by $f(A) = | \cup_{i \in A} S_i|$, is submodular. As in the case of graphs with cuts, we can also extend f to the weighted case. Given a non-negative weight function $w : U \to R^+$, the *weighted coverage function* $f(S) = w(\cup_{i \in A} S_i)$ is also submodular. It is important to note that, in both cases, the function f is in fact monotone.

As the previous examples illustrate, many combinatorial optimization problems can be naturally cast in the language of submodular functions. Hence one is interested not merely in the existence of submodular functions, but also maximizing and minimizing such functions. This motivates the following examples.

Example 3. *Minimum Cut* Consider a graph $G = (V, E)$. Minimizing the cut function $\delta(S)$ over $S \subseteq V$ is the classic minimum cut problem.

Example 4. *(Maximum Cut)* Consider a graph $G = (V, E)$. The maximum cut problem is to find the cut of maximum cardinality in G. This directly corresponds to solving $\max_{S \subseteq V} \delta(S)$, where δ is the cut function given earlier. This problem is NP-Complete. [7]

Example 5. *(Maximum k-Coverage)* Let U be a finite ground set with n subsets $S_1, \ldots, S_n \subseteq U$. The maximum coverage problem is to find a subset $A \subseteq S$ such that $|A| \leq k$ and the coverage function $f(A)$ is maximized. This problem is NP-hard.

The trivial reduction from maximum cut to submodular maximization implies that the general problem of maximizing submodular functions is NP-Hard. Therefore, one cannot hope for a polynomial algorithm to maximize submodular functions under standard assumptions. However, despite this reality, these functions have been extensively studied by both computer scientists and operations researchers for several decades. Since the late 1970's, there has been significant progress in the development of approximation algorithms and special case solutions. It is the aim of this paper to first briefly survey one of these special case results and then more thoroughly examine a recent result by Buchbinder et. al that fully resolves the approximability of non-negative unconstrainted submodular maximization.

2 Special Cases

Before launching into the details of submodular maximization, we will first consider the case of submodular minization. It is interesting to point out that the problem of minimizing submodular function can be solved efficiently.

Definition 2. *(Submodular Minimization)* Given some finite ground set N, and a submodular function f on N, a *minimizer* of f is a subset $A \subseteq N$ such that $f(A) \leq f(B)$ for all other subsets $B \subseteq N$. Submodular minization is the problem of finding a minimizer of f.

Finding a minimum (s, t) cut is a well-known example of submodular minimization. In the case of minimum cut, it is known to be solvable efficiently as an application of the maximum flow-minimum cut theorem. However, these techniques heavily rely on the graph structure inherent in the problem. The general problem of submodular minimization lacks this combinatorial structure. One is simply given some ground set N and an oracle that computes the value of the function on any subset $A \subseteq N$. With only access to these two items, how can we efficiently find a minimizer of f?

The first polynomial time algorithm for submodular minimization was developed by Grotschel et al. in the early 1980's. [3] Grostschel et al.'s approach critically relied on the ellipsoid method developed by Khachiyan for linear programming. This was unsatisfying because, although the ellipsoid algorithm runs in polynomial time, it is notorious for being slow in practice. Worse, the ellipsoid method did not shed insight into the fundamental problem structure. These shortcomings motivated the search for so-called “fully combinatorial algorithms” for submodular minimization. This search was major open problem for more than twenty years until, in 2000, Schrijver presented the first fully combinatorial, strongly polynomial time algorithm for submodular minimization. [8]

In a perfect world, submodular maximization would be as tractable as submodular minimization. However, we have already seen that submodular maximization is NP-Hard, so it is unlikely one can find efficient algorithms for maximization. In some sense, then, one can think of submodularity as a discrete analog of convexity. Efficient algorithms exist to minimize convex functions, yet the problem of maximization remains intractable. When faced with these intractable problems, what can we do? The rest of this write-up will focus on tools for coping with the NP-Hardness of submodular maximization.

3 Submodular Maximization

Definition 3. *Unconstrained Submodular Maximization* Given a finite ground set N and a non-negative submodular function $f : 2^N \to \mathbb{R}^+$ on N, a *maximizer* is a subset $S \subseteq N$ such that $f(S) \geq f(B)$ for all other subsets $B \subseteq N$. Submodular maximization is the problem of finding a maximizer for f, i.e. a subset S such that $f(S)$ is maximized.

We will assume that we have access to a *value oracle* for f. In particular, this means that given some submodular f and a subset S, we can compute $f(S)$ in $\mathrm{O}\big(1\big)$ time. For general submodular functions, it is known that even verifying whether the maximum is greater than zero is NP-Hard and requires an exponential number of queries to the value oracle. This means one cannot find efficient approximation algorithms for general submodular maximization problems.[9] To make the problem more tractable, we also assume that f is non-negative.

We have already seen two examples of NP-Hard problems reduced to unconstrained submodular maximization problems. One of the many paradigms developed for coping with this NP-Hardness is approximation algorithms. Within this paradigm, the algorithms we present will not always produce optimal solutions, but rather settle for “close to optimum” solutions. In order to make this notion rigourous, we will define approximation algorithms as follows.

Definition 4. *(Approximation Algorithm)* An α-approximation algorithm for an optimization problem is a polynomial time algorithm that for all instances of the problem produces a solution whose values is within a factor of α of the value of the optimum solution.

For an α-approximation algorithm, we will call α the *approximation factor* of the algorithm. Conventionally, $\alpha < 1$ for maximization problems. Therefore, an $\frac{1}{2}$ approximation algorithm for submodular maximization is a polynomial time algorithm that always returns a solution, S, whose value, $f(S)$, is at least half the value of $f(OPT)$, where OPT denotes the optimum solution.

As a sanity check, note that the problem of unconstrained submodular maximization is trivial if the function f is monotone. One can simply take the entire ground set N as a solution. Then, using the definition of monotone, f is nondecreasing as $|S|$ increases. In particular, $f(N) = f(OPT)$, so the entire set N gives us an exact solution to the optimization problem. Therefore, we will mainly be interested in the case where f is not monotone, e.g. the cut-function.

4 Main Results

One might naturally ask, does any constant approximation algorithm exist? Feige et al. provided the first affirmative answer to this question in 2007. As a starting point, choosing the set S uniformly at random gives an approximation algorithm of (1/4). This approximation factor can be improved to (2/5) by using local search techniques. However, Feige et al. also proved that for any $\epsilon > 0$, achieving a $(1/2+\epsilon)$ approximation factor requires an exponential number of queries to the value oracle. [9] Therefore, one cannot hope for better than a (1/2) approximation algorithm.

Given this impossibility result, the question then becomes what is the best approximation factor we can achieve? Is a (2/5) approximation factor optimal, or can we find an algorithm that achieves a (1/2) approximation? The remainer of this write-up is devoted to a recent result by Buchbinder et al. that gives a (1/2) approximation factor and thereby fully resolves the approximability of unconstrained submodular maximization. Our treatment will closely follow [6].

4.1 Failure of the Greedy Approach

Although we are interested in the more general case, one possible intuition is that the monotone case might lead us to algorithms for the general case. For example, it is known that in the constrained maximization case, if f is monotone, then the greedy hill-climber algorithm yields a $(1-\frac{1}{e})$ approximation factor. [5] This suggests an almost trivial greedy algorithm for the general case. Unfortunately, we will show that this simple algorithm has an unbounded approximation factor.

```
Algorithm 1: Simply Greedy Algorithm
  Data: Finite set $N$, non-negative submodular function $f$
  Result: subset $S \subseteq N$
    $S \leftarrow \emptyset$;
    while There exists $x \in N$ such that $f(S \cup \{x\}) > f(S)$ do
      Add the best such element $x$ to $S$;
    end while
    return $S$;
```

Proposition. *The simple greedy algorithm has an unbounded approximation factor.*

Proof. Consider the following example. Let $N = \{a_1, \dots, a_n\}$. Define a non-negative submodular function f by

$$f(S) = \begin{cases} 1 & \text{if } |S| = 1 \text{ and } S = \{a_1\} \\ \frac{1}{2} & \text{if } |S| = 1 \text{ and } S \neq \{a_1\} \\ 0 & \text{if } |S| > 1 \text{ and } a_1 \in S \\ \sum_{a_i \in S} f(a_i) & \text{otherwise} \end{cases}$$

On its first iteration, the greedy algorithm will select $S = \{a_1\}$ on its first iteration because a_1 is the best element remaining element. Then, on the subsequent iteration, it will terminate because adding any additional element decreases the objective. However, the optimal solution is instead $OPT = \{a_2, \dots, a_n\}$, and $f(OPT) = (n-1)/2$. Recalling the definition of an approximation factor, the simple greedy algorithm has an approximation factor of

$$\alpha = \frac{f(S)}{f(OPT)} = \frac{2}{n-1}$$

Since an approximation algorithm must work for any instance of the problem, we can make α arbitrarily low by increasing the value of n. Hence, the simple greedy algorithm has an unbounded approximation factor, as desired. □

4.2 Failure of the Reverse Greedy Approach

We have seen a problem instance where the simple greedy algorithm has an unbounded approximation factor. One natural response is to modify the algorithm to avoid this "bad" problem instance. For example, if we were to start with the entire set N and iteratively remove elements to increase $f(S)$, then in the above example we could simply remove a_1 and achieve the optimum solution! This leads us to try a "reverse greedy" approach, which is formally given below.

Algorithm 2: Reverse Greedy Algorithm

```
Data: Finite set $N$, non-negative submodular function $f$
Result: subset $S \subseteq N$
  $S \leftarrow N$;
  while There exists $x \in N$ such that $f(S \setminus \{x\}) > f(S)$ do
    Remove the best such element $x$ from $S$;
  end while
  return $S$;
```

Unfortunately, this algorithm also has an unbounded approximation factor. To prove this, we will show how to construct instances where "reverse greedy" fails out of instances where the straight-forward greedy approach fails. First, we need to define the complement of a submodular function.

Definition 5. Given a non-negative submodular function f, the *complement of* f, denoted $\bar{f}$ is defined as $\bar{f}(S) = f(N \setminus S)$ for any subset $S \subseteq N$.

First, observe that since f is a submodular function, $\bar{f}$ is also a submodular function. Furthermore, suppose OPT maximizes $f(OPT)$, then the subset $N \setminus OPT$ maximizes $\bar{f}$. To see why, assume there exists some set A such that $\bar{f}(A) > \bar{f}(N \setminus OPT)$. Applying the definition of $\bar{f}$, this means that $f(N \setminus A) > f(OPT)$, which is a contradiction. Therefore, $N \setminus OPT$ is indeed a maximizer for $\bar{f}$.

Consider the function $\bar{f}$. By construction, running the "reverse greedy" algorithm on $\bar{f}$ is the same as running the simple greedy algorithm on f. Therefore, given some problem instance where the greedy algorithm has an unbounded approximation factor, one can immediately transform it into an instance where the "reverse greedy" algorithm has an unbounded approximation factor by changing the objective function from f to $\bar{f}$.

4.3 (1/3) Deterministic, Linear Time Algorithm

Given that both the greedy and so-called reverse greedy algorithms fail for unconstrained submodular maximization, it comes as a surprise that it is possible to extract a constant factor approximation by using a modified version of both greedy algorithms run in some coordinated fashion. The main idea is as follows.

We will maintain two solutions, $X = \emptyset$ and $Y = N$, corresponding the greedy and reverse greedy solutions respectively. We make a single pass over all of the elements in N, considering each element exactly once. When we examine element i, we make sure that both X and Y agree on element i by either adding i to X or removing i from Y. We decide whether to add or remove i in a greedy fashion. That is, we compute the marginal benefit gain from adding i to X or removing i from Y and take the action that maximizes this gain. After $|N|$ steps, both X and Y agree on all of the elements, and we output X.

We will first formally present and analyze the deterministic variant of the above algorithm, and show that it obtains an approximation factor of (1/3). Then, we will present a slightly modified and randomized version of this algorithm and show that is obtains a (1/2) approximation factor.

Let $N = \{u_1, \ldots, u_n\}$ be our ground set. Furthermore, let X_i be the greedy solution on iteration i, and, similarly, let Y_i be the reverse greedy solution on iteration i.

Algorithm 3: Deterministic Submodular Maximization

```
Data: Finite set $N$, non-negative submodular function $f$
Result: subset $S \subseteq N$
  $X_0 \leftarrow \emptyset$, $Y_0 \leftarrow N$;
  for $i \leftarrow 1$ to $i = |N|$ do
    $a_i \leftarrow f(X_{i-1} \cup \{u_i\}) - f(X_{i-1})$;
    $b_i \leftarrow f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1})$;
    if $a_i \geq b_i$ then
      $X_i \leftarrow X_{i-1} \cup \{u_i\}$;
      $Y_i \leftarrow Y_{i-1}$;
    else
      $Y_i \leftarrow Y_{i-1} \setminus \{u_i\}$;
      $X_i \leftarrow X_{i-1}$;
    end if
  end for
  return $X_n$ (or equivalently $Y_n$;
```

We first prove the following technical lemma. Intuitively, the lemma states that on every iteration of the algorithm, at least one of the sets X or Y improves in value.

Lemma 1. *For $1 \leq i \leq n$, $a_i + b_i \geq 0$*

Proof. Fix some iteration i. First, observe that $(X_{i-1} \cup \{u_i\}) \cup (Y_i \setminus \{u_i\}) = Y_{i-1}$. Similarly, we also have that $(X_{i-1} \cup \{u_i\}) \cap (Y_i \setminus \{u_i\}) = X_{i-1}$. Combining these observation directly with the definition of submodularity, we obtain

$$\begin{aligned} a_i + b_i &= [f(X_{i-1} \cup \{u_i\}) - f(X_{i-1})] + [f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1})] \\ &= [f(X_{i-1} \cup \{u_i\}) + f(Y_{i-1} \setminus \{u_i\})] - [f(X_{i-1}) + f(Y_{i-1})] \\ &\geq 0 \end{aligned}$$

□

For our next lemma, we need to introduce more notation. Let OPT denote the optimal solution. Define $OPT_i = (OPT \cup X_i) \cap Y_i$. Recall that X and Y agree on element i after iteration i, which means that X_i and Y_i agree on elements $1, \ldots, i$. Combining this observation with the definition, note that OPT_i agrees with X_i and Y_i on the elements $1, \ldots, i$ and agrees with OPT on elements $i+1, \ldots, n$. Initially, $OPT_0 = OPT$, and at termination of the algorithm, $OPT_n = X_n$.

In order to bound the approximation factor of the algorithm, we will therefore attempt to bound the loss in value from OPT_0 to OPT_n. The proof will do this by showing that the loss between any two steps in this sequence is bounded by the total increase in the value of the X and Y.

Lemma 2. *For $1 \leq i \leq n$, $f(OPT_{i-1}) - f(OPT_i) \leq [f(X_i - f(X_{i-1})] + [f(Y_i - f(Y_{i-1})]$.*

Assuming that lemma (2) holds, we will prove that the algorithm has a (1/3) approximation factor.

Proof. Summing up the expression in lemma (2) over every iteration of the algorithm, we obtain

$$\sum_{i=0}^{n}[f(OPT_i - f(OPT_{i-1})] \leq \sum_{i=0}^{n}[f(X_i - f(X_{i-1})] + \sum_{i=0}^{n}[f(Y_i - f(Y_{i-1})]$$

Note that the above sum has a nice telescoping property. Each $i-1$ cancels the i term in the subsequent iteration. Therefore, expanding the sum and cancelling out terms, we are left with

$$f(OPT_0) - f(OPT_n) \leq [f(X_n) - f(X_0)] - [f(Y_n) - f(Y_0)]$$

By our above discussion, $OPT_0 = OPT$ and $OPT_n = X_n = Y_n$. Therefore, this becomes

$$f(OPT) - f(X_n) \leq [f(X_n) - f(X_0)] - [f(X_n) - f(Y_0)]$$

Finally, note that the function f is non-negative, so $f(X_0) + f(Y_0) \geq 0$. Removing these terms,

$$f(OPT) \leq f(X_n) + [f(X_n) - f(X_0)] - [f(X_n) - f(Y_0)] \qquad \leq 3f(X_n)$$

Therefore, $\frac{1}{3}f(OPT) \leq f(X_n)$, as desired.

□

It remains to prove lemma (2).

Proof. Fix iteration i. Without loss of generality, $a_i \geq b_i$. Hence on iteration i, we decided to add u_i to X_{i-1}. Since OPT_i must agree with X_i, this means that $OPT_i = OPT_{i-1} \cup \{u-1\}$. Therefore, to prove the lemma, we must show

$$f(OPT_i) - f(OPT_i \cup \{u_i\}) \leq f(X_i) - f(X_{i-1})$$

Either $u_i \in OPT$ or $u_i \notin OPT$. We will consider both cases independently.

Case 1: Assume $u_i \in OPT$. Then $OPT_i = (OPT_i \cup \{u_i\})$ by definition of OPT_i. Rewriting the inequality, we have $f(OPT_i) - f(OPT_i) = 0 \leq a_i$. To see that $a_i \geq 0$, we will use lemma [1]. By lemma [1], $a_i + b_i \geq 0$. By assumption, $a_i \geq b_i$, so $a_i \geq 0$, completing this case.

Case 2: Assume $u_i \notin OPT$. Then $u_i \notin OPT_{i-1}$. Recall that $OPT_{i-1} = (OPT \cup X_{i-1}) \cap Y_{i-1}$.

Using this definition and the fact that $u_i \notin OPT_{i-1}$, it is clear that $OPT_{i-1} \subseteq Y_{i-1} \setminus \{u_i\}$. This means that we can apply the submodularity of f to obtain

$$f(OPT_{i-1}) - f(OPT_{i-1} \cup \{u_i\}) \leq f(Y_{i-1} \setminus \{u_1\}) - f(Y_{i-1}) = b_i \leq a_i,$$

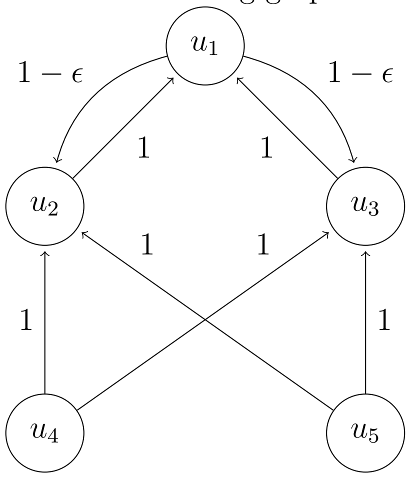
where the last inequality follows by assumption.

In either case, $f(OPT_i) - f(OPT_i \cup \{u_i\}) \leq a_i = f(X_i) - f(X_{i-1})$. The proof for $b_i \geq a_i$ is symmetric. □

4.4 (1/3) Analysis is Tight

We were able to prove that the deterministic variant yields a (1/3) approximation. A natural next question is whether or not this analysis is tight. Perhaps the algorithm actually has a better approximation factor, but our proof was not sufficiently clever. To show that our analysis is tight, we will provide an example where the algorithm produces a solution that is a $(1/3) + \epsilon$ approximation for any $\epsilon > 0$.

Assume that the ground set is $N = \{u_1, \ldots, u_5\}$ and that our submodular function is the directed cut function. The goal is to maximize the weight of the cut function. However, consider the following graph



Let's trace the execution of the algorithm. Initially, $X_0 = \emptyset$ and $Y_0 = \{u_1, \ldots, u_5\}$.

1. On iteration 1, the algorithm considers u_1. Adding u_1 to X_0 gives a marginal increase of 2-2ϵ, while removing u_1 from Y_0 gives a marginal increase of 2. Therefore, $X_1 = X_0$ and $Y_1 = Y_0 \setminus \{u_1\}$.

2. On iteration 2, the algorithm consider u_2. Adding u_2 to X_1 gives a marginal gain of 1, while removing u_2 from Y_1 also gives a marginal gain of 1. Therefore, the algorithm adds u_2 to X_1. Hence $X_2 = \{u_2\}, Y_2 = \{u_2, \ldots, u_5\}$

3. On iteration 3, the algorithm considers u_3. Adding u_3 to X_2 gives a marginal gain of 1, while removing u_3 from Y_2 also gives a marginal gain of 1. Therefore, the algorithm adds u_3 to X_2. Hence $X_3 = \{u_2, u_3\}, Y_3 = \{u_2, \ldots, u_5\}$

4. On iteration 4, the algorithm considers u_4. Adding u_4 to X_3 does not change the value of X_3, and removing u_4 from Y_3 also does not change the objective. Therefore, the algorithm adds u_4 to X_3. Hence, $X_4 = \{u_2, u_3, u_4\}, Y_3 = \{u_2, \ldots, u_5\}$

5. On iteration 5, the algorithm considers u_5. Adding u_5 to X_4 does not change the value of X_4, and removing u_5 from Y_4 also does not change the objective. Therefore, the algorithm adds u_5 to X_4. Hence, $X_5 = \{u_2, u_3, u_4, u_5\}, Y_3 = \{u_2, \ldots, u_5\}$

6. The algorithm returns $X_5 = Y_5 = \{u_2, \ldots u_5\}$.

Inspective of the above graph reveals that $\delta(X_5) = 2$, where δ is the weighted cut function. However, $\delta(u_1, u_4, u_5) = 6 - 2\epsilon$. Therefore, recalling the approximation factor in this instance is

$$\frac{2}{6-2\epsilon} = \frac{1}{3-\epsilon} \leq \frac{1}{3} + \epsilon$$

Therefore, the (1/3) analysis is, in fact, tight.

5 Achieving (1/2) Approximation

The deterministic algorithm ultimately obtains a (1/3) approximation factor. How can we extend this to a (1/2) approximation? The answer lies in adding randomization to the execution. In the deterministic case, we simply compute that marginal gain from adding u_i to the greedy solution and the marginal removing u_i from the reverse greedy solution. Then, we deterministically add or remove u_i to maximize this marginal gain.

In the randomized variant, we will, as before, compute the marginal gain from adding u_i to X or removing u_i from Y. However, we will now make a decision to add or remove u_i according to a probability distribution. Intuitively, we will add u_i to X with high probability if a_i is large and remove u_i from Y if b_i is large. Likewise, if a_i is small or negative, we will add u_i to X with low or 0 probabilty, and remove u_i from Y with low or zero if b_i is small or negative. A formal description of the algorithm is as follows.

```
Algorithm 4: Randomized Submodular Maximization
  Data: Finite set $N$, non-negative submodular function $f$
  Result: subset $S \subseteq N$
    $X_0 \leftarrow \emptyset$, $Y_0 \leftarrow N$;
    for $i \leftarrow 1$ to $i = |N|$ do
      $a_i \leftarrow f(X_{i-1} \cup \{u_i\}) - f(X_{i-1})$;
      $b_i \leftarrow f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1})$;
      $a_i' \leftarrow \max\{a_i, 0\}$, $b_i' \leftarrow \max\{b_i, 0\}$;
      With probability $a_i'/(a_i' + b_i')$ do
            $X_i \leftarrow (X_{i-1} \cup \{u_i\})$;
            $Y_i \leftarrow Y_{i-1}$;
      else with complement probability $b_i'/(a_i' + b_i')$ do
            $Y_i \leftarrow (Y_{i-1} \setminus \{u_i\})$;
            $X_i \leftarrow X_{i-1}$;
    end for
    return  $X$;
    {If $a_i' = b_i' = 0$, then take $a_i'/(a_i' + b_i') = 1$}
```

The proof of the approximation factor is very similar to the proof of the deterministic variant. Our key lemma will have a similiar flavor. However, since we make a randomized choice at each step, X_i, Y_i and OPT_i are all random variables. Therefore, we will attempt to bound the loss along the sequence from $f(OPT_0)$ to the expected output of the algorithm $\mathbb{E}[f(OPT_n)]$. To do this, we will show that the loss between two iterations is upper bounded by the average expected change in our solution.

Lemma 3. *For $1 \leq i \leq n$, over random choices of the algorithm,*

$$\mathbb{E}[f(OPT_{i-1}) - f(OPT_i)] \leq \frac{1}{2} * \mathbb{E}[f(X_i) - f(X_{i-1}) + f(Y_i) - f(Y_{i-1})]$$

For brevity, we will omit the proof of lemma (3). Instead we will assume lemma (3) and show that randomized algorithm has an approximation factor of (1/2) in expectation.

Proof. Taking the sum of lemma (3) over every iteration gives us

$$\sum_{i=0}^{n} \mathbb{E}[f(OPT_{i-1}) - f(OPT_i)] \leq \frac{1}{2} \sum_{i=0}^{n} \mathbb{E}[f(X_i) - f(X_{i-1}) + f(Y_i) - f(Y_{i-1})]$$

As in the deterministic case, this sum has a telescoping property. Each iteration i is canceled by the previous $i-1$ Therefore, if we expand out the sum and cancel terms, we obtain

$$\begin{aligned}\mathbb{E}[f(OPT_0) - f(OPT_n)] &\leq \frac{1}{2}\mathbb{E}[f(X_n) - f(X_0) + f(Y_n) - f(Y_0)] \\ &\leq \frac{\mathbb{E}[f(X_n) + f(Y_n)]}{2},\end{aligned}$$

where the second inequality holds by the non-negativity of f. Recalling that $OPT_0 = OPT$ and $OPT_n = X_n$, as well as the fact that $X_n = Y_n$, this inequality reduces to

$$\begin{aligned} f(OPT) &\leq \mathbb{E}[X_n] + \mathbb{E}[X_n] \\ \implies \frac{1}{2} * f(OPT) &\leq \mathbb{E}[X_n] \end{aligned}$$

Therefore, the approximation factor is $\frac{1}{2}$, as required. $\square$

Recalling the hardness result that one cannot obtain a $(1/2) + \epsilon$ approximation factor for any $\epsilon > 0$, the above analysis is tight. This $(1/2)$ approximation algorithm therefore fully resolves the approximability of unconstrained submodular maximization.

6 Experimental Results

One natural question is how well these algorithms work in practice. Does the observed approximation ratio match what the theory would indicate? To obtain a rough answer to this question, we briefly explore a few experimental results.

As a representative problem instance, we chose the weighted, directed maximum cut problem. Recall that the cut function is submodular, so this is an unconstrained submodular maximization problem. Critically, the cut function was chosen because it is not monotone. As we have seen, unconstrained maximization is trivial in the monotone case, and we wanted a proof of concept rather than trivial optimal solutions. Both the deterministic and randomized variants of Buchbinder et al.'s algorithm were implemented in Java and tested on several benchmark datasets for the maximum cut problem. In each test, the randomized algorithm was run 100 times and the result averaged over all of the trials.

The first benchmark set of graphs consists of problem instances generated by Helmberg and Rendl. [1]. These 54 instances consist of planar and random graphs with $|V|$ ranging from 800 to over 3,000, and edge weights taken from the set $\{-1, 0, 1\}$. The results are summarized below.

	Best Known	Upper bound	Deterministic	Approx	Randomized	Approx
G1	11624	12078	9249	0.8	9455	0.81
G2	11620	12084	9334	0.8	9480	0.82
G3	11622	12077	9126	0.79	9332	0.8
G4	11646	*	9226	0.79	9382	0.81
G5	11631	*	9261	0.8	9458	0.81
G6	2178	*	1376	0.63	1235	0.57
G7	2003	*	1244	0.62	1100	0.55
G8	2003	*	1309	0.65	1113	0.56
G9	2048	*	1374	0.67	1196	0.58
G10	1994	*	1320	0.66	1118	0.56
G11	564	627	401	0.71	300	0.53
G12	556	621	402	0.72	300	0.54
G13	580	645	406	0.7	300	0.52
G14	3060	3187	2459	0.8	2402	0.78
G15	3049	3169	2485	0.82	2444	0.8
G16	3045	3172	2489	0.82	2439	0.8
G17	3043	*	2512	0.83	2468	0.81
G18	988	*	726	0.73	600	0.61
G19	903	*	616	0.68	522	0.58
G20	941	*	655	0.7	598	0.64
G21	931	*	643	0.69	598	0.64
G22	13346	14123	10228	0.77	10253	0.77
G23	13317	14129	10283	0.77	10332	0.78
G24	13314	14131	10242	0.77	10240	0.77
G25	13326	*	10223	0.77	10255	0.77
G26	13314	*	10157	0.76	10230	0.77
G27	3318	*	2207	0.67	1900	0.57
G28	3285	*	2198	0.67	1818	0.55
G29	3389	*	2286	0.67	2000	0.59
G30	3403	*	2167	0.64	1996	0.59
G31	3288	*	2164	0.66	1807	0.55

	Best Known	Upper bound	Deterministic	Approx	Randomized	Approx
G32	1398	1560	1006	0.72	844	0.6
G33	1376	1537	975	0.71	807	0.59
G34	1372	1541	966	0.7	800	0.58
G35	7670	8000	6304	0.82	6243	0.81
G36	7660	7996	6154	0.8	6136	0.8
G37	7666	8009	6231	0.81	6211	0.81
G38	7681	*	6223	0.81	6193	0.81
G39	2395	*	1677	0.7	1600	0.67
G40	2387	*	1598	0.67	1506	0.63
G41	2398	*	1623	0.68	1516	0.63
G42	2469	*	1697	0.69	1595	0.65
G43	6659	7027	5054	0.76	5080	0.76
G44	6648	7022	5087	0.77	5098	0.77
G45	6652	7020	5162	0.78	5166	0.78
G46	6645	*	4990	0.75	5054	0.76
G47	6656	*	5093	0.77	5080	0.76
G48	6000	*	2947	0.49	2735	0.46
G49	6000	*	2937	0.49	2728	0.45
G50	5880	*	2930	0.5	2711	0.46
G51	3846	*	3168	0.82	3102	0.81
G52	3849	*	3112	0.81	3092	0.8
G53	3846	*	3125	0.81	3077	0.8
G54	3846	*	3101	0.81	3075	0.8

The next benchmark set of graphs consists of problem instances described by Festa et al. [2] The first ten have an average of 128 nodes and 300 edges. The next ten have 1,000 nodes with edge density 0.6% on average, and the final ten instances have 2,750 nodes with edge density 0.22% on average. For all instances edge weights are taken from the set $\{-1, 0, 1\}$ The results are summarized below.

	OPT	Deterministic	Approx	Randomized	Approx
G1	110	77	0.7	65	0.59
G2	112	85	0.76	79	0.71
G3	106	72	0.68	74	0.7
G4	114	79	0.69	77	0.68
G5	112	74	0.66	58	0.52
G6	110	71	0.65	70	0.64
G7	112	82	0.73	76	0.68
G8	108	77	0.71	76	0.7
G9	110	71	0.65	70	0.64
G10	112	85	0.76	89	0.79
G11	892	598	0.67	544	0.61
G12	900	614	0.68	575	0.64
G13	884	608	0.69	572	0.65
G14	896	603	0.67	535	0.6
G15	882	610	0.69	555	0.63
G16	886	575	0.65	521	0.59
G17	896	593	0.66	529	0.59
G18	880	590	0.67	553	0.63
G19	898	609	0.68	557	0.62
G20	890	616	0.69	568	0.64
G21	2428	1591	0.66	1479	0.61
G22	2418	1618	0.67	1526	0.63
G23	2410	1612	0.67	1525	0.63
G24	2422	1607	0.66	1489	0.61
G25	2416	1616	0.67	1490	0.62
G26	2424	1620	0.67	1518	0.63
G27	2410	1640	0.68	1496	0.62
G28	2418	1635	0.68	1512	0.63
G29	2412	1623	0.67	1457	0.6
G30	2430	1637	0.67	1514	0.62

The final benchmark set of graphs consist of three instances from the 7th DIMACS Implementation Challenge. The number of nodes varies from 512 to 3,375 and the number of edges from 1,536 to 10,125. Edge weights are all integers. The results are given below.

	Best Known	Deterministic	Approx	Randomized	Approx
G1	281029888	187056862	0.67	183179278	0.65
G2	41684814	27953867	0.67	27809213	0.67
G3	454	292	0.64	200	0.44

Across all three problem instance, both the deterministic and randomized algorithms significantly outperformed the theoretical worst-case approximation factor of (1/3) and (1/2), respectively. Likewise, the deterministic algorithm was competitive with or outperformed the randomized algorithm in almost all cases. However, these experimental results are not sufficient to draw any lasting conclusions about the performance of the algorithms in the general case.

First, the benchmark datasets were choosen because it was easy to find exact or best known solutions for large problem instances rather than for variety of graph structure. We have already seen an example of directed maximum cut where the deterministic maximization algorithm yields a $(1/3) + \epsilon$ approximation factor for any $\epsilon > 0$, yet our algorithms performed quite well on these benchmark instances. In addition, we focused only on a single submodular optimization problem, i.e. maximum cut. The maximum cut problem has additional combinatorial structure that other submodular maximization problems lack. For example, Goemans and Williamson's celebrated semidefinite programming algorithm obtains a 0.878 approximation factor for maximum cut, but one cannot do better than (1/2) on general submodular maximization problems. [4] Therefore, it would be an interesting extension to obtain experimental results on other classes of submodular maximization problems, especially where the objective is not monotone.

7 Conclusions

We have introduced the concept of submodularity and initiated a study of unconstrained submodular maximization. Submodular functions naturally capture the familiar economics principle of diminishing returns. Interestingly, this principle unifies hundreds of diverse problems in combinatorial optimization, making it a rich area of study. Since submodular maximization captures such a wide array of problems, it is unsurprising that unconstrained submodular maximization is NP-Hard. Furthermore, it is known that one cannot achieve better than a (1/2) approximation factor for this problem.

The key achievement of Buchbinder et al. was to fully resolve the question of the approximability of unconstrained submodular maximization by providing a randomized (1/2) approximation algorithm. Although we have seen that several greedy approaches have an unbounded approximation factor, Buchbinder et al.'s algorithm manages to combine two of these greedy techniques into a single algorithm that achieves a (1/3) approximation. As a testament to the power of randomization, this simple (1/3) -approximation algorithm is then transformed into a (1/2)-approximation algorithm by adding or removing elements from our solution according to a carefully choosen probability distribution.

References

[1] Helmberg, C., F. Rendl. 2000. A spectral bundle method for semidefinite programming. SIAM J. Optim. 10 673696

[2] Festa, P., P. M. Pardalos, M. G. C. Resende, C. C. Ribeiro. 2002. Randomized heuristics for the max-cut problem. Optim. Methods Software 7 10331058

[3] Grotschel,M.,Lovasz,L.,Schrijver,A.:The ellipsoid method and its consequences in combinatorial optimization. Combinatorica 1, 169197 (1981)

[4] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM, 42(6):11151145, 1995.

[5] Nemhauser, G. L.; Wolsey, L. A.; Fisher, M. L. (1978), "An analysis of approximations for maximizing submodular set functions I", Mathematical Programming 14 (1): 265294.

[6] Niv Buchbinder, Moran Feldman, Joseph (Seffi) Naor, and Roy Schwartz. 2012. A Tight Linear Time (1/2)-Approximation for Unconstrained Submodular Maximization. In *Proceedings of the 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science (FOCS '12)*. IEEE Computer Society, Washington, DC, USA, 649-658.

[7] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, Complexity of Computer Computations, pages 85103. Plenum Press, 1972.

[8] A. Schrijver, A combinatorial algorithm minimizing submodular functions in strongly polynomial time, Journal of Combinatorial Theory, Series B 80 (2000) 346–355.

[9] Uriel Feige,Vahab S. Mirrokni, and Jan Vondrák. Maximizing non-monotone submodular functions. In FOCS, pages 461 471, 2007.