Online sampling algorithm for submodular maximization

Case study: application to influence maximization on social networks

Abstract

Submodular maximization is a theoretical abstraction of many optimization problems appearing in a wide variety of applications such as viral marketing, image segmentation, information retrieval, etc. The standard method to get an approximately optimal solution is to run a greedy algorithm that selects the item with the maximal marginal contribution to the current solution at each step. In this paper we consider a novel algorithm, which is called Backward Greedy with Down Sampling (abbreviated to BGDS), that deletes elements with the minimum marginal contribution to a random feasible set sampled uniformly at random among currently available items.

We focus on the task of influence maximization (special case of general submodular maximization) that arises in the viral marketing application. We consider the standard independent cascade model over social networks. We implement an efficient value oracle for our submodular function and run multiple experiments of the BGDS algorithm on synthetic instances as well as real data of co-authorship graphs from arXiv. The main implementation challenge is to optimize the running time of our algorithm tailored to the independent cascade model, as we need to speed up the general submodular optimization algorithm, which is too slow for the typical network applications and datasets.

Keywords: submodular maximization, influence maximization, down sampling

Contents

1 Introduction

1.1 Submodular Maximization

Submodular optimization is an area extensively studied in operation research, machine learning, theoretical computer science. It has deep theoretical consequences and wide practical applications everywhere. Its applications include those of viral marketing, information gathering, image segmentation, document summarization, speeding up satisfiability solvers and so on.(see, e.g., [1] for a list of applications.)

In abstract terms, submodularity is a property of a function defined on the subset of a ground sets of elements. It is a discrete analog of a concave function formally specified by the property that the marginal contribution of every element i to a set is decreasing for larger sets. Namely, $f(S \cup \{j\}) - f(S) \leq f(T \cup \{j\}) - f(T)$ if T is a subset of S. An equivalent definition of a submodular function is that $f(S) + f(T) \geq f(S \cap T) + f(S \cup T)$ for any sets S, T.

In general, maximizing over a submodular function is NP-hard, even in the condition where there is no constraint. When there is a cardinality constraint and the submodular function is monotone, we can not approximate within a ratio better that $1 - \frac{1}{e}$. However, to minimize over a submodular function is not hard. We can solve it in polynomial time by minimizing the Lovász extension[2], which is convex. In this article, we mainly talk about maximization over a monotone increasing submodular function under the cardinality constraint, with influence maximization in social network as a concrete example, which will be described in details in the remaining part of the paper.

We illustrate diverse range of applications of submodular optimization by a few examples below.

Sensor Deployment. For example, imagine that we need to deploy a few sensors in the water distribution to detect contamination. We define $f(S)$ as the area that is covered by the selected sensors set S. Observe that the marginal gain of each new sensor is diminishing as we install more other sensors, so f is submodular. Indeed, those sensors double cover more area of the given sensor.

The above scenario is captures by the `Max-k-Coverage` problem. Formally, it is given as follows. With a universal set U. Suppose $A_1, A_2, \cdots, A_n$ are all subsets of U. We want to select a index set S from $\{1, 2, \cdots, n\}$ with the cardinality constraint, so as to maximize the cardinality of the union set of A_i where i takes from all elements in S.

Max Cut. Suppose we have a graph $G = (V, E)$, where V is the set of vertices, E is the set of edges. A cut is defined by a vertex set S. An edge is in the cut if and only if its start vertex is in S and its end point is not in S. The max cut problem is to find a cut with the number of edges in it maximized (in weighted version, summation over all edges in the cut is maximized).

Notice the weight version of the max cut problem is among the classical Karp's 21 NP-complete problems, which reveals the importance of submodular optimization somewhat. And the max cut model is used widely in the field of Theoretical physics and Circuit designing.

Exemplar Based Clustering. Exemplar Based Clustering is a classical data mining application.[3] Suppose we wish to select a set of exemplars, that best represent a massive data set. One approach for finding such exemplars is solving the k-medoid problem, which aims to minimize the sum of pairwise dissimilarities between exemplars and elements of the dataset. More precisely, let us assume that for the data set V we are given a distance function d : $V \times V \to R$ such that $d(\,\cdot\,,\,\cdot\,)$ encodes dissimilarity between elements of the underlying set V. Then, the k-medoid loss function can be defined as follows:

$$L(S) = \frac{1}{|V|} \sum_{e \in V} \min_{v \in S} d(e, v)$$

By introducing an auxiliary element e_0 (e.g., $= \mathbf{0}$, the all zero vector) we can turn L into a positive monotone submodular function:

$$f(S) = L(\{e_0\}) - L(S \cup \{e_0\})$$

And our objective is to maximize f, with some cardinality constraints.

1.2 Influence Maximization

People have been studying the social network for quite a long time. Several famous results, from random graph theory as well as network centrality theory, have become basic tools for people to get a better understanding of the structure of a social network. Many algorithms have been developed on social network for different applications. The social network is the graph that models relationships and interactions among a certain group of people. Usually, vertices are used to represent the individual, and edges are used to represent the relation between two different individuals. And the weight in an edge represents the strength of the relation. A classical task in such a social network graph is to identify important vertices.

One such notion of importance is influence, that naturally arises in the context of advertising campaigns on social networks. More specifically, imagine an advertiser who would like to run an advertising campaign on the network. She can select a few initial nodes and convince them to advertise the product to their friends. Some of these friends might buy the product and further advertise it to their friends by word-of-mouth and so on (see e.g. [4]). The objective is to maximize the number of people who are influenced, i.e., those who are convinced to buy the product at the end. The "influence" in the above example may not only refer to convincing to buy a new product, but also to willingness to adopting an innovation, donating to a charity, reading an article and etc. Influence maximization is different than many other centrality measures, since selecting vertices with highest degree of some kind of other notion of centrality (like degree centrality, distance centrality, spectrum centrality) as the initial set might be suboptimal. That is because central nodes tend to cluster, and for each cluster, there is no need to select too many of nodes in them.

David Kempe, Jon Kleinberg and Ev´a Tardos [5] formalized the problems of two popular kinds of influence maximization models as submodular maximization problems. One of these models is called Independent Cascade Model, and the other is called Linear threshold Model. In this article, we focus on the former cascade model.

Independent Cascade Model. In Independent Cascade Model, every edge from vertex u to v is assigned with a weight $p_{u,v}$, representing the probability that the influence is spread from u to v through this edge. The influence propagation process is as follows. First, we select the initial set of vertices A_0 and activate them. Any vertex that ever becomes active stays active until the end of the process. Every activated node u will have a onetime opportunity to influence (i.e., make active) its inactive neighbors in the network. Specifically, each inactive neighbor v is influenced (i.e., becomes active) by u with the probability $p_{u,v}$. Those spread-of-influence events happen independently over all neighbors of u.

The model is "independent" in the sense that influence propagation through a certain edge can happen at most once. This is because only vertices that were just activated in last round will try to influence other vertices. So the flips of coins for any edge is independent with each other. This property helps us to analyse the problem easier and implement the algorithm with higher efficiency. It captures some important characteristics of influence spread and is conceptually simple for people to understand and study. It has been proved that influence maximization in this model is submodular, and it is monotone increasing obviously. So we take it as a case to evaluate performance of a new general purpose submodular optimization algorithm for the influence maximization under Independent Cascade Model that is put forward hereinbelow.

1.3 Methods for Submodular Maximization

Here we talk about methods to maximize the nonnegative monotone submodular function with cardinality constraints.

Greedy Algorithm. A typical way to deal with submodular maximization is the greedy algorithm.[6] It tries to construct the solution step by step with a partial solution. We keep a set S in the process of doing greedy decision. Initially, S is empty. Whenever the size of S does not exceed the cardinality constraint, we add the vertex i with the highest marginal gain to S, which means, $f(S\cup\{i\})$ is maximal possible for the set S at this step and for all $i \in [n] \setminus S$.

Classical analysis shows that the greedy algorithm achieves a $(1 - \frac{1}{e})$ ratio of approximation.[6] And it's optimal because people have proved that to approximate with a ratio $(1 - \frac{1}{e} + \epsilon)$ for general

nonnegative monotone submodular function is NP-hard.[7]

The greedy algorithm is simple enough, and it gives a good approximation guarantee. Besides, in many situations. it works very well. Therefore, we take the greedy algorithm as the baseline in this paper.

Heuristics, Local Search and Meta-heuristics. Different heuristic algorithms are designed for different special submodular problems, and most of them are quite intuitive, trying to catch some aspects of the problem. For influence maximization of the Independent Cascade Model, "Degree Discount Heuristic (DDH)"[8], "Cordasco et al.'s Method"[9] are two of the most popular methods.

As a special case of combinational optimization problems, the local search and meta heuristics can be taken to solve submodular maximization. For the local search, the neighbour space is usually very large, so we need to use Best from Multiple Selections [10] (BMS) to speed up. For many meta heuristics, such as genetic algorithm, discrete particle swarm optimization algorithm, however, can not scale up.

All of these kinds of methods do not give a approximate guarantee and are incomplete inherently, but in some instances, their performances are good according to some empirical researches.

Up and Down Sampling. The method is used for general submodular optimization on parallel machines. Note that the greedy algorithm is highly sequential algorithm. Specifically, all the steps where we add an element to the current greedy solution S must be done in that specific sequence and thus cannot be run on parallel machines. In other words, the number of rounds of the greedy algorithm on any number of parallel machines is at least k. Note that a typical bottleneck for a parallel computing task is the synchronization steps between different computers. I.e., it is highly desirable to minimize the number of communication rounds between the parallel machines. On the other hand, we can run many asynchronous queries of computing $f(X)$ for independent sets X in parallel. In the formal parallel computation model for general submodular maximization, we aim to minimize the number of adaptive rounds in which parallel machines can exchange information and allow any polynomial in n number of asynchronous queries and other polynomial in n computations in between two consecutive rounds.

For a long time, people know little about the adaptivity complexity of submodular maximization. We only know it is between 1 and $\Omega(n)$ until Singer et al.(2018) [11] put forward a algorithm that needs $O(\log n)$ sequential rounds and guarantees an approximation which is close to $\frac{1}{3}$ within any small ϵ. They use a novel method based on expectation of the discrete uniform set taken from some certain distributions.

The down sampling, is one component of the algorithm. Given a big candidate set N_t, It calculates the expectation on the distribution that is taken uniformly from the sets of generated by deleting an element from N_t (In a formal expression, $N_t - \{j\}$) of size k. And then it removes the elements whose corresponding value is less than a given threshold in a round to get a new candidate set N_{t+1}.

The up sampling, on the other hand, starts from a small set S. In a round, the algorithm generates a series of sets by adding an element to S which is not belongs to S. It calculates the expectation on the k-size discrete uniform distribution from those generated sets. And then the algorithm adds those elements with highest expectation to S.

The Up and Down sampling methods give us some inspirations, and we want to explore the power of the down sampling method in this paper. The idea behind the down sampling is to use average value to deduce maximum value. Intuitively, this method works when the optimization landscape is not very sharp, and the solutions with high values tend to cluster.

1.4 Paper Structure

Here, we give the outline of the paper. In section 1, we talk about the background, concept and applications of submodular functions, as well as the the popular method to solve general submodular maximization problems. Besides, we introduce the influence maximization model in social network.

In section 2, we give the formal definition of some important concepts with some well-known theorems and facts. Besides, we give the framework of the first version of our algorithm BGDS (Backward Greedy with Down Sampling).

In section 3, we optimize our algorithm in different ways with some theoretical inductions, proofs and analyses included.

In section 4, we show the experimental results in some typical settings where the random graph, grid graph and coauthor graph are included.

Section 5 is the conclusion of the whole paper.

2 Preliminary

2.1 Submodularity

Definition 2.1 (Submodularity)**.** A set function $f : 2^X \to R$ is submodular if and only if

$$f(S \cup \{j\}) - f(S) \leq f(T \cup \{j\}) - f(T)$$

for any $T \subset S \subset X$ and $j \in X$, where X is the whole set.

Lemma 2.1. A set function $f : 2^X \to R$ is submodular if and only if

$$f(S \cup T) + f(S \cap T) \leq f(S) + f(T)$$

for any set $S \subset X$, $T \subset X$. [12]

Sometimes, people take the lemma as the definition of submodularity and take Definition 2.1 as a lemma. But we think Definition 2.1 is more natural and useful in this paper.

Definition 2.2 (Monotone Set Function)**.** A set function $f : 2^X \to R$ is monotone (increasing) if and only if

$$f(T) \leq f(S)$$

for any $T \subset S \subset X$, where X is the whole set.

Next we put forward the formal definition of submodular maximization with the cardinality constraint.

Definition 2.3 (Submodular Maximization with the Cardinality Constraint)**.** Suppose f is submodular and monotone, then the following optimization problem is submodular maximization with the cardinality constraint

$$\max \quad f(S) \quad \text{s.t. } |S| \leq k \quad \text{where } |S| \text{ is the size of } S \tag{1}$$

Theorem 2.2. *Submodular Maximization with the Cardinality Constraint is NP-hard.[6]*

This theorem can be shown by a naive reduction from the famous NP-complete problem max cut. So, generally, it is intractable to design a perfect algorithm for submodular maximization with the Cardinality constraint. And there is a stronger result.

Theorem 2.3. *It is NP-hard to approximate (1) with a ratio* $(1 - \frac{1}{e} + \epsilon)$ *for any* $\epsilon > 0$. *[7]*

The following theorem reveals the superiority of the greedy algorithm for the submodular maximization with the cardinality constraint considering this hardness.

Theorem 2.4. *The greedy algorithm approximates (1) with a factor* $(1-\frac{1}{e})$*, given an oracle to calculate* f*.[6]*

If we can not calculate f accurately in polynomial time, we still have a result which is a little weaker.

Theorem 2.5. *The greedy algorithm approximates (1) with a factor* $(1-\frac{1}{e}-O(k\epsilon))$ *using* g *as a proxy of* f*, if* g *is an universal* ϵ*-approximator of* f*, which means* $(1-\epsilon)f(S) \leq g(S) \leq (1+\epsilon)f(S) \quad \forall S \subset X$*. [13]*

2.2 Influence Model

Definition 2.4 (Social Network)**.** A social network N is a weighted graph (V, E, p), where V is the set of vertices, $E \subset V \times V$ is the set of edges, and $p : E \to R^+$ is a function indicating the weight of an edge. A vertex represents an individual, an edge represents the relation between two individuals, and the weight assigned with an edge represents the strength of the relation. Without misleading, we use the notation $e_{u,v}$ to represent the edge (u, v) , and $p_{u,v}$ to represent the weight $p((u, v))$ for convenience.

Definition 2.5 (Independent Cascade Process)**.** Give a social network $N = (V, E, p)$, and let $p_{u,v} \in [0, 1]$ represent the activation probability from u to v, an influence spread process with initial set A_0 can be described by the following pseudocode.

Require: $N = (V, E, p)$, $A_0 \subset V$
  for $i \leftarrow 0, 1, \cdots$ **do**
    if $i = 0$ **then**
      $D \leftarrow A_0$
    else
      $D \leftarrow A_i - A_{i-1}$
    end if
    for $u \in D$ **do**
      for $e_{u,v} \in E$ **do**
        if $v \notin A_i$ **then**
          With probability $p_{u,v}$: $A_{i+1} \leftarrow A_{i+1} \cup \{v\}$
        end if
      end for
    end for
    if $A_{i+1} = A_i$ **then**
      $T \leftarrow A_i$
      return T
    end if
  end for

Definition 2.6 (Independent Cascade Process with Flipping Coins Initially)**.** Given the same setting as Definition 2.5, an influence spread process with Flipping Coins initially with initial set A_0 can be described by the following pseudocode.

Require: $N = (V, E, p)$, $A_0 \subset V$
  $E' \leftarrow E$
  for $e_{u,v} \in E$ **do**
    With probability $1 - p_{u,v}$: $E' \leftarrow E' - e_{u,v}$
  end for
  for $i \leftarrow 0, 1, \cdots$ **do**
    if $i = 0$ **then**
      $D \leftarrow A_0$
    else
      $D \leftarrow A_i - A_{i-1}$
    end if
    for $u \in D$ **do**
      for $e_{u,v} \in E$ **do**
        if $v \notin A_i$ **then**
          $A_{i+1} \leftarrow A_{t+1} \cup \{v\}$
        end if
      end for
    end for
  end for

Notice after getting E', the process is a simple breath first search (BFS).

The difference between process in Definition 2.5 and in Definition 2.6 is that the former determines the existence of an edge online (flip the coin when needing it), and the poster determines the existence

of an edge offline (flip the coin initially). In essence, they are equivalent. This is because it does not matter you flip the coin when you need it or flip the coin initially and hide it and see it when you need it.

Lemma 2.6. Influence process in Definition 2.5 and in Definition 2.6 are equivalent. [5]

This kind of equivalence is important for us to implement our algorithm in a way with high efficiency.

Definition 2.7 (Influence Function in Independent Cascade Model)**.** Let $T_i(S)$ be the returned influenced set in the ith running in the independent cascade process with seed set S, let the influence function be $f : 2^X \to R$, $f(S) = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} |T_i(S)|}{n}$, which is, the expectation of influenced set size with seed set S.

Lemma 2.7. Influence function in Independent Cascade Model is monotone submodular.[5]

Lemma 2.8. Maximizing Influence function in Independent Cascade Model is NP-hard.[5]

2.3 Algorithm Framework

Next, we give the framework of the greedy algorithm as well as our basic Backward Greedy with Down Sampling algorithm. (BGDS)

Algorithm 1 Greedy Framework

```
procedure GREEDY(V, k, f)
    S ← ∅
    while |S| < k do
        Let j ∈ argmax_{i∉S} f(S ∪ {i})
        S ← S ∪ {j}
    end while
end procedure
```

Algorithm 2 Backward Greedy with Down Sampling Framework

```
procedure BGDS_v1(V, k, f)
    S ← V
    while S > k do
        for a ∈ S do
            D_{S\{a}} ← uniform over subsets of S − {a} of size k
        end for
        Let j ∈ argmax_{a∈S} E_{R∼D_{S\{a}}}[f(R)]
        S ← S − {j}
    end while
end procedure
```

3 Implementation

3.1 Complexity Analysis of BGDS_v1

In general, the BGDS_v1 is quite slow. The time complexity is $O(|V|^2 so)$, where $|V|$ is the number of elements in the whole set, s is the number of trials to estimate the expectation within a reasonable error, and o is the time complexity of the oracle to get $f(S)$ given S.

In the outer loop, there are $O(|V|)$ iterations to delete elements from S one in a round. In each round, we need to generate $O(|V|)$ distributions on average. And for each distribution, we need s calls of the oracle to estimate the expectation where the running time of oracle is $O(o)$, so the total complexity is $O(|V|^2 so)$

This complexity means it is insane to run BGDS in a non-toy instance in general. For example, for the influence maximization problems of the Independent Cascade Model, the running complexity

of the oracle is $O(g|E|)$ to have a reasonable estimation of $f(S)$, where g is the number of trials of Independent Cascade Process in Definition 2.5, which depends on the variance of $f(S)$ theoretically , $|E|$ is the size of edge set. In this setting, the total complexity is $O(|V|^2 sg|E|)$. Based on some experiments, to set $g = 10000$ gives a relative stable result. And the simulation of the process in Definition 2.5 can be done with methods that is similar to the Breadth First Search. If we run the BGDS in a graph $G = (V, E)$, where $|V| = 200, |E| = 1000$, and set $s = 1000$, the total amount of calculation will be about 6.4×10^{14}, which costs about 6 to 7 cpu running days for an ordinary single core cpu in 2022.

3.2 Faster Implementation of BGDS in ICM

Now, we focus on how to implement the BGDS for influencing maximization problem in Independent Cascade Model efficiently.

Good news is, if the influence network is symmetric, we can implement the algorithm in a much faster manner. The following definition and theorem shows that we can replace a symmetric directed model with an undirected model, on which snapshots tricks can be made to accelerate the algorithm quite a lot.

Definition 3.1 (Independent Cascade Process of Undirected Network with Flipping Coins initially)**.** Given an symmetric social network $N = (V, E, p)$, and let $p_{u,v} \in [0, 1]$ be the weight assigned to the edge $e_{i,j}$. (We have $e_{i,j} \in E \iff e_{j,i} \in E$ and $p_{i,j} = p_{j,i}$ considering the symmetry.) An influence spread process of undirected network with Flipping Coins initially with initial set A_0 can be described by the following pseudocode.

Require: $N = (V, E, p)$, $A_0 \subset V$
  $E' \leftarrow E$
  for $e_{u,v} \in E$ **do**
    if $e_{u,v} \in E'$ **then**
      With probability $1 - p_{u,v}$: $E' \leftarrow E' - e_{u,v} - e_{v,u}$
    end if
  end for
  for $i \leftarrow 0, 1, \cdots$ **do**
    if $i = 0$ **then**
      $D \leftarrow A_0$
    else
      $D \leftarrow A_i - A_{i-1}$
    end if
    for $u \in D$ **do**
      for $e_{u,v} \in E$ **do**
        if $v \notin A_i$ **then**
          $A_{i+1} \leftarrow A_{t+1} \cup \{v\}$
        end if
      end for
    end for
  end for

Theorem 3.1. *Influence process in Definition 3.1 and in Definition 2.5 are equivalent if the social network graph is symmetric.*

Proof. In Definition 2.5, the event that u tries to influence v and the event that v tries to influence u are impossible to happen both. So when u tries to influence v or v tries to influence u, to flip a coin immediately is equivalent to move to see the flipping result corresponding to the undirected edge (u, v) or (v, u) done in the beginning, which means, to check whether (u, v) exists in E' of the process in Definition 3.1. So the process in Definition 3.1 is equivalent to running a BFS in $G' = (V, E')$ in the process of Definition 2.5. □

Remark. Notice that the difference between the process in Definition 2.6 and in Definition 3.1 is only that in Definition 2.6, we determine the existences of two edges which is reverse to the other separately

with two flips, but in Definition 3.1, we determine them simultaneously with one flip. From another perspective, if we see the existence of an directed edge as a binary random variable, the correlation coefficient is 0 in Definition 2.6, whereas 1 in Definition 3.1, for any two random variables whose edge is reverse to the other.

Next, we optimize the original version of BGDS for the symmetric ICM in three steps.

3.2.1 Snapshots Techniques and Preprocessing

In the original version of Backward Greedy with Down Sampling, we need to run g simulations whenever we try to calculate $f(R)$, it costs much time, and because we do not know what the graph structure will be with instantiation in Definition 2.6, there is no room for us to optimize. But if the social network graph is symmetric, we can use the equivalence of Definition 3.1 and Definition 2.5. Instead of generating g graphs whenever calculating $f(R)$, we generate g graphs initially, and reuse the g graphs when needing to calculate $f(R_1), f(R_2), \cdots$. The graphs are generated in a fashion of the first part of Definition 3.1, resulting to some undirected graphs.

Next we can do some preprocessing works to accelerate the evaluation of $f(S)$. For each graph instance generated in the very beginning, say G_i, we find all connected components and the size of them using BFS or the disjoint union set. So in this graph instance, which is corresponding to a trial in Definition 3.1, the size of the influenced set is the total size of the components that is hit by the seed set R. And $f(R)$ is estimated by the average of the former values of all g graph instances.

In the implementation, V stands for the set of vertices, E stands for the set of edges, the triple (V, E, p) stands for a social network, and the pair (V, E) stands for a graph instance. In each graph instance (V_i, E_i), we run the procedure "preprocessing" on it to get two important variables – "vertex2Compnent" and "componentSize", which represents the map from a vertex to it's corresponding connected component id and the sizes of all components in the graph instance respectively.

In the procedure "preprocessing", we sustain a set $visSet$ representing the vertices that have been visited by the BFS procedure. For any vertex that has not been in $visSet$, we start a procedure BFS to find all vertices that is connected to v in the undirected graph. After that, we count the values of the map "componentSize" in procedure $CountValues$. The details of the two procedure are as follows.

procedure BFS(v,E, $visSet$, $componentCnt$)
  $partialVertex2Component \leftarrow emptyMap$
  $q \leftarrow emptyQueue$
  $q.push\_back(v)$
  while $q \neq emptyQueue$ **do**
    $x \leftarrow q.front()$
    $q.pop()$
    $visSet \leftarrow visSet \cup \{x\}$
    $partialVertex2Component[x] \leftarrow componentCnt$
    for $e_{x,y} \in E$ **do**
      if $y \notin visSet$ **then** $q.push\_back(y)$
      end if
    end for
  end while
  return $visSet, partialVertex2Component$
end procedure
procedure CountValues(map)
  $valCnts \leftarrow emptyMap$
  for $(k, v) \in map$ **do**
    if $v \in valCnts.keys$ **then**
      $valCnt[v] \leftarrow valCnt[v] + 1$
    else
      $valCnt[v] \leftarrow 1$
    end if
  end for
end procedure

After that, we delete the vertex according to the expectation of the oracle. The "oracle" procedure, take a seed set S, a group of vertex2Components, and a group of componentSizes, two of which are generated by the "preprocessing" procedure. And it finds the total size of all connected components that contain at least 1 vertex of S fast, by using the map vertex2Component and componentSize. In details, it iterators all vertices of S, finds the connected component of a vertex, and identifies whether it is the first time to find this component by recording a map. If it is, the component size will be the total value (sum_i in the pseudocode), and the average of the total value will be returned after the works on all preprocessed graphs are done.

The left is the same as the basic framework in Algorithm 2. Now, we move on to analysis the complexity of the algorithm. We implements all sets, maps in a hash table way, so we can store and access an element in $O(1)$. (Actually, it is simple enough because the keys are in the scope of the integers from 1 to $|V|$, so we can implements them with simple arrays (or std::vector $\langle$int$\rangle$ in C++), which is equivalent to an identity hashing mapping.)

The time complexity of the procedure "GetGraphInstance" is $O(|E|)$, becauce for each pair of edges, we flip a coin to decide the existence of them in the generated graph instance.

The time complexity of the procedure "Preprocessing" is $O(|V|+|E|)$, because the loop which calls the procedure "BFS" is $O(|V|)$ for each vertex will be tagged in the state "visited" once. However, $O(|E|)$ edges may be checked in the BFS. And the procedure "CountValues" is $O(|V|$ because there are only $O(c)$ access and storage of an hash map, where c is the number of connect component in a graph instance, which is bounded to be smaller than $|V|$.

The time complexity of the procedure "Oracle" is $O(g|S|)$, because we need to iterate g graph instances, in which $|S|$ vertices will be processed in O(1). Notice that in the whole algorithm, we only evoke the oracle with $|S| = k$, where k is the predetermined cardinality constraint value.

The total complexity of the optimized algorithm is $O(g|E|+|V|^2sgk) = O(|V|^2sgk)$, considering all loops and invoking of other procedures, where s is the number of trials to estimate the expectation, g is the number of trials to estimate the influence function $f(S)$, and k is the predetermined cardinality constraint value. Typically, k is way less than $|E|$, so compared to the complexity $O(|V|^2sg|E|)$ in the original version, the optimized one improves the efficiency very much.

Algorithm 3 Backward Greedy with Down Sampling with Snapshots Techniques

procedure BGDS_v2(V,E,p)
  for $i \leftarrow 1, \cdots, g$ **do**
    $V_i, E_i \leftarrow$ GetGraphInstance(V, E, p)
    $vertex2Component_i, componentSize_i \leftarrow$ preprocessing(V_i, E_i)
  end for
  $vertex2Components \leftarrow [vertex2Component_1, vertex2Component_2, \cdots, vertex2Component_g]$
  $componentSizes \leftarrow [componentSize_1, componentSize_2, \cdots, componentSize_g]$
  $S \leftarrow V$
  while $S > k$ **do**
    for $a \in S$ **do**
      $\mathcal{D}_{S\setminus\{a\}} \leftarrow$ uniform over subsets of $S - \{a\}$ of size k
    end for
    Let $j \in \text{argmax}_{a\in S}\, \mathbb{E}_{R\sim\mathcal{D}_{S\setminus\{a\}}}[\text{Oracle}(R, vertex2Components, componentSizes)]$
    $S \leftarrow S - \{j\}$
  end while
end procedure

procedure GetGraphInstance(V,E, p)
  $E' \leftarrow E$
  for $e_{u,v} \in E$ **do**
    if $e_{u,v} \in E'$ **then**
      With probability $1 - p_{u,v}$: $E' \leftarrow E' - e_{u,v} - e_{v,u}$
    end if
  end for
  return (V, E')

end procedure

procedure PREPROCESSING(V, E)
  $visSet \leftarrow \emptyset$
  $vertex2Compnent \leftarrow emptyMap$
  $componentCnt \leftarrow 0$
  for $v \in V$ **do**
    if $v \notin visSet$ **then**
      $componentCnt \leftarrow componentCnt + 1$
      $visSet, partialVertex2Component \leftarrow BFS(v, E, visSet, componentCnt)$
      $vertex2Compnent \leftarrow vertex2Compnent \cup partVertex2Component$
    end if
  end for
  $componentSize \leftarrow CountValues(vertex2Compnent)$
  return $vertex2Compnent, componentSize$
end procedure

procedure ORACLE($S, vertex2Components, componentSizes$)
  $g \leftarrow |vertex2Components|$
  $vertex2Component_1, vertex2Component_2, \cdots, vertex2Component_g \leftarrow vertex2Components$
  $componentSize_1, componentSize_2, \cdots, componentSize_g \leftarrow componentSizes$
  for $i \leftarrow 1, \cdots, g$ **do**
    $sum_i \leftarrow 0$
    $hitComp \leftarrow \emptyset$
    for $v \in S$ **do**
      $compId \leftarrow vertex2Componet_i[v]$
      if $compId \notin hitComp$ **then**
        $hitComp \leftarrow hitComp \cup \{compId\}$
        $sum_i \leftarrow sum_i + componentSize_i[compId]$
      end if
    end for
  end for
  return $\frac{\sum_{i=1}^{g} sum_i}{g}$
end procedure

3.2.2 Calculate Expectation Analytically

With the information acquired in the preprocessing step, we can do more than just accelerating the evaluation of the influence function f. In fact, we can calculate $\mathbb{E}_{R\sim\mathcal{D}_a}[f(R)]$ analytically in a very fast way.

In influence maximization model, the influence function f defined in Definition 2.7 is some kind of forms of expectation actually. And the associate distribution is about the generated graph instance with the generating probability. So $\mathbb{E}_{R\sim\mathcal{D}_S}[f(R)]$ can be rewritten into $\mathbb{E}_{R\sim\mathcal{D}_S}[\mathbb{E}_{G\sim\mathcal{D}_G}[h(G,R)]]$, where $h(G,R)$ is the number of influenced set in an influence process given the seed set R and the graph instance G. Thus, we can change the order of the two expectation notation. Formally, $\mathbb{E}_{R\sim\mathcal{D}_S}[\mathbb{E}_{G\sim\mathcal{D}_G}[h(G,R)]] = \mathbb{E}_{G\sim\mathcal{D}_G}[\mathbb{E}_{R\sim\mathcal{D}_S}[h(G,R)]]$.

In other words, instead of sampling the set R and the sample the graph instance G, we can sample the graph instance G and then sample the set R to get the same result. The benefit is, we are not really bounded to sample the set R given a graph instance G, but can calculate the expectation $\mathbb{E}_{R\sim\mathcal{D}_S}[h(G,R)]$ analytically. And that is the point.

For the simplicity of the formula, we take the notation w_i to represent the size of component i, which can be gained easily using the returned map "componentSize" after running procedure "preprocessing" on the graph instance $G = (V, E)$. We sustain an array $[a_1, a_2, \cdots, a_c]$ to denote the numbers of active vertices in every component. a_i is the number of active vertices in component i. Let the corresponding whole set for the discreet uniform distribution D_S be S, and the set of vertices in component i be C_i, we define active set for component i under distribution D_S as the intersection of S and C_i. Formally, $A_i = S \cap C_i$, $a_i = |A_i|$.

Let g_i be the contribution of number of influence set of component i, and recall k is the cardinality constraint value, which is also the size of R, we have

$$\mathbb{E}_{R\sim\mathcal{D}_S}[h(G,R)] = \mathbb{E}_{R\sim\mathcal{D}_S}[\sum_{i=1}^{c} g_i] = \sum_{i=1}^{c}\mathbb{E}_{R\sim\mathcal{D}_S}[g_i] = \sum_{i=1}^{c} w_i(1-\frac{\binom{|S|-a_i}{k}}{\binom{|S|}{k}}) \tag{2}$$

Here, the first equation is because of the definition of h and g_i, the second equation is due to the linearity of the expectation operator. For the third equation, considering component i either contributions 0 with probability $\frac{\binom{|S|-a_i}{k}}{\binom{|S|}{k}}$ (selecting k elements from $S-A_i$ v.s. selecting k elements from S), or contribute w_i with probability $1-\frac{\binom{|S|-a_i}{k}}{\binom{|S|}{k}}$, the proof is obvious.

So instead of calling the "oracle" procedure for s samples of sets to estimate the estimate the expectation, we can use formula (1) to calculate it. In this way, the total complexity changes from $O(|V|^2sgk)$ to $O(|V|^2cg)$, where c is average number of components for sampled graph instances, noticing that we can preprocess the needed binomial $O(|V|)$ coefficients in $O(|V|k)$ time.

We can do more with formula (1) to eliminate the coefficient c in complexity. For convenience, we take the notation $\Psi(S) = \sum_{i=1}^{c} w_i(1-\frac{\binom{|S|-a_i}{k}}{\binom{|S|}{k}}) = \mathbb{E}_{R\sim\mathcal{D}_S}[h(G,R)]$. Notice that the algorithm only cares the relative value of $\Psi(S-\{v\})$ for different vs. So we do not need to really calculate $\Psi(S)$. Let $u\in C_i$, $v\in C_j$, we have

$$\begin{aligned}
\Psi(S-\{u\})-\Psi(S-\{v\}) =& \left(\sum_{r\neq i} w_r(1-\frac{\binom{|S|-1-a_r}{k}}{\binom{|S|-1}{k}}) + w_i(1-\frac{\binom{|S|-1-(a_i-1)}{k}}{\binom{|S|-1}{k}})\right) - \\
& \left(\sum_{r\neq j} w_r(1-\frac{\binom{|S|-1-a_r}{k}}{\binom{|S|-1}{k}}) + w_j(1-\frac{\binom{|S|-1-(a_j-1)}{k}}{\binom{|S|-1}{k}})\right) \\
=& \frac{w_i\left(\binom{|S|-1-a_i}{k}-\binom{|S|-a_i}{k}\right) - w_j\left(\binom{|S|-1-a_i}{k}-\binom{|S|-a_i}{k}\right)}{\binom{|S|-1}{k}} \\
=& \frac{-w_i\binom{|S|-1-a_i}{k-1} + w_j\binom{|S|-1-a_j}{k-1}}{\binom{|S|-1}{k}}
\end{aligned} \tag{3}$$

So, $\Psi(S-\{u\}) > \Psi(S-\{v\})$ is equivalent to $w_i\binom{|S|-1-a_i}{k-1} < w_j\binom{|S|-1-a_j}{k-1}$. Let $\psi(v) = w_i\binom{|S|-1-a_i}{k-1}$, to find u to make $\Psi(S-\{x\})|_{x=u}$ maximal is equivalent to find u to make $\psi(x)|_{x=u}$ minimum. And $\psi(x)$ can be evaluated in $O(1)$ with binomial coefficients preprocessed, so the acceleration is considerable.

Actually, there are multiple graph instances on which evaluations of Ψ are needed, and it is the average values of the evaluations that direct the algorithm to choose the element to delete from S. So, we are to find the $u\in S$ that minimizes $\frac{1}{g}\sum_{i=1}^{g}\psi^{(i)}(x)|_{x=u}$, where $\psi^{(i)}(x) = w^{(i)}_{y(x,i)}\binom{|S|-1-a^{(i)}_{y(x,i)}}{k-1}$, where $y(x,i)$ is the component id of the vertex x in the graph instance i, $w^{(i)}_y$ and $a^{(i)}_y$ maintains the corresponding size and active vertices set size of the component y in the graph instance i.

We can see the main difference between BGDS_v2 and BGDS_v3 is the criterion to select element to remove form the potential set S. In BGDS_v2, it is $\text{argmax}_{a\in S}\mathbb{E}_{R\sim\mathcal{D}_{S\setminus\{a\}}}[\text{Oracle}(\cdots)]$. In BGDS_v3, it is $\text{argmin}_{u\in S}\frac{1}{g}\sum_{i=1}^{g}\psi^{(i)}(u)$. Both of them are to estimate $\text{argmax}_{a\in S}\mathbb{E}_{R\sim\mathcal{D}_{S\setminus\{a\}}}[f(R)]$, but latter is much easier to compute and seems to be more efficient.

The time complexity of this BGDS_v3 is $O(|V|^2g)$. The deletion loop dominates the whole algorithm. And in that loop, to select the j to delete is $O(|S|g)$ for we are checking every element in S and there are g items, each of which can be calculated in $O(1)$, to sum up. And to maintain the arrays a is $O(g)$, which does not dominate. So, with $|S|$ decreasing from $|V|$ to k, the time complexity is $O(|V|^2g)$, which is another great improvement compared to the $O(|V|^2sgk)$ in BGFS_v2.

Algorithm 4 Backward Greedy with Down Sampling with Calculating Expectation Analytically

procedure BGDS_v3(V,E,p)
  for $i \leftarrow 1, \cdots, g$ **do**
    $V_i, E_i \leftarrow \text{GetGraphInstance}(V, E, p)$
    $vertex2Component_i, componentSize_i \leftarrow \text{preprocessing}(V_i, E_i)$
    for $v \leftarrow 1, \cdots, |V|$ **do**
      $y(v, i) := vertex2Component_i[v]$
    end for
    for $c \leftarrow 1, \cdots, |componentSize_i|$ **do**
      $w_c^{(i)} := componentSize_i[c]$
      $a_c^{(i)} \leftarrow w_c^{(i)}$
    end for
  end for
  $S \leftarrow V$
  while $S > k$ **do**
    Let $j \in \text{argmin}_{u \in S} \frac{1}{g} \sum_{i=1}^{g} \psi^{(i)}(u)$, where $\psi^{(i)}(u) = w_{y(u,i)}^{(i)} \binom{|S|-1-a_{y(u,i)}^{(i)}}{k-1}$
    $S \leftarrow S - \{j\}$
    for $i \leftarrow 1, 2, \cdots, g$ **do**
      $a_{y(j,i)}^{(i)} \leftarrow a_{y(j,i)}^{(i)} - 1$
    end for
  end while
end procedure

3.2.3 Two Stages Deletion (Batch Deletion)

During the experiments, we find that in the beginning, some $\frac{1}{g} \sum_{i=1}^{g} \psi^{(i)}(u)$ are quite close to each other, so it is hard to identify which element to remove. And when the potential set S is very large, there are many elements that are redundant, so we can delete not only an element but a batch of elements, which accelerate the algorithm a lot.

When the alternative set S is not very large, delete too many elements in a round will cause severe degeneration. So in that situation, we still delete one element in a round.

Our strategy is as follows, we divide the whole process of deleting elements into two stages. Let the alternative set be S. Given a super parameter α ($\alpha > 1$), If $|S| > \alpha\sqrt{|V|}$, we are in the stage one, the batch deletion stage. We delete $\sqrt{|V|}$ elements whose corresponding $\frac{1}{g} \sum_{i=1}^{g} \psi^{(i)}(x)$ is in the $\sqrt{|V|}$ smallest list. If $|S| \leq \sqrt{|V|}$, we are in the stage two, the one deletion stage, we delete the element one by one, just as we have been doing before this subsubsection.

And it is our final version of the algorithm BGDM. The difference between BGDS_v3 and BGDS_v4 is that in BGDS_v4 we add a branch **while** $|S| > \alpha|V|$ **do** $\cdots$ to delete multiple elements in a time. The time complexity now is $O(|E|g + |V|\sqrt{|V|}g + |V|k)$. Because in both stage 1 and 2, there are $O(\sqrt{|V|})$ iterations in the outer loop, and this loop may not dominate if the graph instances are quite dense (when $|E| > |V|\sqrt{|V|}$). And the $O(|V|k)$ comes from calculating $O(|V|)$ binomial coefficients. If we only care about the number of vertices in a sparse graph, i.e. $|V|$, the complexity is $O(|V|\sqrt{|V|})$, which is brilliant. This complexity result means the algorithm is runnable in a medium scale social network with tens of thousands vertices in an ordinary personal computer.

Algorithm 5 Backward Greedy with Down Sampling with the Batch Deletion

procedure BGDS_v4(V,E,p, α)
  for $i \leftarrow 1, \cdots, g$ **do**
    $V_i, E_i \leftarrow \text{GetGraphInstance}(V, E, p)$
    $vertex2Component_i, componentSize_i \leftarrow \text{preprocessing}(V_i, E_i)$
    for $v \leftarrow 1, \cdots, |V|$ **do**
      $y(v, i) := vertex2Component_i[v]$
    end for
    for $c \leftarrow 1, \cdots, |componentSize_i|$ **do**
      $w_c^{(i)} := componentSize_i[c]$
      $a_c^{(i)} \leftarrow w_c^{(i)}$
    end for
  end for
  $S \leftarrow V$
  while $|S| > \alpha\sqrt{|V|}$ **do**
    $b \leftarrow \min(\sqrt{|V|}, |S| - \alpha\sqrt{|V|})$
    $\{j_1, j_2, \cdots, j_b\} \leftarrow \text{argminK}_{u \in S} \left(\frac{1}{g} \sum_{i=1}^{g} \psi^{(i)}(u), b \right)$, where $\psi^{(i)}(u) = w_{y(u,i)}^{(i)} \binom{|S|-1-a_{y(u,i)}^{(i)}}{k-1}$
    $S \leftarrow S - \{j_1, j_2, \cdots, j_b\}$
    for $i \leftarrow 1, 2, \cdots, g$ **do**
      for $r \leftarrow 1, 2, \cdots, b$ **do**
        $a_{y(j_r,i)}^{(i)} \leftarrow a_{y(j_r,i)}^{(i)} - 1$
      end for
    end for
  end while
  while $|S| > k$ **do**
    Let $j \in \text{argmin}_{u \in S} \frac{1}{g} \sum_{i=1}^{g} \psi^{(i)}(u)$, where $\psi^{(i)}(u) = w_{y(u,i)}^{(i)} \binom{|S|-1-a_{y(u,i)}^{(i)}}{k-1}$
    $S \leftarrow S - \{j\}$
    for $i \leftarrow 1, 2, \cdots, g$ **do**
      $a_{y(j,i)}^{(i)} \leftarrow a_{y(j,i)}^{(i)} - 1$
    end for
  end while
end procedure

4 Experimental Results

We have done some experiments to test the performance of the algorithm BGDM in practice, and find that it always achieves comparable results as the baseline, the Greedy. And in some situations, it beats the Greedy.

We mainly consider three families of graphs. The first is the grid graphs, which is easy for visualization. The second is the Erdős–Rényi random graphs, which is diverse and can simulate many typical situations by changing the parameter for generating edges of the model. The above two families are artificial (generated by some Python scripts), so we can run algorithms on any scale and parameter to cover everything we care.

The third one is the co-authorship graph. We crawl the raw data from the e-print arXiv. And we extract the information about the co-authorship. The co-authorship is, for example, that person a and person b are both the writers of the same article. And it is widely believed that the co-authorship graph can represent some natures of social networks in real life.

The algorithms we run is the Greedy and the BGDS_v4. The Greedy is very straightforward, which uses the framework in Algorithm 1, with the oracle put forward in Algorithm 3 to evaluate f. And we find that snapshots techniques and preprocessing is also applicable in the Greedy for acceleration. For the sake of fairness, we also implemented it in the Greedy. Based on all the experiments we have done, we find that if the network is neither very sparse nor very dense, our BGSS_v4 outperforms the Greedy. Luckily, it is often the case for the social networks in the real life.

Algorithm parameters setting
In both algorithms (the Greedy, BGDS_v4), following the convention taken in [5], we set the number of graph instance $g = 10000$. When the algorithms return the solution set T, we do 160000 trials to get a estimator of f(T) with very high precision, which we see as the true value of $f(T)$. For the BGDS_v4, we set the super parameter $\alpha = 4.0$, and do not tune it during all experiments for simplicity and fairness.

Experimental Environment
Operation system: Windows 10 Home 19043.1586
CPU: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz (4 cores and 8 threads)
RAM: 16GB
Compiler: gcc 10.3.0
Compiling options: -Wall -g -O3

4.1 Grid graphs

Here we consider the square two-dimension grid graph, which is very symmetric. For instance, the following is a 3×3 grid graph, with 'o' representing the vertices, and '$---$' or ' | ' representing the edge.

```
o---o---o
|   |   |
o---o---o
|   |   |
o---o---o
```

For comparison of the quality of the solution of the two algorithms, we consider a medium size 35×35 grids, and set the influence probability $p = 0.5$ for every edge. In the above figure, the horizontal axis is the target size, which is the k mentioned before, which is about the cardinality constraint of the solution set. For each target set size range in $[5, 20]$, we conduct 5 experiments independently, and take the average to be the result.

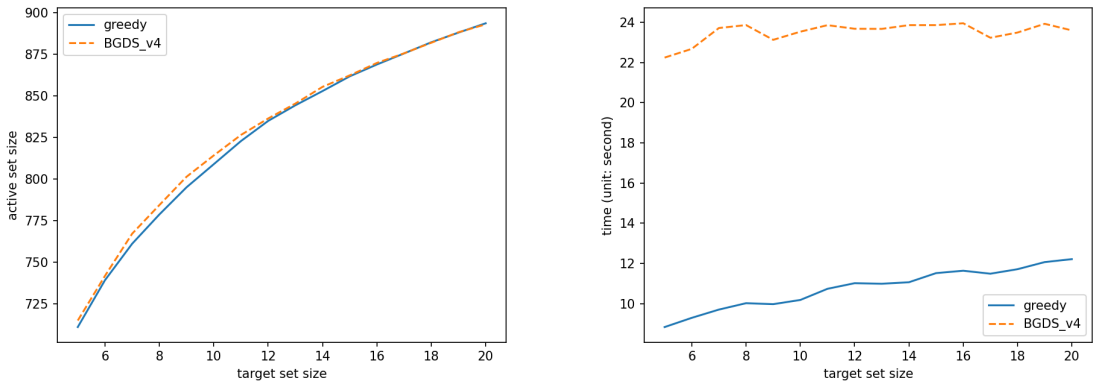

Figure 1: Result for the grid graphs with $|V| = 1225$, $p = 0.5$

In the left picture in figure 1, the orange dash line, which represents the solution value got by our BGDS algorithm, is almost above the blue solid line everywhere, which means our BGSD algorithm performs better uniformly the Greedy, except when the target size becomes big, the two algorithm perform both well equally. In the right picture, we can see the cost time does not dependents on the target size k in our BGDS algorithm, whereas the running of the Greedy depends linear on k, which validates our mathematical reduction in the former sections. Although the BGDS algorithm runs slower the greedy, it is acceptable in practice. And the time comparison will be omit in next subsections because the pattern of it does not change in our experiments.

For convenient visualization, we run the two algorithm on a 10×10 grid with $p = 0.5, k = 4$, and draw the gained solution set as follows.

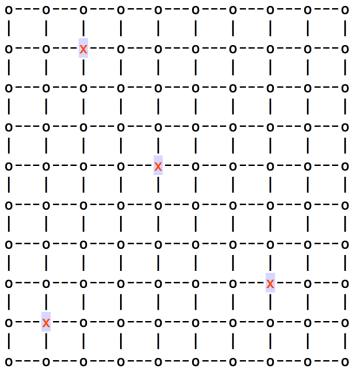

Figure 2: Solution set of the Greedy

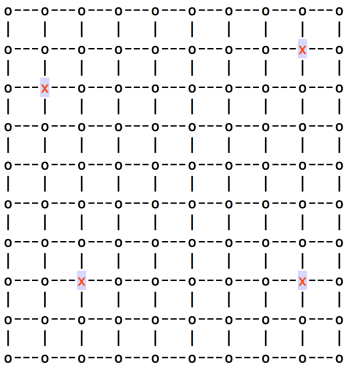

Figure 3: Solution set of the BGDS

The red 'x' is the in the selected set of the algorithm. We find that the choose a vertex in the center, which is suboptimal. And the BGDS algorithm can select vertices which are quite scattered. In this instance, let T_G, T_B be the set of selected vertices by the Greedy, the BGDS respectively. We have $\hat{f}(T_G) = 66.97$, $\hat{f}(T_B) = 68.99$, with the standard error about 0.36, where $\hat{f}$ is an estimator of the influence function f.

4.2 Random Graphs

We use the well-known Erdős–Rényi random graphs model to generate random graphs with different structures, after which we assign a universal influence probability p to all edges to form a social network. We take $|V| = 1000$, $p = 0.5$ in the experiment. The generating probability gp in the ER model is set to $\frac{0.9}{|V|}, \frac{1.1}{|V|}, \frac{2.0}{|V|}, \frac{3.0}{|V|}, \frac{0.9log|V|}{|V|}, \frac{1.1log|V|}{|V|}, \cdots$ to generate graphs with different structures.

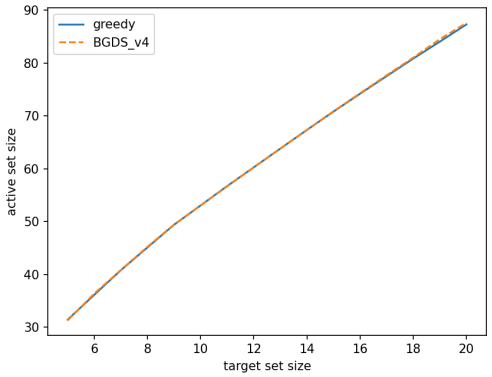

Figure 4: Random graph with $gp = \frac{0.9}{|V|}$

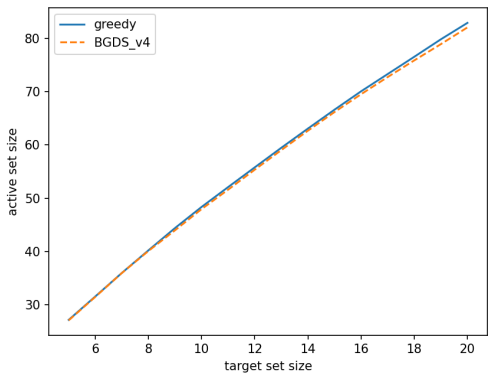

Figure 5: Random graph with $gp = \frac{1.0}{|V|}$

With the generating probability gp increasing, the graph become denser and more connective. When the graph is sparse, there is no giant connected component, and the connectivity in a connect component is not strong. The Greedy works because the chance that the Greedy makes a short-sighted decision is low, so it is near optimal.

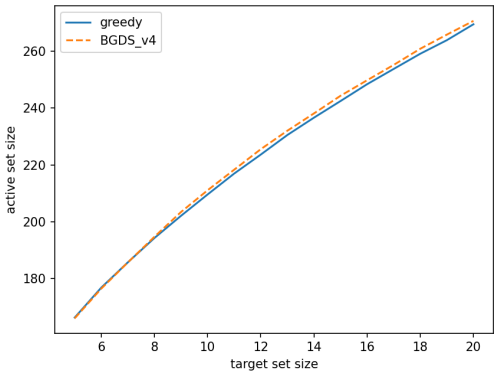

Figure 6: Random graph with $gp = \frac{2.0}{|V|}$

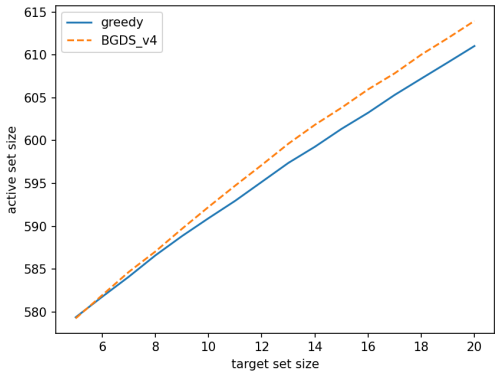

Figure 7: Random graph with $gp = \frac{3.0}{|V|}$

When the graph grows denser, the giant component appears, and there are many small components with size $O(\log(|V|))$. In this situation, our BGDS algorithm outperforms the Greedy.

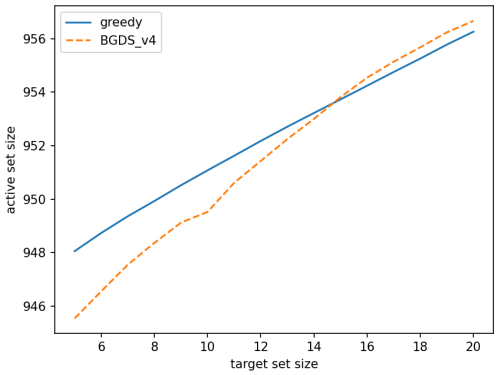

Figure 8: Random graph with $gp = \frac{0.9log|V|}{|V|}$

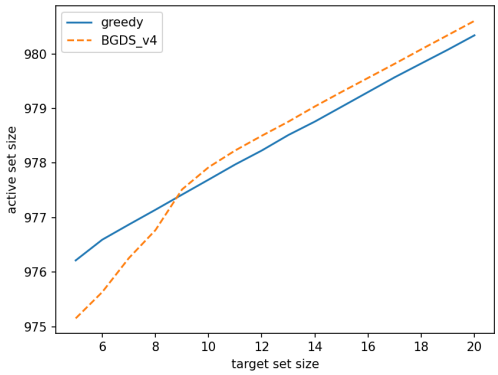

Figure 9: Random graph with $gp = \frac{1.1log|V|}{|V|}$

When the graph become very dense, there is a giant component with strong connectivity and a few isolated vertices. In this situation, the Greedy is near optimal because we only need to select 1 vertex for the giant connected component to influence most of the vertices in the giant component. But when the target size is a little big, the BGDS algorithm also give a good solution, or even better than the Greedy slightly. If the random graph becomes denser, the generated graph is connected with probability 1.0 according to the random graph theory. Moreover, if the connectivity is very very strong(the influence probability p should also be put into consideration), the problem is trivial. We just need to choose any set to influence all the vertices.

4.3 Co-authorship Graphs

We take the co-authorship data from the whole lists of papers of the recent 10 years in the section computational complexity of computer science in the arXiv database(www.arxiv.org). Each researcher who has at least one paper with co-author(s) is corresponding to a vertex in the collaboration network. We insert a bidirectional edge for each pair of authors in every paper that has at least 2 authors. This may lead to parallel edges when 2 researchers have co-authored more than one papers, we keep them because they represent the strength of the relation to some extent. Actually, when there are $c_{u,v}$ edges between u and v, and influence try to go through any of them independent with probability $p_{u,v}$, a single undirected edge between u and v with influence probability $1-(1-p_{u,v})^{c_{u,v}}$ is equivalent. So we

substitute those $c_{u,v}$ parallel edges with the single edge. [5] We assign a universal influence probability $p = 0.1$ for every original edge. Finally, we get a network $N = (V, E, p)$ with $|V| = 6649$, $|E| = 28984$.

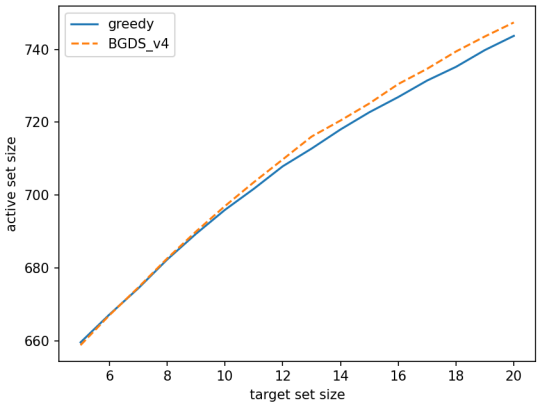

Figure 10: Result for co-authorship network

In the figure, we can find that when the target size is small, the result of two algorithm are almost the same. Actually, if we dig in to compare the selected set of the two algorithm, they are quite similar, and usually there are only 0 to 1 different elements in the two set. When the target size become large (more than 10), our BGDS_v4 achieves higher active set size. From the whole perspective, our algorithm outperforms the Greedy in this co-authorship network.

5 Conclusion

In this paper, we put forward a general framework, BGDS, to solve a class of important problems – submodular maximization. And we took the influence maximization as a case to implement it with high efficiency. We did some experiments to explore the effect of the algorithm in practice.

It is possible to tune some parameters finely, such as the size of samples of the graph g, the threshold parameter α and study the effect of them. It is also possible to use the local search method to improve the quality of the solution given by the BGDS. Moreover, it's possible to apply the algorithm framework to other influence model or even other submodular maximization problems.

Another idea is to change the backward framework and include some forward information, just like what is done in [11]. It seems challenging but interesting, just like the topic studied in this paper for me when I started the graduating project in the very beginning.

References

[1] LIU Y, CHONG E, PEZESHKI A, et al. Submodular optimization problems and greedy strategies: A survey[J]. Discrete Event Dynamic Systems, 2020, 30(3): 381-412.

[2] GRöTSCHEL M, LOVáSZ L, SCHRIJVER A. The ellipsoid method and its consequences in combinatorial optimization[J]. Combinatorica, 1984, 4(4): 291-295.

[3] BADANIDIYURU A, MIRZASOLEIMAN B, KARBASI A, et al. Streaming submodular maximization: Massive data summarization on the fly[C]//Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014: 671-680.

[4] BANERJEE S, JENAMANI M, PRATIHAR D K. A survey on influence maximization in a social network[J]. Knowledge and Information Systems, 2020, 62(44–46).

[5] KEMPE D. Maximizing the spread of influence through a social network[J]. Proc.of Acm Sigkdd Intl Conf.on Knowledge Discovery Data Mining, 2003.

[6] NEMHAUSER G L, WOLSEY L A, FISHER M L. An analysis of approximations for maximizing submodular set functions—i[J]. Mathematical Programming, 1978, 14(1): 265-294.

[7] FEIGE U. A threshold of ln n for approximating set cover[M]. ACM, 1996.

[8] CHEN W, WANG Y, YANG S. Efficient influence maximization in social networks[C/OL]//KDD '09: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York, NY, USA: Association for Computing Machinery, 2009: 199–208. https://doi.org/10.1145/1557019.1557047.

[9] CORDASCO G, GARGANO L, MECCHIA M, et al. A fast and effective heuristic for discovering small target sets in social networks[C]//LU Z, KIM D, WU W, et al. Combinatorial Optimization and Applications. Cham: Springer International Publishing, 2015: 193-208.

[10] CAI S. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs[J]. AAAI Press, 2015.

[11] BALKANSKI E, SINGER Y. The adaptive complexity of maximizing a submodular function [C/OL]//STOC 2018: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing. New York, NY, USA: Association for Computing Machinery, 2018: 1138–1151. https://doi.org/10.1145/3188745.3188752.

[12] SCHRIJVER A. Combinatorial optimization - polyhedra and efficiency[M]. Springer, 2003.

[13] HOREL T, SINGER Y. Maximization of approximately submodular functions[C/OL]//LEE D, SUGIYAMA M, LUXBURG U, et al. Advances in Neural Information Processing Systems: volume 29. Curran Associates, Inc., 2016. https://proceedings.neurips.cc/paper/2016/file/81c8727c62e800be708dbf37c4695dff-Paper.pdf.