Engineering Scalable Influence Maximization

by

Akshay Khot
B.Eng., University of Mumbai, India, 2013

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Engineering Scalable Influence Maximization

by

Akshay Khot
B.Eng., University of Mumbai, India, 2013

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member
(Department of Computer Science)

ABSTRACT

In recent years, social networks have become an important part of our daily lives. Billions of people daily use Facebook and other prominent social media networks. This makes them an effective medium for advertising and marketing. Finding the most influential users in a social network is an interesting problem in this domain, as promoters can reach large audiences by targeting these few influential users. This is the influence maximization problem, where we want to maximize the influence spread using as few users as possible. As these social networks are huge, scalability and runtime of the algorithm to find the most influential users is of high importance.

We propose innovative improvements in the implementation of the state-of-the-art sketching algorithm for influence analysis on social networks. The primary goal of this thesis is to make the algorithm fast, efficient, and scalable. We devise new data structures to improve the speed of the sketching algorithm. We introduce the compressed version of the algorithm which reduces the space taken in the memory by the data structures without compromising the runtime. By performing extensive experiments on real-world graphs, we prove that our algorithms are able to compute the most influential users within a reasonable amount of time and space on a consumer grade machine. These modifications can further be enhanced to reflect the constantly updating social media graphs to provide accurate estimations in real-time.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My supervisor, Dr. Alex Thomo, for his kind and patient supervision, and attention to detail. I am deeply grateful for his support and mentoring throughout my master's program.

My teammate, Diana Popova, for her ideas, inspiration, and friendship. This thesis would not be accomplished without her efforts.

Dr. Venkatesh Srinivasan, for serving on my thesis supervisory committee.

DEDICATION

This thesis is dedicated to my loving grandmother.
For teaching me the value of education.

Chapter 1

Introduction

In recent years, the popularity of social networks such as Facebook, Instagram, and Twitter has skyrocketed. As of 2017, Facebook has close to 2 billion users, YouTube has more than a billion users, and Twitter has 300 million users [29]. These tools play a fundamental role as a medium for the spread of information, ideas, and influence. Social media has also become a very good tool for companies and organizations to create and maintain a positive brand image and to connect with their followers. The primary reason for the success of social media is that they do the job of connecting people exceptionally well.

Social media is also becoming very popular as a marketing platform. It enables a promoter to advertise a product to a large population in a short period of time. As there are millions of people using these social media websites, the potential reach for any product can be in millions of users. Hence, they have become an attractive medium for marketers for product promotion.

An interesting problem in this domain of social media marketing is to find the most influential users in a social media network, which can effectively influence as many users as possible. By targeting these influential users, promoters can effectively reach a large number of people, increasing sales and profits. This way of product promotion is also very cost-effective compared to traditional media of advertising such as TV and radio. It can result in a product getting popularity by word-of-mouth effect, using as few resources as possible.

As these social media networks are massive, consisting of billions of users, any algorithm processing and analyzing them must be very efficient, both in terms of time and space. The focus of this thesis is to improve the speed of the process of finding most influential users and to do this efficiently using a consumer grade machine.

1.1 Motivation

To sell any consumer product successfully, it needs to be marketed effectively. Companies and promoters are always looking for new and innovative ways to market their products. They want a medium that will reach a larger population and is cost-effective. In the past, it used to be newspapers, radio, and television. Since last decade, blogs and social media have become an immensely useful tool to reach to a wider audience in a cost-effective way. Social media platforms such as Facebook, Instagram, Twitter, LinkedIn, Snapchat, etc. have gained a huge popularity among users. These social media platforms are a part of users' daily lives. Apart from their primary intended use of keeping in touch with loved ones, many use them for networking, entertainment, news and much more.

An interesting result of so many people using these media has been social media influencers. These include people or channels on these networks which have a wide following. For example, the YouTube channel of Michelle Phan has almost 9 million subscribers [33] as of 2017. That means every video uploaded by her is watched by millions of people, over and over. This is an excellent opportunity for marketers to promote their product. For example, a beauty product company wanting to sell their makeup products can give free samples to Michelle Phan and ask her to promote them on her channel, resulting in getting millions of views and subsequent purchases.

Targeting these influential users specifically is in the interest of the promoters. Once these influencers promote a product on their platform, it's instantly reached to their subscribers. As the subscribers can view this content repeatedly, there is no expiry date or additional cost on this type of advertising.

1.1.1 Direct Marketing vs. Targeted Marketing

Social media promotion is efficient and reduces wasted resources. When a company promotes its products on traditional media such as TV or radio, they don't have any control over the consumers of those media, hence they cannot be specific. It usually results in the market mismatch, as not everyone who watches TV or listens to radio is interested in the same category of products. It's much better to target a niche market with a high response rate than a larger market with low response. Social media platforms help to target the right audiences for a product. For example, when marketers promote a product via a YouTube channel, they have a fair idea of the consumers of that channel, so they can target them effectively. Direct Marketing using social media can be very effective in contrast to mass marketing, where a product is promoted indiscriminately to all potential customers. In direct marketing, marketers first attempt to select the customers which are likely to be profitable, and market only to those. If done successfully, this approach can be very cost-effective and significantly increase profits.

Using social media for product promotion can also lead to real relationship building. In contrast to radio and television, social media allows companies or promoters to interact with their customers, to start and cultivate relationships. It also helps them to get insights into their daily lives and adjust and improve the marketing strategy. Also, a strong social media presence builds brand loyalty, which is very important for the long term commitment of customers.

One limitation of direct marketing is the treatment of all future customers as equal and independent entities. In reality, a person's decision to buy a product is often strongly influenced by her friends, celebrities or the people she follows on various social media networks. Marketing based on such recommendations can be much more cost-effective in contrast to directly targeting to all potential customers. It leverages the selected few customers(influencers) to carry out the further work of promoting the product. This type of marketing is called viral marketing because of its similarity to the spread of a viral disease from one person to another in a relatively short period of time. As social media networks provide easy access and communication among users, this form of marketing can be effectively utilized to promote the products. The

premise of viral marketing is that by initially targeting a few "influential" members of the network - for example, giving them free samples of the product - a product is recursively recommended by each individual to his/her friends to create a large cascade of further adoptions.

Figure 1.1 displays a social network represented as a graph [23]. Each node represents a user and an edge in the graph represents the relationship between two users. It's observed that there are certain users which are connected to a large number of other users. These are the influential users in the network. Specifically targeting these influential users, instead of everyone in the network will result in a very good spread with much less cost.

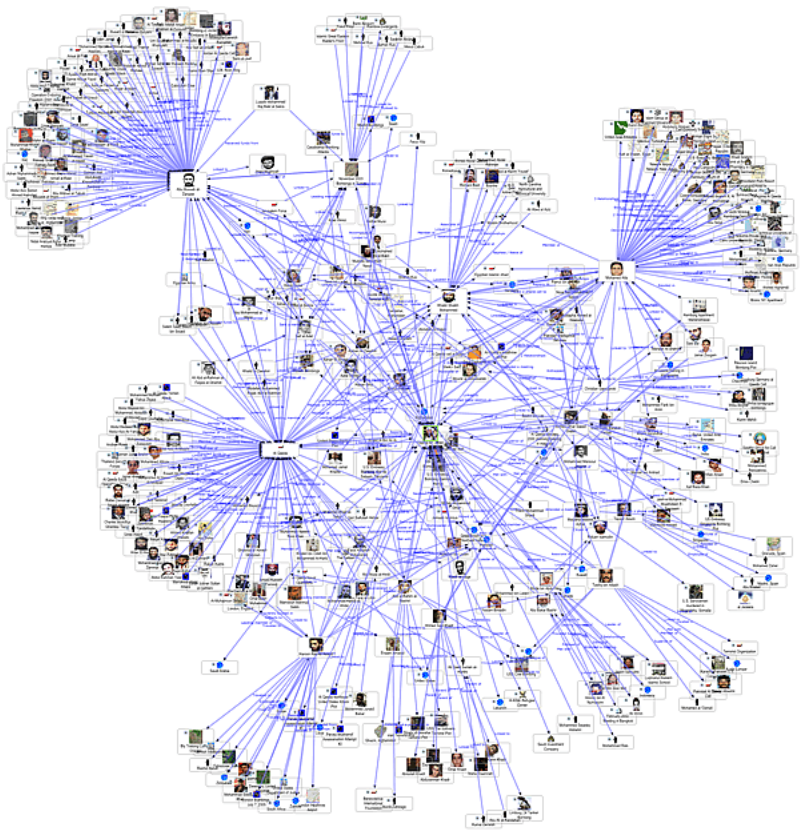


Figure 1.1: Graphical Representation of a Network

1.1.2 Challenges Involved

Modern social media networks are of massive size. There are billions of users on these networks with billions of connections. Apart from the size, these networks are not static. Thousands of people join them every day, introducing new relationships. This makes these networks highly dynamic in nature and continuously evolving over time, where new relationships are getting formed constantly. As a result, the outcomes concerning estimation and maximization of influence can quickly become outdated. Even with state-of-the-art scalable algorithms, to re-run them on these massive networks from scratch becomes expensive.

A marketer cannot just randomly pick users to hand out free samples, due to time and cost constraints. These users need to be picked strategically so that using as few users as possible, the product can reach to as many people as possible. Hence this becomes an optimization problem, i.e. to find a node in this graph which can potentially connect to a larger number of nodes in the graph, and to do this in an efficient way. Given the huge size of these networks and their dynamic nature, there are some challenges that have to be met to solve this optimization problem effectively and efficiently.

Kempe *et al.* [15] first formulated the influence maximization as a combinatorial optimization problem. They devised several probabilistic cascade models and presented a general greedy approach for this problem. They adopted the Monte-Carlo simulation to approximate the influence spread. Though this approach is simple and effective, it is too slow for the large graphs, and it is computationally expensive. This inefficiency has led to different algorithms that aim to reduce the computation overhead.

Very recently, Borgs *et al.* devised a state-of-the-art sketching algorithm to solve this problem efficiently [4]. He utilized the independent cascade model developed by Kempe *et al.* [15]. Although it significantly improves upon previously devised methods in terms of asymptotic performance, its practical efficiency is rather unsatisfactory. In short, no existing influence maximization algorithm can scale to million-node graphs with considerable accuracy.

1.2 Contributions

We build on the state-of-the-art sketching algorithm developed by Borgs *et al.* [4], to improve its efficiency and speed. We introduce new, efficient data structures, called flat arrays which provide significant speed improvements over the naive implementation of the same algorithm, at the same time guaranteeing the accuracy.

The key idea underlying this thesis is to focus on the intermediate data structures that are used in the sketching algorithm [4]. We improve the data structures to optimize the speed of the algorithm. Flat arrays are plain Java arrays which are used to store the data in an efficient manner and use it to compute the influential nodes. Using flat arrays, we can obtain an accurate estimate of the influence spread within milliseconds, and select highly influential vertices faster than the naive implementation. Experimental evaluations using real networks with millions of edges have shown the efficiency, scalability, and accuracy of our improvements in the algorithm. Our experimental results show that the proposed improvements consistently outperform the naive implementation in terms of computation efficiency, and is often orders of magnitude faster. We also develop and test a compressed version of flat arrays to improve the space efficiency on large graphs.

The goal of this thesis is to speed-up the computation of maximum influential nodes in large-scale networks, and we would like to achieve this using only a consumer-grade machine. More precisely, our contributions are as follows.

1. We provide new data structures which require reasonable memory for computing most influential nodes and estimating the influence.

2. We conduct extensive experiments on a selection of real-world network datasets. The results show that our improvements enable the algorithm to compute the most influential nodes within very reasonable time and space on a consumer-grade machine.

3. We describe a compressed version of flat arrays which provides considerable savings in the space taken by intermediate data structures.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a brief literature review of the influence maximization problem. We describe the history of the problem, the early attempts to solve it, different approaches taken to solve it and the limitations and advantages of each approach.

In Chapter 3 we provide a thorough background for the influence maximization problem by describing basic graph theory. We give a formal definition of the problem and describe the state-of-the-art sketching method designed to solve the problem. Then a detailed description of the algorithm is provided.

Chapter 4 explains in detail how the data structures used in the original algorithm are modified to achieve the speed improvements and scalability. In doing so, we compare our implementation to that of other researchers when appropriate. This chapter is broken into several sections, one for each phase of the algorithm, and an additional section about general improvements. We also provide a compressed implementation of the flat arrays and describe in detail the modulo arithmetic done to achieve the compression.

In Chapter 5 we provide the experimental results that show the significant speed improvements on large graphs having millions of edges. We compare them against the naive implementation in C++ and Java to prove the speed enhancements. Then we compare the flat arrays with their compressed version. By doing so, we prove that the space improvements in compressed version do not affect its runtime, which is comparable to uncompressed version.

Finally, in Chapter 6 we offer our conclusions and discuss future improvements.

Chapter 2

Related Work

Recently, with the rapid development of online social networks such as Facebook, Twitter, and Google Plus, a bulk of research has been conducted for studying the influence diffusion in social networks. The problem of mining a network to identify the potential customers to market to was first studied by Domingos and Richardson [1]. They delved into the business value of mining a network to find the influential nodes and also realized the value of wealth of data from which the necessary network information can be mined. Since then, with the popularity of new social networks, this data has been exploded. They proposed a general framework for analyzing this data and to optimize the influence and modeled the social networks as Markov random fields, where each customer's probability of buying is a function of both the intrinsic desirability of the product for the customer and the influence of other customers. They emphasized the *network value* of a customer, i.e. the expected profit from sales to other customers she may influence to buy, the customers influenced by them and so on recursively. It provided a solid foundation to exploit the value of viral marketing using networks.

Domingos and Richardson compare three marketing strategies: mass marketing, traditional direct marketing, and the network-based marketing. In all scenarios, mass marketing resulted in negative profits, and network-based marketing led to higher profits. This proved the importance of network-based marketing over the traditional mass marketing.

Kempe *et al.* [15], posed the problem of choosing influential sets of individuals as a problem in discrete optimization and showed that influence maximization problem is NP-hard. They also provide a formal framework of the Independent Cascade model, which forms the basis of our approach. In this model, a node u succeeds in activating a neighbor v with a certain probability. This model is discussed further below. Using this model, they presented a simple greedy strategy to find the influential nodes in the network, which repeatedly picks the node with the maximum marginal gain and adds it to the seed set, until the budget k is reached.

Greedy and its modifications have been a gold standard for the monotone submodular functions optimization. Influence maximization is a special case of such optimization, and the theoretical guarantee of approximate greedy holds for it. Greedy provides an approximation within a factor of $(1 - \frac{1}{e} - \epsilon)$ for any $\epsilon > 0$, in polynomial time. Greedy starts with an empty seed set S. In each iteration, it adds the node u that maximizes the influence of $S \cup u$. Approximating Greedy is similar to exact Greedy, but in each iteration we select a seed node with marginal gain, that is, within a small error ϵ of the maximum, with high probability: the influence of the Greedy solution with s seeds is at least $(1 - (1 - \frac{1}{s})^s) \geq 63\%$ of the best possible for any seed set of the same size.

The simple greedy strategy has a few drawbacks.

1. There is no efficient way to compute the influence spread, and this was later proven to be #P-hard. Kempe *et al.* resorted to using the Monte-Carlo simulation to approximate the influence spread, which although is effective, but very slow for large graphs.

2. The Monte-Carlo simulations are run many times to obtain an accurate estimate of spread, which can be very expensive.

3. It makes $O(nk)$ calls to the spread estimation procedure(Monte Carlo), where n is the number of nodes in the graph and k is the size of the seed set to be picked.

Several performance improvements to GREEDY have thus been proposed. Leskovec *et al.* [20] proposed CELF, which is the lazy evaluation of the influence, which is per-

formed only when a node is a candidate for the highest marginal contribution. Chen *et al.* [8] took a different approach, using the reachability sketches of Cohen [9] to speed up the reevaluation of the marginal contribution of all nodes. Though it is more effective than naive GREEDY, the best current implementations of GREEDY do not scale to networks beyond 10^6 edges [7], which are quite small by modern social networks, which range in billions of edges. Hence, this algorithm is not scalable to very large graphs.

Goyal *et al.* [12] proposed SIMPATH, for influence maximization under the linear threshold model that addresses the drawbacks of the greedy algorithm by incorporating several clever optimizations. SIMPATH builds on the CELF optimization that iteratively selects seeds in a lazy forward manner. However, instead of using expensive MC simulations to estimate the spread, they show that under the Linear Threshold model, the spread can be computed by enumerating the simple paths starting from the seed nodes. As probabilities of paths diminish rapidly as they get longer, the majority of the influence flows within a small neighborhood. Thus the spread can be computed accurately by enumerating paths within a small neighborhood.

Chen *et al.* [5] proved that computing the exact influence of a seed set is #P-hard. They study the spread of influence through a social network. They showed strong inapproximability results for several variants of this problem. Their main result states that the problem of minimizing the size of S, while ensuring that targeting S would influence the whole network into adopting the product is hard to approximate within a polylogarithmic factor. This implies similar results if only a fixed fraction of the network is ensured to adopt the product. Further, the hardness of approximation result continues to hold when all nodes have majority thresholds or have a constant degree and threshold two.

Existing methods of influence maximization can be classified into following three types: *simulation-based*, *heuristic-based* and *sketch-based*. Simulation-based methods repeatedly simulate the diffusion process to estimate the influence spread. Heuristic-based methods avoid the use of Monte-Carlo simulations by restricting the spread of influence into communities. These are more scalable than simulation-based methods but they produce the solutions of poor quality. Sketch-based methods, in particular,

the reverse sketching method introduced by Borgs *et al.* [4], repeatedly build sketches by performing repeated reverse simulations.

To support massive graphs, several studies have proposed algorithms which are specific to the Independent Cascade model. They work directly with the edge probabilities instead of simulations. Hence they cannot be reliably applied to a set of arbitrary instances. Based on Kempe's standard independent cascade model of network diffusion, Borgs *et al.* provided a state-of-the-art sketching algorithm [4] to find a set of k initial seed nodes in the network so that the expected size of the resulting cascade is maximized. This algorithm obtains the near-optimal approximation factor of $(1 - \frac{1}{e} - e)$, for any $e > 0$, in time $O(m + n)$. The algorithm is runtime-optimal and substantially improves upon the previously best-known algorithms which run in time $\Omega(mnk.POLY(\epsilon^{-1}))$. The sketching algorithm also allows early termination.

A notable part of the sketching algorithm is that its runtime is independent of the number of seeds k. This algorithm is randomized, and it succeeds with the probability 0.6. The algorithm is explained in detail in the sections below. Sketching algorithm provides theoretical guarantees on the approximation quality and has good asymptotic performance, but large constants.

Recently, Tang *et al.* [31] developed TIM, which engineers the mostly theoretical sketching algorithm to obtain a scalable implementation with guarantees. A major disadvantage of this approach is that it only works for a pre-specified seed set, as opposed to GREEDY, which produces a sequence of nodes.

Although sketching algorithm is efficient and produces near-optimal results [4], it assumes the underlying network graph is static, i.e. the nodes and edges do not change while the algorithm is running. However, in real life social networks, the graph is changing all the time. As new people join a social network, or leave it, and add or remove new friends, there are new nodes and edges forming all the time. Hence the graph is dynamically evolving, all the time.

To solve this fundamental problem, Ohsaka *et al.* [22] provided a dynamic implementation of the Sketching algorithm by redesigning the data structures of the sketching algorithm. They guarantee the non-degeneracy of the solution accuracy. In addition, they introduce a reachability tree-based technique and a skipping method,

that reduces the time consumption required for edge/vertex deletions and vertex additions respectively. They introduce counter-based random number generators, which improve the space efficiency. The dynamic algorithm can reflect a graph modification within a time of several orders of magnitude smaller than that required to reconstruct an index from scratch. It is able to estimate the influence spread of a vertex set accurately within a millisecond, and select highly influential vertices at least ten times faster than the static algorithm.

A parallel work has been done by Zhuang *et al.* [34] for the dynamically updating graphs. Specifically, the network changes over time and the changes can be only observed by periodically probing some nodes for the update of their connections. The primary strategy then is to probe a subset of nodes in a social network so that the actual influence diffusion process in the network can be best uncovered with the probing nodes. The authors propose a novel algorithm to approximate the optimal solution. The algorithm probes a small portion of the network and tries to minimize the possible error between the observed network and the real network.

Chapter 3

Background

A social network plays a fundamental role as a medium for the spread of information, ideas and influence among the users. *Influence Maximization* is the problem of finding the k number of seed users in a social network, which can be targeted so that the number of users influenced by these seeds is maximized. *Influence Estimation* is the problem of estimating the number of influenced users, given a seed set of users. The motivation behind these problems is the cost-effective marketing strategy called viral marketing, where products are promoted by presenting free or discounted items to a selected group of highly influential users. Targeting these influential users can result in the product reaching a large number of users and thus increasing sales.

The influence maximization problem has been extensively studied in the literature. To analyze a social network, it needs to be modeled in a structure which provides a convenient way to represent the users and their relationships. Graphs are suitable for this purpose, as we can model the users and relationships as nodes and edges in the graph, respectively. In section 3.1, we provide a brief background on *graph theory* and the algorithm used to traverse a graph. Then we describe the *Independent Cascade Model*, introduced by Kempe *et al.* [15], which is used to model the process of information diffusion. Then, we define the above problems formally and the state-of-the-art sketching algorithm designed to solve them. Finally, we describe WebGraph [2], a graph framework we used to load and analyze graphs efficiently.

3.1 Graph Theory

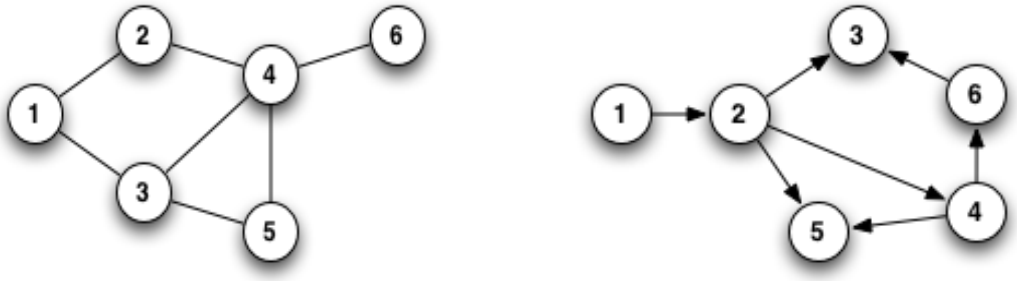
A *graph* is a way of specifying relationships among a collection of items. In a mathematical and computer science domain, graph theory is the study of graphs that concern with the relationship among edges and vertices. Graphs are useful because they serve as mathematical models of network structures.

A graph $G = (V, E)$ consists of a finite set of nodes V, called the vertex set, and a finite set of edges E. Two nodes are said to be neighbors if they are connected by an edge.

Influence maximization problem can be modeled as an optimization problem in the graph theory. We can represent a social media network as a graph, where the users in the network represent the nodes and the relationship between them can be thought of as an edge. Then the influence maximization problem resolves to finding the nodes in the graph which can reach as many nodes as possible.

- Undirected graph: The edges do not have a direction. i.e. for any two nodes u and v, (u, v) and (v, u) is the same edge. It can also be thought of as a bidirectional graph.

- Directed graph: The edges have directions, i.e. (u, v) and (v, u) are 2 different edges. An undirected graph can be considered as a special case of a directed graph as each undirected edge can be represented by two directed edges.



(a) Undirected Graph (b) Directed Graph

Figure 3.1: Directed and Undirected Graphs

Often, social graphs are undirected, as for the Facebook friends graph. But they can be directed graphs, as for example the graphs of followers on Twitter or Instagram.

Graph Traversal

Graph traversal is the process of visiting the nodes in a graph in some systematic order. This has various applications, such as to find the shortest distance between two given nodes or to find the connected nodes. In an undirected graph, we follow all edges from a node, whereas in a directed graph we follow only outgoing edges. In relation to a social network graph, we can traverse a social network to find all the users connected to any given user.

Various efficient algorithms have been developed to traverse the graph. Most commonly used graph traversal algorithms are Breadth-First-Search and Depth-First-Search. In this work, we choose Breadth-First-Search, but Depth-First-Search would work as well, obtaining the same results.

Given a graph G and a starting vertex s, BFS proceeds by exploring edges in G to find all the vertices for which there is a path from s. A good way to visualize what the BFS algorithm does is to imagine that it is building a tree, one level of the tree at a time. It adds all children of the starting vertex before it begins to discover any of the grandchildren. Pseudo code for the BFS algorithm is provided below.

Algorithm 1 Breadth-First-Search Algorithm

Input: G

Output: Vertex Set $P \subseteq V$ of visited nodes

```
1:  procedure BFS(G, root)
2:      Q ← Initialize empty Queue
3:      visited ← []
4:      Q.enqueue(root)
5:      while Q ≠ ∅ do
6:          v ← Q.dequeue()
7:          if v ∉ visited then
8:              visited ← v
9:          Queue ← adjacent(v)
10:     return visited
```

Given a graph with vertex set V and edge set E, BFS takes $O(V + E)$ time. Its worst-case space complexity is $O(b^d)$, where b is the maximum branching factor of the search tree and d is the depth of the tree.

To formulate the influence maximization and influence estimation problem as a graph optimization problem, we need a way to project the process of influencing users in a social media network as activating the nodes in a graph. For this purpose, we adopt the standard information diffusion model, called the *Independent Cascade(IC)* model, which is described below.

3.2 Independent Cascade Model

Developed by Kempe *et al.* [5], this model uses a directed, edge-weighted graph G with n nodes and m edges. The graph represents the underlying network. Influence spreads via a random process, beginning at a set S of nodes. As each node is activated, it can activate its neighbors. Whether it will actually activate its neighbors depends on the activation probability assigned to the edge between these two nodes.

In the IC model, given a graph $G = (V, E, p)$ and a vertex set $S \subset V$, called a

seed set, we first activate the vertices in S. Then, the process evolves in discrete steps according to the following randomized rule. When a vertex u becomes active for the first time at the step t, it has a single chance to activate each current inactive vertex v among its neighbors. It will succeed with probability p_{uv}. If u succeeds, then v will become active in the next step, i.e. step $t+1$. Whether or not u succeeds, it can not make any further attempts to activate v in subsequent steps. This process continues until no further activation is possible.

The weight of the edge $e = (u, v)$ represents the probability that the process spreads along edge e from u to v. Let $I(S)$ denote the total number of nodes eventually activated by this process. Then $E[I(S)]$, which is the expected value of $I(S)$ can be considered as the influence of set S.

The IC model captures the intuition that influence can spread stochastically through a network. IC model has become one of the important models of influence spread, since its introduction.

3.3 Problem Statement

Let $G = (V, E, p)$ be a directed influence graph, where V is a vertex set of size n, E is an edge set of size m, and $p : E \rightarrow [0, 1]$ is a propagation probability function that maps a probability value between a pair of vertices. Let set S be a set of seeds. For a vertex v in G, we use $d^+_G(v)$ and $d^-_G(v)$ to denote the out-degree and the in-degree of v, respectively. For an integer k, we use $[k]$ to denote the set $\{1, 2, ..., k\}$.

This paper adopts the standard information diffusion IC model, which is described above. The influence spread of a seed set S under the IC model, denoted by $\sigma(S)$, is defined as the expected total number of influenced vertices for S. We focus on two problems, namely *influence estimation* and *influence maximization*. Based on these notations, we can define both the problems as shown below:

Problem 1: Influence Estimation. Given a graph $G = (V, E, p)$ and a vertex set $S \subseteq V$, compute the influence spread $\sigma(S)$ of S. This problem tries to estimate the expected number of vertices that would be influenced if the seed set is activated.

Problem 2: Influence Maximization. Given a graph $G = (V, E, p)$ and an

integer k, find a vertex set $S \subseteq V$ of size k that maximizes $\sigma(S)$. Influence maximization is an optimization problem. Given an integer k, the goal is to find out a seed vertex set of size k which comprises of most influential vertices in the graph.

3.4 Sketching Algorithm

Any algorithm working with massive, dynamic social media networks need to have the near-linear runtime. This speed requirement has encouraged a good amount of work to develop fast, heuristic methods of finding influential individuals in social networks. This line of work has mostly been experimental and observational, rather than based on theory. Sketching algorithm was devised by Borgs *et al.* [4]. This algorithm is runtime-optimal up to a logarithmic factor and substantially improves upon the previously best-known algorithms. It uses the standard independent cascade model of influence spread.

3.4.1 Motivation

To describe the sketching algorithm, it's necessary to understand how the problem can be solved in a greedy, brute-force way. The problem is to find the set of nodes with the highest influence. This boils down to finding one node which has the highest influence and repeating this process k times to find the seed set S of size k. One strategy would be to estimate the influence of every node and then take top k nodes which have the highest influence. But this brute-force approach is computationally expensive.

The basic idea behind the sketching algorithm is *polling*, where a node v is selected uniformly at random. Then, the set of nodes that would influence v is determined. Intuitively, if this process is repeated multiple times, and a certain node u appears often as an influencer, there is a good chance that u is the most influential node. The authors show that the probability a node u appears in a set of influencers is proportional to $E[I(u)]$, where $I(u)$ is the random number of nodes that are eventually infected. This number can be estimated accurately with relatively few repetitions of

the polling process. Notably, the runtime of the sketching algorithm is independent of the number of seeds k.

3.4.2 Algorithm

Sketching algorithm takes a social network graph G as an input, and returns the k number of most influential users. The algorithm proceeds in two steps. First, the random sampling technique is repeatedly applied to generate sketches, which form the intermediate index. Each sketch corresponds to a set of individuals that was influenced by a randomly selected node in the transpose graph. This index encodes the influence estimates. For a set of nodes S, the total degree of S in the index is approximately proportional to the influence of S in the original graph.

In the second step, a standard greedy algorithm is run on this index to return a set of size k of approximately maximal total degree. Algorithm 2 provides the pseudo-code for the influence maximization algorithm. It details the procedures to construct the index(I) and compute k most influential nodes using that index.

Algorithm 2 Influence Maximization

Input: Precision parameter $\epsilon \in (0, 1)$, directed edge-weighted graph G

Output: Vertex Set $S \subseteq V$ of size k

$W \leftarrow 144(m+n)\epsilon^{-3}log(n)$
$I \leftarrow BuildIndex(W)$
return $BuildSeedSet(I, k)$
procedure $BuildIndex(W)$
    Initialize $I = (V, \emptyset)$
    while $weight(I) < W$ **do**
        $v \leftarrow$ random node of G
        $Z \leftarrow$ influence spread, starting from v in G^T.
        Add Z to index I.
    return I
procedure $BuildSeedSet(I, k)$
    for $i = 1, ..., k$ **do**
        $v_i \leftarrow argmax_v\{deg_I(v)\}$
        Remove v_i and all incident edges from I
        $S \leftarrow S \cup \{v_i\}$
    return S

3.4.3 Advantages

- Sketching algorithm can be modified to allow early termination, providing a trade-off between runtime and approximation quality.
- The runtime is independent of the number of seeds k. It is also close to optimal, as it has a lower bound of $\Omega(m + n)$ on the time required, where m and n are the number of nodes and edges in the graphs, respectively.

3.4.4 Example

Consider following graph G with $n = 5$ nodes and $m = 7$ edges. It represents a social network with 5 users. For simplicity, the edges are considered bidirectional.

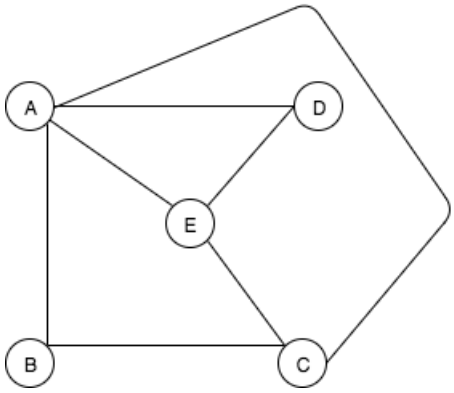


Figure 3.2: Graph G

The sketching algorithm repeatedly picks up random nodes and keeps track of which nodes influenced each node. It repeats itself until the weight of the index does not exceed a pre-determined weight. Steps involved in the sketching algorithm are as follows.

1. It picks up a random node from the graph, e.g. A and performs a reverse BFS(BFS in the graph with the edges reversed, i.e. *transpose graph*) with A as root to find out which nodes could influence A.

2. All the nodes which influence A are stored in a collection called *marked*.

3. It updates the weight of the graph depending on the number of nodes influenced. Then it repeats the above process until the weight exceeds the pre-determined weight.

The results of this process are shown in the Table 3.1.

v	marked
C	A, B, C, E
B	A, B, C
A	A, B, C, D, E
D	A, D, E
E	A, C, D, E

Table 3.1: Influenced nodes after performing reverse BFS

Each record in the Table 3.1 is called a *sketch*. After this process is finished, the algorithm finds out which nodes participated in the most number of sketches. For this, it iterates through *marked* collection for each node and counts in how many sketches that node participated in. For example, node A participates in 5 sketches above. Table 3.2 shows the nodes and the number of sketches they participated in.

v	Number of sketches
A	5
B	3
C	4
D	3
E	4

Table 3.2: Number of sketches for each node

The node that participated in the maximum number of sketches is the most influential node. In this example, node A is the most influential node.

In the next section, we briefly describe a graph compression framework, WebGraph, which is used for decreasing space complexity and provide easy read/write access to the underlying graph.

3.5 WebGraph

WebGraph is a framework for graph compression aimed at studying web graphs, developed by Boldi and Vigna [2, 1]. It provides simple ways to manage very large graphs, exploiting modern compression techniques. According to the website [3], it consists of:

- A set of flat codes suitable for storing web graphs.
- Algorithms for compressing web graphs that exploit gap compression. The algorithms are controlled by several parameters, which provide different trade-offs between access speed and compression ratio.
- Algorithms for accessing a compressed graph without actually decompressing it, using lazy techniques that delay the decompression until it is actually necessary.
- Algorithms for analyzing very large graphs.
- A complete, documented implementation of the algorithms above in Java.
- Datasets for very large graphs (e.g., a billion of links).

Some of the works that have successfully used WebGraph for scaling various algorithms to big graphs are [6, 16, 27, 28, 32]. In this thesis, we implemented sketching algorithm [4] and our new algorithms using the WebGraph API for random access. We use a special instance of WebGraph, called *ImmutableGraph*. It is a simple class representing an immutable graph. This graph is computed once, then stored and accessed repeatedly. To load a graph, we use the function *ImmutableGraph.load()*. The framework provides convenient methods to access the nodes and perform various graph-related operations on it.

Chapter 4

Improvements in Sketching Algorithm

In the preceding chapter, we introduced the sketching algorithm developed by Borgs *et al.* [4], which is the state-of-the-art algorithm for *influence estimation* and *influence maximization*. In this chapter, we give details of the naive implementation of sketching algorithm. We show its limitations and provide a detailed analysis of the improvements using flat arrays. Then we briefly describe the implementation of *Dynamic Influence Maximization*(DIM), which was developed by Ohsaka *et al.* [22]. Finally, we describe a compressed version of flat arrays and compare both implementations.

The basic idea behind our improvements using flat arrays is to make the intermediate data structure, i.e. Index(I) more efficient. The Naive implementation of sketching algorithm used a 2-dimensional list of lists to build the index. In programming, a list is a collection of heterogeneous objects. It provides convenient methods and functions to read/write/update the elements. A List is not fixed in size and can grow and shrink as needed. Though this makes a list easier to use than arrays, there's also a computation cost involved, making it slow for large computations over a longer period of time.

We overcome this drawback of the list by replacing it with arrays, which provide continuous storage of elements in memory. Flat arrays provide some advantages over

a list. First, as the elements are stored sequentially in the memory, if we know the location of the element, we can access that element in constant time. When we are building the index, they provide fast access and insertion. Second, when we want to retrieve the k most influential nodes and update the index, flat arrays provide constant-time search and update, making the computation fast and efficient.

Figure 4.1 shows the comparison between lists and arrays for read/write operations [10].

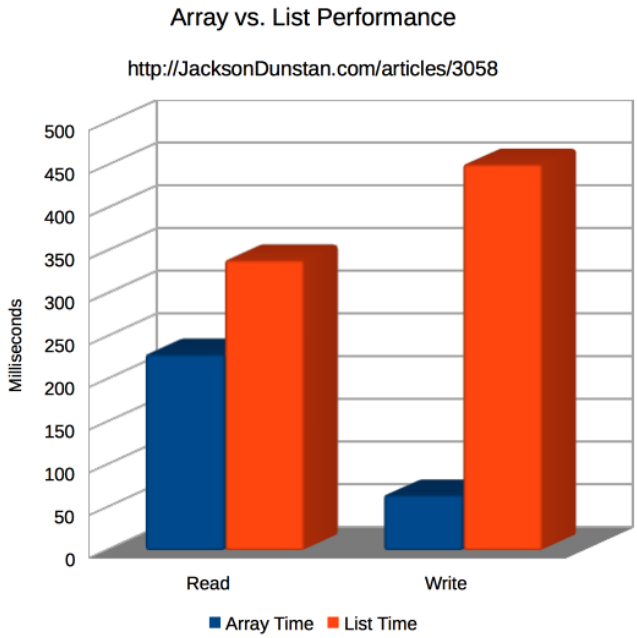


Figure 4.1: Arrays vs. List Performance

As seen above, reading from a list takes 47% longer than reading from an array. For writing, the list takes a 695% longer than array [10]. This difference increases as the size of the graphs and hence the number of computations increase.

In general, a list is fine to use and very convenient for small to medium sized datasets and computations. However, as its internal implementation involves calls to functions and *if/else* statements, it becomes slow as the input size and the number of computations increase.

4.1 Naive Implementation

Sketching algorithm uses an intermediary data structure I, which represents the intermediate index. During its initialization, I is populated with a list of lists of Integers. The algorithm proceeds as follows: We pick a random node v from the graph and perform a reverse Breadth First Search on it to find out which nodes can influence v. For all the nodes affected by v, we pick the list in I corresponding to that node, and add the *sketch_number* that node participated in the list. This process is repeated until the weight of the index reaches a pre-determined weight(W), as described in the sketching algorithm.

In the second step, to compute the k number of influential nodes, the algorithm loops through I, to find out the node that participated in the most number of sketches. Once it finds the node with the largest influence(max_node), it updates the index I, by removing max_node and all the sketches that it participated in. Then it recursively computes the next most influential node. Next, we consider the problem of estimating the influence of S nodes. The number of vertices that a vertex v influences can be approximated by $n.|I_v|.|I|$. The expected value is equal to the influence $\sigma(\{v\})$. Similarly, for a vertex set S, the number of vertices that S influences can be approximated by $n.|I_S|.|I|$.

There are three performance issues in this approach. The first one is the large time taken to compute the k number of most influential nodes in the graph. A list uses internal helper functions to provide convenient read/write access to the elements. These functions include *if-else* statements and internal loops which have a performance cost. The time complexity of computing is $O(n^2)$ which is impractical for big graphs running on a consumer-grade machine.

The second issue is memory usage. For the large number of graphs which have millions of nodes and edges, populating the index can turn quite expensive which may exceed the available memory in a consumer-grade machine. Finally, sketching algorithm is randomized, i.e. we pick a random node from the graph. As the elements in a list are not arranged sequentially in the memory, it has to do a lot of jumping to different memory locations to access the elements.

These three issues effectively make the naive list implementation slow for analyzing and processing big graphs which involve large computations, in a recursive manner. In order to break these bottlenecks, We propose *flat arrays*, which are described in section 4.2.

4.1.1 Algorithm

Algorithm 3 Implementation: Sketching Algorithm

Input: G, k, S

Output: Vertex Set $P \subseteq V$ of size k, $\sigma(S)$

1: $I \leftarrow Build\_Index(G)$
2: $P \leftarrow Maximize\_Influence(I, k)$
3: $\sigma(S) \leftarrow Estimate\_Influence(I, S)$
4: return $P, \sigma(S)$

Algorithm 4 $Build\_Index(G)$

1: $I \leftarrow 2D$ list of Integers
2: $W \leftarrow \beta * (m + n) * log(n)$
3: $weight \leftarrow 0, sketch\_num \leftarrow 0$
4: **while** $weight < W$ **do**
5: $v \leftarrow$ random node of G
6: $marked \leftarrow BFS(v)$
7: $out\_degree \leftarrow 0$
8: **for all** $u \in marked$ **do**
9: $I.get(u).add(sketch\_num)$
10: $out\_degree \leftarrow out\_degree + G.out\_deg(u)$
11: $weight \leftarrow weight + out\_degree$
12: $sketch\_num \leftarrow sketch\_num + 1$
13: return I

Algorithm 5 $Maximize\_Influence(I, k)$

```
1: S ← {}
2: for i ← 1, .., k do
3:     max_node ← node with the highest influence in node_infl
4:     S ← S ∪ {max_node}
5:     nodes_in_sketch ← I.get(max_node)
6:     for all u ∈ V do
7:         I.get(u).removeAll(nodes_in_sketch)
8:     I.get(max_node).clear()
9: return S
```

Algorithm 6 $Estimate\_Influence(I, S)$

```
1: if S = {v} then
2:     return n.|I_v|/|I|
3: else
4:     return n.| ∪_{v∈S} I_v|/|I|
```

4.1.2 Example

Let's consider the following graph.

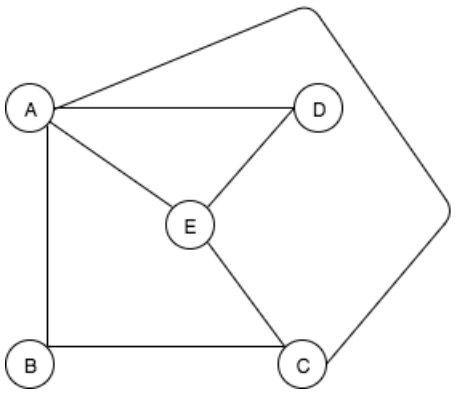


Figure 4.2: A Social Network as a Graph

We repeatedly pick up random nodes from the graph and perform BFS on it to find out which nodes can influence the randomly picked node. For simplicity, the edges are

assumed bidirectional. Let's call the randomly picked node v. The influencer nodes along with the influenced node are stored in the *marked* array. We also keep track of the *sketch_number*, which is just the iteration counter keeping track of random node picks. At the end of this process, we get following:

sketch_number	**v**	**marked**
0	B	A, B, C
1	D	A, D, E
2	A	A, B, C, D, E
3	C	A, B, C, E
4	E	A, C, D, E

Table 4.1: Intermediate Index

At the same time marker array is filled with the influencer nodes, the algorithm fills up the data structure I as follows. For each node in the marked array, it goes to the corresponding node in I, which is a list and adds the current sketch_number to that list. So, I looks like this at the end of the process.

$$\left\{\begin{array}{cccccc} \text{A:} & 0 & 1 & 2 & 3 & 4 \\ \text{B:} & 0 & 2 & 3 & - & - \\ \text{C:} & 0 & 2 & 3 & 4 & - \\ \text{D:} & 1 & 2 & 4 & - & - \\ \text{E:} & 1 & 2 & 3 & 4 & - \end{array}\right\}$$

In the end, calculating the most influential node is just the matter of computing the node in I with the most number of sketches. The more sketches the node participated in, the higher number of nodes it can influence. And this process can be repeated k times to get k most influential nodes.

4.2 Flat Arrays

In this section, we describe the implementation improvements using new data structures. We make use of three arrays, namely *sketches*, *nodes*, and *node_infl*. The idea behind using flat arrays is to provide fast, random access to any index in it. We use a very large number to set the initial size of these arrays. Although all arrays don't need this much storage, we trim them later to make efficient use of the memory. Initially, *sketches* and *nodes* are populated with -1. This indicates an invalid state later in the computation.

sketches array is used to store the sketch number each node participates in. After we perform reverse BFS on a randomly picked node, all the nodes that it influences, including the node itself are part of this sketch. We store that sketch number in *sketches* array.

nodes array is used to store all the nodes that can be influenced in the reverse-BFS process, including the influencer node itself. This array helps to compute and keep track of the influence of each node later in the computation.

node_infl array is used to store the influence of the randomly picked node. For every sketch it participates in, we increment the counter for that node by 1.

We start with picking a random node(v) from the graph. Then we perform BFS on v, which gives us all the nodes which can be influenced by v. Then we loop over these nodes to fill our arrays as follows:

- ***sketches***: For each node that can be reached, fill out *sketches* array by the sketch number in which v participated in.
- ***nodes***: For each node u that can be reached, fill the *nodes* array with u.
- ***node_infl***: For each node u that can be reached, increment the location of u in *node_infl* by 1. As we perform BFS on a transpose graph, the more a node can be influenced, the higher its influence.

After all the nodes which can be influenced by the randomly picked node are processed, we increment the sketch number and repeat the above process by randomly

picking another node. The above process is repeated until the weight of the index does not exceed W.

After the graph has been processed, we cut off the empty tails of the arrays to make them shorter and more efficient.

Seed Computation

The above process describes how the intermediate index is populated by using flat arrays. Once the index is computed, we can compute the most influential nodes using the index.

Finding the most influential node is straightforward, as we have the influence of each node in *node_infl* array. We only have to loop through the array to find the node with the maximum influence. Once we have the most influential node, according to the original sketching algorithm [4], we need to remove this node and all the sketches this node participated in from the index. For large graphs, this step can be very time consuming, if we use naive implementation of a list. This is where Flat Arrays approach really shines. As we have direct access to each element in $O(1)$ constant time, we can manipulate our index much faster. The process of updating our index is described below.

Once we have found the most influential node, i.e. *infl_max*, we scan *nodes* array to find the index(j) of the *infl_max* node. The number in the sketches array at the j'th location is the redundant sketch number. As sketch numbers are added to the *sketches* array in numerical order, the same redundant sketch number can be found before and after the *infl_max* node. Then we need to update all arrays. We begin by removing all the redundant sketch numbers appearing before and after the *infl_max node*, and we replace them with -1, to indicate that it's an invalid sketch number. We also replace the corresponding nodes array locations with -1, to indicate that nodes which can be influenced by *infl_max* node are now redundant. Then, we decrement the influence of that node by 1.

Finally, we update the sketches array at the infl_max node location by replacing it -1. We can not do this before updating the rest, as we need to know the location of the max node. Also, we replace the *max_node*'s index in nodes array by -1. We

also set the influence of the *infl_max* node to 0. Once our index is updated, we repeat this process recursively until we have k most influential nodes.

Advantages

- Arrays are fixed in size. No extra space/memory is allocated for array elements after they are initialized. Hence there is no memory overflow or shortage of memory in arrays.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc. A basic for loop is used to loop over the array elements. It is faster than the internal helper functions provided by list for a convenient access.
- Random(direct) access implies the ability to access any entry in an array in constant time, independent of its position in the array and of array's size. This is a big advantage over the list for the randomized sketching algorithm.
- Arrays provide contiguous memory allocation for its elements, which provides Faster read/write operations.

4.2.1 Algorithm

Algorithm 7 $Build\_Index(G)$

1: $sketches \leftarrow [INT\_MAX]$
2: $nodes \leftarrow [INT\_MAX]$
3: $node\_infl \leftarrow [n]$
4: **for** $i \leftarrow 0, .., INT\_MAX$ **do**
5: $\quad sketches[i] \leftarrow -1$
6: $\quad nodes[i] \leftarrow -1$
7: $W \leftarrow \beta * (n + m) * log(n)$
8: $weight \leftarrow 0, sketch\_num \leftarrow 0, count\_sketches \leftarrow 0$
9: **while** $weight < W$ **do**
10: $\quad v \leftarrow$ random node of G
11: $\quad marked \leftarrow BFS(v)$
12: $\quad$ out_degree $\leftarrow 0$, iteration $\leftarrow 0$
13: $\quad$ **for all** $u \in marked$ **do**
14: $\quad\quad sketches[count\_sketches + iteration] \leftarrow sketch\_num$
15: $\quad\quad nodes[count\_sketches + iteration] \leftarrow u$
16: $\quad\quad node\_infl[u] \leftarrow node\_infl[u] + 1$
17: $\quad\quad iteration \leftarrow iteration + 1$
18: $\quad\quad out\_degree \leftarrow out\_degree + G.out\_deg(u)$
19: $\quad weight \leftarrow weight + out\_degree$
20: $\quad sketch\_num \leftarrow sketch\_num + 1$
21: $\quad count\_sketches \leftarrow count\_sketches + G.size()$
22: return I

Algorithm 8 Maximize_Influence(I, k)

```
1:  procedure Maximize_Influence(I, k)
2:      S ← ∅
3:      for i ← 1, .., k do
4:          v_i ← node with the highest influence in node_infl
5:          S ← S ∪ {v_i}
6:          for j ← 1, .., count_sketches do
7:              if nodes[j] == max_node then
8:                  redundant_sketch ← sketches[j]
9:                  l ← j + 1
10:                 while sketches[l] = redundant_sketch do
11:                     node_infl[nodes[l]] ← node_infl[nodes[l]] − 1;
12:                     sketches[l] ← −1;
13:                     nodes[l] ← −1;
14:                     l ← l + 1;
15:                 ll ← j − 1
16:                 while sketches[ll] = redundant_sketch do
17:                     node_infl[nodes[ll]] ← node_infl[nodes[ll]] − 1;
18:                     sketch_I[ll] ← −1;
19:                     nodes[ll] ← −1;
20:                     ll ← ll − 1;
21:                 sketches[j] ← −1
22:                 nodes[j] ← −1
23:         node_infl[max_node] ← 0
24:     return S
```

Algorithm 9 Implementation: Flat Arrays

Input: G, k

Output: Vertex Set $P \subseteq V$ of size k

1: $I \leftarrow Build\_Index(G)$

2: $P \leftarrow Maximize\_Influence(I, k)$

3: return $\sigma(S), P$

4.2.2 Example

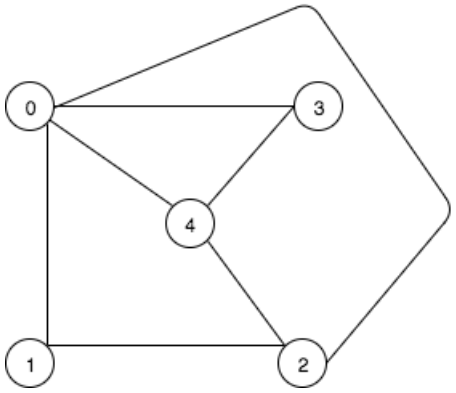


Figure 4.3: Graph

At the beginning of the algorithm, we initialize 3 arrays as follows:

sketches:	-1	-1	-1	...	-1

nodes:	-1	-1	-1	...	-1

node_infl:	0	0	0	0	0

For the first sketch, i.e. *sketch_number* = 0, node number 0 is selected randomly. Then *reverse-BFS* is performed to find out which nodes influence node 0 and we store them in *marked*. For each of these nodes in *marked*, we store the corresponding *sketch_number*(0) in *sketches* array, the influencer node number in *nodes* array, and we increment the influencer node's influence by 1. At the end of the first pass, the arrays are populated as follows:

sketches:	0	0	0	0	0	-1	...	-1

nodes:	0	1	2	3	4	-1	...	-1

node_infl:	1	1	1	1	1

Then we repeat the same process until the weight of the graph exceeds the predefined limit(W). At the end of this process, the flat arrays contain the following data.

sketches:	0	0	0	0	0	1	1	1	2	2	2	2	3	3	3	4	4	4	4

nodes:	0	1	2	3	4	0	1	2	0	1	2	4	0	3	4	0	2	3	4

node_infl:	5	3	4	3	4

Influence Estimation

The next step in the process is to compute k most influential nodes in the graph. Taking top k nodes from the *node_infl* array will not suffice, as we do not want absolute influence, but marginal influence. Hence, after we get an influential node, we need to remove all the sketches that node participated in and hence rearrange our flat arrays.

To compute the k most influential nodes from the flat arrays, we first find the node with the highest influence in the *node_infl* array. Then, we update the *sketches* and *nodes* arrays by removing all the *sketch_numbers* the most influential node participated in. The above procedure is repeated $k-1$ times.

For example, most influential node according to the *node_infl* array is node 0, with the influence of 5. Next, we find the sketches where node 0 participated, and we replace them by -1. We also decrement the node influence of corresponding nodes by 1 to get the accurate node influence. Finally, we set the influential node's influence to 0.

After the first update, the arrays are:

sketches	-1	-1	-1	-1	-1	1	1	1	2	2	2	2	3	3	3	4	4	4	4

nodes:	-1	-1	-1	-1	-1	0	1	2	0	1	2	4	0	3	4	0	2	3	4

node_infl	0	2	3	2	3

We repeat this process $k-1$ times to get the k most influential nodes in the graph.

4.3 Compressed Flat Arrays

In this section, we briefly describe a compression technique we used to improve the memory and space used by flat arrays. This serves two purposes. First, there is the obvious advantage of reducing the space taken by the algorithm. Second and most important, experimental results show that despite the considerable savings in the memory and space and more computations, the processing time does not increase significantly for the computation of influential nodes. This helps prove that the flat array implementation is runtime optimal.

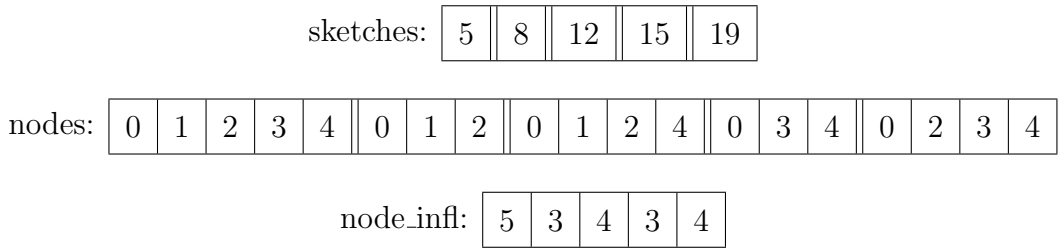
In the uncompressed version, we fill out the *sketches* array as follows. After BFS is performed on a randomly picked node, for each node v that can be influenced, we store the sketch number in *sketches*. In the above example, after the index is built, the arrays are populated as follows:

sketches:	0	0	0	0	0	1	1	1	2	2	2	2	3	3	3	4	4	4	4

nodes:	0	1	2	3	4	0	1	2	0	1	2	4	0	3	4	0	2	3	4

node_infl:	5	3	4	3	4

There is redundancy in the *sketches* array, where the sketch number is repeated n times for each sketch, where n is the number of nodes which can be influenced in each sketch. The main idea behind our compression is to reduce this redundancy and memory consumption. We achieve compression in the flat arrays during the index construction phase, by reducing what information we store in *sketches*. Instead of storing the sketch number for each influenced node, we accumulate the total number of nodes in each sketch and store this value in *sketches*. Hence, after the index is built, the arrays look as follows:



In the second phase of seed computation, we first find the node with the highest influence using *node_infl* array. Then we update the index by recalculating the influence of nodes. For this, we need to know which particular sketch numbers a particular node belongs to. In the uncompressed version, this is straightforward as we just look for the corresponding *sketches* element belonging to that particular node. For example, in the first example, if we want to know which sketch the node 3 belongs to, we look for the element in *sketches* at the same index, i.e. 0. Hence, node 3 belongs to sketch number 0.

In the compressed version, we cannot access the sketch number using the *sketches*, as it does not store the individual sketch numbers. Rather, it stores the accumulated sum of the number of nodes which can be influenced. Hence, to find out the sketch number which corresponds to a node, we first find the lowest element in the *sketches*, which is greater than the index of that node. That element's index in *sketches* is the sketch number that the node belongs to.

For example, we want to find out the sketch number to which node 4 at index 11 belongs. We loop through *sketches* to find the lowest element greater than 11. It is 12, at index 2. Hence node 4 belongs to sketch number 2.

4.3.1 Algorithm

Algorithm 10 $Build\_Index(G)$

1: $sketches \leftarrow [INT\_MAX/50]$
2: $nodes \leftarrow [INT\_MAX]$
3: $node\_infl \leftarrow [n]$
4: **for** $i \leftarrow 0, .., INT\_MAX$ **do**
5: $\quad sketches[i] \leftarrow -1$
6: $\quad nodes[i] \leftarrow -1$
7: $W \leftarrow \beta * (n + m) * log(n)$
8: $weight \leftarrow 0, sketch\_num \leftarrow 0, count\_sketches \leftarrow 0$
9: **while** $weight < W$ **do**
10: $\quad v \leftarrow$ random node of G
11: $\quad marked \leftarrow BFS(v)$
12: $\quad$ out_degree $\leftarrow 0$, iteration $\leftarrow 0$
13: $\quad$ **for all** $u \in marked$ **do**
14: $\quad\quad nodes[count\_sketches + iteration] \leftarrow u$
15: $\quad\quad node\_infl[u] \leftarrow node\_infl[u] + 1$
16: $\quad\quad iteration \leftarrow iteration + 1$
17: $\quad\quad out\_degree \leftarrow out\_degree + G.out\_deg(u)$
18: $\quad sketches[sketch\_num] \leftarrow accumulated\_sketches$
19: $\quad weight \leftarrow weight + out\_degree$
20: $\quad sketch\_num \leftarrow sketch\_num + 1$
21: $\quad count\_sketches \leftarrow count\_sketches + G.size()$
22: return I

Algorithm 11 Maximize_Influence(I, k)

```
1:  procedure Maximize_Influence(I, k)
2:      S ← ∅
3:      for i ← 1, .., k do
4:          v_i ← node with the highest influence in node_infl
5:          S ← S ∪ {v_i}
6:          for j ← 1, .., count_sketches do
7:              if nodes[j] == max_node then
8:                  for sn ← 0, ..., sketch_num do
9:                      if j < sketches[sn] then
10:                         redundant_sketch = sn
11:                 l ← j + 1
12:                 while l < sketches[redundant_sketch] do
13:                     node_infl[nodes[l]] ← node_infl[nodes[l]] − 1;
14:                     sketches[l] ← −1;
15:                     nodes[l] ← −1;
16:                     l ← l + 1;
17:                 ll ← j − 1
18:                 while ll > sketches[redundant_sketch − 1] do
19:                     node_infl[nodes[ll]] ← node_infl[nodes[ll]] − 1;
20:                     sketch_I[ll] ← −1;
21:                     nodes[ll] ← −1;
22:                     ll ← ll − 1;
23:                 nodes[j] ← −1
24:         node_infl[max_node] ← 0
25:     return S
```

Algorithm 12 Implementation: Flat Arrays

Input: G, k

Output: Vertex Set $P \subseteq V$ of size k

1: $I \leftarrow Build\_Index(G)$

2: $P \leftarrow Maximize\_Influence(I, k)$

3: return $\sigma(S), P$

Chapter 5

Experimental Results

The primary goal of this thesis is to improve the runtime and efficiency of the state-of-the-art sketching algorithm by replacing the list with flat arrays. In this chapter, we performed experimental analysis by evaluating our new algorithm provided in the preceding chapter on several real-world graph datasets. We compare four implementations of Borgs Sketching algorithm, namely:

- DIM by Ohsaka *et al.* [22] in C++
- List implementation in Java
- Improved flat arrays in Java
- Compressed version of flat arrays in Java

These implementations are compared in terms of runtime, space efficiency and the resources taken to compute the influential nodes and to estimate the influence of a set of nodes.

Section 5.1 introduces multiple real-world graph datasets we used in tests. In section 5.2, details of equipment and codes implemented are presented. In section 5.3, we compare the three implementations and provide charts that reflect the speed improvements using flat arrays over naive implementations.

5.1 Datasets

We perform the experiments on the following 5 graph datasets:

- DBLP bibliography, scientific collaboration network *(dblp-2010)*
- Dynamically Evolving Large-scale Information Systems Dataset *(uk-2007-05@100000)*
- Italian CNR domain *(cnr-2000)*
- A small crawl of the .in domain performed for the Nagaoka University of Technology *(in-2004)*
- A crawl of the .eu domain *(eu-2005)*

All the graph datasets are available for download at the Laboratory for Web Algorithmics (http://law.di.unimi.it/datasets.php). Properties of these datasets, namely the number of vertices(n) and edges(m) are displayed in Table 5.1.

Dataset	n	m
dblp-2010	326,186	1,615,400
UK100K	862,664	3,050,615
cnr-2000	325,557	3,216,152
in-2004	1,382,908	16,917,053
eu-2005	862,664	19,235,140

Table 5.1: Properties of datasets ordered by m.

Each graph contains three files, namely *.graph, .offsets,* and *.properties*. We test the efficiency and runtime of the above programs across three parameters, namely k, β, and p. These parameters are described below.

- k: Total number of nodes which are the most influential nodes in the graph.
- β: Precision Parameter, which controls the runtime of the sketching algorithm.
- p: A propagation probability representing the magnitude of influence between a pair of vertices, ranging from 0 to 1.

Our programs take these three files and three parameters, k, β, and p as input. Ranges of k, β, and p are given in Table 5.2.

Parameter	**Range**
k	1, 5, 100, 1000
β	2, 4, 8, 16, 32, 64, 128
p	0.05, 0.1

Table 5.2: Ranges of parameters k, β and p.

5.2 Equipment

All of our experiments are conducted on University of Victoria server(CANIS). It uses a 2.10 GHz processor with Intel(R) Xeon(R) E5-2620 v2(12-core) CPU and 128GB RAM.

We implemented all the algorithms in Java, and use the Java version "OpenJDK 1.8.0". In experiments, we allow each Java program to use a maximum of 128 GB heap size.

5.3 Comparisons

In the first part of our experiment, we compare the naive implementations with flat arrays, varying β and having a fixed k. In the second part, we fix β and vary k. Finally, we vary the probability(p) assigned to the edges, keeping β and k fixed. The Borgs sketching algorithm has two parts to it, namely:

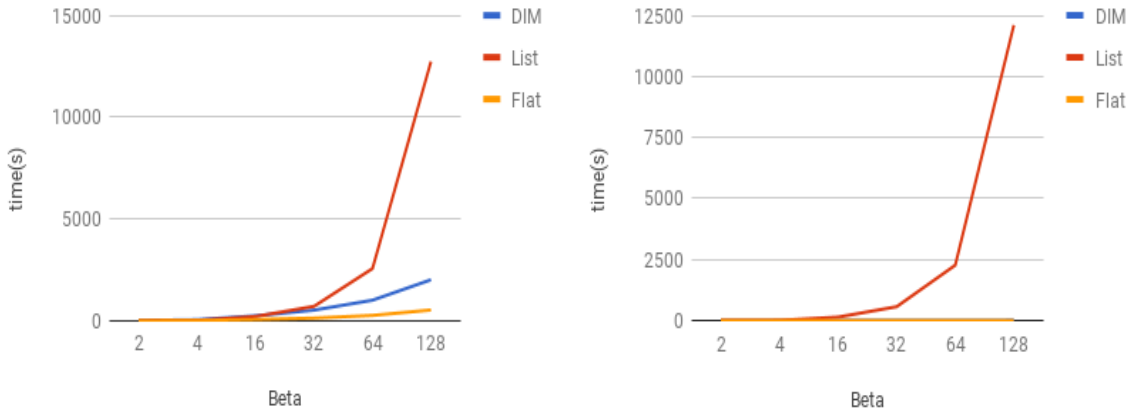
- Building the index by repeatedly adding sketches (index generation)
- Using this index to compute the most influential nodes (seed computation)

We first track the total time taken to compute the most influential nodes, which includes the time taken to build the index and to find the influential nodes. Then we separately tracked the time taken to compute the seeds, and this is where we observe significant speed improvements with flat arrays.

In the figures that follow, figures labeled (a) show the total time taken by each algorithm which includes the time taken to construct the index and compute the seeds, whereas figures labeled (b) only shows the time taken to compute the seeds.

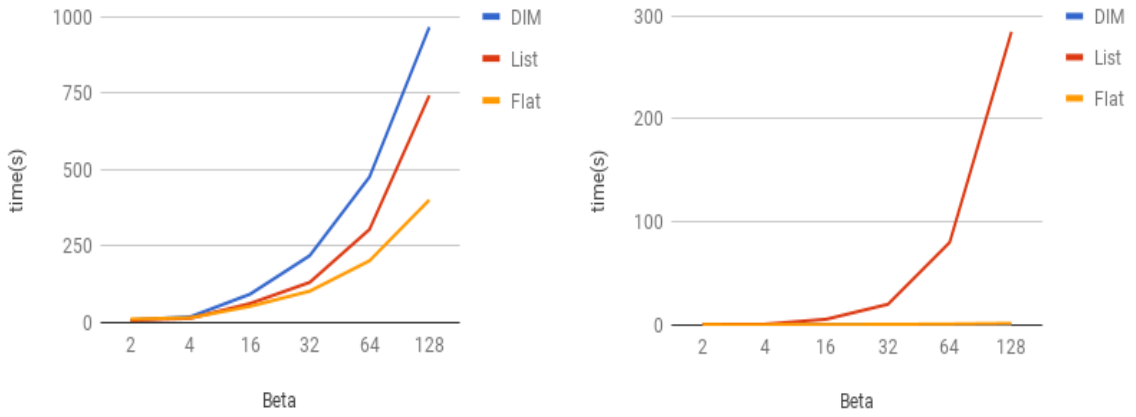
5.3.1 Fixed k, fixed p, vary β

We fixed $k = 5$ and $p = 0.1$, and varied β from 2 to 128, and computed the time taken to create sketches and compute the k most influential nodes. Figure 5.1 to 5.5 show results for computing $k = 5$ most influential nodes with varying β.



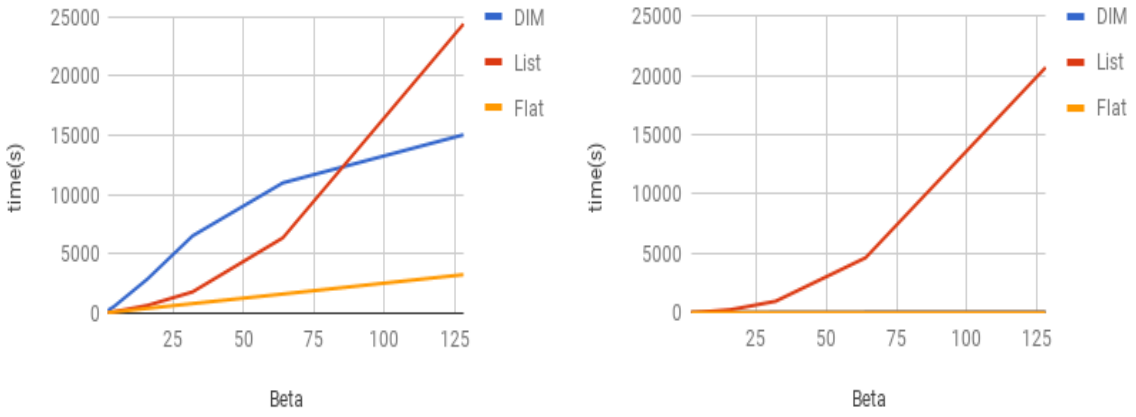
(a) Total Time (b) Time to compute seeds

Figure 5.1: cnr-2000



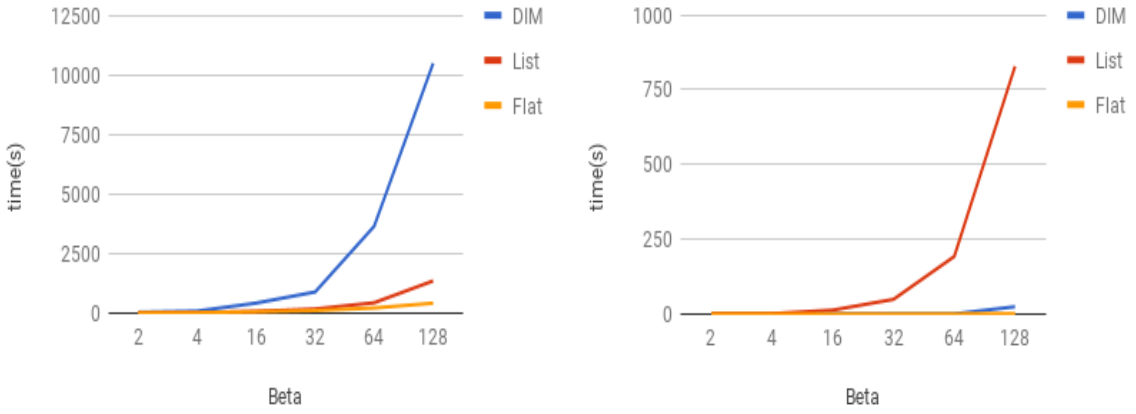
(a) Total Time (b) Time to compute seeds

Figure 5.2: uk-2007-05



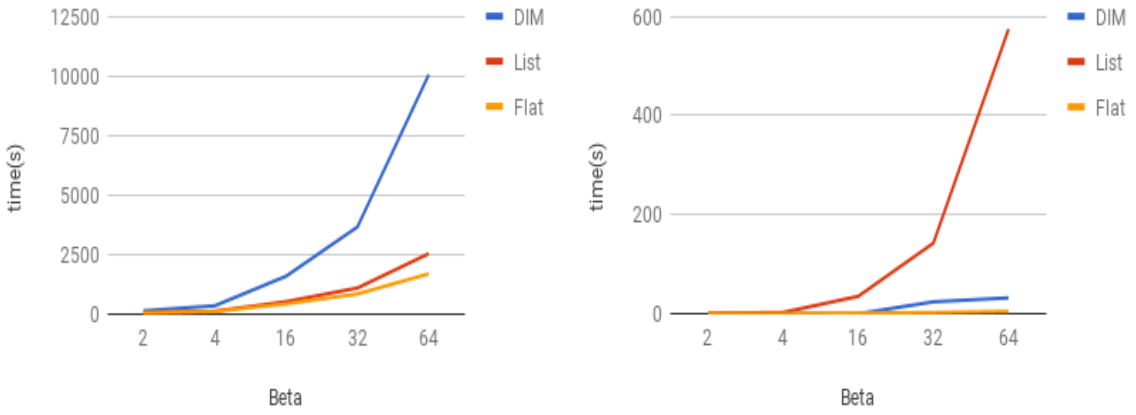
(a) Total Time (b) Time to compute seeds

Figure 5.3: in-2004



(a) Total Time (b) Time to compute seeds

Figure 5.4: dblp-2010



(a) Total Time (b) Time to compute seeds

Figure 5.5: eu-2005

Charts clearly show that flat arrays outperform the naive implementation by orders of magnitude as β increases. Though they start at a relatively same point, as β increases, the time taken to compute the most influential nodes increases exponentially for naive implementation, whereas flat arrays stay nearly constant and show a slow increase.

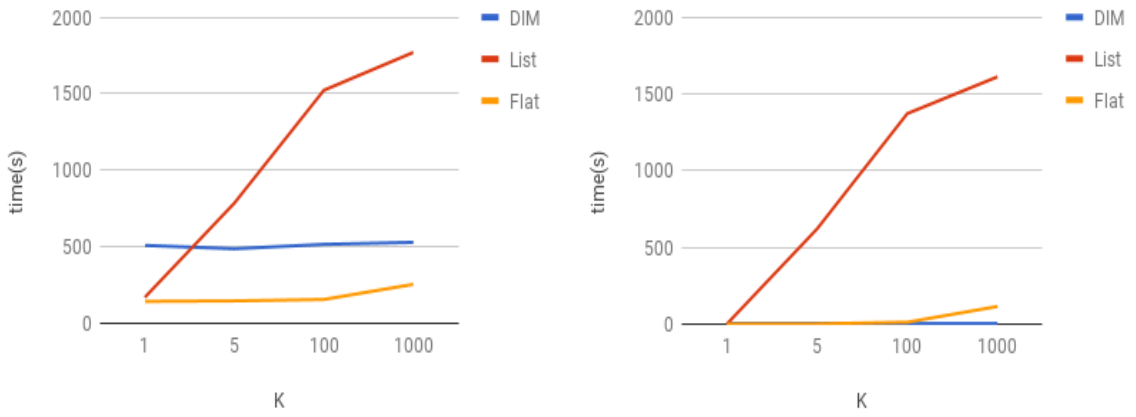
An interesting pattern is observed from the above graphs. It seems the total time taken by C++ naive implementation increases exponentially, as β increases, whereas the time taken to compute the seeds remains relatively constant. In contrast, the naive Java implementation, total time remains relatively low, as β increases, but the time taken to compute the seeds increases exponentially with β. Finally, it's clear from the above charts that increasing β has no significant impact on the total time for flat arrays. Also, the time taken to compute the seeds remain constant throughout.

Conclusion

As seen in the above charts, for flat arrays, the time taken to compute the k most influential nodes remains almost constant, whereas for naive implementation it increases exponentially. Hence it's proved that increasing β has no impact on the seed computation, and in terms of total time, flat arrays outperform both C++ and Java versions of the naive algorithm.

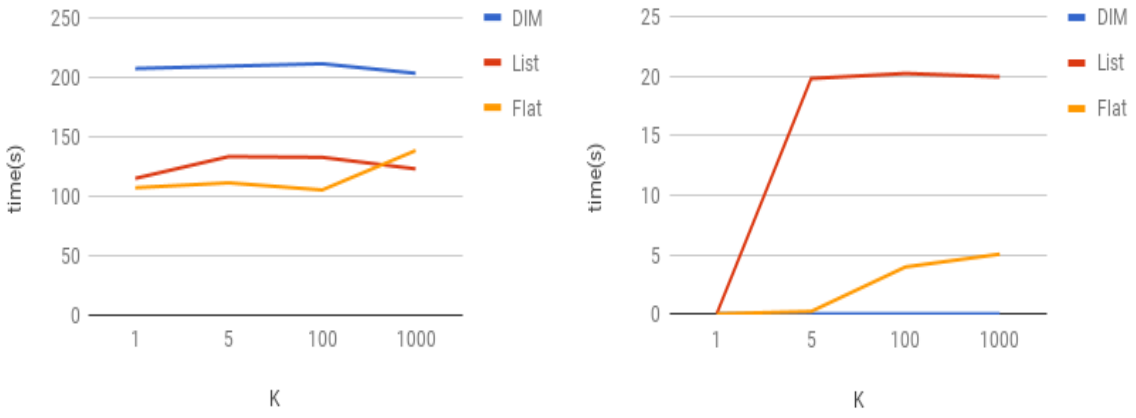
5.3.2 Fixed β, fixed p, vary k

We fixed $\beta = 32$ and $p = 0.1$, and varied k from 1 to 1000, and computed the time taken to create the sketches and compute the k most influential nodes. Figure 5.6 to 5.10 show results for the process of seed computation with $\beta = 32$ to compute k most influential nodes.



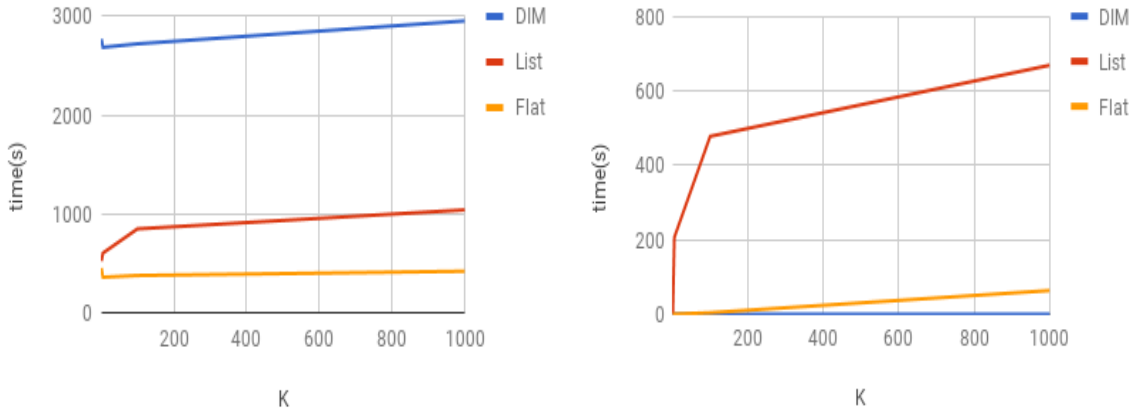
(a) Total Time (b) Time to compute seeds

Figure 5.6: cnr-2000



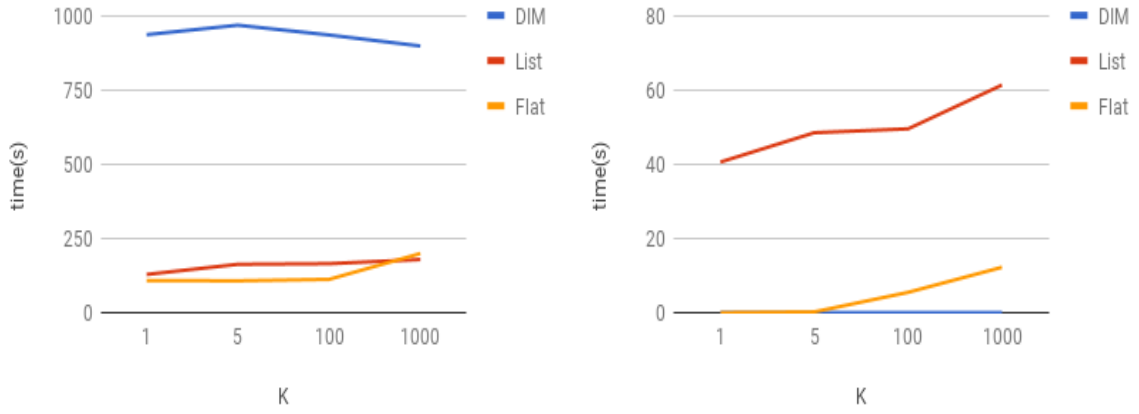
(a) Total Time (b) Time to compute seeds

Figure 5.7: uk-2007-05



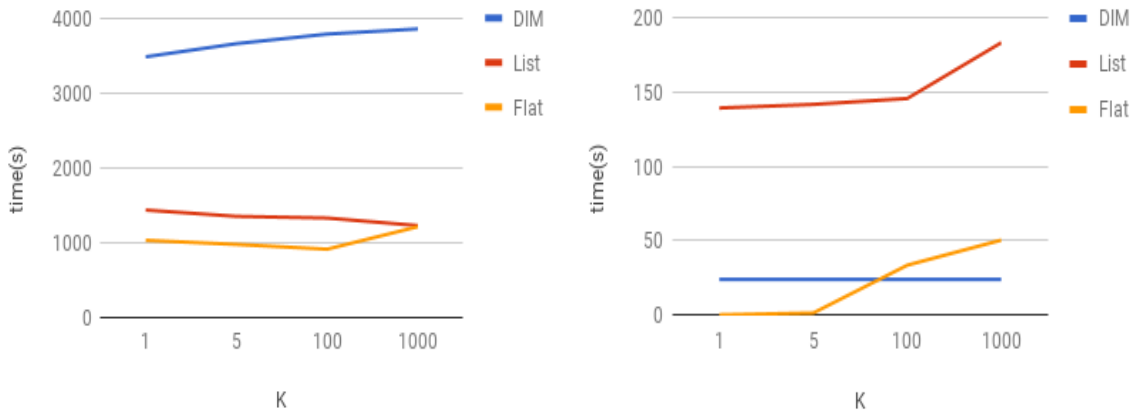
(a) Total Time (b) Time to compute seeds

Figure 5.8: in-2004



(a) Total Time (b) Time to compute seeds

Figure 5.9: dblp-2010



(a) Total Time (b) Time to compute seeds

Figure 5.10: eu-2005

The above charts clearly show that the time taken to generate sketches and compute the k most influential nodes is significantly less when we use flat arrays.

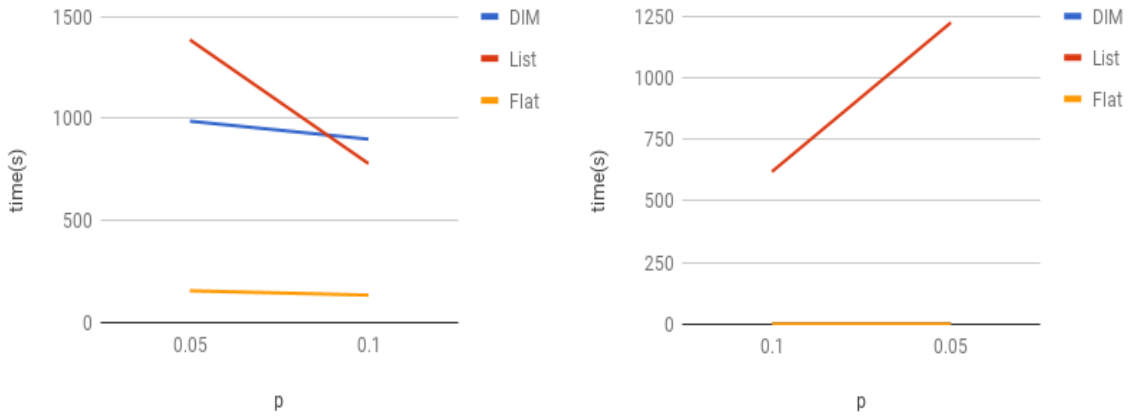
For relatively smaller graphs such as *uk-2007-05* and *dblp-2010*, the Java versions of both naive and flat arrays perform rather similarly, but flat arrays show huge improvements with larger graphs such as *in-2004*. As with vary β, the time taken to compute seeds is minuscule for flat arrays, proving once again that for flat arrays, seed computation is independent of k.

Conclusion

Flat arrays take less time to compute the influential nodes. Also, the runtime is independent of k. This is a big advantage as it means once the index is constructed using flat arrays, we can find out the k most influential nodes in a relatively constant time.

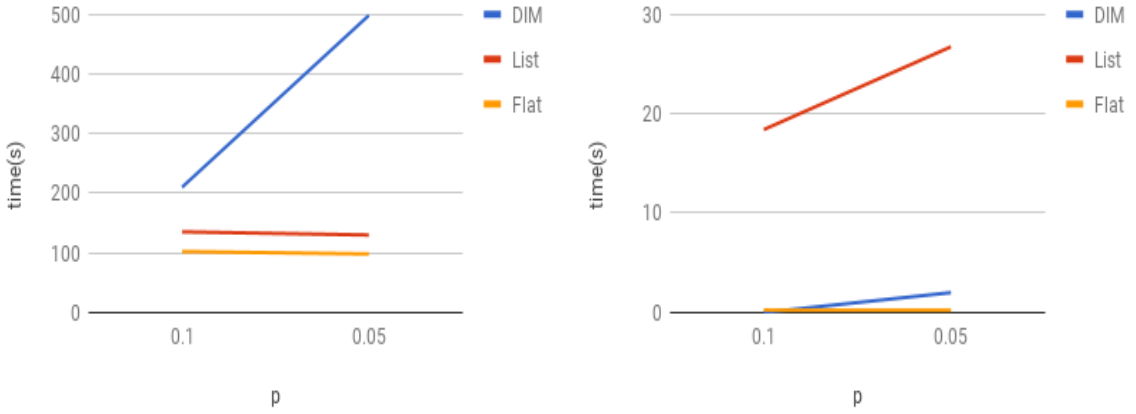
5.3.3 Fixed β, fixed k, vary p

Finally, we fix $\beta = 32$, $k = 5$, and vary p from 0.1 to 0.05 to compute the total time taken to compute the seeds.



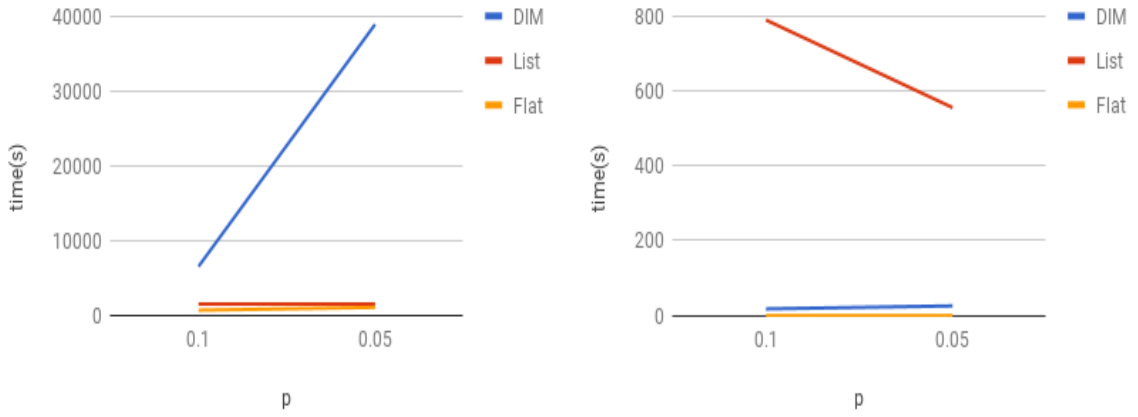
(a) Total Time (b) Time to compute seeds

Figure 5.11: cnr-2000



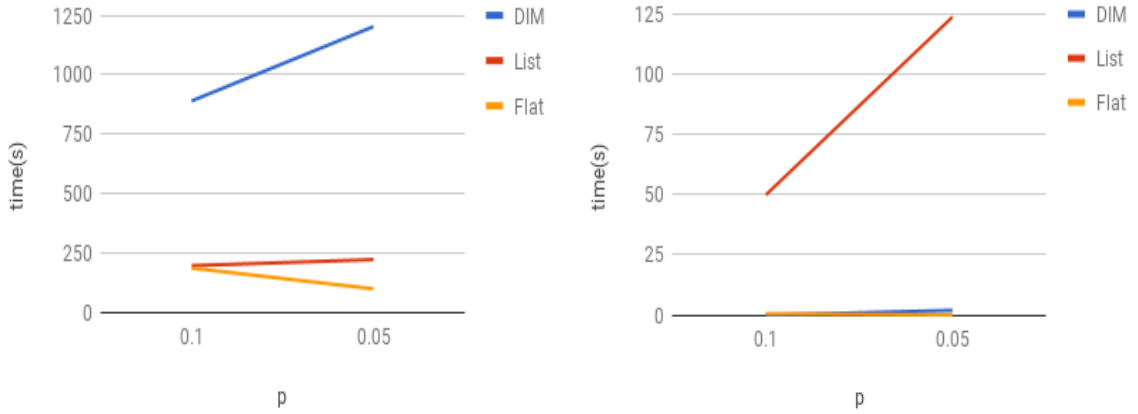
(a) Total Time (b) Time to compute seeds

Figure 5.12: uk-2007-05



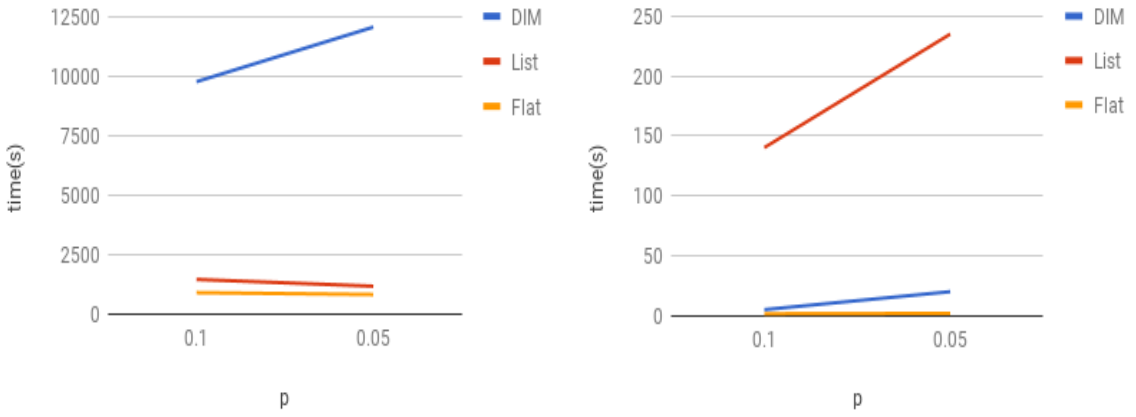
(a) Total Time (b) Time to compute seeds

Figure 5.13: in-2004



(a) Total Time (b) Time to compute seeds

Figure 5.14: dblp-2010



(a) Total Time (b) Time to compute seeds

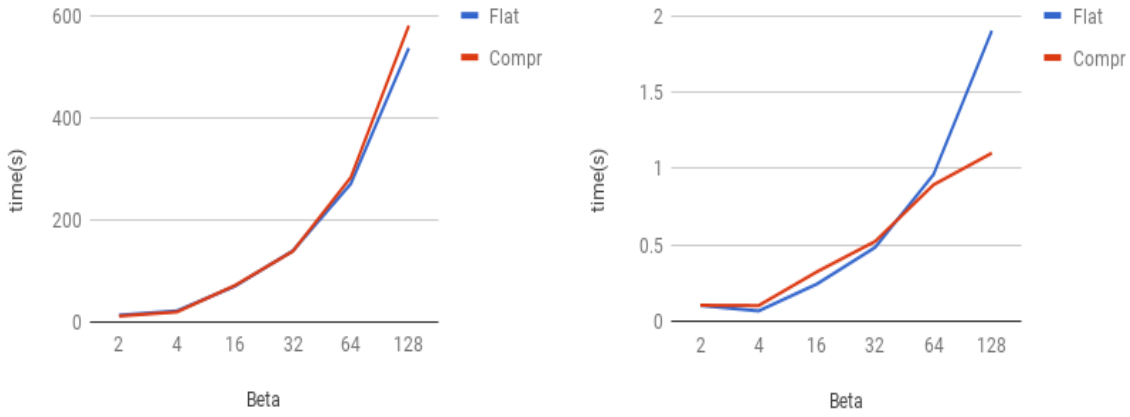
Figure 5.15: eu-2005

According to the Kempe's independent cascade model [15], we call an edge uv active if the activation value $x_i \leq p_u v$, and inactive otherwise. That means, for a high probability, there's more chance that the edges are active. So more edges are included in the index, thereby increasing the computations.

The charts show a very interesting pattern, somewhat similar in the case when we varied β. For the total time, both Java versions of naive and flat array implementations show somewhat similar results, flat arrays being little faster. In case of seed computation, flat arrays and C++ version remain constant, whereas naive Java version increase proportionately, proving seed computation is independent of p for flat arrays.

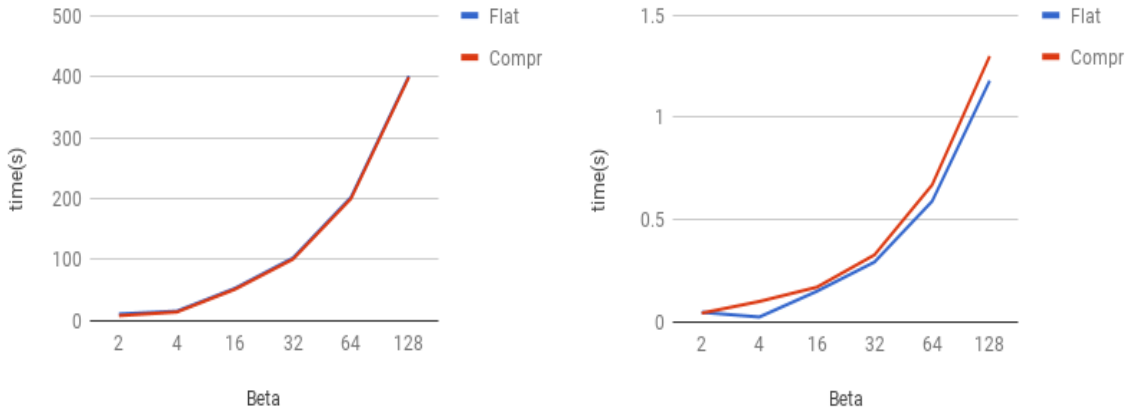
5.3.4 Compressed - Fixed k, fixed p, vary β

In this section, we compare the flat arrays with their compressed version. We fixed $k = 5$ and $p = 0.1$, and varied β from 2 to 128, and computed the time taken to create sketches and compute the k most influential nodes.



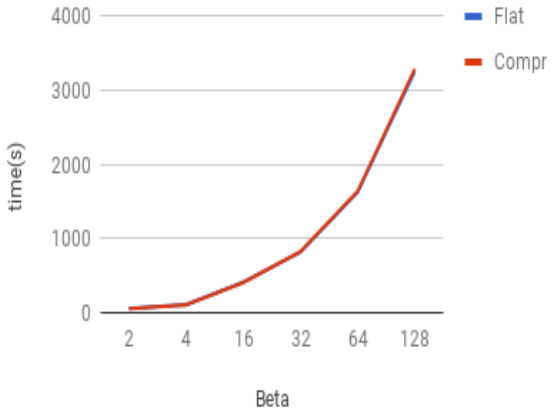
(a) Total Time (b) Time to compute seeds

Figure 5.16: cnr-2000

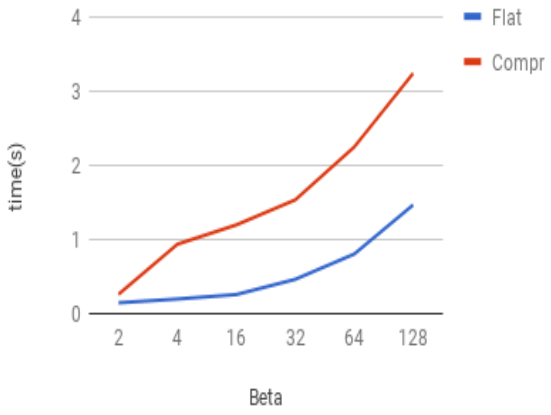


(a) Total Time (b) Time to compute seeds

Figure 5.17: uk-2007-05

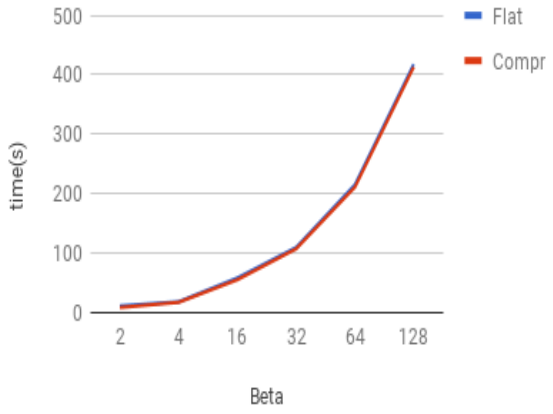


(a) Total Time

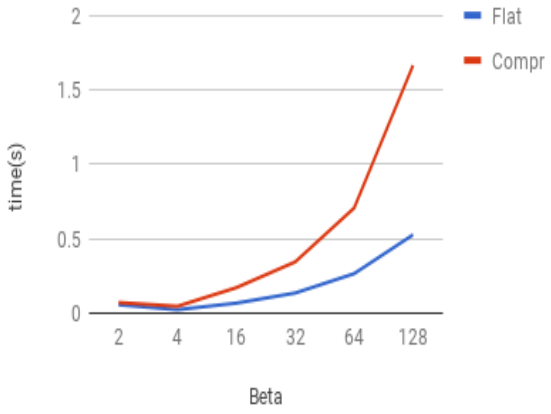


(b) Time to compute seeds

Figure 5.18: in-2004

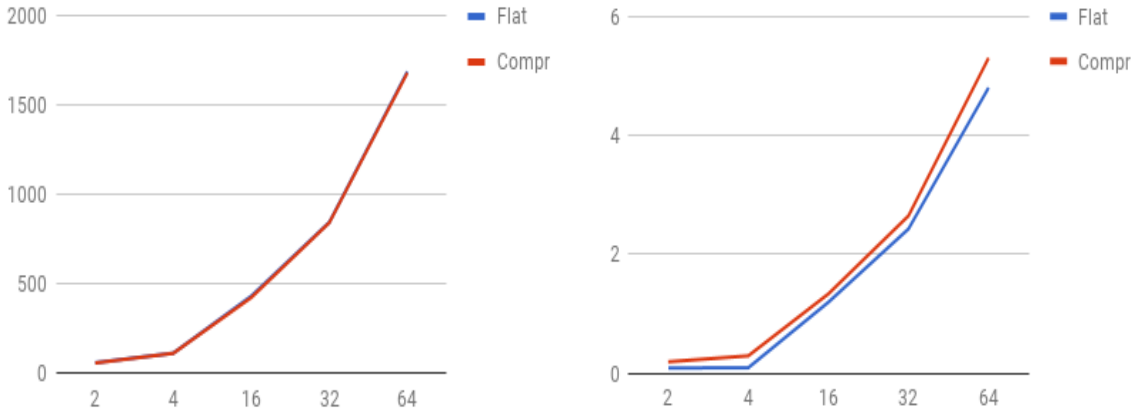


(a) Total Time



(b) Time to compute seeds

Figure 5.19: dblp-2010



(a) Total Time (b) Time to compute seeds

Figure 5.20: eu-2005

In the above charts, we can see that the total time taken by the compressed implementation is almost similar to flat arrays. For a smaller graph such as cnr-2000, we notice a negligible difference in the run-time. If we compare the time taken for computing seeds, for the smaller graphs, the runtime of compressed is comparable to that of flat arrays. For large graphs, compressed version takes a little bit more time than flat arrays, as β increases.

5.3.5 Conclusion

Extensive testing with the graphs shows that flat arrays outperform both naive Java and C++ implementations, both in terms of the index generation and seed computation. Flat arrays not only show significant speed improvements but are much more efficient in terms of using native data structures(Java arrays) and taking relatively less space in memory, thus providing efficient computation.

We also compared the results accuracy, and the results obtained by flat arrays are same as naive implementation. The major benefit of flat arrays shows up in the seed_compute procedure, where we compute k most influential nodes recursively. For flat arrays, this procedure takes constant time, irrespective of β or k, whereas naive implementation takes exponential time to compute seeds.

Finally, comparing the compressed version of flat arrays with the uncompressed version shows that the compression achieves significant reductions in the memory taken to store the arrays. At the same time, it doesn't perform any worse than the uncompressed version. This proves that we can achieve memory efficiency without compromising the runtime of flat arrays.

Chapter 6

Conclusions and Future Work

This thesis introduced an implementation of the state-of-the-art Borgs Sketching algorithm, for computing k most influential nodes in a graph, using a highly efficient graph compression framework WebGraph. Then we introduced an innovative data structure for building and storing index, using built-in arrays in Java programming language. Finally, we described how we can optimize the memory and space requirements for the flat arrays using the compression technique.

Borgs Sketching algorithm tries to solve the following problem: Given a directed graph G, and a positive integer k, how to compute k most influential nodes in G? Because the real-life social media graphs can be huge, the speed of the algorithm is very important.

We focused on improving the speed of the Borgs sketching algorithm. For this, we introduced new data structures for storing the index, called flat arrays. It consists of three flat Java arrays, which store the index information. During the sketch creation process, these arrays are populated, following the Borgs sketching algorithm. Then in the *seed_computation* phase, these arrays are used to compute the k most influential nodes. We then compared our new data structures' implementation with the naive implementation of the Borgs sketching algorithm. We present extensive experiments on a selection of real-world network datasets. The biggest graph we used is *in-2004* which has about a million nodes and 19 million edges. Our results show that the sketching algorithm using flat arrays is able to compute the k most influential

nodes in a significantly shorter period of time, and it consistently outperforms the naive implementation; at the same time producing accurate results. The k most influential nodes computed by naive implementation vs. the flat arrays consistently overlap, thus verifying the accuracy of our data structures. Finally, we compare the uncompressed version of the flat arrays with their compressed version. The results show that compression achieves space efficiency without affecting the runtime of flat arrays.

In our implementation of Borg's Sketching algorithm using Independent Cascade Model, we assigned random probabilities to each edge for spreading the influence. These probabilities are not known in most real world network. Further research can be done to figure our good heuristics and algorithms which try to estimate the influence diffusion probabilities (cf. [25, 11, 19]).

What we have done in this thesis provides the basis for future work in several directions. First, we would like to extend our results to richer models of graphs, such as those where the edges are labeled by the type of the connections between users, e.g. family, colleague, classmate, etc (cf. [13, 26, 30]). Also, recent advancements in parallel computing can be utilized to make an efficient use of the multiple cores and machines to run the algorithm in parallel which can produce dramatic improvements in speed (cf. [14, 21]). Third, we would like to work towards improvements in the dynamically updating graphs, thus mimicking the real-life social media networks. Finally, we would like to explore the connections between influence estimation and trust prediction [17, 18] as well as the processes observed in learning the news in social networks [24].

Appendix A

Additional Information

A.1 List vs. Flat Array Implementation in Java

```
import java.util.Random;

import java.util.ArrayDeque;
import java.util.BitSet;
import java.util.Deque;
import java.util.Random;
import java.util.List;
import java.util.ArrayList;
import it.unimi.dsi.webgraph.ImmutableGraph;

public class InfluenceMax {
    ImmutableGraph G;
    int n;
    long m;
    double W;

    double p;
    int k = 5;
    int beta = 2;

    int[] permutation;
    BitSet marked;


    public InfluenceMax(String basename, double prob) throws Exception {

        G = ImmutableGraph.load("sym-noself/" + basename);
        n = G.numNodes();
        m = G.numArcs();
        W = beta * (n + m) * Math.log(n);
        p = prob;
```

```
        System.out.println(", Beta = " + beta + ", n="+ n + ", m=" + m + ",
            k=" + k + ", p=" + p + "\n");

        marked = new BitSet(n);
        permutation = new int[n];

        for(int i=0; i<n; i++)
            permutation[i] = i;

        Random rnd = new Random();
        for (int i=n; i>1; i--) {
            int j = rnd.nextInt(i);
            int temp = permutation[i-1];
            permutation[i-1] = permutation[j];
            permutation[j] = temp;
        }

    }

    void BFS(int v, BitSet marked) {

        Random random = new Random();

        Deque<Integer> queue = new ArrayDeque<Integer>();

        queue.add(v);
        marked.set(v);

        while (!queue.isEmpty()) {
            int u = queue.remove();
            int[] u_neighbors = G.successorArray(u);
            int u_deg = G.outdegree(u);

            for (int ni = 0; ni < u_deg; ni++) {
                int uu = u_neighbors[ni];
                double xi = random.nextDouble();

                if (!marked.get(uu) && xi < p) {
                    queue.add(uu);
                    marked.set(uu);
                }
            }
        }
    }
}

public class InfluenceMax_flat extends InfluenceMax {

    int nMAX = 10000000;
    int[] sketches;
    int[] nodes;
    int[] node_infl;

    int count_sketches;

   public InfluenceMax_flat(String basename, double p) throws Exception {
     super(basename, p);
```

```
   sketches = new int[nMAX];
     nodes = new int[nMAX];
     node_infl = new int[n];
     for(int i=0;i<nMAX;i++)
     {
         sketches[i] = -1;
         nodes[i] = -1;
     }

     get_sketch();
}

public void get_sketch() {

     long sketchStartTime = System.currentTimeMillis();

    double weight_of_current_index = 0.0;
    int index = 0;
     int sketch_num = 0;

     count_sketches = 0;
     Random gen_rnd = new Random();

    while(weight_of_current_index < W) {
      int v = permutation[gen_rnd.nextInt(n)];
        marked.clear();
        BFS(v,marked);

        int total_out_degree = 0;
         int iteration = 0;
        for (int u = marked.nextSetBit(0); u >= 0; u =
            marked.nextSetBit(u+1))
        {
             sketches[count_sketches + iteration] = sketch_num;
             nodes[count_sketches + iteration] = u;
             node_infl[u] = node_infl[u] + 1;
             iteration = iteration++;
            total_out_degree += G.outdegree(u);
        }
         // add future correction
         //weight_of_current_index += total_out_degree;
        weight_of_current_index += (marked.cardinality() +
            total_out_degree);
        index = ( index + 1 ) % n;
         sketch_num++;
         count_sketches += marked.cardinality();
    }

     System.out.println("Number of Sketches: " + sketch_num);

     // Cutting off the tails of sketches and nodes arrays, making the
         arrays shorter
     int[] iSketch = new int[count_sketches + 1];
     System.arraycopy(sketches,0,iSketch,0,count_sketches);

     int[] iNode = new int[count_sketches + 1];
```

```
        System.arraycopy(nodes,0,iNode,0,count_sketches);

        long sketchEndTime = System.currentTimeMillis() - sketchStartTime;

        System.gc();

        int set_infl = 0;

        long seedStartTime = System.currentTimeMillis();
        get_seeds(iSketch, iNode, node_infl, k, count_sketches, sketch_num,
            set_infl);
        long seedEndTime = System.currentTimeMillis() - seedStartTime;

        System.out.println("Compute_Sketches: " + sketchEndTime/1000.0 + "
            seconds");
        System.out.println("Compute_Seeds: " + seedEndTime/1000.0 + "
            seconds");
    }

     void get_seeds(int[] sketches, int[] nodes, int[] node_infl, int k,
        int count_sketches, int sketch_num, int set_infl) {

        int infl_max = 0;
        int max_node = 0;
        int total_infl = 0;

        for(int v=0;v<n;v++)
        {
            if(node_infl[v] < 1)
                continue;
            else
            {
                int temp = node_infl[v];
                if(temp > infl_max)
                {
                    infl_max = temp;
                    max_node = v;
                }
            }
        }

        infl_max = node_infl[max_node] * n / sketch_num;
        total_infl = set_infl + infl_max;

        System.out.println("Max Node = " + max_node + ", Its Influence = "
            + infl_max);

        if((k - 1)==0) {
            System.out.println("Total Influence of " + this.k + " nodes = "
                + total_infl + "\n");
            return;
        }

        // Re-calculating the influence of the remaining nodes: remove max
            node and the sketches it participated in
        // plus re-calculate the influence
        for(int j=0;j<count_sketches;j++)
```

```
        {
            if(nodes[j] == -1)
                continue;
            else
            {
                if(nodes[j] == max_node)
                {
                    int redundant_sketch = sketches[j];

                // As sketches are added to the array in numerical order,
                    the same redundant sketch can be found before and after
                    the max node
                    int l = j+1;
                    while(sketches[l] == redundant_sketch) {
                        node_infl[nodes[l]] = node_infl[nodes[l]] - 1;
                        sketches[l] = -1;
                        nodes[l] = -1;
                        l++;
                    }
                    if(j>0) // (j!=0) Boundary of the arrays sketches and
                        nodes
                    {
                        int ll = j-1;
                        while(sketches[ll] == redundant_sketch) {
                            node_infl[nodes[ll]] = node_infl[nodes[ll]] - 1;
                            sketches[ll] = -1;
                            nodes[ll] = -1;
                            ll--;
                        }
                    }
                    sketches[j] = -1;
                    nodes[j] = -1;
                }
            }
        }
        node_infl[max_node] = 0;

        get_seeds(sketches, nodes, node_infl, k-1, count_sketches,
            sketch_num, total_infl);
    }


}

public class InfluenceMax_list extends InfluenceMax{

   public InfluenceMax_list(String basename, double p) throws Exception {
        super(basename, p);
        get_sketch();
   }

   public void get_sketch() {

       long sketchStartTime = System.currentTimeMillis();

        List<List<Integer>> I = new ArrayList<>();
```

```
        for(int j=0;j<n;j++)
            I.add(new ArrayList<>());

       double weight_of_current_index = 0.0;
       int index = 0;
        int sketch_num = 0;

        Random gen_rnd = new Random();

       while(weight_of_current_index < W)
       {
         int v = permutation[gen_rnd.nextInt(n)];
           marked.clear();
           BFS(v,marked);

           int total_out_degree = 0;
           for (int u = marked.nextSetBit(0); u >= 0; u =
               marked.nextSetBit(u+1))
           {
                I.get(u).add(sketch_num);
               total_out_degree += G.outdegree(u);
           }
           weight_of_current_index += total_out_degree;
           index = (index+1) % n;
            sketch_num++;
       }

       long sketchEndTime = System.currentTimeMillis() - sketchStartTime;


       System.out.println("Number of Sketches: " + sketch_num);
        System.gc();

        int set_infl = 0;

        long seedStartTime = System.currentTimeMillis();
        get_seeds(I, k, sketch_num, set_infl);
        long seedEndTime = System.currentTimeMillis() - seedStartTime;

        System.out.println("Compute_Sketches: " + sketchEndTime/1000.0 + "
            seconds");
        System.out.println("Compute_Seeds: " + seedEndTime/1000.0 + "
            seconds");
}

 void get_seeds(List<List<Integer>> I, int k, int sketch_num, int
    set_infl) {

        int infl_max = 0;
        int max_node = 0;
        int total_infl = 0;

        for(int v=0;v<n;v++)
        {
            if(I.get(v).size() < 1)
                continue;
            else {
```

```
                int temp = I.get(v).size();
                if(temp > infl_max) {
                    infl_max = temp;
                    max_node = v;
                }
            }
        }

        infl_max = I.get(max_node).size() * n/sketch_num;
        total_infl = set_infl + infl_max;

        System.out.println("Max Node = " + max_node + ", Its Influence = "
            + infl_max);

        if((k - 1)==0) {
            System.out.println("Total Influence of " + this.k + " nodes = "
                + total_infl + "\n");
            return;
        }

        List<Integer> nodes_in_max_node = I.get(max_node);
        for(int u=0;u<n;u++) {
            if((I.get(u).size() < 1) || (u == max_node))
                continue;
            else
                I.get(u).removeAll(nodes_in_max_node);
        }
        I.get(max_node).clear();

        get_seeds(I, k-1, sketch_num, total_infl);
    }


}

public class InfluenceMax_compress extends InfluenceMax {

    int nMAX = 10000000;
    int[] sketches;
    int[] nodes;
    int[] node_infl;

    public InfluenceMax_compress(String basename, int beta) throws
        Exception {
        super(basename, beta);

        sketches = new int[nMAX/50];
        nodes = new int[nMAX];
        node_infl = new int[n];

        for(int i=0; i<n; i++)
            permutation[i] = i;

        for(int i=0;i<nMAX;i++)
        {
            nodes[i] = -1;
```

```
    }
    for(int i=0;i<nMAX/50;i++)
    {
        sketches[i] = -1;
    }

    get_sketch();
}

void get_sketch() {

    long sketchStartTime = System.currentTimeMillis();

    double weight_of_current_index = 0.0;
    int index = 0;
    int sketch_num = 0;

    int accumulated_sketches = 0;
    Random gen_rnd = new Random();

    while(weight_of_current_index < W)
    {
        int v = permutation[gen_rnd.nextInt(n)];
        marked.clear();
        BFS(v,marked);

        int iteration = 0;
        int total_out_degree = 0;

        for (int u = marked.nextSetBit(0); u >= 0; u =
            marked.nextSetBit(u+1))
        {
            node_infl[u] = node_infl[u] + 1;
            nodes[accumulated_sketches + iteration] = u;
            iteration++;
            total_out_degree += G.outdegree(u);
        }
        accumulated_sketches += marked.cardinality();
        sketches[sketch_num] = accumulated_sketches;
        weight_of_current_index += total_out_degree;
        index = ( index + 1 ) % n;
        sketch_num++;
    }

    System.out.println("Number of Sketches: " + sketch_num);

    int[] iSketch = new int[sketch_num + 1];
    System.arraycopy(sketches,0,iSketch,0,sketch_num);

    int[] iNode = new int[accumulated_sketches + 1];
    System.arraycopy(nodes,0,iNode,0, accumulated_sketches);

    long sketchEndTime = System.currentTimeMillis() - sketchStartTime;

    System.gc();

    int set_infl = 0;
```

```
    long seedStartTime = System.currentTimeMillis();
    get_seeds(iSketch, iNode, node_infl, k, accumulated_sketches,
        sketch_num, set_infl);
    long seedEndTime = System.currentTimeMillis() - seedStartTime;

    System.out.println("Compute_Sketches: " + sketchEndTime/1000.0 + "
        seconds");
    System.out.println("Compute_Seeds: " + seedEndTime/1000.0 + "
        seconds");
}

void get_seeds(int[] sketches, int[] nodes, int[] node_infl, int k,
    int accumulated_sketches, int sketch_num, int set_infl) {

    // Calculating the node with max influence
    int infl_max = 0;
    int max_node = 0;
    int redundant_sketch = 0;
    int total_infl = 0;

    for(int v=0;v<n;v++)
    {
        if(node_infl[v] < 1)
            continue;
        else
        {
            int temp = node_infl[v];
            if(temp > infl_max)
            {
                infl_max = temp;
                max_node = v;
            }
        }
    }

    infl_max = node_infl[max_node] * n / sketch_num;
    total_infl = set_infl + infl_max;

    System.out.println("Max Node = " + max_node + ", Its Influence = "
        + infl_max);

    // Stopping condition: no need to re-calculate the influence, if we
        already got the k seeds
    if((k - 1)==0) {
        System.out.println("Total Influence of " + this.k + " nodes = "
            + total_infl + "\n");
        return;
    }

    // Re-calculating the influence of the remaining nodes: remove max
        node and the sketches it participated in
    // plus re-calculate the influence
    for(int j=0;j<accumulated_sketches;j++)
    {
        if(nodes[j] == -1)
            continue;
```

```
            else
            {
                if(nodes[j] == max_node)
                {
                    for(int sn = 0; sn < sketch_num; sn++)
                    {
                        if(j < sketches[sn])
                        {
                            redundant_sketch = sn;
                            break;
                        }
                        else
                            continue;

                    }

                    // As nodes are added to the nodes array in numerical
                        order, the nodes for the same redundant sketch can
                        be found before and after the max node
                    int l = j+1;
                    while(l < sketches[redundant_sketch]) {
                        node_infl[nodes[l]] = node_infl[nodes[l]] - 1;
                        nodes[l] = -1;
                        l++;
                    }

                    if(j>0 && redundant_sketch > 0) // Boundary of the
                        arrays sketches and nodes
                    {
                        int ll = j-1;
                        while(ll >= sketches[redundant_sketch - 1]) {
                            node_infl[nodes[ll]] = node_infl[nodes[ll]] - 1;
                            nodes[ll] = -1;
                            ll--;
                        }
                    }

                    nodes[j] = -1;
                }
            }
        }
        node_infl[max_node] = 0;

        get_seeds(sketches, nodes, node_infl, k-1, accumulated_sketches,
            sketch_num, total_infl);
    }
}

public class Main {

    public static void main(String args[]) throws Exception {

        long startTime1, startTime2, startTime3, estimatedTime1,
            estimatedTime2, estimatedTime3;
        double[] probs = {0.1, 0.01};

        String graphName = "cnr-2000-t";
```

```
        System.out.println("\nG = " + graphName);
        System.out.println("=======================");


        for(int i=0; i<probs.length; i++) {

            System.out.print("List");
            startTime1 = System.currentTimeMillis();
                new InfluenceMax_list(graphName, probs[i]);
            estimatedTime1 = System.currentTimeMillis() - startTime1;
            System.out.println("Total time(List): " + estimatedTime1 /
                1000.0 + " seconds\n");

            System.out.println("----------------------");

            System.out.print("Flat");
            startTime2 = System.currentTimeMillis();
                new InfluenceMax_flat(graphName, probs[i]);
            estimatedTime2 = System.currentTimeMillis() - startTime2;
            System.out.println("Total time(Flat): " + estimatedTime2 /
                1000.0 + " seconds\n");

            System.out.println("-----------------------------------------------------------

            System.out.print("Compressed");
            startTime3 = System.currentTimeMillis();
                new InfluenceMax_compress(graphName, probs[i]);
            estimatedTime3 = System.currentTimeMillis() - startTime3;
            System.out.println("Total time(Flat): " + estimatedTime3 /
                1000.0 + " seconds\n");

            System.out.println("-----------------------------------------------------------
        }


    }

}
```

Bibliography

[1] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[2] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[3] Vigna Boldi. Webgraph.

[4] Christian Borgs, Michael Brautbar, Jennifer T. Chayes, and Brendan Lucier. Maximizing social influence in nearly optimal time. In Chandra Chekuri, editor, *SODA*, pages 946–957. SIAM, 2014.

[5] Ning Chen. On the approximability of influence in social networks. In Shang-Hua Teng, editor, *SODA*, pages 1029–1037. SIAM, 2008.

[6] Shu Chen, Ran Wei, Diana Popova, and Alex Thomo. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1553–1562. ACM, 2016.

[7] Wei Chen, Chi Wang, and Yajun Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In Bharat Rao, Balaji

Krishnapuram, Andrew Tomkins, and Qiang Yang, editors, *KDD*, pages 1029–1038. ACM, 2010.

[8] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *KDD*, pages 199–208, 2009.

[9] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[10] Jackson Dunstan. Array vs. list performance, 2015.

[11] Amit Goyal, Francesco Bonchi, and Laks VS Lakshmanan. Learning influence probabilities in social networks. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 241–250. ACM, 2010.

[12] Amit Goyal, Wei Lu, and Laks V. S. Lakshmanan. Simpath: An efficient algorithm for influence maximization under the linear threshold model. In Diane J. Cook, Jian Pei, Wei Wang, Osmar R. Zaane, and Xindong Wu, editors, *ICDM*, pages 211–220. IEEE Computer Society, 2011.

[13] Gosta Grahne and Alex Thomo. Algebraic rewritings for optimizing regular path queries. *Theoretical Computer Science*, 296(3):453 – 471, 2003.

[14] Xin Hu, Fangming Liu, Venkatesh Srinivasan, and Alex Thomo. k-core decomposition on giraph and graphchi. In *Advances in Intelligent Networking and Collaborative Systems, The 9th International Conference on Intelligent Networking and Collaborative Systems, INCoS-2017*, pages 274–284, 2017.

[15] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 137–146, New York, NY, USA, 2003. ACM.

[16] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[17] Nikolay Korovaiko and Alex Thomo. Predictingtrust from user ratings. In *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT 2012)*, pages 263–271, 2012.

[18] Nikolay Korovaiko and Alex Thomo. Trust prediction from user-item ratings. *Social Network Analysis and Mining*, 3(3):749–759, 2013.

[19] Sylvain Lamprier, Simon Bourigault, and Patrick Gallinari. Influence learning for cascade diffusion models: focus on partial orders of infections. *Social Network Analysis and Mining*, 6(1):93, 2016.

[20] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne M. VanBriesen, and Natalie S. Glance. Cost-effective outbreak detection in networks. In Pavel Berkhin, Rich Caruana, and Xindong Wu, editors, *KDD*, pages 420–429. ACM, 2007.

[21] Junnan Lu and Alex Thomo. An experimental evaluation of giraph and graphchi. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2016*, pages 993–996, 2016.

[22] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken ichi Kawarabayashi. Dynamic influence analysis in evolving networks. *PVLDB*, 9(12):1077–1088, 2016.

[23] M. A. Porter, J.-P. Onnela, and P. J. Mucha. Influencers in social networks. *Fmsasg.com*, 2017.

[24] Krishnan Rajagopalan, Venkatesh Srinivasan, and Alex Thomo. A model for learning the news in social networks. *Annals of Mathematics and Artificial Intelligence*, 73(1):125–138, 2015.

[25] Kazumi Saito, Masahiro Kimura, Kouzou Ohara, and Hiroshi Motoda. Learning continuous-time information diffusion model for social behavioral data analysis. *Advances in Machine Learning*, pages 322–337, 2009.

[26] Maryam Shoaran and Alex Thomo. Fault-tolerant computation of distributed regular path queries. *Theoretical Computer Science*, 410(1):62 – 77, 2009.

[27] M. Simpson, V. Srinivasan, and A. Thomo. Clearing contamination in large networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1435–1448, June 2016.

[28] Michael Simpson, Venkatesh Srinivasan, and Alex Thomo. Efficient computation of feedback arc set at web-scale. *Proceedings of the VLDB Endowment*, 10(3):133–144, 2016.

[29] statista.com. Global social media ranking 2017 — statistic, 2017.

[30] Dan C. Stefanescu, Alex Thomo, and Lida Thomo. Distributed evaluation of generalized path queries. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 610–616. ACM, 2005.

[31] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. *CoRR*, abs/1404.0900, 2014.

[32] Babak Tootoonchi, Venkatesh Srinivasan, and Alex Thomo. Efficient implementation of anchored 2-core algorithm. In *Proceedings of ASONAM'17*, pages 1009–1016, 2017.

[33] youtube.com. Michelle phan, 2017.

[34] Honglei Zhuang, Yihan Sun, Jie Tang, Jialin Zhang, and Xiaoming Sun. Influence maximization in dynamic social networks. In Hui Xiong, George Karypis, Bhavani M. Thuraisingham, Diane J. Cook, and Xindong Wu, editors, *ICDM*, pages 1313–1318. IEEE Computer Society, 2013.