Scalable Analytics of Massive Graphs

by

Diana Popova
M.Sc., National Research University, Moscow, Russia, 1976

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Scalable Analytics of Massive Graphs

by

Diana Popova
M.Sc., National Research University, Moscow, Russia, 1976

Supervisory Committee

Dr. Alex Thomo, Supervisor
Department of Computer Science

Dr. Lin Cai
Department of Electrical and Computer Engineering

Dr. Bruce Kapron
Department of Computer Science

Dr. Wendy Myrvold
Department of Computer Science

Abstract

Graphs are commonly selected as a model of scientific information: graphs can successfully represent imprecise, uncertain, noisy data; and graph theory has a well-developed mathematical apparatus forming a solid and sound foundation for graph research. Design and experimental confirmation of new, scalable, and practical analytics for massive graphs have been actively researched for decades. Our work concentrates on developing new accurate and efficient algorithms that calculate the most influential nodes and communities in an arbitrary graph. Our algorithms for graph decomposition into families of most influential communities compute influential communities faster and using smaller memory footprint than existing algorithms for the problem. Our algorithms solving the problem of influence maximization in large graphs use much smaller memory than the existing state-of-the-art algorithms while providing solutions with equal accuracy. Our main contribution is designing data structures and algorithms that drastically cut the memory footprint and scale up the computation of influential communities and nodes to massive modern graphs. The algorithms and their implementations can efficiently handle networks of billions of edges using a single consumer-grade machine. These claims are supported by extensive experiments on large real-world graphs of different types.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I owe my profound gratitude to wonderful people, supporting my efforts and helping me all the way. I would love to include in this dissertation the names of all the people I have felt lucky and privileged to meet and work with, but it is impossible, because there are so many of them.

Dr. Alex Thomo, my best in the world supervisor. I attended all of the courses you taught, and they provided a solid foundation for my research work. But even more important for me was your role as my co-researcher, colleague, and supporter. I do not think I would be so happy, relaxed, and successful with any other supervisor. Your unwavering support meant the world to me, in my life as a graduate student. Your faith in my ability to complete the research and produce new interesting results helped me through moments of self-doubt. Your ideas inspired me to explore new avenues in my work.

Dr. Ken-ichi Kawarabayashi, my Japanese host supervisor for three summer studies. You generously shared with me your time and your knowledge. Each of our meetings gave me fresh impetus to proceed. Our collaboration on research papers was a great and satisfying experience for me. A huge part of this dissertation is written about the research that was inspired by your comments and suggestions.

Computer Science Department of UVic: professors, computer support team, office workers. Student teams I worked with on research and publications. Japanese team of Kawarabayashi Large Graph Project.

Thank all of you from the bottom of my heart, and I am happily looking forward to working with you in the future.

Chapter 1

Introduction

Massive, complex, interlinked information is collected by scientific research in different spheres of natural and social sciences. Graphs are commonly selected as a model of such information: graphs can successfully represent imprecise, uncertain, noisy data; graphs are well suited for data structure analysis; and graph theory has a well-developed mathematical apparatus forming a solid and sound foundation for graph research.

Connections between people or entities are modelled as graphs, where nodes represent the people or entities, and edges represent the connections. Many large graphs have been constructed this way coming from a multitude of systems and applications, such as social and web networks, product co-purchases, and protein interaction networks, to name a few.

For example, a social network can be modelled as a directed or undirected graph where individuals correspond to nodes, and connections between individuals correspond to edges. Edges/connections between graph nodes might represent affinity; *e.g.*, familiarity, or friendship, or following (as in "Twitter follower" or "Youtube channel subscriber").

Analyzing graph structure has been shown to be highly beneficial in practical applications. In this dissertation, we describe our research on two important problems in graph analytics: (1) finding influential communities, and (2) computing the most influential nodes. Our work concentrates on data structures used by the algorithms for keeping the intermediate results of computations in main memory. The intermediate results are further used for computing the most influential communities or nodes.

The main contribution of this work is designing data structures and algorithms that drastically cut the memory footprint and scale up the computation of influential communities and nodes to massive modern graphs with billions of edges.

1.1 Influential Communities

One of the most important tasks in analyzing graphs is finding communities of nodes that have close ties with each other [12, 17, 19, 40]. Discovering communities is of great importance in sociology, biology, computer science, and other disciplines where systems are often represented as graphs [14]. Communities are usually conceived as subgraphs with a high density of links within the subgraph and a comparatively lower density of links with the rest of the graph. The existence of community structure indicates that the nodes of the network are not homogeneous but divided into classes, with a higher probability of connections between nodes of the same class than between nodes of different classes.

Algorithms for finding communities in networks often rely only on structural information and search for cohesive subsets of nodes. Many works implicitly or explicitly assume that structural communities represent groups of nodes with similar non-topological properties or functions. In practice however, we would like to find communities that are not only cohesive, but also influential or important. For example, we would like to discover well-connected communities of *prolific* celebrities, *highly-cited* researchers, *outspoken* individuals, *authoritative* financial analysts, *etc.* It is far from certain that the communities reflecting only the graph structure are an adequate means to achieving this goal.

One of the drivers of community detection is the possibility to identify node classes, and to infer their attributes, when they are not directly accessible via experiments or other channels. Yang and Leskovec [46] found that the match between topological and supposed "ground truth communities" (metadata groups) has proven to be a challenging task, for all methods evaluated in their analysis. This questions the usefulness of the purely topological community detection algorithms to extrapolate the hidden (non-topological) features of the nodes.

To capture a non-topological property of communities, Li, Qin, Yu, and Mao introduced a novel community model called "k-influential community" [24] based on the concept of k-cores [37], with numerical values representing "influence" assigned to the nodes. They formulated the problem of finding the top-r most important communities as finding r connected k-core subgraphs ordered by the lower-bound of their importance/influence. Each

node gets its own value of influence (for example, PageRank, or the number of citations, or some other numerical measure of importance), and the influence of the community is defined as the smallest influence among the nodes that belong to the community.

Li *et al.* model embeds the node importance/influence into the process of community discovery. Based on this model, the problem of the influential community search is to efficiently find the top-r connected k-core communities in a network. Finding communities in a network is typically hard [14]. The number of communities within the network is unknown and the communities are often of unequal size and/or density. Straightforward search for the top-r k-core communities in a large network is impractical because there could be a large number of communities that satisfy the core constraint, and for each community, we need to check its importance. Despite these difficulties, several algorithms for top-r k-core community detection have been developed by Li *et al.* [24] with varying levels of time and space complexity.

In this dissertation, we focus on the data structures for an implementation of the Li *et al.* community model [24]. We propose new data structures for keeping the intermediate results and the fast new algorithms developed for computing k-influential communities. To fit massive graphs into memory, we use the Webgraph compression framework of Boldi and Vigna [5]. The compression is very high: even the largest graph we tested, Clueweb (75 billion edges), can easily fit in memory of a consumer-grade machine. Compressed graphs in the Webgraph format represent one of the data structures we successfully used for reducing the memory footprint.

We present:

1. Fast forward algorithms for computing top-r k-influential communities. Our algorithms achieve orders of magnitude speed-up compared to the direct algorithm of [24].

2. Backward algorithms for fast computing of the most influential communities. When the graph is big and r relatively small, these algorithms perform best and produce the result by only accessing a small portion of the graph.

3. Extensive experiments on large and very large graphs. Our biggest graph is Clueweb with about 1 billion nodes and 75 billion edges. The tests show that we are able to compute communities for every combination of k and r in a large range of values using the forward

algorithms. We can do this faster for a good number of k and r combinations using the backward algorithms.

With our implementations, we show that we can efficiently handle massive networks using a single consumer-grade machine within a reasonable amount of time. Details are available in Chapter 3.

1.2 Influence Maximization

Another actively researched problem in graph structure discovery is the problem of *influence maximization* (IM): in an arbitrary graph, given a size k, find a subset of the nodes S of size k that maximizes some *influence function*. A commonly used influence function is reachability as described in [15, 36, 21]: The network is modelled as a probabilistic directed graph where entities correspond to nodes. The graph and edge existence probabilities are given to an IM algorithm as input. The algorithm selects each edge (to reach a neighbouring node) with the given probability. That is, we are dealing with *probabilistic* reachability. Given a set of *seeds* (initial nodes), *influence estimation* (IE) is calculated as the expected total number of nodes reachable from all the seeds in the set S. The probabilistic influence *spread* is defined as the number of reachable nodes for a given edge probability.

Kempe *et al.* [21] researched several influence spread models, including the *Independent Cascade* (IC) model [15]. In this model, the probability of edge existence is an independent random variable. It is assigned to each directed edge (u, v). Starting from a node, in each step, information spreads to the node's neighbours with probability corresponding to the level of influence of the node over each neighbour. Kempe *et al.* [21] showed that IM on the IC model is *monotone* and *submodular*, and therefore, a Greedy algorithm produces good quality solutions. IM on the IC model encodes the classic maximum coverage problem and is NP-hard as shown in [21]. For a practical IM algorithm, Kempe *et al.* proposed using an *approximate* Greedy algorithm. The influence of the approximate Greedy solution with a given number of seeds is $(1 - 1/e - \epsilon)$ of the optimal solution, for any $\epsilon > 0$ (proven in [28]). IC became a standard model of influence spread, and we are using it for our algorithms.

In 2013, a different method for solving IM was proposed by Borgs *et al.* [7]: the Reverse

Influence Sampling (RIS) method[1]. The idea is to select a node v uniformly at random, and determine the set of nodes that *would have influenced* v. This can be done by simulating the influence process using the IC model in the graph with the directions of edges reversed (*transpose* graph). If a certain node u appears often as influential for different randomly selected nodes, then u is a good candidate for a most influential node. RIS is a fast algorithm for IM, obtaining the approximation factor of $(1 - 1/e - \epsilon)$, for any $\epsilon > 0$, in time $O((m + n)k\epsilon^{-2}\log(n))$, where n is the number of nodes, m is the number of edges, and k is the number of seeds (proven in [7]). But RIS needs to sample nodes many times and consumes vast amounts of memory resources needed for keeping the sampling results. The problem of scalability remains.

We propose:

1. A new approach to IM: by minimizing the memory footprint of IM algorithms, we significantly increase the size of graphs that can be processed.

2. Data structures that drastically shrink the memory footprint, while preserving the intermediate results of IM computation necessary for calculating the seed set.

3. New accurate and space-efficient IM algorithms: NoSinges, NoSinglesTopNodes, CutTheTail1, and CutTheTail2.

4. Experiments on 15 real-world, different types of graphs conducted on a consumer-grade laptop with 16 GB RAM, with statistical analysis of the results.

5. Experimental comparison of the quality of solution and space performance of our algorithms *vs.* Dynamic Influence Maximization (DIM) [32] and Dynamic Stop-And-Stare (D-SSA) [30] algorithms. Memory required for our algorithms is orders of magnitude smaller than the one for DIM (up to 50,000 times smaller) or D-SSA (up to 3000 times smaller), for the same graph and quality of solution.

We present a thorough analysis of our algorithms concluding that it is practical to compute IM for large networks on a laptop, and keep the intermediate results for future use. Details are available in Chapters 5, 6, and 7.

The publications based on the work of this dissertation are as follows:

1. CIKM 2016 [8]. The paper describes research on k-influential community discovery.

[1]The latest version 5 of the paper, issued on the 22nd of June 2016, can be found at `https://arxiv.org/pdf/1212.0884.pdf`

Details can be found in Chapter 3.

2. EDBT 2018 [34]. The paper reports on research of data structures used for keeping intermediate results of IM computation. This research is presented in Chapter 5.

3. SSDBM 2018 [35]. A new space-efficient algorithm, NoSingles, is presented in the paper and proven to be faster and have smaller memory footprint than the existing IM algorithms. Details are in Chapter 6.

4. Submitted to VLDB 2019 [33]. This paper presents new heuristic algorithms for IM and research on their accuracy and efficiency. Details can be found in Chapter 7.

Chapter 2

Problem Statement for Influential Communities

This chapter starts with basic definitions necessary for this dissertation. In the following sections, influential community definitions and formal problem statements are given.

2.1 Basic Graph Definitions

A *graph* is an ordered pair $G = (V, E)$, where V is a set of *nodes* and E is a set of *edges*. A node is a fundamental unit of a graph, featureless and indivisible. An edge is a 2-element subset of V: an edge between node u and node v is denoted as (u, v). The nodes u and v are said to be *adjacent* to one another, and the edge (u, v) is said to be *incident* to u and v.
Note: nodes are often called "vertices", and edges are called "arcs". In this dissertation, we are using the terms "node" and "edge" for the elements of graphs.
The *order* of a graph is $|V|$, its number of nodes. The *size* of a graph is $|E|$, its number of edges. Throughout the dissertation, we denote the order as n, and the size as m.
In an *undirected graph*, edges are unordered pairs of nodes, while in a *directed graph*, edges are ordered pairs: one node is the tail, and the other is the head. For a directed edge (u, v), u is the tail, and v is the head.
On a diagram of a graph, a node is usually represented with a circle and an edge as a line, for an undirected graph, or an arrow, for a directed graph. The line or arrow connects two nodes. If an edge is drawn as an arrow, it points from the "tail" node to the "head" node.
The *degree* of a node is the number of edges incident to it.
A *neighbourhood* of a node v in a graph G is the set of all nodes in G adjacent to v.

A *subgraph* of a graph G is another graph whose nodes and edges are subsets of V and E of G. An *induced subgraph* of a graph G is a subgraph whose nodes are a subset of V and edges include *all* the edges of G connecting the subgraph nodes.

2.2 Influential Community Definitions

All our algorithms for the influential communities discovery are written for undirected graphs, and the following definitions are applicable to undirected graphs.
A graph *path* is a sequence of distinct edges that connect a sequence of distinct nodes.
A *connected component* of an undirected graph is a subgraph where any two nodes are connected to each other by paths.

A problem of large graphs analytics in this dissertation is finding a decomposition of a graph into a family of *communities*. Communities are usually conceived as subgraphs with a high density of edges within the subgraph and a comparatively lower density of edges with the rest of the graph. We would like to find communities that are not only cohesive, but also influential or important. To capture such communities, Li *et al.* introduced a novel community model called "k-influential community" [24], with values of "influence" or "importance" assigned to the nodes. The k-influential community model is based on the concept of k-cores. The next subsection describes k-cores and provides the necessary definitions.

2.2.1 The k-core decomposition of a graph

In 1983, S.B. Seidman defined a k-core [37] as follows:

"Let G be a graph. If H is a subgraph of G, $\delta(H)$ will denote the minimum degree of H; each point of H is thus adjacent to at least $\delta(H)$ other points of H. If H is a maximal connected (induced) subgraph of G with $\delta(H) \geq k$, we say that H is a k-core of G."

Note that Seidman called a graph node "a point". In this dissertation, we present a slightly modified definition of k-core:

Definition 1. *In a subgraph H of a graph G, $\delta(H)$ denotes the minimum degree of any node in H. If H is a maximal induced subgraph of G with $\delta(H) \geq k$, we say that H is a k-core of*

G.

The difference is in omitting the adjective "connected" for the subgraph H: the k-core might not be connected, but consists of more than one connected components.

The above definition can be illustrated with an example:

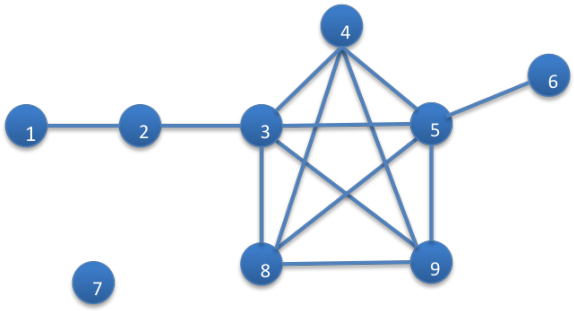


Figure 2.1: Toy graph to illustrate k-core decomposition.

1	2	3	4	5	6	7	8	9	index
1	1	4	4	4	1	0	4	4	core number

Figure 2.2: Core numbers for the nodes.

In Figure 2.1 graph nodes have their IDs as integers printed inside the circle. The k-core decomposition is performed by systematically deleting the nodes with the smallest degree and their incident edges, starting from the singletons (if they are present, otherwise starting from a smallest degree node) and going through the maximum possible k for a given graph. For the example in Figure 2.1:

1. The starting value for k equals zero, because the graph contains a singleton (node 7). The 0-core includes the whole graph.

2. To get the 1-core, we delete node 7. The remaining nodes {1 – 6, 8, 9} with the incident edges constitute the 1-core.

3. To get the 2-core, we have to delete nodes 1 and 6 with their incident edges. After node 1 and edge (1,2) are deleted, we have to delete node 2, as its degree now equals one and hence node 2 cannot be included in 2-core. The remaining nodes {3 – 5, 8, 9} with the incident edges constitute the 2-core.

4. To get the 3-core, the nodes with degree less than three are deleted. But there are no such nodes in the remaining subgraph; so, according to Definition 1, the nodes {3 – 5, 8, 9}

with the incident edges is a 3-core.

5. To get the 4-core, the nodes with degree less than four are deleted. But there are no such nodes in the remaining subgraph; so, according to Definition 1, the nodes $\{3-5, 8, 9\}$ with the incident edges is a 4-core.

6. To get the 5-core, the nodes with degree less than five are deleted. After deleting such nodes, the graph became empty (no nodes left), and we conclude that the maximum k for the example is four.

The result of the k-core decomposition can be presented as a list of graph nodes with their corresponding *core numbers*.

Definition 2. *The core number of a node is the maximum value of k in all k-cores the node participates in.*

In Figure 2.2, the core numbers for all the nodes are listed in an array. The index of an array element is the node ID, and the element value equals the core number of the node. Thus, the core number of node 7 equals zero, while the core number of node 3 equals four. It is interesting to note that in the example, there are no nodes with core numbers equal to two or three. It is often the case with real-world graphs: core numbers have large "gaps" in sequences. More discussion of this effect follows in Chapter 3.

Observation 1. *The k-core decomposition of a graph is* **unique** *and* **deterministic**. *If a graph contains several smallest degree nodes, it does not matter which node (with the smallest degree) the deletions start from: the core numbers calculated for the graph nodes will be the same.*

Now we are ready to define k-influential communities.

2.2.2 The k-influential community definitions

Consider an undirected graph $G = (V, E)$. An importance/influence weight array w of size n is given, such that $w[v]$ is the weight of $v \in V(G)$. These weights can represent centrality scores, publication indices (p-index or h-index, see [41]), wealth, social status, *etc.* A strict total order is assumed on the weights of array w; this can be achieved by breaking ties based on the lexicographical order of node IDs.

Definition 3. *Given an undirected graph G and an induced subgraph H of G, the weight of H is defined as the minimum weight of the nodes in H*[1]*.*

The idea of the influential community model of [24] is to extract connected subgraphs of high influence/weight.

Definition 4. *Given an undirected graph G and an integer k, a k-influential community is an induced subgraph H^k of G that meets all the following constraints.*
Connectivity: H^k is connected;
Cohesiveness: each node u in H^k has degree at least k;
Maximal structure: there is no other induced subgraph $\widetilde{H}$ such that
(1) $\widetilde{H}$ satisfies the connectivity and cohesiveness constraints,
(2) $\widetilde{H}$ contains H^k, and
(3) weight of $\widetilde{H}$ = weight of H^k.

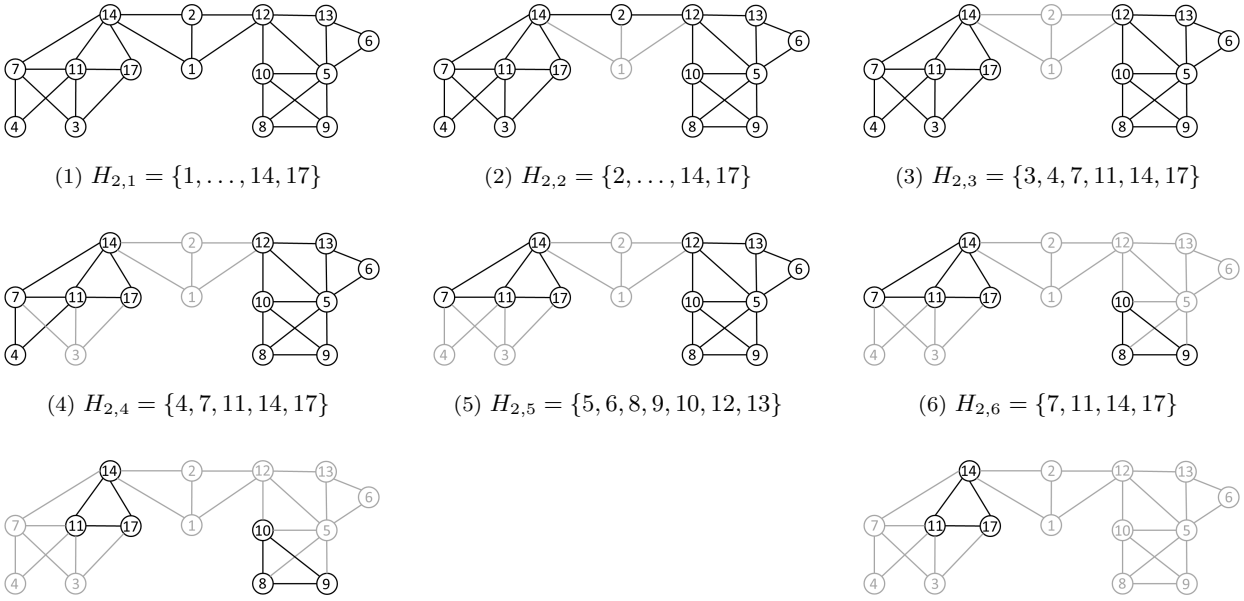
The cohesiveness constraint indicates that a k-influential community is a (subgraph of) the k-core. In general, a k-core is not necessarily connected, *i.e.* it can contain several connected components. If this is the case, the k-core will contain several k-influential communities. With the connectivity and cohesiveness constraints, we can ensure that a k-influential community is a connected and cohesive subgraph. With the maximal structure constraint, we can guarantee that any k-influential community *cannot* be contained in another k-influential community with an equivalent influence. In this dissertation, we use the following definition:

Definition 5. *Subgraphs satisfying the maximal structure constraint as defined above, are called Maximal Connected Components (MCCs).*

An example of a k-influential communities extraction, also called a k-influential decomposition, is presented in Figure 2.3. An integer k is an input parameter for a k-influential decomposition. The decomposition starts from finding the k-core of the graph G. The k-core is denoted C_k. The graph depicted in Figure 2.3.1 is the 2-core of some graph G, and can be denoted as C_2.

For simplicity, the weights of nodes are set to be equal to their IDs. The k-influential decomposition is processed by "peeling off" the graph, that is, by a systematic deletion of

[1] See [24] for the case studies showing the benefits of the proposed model as well as its advantages over other models.



(1) $H_{2,1} = \{1, \ldots, 14, 17\}$ (2) $H_{2,2} = \{2, \ldots, 14, 17\}$ (3) $H_{2,3} = \{3, 4, 7, 11, 14, 17\}$

(4) $H_{2,4} = \{4, 7, 11, 14, 17\}$ (5) $H_{2,5} = \{5, 6, 8, 9, 10, 12, 13\}$ (6) $H_{2,6} = \{7, 11, 14, 17\}$

(7) $H_{2,7} = \{8, 9, 10\}$ (8) $H_{2,8} = \{11, 14, 17\}$

Figure 2.3: k-influential decomposition for $k = 2$. The node weight equals the node ID. The greyed out nodes and edges are deleted.

a smallest weight node with its incident edges. The remaining nodes are checked to make sure that their degrees equal at least k. If the degrees became less than k, the nodes are also deleted, together with their incident edges. For example, when we delete node 5 in Figure 2.3.5, nodes 6, 12, and 13 are recursively deleted as well. The deleted nodes and edges are greyed out in Figure 2.3.6.

As the decomposition proceeds, after each deletion of the smallest weight node (and all other nodes that do not belong to the k-core anymore, together with their incident edges), the resulting subgraphs H become smaller and smaller. It is clearly seen in Figure 2.3. Each resulting subgraph H contains one or more k-influential communities, with higher and higher weights. Recall, that the weight of a k-influential community is equal to the smallest weight of its nodes, according to Definition 3.

The next k-influential community to discover during the k-influential decomposition is picked up among the communities of the current subgraph H. It is denoted $H_{k,i}$, where k is the value of the k-core, and i is the number of iteration in H. According to Definition 5, an $H_{k,i}$ is the MCC containing the node of the smallest weight, $v_{k,i}$. For example, in Figure 2.3.7,

$v_{2,7} = 8$ and $H_{2,7} = \{8, 9, 10\}$. Note, that notation $\{8, 9, 10\}$ means *the subgraph induced by the set of nodes* $8, 9, 10$.

The described process of graph k-influential decomposition is applied to undirected graphs with fixed edges (as opposed to probabilistic graphs, with probabilities of edge existence).

Observation 2. *The k-influential decomposition of a graph is* **unique** *and* **deterministic**, *for a given k and nodes' influence. The influence ties are resolved by ranking the nodes in their lexicographical order.*

It can be verified that either $H_{k,i} \supset H_{k,j}$, for $1 \leq i < j$, or $H_{k,i} \cap H_{k,j} = \emptyset$. The first case happens when $v_{k,i}$ is in the same MCC as $v_{k,j}$, whereas the second happens when they are not. There are possibly several chains of such $\supset$ containments. For Figure 2.3, we have two such chains, $H_{2,1} \supset H_{2,2} \supset H_{2,3} \supset H_{2,4} \supset H_{2,6} \supset H_{2,8}$, and $H_{2,1} \supset H_{2,2} \supset H_{2,5} \supset H_{2,7}$. The last k-influential community in each chain does not contain any other k-influential community.

Definition 6. *A k-influential community that does not contain any other k-influential communities is called a* non-containing *k-influential community.*

After finding all k-influential communities, we might need to output only the most influential ones. Li *et al.* [24] introduced another parameter, r, which is the number of the most important/influential k-influential communities to output.

Now we define two top-r k-influential community problems.

Problem 1 (Influential Community Problem)**.** *Given a graph G, and two positive integers k and r, discover the top-r (w.r.t. weight) k-influential communities of G.*

Problem 2 (Non-containing Influential Community Problem)**.** *Given a graph G, and two positive integers k and r, discover the top-r (w.r.t. weight)* non-containing *k-influential communities of G.*

Li *et al.* [24] designed and implemented two algorithms solving the above problems. We would like to thank Dr. R. Li for sharing the code for their implementations with us. We studied the problems and realized that Li *et al.* algorithms have a limited practical usage due to rather large main memory consumption. To scale up the solution to massive modern

networks, new design ideas are called for. We present our algorithms for Problem 1 and 2 in Chapter 3.

Chapter 3

Influential Communities Solution

In this chapter, we describe the data structures and the algorithms we designed and implemented for k-influential community discovery. A formal definition of a k-influential community is given in Section 2.2.2.

3.1 k-influential Decomposition

A k-influential decomposition of a graph starts from the k-core decomposition. We implemented in Java the $O(m)$ algorithm for k-core decomposition designed by V. Batagelj and M. Zaversnik [2]. The Batagelj and Zaversnik algorithm is a sophisticated algorithm achieving linear time and space complexity by using only flat (one-dimensional) arrays. The researchers expertly use this simple data structure to manipulate the input graph adjacency list. The algorithm outputs the list of graph nodes with the corresponding core numbers.

Our implementation outputs not only core numbers for all the nodes, but also provides a list of *distinct* core numbers for the processed graph. As we noted in Section 2.2.1, the computed core numbers have "gaps" in their sequence. For example, in the k-core decomposition of one of the graphs we tested, Arabic-2005, the maximum value of k is 3,247, but the list of distinct core numbers is only 743 elements long. Characteristics of Arabic-2005 are listed in Table 3.1, as well as the five largest k values found in its decomposition.

n	**m**	**dmax**	**kmax**	**k-avg**
22.7 M	640 M	575,628	3247	28.14
Five largest k for cores				
3247	3240	3127	2087	2086

Table 3.1: Characteristics of Arabic-2005.

We see that after the 3247-core, the Arabic-2005 graph contains a 3240-core (a gap of six missing core numbers); after the 3240-core the next one is the 3127-core (a gap of twelve missing core numbers); after the 3127-core the next one is the 2087-core (a gap of over one thousand missing core numbers!). We will use the information from the distinct core number list to choose k for the k-influential decomposition.

3.1.1 Influential Community Example

Using the authorship network from ArnetMiner (`http://arnetminer.org`) and weighing authors by their productivity index (p-index defined in [41]), we computed for $k = 3$ and $k = 6$ the top-1 communities as shown in Figure 3.1. The influence of authors in presented communities is undisputed, with the first community having a higher p-index (an influence measure for this example) than the second. In general, k can serve as a tradeoff between cohesiveness and importance/influence: The higher the k, the higher the cohesiveness, but lower the influence of computed communities.

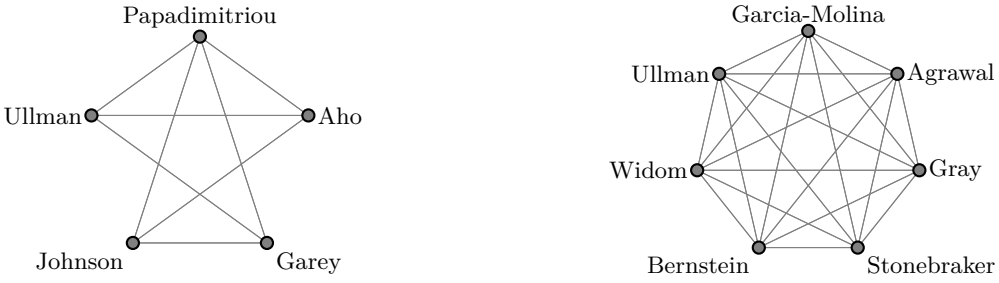


Figure 3.1: Arnet: [left] top-1, $k = 3$, [right] top-1, $k = 6$.

How did we get these results? We used the peel-off procedure described in Section 2.2.2, which is the algorithm proposed by Li *et al.* [24, Algorithm 2]. We call this algorithm *C-original*, and it solves Problem 1. The algorithm is presented below.

The bottleneck of C-original is the very large number of MCC computations it executes starting from each minimum weight $v_{k,i}$. Along the way, we need to keep a cache of the last r influential communities thus discovered. This is because the communities are generated in reverse order of their influence. Next, we present new algorithms for Problem 1 and Problem 2 that drastically reduce the number of MCC computations or completely eliminate them.

Algorithm 1 Top-r influential communities (C-original)

Input: G, w, k, r

Output: $H_{k,\tau_{p-r+1}}, \ldots, H_{k,\tau_p}$

```
1: C ← C_k(G)
2: i ← 1, cache ← ∅
3: while C ≠ ∅ do
4:     Let v be a minimum-weight node in C
5:     τ ← w[v]
6:     H ← MCC(C,v)
7:     if cache.size() = r then
8:         cache.deleteFirst()
9:     cache.addLast(H, τ)
10:    RDelete(C,v)
11:    i ← i + 1
12: Output cache
```

Algorithm 2 Procedure RDelete

```
1: procedure RDELETE(C, v)
2:     for all u ∈ N_C(v) do
3:         Delete edge (u, v) from C
4:         if d_C(u) < k then
5:             RDelete(C, u)
6:     Delete v from C
```

3.2 Forward Algorithms

Let us first analyze algorithm C-original described in the previous section. Since the complexity of MCC is $O(m)$ and we compute it for each node, the time complexity of C-original is $O(m \cdot n)$, which is impractical for big graphs. Regarding space complexity: we need to remember the last r communities computed so far. Since we only store the nodes of these communities, the space complexity is $O(m + n \cdot r)$. For small r (say, not more than 10), we can say that the second term $n \cdot r$ is absorbed by the first, m. However, for larger r's, $n \cdot r$ becomes eventually bigger than m. Therefore, the algorithm has also a memory bottleneck.

In the following we describe our proposed algorithms for Problem 1. They outperform

C-original by orders of magnitude.

3.2.1 Algorithm C1

What takes most of the time in C-original is computing MCC's for each $C_{k,i}$. The early $C_{k,i}$s are especially expensive as they can be quite big in size. Furthermore, often a $C_{k,i}$ is just *slightly* smaller than the previous one, $C_{k,i-1}$. In practice, for most of early iterations, the peel-off process does not remove more than few nodes. We can observe this fact even in the small example we presented in Figure 2.3. Therefore, many MCC computations are performed on almost identical graphs.

Peel-offs are performed by a recursive delete procedure, *RDelete*, which takes as parameters a k-core subgraph C and a node v. It deletes v from C, then recursively deletes all v's neighbours whose degree becomes less than k, until there are no more nodes with degree less than k. In the end, what remains of C is either a k-core subgraph, or an empty graph. *RDelete* is inexpensive to execute: The total time spend on all *RDelete* calls together is $O(m)$, i.e. not more than just traversing the graph. So, the bottleneck is MCC computations.

How can we reduce MCC computations? We only need to run MCC for the last r iterations because only these iterations compute the top-r results.

The problem is that we do not know beforehand how many iterations there will be in total. However, this can be found by running the logic of node removal twice. A new algorithm, C1, is given in Algorithm 3.

The first run of node removals (lines 1-5) does not compute MCC at all. It selects the minimum weight node $v_{k,i}$ (variable v) from the current $C_{k,i}$ (variable C), then peels off $v_{k,i}$ by calling *RDelete*, and records the iteration by incrementing variable i. The purpose of this run is to find out how many iterations are needed. The final value of i will be the total number of iterations.

The second run of node removals (lines 6-13) starts anew. C is reinitialized before the second run. Knowing the total number of iterations i, MCCs are computed only in the last r iterations: we use j instead of i, and only compute an MCC when $j > i - r$. Now we can verify the following theorems.

Algorithm 3 Top-r influential communities (C1)

Input: G, w, k, r

Output: $H_{k,p-r+1}, \ldots, H_{k,p}$

```
1: C ← C_k(G), i ← 1
2: while C ≠ ∅ do
3:     Let v be a minimum-weight node in C
4:     RDelete(C,v)
5:     i ← i + 1
6: C ← C_k(G), j ← 1
7: while C ≠ ∅ do
8:     Let v be a minimum-weight node in C
9:     if j > i − r then
10:        H ← MCC(C,v)
11:        Output H
12:    RDelete(C,v)
13:    j ← j + 1
```

Theorem 1. *Algorithm C1 correctly computes all the top-r MCC communities of a given graph G.*

Since we run MCC only r times, we have that

Theorem 2. *The time complexity of C1 is $O(m \cdot r)$.*

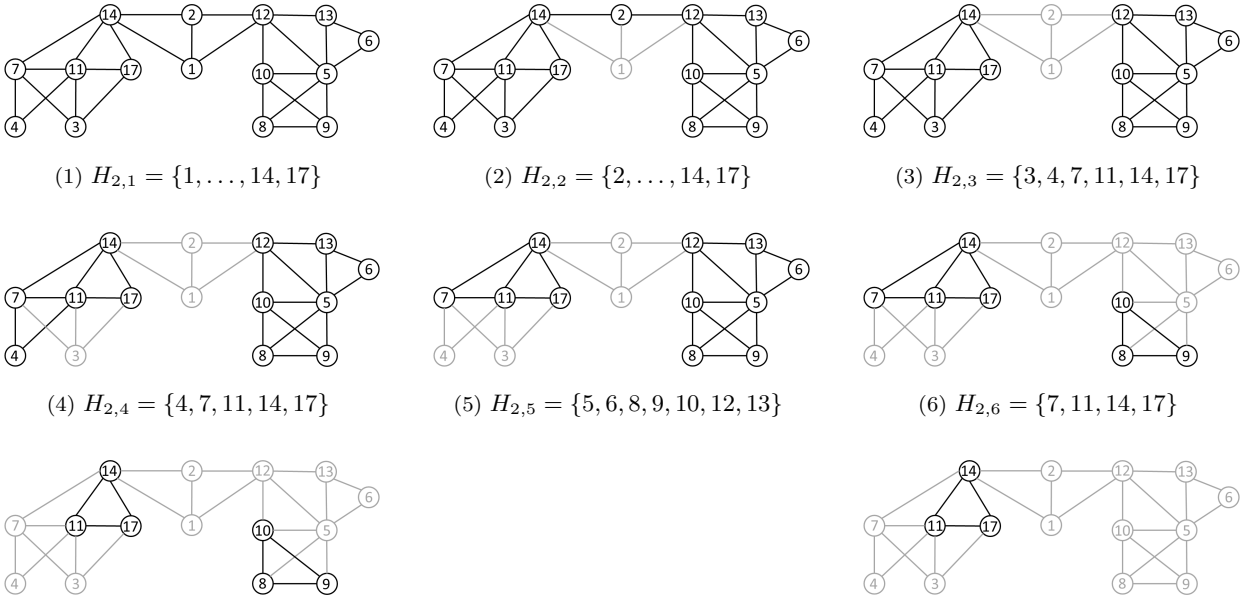
$O(m \cdot r)$ is much smaller than $O(m \cdot n)$ for practical values of r. Note that $O(m \cdot r)$ is only a loose upper bound for C1, because the last r iterations potentially operate on very small subgraphs obtained after deleting most of the nodes in the first $i - r$ iterations. Therefore, in practice, MCC computations of the last r iterations cost significantly less than $O(m)$. In our experiments, we observe C1 to be orders of magnitude faster than C-original.

Regarding space complexity, note that it is no longer necessary to store the last r communities computed so far. We only compute the very last r communities, which not only are the smallest r communities, but can also be printed (or saved) right away. Therefore we state the following theorem.

Theorem 3. *The space complexity of C1 is $O(m)$.*

Next, we present a better algorithm which reduces the time (almost) by half.

3.2.2 Algorithm C2



(1) $H_{2,1} = \{1, \ldots, 14, 17\}$ (2) $H_{2,2} = \{2, \ldots, 14, 17\}$ (3) $H_{2,3} = \{3, 4, 7, 11, 14, 17\}$

(4) $H_{2,4} = \{4, 7, 11, 14, 17\}$ (5) $H_{2,5} = \{5, 6, 8, 9, 10, 12, 13\}$ (6) $H_{2,6} = \{7, 11, 14, 17\}$

(7) $H_{2,7} = \{8, 9, 10\}$ (8) $H_{2,8} = \{11, 14, 17\}$

Figure 3.2: k-influential decomposition for $k = 2$. The node weight equals the node ID. The greyed out nodes and edges are deleted.

The question to discuss: How to avoid running the second **while** loop as in Algorithm 3 and cut the running time in (about) half?

We introduce a hash-based structure I, which we call the *iteration-delete-history*. This is a hash-table indexed by i, the iteration number. We store in $I(i)$ a list of nodes deleted in iteration i. For an illustration, consider Figure 3.2. For this example, we have 8 iterations, and

$I(1) = \{1\}$,

$I(2) = \{2\}$,

$I(3) = \{3\}$,

$I(4) = \{4\}$,

$I(5) = \{5, 6, 12, 13\}$,

$I(6) = \{7\}$,

$I(7) = \{8, 9, 10\}$,

$I(8) = \{11, 14, 17\}$.

The algorithm using the iteration-delete-history I is given in Algorithm 4.

Algorithm 4 Top-r influential communities (C2)

Input: G, w, k, r

Output: $H_{k,p}$, ..., $H_{k,p-r+1}$

```
1: C ← C_k(G), i ← 1, I ← ∅
2: while C ≠ ∅ do
3:     Let v be a minimum-weight node in C
4:     I(i) ← ∅
5:     RDelete2(C,v,I,i)
6:     i ← i + 1
7: alive ← 0
8: for j = i downto i − r + 1 do
9:     for all v ∈ I(j) do
10:        alive[v] ← 1
11:    v ← I(j).first()
12:    H ← MCC(G, v, alive)
13:    Output H
```

Structure I is populated during the run of the **while** loop. More specifically, it is populated in a modified *RDelete* procedure.

Algorithm 5 Procedure RDelete2

```
1: procedure RDELETE2(C, v, I, i)
2:     for all u ∈ N_C(v) do
3:         Delete edge (u, v) from C
4:         if d_C(u) < k then
5:             RDelete2(C, u)
6:     Delete v from C
7:     I(i).add(v)
```

The modified *RDelete*, which we call *RDelete2*, takes two extra parameters, I and i, and has one extra operation, the insertion of v to $I(i)$. More specifically, it adds v to $I(i)$ (Algorithm 5), which is the last operation in *RDelete2*.

Since the procedure is recursive, all the deleted nodes in iteration i will be inserted into $I(i)$. We implemented I as a flat array of dimension n accompanied by another array storing the positions of bucket boundaries. Since the buckets of I are filled out in order of increasing i, each operation on I takes constant time.

We do not execute any MCC computations in the **while** loop of Algorithm 4. Once the **while** loop completes, we start running the necessary r MCC computations in the subsequent **for** loop. However, since the nodes are deleted at this point, we need each time to make some nodes *alive* again. The **for** loop goes downwards starting from the maximum iteration number, i, and ending in $i - r + 1$. First, we make "alive" the nodes deleted in the last iteration of the **while** loop, then the nodes deleted in the second last iteration, and so on. We record the nodes that become alive in an array called *alive*. Each time we make a set of nodes alive, we run an MCC computation. The MCC computation works on the original graph, G, consulting array *alive* as it performs a Depth-First-Search (DFS). Only the alive nodes are considered for computing the connected components. This version of MCC is given in Algorithm 6.

Algorithm 6 MCC with alive array

```
1: procedure MCC(G, v, alive)
2:     cc ← ∅
3:     MCC-DFS(G, v, alive, cc)
4:     return cc
5: procedure MCC-DFS(G, v, alive, cc)
6:     cc.add(v)
7:     for all u ∈ N_G(v) do
8:         if alive[u] = true and u ∉ cc then
9:             MCC-DFS(G, u, alive, cc)
```

It can be verified that algorithm C2 produces the same result as C1, just in reverse order, i.e. $H_{k,p}$, ..., $H_{k,p-r+1}$. Therefore, we can state the following theorem.

Theorem 4. *Algorithm C2 correctly computes all the top-r influential communities of a given graph G.*

The asymptotic time complexity of C2 is the same as that of C1; however, in terms of

constant factors, C2 is about twice as fast as C1.

For the space complexity, observe that structure I takes $O(n)$ space, which is absorbed by $O(m)$ needed to hold the graph (typically true for a compressed graph as well).

Therefore, we state the following theorem.

Theorem 5. *The space complexity of C2 is $O(m)$.*

3.2.3 Algorithm NC

In [24], the computation of *non-containing* (NC) communities is done by modifying the C-original (Algorithm 1) to check each time whether upon calling *RDelete* in an iteration i all the nodes of $H_{k,i-1}$ (of the previous iteration) are deleted. In such a case, it can be concluded that $H_{k,i-1}$ is a non-containing community. We call this algorithm NC-original; it still calls a MCC procedure to calculate $H_{k,i-1}$. As such, the performance of NC-original is similar to C-original. For big graphs, both of them are not practical.

Here we propose another algorithm that completely eliminates the need to run MCC computations. All the information we need for the computation of non-containing communities is in the iteration-delete-structure, I, that we maintain. We formulate the following definition and then a lemma.

Definition 7. *Given a node v, the* current degree *of v is the number of alive neighbours of v.*

We record current degrees in an array d. While a node v is alive, $d[v]$ will contain the current degree of v. When v is deleted, $d[v]$ is not updated anymore, i.e. for the deleted nodes, d will remember their degrees at the time (iteration) of their deletion. Now, if in some iteration i, we have that for each $v \in I(i)$, $d[v] = 0$, then all the nodes neighbouring some node in $I(i)$ are gone (already deleted), i.e. the set of nodes in $I(i)$ was the last standing community in a community containment chain. Based on this reasoning we have the following lemma.

Lemma 1.

1. *For each non-containing $H_{k,i}$, $H_{k,i} = I(i)$.*

2. Let $i \geq 1$. If for each $v \in I(i)$, $d[v] = 0$, then $I(i)$ is a non-containing influential community.

Proof. (1) can be verified from the description of iteration-delete-history data structure in Section 3.2.2 and Definition 6. For (2), suppose that $I(i)$ is not a non-containing influential community, i.e. we have that $I(i) \supset H_{k,i+1}$, and this is a strict containment. Since $I(i)$ is a connected component, there exist at least one edge between some node $v \in I(i)$ and some node $u \in H_{k,i+1} \setminus I(i)$. Hence $d[v] \geq 1$, which is a contradiction. □

Algorithm NC1 pseudocode is shown in Algorithm 7. We also modified *RDelete2* to update array d during deletions. The modified procedure, *RDelete3*, is shown in Algorithm 8.

Algorithm 7 Top-r non-containing communities (NC1)

Input: G, w, k, r

Output: Top-r non-containing $H_{k,j_{\max}-r+1}, \ldots, H_{k,j_{\max}}$

```
C ← C_k(G)
for all node v of C do
    d[v] = d_C(v)
i ← 1, I ← ∅, j ← 1
while C ≠ ∅ do
    Let v be a minimum-weight node in C
    I(i) ← ∅
    RDelete3(C, v, I, i)
    isNC ← true
    for all v ∈ I(i) do
        if d[v] > 0 then
            isNC ← false
    if isNC = true then
        H ← I(i)
        Output H
        j ← j + 1
        if j > r then
            break
    i ← i + 1
```

Algorithm 8 Procedure RDelete3

```
1: procedure RDELETE3(C, v, I, i)
2:     Mark v
3:     for all u ∈ N_C(v) do
4:         d[u] ← d[u] − 1;
5:         if u is not marked and d[u] < k then
6:             RDelete3(C, u)
7:     Delete v from C
8:     I(i).add(v)
```

NC1 completely eliminates MCC computations. It starts by initializing C to $C_k(G)$, and the current node degrees to their degrees in C. In the **while** loop, after populating $I(i)$ via *RDelete3*, we check to see whether all the nodes in $I(i)$ have a degree of zero (lines 10-12). If true, then $I(i)$ is a non-containing community.

Based on the above reasoning and Lemma 1, we state the following theorem.

Theorem 6. *Algorithm NC1 correctly computes all the top-r non-containing influential communities of a given graph G.*

NC1 only iterates once over the graph and the only structure it uses is I. Therefore, we have the following theorem.

Theorem 7. *The time and space complexity of algorithm NC1 is $O(m)$.*

3.3 Backward Algorithms

So far, the algorithms we presented were forward; they were peeling off the graph from the lowest weight nodes to the highest. Such an approach is reasonable when r (in top-r) is big. However, imagine what happens when r is moderate, say we want to see the top-10 communities quickly. With the forward approach, we would need to start working our way up from the smallest weight nodes, and only at the end of the computation be able to see the top communities.

The approach we propose in this section is backward. It starts with a state where all the nodes are initially considered "deleted". Then, in each iteration, we "resurrect" a deleted

node v of the highest-weight (among the deleted nodes) and see whether v and the other resurrected nodes before v are able to form a k-core. In such a case, we claim (and show) that v is a $v_{k,i}$ for some i.

The benefit of this idea is that we can produce top-r communities for moderate r quickly without processing the majority of low weight nodes. For moderate values of r the time required is better than for the forward approaches as only a small part of the graph is accessed. This is especially pronounced for big graphs. As r grows, the time taken by the two approaches starts converging. Eventually, for some r, the backward approach will take more time than the forward one, as the determination whether the resurrected nodes form a k-core takes more and more time.

3.3.1 Algorithm C3

We present the pseudocode for the backward computation of influential communities (C3) in Algorithm 9.

Algorithm 9 Top-r influential communities (C3)

Input: G, w, k, r

Output: $H_{k,p}$, ..., $H_{k,p-r+1}$

```
1:  for all v ∈ V do
2:      alive[v] ← false
3:      cores[v] ← 0
4:  i ← 1
5:  for j = n downto 1 do
6:      Let v be a maximum-weight deleted node in V
7:      alive[v] ← true
8:      updateCores()
9:      if cores[v] ≥ k then
10:         H ← MCC(G, v, cores)
11:         Output H
12:         i ← i + 1
13:         if i > r then
14:             break
```

We start by making all the nodes "deleted". Then we resurrect nodes in order of their importance starting from the most important node. Each time, we update the core values of nodes made alive so far. For this we call the *updateCores* procedure, which detects whether the core numbers of the alive nodes have the potential to be updated, and if so, it updates them. Often there is no need to update cores because the node just resurrected does not have sufficient connections with the nodes already resurrected. If the node just resurrected, say v, happens to have a core value that is greater or equal to k, then we conclude that v is one of the $v_{k,i}$ nodes, (minimum weight vertice in $C_{k,i}$), and as such, we compute MCC starting from v and using only the alive nodes having a core number larger or equal to v. This version of MCC computation only considers a node u if $cores[u] \geq k$. It is very similar to Algorithm 6. Instead of condition $alive[u] = true$, we will have $cores[u] \geq k$.

To show the soundness and completeness of Algorithm 9, we first present the following lemmas.

Lemma 2. *Let v be the maximum-weight deleted node in V that gets resurrected in a given iteration. If v belongs to a k-core of alive (resurrected earlier) nodes, then, v is the minimum weight node in $H_{k,i}$, i.e. $vis v_{k,i}$.*

Proof. Suppose node $v_{k,i}$ is already alive and $w[v_{k,i}] < w[v]$. This contradicts the logic of resurrection: each time, we resurrect the maximum-weight node from the deleted nodes. $v_{k,i}$ with $w[v_{k,i}] < w[v]$ could not be resurrected earlier than v and could not be already among the alive nodes. □

Now, we can show that we do not miss any $v_{k,i}$ in the backward direction. We give the following lemma, whose proof follows directly from the definitions and so we omit it.

Lemma 3. *Let $i \in [1, r]$. There exists a node v, such that $v = v_{k,i}$, and v is in a k-core, which includes v and all the other nodes u with $w[u] > w[v]$.*

Based on Lemmas 2 and 3 and the fact that we only produce the top-r results, we can state the following theorem.

Theorem 8. *Algorithm 9 correctly computes all and only the top-r influential communities.*

The time complexity of C3 is quadratic from a worst case perspective. This is because we call *updateCores* for each resurrection. However, we have degree conditions in *updateCores* to only look for updates if the resurrected node is well connected to the other resurrected nodes, thus reducing the number of updates significantly. In practice, C3 can be much faster than C2 for moderate r and big graphs.

3.3.2 Algorithm NC2

For non-containing communities, we can also construct a backward approach. Let us recall the forward approach of [24] for non-containing communities. In order to determine whether $H_{k,i-1}$ is non-containing, the algorithm checks if all the nodes of $H_{k,i-1}$ are deleted in the next iteration i.

For the backward approach, we will use the same idea, but in a different way. In a nutshell, when we resurrect a node v, and it happens to be a minimum weight node in a k-core, we compute the corresponding community, say H; then we check to see whether any element of H participates in any community discovered earlier. If not, H is non-containing.

Our backward algorithm, NC2, is given in Algorithm 10.

In order to achieve maximum efficiency (which is crucial, especially for big graphs), we opt for a boolean array, *inPC* (**in**-a-**P**reviously-discovered-**C**ommunity) is used to record the nodes that participate in some community discovered earlier.

Using a boolean array makes the complexity of checking whether a node participates in a previously discovered community takes constant time (in contrast, a hash-based set would only give constant time on average[1]). We handle the population of *inPC* and the membership check of nodes in it in a modified MCC procedure (Algorithm 11).

3.3.3 Core Update upon Node Resurrection

The *updateCores* procedure needed by Algorithms 9 and 10 comes with its own set of challenges. We have two options: either use an incremental core update algorithm, such as the

[1]As argued in [22], using hash-based sets for the nodes of big graphs (or subsets of them) gives a rather unsatisfactory performance.

Algorithm 10 Top-r non-containing communities (NC2)

Input: G, w, k, r

Output: Top-r non-containing $H_{k,j_{\max}-r+1}$, ..., $H_{k,j_{\max}}$

```
1: for all v ∈ V do
2:     alive[v] ← false
3:     inPC[v] ← false
4:     cores[v] ← 0
5: i ← 1
6: for j = n downto 1 do
7:     Let v be a maximum-weight deleted node in V
8:     alive[v] ← true
9:     updateCores()
10:    if cores[v] ≥ k then
11:        isNC ← true
12:        H ← MCC(G, v, cores, isNC)
13:        if isNC = true then
14:            Output H
15:            i ← i + 1
16:            if i > r then
17:                break
```

Algorithm 11 MCC with *alive* and *inPC* arrays

```
1: procedure MCC(G, v, alive, inPC, isNC)
2:     cc ← ∅
3:     MCC-DFS(G, v, alive, cc, inPC, isNC)
4:     return cc
5: procedure MCC-DFS(G, v, alive, cc, inPC, isNC)
6:     cc.add(v)
7:     if inPC[v] = true then
8:         isNC ← false
9:     else
10:        inPC[v] ← true
11:    for all u ∈ N_G(v) do
12:        if cores[u] ≥ k and u ∉ cc then
13:            MCC-DFS(G, u, alive, cc, inPC, isNC)
```

one proposed in [25] or recompute the cores using the Batagelj and Zaversnik (BZ) algorithm [2]. We implemented both and compared them. The incremental core update of [25] considers the addition of each edge separately. Hence, the addition of a node triggers a sequence of core updates, one for each edge coming from the added node.

Compared to the re-computation of cores using the procedure of [2], the procedure of [25] was faster for small to moderate graphs and for small r's, but as the graphs and r grow in size, the re-computation of cores by the procedure of [2] is faster. In our case, we have many node resurrections, and it turned out that re-computing the cores using the BZ algorithm was faster (see Section 3.4).

3.3.4 Modified BZ Algorithm

If the current degree of v (in the subgraph induced by the alive nodes) happens to be greater or equal to k, we call *ModBZ* (Algorithm 12). In order to use the Batagelj and Zaversnik (BZ) algorithm [2], we need to properly adapt it so that it remains fast in spite of changing graph parameters (which is the case as we incrementally resurrect nodes). In the following, we give some details about the BZ algorithm and then describe our adaptations.

At a high level, BZ computes the core decomposition by recursively deleting the node with the lowest degree. The deletions are not physically done on the graph; an array is used to capture (logical) deletions. The notion of "deleted nodes" in core computations is different from that considered at the start of the backward algorithms, and as such, it is recorded and handled differently. For the BZ algorithm, to achieve high performance, everything needs to be implemented as flat arrays so that each logical deletion costs (precisely) constant time. As shown in [22], using hash-based structures makes the algorithm take orders of magnitude longer to complete. There are several arrays needed for the modified BZ algorithm (ModBZ, Algorithm 12). They are as follows.

1. The array *degrees* records the degree of each node considering only alive nodes. This array is global and with a dimension of n, where n is the number of all nodes, alive or not.

2. The array *cores* records at any given time for any alive node v the degree of v considering only the alive, and not-yet-deleted by BZ, nodes. In the end, *cores* will contain the core

numbers of each node considering only alive nodes. In *ModBZ*, we make this array global and with a dimension of n.

3. The array *vert* contains the alive nodes in ascending order of their degrees. We make this array local and with a dimension of *n_alive*, where *n_alive* is the number of alive nodes.

4. The array *pos* contains the indices of the nodes in *vert*, i.e. $pos[v]$ is the position of v in *vert*. We make this array local and with a dimension of *n_alive*.

5. The array *bin* stores the index boundaries of the node blocks having the same degree in *vert*. We make it local and with a dimension of *m_alive*, which is the greatest degree in the graph induced by the alive nodes.

In addition to the above arrays, we use two new arrays for *ModBZ*, *al* and *al_idx*. We make them global with a dimension of n. They are defined as follows.

6. The array *al* stores the alive nodes. When a node v is resurrected, we store v in $al[n\_alive]$ and increment *n_alive*.

7. The array *al_idx* contains the indices of the nodes in *al*, i.e. $al\_idx[v]$ is the position of v in *al*.

In line 2 of Algorithm 12, arrays *vert*, *pos*, and *bin* are initialized. The main logic is outlined in lines 3–16. The top **for** loop runs for each node, 0 to *n_alive*, scanning the array *vert*. We obtain a node id from the array *vert*, translate it to an id, v, in the normal $[0, n]$ range, and check whether it is alive. We only continue the computation if v is alive. Since the array *vert* contains the alive nodes in ascending order of their degrees, and v is the not-yet deleted node of the lowest degree, the coreness of v is its current degree considering only the alive, and not-yet-deleted by the procedure *ModBZ*, nodes, i.e. $cores[v]$.

After logical deletion of v, we process each neighbour u of v with $cores[u] > cores[v]$ (line 8). Node u current degree, $cores[u]$, is decremented (line 16). However before that, u is moved to the block on the left in the array *vert* since its degree will be one less. This is achieved in constant time (lines 9-15).

These operations are made possible by the existence of the array *al_idx*, which translates node ids to the $[0, n\_alive]$ range required by the local arrays. Specifically, u is swapped with the first node, w, in the same block in the array *vert*. Also, the positions of u and w are swapped in the array *pos*. Then, the block index in the array *bin* is updated incrementing it

Algorithm 12 Modified BZ algorithm (ModBZ)

```
1: procedure ModBZ(G)
2:     initialize(vert, pos, bin, cores, G)
3:     for all i ← 0 to n_alive do
4:         v ← al[vert[i]]
5:         if v not alive then
6:             continue
7:         for all alive u ∈ N_G(v) do
8:             if cores[u] > cores[v] then
9:                 du ← cores[u], pu ← pos[al_idx[u]]
10:                pw ← bin[du], w ← al[vert[pw]]
11:                if u ≠ w then
12:                    pos[al_idx[u]] ← pw
13:                    vert[pu] ← al_idx[w]
14:                    pos[al_idx[w]] ← pu
15:                    vert[pw] ← al_idx[u]
16:                bin[du]++, cores[u]−−
17: procedure initialize(vert, pos, bin, G)
18:     for all v ← 1 to n_alive do
19:         cores[al[v]] = degrees[al[v]], bin[cores[al[v]]]++
20:     start ← 0
21:     for all d ← 0 to md_alive do
22:         num ← bin[d], bin[d] ← start
23:         start ← start + num
24:     for all v ← 0 to n_alive do
25:         pos[v] ← bin[cores[al[v]]]
26:         vert[pos[v]] ← v
27:         bin[cores[al[v]]]++
28:     for all d ← md_alive downto 1 do
29:         bin[d] ← bin[d − 1]
30:     bin[0] ← 0
```

Algorithm 13 Core update using *ModBZ*

```
1: procedure updateCores(G, v)
2:     al[n_alive] ← v, al_idx[u] = n_alive, n_alive++
3:     updateDegrees(v)
4:     if degrees[v] ≥ k then
5:         ModBZ()
6: procedure updateDegrees(G, v)
7:     for all alive u ∈ N_G(v) do
8:         degrees[v]++, degrees[u]++
9:         md_alive ← max{md_alive, degrees[v], degrees[u]}
```

by one (line 16), thus losing the first element of the block, u, which becomes the last element of the previous block.

The procedure *ModBZ* is invoked by the procedure *updateCores* (Algorithm 13). The latter starts by recording the resurrected node v in the array *al* and updating the arrays *al_idx* and *n_alive*. Next, it calls the procedure *updateDegrees* to update the degrees (in array *degrees*) of alive nodes affected by v's resurrection. The procedure *updateDegrees* also updates the value of the maximal degree of alive nodes, *md_alive*, on the fly, so that it is ready for the procedure *ModBZ* to use.

3.4 Experimental Results

We performed our analysis by extensive experiments on several real-world graphs. The experiments were divided into two parts: first, we evaluated the performance of different algorithms and eliminated from consideration those that were slower than the others by a large degree; and second, we conducted a broad testing of the remaining algorithms, with the goal of finding the best solutions for extracting the most influential communities.

We implemented all the algorithms in Java and used Webgraph [5] as a graph compression framework. We chose Webgraph because of excellent compression ratios it achieves and also because it is actively maintained and continuously improved (`http://webgraph.di.unimi.it`).

Dataset	n	m	$d_{\max}$	$k_{\max}$
AstroPhysics	133.2 K	396 K	504	56
LiveJournal	4.8 M	43 M	20,333	372
UK2002	18.4 M	262 M	194,955	943
Arabic2005	22.7 M	640 M	575,628	3,247
UK2005	39.4 M	936 M	1,776,858	588
Webbase2010	118 M	1,020 M	816,127	1,506
Twitter2010	41.7 M	2,405 M	2,997,487	2,488
Clueweb	978.4 M	74,744 M	75,611,696	4,244

Table 3.2: Datasets ordered by m. The two last columns give the maximum degree and maximum core number.

Param.	**Range**
k	2, 4, 8, 16, 32, 64, 128, 256, 512
r	10, 20, 40, 80, 160, 320

Table 3.3: Parameters k and r, and their ranges.

Webgraph decompresses on the fly only the part of the graph being accessed. The decompression is very fast; we did not observe any noticeable delay compared to an uncompressed hash map graph representation (when the latter could fit in memory). This is because the footprint of a Webgraph representation is much smaller than a hash map representation allowing for better cache utilization.

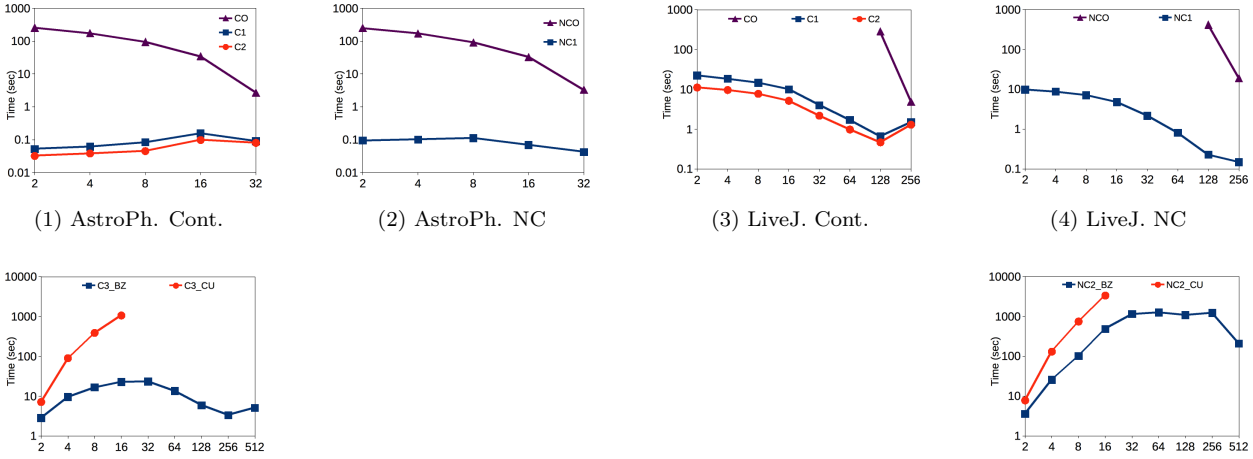
Datasets. The graphs we used were obtained from `http://law.di.unimi.it/datasets.php`. They vary from medium to massive sizes (Table 3.2). Each directed graph was converted to an undirected one by adding for each edge (u, v), its inverse (v, u), if it did not exist; also, self-loops (*e.g.*, (v, v)) were eliminated. The edge numbers in Table 3.2 refer to this version of the graphs. We assigned random weights to the nodes of each graph. The total order of weights was strictly enforced.

For all the datasets, but Clueweb, a timeout of one hour was set for each algorithm to run. For Clueweb, the timeout was set to 24 hours (albeit we did not need more than a few hours for most of the algorithm runs).

Equipment. The results for the first seven datasets in Table 1 (from AstroPhysics to

Twitter2010) were obtained on a consumer-grade laptop: processor 3.4GHz Intel Core i7 (4-core), 16GB RAM, running OS X Yosemite.

Clueweb could not be tested on our laptop because the compressed graph uses about 26 GB of space and would not fit into memory.[2] Clueweb was tested on a machine with 2.10 GHz Intel(R) Xeon(R) E5-2620 v2 (6-core) CPU and 64 GB RAM, running Ubuntu Server 14.04.3 LTS. Note that the processor speed is lower than that of our laptop, but the memory is larger. Despite its big memory, the machine had a price of about $3K, qualifying it as a consumer-grade machine.



(1) AstroPh. Cont. (2) AstroPh. NC (3) LiveJ. Cont. (4) LiveJ. NC

(5) UK2002 Cont. (6) UK2002 NC

Figure 3.3: Original and proposed algorithms on AstroPh. and LiveJ. when varying k ($r = 40$), first row. BZ versus CU, second row.

3.4.1 Testing Original Algorithms

We started by comparing the direct algorithms of [24], C-original (CO) and NC-original (NCO), with our counterparts, C1, C2, and NC1. We were only able to obtain results for CO and NCO using the first two (smaller) datasets, AstroPhysics and LiveJournal. Figure 3.3 shows the results of the comparison: (1), (2) are for AstroPhysics, and (3), (4) are for LiveJournal. C1 and C2 outperform CO, and NC1 outperforms NCO, in both cases by orders of magnitude. For LiveJournal, CO and NCO were only able to produce results for $k = 128$ and $k = 256$ (for these values, C_k was small enough for them to handle). Thus, we

[2] We refer to the version of the graph described above.

Name	Description	Core Update
C3_CU	Algorithm 9	Incremental
NC2_CU	Algorithm 10	Incremental
C3_BZ	Algorithm 9, Algorithm 13	Recomputing
NC2_BZ	Algorithm 10, Algorithm 13	Recomputing

Table 3.4: Backwards Algorithms.

Name	Description	Problem
C1	Algorithm 3	1
C2	Algorithm 4	1
C3_BZ	Algorithm 9, Algorithm 12	1
NC1	Algorithm 7	2
NC2_BZ	Algorithm 10, Algorithm 12	2

Table 3.5: Proposed Algorithms.

eliminated CO and NCO from further testing; they could not produce results on large-scale graphs within a reasonable amount of time.

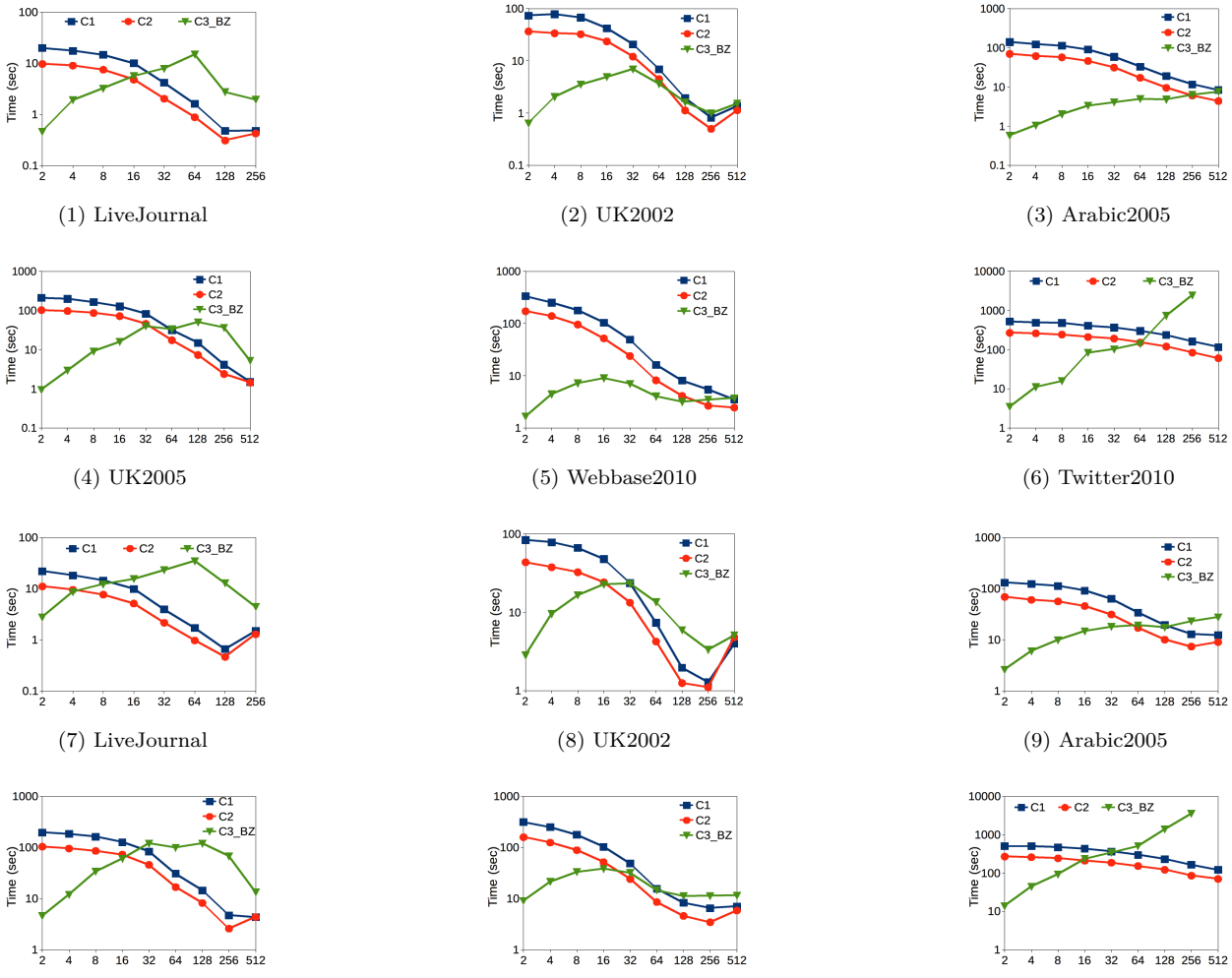
Testing Core Updates. The *updateCores* procedure (Section 3.3) used in the backward algorithms was implemented using two different approaches: the incremental core update (CU) algorithm proposed by Li, Yu, and Mao in [25], and the modified BZ algorithm, ModBZ (Algorithm 12). The first approach was implemented in C3_CU and NC2_CU, and the second in C3_BZ and NC2_BZ. (Table 3.4).

These algorithms were tested extensively on several smaller and medium size graphs. Here, we are presenting the results on UK 2002, which was the largest dataset for which the computation completed (at least for several k values) for C3_CU and NC2_CU. Figure 3.3.5 and 3.3.6 show that C3_CU and NC2_CU were slower than C3_BZ and NC2_BZ, respectively. Incremental core updates were slower than the k-core re-computation.

After this analysis, we eliminated C3_CU and NC2_CU from further consideration; they could not be feasibly used for a community extraction on large-scale graphs in a reasonable amount of time.

3.4.2 Main Testing

The bulk of testing was done for the algorithms in Table 3.5. We start by presenting test results and analysis for LiveJournal, UK 2002, Arabic 2005, UK 2005, Webbase 2010, and Twitter 2010. We omit results on AstroPhysics as this dataset is relatively small. The testing results of the largest dataset, Clueweb, are presented separately.



(1) LiveJournal (2) UK2002 (3) Arabic2005

(4) UK2005 (5) Webbase2010 (6) Twitter2010

(7) LiveJournal (8) UK2002 (9) Arabic2005

(10) UK2005 (11) Webbase2010 (12) Twitter2010

Figure 3.4: Containing Communities: Performance when varying k (first two rows: $r = 10$, last two rows: $r = 40$).

Problem 1: Computing containing communities.

Figure 3.4 and 3.5 show results for computing containing communities when k and r are varied, respectively.

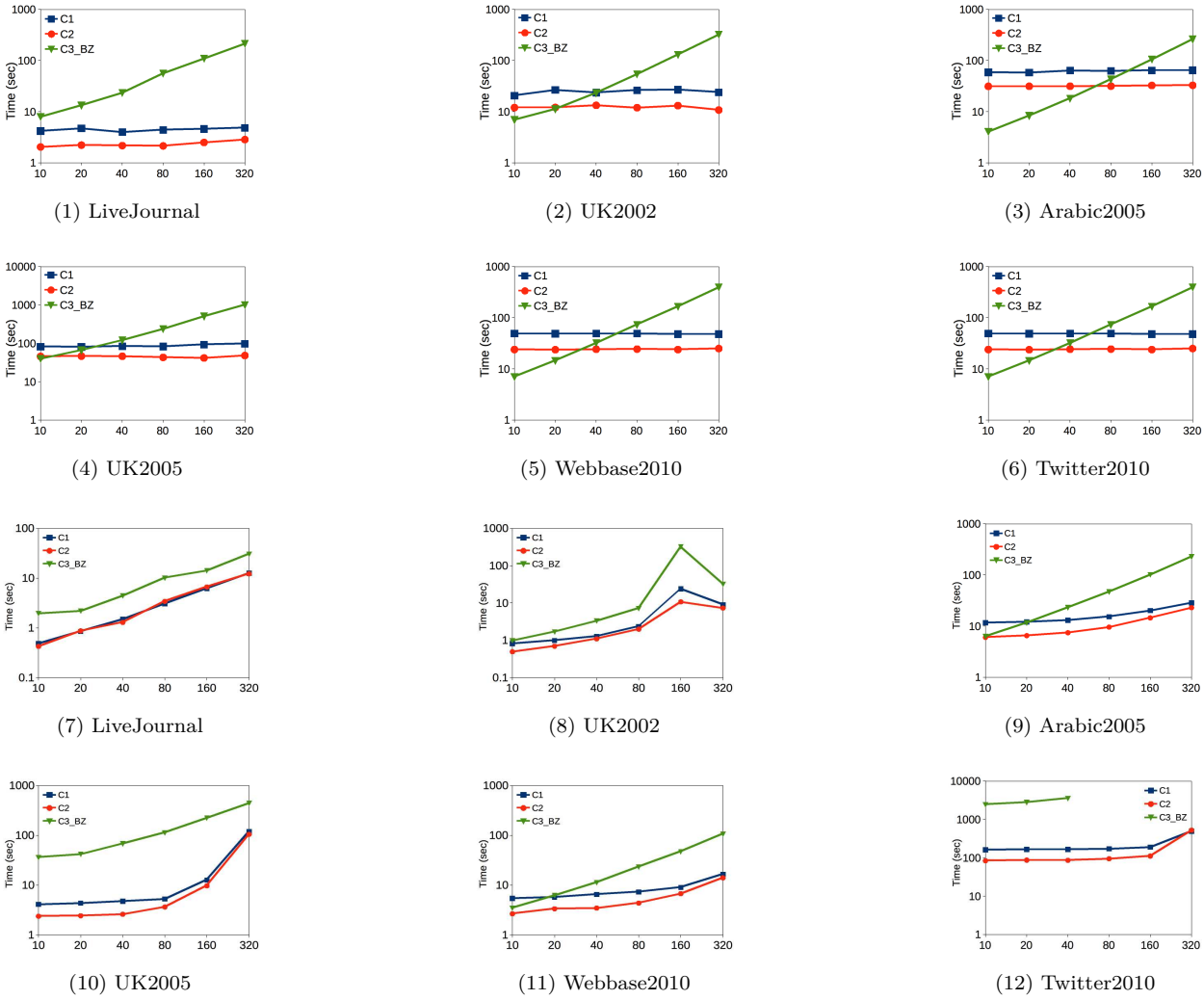


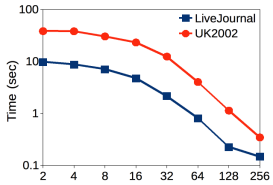
Figure 3.5: Containing Communities: Performance when varying r (first two rows: $k = 32$, last two rows: $k = 256$).

Figure 3.4 shows the performance when k is varied. The value of r is 10 for 3.4.1–3.4.6, and 40 for 3.4.7–3.4.12.

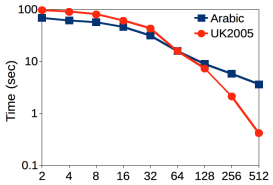
Figure 3.5 shows the performance when r is varied. The value of k is 32 for 3.5.1–3.5.6, and 256 for 3.5.7–3.5.12.

Analysis of results. (1) The charts clearly show that C2 outperforms C1 for all values of k and r tested. This is expected, because C2 makes only one pass over the graph as opposed to two that C1 does (Algorithm 3 and 4).

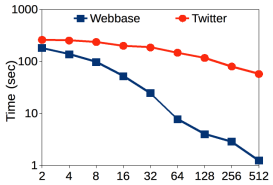
(2) The run times of C1 and C2 decrease as the k-core subgraph, C_k, becomes smaller with



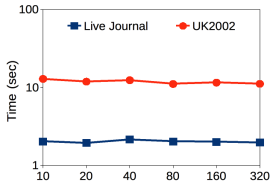
(1) $r = 40$



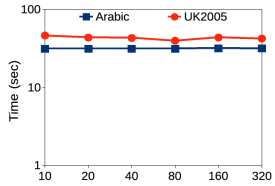
(2) $r = 40$



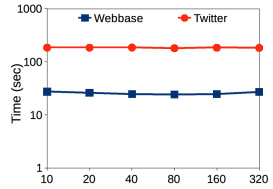
(3) $r = 40$



(4) $k = 32$

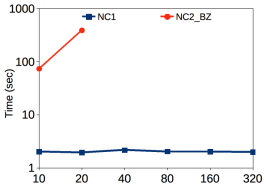


(5) $k = 32$

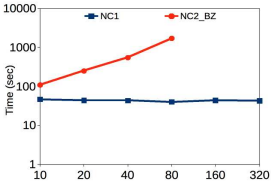


(6) $k = 32$

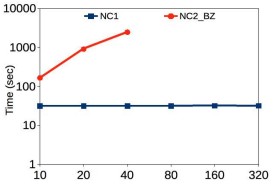
Figure 3.6: Non-Containing Communities: Performance when varying k and r.



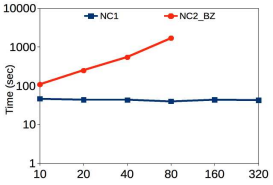
(1) LiveJournal



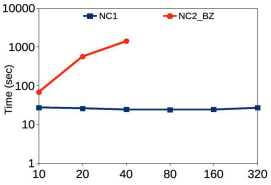
(2) UK2002



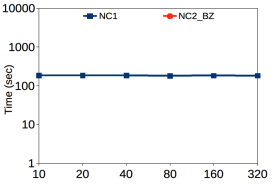
(3) Arabic2005



(4) UK2005



(5) Webbase2010



(6) Twitter2010

Figure 3.7: Non-Containing Communities: Performance when varying r ($k = 32$).

the increase of k (Figure 3.4).

For a fixed k, the runtime becomes greater as the number of top communities to compute, r, grows (Figure 3.5).

(3) The runtime of C1 and C2 depends also on the graph structure. For an example, let us consider Figure 3.4.2 and 3.4.8.

The run times for both C1 and C2 go down for $k = 256$, but then suddenly go up for $k = 512$. The C_k sizes for $k = 256$ and $k = 512$ are 18,179 and 2,951 nodes, respectively. So, to retrieve the same number of communities from a much smaller graph takes longer! To

find out why, we conducted a further analysis to explain this surprising result.

Let us focus on UK 2002. We conducted multiple tests with different values of k, up to and including $k_{\max}$, which is 943 for UK 2002. We analyzed the nodes that the influential communities were constructed from. The tests show that (a) all influential communities for $k = 256$ are retrieved from the same big ($k = 943$) clique by eliminating just one, least influential, node after another. This takes very little time, and the runtime for $k = 256$ becomes quite low as we can see in the chart; (b) for $k = 512$, there were not enough influential nodes in a similar clique, so some influential communities were retrieved from a different cluster with high cohesiveness and relatively high importance/influence. This "switch" to a different area of the graph takes more time than staying steady (due to a loss of locality) because we have to decompress a different portion of graph G, and the run time of algorithms became somewhat larger. On the charts, it is seen as an increase of the run time.

It must be noted, however, that the effect described above is noticible only on the charts for the smaller graphs, with run times of 1 to 2 seconds. For bigger graphs, the fluctuations in importance (weights) distribution and their interplay with the graph structure does not influence the running time noticeably. This happens because the overhead of switching to a different part of C_k is absorbed by the time needed for the rest of the computation. This in turn means that the 'hills and valleys' on Figure 3.4 and 3.5 for the smaller graphs, LiveJournal and UK 2002, are of no great significance from a practical point of view.

Let us now focus on the performance of the C3_BZ algorithm. We make the following observations: (1) For small r's and k's, C3_BZ is a good choice; this is because we only need to perform few core re-computations (small r) and most of these re-computations are not wasted, i.e. we are able to find new members of k-core quite often when we resurrect nodes (small k). (2) The bigger the graph, the better the chance that using C3_BZ will give better performance for small to moderate r's and k's; this will be significantly more pronounced for Clueweb (presented later in this section).

Finally, let us consider the performance of C1, C2, and C3_BZ as r varies in Figure 3.5. We see that the runtime of C1 and C2 is pretty much constant. This happens because most of their time is spent on peeling off the graph until no node remains; this is the same

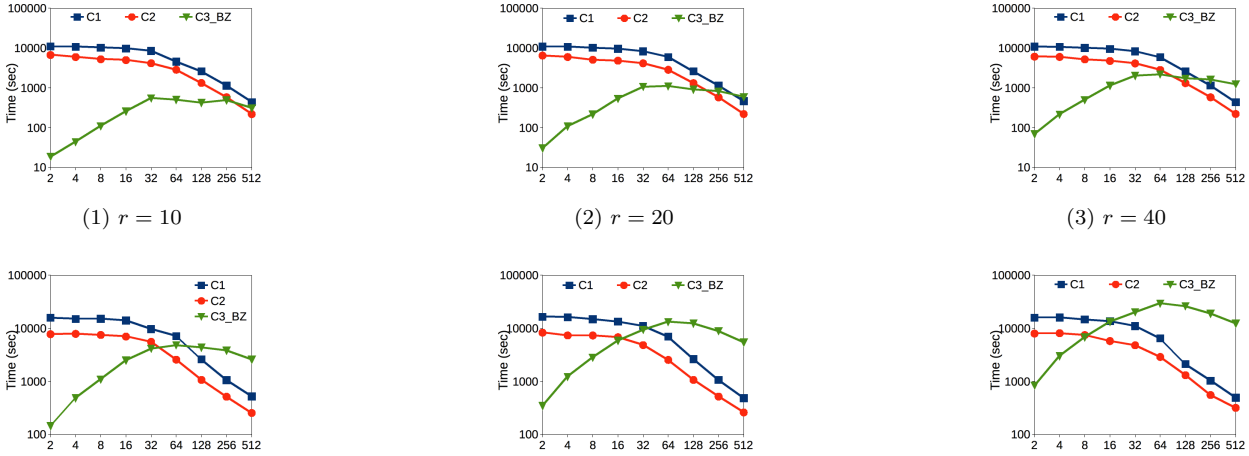
regardless of r. The value of r determines the number of MCC runs in C1 and C2; but these MCC runs become negligible in C1 and C2 as they are only performed in the end, after most of the nodes of the graph have been deleted. C3_BZ is sensitive to r. As expected, the greater the value of r, the longer C3_BZ takes to complete.

For Arabic 2005, Webbase 2010, and Twitter 2010, C3_BZ can compute the top communities for small values of r more quickly that the C1 and C2. The better performance of C3_BZ for small r becomes significantly more pronounced for Clueweb, which is an order of magnitude bigger than the datasets in Figure 3.5 (described later in this section).

Problem 2: Computing non-containing communities.

Figure 3.6 shows results for non-containing communities; (1)–(3) show the performance of NC1 for varying k on different datasets, and (4)–(7) show the performance of NC1 for varying r on the same datasets. We see that the runtime of NC1 quickly goes down as k increases, (1)–(3). Also, NC1 continues to not be sensitive to r, (4)–(6). The performance of NC2_BZ (backward approach) was not competitive for these six datasets (not shown in the interest of clarity). Nevertheless, for very large graphs, such as Clueweb, as we show later, NC2_BZ is more useful.

3.4.3 Experiments on Clueweb



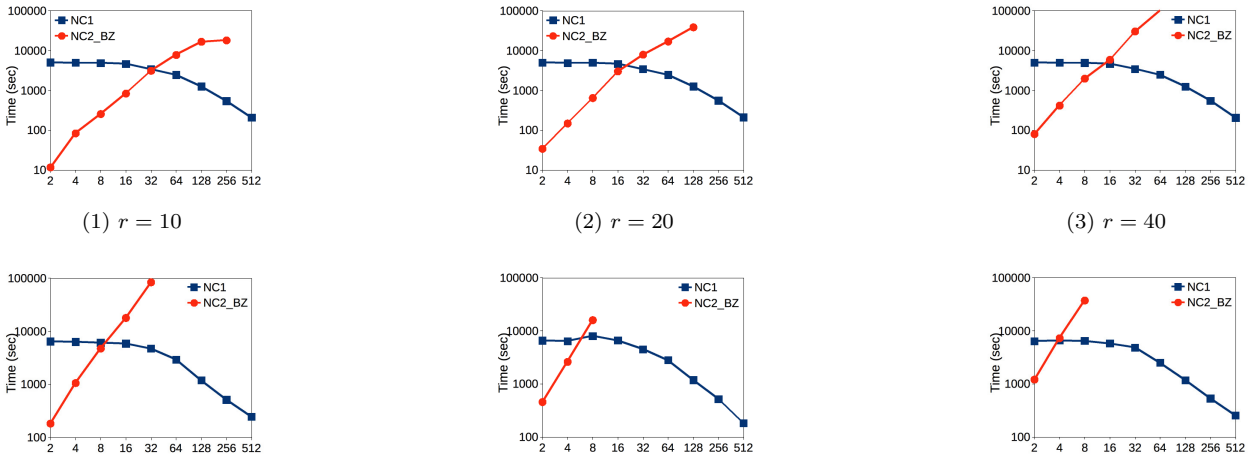
(1) $r = 10$ (2) $r = 20$ (3) $r = 40$

(4) $r = 80$ (5) $r = 160$ (6) $r = 320$

Figure 3.8: Clueweb: Containing Communities. Performance when varying k.

A special case for testing the proposed algorithms was using the Clueweb dataset. This

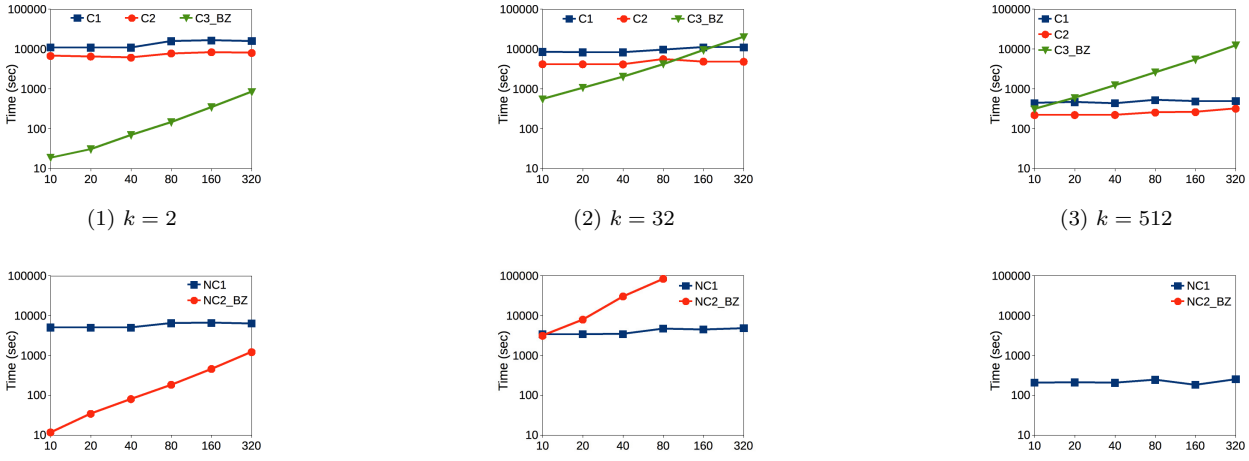
very large dataset is the only one that could not be tested on our laptop: even compressed, it takes 26 GB of memory. Clueweb was tested on a bigger computer with 2.10GHz Intel(R) Xeon(R) E5-2620 v2 (6-core) CPU and 64GB RAM. Note that the processor speed is lower than that of our laptop, but the memory is larger. The machine can still be considered consumer-grade as its price was within a range of $3K. For Clueweb we used a timeout of 24 hrs, albeit we often did not need such a large timeout.



(1) $r = 10$ (2) $r = 20$ (3) $r = 40$

(4) $r = 80$ (5) $r = 160$ (6) $r = 320$

Figure 3.9: Clueweb: Non-Containing Communities. Performance when varying k.



(1) $k = 2$ (2) $k = 32$ (3) $k = 512$

(4) $k = 2$ (5) $k = 32$ (6) $k = 512$

Figure 3.10: Clueweb: Performance when varying r. (a), (b), and (c) - Cont. Communities; (d), (e), and (f) - NC Communities.

Figure 3.8 and 3.9 show in detail the performance of the algorithms when varying k. It is clear that the backward approaches implemented in C3_BZ and NC2_BZ are to be used for

extracting the top communities for several combinations of k and r (*e.g.*, $k = 2, \ldots, 256$ and $r = 10$ for C3_BZ, or $k = 2, \ldots, 32$ and $r = 10$ for NC2_BZ). In several such cases, C3_BZ and NC2_BZ proved to have orders of magnitude better performance than the forward algorithms. Clueweb testing confirmed the trend noticed before: the bigger the graph, the more beneficial it is to use C3_BZ and NC2_BZ. With k and r increasing, these algorithms keep the better performance on large graphs longer than on the smaller graphs.

On the other hand, if the application is to extract many communities, (roughly, $k > 32$ and $r > 40$), the forward algorithms C1, C2 and NC1 are recommended. For example, in Figure 3.10.6, when computing non-containing communities, it is only NC1 that is able to produce results within 24 hours.

In a nutshell, the results show that we are able to compute containing and non-containing communities for every combination of k and r on Clueweb using the forward algorithms. We can do that faster for a good number of k and r combinations using the backward algorithms. Being able to scale to Clueweb is a significant contribution because this dataset is an order of magnitude bigger than the second large dataset we consider, Twitter 2010, as well as the datasets tested in [24].

In 2018, Bi, Chang, Lin, and Zhang [3] presented a new algorithm, LocalSearch, to efficiently compute top influential communities. Research into the problem of influential communities discovery is continuing.

Chapter 4

Problem Statement for Influence Maximization

All our algorithms for the influence maximization and influence estimation are written for directed graphs, and the following definitions are applicable to the directed graphs.
If a directed edge goes from node u to node v, it is denoted as (u, v). The directed edge (v, u) is called the *inverted* edge of (u, v).
A directed graph with all its edges inverted is called a *transpose* graph of the original (before inversion) graph.
A directed graph *path* is a sequence of distinct directed edges that leads from a node to another node via a sequence of distinct nodes.
If a path is leading from node v to node y, y is *reachable* from v.
If an edge connects two nodes, the edge head is called a *neighbour* of the edge tail. A node *neighbourhood* consists of all its neighbours.

4.1 Notation

Let $G = (V, E, p)$ be a directed probabilistic graph, where node set $|V| = n$, edge set $|E| = m$, and $p : E \to [0, 1]$ is a probability function on edge existence. In this dissertation, we consider the case where p is constant, i.e., for some constant number $0 < c \leq 1$, it holds that $p(e) = c$ for all $e \in E$.

4.2 Independent Cascade

Kempe *et al.* [21] formalized the Independent Cascade (IC) model proposed in [15]. The IC model proceeds as follows. Starting from a set $S \subseteq V$ of initial nodes, influence spreads in discrete rounds/steps following a randomized rule. When a node u gets influenced for the first time at a step t, it is given a single chance to influence each currently uninfluenced node v among its neighbours. It will succeed with an independent random probability p_{uv}. If u succeeds, then v will become influenced in the step $t+1$. Whether or not u succeeds, it cannot make any further attempts to influence its neighbours in subsequent steps: the edge (u, v) was "touched" and cannot be tried for the influence propagation from u to v again. Influenced neighbours of u, in their turn, have one chance to influence their uninfluenced neighbours forming a *cascade* of influence propagation. Hence, the name – Independent Cascade. The influence propagation continues until no further influence spread is possible, because all the edges incident to the influenced nodes are already touched.

In our algorithms, each random trial consists of generating an independent random number x in the range $[0, 1]$, and comparing x with the given probability p of the edge. The probability reflects the influence of node u over its neighbours. For each edge from u to a neighbour v: if $x \leq p_{uv}$, the influence will spread to v, otherwise v is left uninfluenced.

Let *I(S)* be the (random) number of nodes that are eventually influenced by this process. Then we think of the expectation of *I(S)* as the influence of set S. The optimization problem is to find set S maximizing $E[I(S)]$ subject to $|S| \leq k$, where k is a given number of initial nodes (seeds). Kempe *et al.* [21] showed that $E[I(\cdot)]$ is a non-negative submodular monotone function, and hence the problem of maximizing $E[I(\cdot)]$ can be approximated to within a factor of $(1 - 1/e - \epsilon)$ for any $\epsilon > 0$, in polynomial time, via a Greedy hill-climbing method.

Definition 8. *The* influence spread *of a seed set S, denoted by $\sigma(S)$, is defined as the expected total number of reachable nodes for S.*

IC became a standard model of information diffusion, and we are using it for our algorithms.

Let us define the IM and IE problems formally.

4.3 IM and IE Problems

Problem 3 (Influence Estimation Problem (IE)). *Given a graph $G = (V, E, p)$ and a node set $S \subseteq V$, compute the influence spread $\sigma(S)$ of S.*

Problem 4 (Influence Maximization Problem (IM)). *Given a graph $G = (V, E, p)$ and an integer k, find a node set $S \subseteq V$ of size k that maximizes $\sigma(S)$.*

4.4 Greedy Method

A Greedy algorithm for IM starts with an empty seed set S. In each iteration, it adds to S a new seed such that this node together with the current set S maximizes the influence spread. How can the influence spread be calculated? We do not have an oracle access to $E[I(\cdot)]$, but influence spread could be estimated in the following way: for each node $v \in V \setminus S$, the influence spread of $S \cup v$ is estimated by repeated simulations of influence propagation in the graph. The simulation is usually repeated 10,000 – 20,000 times, and an average of the information spreads is computed. Running a random (with a given probability for each edge existence) process of influence propagation on a graph for each node is costly: the Greedy algorithm takes $O(knRm)$ time to complete, where k is the number of seeds to calculate, n and m are the number of nodes and edges in the graph, and R is the number of repetitions.

In 2009, Chen *et al.* [10] offered an improved Greedy method that would allow a sufficiently good estimate of information spread in much less time: instead of running a random influence propagation for each node separately, they suggested calculating the influence spread for all the nodes $v \in V \setminus S$ at once in a random graph. The random graph is created by eliminating the edges of the original graph with probability $1 - p$. This algorithm takes $O(kRm)$ time to complete.

4.5 Previous Work

Building on the results of Kempe *et al.*, substantial research has been done on developing an *approximation* algorithm for IM [21, 10, 13, 11] and IE [26, 29]. Each of these research

efforts advanced our understanding of the underlining mathematical limitations of Kempe approach.

In recent years, a number of IM computing algorithms have been developed, both heuristic [20, 10, 9] and with theoretical guarantee [16, 31, 43, 42, 32, 30]. However, in spite of successful speed-up techniques, modern massive networks require even faster algorithms; moreover, the required main memory is so large that to successfully run these algorithms, very large and expensive computers are needed.

Arora *et al.* demonstrated in [1] that none of the existing algorithms satisfies the triad: quality of spread, runtime efficiency, and low memory footprint. Arora *et al.* tested eleven most prominent and broadly recognized algorithms on a specifically designed benchmarking framework. To ensure test fairness, Arora *et al.* IM framework measures runtime, required memory, and solution accuracy for each algorithm with the same parameters on the same graphs, using the same machines. Tests results were combined into a "decision tree" that would guide the consumers to the most appropriate algorithm for their requirements. Arora *et al.* [1] testing showed that there are algorithms satisfying two of the triad: quality and runtime efficiency, or runtime and low memory footprint, or quality and low memory footprint, but not all three requirements. Note, that Arora *et al.* [1] tested only the algorithms published before May 2016.

In 2013, a different approach was proposed by Borgs *et al.* [7]: Reverse Influence Sampling (RIS) method[1]. The idea is to select a node v uniformly at random, and determine the set of nodes that *would have influenced* v. If a certain node u appears often as influential for different randomly selected nodes, then u is a good candidate for the most influential node. RIS is a fast algorithm for IM, obtaining the approximation factor of $(1 - 1/e - \epsilon)$, for any $\epsilon > 0$, in time $O((m + n)k\epsilon^{-2}\log(n))$, where n is the number of nodes, m is the number of edges, and k is the number of seeds [7]. But, just as for all other IM algorithms, the required main memory for keeping the sampling results is so large that to successfully run RIS, expensive computers with vast amount of main memory are needed even for the medium-size graphs.

Existing state-of-the-art approximation algorithms for IM and IE problems are not scal-

[1]The latest version 5 of the paper, issued on the 22nd of June 2016, can be found at `https://arxiv.org/pdf/1212.0884.pdf`

able to massive real-life graphs with billions of edges: if we want a guaranteed approximation to the optimal solution, the time and space complexity is such that no computer could calculate a set of seeds or estimate the given set of seeds influence without running out of memory. The goal of the ongoing research is to find a new effective way of resolving IM and IE problems by developing algorithms allowing to scale up the guaranteed approximation solutions.

Most of research teams employing the RIS method (for example, [16, 43, 42, 30, 18, 29]) pay particular attention to runtime efficiency, designing sophisticated algorithms aimed at cutting the number of random samples on the graph and therefore the runtime. We approached the problem of IM scalability from a different angle: We researched different data structures used for keeping the intermediate results of IM computing. Our main goal was to cut the memory footprint, making it possible to run large graphs on consumer-grade machines. That is, we are specifically dealing with *space complexity*.

Our algorithms can use any theoretically proven bound on the number of samples, with the same effect of drastically lowering the memory footprint needed for keeping the intermediate results of sampling. While any theoretically guaranteed formula for the number of samples can be used by our algorithms, throughout our research we have been using Borgs *et al.* formulae from [7, Theorem 3.1], or [7, Theorem 4.1]. This is done for two reasons: (1) the formulae guarantee an approximation to optimal solution that could be explicitly calculated, and (2) for consistency, when evaluating the scalability of our algorithms. Detailed description and an analysis of RIS method follow.

4.6 Reverse Influence Sampling (RIS) method

RIS was proposed by Borgs *et al.* [7][2] in 2013. The idea in RIS is to select a node v uniformly at random, and determine the set of nodes that *would have influenced* v. This can be done by simulating the influence spread using the IC model in the graph with the directions of edges reversed ("transposition of the graph", as it is called in [7]). The random selection of a node is repeated many times, with replacement (that is, a node, say v, can be selected more

[2]The latest version 5 of the paper, issued on the 22nd of June 2016, can be found at `https://arxiv.org/pdf/1212.0884.pdf`

than once). If a certain node u appears often as influential for different randomly selected nodes, then u is a good candidate for the most influential node.

4.6.1 Hypergraph Building.

To find the "influencers", Borgs *et al.* [7] propose to repeatedly run the graph search on the *transpose* (with the directions of edges reversed) graph. This is a randomized approach: the probability of each edge existence is defined by a given probability function p. The resulting list of nodes reached by a search via existing edges is called a *sketch*. Sketches are numbered, and the number of a sketch is assigned to it as its ID. As a result of multiple searches starting from randomly selected nodes, RIS creates a structure called a *hypergraph*, H. H stores the information derived from the sketches. H is consequently used for computing the seeds.

In order to determine the time at which we stop running searches, we calculate the target *weight* of the hypergraph, before starting the sampling. Borgs *et al.* [7] denoted the target weight by R, and we use this notation. The calculation of R is using graph characteristics (*e.g.*, number of nodes and edges) and given value of the allowed error. The current weight of the hypergraph, H_weight, is calculated after each sample and compared with the target weight R. H_weight is calculated as the number of edges "touched" during the graph searches.

Each simulation of information spreading through the transpose graph starting from a random node v, "touches" all the outgoing edges incident to v: each edge, with the given probability of its existence p, is marked as existing; with the probability $1 - p$, the edge is marked as non-existing. The search follows the existing edges to the neighbouring nodes; for each reached node, the process of "touching" all its edges is repeated. The search completes when at some point, there are no more edges to follow: all the edges of the most recently reached nodes were "touched" and marked non-existing. The number of "touched" edges is added to the previously calculated H_weight. Note, that all the "touched" edges regardless of their existence or non-existence are counted in H_weight. In other words, H_weight is equal to the sum of the out-degree taken over each node in H that is built so far. The out-degree of a node in the transpose graph equals the in-degree of the same node in the

original graph.

The hypergraph H can be implemented as a list of lists: a list of graph nodes, where each node has the list of sketch IDs it participated in. In this case, the hypergraph looks like a graph represented by the adjacency list. In our research, we abuse the term and treat hypergraph as a "normal" graph, though keeping in mind that nodes and sketches are completely different entities.

4.6.2 Approximating IM and IE using Hypergraph.

We now explain how to use hypergraphs to approximately estimate the influence spread. Borgs *et al.* have proven the following observation.

Lemma 4. *[7, Observation 3.2] Any node set S intersects a (random) set of influencers with probability $\sigma(S)/n$. In particular, the probability of a node u appearance in a set of influencers is $\sigma(\{u\})/n$.*

Hence, we are able to obtain an unbiased estimator for the influence spread by computing the fraction of lists that overlaps with a given set of nodes (multiplied by n). It should be also noted that a simple application of Chernoff's bound tells us that for precision parameters $\epsilon, \delta \geq 0$ and a node set S, if we have at least $O(\epsilon^{-2} \ln \delta^{-1})$ sets of influencers, the estimator approximates the value of $\sigma(S)$ within an additive error of ϵn with a probability at least $1 - \delta$.

To find the set of seeds, the approximate Greedy algorithm is run on the hypergraph. The highest degree in the hypergraph, that is, the longest list of sketch IDs, defines the most influential node. After calculating a seed, to avoid overlapping of the spheres of influence, the seed sketch IDs are removed from the hypergraph, thus decreasing the hypergraph degree for each node that participated in the same sketches as the already found seed. The calculation of the most influential node repeats on the reduced hypergraph, until the number of found seeds equals the given parameter k.

Theoretically, RIS achieves a near-linear time complexity; Borgs *et al.* proved the following theorem establishing the guaranteed approximation to the optimal solution:

Theorem 9. *[7, Theorem 3.1] For any $\epsilon \in (0, 1)$, if we set $R = cmk\epsilon^{-2} \log(n)$, where*

$c = 4(1+\epsilon)(1+1/k)$, then RIS returns a set of seeds S with $\sigma(S) \geq (1-1/e-\epsilon)OPT$, where OPT is the optimal influence spread, with probability at least 3/5.

Moreover, RIS can be modified to allow an early termination: if it is terminated after $O(\beta(m+n)\log(n))$ steps for some $\beta < 1$ (which can depend on n), then it returns a solution with approximation factor $O(\beta)$:

Theorem 10. *[7, Theorem 4.1] For any $\beta < 1$, Algorithm 2 returns, with probability of at least 3/5, a node with expected influence at least $min\{1/4, \beta\}OPT$. Its runtime is $O(\beta(n+m)\log(n))$.*

The Borgs *et al.* [7] algorithm is randomized, and it succeeds with probability 3/5; this success probability can be improved up to $1 - 1/n$ with $\log(n)$ repetitions.

4.6.3 Practical Challenges of RIS

Despite the strong results in theory, RIS suffers from practical inefficiency.

Running Time. Though R is nearly linear to graph size and so is the time complexity, RIS requires a huge number of edges to consider because of rather large constants. For example, for a relatively small graph with 100K nodes and about 3M edges, RIS needs to "touch" over 150B edges for $k = 10, \epsilon = 0.1$, and many times more if we need more seeds or lower error. As a result, if we run RIS on modern social networks with millions of nodes and hundreds of millions of edges, it takes days to complete a single run, even for a powerful computer.

Space Consumption. We need to keep the hypergraph in main memory for the calculation of seeds. How big will it be? Note, that the hypergraph weight R defines only how many edges of the original graph RIS has to "touch", not how many of these edges RIS will follow. But the hypergraph size is defined by the number of edges RIS follows. Let us call this number "size of hypergraph", or size of H. Consider the sampling process: a node v is picked up at random; each outgoing edge of v is "touched". With probability p, an edge is selected to follow to a neighbouring node u, while with probability $(1-p)$, the edge is ignored. Since the probability p is usually picked up in the range 0.1 – 0.001 for social networks (trivalency model; see, for example, [20]), it is much more probable that an edge

will be ignored than followed. This means that size of $H << R$: With each sketch taken, the weight R will increase much faster than the number of edges in the hypergraph, H. This is good news: we do not need to keep R integers, which would exceed main memory capacity for most machines. Still, H is large: for a graph with 100K nodes and about 3M edges, the hypergraph took 106 GB (when $k = 10, \epsilon = 0.1$) in main memory. For larger graphs, the space needed for RIS makes it impossible to run on a consumer-grade machine.

We researched several data structures for storing the IM sampling results. We also designed and implemented several new algorithms for managing these data structures and calculating the seed set. The details are available in Chapter 5, 6, and 7.

Chapter 5

Influence Maximization (IM) Solutions

The Influence Maximization (IM) problem and a method of its solution, Reversed Influence Sampling (RIS), are defined in Section 4.3. In this chapter, we describe in detail RIS possible implementations. Using RIS, our data structures and algorithms drastically increase the scalability of the IM solution compared to the IM algorithms proposed by other researchers.

5.1 Using Array Data Structure

For an IM solution, we developed three different algorithms implementing the Reversed Influence Sampling (RIS) method, described in the previous chapter. Each algorithm uses a distinct data structure for storing the hypergraph. We compared the performance of different data structures on a consumer-grade laptop.

5.1.1 Two-Dimensional List (2DL)

We started with a straightforward implementation of RIS, where we use a two-dimensional list structure (list of lists) for storing the hypergraph. Algorithm 14 shows the pseudocode of this implementation.

For a better performance, we added the following improvements: 1. Using the Webgraph [5] format for the input inverse graph (saves space); 2. Using Java 8 parallel streams and lambda expressions (speeds up performance by executing several reachability procedures in parallel); 3. Applying the BitSet structure for marking deleted sketches (speeds up the marginal influence calculation); and 4. Utilizing Leskovec *et al. [23]* Lazy Greedy technique

Algorithm 14 2DL

Input: directed graph G with n nodes and m edges, coefficient β, number of seeds k

Output: seeds set $S \subseteq V$ of size k, spread $\sigma(S)$

1: $R \leftarrow \beta * m * k * log(n)$
2: $H \leftarrow BuildHypergraph(R)$
3: return $GetSeeds(H)$
4: **procedure** BUILDHYPERGRAPH(R)
5: **while** $H\_weight < R$ **do**
6: $v \leftarrow$ random node of G^T
7: $sketch \leftarrow$ reachable nodes in G^T starting from v
8: **for** each node $u \in sketch$ **do**
9: append $sketchID$ to u's list of sketches
10: $count[u] \leftarrow count[u] + 1$
11: return hypergraph H
12: **procedure** GETSEEDS(H)
13: $S \leftarrow \emptyset$, $\sigma(S) \leftarrow 0$
14: **for** $i = 1, ..., k$ **do**
15: seed $v_i \leftarrow argmax_v\{count[v]\}$
16: $S.insert(v_i)$
17: $\sigma(S) \leftarrow \sigma(S) + count[v_i]$
18: remove the sketches containing v_i
19: output S, $\sigma(S)$

(speeds up the seed calculation). Influence estimation is part of the seed calculation. With minor changes, the code provides the solution for IE problem, when a seed set is input.

We compared the runtime and space of 2DL and DIM, a state-of-the-art implementation of RIS by Ohsaka *et al.* [32]. For both algorithms, we used the same lower bound on the hypergraph weight from [7, Theorem 4.1]. Our 2DL implementation significantly outperforms DIM. Testing results are discussed in detail in Section 5.2.1.

5.1.2 Flat Arrays (FA)

Flat arrays is one of the simplest data structures in Java and other languages. In our implementation, it is a one-dimensional list of indexed integers. FA implementation modifies

the BuildHypergraph procedure of 2DL (Section 5.1.1). The pseudocode is shown in Algorithm 15. Seed calculation in FA implements the same logic as GetSeeds procedure in Algorithm 14.

To store the hypergraph, FA creates two flat, one-dimensional, arrays of integers: *sketches* and *nodes*. *sketches* stores the sketch ID, *nodes* stores the node IDs reached by this sketch. Arrays *sketches* and *nodes* are synchronized, so that knowing the index of a sketch, we can easily find the corresponding nodes, and *vice versa*. In the following figure we show an example of a *sketches* array (first row) and corresponding *nodes* array (second row):

0	0	0	0	1	1	2	2	3	4	4	5
0	1	2	3	2	3	1	3	2	2	3	0

In this example, sketch IDs and their corresponding node IDs are divided by double bars from other sketches: sketch 0 contains four nodes: 0, 1, 2, and 3; sketch 1 contains two nodes: 2 and 3; and so on. Note that sketches are listed in ascending order, and the corresponding nodes for each sketch are listed in ascending order as well. We use these features for speeding up the calculation of seeds. Testing FA on real-life graphs shows its better usage of space and faster performance than 2DL (Section 6.5.1).

Algorithm 15 FA

```
1:  procedure BuildHypergraph(R)
2:      initialize sketches and nodes arrays to -1
3:      while H_weight < R do
4:          v ← random node of G^T
5:          sketch ← reachable nodes in G^T starting from v
6:          for each node u ∈ sketch do
7:              sketches[i] ← sketchID
8:              nodes[i] ← u
9:              count[u] ← count[u] + 1
10:     return hypergraph H = (sketches, nodes)
```

Algorithm 16 CS-FA

```
1:  procedure BuildHypergraph(R)
2:      initialize nodes array to -1
3:      while H_weight < R do
4:          v ← random node of G^T
5:          sketch ← reachable nodes in G^T starting from v
6:          for each node u ∈ sketch do
7:              node_count ← node_count + 1
8:              add nodeID to array nodes
9:              count[u] ← count[u] + 1
10:         add node_count to array sketches
11:     return hypergraph H = (sketches, nodes)
```

5.1.3 Compressed Flat Arrays (CS-FA)

Here we present a more efficient implementation, the CS-FA algorithm (Algorithm 16) that creates the hypergraph similar to FA (Algorithm 15). Seed calculation in CS-FA implements the same logic as GetSeeds procedure in Algorithm 14. The main difference between CS-FA and FA is the design of the *sketches* array: CS-FA stores the accumulated count of nodes included in sketches, thus making the *sketches* array compressed. Now we do not need to store sketch ID's explicitly: sketch ID's are the indexes in array *sketches*. The example below shows that sketch 0 includes three nodes, sketch 1 includes (5 - 3) = 2 nodes, and so on. The *nodes* array lists the corresponding nodes.

sketches:	3	5	6	8	10

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

When we want to retrieve a sketch with ID, say i, we need to find where its nodes start in the nodes array. This is given by the number stored in $sketches[i-1]$ or 0 if $i = 0$. Testing shows CS-FA's smaller footprint and better run time than 2DL's or FA's (Section 6.5.1).

Note that Algorithm 16 does not require initialization of array *sketch* to (-1): a default value of 0 is fine, as no element can be less than 1 (any sketch must contain at least one node, the root).

5.2 Experimental Results on Arrays

We tested our IM and IE solutions by extensive experiments on several real-world graphs. All the presented results are achieved on a consumer-grade laptop with 16GB of main memory.

We implemented the algorithms in Java 8 taking advantage of parallel streams and lambda expressions, and used Webgraph [5] as a graph compression framework. We compared our implementations with each other and with the DIM algorithm, implemented in C++ [32]. We used the DIM code from

`https://github.com/todo314/dynamic-influence-analysis`.

Datasets. The datasets are available from
the Laboratory for Web Algorithmics (http://law.di.unimi.it/datasets.php).

Dataset	n	m
UK100K	100,000	3,050,615
CNR-2000	325,557	3,216,152
EU-2005	862,664	19,235,140

Table 5.1: Datasets ordered by m.

While the size of the networks we considered is in the medium range, since each node can be sampled many times (sampling with replacement), the count of edges touched by the algorithms is in the billions. For example, the smallest of presented datasets, UK100K, requires a hypergraph with a weight of at least 5.6 billion, in order to produce the IM solution for $\beta = 16$ and $k = 100$.

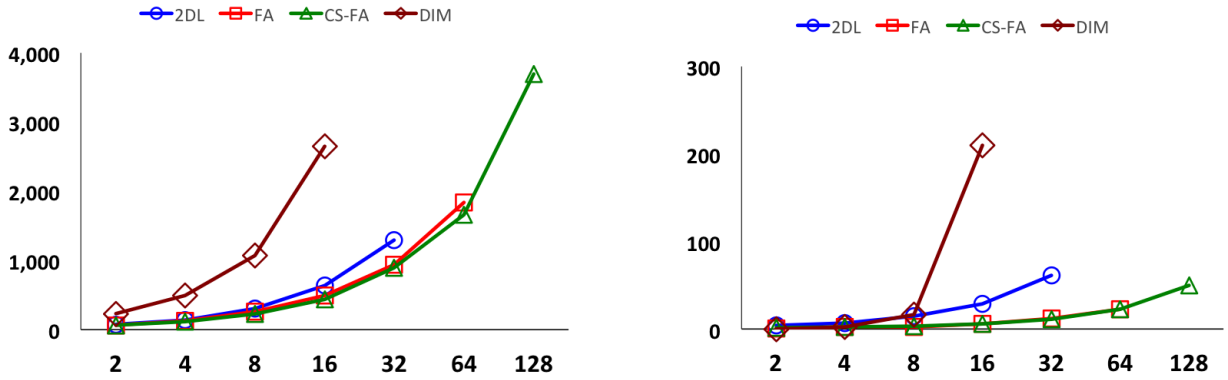
Equipment. The experiments were conducted on a laptop with processor 2.2 GHz Intel Core i7 (4-core), RAM 16GB 1600 MHz DDR3, running OS X Yosemite.

Parameters. The parameters we use in our testing are as follows. k is the number of seeds in the seed set, β is a coefficient in Borgs *et al.* formula for the hypergraph weight[1], and p is the probability of edge existence. The tests are conducted varying k and β for $p = 0.1$.

[1]Theorem 4.1 in [7], version 5, updated June 22, 2016.

5.2.1 Comparison of arrays, 2D list, and DIM performance

Figure 5.1 shows the total running time and the time used for seed calculation by DIM *vs.* our implementation of 2DL *vs.* FA *vs.* CS-FA. The test shown was conducted on CNR-2000, for $k = 10$ and $p = 0.1$, varying β in powers of 2, from 2 to 128.



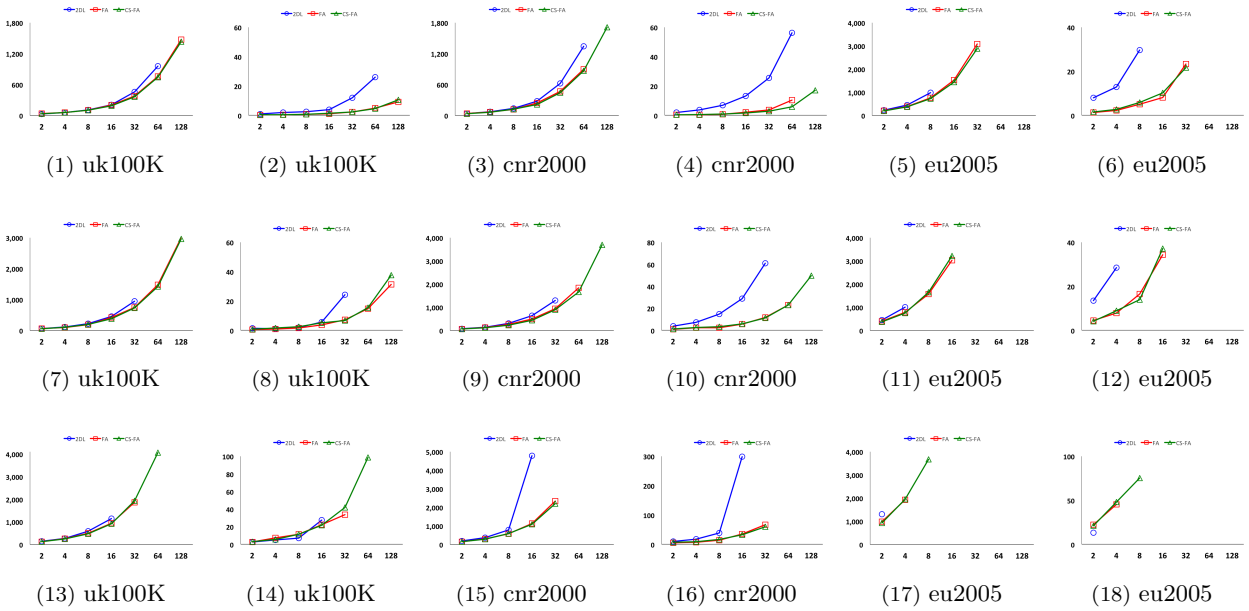
(1) Total Time (sec)

(2) Seeds Time (sec)

Figure 5.1: Processing time for cnr-2000; k=10, varying β.

The two-dimensional list implementations, DIM and 2DL, run slower and require more memory than array implementations. The reason for comparatively poor performance of 2D list implementations is the fragmentation of main memory, when allocating space for each second-dimension list of sketch numbers for a node. This causes the memory manager to perform a lot of work trying to rearrange memory blocks. Improvements implemented in 2DL listed in Section 5.1.1 allow for a better time performance on both the hypergraph computation and seed calculation, compared to DIM. For example, for $\beta = 16$, DIM took three times longer than 2DL to produce the result. The running times of FA and CS-FA are almost identical with each other. This is good for CS-FA; the compression we perform not only does not slow down CS-FA, but it makes CS-FA slightly faster due to better memory utilization. Both FA and CS-FA are faster than 2DL and DIM.

On both charts in Figure 5.1, some data points are missing, because of the required memory being higher than what is available on the machine, and the runs were terminated with OutOfMemory exception. 2DL and FA can handle runs with a β up to 32 and 64, respectively, while CS-FA can handle β equal to 128 due to its smaller memory footprint. That is, CS-FA scales the most, about 8 times more than DIM.

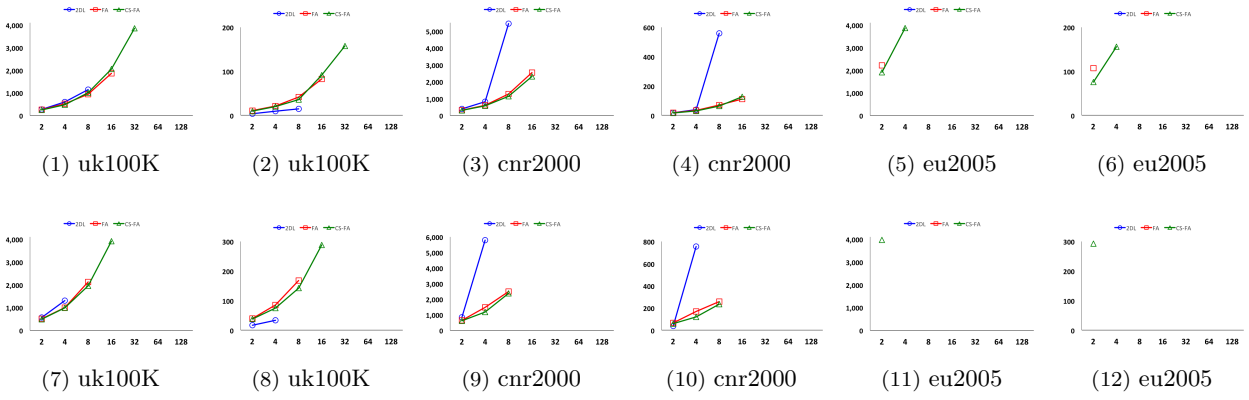


(1) uk100K (2) uk100K (3) cnr2000 (4) cnr2000 (5) eu2005 (6) eu2005

(7) uk100K (8) uk100K (9) cnr2000 (10) cnr2000 (11) eu2005 (12) eu2005

(13) uk100K (14) uk100K (15) cnr2000 (16) cnr2000 (17) eu2005 (18) eu2005

Figure 5.2: Total time (sec), and seeds time (sec). Per row, $k = 5$, 10, 25.



(1) uk100K (2) uk100K (3) cnr2000 (4) cnr2000 (5) eu2005 (6) eu2005

(7) uk100K (8) uk100K (9) cnr2000 (10) cnr2000 (11) eu2005 (12) eu2005

Figure 5.3: Total time (sec), and seeds time (sec). Per row, $k = 50$, 100.

Figure 5.2 and 5.3 show the performance of 2DL, FA, and CS-FA when parameters k and β are growing, from the first chart in Figure 5.2, where all three implementations could run to completion, till the last one in Figure 5.3, where only the most efficient data structure (CS-FA) could produce one result, for the lowest β. The larger the β and k, the longer it takes for building the hypergraph and calculating the seeds, within one graph. This can be seen in the charts, while following a column from the top chart down. The larger the graph, the longer it takes for building the hypergraph and calculating the seeds. This can be seen

in the charts, while following a row from the left chart to the right.

Another effect demonstrated by the charts in Figure 5.2 and 5.3 is the increasing number of missed data points. When space is tight, a spike of processing time occurs, as can be seen in Figure 5.1.2 for DIM, and in several charts for 2DL (for example, charts 5.3.3 and 5.3.4). Further increase of hypergraph size causes the run termination without result. All tests were run on the same laptop, with the same amount of available space, but the results show that CS-FA implementation uses the space most effectively.

The largest hypergraph, successfully created and processed on the laptop, touched almost 5.6 billion edges. This hypergraph was processed by CS-FA (Section 5.1.3). It took CS-FA one hour six minutes, including five minutes for calculating 100 seeds. We do not know another algorithm that can process such a hypergraph on a comparable machine.

Finally, in Figure 5.1, 5.2 and 5.3, we see that the time for calculating seeds is only a small part of the total time.

Chapter 6

NoSingles algorithm

In this chapter, we continue research on the IM problem: we design a new data structure for keeping the intermediate results and a novel algorithm, NoSingles (NS). We prove that NS is an algorithm with a theoretical guarantee for the achieved approximation to the optimal solution of the IM problem: NS upholds the theoretically guaranteed approximation to the optimal solution of Reversed Influence Sampling (RIS).

6.1 Data Structures for Hypergraph

In this section, we compare the design and implementation of two different hypergraph data structures. One is suggested by Borgs *et al.* [7] (and also in their US Patent US9367879B2). It is implemented in several recent papers, including DIM [32]. The other is a new data structure that we designed, implemented, and tested.

6.1.1 DIM hypergraph structure

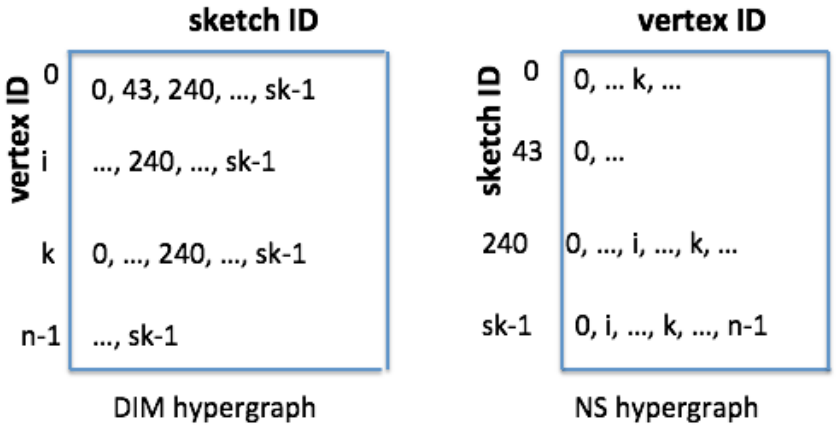


Figure 6.1: Comparing DIM and NS hypergraph data structures.

One of the recent algorithms implementing the RIS method is DIM [32]. Essentially (though with interesting shortcuts and improvements), DIM implements a hypergraph as a list of lists following the suggestion of Borgs *et al.* [7]. Figure 6.1 shows the DIM data structure.

A DIM hypergraph is built as a list of graph nodes, from 0 through $n-1$. Each node has a corresponding list of sketch IDs. After each sample, the resulting sketch ID is added to the lists of all nodes participating in the sketch. At the end of the sampling process, the most influential node will have the longest list of sketch IDs. Note that the whole hypergraph must be available for random access during the time of building it, to update the hypergraph with each new sketch.

6.1.2 The Webgraph data structure for Hypergraph

So far, as described in Chapter 5, the hypergraph data structure consuming the least memory proved to be the compressed arrays (Section 5.1.3). This finding led us to search for a new, even more efficient, data structure that would compress IM intermediate results, while providing random access to data when computing the seeds. We found the Webgraph to be the best data structure for the purpose. The Webgraph framework [5] is based on a compression technique allowing to decrease the graph size down to 10% of its edge list size. It also includes multiple algorithms implemented in Java allowing quick and easy manipulation of compressed graphs. Some other research works that also make use of compressed graph structures offered by Webgraph and are able to scale algorithms to very large graphs are [45, 44, 39, 38]. We use the Webgraph framework to build and store the hypergraph.

Let us compare how hypergraphs are built and stored using Webgraph data structure vs. the data structure suggested by Borgs *et al.* in their RIS description [7]. After its build is completed, the RIS hypergraph exists in main memory for a short time needed for seed calculation. The hypergraph is being corrupted by the process of deleting the sketch IDs for the found seeds and wiped out when the seed calculation is completed. This read-once hypergraph is computationally expensive; can we make the cost-per-usage lower? Our solution is to use the Webgraph format to store the hypergraph on a secondary memory

medium, with a possibility to load the hypergraph into the main memory for seed calculation. The hypergraph in main memory will be wiped out after the seed calculation, but it persists on the disk. We can re-use the hypergraph for computing the influence spread of different sets of seeds. We implemented and tested two algorithms for building the hypergraph using Webgraph.

Algorithm 17 TextHypergraph

Input: directed graph G, weight R

Output: hypergraph H

1: Load G, create empty text file *edge_list*
2: $H\_weight \leftarrow 0$
3: **while** $H\_weight < R$ **do**
4: $v \leftarrow$ random node of G
5: $sketch \leftarrow$ BFS[1] from $root = v$
6: **for** each $u \in sketch$ **do**
7: new line in $edge\_list \leftarrow (u, sketchID)$
8: $H\_weight \leftarrow += sketch\_weight$
9: save *edge_list* on disk
10: sort *edge_list* on disk by nodeID
11: convert *edge_list* on disk into H (Webgraph)

Algorithm 17, TextHypergraph. The TextHypergraph algorithm builds a text file of hypergraph edges and then converts the edge list to Webgraph. For each sketch, a hypergraph edge (nodeID, sketchID) is placed on a separate line in a text file. The text file of edges is saved on, *e.g.*, disk, then sorted by the edge and converted to Webgraph format. We used a custom implementation of merge sort on disk that could sort a text file with the size of up to 2TB on our laptop. Conversion to Webgraph is easy: it is one command issued from the command line[2]. This build formats the hypergraph as compressed adjacency lists {nodeID: sketchIDs}, that is, we get the Borgs *et al.* hypergraph format, but compressed into a Webgraph.

Algorithm 18, BuildHypergraph. For the BuildHypergraph algorithm, the Web-

[2]using class ArcListASCIIGraph; `http://webgraph.di.unimi.it/docs/it/unimi/dsi/webgraph/ArcListASCIIGraph.html`

Algorithm 18 BuildHypergraph

Input: directed graph G, weight R

Output: hypergraph H

1: Load G, create new empty Webgraph H
2: $H\_weight \leftarrow 0$
3: **while** $H\_weight < R$ **do**
4: $v \leftarrow$ random node of G
5: $sketch \leftarrow$ reachable nodes in G starting from v
6: append $sketch$ to H
7: $H\_weight \leftarrow += sketch\_weight$
8: return H

graph is built directly in main memory and then saved on disk. There are no ancillary programs/scripts/commands, the build is coded as a part of a Java program implementing the whole process of computing an IM or IE solution. The Webgraph is built by adding, sequentially, each calculated sketch to the stack of the previously saved sketches[3]. The build ends up with the resulting hypergraph as the compressed adjacency lists in the format {sketchID: nodeIDs}.

We tested the TextHypergraph and BuildHypergraph algorithms on several graphs (Table 6.1). We tested the algorithms using the Borgs *et al.* formula from [7, Theorem 4.1] with several different values for the coefficient β. We present here the results for cnr2000. The results for the other graphs are similar.

Figure 6.2 shows that Algorithm 18, BuildHypergraph, achieves an overall time performance and space consumption that is much better than Algorithm 17.

We used Algorithm 18, BuildHypergraph, for our implementation of the Webgraph data structure that is shown in Figure 6.1 right. The NoSingles (NS) algorithm (described in detail in Section 6.3) uses Algorithm 18 for building the hypergraph. The NS algorithm saves the sketches adding them, one by one, to a stack of sketches, in no particular order: as long as each sketch ID is unique, the order is irrelevant. This makes the build simpler and quicker. This building process is easily adapted for a parallel processing of sampling:

[3] using class IncrementalImmutableSequentialGraph; `http://webgraph.di.unimi.it/docs/it/unimi/dsi/webgraph/IncrementalImmutableSequentialGraph.html`

(1) Space, GB

(2) RunTime, sec

Figure 6.2: TextFile (Text) *vs.* Webgraph (WG), varying β.

each machine core takes samples and stacks them; when the sampling is finished, the stacks are merged together. Note that NoSingles does not need random access to the hypergraph during its build.

6.2 Parallel Build of Hypergraph

Sequential sampling takes samples on the input graph G, one by one, and stores the resulting sketches in hypergraph H. As long as each sketch has a unique ID, they can be computed and stored in a random order. This makes it possible to take samples in parallel and then combine the resulting sketches.

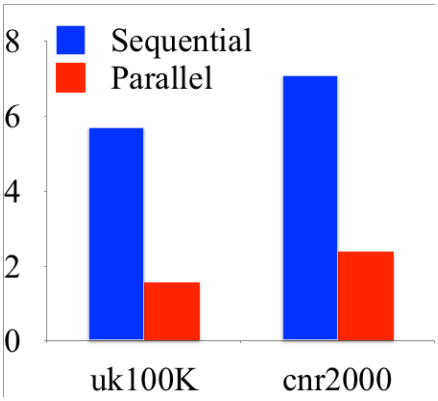
We tested the sequential and parallel (8 cores) sampling on four real-world graphs (Table 6.1). The weight for the hypergraphs was defined by the Borgs *et al.* formula[4], with $\beta = 32$ and $k = 5$.

Dataset	n	m
uk100K	100 K	3 M
cnr2000	326 K	3.2 M
eu2005	863 K	19.2 M
arabic2005	22.7 M	640 M

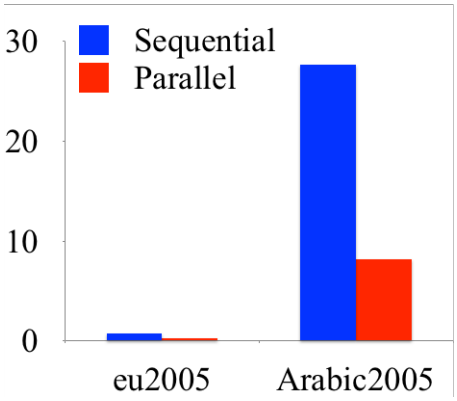
Table 6.1: Datasets for building a hypergraph.

As Figure 6.3 shows, for all the tested graphs, the overall time is significantly lower

[4]Theorem 4.1 in [7], version 5, updated June 22, 2016.



(1) UK, CNR



(2) EU, Arabic

Figure 6.3: Time performance Sequential *vs.* Parallel Sampling. (1) - minutes; (2) - hours.

when processing is done in parallel, in spite of the overhead of storing separately and then combining the partial hypergraphs. Note, that the number of samples is calculated by a formula which could be replaced by an input number of samples; it is an easy modification of our code that could be used, *e.g.* for Monte Carlo parallel sampling.

6.3 The No_Singles algorithm

Now we present our main algorithm, NoSingles, which significantly reduces the space for storing the RIS hypergraph, while fully preserving the approximation guarantee. The pseudocode is presented in Algorithm 19.

The idea behind the NoSingles algorithm is simple: why keep the single-node sketches? If random sampling picks up a node v, attempts to run Breadth-First-Search (BFS) with v as the root, and found no edges to follow, v gives us no information about its possible "influencers". So, NoSingles keeps the two-or-more-node sketches, but not the single-node sketches. It must be noted that as we are using the Borgs *et al.* bound for calculating the required number of samples, we have to include the single-node sketches into the marginal influence calculation. We do it by increasing by one the count of sketches for the randomly selected node.

Algorithm 19 NoSingles

Input: directed graph G, precision $\epsilon \in (0,1)$, number of seeds k

Output: seeds set $S \subseteq V$ of size k, spread $\sigma(S)$

1: $c \leftarrow 4(1+\epsilon)(1+1/k)$
2: $R \leftarrow cmk\epsilon^{-2}\log(n)$
3: $H \leftarrow$ BuildHypergraph(R)
4: return GetSeeds(H)
5: **procedure** BuildHypergraph(R)
6: $sk\_degree \leftarrow 0$
7: $sk\_num \leftarrow 0$
8: **while** $H\_weight < R$ **do**
9: $v \leftarrow$ random node of G^T
10: $sk \leftarrow$ BFS in G^T starting from v
11: $sk\_num = sk\_num + 1$
12: **for** each $u \in sk$ **do**
13: $node\_cover[u] \leftarrow node\_cover[u] + 1$
14: $sk\_degree \leftarrow sk\_degree + G^T.out - degree(u)$
15: **if** $sk\_cardinality > 1$ **then**
16: append sk to hypergraph H
17: $H\_weight \leftarrow H\_weight + sk\_degree$
18: return H
19: **procedure** GetSeeds(H)
20: $S \leftarrow \emptyset$
21: $\sigma(S) \leftarrow 0$
22: **for** $i = 1, \ldots, k$ **do**
23: $v_i \leftarrow argmax_v\{node\_cover[v]\}$
24: $S.insert(v_i)$
25: $\sigma(S) \leftarrow \sigma(S) + node\_cover[v_i] * n/sk\_num$
26: scan H
27: **if** $v_i \in sk_j$ **then**
28: **for** each $u \in sk_j$ **do**
29: $node\_cover[u] \leftarrow node\_cover[u] - 1$
30: $node\_cover[v_i] \leftarrow 0$
31: output S, $\sigma(S)$

The major differences of the NoSingles algorithm comparing to DIM and other RIS-based algorithms:

1. After running BFS, the sketch is stored in the format {sketch ID: nodes reached by BFS}. A stack of the sketches forms the *NoSingles hypergraph*.

2. While counting *all* the sketches, the algorithm stores only the sketches with two or more nodes (hence, the name for this algorithm).

3. NoSingles stores the hypergraph in a compressed form (using Webgraph) on a secondary memory medium (*e.g.*, hard drive).

4. On a secondary memory medium, in a binary file, we also store an array, *node_cover*, with the count of sketches for each node. Note that while the single-node sketches are not saved, we do count them and store the count of sketches for each node.

5. For computing the seed set, the algorithm loads the NoSingles hypergraph and *node_cover* into the main memory, and processes them. A consequence of NoSingles hypergraph format, {sketch: node IDs}, is the necessity to scan the full saved hypergraph after each seed is computed, as shown in Algorithm 19. However, we scan only the stored two-or-more-nodes sketches. But we use the saved total number of sketches for calculating the marginal influence.

6.3.1 Analysis of NoSingles algorithm

Observation 3. *The NoSingles hypergraph can be seen as the transpose of the Borgs* et al. *hypergraph.*

Indeed, if we imagine Borgs *et al.* hypergraph to be a "normal" graph presented as adjacency lists, the NoSingles hypergraph is its transpose: Borgs *et al.* {node: sketch IDs} adjacency lists are replaced by {sketch: node IDs} ones.

Theorem 11. *Algorithm NoSingles correctly computes a set of seeds preserving the approximation guarantee proved in [7, Theorem 3.1].*

Proof. Let us describe the process of building the hypergraph and computing the seeds by NoSingles algorithm and compare it to [7, Algorithm 1]:

1. NoSingles is taking the same number of samples, R, as mandated in [7]. In lines 1 and 2, the targeted weight of the hypergraph, R, is calculated by Borgs *et al.* formulae;
in lines 12 and 15, the *while* loop calculates the current weight, H_weight; and
in line 7, more samples are taken, if $H\_weight < R$.

2. Each sketch is adding +1 to the count of sketches for each node participating in it. The *for* loop in lines 10 – 12 scans the sketch and updates the counts. These counts exactly correspond to the degrees of the nodes in the Borgs *et al.* hypergraph.

3. Line 13 (*if* statement) checks the number of nodes in the sketch and line 14 appends the two-or-more-node sketches to the hypergraph.

4. In line 21, the next seed is found as a node with a largest count of sketches. The seed is added to the set of seeds S in line 22, and the influence spread σ is updated in line 23. This step corresponds to finding a node with maximum degree in the Borgs *et al.* hypergraph.

5. Calculating the marginal influence after each seed is found, NoSingles scans the hypergraph looking for all the sketches containing the most recently found seed (line 24). When such a sketch is found (line 25), all the nodes in it have their sketch counts decreased by 1. In Borgs *et al.* Algorithm 1, this step is exactly the same, but performed by scanning the Borgs *et al.* hypergraph and decreasing the degree of nodes by deleting the sketches that the recently found seed participated in.

6. NoSingles sets the count for the most recently found seed to zero in line 28. After that step, the count for each node is updated. The same result is achieved in the Borgs *et al.* Algorithm 1 by deleting the hypergraph row for the seed.

7. NoSingles recurses to Step 4, to find the next seed, exactly like the Borgs *et al.* Algorithm 1 does.

The NoSingles hypergraph contains only the sketches with more than one node. Will this affect any step in the above description? Yes, Step 5: after Step 5 complets, the sketch count for the seed could be not zero, because the seed could participate in single-node sketches and they were included in the count, but NoSingles hypergraph did not store them, and the count for the seed was not decreased by the number of single-node sketches it participated in. In the Borgs *et al.* algorithm, the degree of seed is always zero after deleting the seed sketches. But NoSingles rectifies it in Step 6: the sketch count is set up to zero, making the

updated NoSingles hypergraph exactly the same as the transpose of the updated Borgs *et al.* hypergraph.

Thus, NoSingles solves the IM problem by following all the steps of the RIS algorithm (Section 4.6), and the approximation guarantee proved in [7, Theorem 3.1] holds for the NoSingles algorithm. □

6.3.2 Advantages of the NoSingles algorithm

A list of major advantages of NoSingles algorithm comparing to other RIS implementations includes the following:

1. **Speeding up** the building of the hypergraph using parallelization. Storing sketches in the form {sketch: node IDs} allows for an easy implementation of parallelization: the sketches are appended to the data structure, one after the other. Each processor core does it independently. As the process of selecting the initial node (the root for BFS) is random, the order of sketch enumeration does not matter. As long as each sketch ID is unique, the algorithm creates a valid NoSingles hypergraph by simply appending the core partial hypergraphs one after the other. This is not easy to do for updating the two-dimensional (2D) list of nodes {node: sketch IDs} (Borgs *et al.* hypergraph), as sketches must be uniquely numbered. If we create several 2D lists of nodes, one in each core, the problem of merging them is not trivial: each node, in each core, will have different sketches listed under the same ID. For example, in each core a node will have sketch0; but these sketches are, in fact, different samples taken by every core independently. If, alternatively, we decide to update a global list of nodes by each core, we run into the problems of locks, deadlocks, and collision.

2. **Significant savings in space** by not storing single-node sketches, while preserving the approximation guarantee, as proved in Theorem 11. It is not possible to do this for the Borgs *et al.* hypergraph: if we do not include the IDs of one-node sketches, these nodes degrees in Borgs *et al.* hypergraph will be wrong (too low) and their influence will be calculated incorrectly.

3. NoSingles can be **scaled up** to compute the IM and IE solution for graphs of millions of nodes and hundreds of millions of edges using a consumer-grade laptop.

Dataset	n	m	**type**
uk100K	100 K	3 M	web graph
cnr2000	326 K	3.2 M	web graph
eu2005	863 K	19.2 M	web graph
ljournal2008	5.4 M	79 M	social network
arabic2005	22.7 M	640 M	web graph

Table 6.2: Datasets for statistics.

4. Furthermore, it is possible to build and store the NoSingles hypergraph once, and use it multiple times for different scenarios. For example, in the context of viral marketing, it is possible to play **"what if" scenarios** varying seeds and calculating the influence for each set (IE problem).

6.3.3 Statistics of sketch cardinality

We gathered statistics on the proportion of single-node sketches. For this purpose, we sampled one million sketches and recorded their cardinality. Tables 6.3 and 6.4 show some of the statistics gathered for the graphs listed in Table 6.2.

Dataset	min	max	**median**	**1node sketches**
uk100K	1	55743	**1**	**66%**
cnr2000	1	16222	**1**	**75%**
eu2005	1	462386	**1**	**58%**
ljounal2008	1	1841214	**1**	**58%**
arabic2005	1	4381431	**1**	**69%**

Table 6.3: Sketch Cardinality Statistics ($p = 0.1$).

Statistics clearly show that for the real-world web graphs and social networks, excluding the single-node sketches greatly reduces the number of sketches in the NoSingles hypergraph. Specifically, we made the following observations:

1. In all the tests, the **median** value of sketch cardinality is 1; that is, at least 50% of the sketches include just one node;

2. In the last column of the Tables 6.3, 6.4, we see how many single-node sketches are

Dataset	min	max	**median**	**1node sketches**
uk100K	1	2925	**1**	**91%**
cnr2000	1	794	**1**	**96%**
eu2005	1	858	**1**	**90%**
ljounal2008	1	78018	**1**	**90%**
arabic2005	1	20708	**1**	**93%**

Table 6.4: Sketch Cardinality Statistics ($p = 0.01$).

computed in each graph. It is at least 58% when $p = 0.1$, and at least 90% when $p = 0.01$;

3. Space saving depends on the probability of edge existence: the smaller the probability, the bigger the space saving achieved by NoSingles.

Experimental comparison of time and space performance of NoSingles *vs.* other state-of-the-art algorithms is described in Section 6.5.1.

6.3.4 Ranking nodes by Marginal Influence

In practical applications, it might be advantageous to have a full ranking for all graph nodes by their influence. For example, in viral marketing, to find a "sweet spot" (the best tradeoff between the money spent and the information spread) might mean to find a better solution than to define beforehand the number of seeds and then compute who are the individuals to send the free goods to.

NoSingles can be used for full graph decomposition by marginal influence with the following modifications:

1. The hypergraph weight R is calculated by Borgs *et al.* formula $R = 72(m+n)\epsilon^{-3}\log(n)$ from [7, Theorem 3.1]. This bound does not depend on the number of seeds. The bound can also be lowed with the correspondingly reduced guarantee (Theorem 4.1, *ibid.*);

2. Instead of calculating the given number of seeds, marginal influence is calculated for all n nodes in the graph.

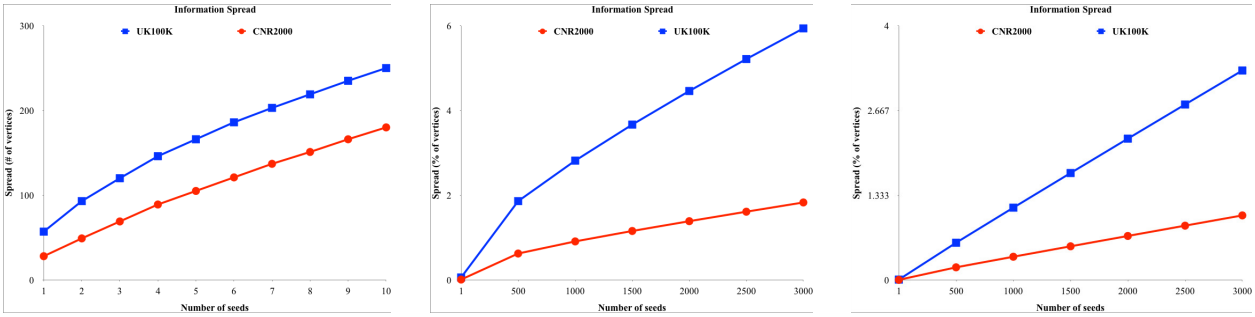
Modified procedure *GetSeeds*, named *Decomposition*, is presented in Algorithm 20. Note that we output the full set of graph nodes and the corresponding set of the information spreads, in the order of nodes' marginal influence. We can calculate the accumulated in-

formation spread without accumulating the error, because Greedy produces a sequence of seeds with information spreads for each prefix upholding the approximation guarantee (with respect to the optimal IM solution with the same number of seeds).

Algorithm 20 Graph Decomposition By Influence

```
1:  procedure DECOMPOSITION(H)
2:      S ← ∅
3:      σ(S) ← ∅
4:      for i = 1, ..., n do
5:          v_i ← argmax_v{node_cover[v]}
6:          S.insert(v_i)
7:          σ(S) = σ(S) + node_cover[v_i] * n/sk_num
8:          scan H
9:          if v_i ∈ sk_j then
10:             for each u ∈ sk_j do
11:                 node_cover[u] ← node_cover[u] − 1
12:         node_cover[v_i] ← 0
13:     output S, σ(S)
```

We present a decomposition by marginal influence (3000 most influential seeds) for cnr2000 and uk100K graphs in Figure 6.4.



(1) 10 seeds; $p = 0.01$ (2) 3000 seeds; $p = 0.01$ (3) 3000 seeds; $p = 0.001$

Figure 6.4: Decompositon by marginal influence.

We used Borgs *et al.* Theorem 4.1 in [6] for the hypergraph weight calculation, with coefficient $\beta = 10$. We used $p = 0.01$ and 0.001. Charts in Figure 6.4 show the information spread distribution. Chart 6.4.1 shows the number of nodes (information spread in absolute numbers) reached by the first ten seeds. Charts 6.4.2 and 6.4.3 show the percentage of

nodes (information spread ratio to the graph number of nodes) reached by 1, 500, 1000, ... , and 3000 seeds. The charts demonstrate that the information spread follows the law of diminishing marginal returns: the most "profitable" seed is the first one, and the more seeds are computed, the less information spread is added per seed. After a certain point, the gain is so small that it might make little sense to add more seeds. These findings agree with the typical web graph structure: a majority of entities have few, if any, connections to the rest of the network. If such a node is added as a seed, the spread gets only one more node.

6.4 The NoSinglesTopNodes algorithm

A consequence of the NoSingles hypergraph format, {sketch: node IDs}, is the necessity to scan the full hypergraph after each seed is computed, as shown in Algorithm 19. The scan finds all the seed sketches, deletes the sketches, and decreases by one the counts for all the nodes in each found sketch. This takes a lot of time as is shown in the Experimental Results, Section 6.5.4: as the number of seeds to compute grows, the seed calculation time grows significantly.

For the hypergraph in the Borgs *et al.* format, {node: sketch IDs}, it is often advantageous to use an accelerated version of the greedy algorithm called *Lazy Greedy* [27, 23], where only highly influential nodes are getting updated after each seed. To use this technique on the NoSingles hypergraph, we propose the NoSinglesTopNodes algorithm, with the following enhancements to procedure *GetSeeds*:

1. Put the nodes into a priority queue in the order of their influence; the head of the queue is the first seed;

2. For the top nodes in the priority queue, create a partial hypergraph in the format {node: sketch IDs};

3. Use Lazy Greedy acceleration on the partial hypergraph.

The pseudocode for the updated *GetSeeds* procedure is shown in Algorithm 21.

Algorithm 21 GetSeeds_topNodes

Input: hypergraph H, array *node_cover*

Output: seeds set $S \subseteq V$ of size k, spread $\sigma(S)$

1: $S \leftarrow \emptyset$
2: $\sigma(S) \leftarrow 0$
3: priority queue $pq \leftarrow$ nodes in order of influence
4: MAP: hash map *top_map* $\leftarrow$ top 20% nodes from pq
5: **for** seed $s = 1, \ldots, k$ **do**
6: PULL: pull *pq_head*
7: **if** *pq_head* $\in$ *top_map* **then**
8: **if** *pq_head* marked "recalculated" **then**
9: $S.insert(pq\_head)$ as the next seed s
10: $\sigma(S) \leftarrow \sigma(S) + node\_cover[s] * n/sk\_num$
11: $node\_cover[s] \leftarrow 0$
12: **else**
13: update *node_cover[pq_head]* in *node_cover*
14: put *pq_head* back into pq
15: mark *pq_head* "recalculated"
16: go to PULL
17: **else**
18: go to MAP with current pq
19: output S, $\sigma(S)$

6.4.1 Analysis of NoSinglesTopNodes algorithm

We used the Pareto principle for defining the top nodes as 20% of all the graph nodes, which have the highest positions in the priority queue. The priority queue nodes are constantly shifting their positions (lines 13 – 15 in Algorithm 21). When the next head of the queue is pulled, its influence is decreased if it has any sketches in common with already selected seeds. Then the node is placed back into the priority queue according to its decreased influence and marked as "recalculated". This re-shifting of the queue continues until the algorithm gets the head which is marked "recalculated". This head becomes the next seed.

As the hash map is holding only 20% of the graph nodes, it might happen that the priority

queue head falls within the other 80% of the nodes. In this case, the algorithm will re-build the hash map using the current positions of nodes in the queue. This way, we will always have only 20% of the nodes converted into format {node: sketch IDs}. However, it increases the required memory space for the algorithm completion compared to NoSingles. Here we are dealing with the tradeoff between runtime and memory consumption: we can significantly speed up the seeds calculation, especially if we need many seeds, but we will not be able to run the algorithm on larger graphs due to memory limitations.

Testing and comparison of NoSingles *vs.* NoSinglesTopNodes is described in Section 6.5.4.

6.5 Experimental Results

Dataset	n	m	**type**
uk100K	100 K	3 M	web graph
dblp2010	326 K	1.6 M	collaboration network
cnr2000	326 K	3.2 M	web graph
amazon2008	735 K	5.2 M	e-commerce graph
in2004	1.4 M	16.5 M	web graph
arabic2005	22.7 M	640 M	web graph

Table 6.5: Test datasets ordered by the number of edges m.

We implemented all the algorithms in Java 8 and used Webgraph [5] as a graph compression framework (`http://webgraph.di.unimi.it`).

Datasets. The graphs we used were obtained from
the Laboratory for Web Algorithmics [5, 4] (`http://law.di.unimi.it/datasets.php`). They vary from smaller to medium to larger sizes (Table 7.1). We picked up graphs of different types, to test our algorithms performance on arbitrary graphs.

Equipment. All the experiments were conducted on a laptop with processor 2.2 GHz Intel Core i7 (4-core), RAM 16GB 1600 MHz DDR3, running OS X Yosemite. An exception: the tests in Section 6.5.1 were conducted on a different machine, as described below.

6.5.1 NoSingles *vs.* DIM and D-SSA

A large comprehensive review and testing of the existing IM algorithms was conducted by Arora *et al.* [1]. Arora *et al.* tested several renowned IM algorithms, among them CELF++ [16], TIM [43], IRIE [20], PMC [31], and presented a comparative analysis of their performance, both runtime and memory consumption. The algorithms Arora *et al.* tested were published before May 2016, so they did not include some recent interesting and promising IM algorithms. We decided to test our NoSingles algorithm *vs.* new IM algorithms, DIM [32] and D-SSA [30], and perform a comparative analysis of the results.

Comparison of time and space performance of NoSingles with the DIM and D-SSA algorithms was done on an expensive and powerful machine with the following characteristics: CPU=Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, running OS CentOS, with RAM=1TB. We could not test these algorithms on the laptop, because both DIM and D-SSA require large memory to run IM and IE even on medium-size graphs. In Chapter 5, we show how quickly DIM consumes all the available memory on our laptop (RAM=16 GB).

6.5.2 NoSingles *vs.* DIM

Figure 6.5 shows testing results when NoSingles was compared to DIM [32] while processing IM for the graph *cnr*2000 - a medium-size web graph with 326K nodes and 3.2M edges.

Both algorithms use the RIS method, with the lower bound on the hypergraph weight as defined in Theorem 4.1 in [7]. Parameters used by both algorithms were identical throughout testing: the Borgs *et al.* coefficient $\beta = 32$, the number of calculated seeds varied: $k = 5$, 10, 25, and 50, and the probability of edge existence was taken as $p = 0.1$, 0.01, and 0.001. This ensures that the IM solutions computed by NoSingles and DIM have the same approximation guarantees.

NoSingles uses parallel sampling described in Section 6.2 and builds the NoSingles hypergraph, while DIM implements RIS by sequential sampling and builds the Borgs *et al.* hypergraph (Figure 6.1).

Charts in Figure 6.5 demonstrate that:

1. The space consumed by NoSingles is orders of magnitude smaller: roughly, 450 times

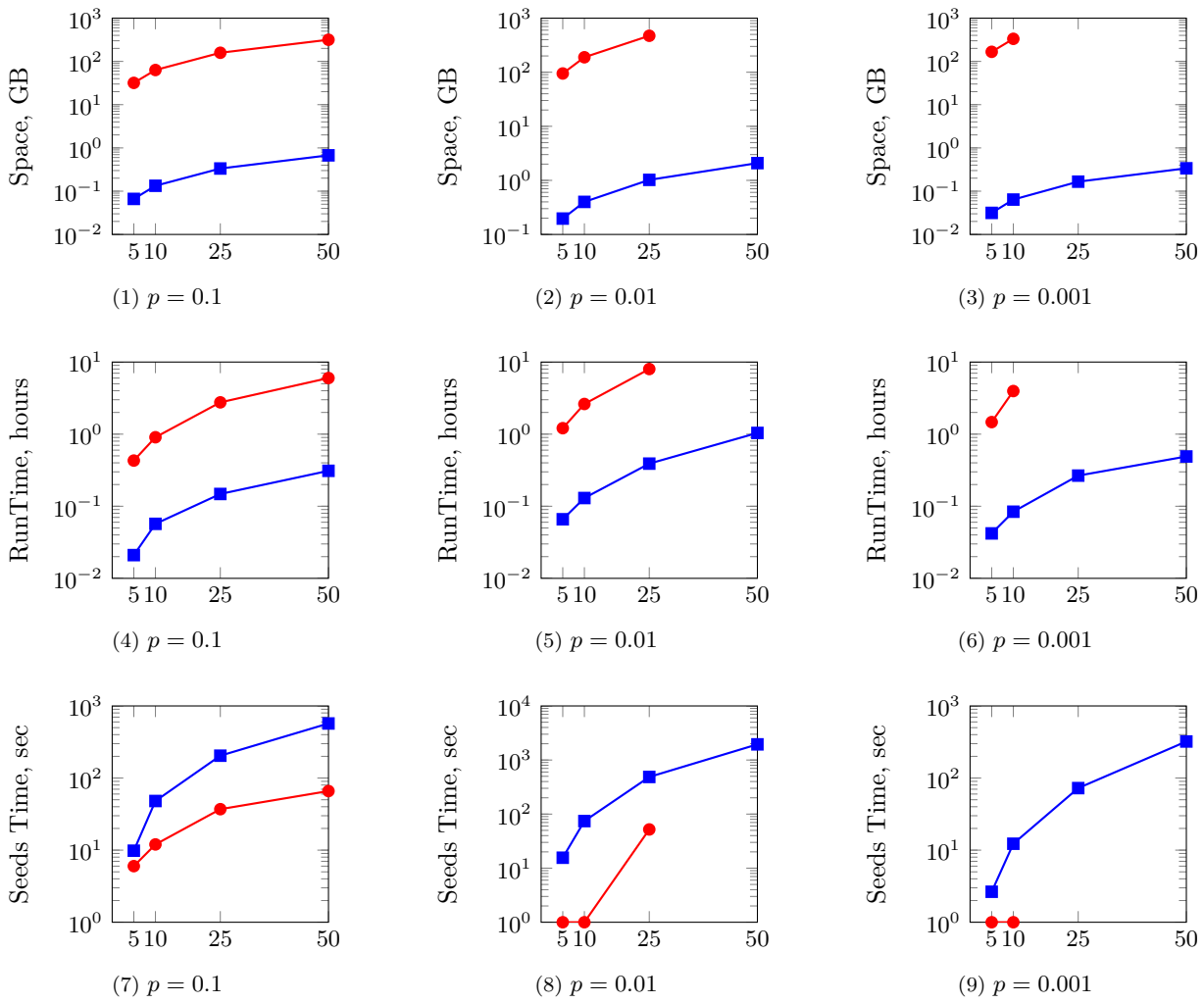


Figure 6.5: NoSingles *vs.* DIM, varying k; NS, DIM.

smaller for $p = 0.1$, 500 times smaller for $p = 0.01$, and 5000 times smaller for $p = 0.001$,

2. The time taken by the whole computation (Total Time), is lower for NoSingles, for all tested k, but time taken by the seed calculation (Seeds Time), is lower for DIM.

Space. The space consumption is shown in Figure 6.5 (1), (2), and (3). In our testing, we assigned the same p for all graph edges. Missing values for DIM in Figure 6.5 (2) and (3) mean that for smaller probability p and larger k (and the correspondingly larger hypergraph weight), DIM consumed all the memory (1 TB) and stopped processing. While taking a sample, the algorithm (NoSingles or DIM) compares an independent random variable x ($0 \leq x \leq 1$) with p and either follows the edge (if $x < p$) or ignores it. It is obvious that the

smaller p gets, the more edges the algorithm has to compare to x before finding an edge to follow. This leads to creating many light-weight sketches: the algorithm did not find many edges to follow, and the information spread is low. This, in turn, leads to increase in the number of sketches to take for a given hypergraph weight, and a corresponding expansion of the space needed for keeping these sketches. And here is the explanation of the NoSingles great advantage in space performance: NoSingles does not keep the lightest-weight sketches, *e.g.*, single-node sketches, while DIM keeps them all. The NoSingles space advantage grows with lowering p, and when DIM stops processing without producing a solution, NoSingles actually lowers its memory consumption for the same hypergraph weight: compare Figure 6.5 (2) and (3) for k =25 and 50.

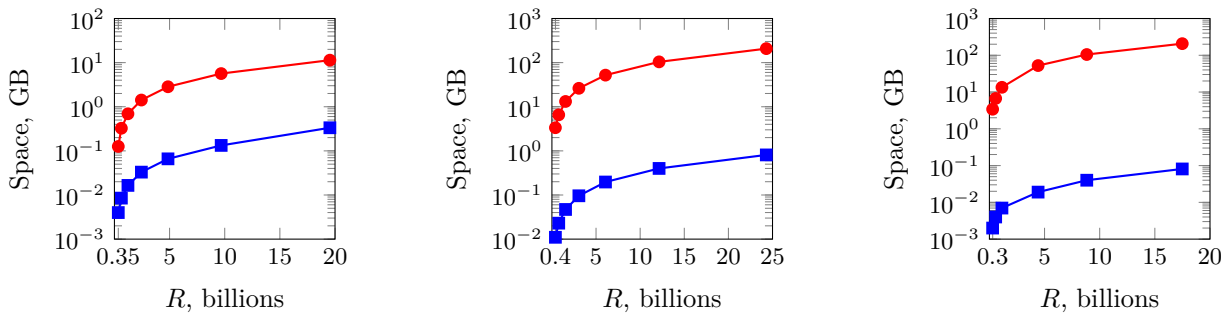
RunTime. The RunTime consists, mostly, of the time for building the hypergraph and the time for calculating the seeds. Figure 6.5 (4), (5), and (6) shows that NoSingles spends less time overall for calculating IM solutions, for all k. The main reason for this advantage is the parallel processing (48 cores) employed by NoSingles. It also helps that the NoSingles hypergraph is much smaller, and not much time is spent by the memory manager for allocating additional space as hypergraph grows.

Seeds Time. Figure 6.5 (7), (8), and (9) show that the time for calculating seeds is much lower for DIM than for NoSingles. This happens because of the different format of each hyperedge in the hypergraph: DIM creates the Borgs *et al.* hypergraph {node: sketch IDs}, and NoSingles builds the NoSingles hypergraph {sketch: node IDs}. DIM evaluates the marginal influence as the degree of the node' hyperedge, and can quickly update the hypergraph using Lazy Greedy acceleration [27]. NoSingles has to scan the whole hypergraph after each computed seed, and cannot use the Lazy Greedy technique when updating.

6.5.3 NoSingles *vs.* D-SSA

Figure 6.6 shows testing results when NoSingles was compared to D-SSA [30] while computing IM for graph *cnr*2000 - a medium-size web graph with 326K nodes and 3.2M edges.

Both algorithms use the RIS method, but different techniques for calculating the hypergraph weight to ensure the approximation guarantee. The weight of the hypergraph is the



(1) $p = 0.1$ (2) $p = 0.01$ (3) $p = 0.001$

Figure 6.6: NoSingles *vs.* D-SSA ; NS, D-SSA.

number of edges considered by the algorithm. While NoSingles uses a pre-calculated weight and builds the hypergraph until the weight is reached, D-SSA (Dynamic Stop-and-Stare Algorithm) starts with building a small hypergraph, then explores it to decide whether it is enough for calculating IM solution, and either continues building a larger hypergraph, or stops building and starts computing seeds. On the same graph, D-SSA can decide that a smaller hypergraph is appropriate for computing a large number of seeds, while a larger hypergraph is needed for computing a small number of seeds. This difference in calculating the hypergraph weight makes it impossible to use the same parameters for testing as we did in 6.5.2.

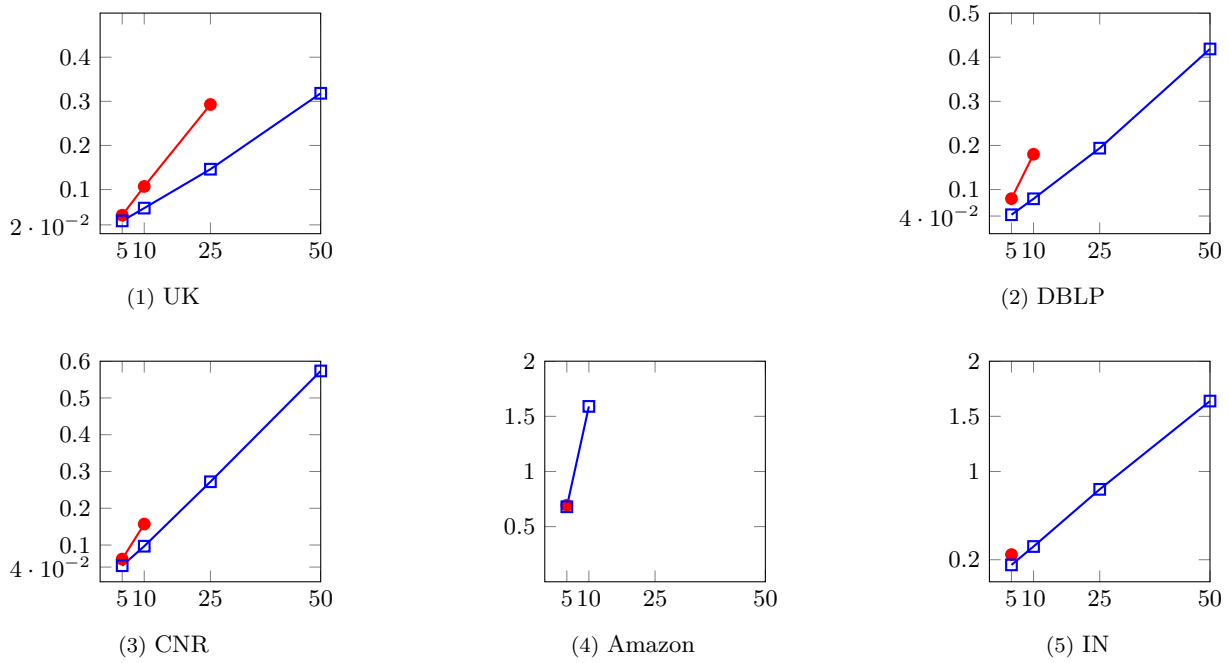
This is how we overcame the challenge of fair comparison between NoSingles and D-SSA: D-SSA was tested with different parameters, and the weight of its hypergraph and the memory used for its storage were recorded. Then we tuned the parameters for NoSingles in such a way that the NoSingles hypergraph weight would be close (within +20%) to the D-SSA hypergraph weight. In all the tests, we picked up a higher weight for the NoSingles hypergraph giving some advantage to D-SSA. In the charts presented in Figure 6.6, the X axis shows different weights for the hypergraph calculated by D-SSA. While taking a sample, algorithm (NoSingles or D-SSA) compares an independent random variable x ($0 \leq x \leq 1$) with p and either follows the edge (if $x < p$) or ignores it. We assigned the same p for all graph edges, in both algorithms.

The charts in Figure 6.6 demonstrate that for all hypergraph weights, the space consumed by NoSingles is orders of magnitude smaller than the required space for D-SSA: roughly, 35

times smaller for $p = 0.1$, 300 times smaller for $p = 0.01$, and 3000 times smaller for $p = 0.001$. The charts show that NoSingles space advantage grows with smaller p.

Testing D-SSA algorithm clearly shows that the NoSingles space advantage does not depend on a particular method used for calculating the weight of the hypergraph. For any hypergraph weight, NoSingles uses much less space to store the hypergraph.

6.5.4 NoSingles *vs.* NoSinglesTopNodes performance



(1) UK (2) DBLP (3) CNR (4) Amazon (5) IN

Figure 6.7: RunTime (hrs) NoSingles *vs.* NoSinglesTopNodes; $p = 0.1$; NS, NST.

The goal of the NoSinglesTopNodes design (Section 6.4) is to decrease the time for seed calculation. The NoSingles hypergraph format, {sketch: node IDs}, necessitates a full scan of the hypergraph for each seed. With the hypergraph sizes into billions of edges, the seed calculation takes too long. NoSinglesTopNodes transposes part of the hypergraph into format {node: sketch IDs}, which makes it possible to use Lazy Greedy acceleration [27, 23] and significantly speed up the seed calculation. We ran NoSingles and NoSinglesTopNodes algorithms on five graphs from Table 6.5 and compared their time performance. We picked up graphs of different types, with different densities and vastly different structures: for example, amazon2008 depicts similarities between books, while in2004 is a web graph of the

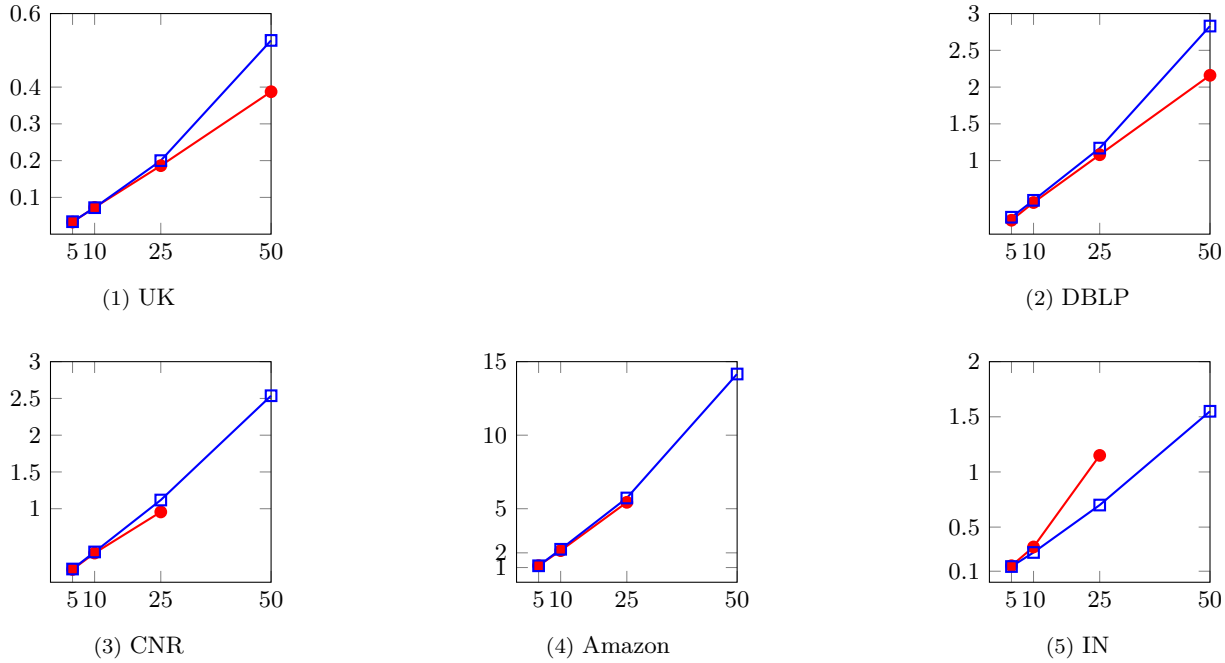


Figure 6.8: RunTime (hrs) NoSingles *vs.* NoSinglesTopNodes; $p = 0.01$; NS, NST.

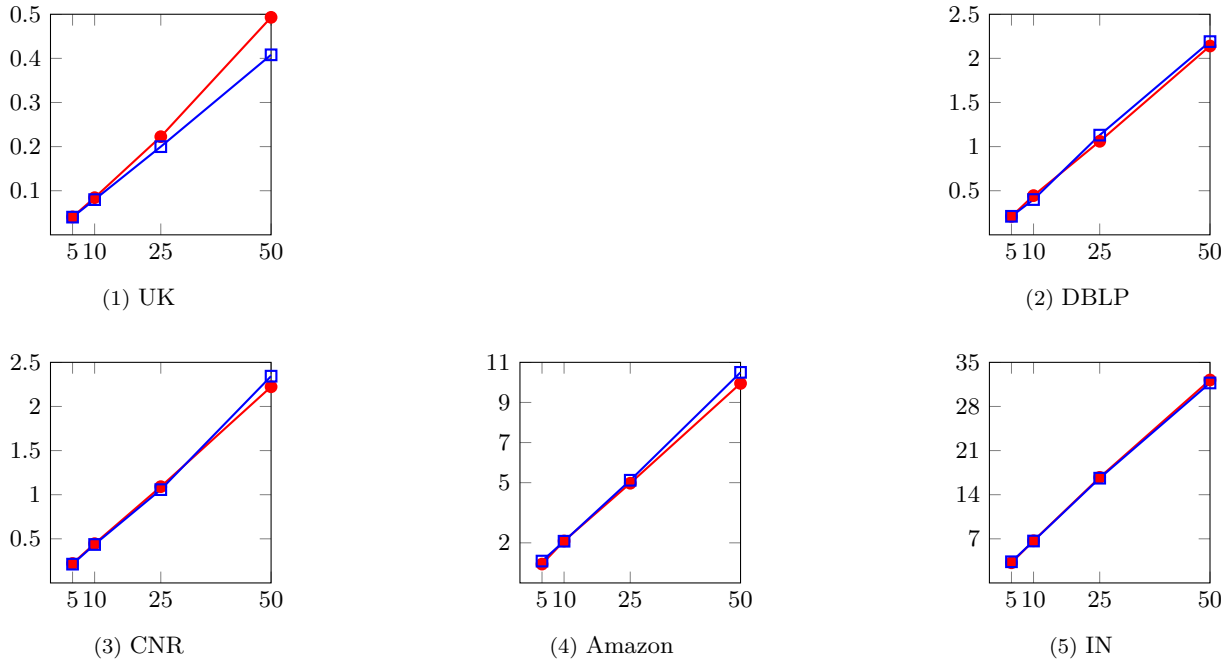


Figure 6.9: RunTime (hrs) NoSingles *vs.* NoSinglesTopNodes; $p = 0.001$; NS, NST.

`.in` domain crawled by the Nagaoka University of Technology. Our intention was to test the performance of NoSinglesTopNodes when dealing with different graph types and densities. Figure 6.7, 6.8, and 6.9, show the total time for completion with different edge existence probabilities (0.1, 0.01, and 0.001), while varying the number of seeds k.

Missing values. If a value is missing from a chart, it means that the algorithm did not complete the run and issued an OutOfMemory exception.

a. In all the tests, the NoSingles hypergraph was successfully built and stored; 16 GB memory was enough. Note that the hypergraphs are identical (if we disregard randomness) for NoSingles and NoSinglesTopNodes.

b. In almost all the tests, NoSingles successfully computed the seeds. The two tests not completed by NoSingles are shown in Figure 6.7.4. In these charts for amazon2008, the NoSingles values are missing for $k = 25$ and $k = 50$. The amazon2008 graph proved rather difficult for IM calculation: the average degree of nodes is low and most nodes have the same degree. The graph appears almost homogeneous, which is natural for the depiction of similarities between books. For our algorithms, it means that most sketches contain $2 - 3$ nodes. Actually, for this kind of graph it might be sufficient to pick up the seeds at random, without any calculations.

c. For all the tests, NoSinglesTopNodes issues the OutOfMemory exception at lower numbers of seeds (and, correspondingly, smaller hypergraphs). From the presented 30 tests, NoSinglesTopNodes failed to complete in 14 tests compared to only 2 failed tests for NoSingles. This is to be expected, as NoSinglesTopNodes requires additional memory for the transpose partial hypergraph.

Impact of p value. We tested our algorithms with different values of p (probability of edge existence). The value of p proved to significantly impact the test result: with smaller values of p, the sketches contain fewer nodes, with all the other factors fixed. If we follow the charts for the same graph from top row down, we see fewer and fewer missing values; this means that less memory is needed for the hypergraph and seed calculation. The explanation for this effect is the growing number of single-node sketches that are not stored. This is true for both algorithms, NoSingles and NoSinglesTopNodes.

Time performance. The motivation for NoSinglesTopNodes development was to signifi-

cantly decrease the seed calculation time, especially when a large number of seeds is to be computed.

a. The tests show that the best time performance for NoSinglesTopNodes compared to NoSingles is achieved for larger k.

b. However, when $p = 0.1$, the effect of using NoSinglesTopNodes is negative: the overhead of building the transpose hypergraph exceeds the possible gain in speed for seed calculation. In most tests with this p the partial hypergraph was re-build several times (for the updated top 20% nodes), and this process slows down the seed calculation to the point of making NoSinglesTopNodes too slow.

Bottomline. We recommend using the NoSinglesTopNodes algorithm when a large number of seeds is calculated for a social network with a low probability of information diffusion (for example, 0.001). Sufficient memory on your machine is required for building the partial hypergraph in the format {node: sketch IDs}.

6.5.5 IM for a large graph on a laptop

To test the scalability of NoSingles, we successfully ran arabic2005 on our laptop (RAM=16 GB). We used full Borgs *et al.* hypergraph weight formula ([7, Theorem 3.1]). Parameters, intermediate values, and results are presented in Tables 6.6 – 6.8, where n is the number of graph nodes, m is the number of edges, ϵ is the allowed error, p is the edge existence probability, k is the number of seeds to compute, R is the calculated weight of the hypergraph, sk is an abbreviation for "sketches", and H is an abbreviation for "hypergraph".

Dataset	n	m	ϵ	p	k
arabic2005	22.7 M	0.63 B	0.2	0.001	5

Table 6.6: Parameters.

R	sk, total	sk, saved	H size, edges
6.4 T	2.5 B	36.3 M	2.7 B

Table 6.7: Intermediate results.

H **space**	H time	Seeds time	accuracy	confidence
1 GB	90.5 hrs	136.5 sec	0.43	0.6

Table 6.8: Results.

The results demonstrate that NoSingles can store all the relevant information for an IM solution in a very compact format. In the example, a large graph with tens of millions of nodes and hundreds of millions of edges was processed by a consumer-grade laptop using the full Borgs *et al.* bound, with guaranteed accuracy (the guaranteed approximation to optimal) and guaranteed confidence. Moreover, the IM hypergraph is stored on a secondary memory medium, and can be loaded into main memory and re-processed, for example, for evaluating the information spread for a set of given seeds.

Chapter 7

CutTheTail algorithms

We continue our research into a scalable IM solution for massive graphs. We designed, implemented, and tested new algorithms that further decrease the required memory. The first algorithm described in this chapter is CutTheTail1 (CTT1).

7.1 CTT1 Algorithm

The logic of CTT1 algorithm is based on the following assumption:

Assumption 1. *80% of the nodes in the graph that have low out-degrees will never become seeds.*

In other words, we assume that 20% of the highest out-degree nodes is the pool where we will find all the seeds. This is an application of the Pareto principle to the out-degree of nodes in the original graph (with the original edge directions).

7.1.1 CTT1 Hypergraph

The CTT1 hypergraph follows the NoSingles hypergraph design: we stack the sketches, one by one, into a hypergraph (Figure 6.1). This design allows for a dynamic, in real time, analysis of each sketch and making an instant decision: save this sketch or not. With parallel processing, each core makes this decision on its own, independently of all other cores.

CTT1 sums up the out-degrees of all the nodes participating in a sketch, thus calculating the *combined out-degree* of the sketch, and does not save the sketches with the combined out-degree at less than the 80th percentile. Additionally, CTT1 does not save sketches containing only one node, regardless of this node's out-degree (thus repeating the NoSingles logic described in Section 6.3). Calculation of the 80th percentile of node out-degrees is done

Algorithm 22 BuildHypergraphCTT1

Input: directed graph G, weight R, int *node_tail*

Output: hypergraph H

```
1: sk_deg ← 0
2: sk_num ← 0
3: while H_weight < R do
4:     v ← random vertex of G^T
5:     sk ← BFS in G^T starting from v
6:     sk_num = sk_num + 1
7:     sk_outdeg ← 0
8:     for each u ∈ sk do
9:         node_cover[u] ← node_cover[u] + 1
10:        sk_deg ← sk_deg + G^T.outdeg(u)
11:        sk_outdeg ← sk_outdeg + G.outdeg(u)
12:    if sk_cardinality > 1 and sk_outdeg > node_tail then
13:        append sk to hypergraph H
14:    H_weight ← H_weight + sk_deg
15: return H
```

beforehand. The 80th percentile is stored into an integer *node_tail*. The *BuildHypergraph* (Algorithm 22) procedure gets *node_tail* as an input parameter. If we imagine a list of sketches sorted by their nodes combined out-degrees, we are cutting off a long "tail" of sketches with low out-degrees; this is why we named the algorithm "CutTheTail".

In Table 7.5 (Section 7.3), we compare the number of sketches created by sampling *vs.* saved by CTT1 *vs.* saved by NoSingles. It clearly shows that CTT1 saves much fewer sketches than NoSingles. We also show the charts comparing the space required for storing the CTT1 hypergraph with the space for other IM algorithms (Figure 7.4, 7.5). Note that for some graphs, a significant number of dropped sketches by CTT1 does not translate into a significant saving in space; this is explained by the small size of the dropped sketches.

Algorithm 23 GetSeeds

Input: hypergraph H, array *node_cover*, number of seeds k

Output: seeds S, set influence $\sigma(S)$

```
1:  S ← ∅
2:  σ(S) ← 0
3:  for i = 1, ..., k do
4:      v_i ← argmax_v{node_cover[v]}
5:      S.insert(v_i)
6:      σ(S) ← σ(S) + node_cover[v_i] * n/sk_num
7:      scan H
8:      if v_i ∈ sk_j then
9:          for each u ∈ sk_j do
10:             node_cover[u] ← node_cover[u] − 1
11:         remove sk_j from H
12:     node_cover[v_i] ← 0
13: return S, σ(S)
```

7.1.2 CTT1 Seeds Computing

While creating the CTT1 hypergraph, the *BuildHypergraph* procedure also stores the number of sketches for each node in an integer array *node_cover* (line 9 in Algorithm 18). The *node_cover* index corresponds to a node ID, and the stored integer is equal to the number of sketches the node participated in; we call such an integer "count". After the build is completed, the array of counts is saved on the secondary memory along with the CTT1 hypergraph. The seed calculation starts with loading the hypergraph and the *node_cover* array into main memory and finding the largest count in *node_cover*. The corresponding node is the first seed. Actually, if we want to find just one seed, there is no need to build and store a hypergraph; it would suffice to take all the samples, update the *node_cover* array, and pick up a node with the largest count.

However, we usually need more than one seed. In this case, after finding the first seed, we need to update the *node_cover* array and the hypergraph, to avoid shared influence domains. Algorithm 23 shows how we do it.

We scan the hypergraph (line 7) and if we find the first seed in a sketch (line 8), we decrease the counts in *node_cover* by 1 for all the nodes in the sketch (lines 9 - 10); then we remove the sketch from the hypergraph (line11) and set to zero the count for the first seed (line 12).

To find the next seed, we find a node with the largest count in the updated *node_cover* array (line 4), add it to the set of seeds (line 5), and calculate its influence (line 6) by the formula provided by Borgs *et al.* in [7]. If we need more seeds, we update the *node_cover* array and the hypergraph as described above. The process is repeated until we find all k seeds.

7.1.3 Analysis of CTT1

Quality. An important question to answer is the accuracy of CTT1, or the quality of CTT1 solution for IM and IE. How does cutting off a potentially large number of sketches affects the resulting solution? The first seed is always calculated correctly, holding the RIS theoretical guarantee of its quality, because CTT1 finds it and calculates its spread following the RIS method exactly.

Let us consider finding the second seed. To do it, we need to:

1. Remove from the CTT1 hypergraph all the sketches the first seed participated in, and
2. Lower the counts for all other nodes participating in the same sketches.

We can easily do (1): all the sketches the first seed participated in are removed and if there are some sketches with the first seed that were not stored by CTT1, they can be considered "removed" as well. Step (2) is more difficult: if there is a sketch with the first seed that was not saved, and if the sketch included other nodes, the counts for all these nodes cannot be decreased by one. Consequently, the counts for all the nodes in dropped sketches with the first seed could be inflated. What is the probability of CTT1 not saving a sketch containing the first seed? It can happen only if Assumption 1 was wrong, and the first seed is at a lower than the 80th percentile of out-degrees.

The same reasoning can be applied to all the found seeds, not just the first one: we can get an inaccurate result only if

1. a previously selected seed is at a lower than the 80th out-degree percentile, and

2. consequently, the *node_cover* update is inaccurate: the counts for the other nodes in dropped sketches are not decreased.

This makes it possible to incorrectly pick up the next seed(s): the count of a node is inflated, and it might, erroneously, become the largest in the *node_cover*.

Is it possible that our Assumption 1 is wrong? Well, we can imagine a node influencing just one other node, but enjoying an enormous influence overall. This can happen if this other node has huge influence over many. We all heard about a "shadow figure behind the throne": a person influencing just a king (or just a queen, or, in modern times, just a president) and making history. Another question: how often does it happen in an arbitrary graph?

We conducted hundreds of tests on over a dozen graphs of different types, modelling different phenomena, and all the results show high IM solution quality of CTT1 algorithm. Not one test showed the inflation of spread estimation by CTT1. Details can be seen in Section 7.3.2. We feel justified to claim a high accuracy of CTT1, for practical IM computations.

Time. CTT1 requires additional computations:

1. Calculate the 80th out-degree percentile , and

2. Compute the combined out-degree of each sketch, and compare it with the *node_tail*.

Calculation time for (1) is negligible compared with the sampling time, but (2) can increase the runtime of CTT1 comparatively to NoSingles runtime for the same input. On the other hand, CTT1 takes less time than NoSingles saving sketches (because it saves a smaller number of sketches), and CTT1 creates a smaller hypergraph, where computing seeds takes less time. Testing shows that in most cases, CTT1 is faster than NoSingles (Section 7.3.6).

Space. CTT1 requires loading the original graph into the main memory and keeping it there, along with the transpose (with the edge directions reversed) graph, during the hypergraph creation. The original graph is needed for calculation of the combined out-degree in each sketch. It increases the pressure on main memory comparatively to NoSingles where we need only the transpose graph. But the CTT1 hypergraph is smaller, and experiments (Section 7.3.5) show that this saving is more than enough to cover the additional memory for

the original graph. The CTT1 run takes less memory than NoSingles, for all tested graphs.

7.2 The Cut_The_Tail2 (CTT2) algorithm

This section describes another IM algorithm, CutTheTail2 (CTT2), that also drops some sketches, but the logic of CTT2 algorithm is based on a different assumption:

Assumption 2. *Sketches with relatively few nodes can be ignored when calculating the seeds.*

The CTT2 algorithm implements the RIS method of sampling, but conducts a dynamic, in real time, analysis of each sketch and makes an instant decision of saving or dropping it based on the sketch cardinality.

7.2.1 CTT2 definition of "tail"

Again, as in the NoSingles (Chapter 6) and CTT1 (Section 7.1) algorithms, we are "cutting off" the tail of shortest sketches. The difference between these algorithms is in the method of defining the tail. The CTT2 algorithm saves all sketches with cardinality (the number of participating nodes) exceeding the *sk_tail* value.

The value of *sk_tail* is calculated by an additional, before the hypergraph build starts, sampling of the graph. After multiple tests on many different graphs, we came up with an empirical formula for the number of samples to take for the *sk_tail* calculation:

$$sk\_limit = max(n/100, 10000) \tag{7.1}$$

where n is the number of nodes in the graph.
Our experience with using this formula shows that *sk_limit* is not too big to significantly slow down the CTT2 algorithm, nor is it too small for the calculated *sk_tail* to reflect the graph structure. From each sample, we save the cardinality (the number of participating nodes) of the resulting sketch. After taking *sk_limit* samples, we find the longest sketch; its cardinality is saved in an integer, *max_card*.

The value of *sk_tail* is calculated by the following formula:

$$sk\_tail = max(min(0.1 * max\_card/\log(k), 100), 2), \tag{7.2}$$

where k is the number of seeds to compute.

The formula establishes the range of *sk_tail* values. The CTT2 algorithm will

1. drop all the sketches containing 1 or 2 nodes, and
2. save all the sketches containing more than 100 nodes

for any graph, regardless of the initial sampling results. For a particular graph, *sk_tail* can get a value between 2 and 100. The value depends on k: the possible error (in spread estimate) might get larger the more seeds CTT2 computes. By dividing by $\log(k)$, we decrease *sk_tail*.

Algorithm 24 BuildHypergraph2

Input: directed graph G, weight R, int *sk_tail*

Output: hypergraph H

```
1:  sk_deg ← 0
2:  sk_num ← 0
3:  while H_weight < R do
4:      v ← random node of G^T
5:      sk ← BFS in G^T starting from v
6:      sk_num = sk_num + 1
7:      for each u ∈ sk do
8:          node_cover[u] ← node_cover[u] + 1
9:          sk_deg ← sk_deg + G^T.outdeg(u)
10:     if sk_cardinality > sk_tail then
11:         append sk to hypergraph H
12:     H_weight ← H_weight + sk_deg
13: return H
```

7.2.2 CTT2 hypergraph

Algorithm 24 shows that the CTT2 hypergraph build is similar to the CTT1 one. However, CTT2 does not have to calculate the combined out-degree in the original graph for each sketch, saving time and space. The original graph does not have to be loaded into main memory, because all computing is done on the transpose graph, using the formulae 7.1 and 7.2.

7.2.3 CTT2 Seed Computation

The CTT2 algorithm computes the seeds exactly as CTT1 does (Algorithm 23):

1. It finds the largest value in the *node_cover* array and adds the corresponding node to the seed set S, and

2. Then it scans the hypergraph, finds all the sketches the newly selected seed participated in, and updates *node_cover* and the hypergraph accordingly.

Step (2) is done only if S does not contain k seeds yet.

7.2.4 Analysis of CTT2

Quality. The CTT2 algorithm drops the sketches that contain fewer than *sk_tail* nodes. The dropped sketches might include influential nodes. The first seed is always selected correctly, as the *node_cover* array counts all the taken sketches including those that are not saved. The theoretical guarantee for accuracy (the guaranteed approximation to optimal) by Borgs *et al.* [7] holds for the first seed.

To find the second seed, CTT2 updates the *node_cover* array decreasing the counts for all the nodes participated in the same sketches with the first seed. It is possible that some of the sketches with the first seed were not saved; then, after the update, all the nodes participating in them will have their counts higher than should be. It might lead to selecting as the second seed a node with a lower real count than another node that did not participate in the first seed's dropped sketches.

The same reasoning applies to all the other seeds selected after the second one. It is clear that a possible error is accumulating as the number of seeds increases. The intuition underlining our formula for calculating *sk_tail* is that highly influential nodes, as a rule, participate in longer sketches. We are assuming that the small number of short sketches with the highly influential nodes will not lead to a big error. Section 7.3.2 shows test results that support this intuition.

Time. CTT2 requires additional computations:

1. Calculate the *sk_tail* , and

2. Compare the cardinality of each sketch with *sk_tail*.

The calculation time for (2) is negligible compared with the sampling time, but calculating *sk_tail* requires taking additional 10,000 or more ($n/100$) samples before the hypergraph build. Section 7.3.6 shows that the time increase is noticeable only for small graphs, where 10,000 additional samples take time comparable to the hypergraph build. On the other hand, CTT2 takes less time than NoSingles or CTT1 on sketch saving (because it saves a smaller number of sketches), and CTT2 creates a smaller hypergraph, where computing seeds takes less time. Tests (Section 7.3.6) show that in most cases, CTT2 is faster than NoSingles and CTT1; in some cases, significantly faster.

Space. The CTT2 algorithm requires additional initial sampling before the hypergraph creation. Results (cardinality of the sketches) are saved in an integer array. The memory taken by this array is very small (10,000 integers or, for graphs with more than one million nodes, $n/100$ integers), and it is released immediately after calculating *max_card*. CTT2 takes less memory than NoSingles or CTT1, for all tested graphs (Section 7.3.5).

Dataset	n	m	type
WordAsn	10.6K	72K	association, directed
Caida	65.5K	106.7K	social, directed
FB	4K	176K	social, undirected
EnronD	69K	275K	e-mails, directed
Enron	36.7K	368K	e-mails, undirected
Deezer	54.6K	996K	social, undirected
DBLP2010	326 K	1.6 M	collaboration, undirected
UK100K	100 K	3 M	web, directed
CNR2000	326 K	3.2 M	web, directed
DBLP2011	986K	6.7M	collaboration, undirected
Arabic2005	23M	640M	web, directed

Table 7.1: Test datasets ordered by m.

7.3 Experimental Results

All tests were conducted using Borgs *et al.* formula for the number of samples from Theorem 3.1 in [7]:

$$R = (c * m * k * \epsilon^{-2} * \log(n)) \tag{7.3}$$

where

$$c = 4.0 * (1 + \epsilon) * (1 + 1/k). \tag{7.4}$$

Borgs *et al.* proved that formula 7.3 guarantees the approximation factor of $(1 - 1/e - \epsilon)$, for any $\epsilon > 0$, in time $O((m + n)k\epsilon^{-2} \log(n))$, with confidence of at least 3/5.

Throughout the experiments, we used $\epsilon = 0.1$. We implemented the algorithms in Java 8 and used Webgraph [5] as a graph compression framework (`http://webgraph.di.unimi.it`).

Datasets. The graphs we used were obtained from
Stanford Large Network Datasets Collection (`https://snap.stanford.edu/data/`), and
Laboratory for Web Algorithmics (`http://law.di.unimi.it/datasets.php`).
They vary from smaller to medium to larger sizes (Table 7.1). We purposefully picked up graphs of different types, to thoroughly test our algorithms performance.

Equipment. All the experiments were conducted on a laptop with processor 2.2 GHz Intel Core i7 (4-core), RAM 16GB 1600 MHz DDR3, running OS X Yosemite.

7.3.1 Confidence in the approximation

The first question we would like to answer is: "How much confidence can we have in the IM solution when using the Borgs *et al.* formula?" In [7], Borgs *et al.* explain that, theoretically, the confidence in an IM solution computed by the RIS method, is "at least 3/5". It is just 60% confidence, because there is a danger that, by chance, RIS could pick up a dense part of the graph and reach the required weight of the hypergraph without enough samples. Then the sampling does not represent the graph structure adequately, and the IM solution could be a poor approximation to optimal. To increase the confidence, Borgs *et al.* advise to run RIS $\log(n)$ times and use the largest number of samples taken, for the confidence to rise from 3/5 to $(1 - 1/n)$. We did a number of tests on different graphs, varying the probability of edge existence, to see how much the number of samples changes from run to run. Tables 7.2 and 7.3 present the results for UK100K and DBLP graphs. We got similar results for other graphs and probabilities.

The tests show that the number of samples taken by our algorithms varies negligibly. After

p	samples taken	mean	St. Deviation	St.Deviation/mean
0.1	504,244	503,651	1,899	0.38%
	506,257			
	503,812			
	505,355			
	500,904			
	504,016			
	504,385			
	502,700			
	504,063			
	498,966			
	501,107			
	504,427			
	506,203			
	502,782			
	504,890			
	503,593			
	504,357			
0.001	2,319,933,031	2,319,258,334	792611	0.03%
	2,319,182,149			
	2,318,487,182			
	2,320,343,923			
	2,319,352,759			
	2,320,169,958			
	2,319,320,032			
	2,318,046,484			
	2,319,662,355			
	2,318,291,684			
	2,319,869,627			
	2,318,774,277			
	2,320,434,972			
	2,319,981,483			
	2,318,148,705			
	2,318,818,136			
	2,318,574,922			

Table 7.2: UK100K: Samples taken by $\log(n) = 17$ runs.

p	samples taken	mean	St. Deviation	St.Deviation/mean
0.1	1,292,998 1,283,413 1,280,475 1,283,296 1,283,945 1,286,285 1,281,325 1,287,256 1,282,233 1,281,837 1,283,165 1,286,173 1,280,482 1,283,438 1,281,829 1,281,448 1,283,212 1,290,504	1,284,073	3,405	0.27%
0.001	8,955,279,977 8,955,879,080 8,955,653,757 8,955,313,445 8,955,803,842 8,955,699,235 8,955,381,220 8,955,604,093 8,955,456,078 8,955,480,860 8,955,758,342 8,955,403,385 8,955,417,065 8,955,709,034 8,955,517,328 8,955,558,261 8,955,218,238 8,955,615,097	8,955,541,574	188,130	0.002%

Table 7.3: DBLP: Samples taken by $\log(n) = 18$ runs.

hundreds and hundreds of tests, we never encountered a situation where the difference in the number of samples appears to be significant; it is always well below 1%. From Tables 7.2 and 7.3, we can also see that with probability of edge existence p decreasing, the difference in the sample numbers (relative to the number of samples taken) is decreasing too. We came to the conclusion that, for the real-world graphs, the confidence in the IM solution is higher than the theoretically proven 60% and approaches 100%. We can run our algorithms just once, and the number of samples taken by them will be adequate for a good approximation to optimal.

7.3.2 Accuracy of Spread Estimation

Graph	k	p	NS	MC	**NS diff**	CTT1	MC	**CTT1 diff**	CTT2	MC	**CTT2 diff**
UK100K	5	0.1	15,040.94	14,989.30	**51.64**	15,078.54	14,988.20	**90.34**	14,920.17	14,984.00	**-63.83**
		0.05	4,139.24	4,136.41	**2.83**	4,158.83	4,138.54	**20.29**	4,131.67	4,133.76	**-2.09**
		0.01	166.02	166.20	**-0.18**	166.35	166.04	**0.31**	166.57	166.04	**0.54**
		0.005	61.61	61.56	**0.04**	61.76	61.56	**0.20**	61.67	61.56	**0.11**
		0.001	15.19	15.19	**0.00**	15.17	15.19	**-0.02**	15.16	15.19	**-0.03**
UK100K	10	0.1	16,655.96	16,620.80	**35.16**	16,617.89	16,617.00	**0.89**	16,639.59	16,619.00	**20.00**
		0.05	5,141.24	5,134.82	**6.42**	5,130.09	5,136.73	**-6.64**	5,145.54	5,141.64	**3.90**
		0.01	250.12	250.07	**0.05**	250.30	250.34	**-0.04**	250.75	250.00	**0.75**
		0.005	90.07	90.11	**-0.04**	90.00	90.11	**-0.11**	90.08	90.11	**-0.03**
		0.001	24.30	24.34	**-0.03**	24.32	24.34	**-0.02**	24.33	24.34	**-0.01**
DBLP	5	0.1	26,011.34	25,994.60	**16.74**	25,998.84	25,989.80	**9.04**	26,126.97	25,859.80	**267.17**
		0.05	3,974.71	3,967.95	**6.76**	3,970.59	3,970.13	**0.46**	3,964.74	3,909.11	**55.63**
		0.01	27.81	27.78	**0.03**	27.80	27.78	**0.02**	27.85	27.78	**0.07**
		0.005	12.63	12.62	**0.01**	12.61	12.63	**-0.02**	12.61	12.62	**-0.01**
		0.001	6.22	6.21	**0.01**	6.22	6.21	**0.01**	6.21	6.21	**0.00**
DBLP	10	0.1	26,191.10	26,134.20	**56.90**	26,160.32	26,126.80	**33.52**	26,231.23	25,862.90	**368.33**
		0.05	4,179.21	4,181.51	**-2.30**	4,194.50	4,179.77	**14.73**	4,280.93	3,968.95	**311.98**
		0.01	46.47	46.43	**0.03**	46.51	46.43	**0.08**	46.51	46.43	**0.08**
		0.005	22.61	22.65	**-0.04**	22.64	22.65	**-0.02**	22.64	22.65	**-0.01**
		0.001	12.08	12.08	**0.01**	12.06	12.08	**-0.02**	12.09	12.07	**0.03**

Table 7.4: Comparison of Spread Estimations.

We researched the accuracy of spread estimation calculated by our algorithms. An accurate estimation of influence spread (the estimated number of reached nodes by a set of

seeds) is an important factor in ensuring the high quality of the IM solution. IM algorithms estimate the influence spread anew after finding each seed. An incorrect estimation can lead to picking up a wrong node as the next seed. For example: a sketch with the newly found seed was not saved, and consequently the counts for the nodes participating in this sketch cannot be decreased by one. All the counts become inflated. If a node participated in multiple sketches with the newly found seed, its count can be inflated significantly, and this node could be erroneously picked up as the next seed.

Benchmark. As a benchmark, we use Monte Carlo simulation of influence spread in the original graph. The benchmark takes as an input two text files: the graph edge list, and the list of seeds. The output of the benchmark is an estimation of the information spread. It is calculated as an average of spreads reached by simulations. We tuned the benchmark for 20,000 simulations for each estimation.

Spread Estimation by NS, CTT1, CTT2 *vs.* the Benhmark. We tested spread

seeds in S	estim by CTT2	estim by MC	difference	%
1	3552.48	3561.28	-8.8	**-0.25**
2	3716.83	3717.99	-1.16	**-0.03**
3	3838.85	3842.97	-4.12	**-0.11**
4	3905.16	3903.04	2.12	**0.05**
5	3969.07	3968.24	0.83	**0.02**
6	4031.5	3972.57	58.93	**1.48**
7	4093.89	3976.49	117.4	**2.95**
8	4156.26	3970.52	185.74	**4.68**
9	4218.6	3973.32	245.28	**6.17**
10	4280.93	3968.95	311.98	**7.86**

Figure 7.1: Influence Spread. DBLP, $k = 10$, $p = 0.05$

estimations for five edge probabilities on eleven graphs. Table 7.4 presents the results for two graphs of different types and two probabilities p. The table displays the spread calculated by NS, CTT1, and CTT2 side by side with the benchmark spread ("MC" in the heading). The presented results are very similar to all other tested graphs: the spread estimation by NS or CTT1 is practically identical to the one computed by the benchmark; the estimation by CTT2, in some cases, gets slightly inflated. Note, that as probability of edge existence goes down, the results by all algorithms are getting closer and closer to the benchmark.

In our testing, the largest difference of spread estimation by CTT2 *vs.* the benchmark was 8% . Fig. 7.1 shows details of this estimation: the influence spread of the set of seeds S starting from the first seed and adding seeds one by one. It clearly shows that after adding

the seeds 8 – 10, the overall spread of S decreases. If "closely related" nodes are picked up, the combined spread could become smaller. Seeds 8 – 10 are probably "siblings" (directly connected) of one or more of the previously selected seeds.

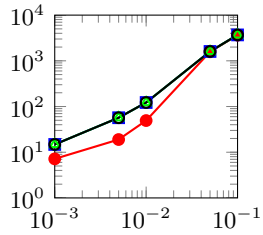
We observed this effect (an IM algorithm picking up closely related nodes) throughout our testing. It is an inherent weakness of Greedy, when local maximum is picked up, though other combinations of seeds would produce a better global solution. The Greedy weakness is intensified by CTT2 dropping large number of sketches. Future research could provide further insight into this concern.

7.3.3 Statistics on Sketches Saved

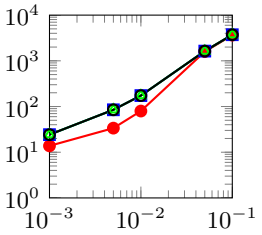
Graph	k	p	NStaken	NSsaved	%	CTT1taken	CTT1saved	%	CTT2taken	CTT2saved	%
UK	5	0.1	504,104	172,671	34.25	500,612	161,354	32.23	500,612	161,354	32.23
		0.05	3,781,711	904,552	23.92	3,777,766	839,588	22.22	3,800,353	278,757	7.34
		0.01	616,675,480	55,864,238	9.06	616,689,630	50,704,781	8.22	615,797,837	4,102,052	0.67
		0.005	1,526,286,139	86,516,196	5.67	1,526,807,116	78,136,116	5.12	1,526,802,460	5,288,614	0.35
		0.001	2,318,758,443	41,562,254	1.79	2,318,647,899	37,168,980	1.60	2,319,550,427	12,143,373	0.52
UK	10	0.1	1,005,292	343,319	34.15	1,008,624	324,821	32.20	1,006,479	188,455	18.72
		0.05	7,575,294	1,809,956	23.89	7,622,432	1,691,236	22.19	7,542,918	553,720	7.34
		0.01	1,232,812,184	111,676,143	9.06	1,232,442,194	101,325,197	8.22	1,233,408,606	7,433,148	0.60
		0.005	3,054,992,501	173,175,531	5.67	3,055,627,402	156,372,643	5.12	3,054,695,748	8,805,332	0.29
		0.001	4,638,051,350	83,155,221	1.79	4,638,029,103	74,375,955	1.60	4,637,274,400	12,479,372	0.27
DBLP	5	0.1	1,284,575	402,773	31.35	1,285,569	347,623	27.04	1,280,650	101,726	7.94
		0.05	35,719,493	6,697,295	18.75	35,750,958	5,828,306	16.30	35,797,155	421,462	1.18
		0.01	7,345,860,286	338,513,636	4.61	7,345,906,804	299,282,101	4.07	7,345,585,104	60,801,600	0.83
		0.005	8,303,430,952	198,025,876	2.38	8,303,805,759	175,565,190	2.11	8,303,626,741	20,200,381	0.24
		0.001	8,955,580,165	44,014,278	0.49	8,955,521,972	39,116,784	0.44	8,955,785,780	1,011,315	0.01
DBLP	10	0.1	2,566,013	805,633	31.40	2,567,324	694,218	27.04	2,572,919	203,434	7.91
		0.05	71,593,557	13,419,844	18.74	71,328,170	11,630,416	16.31	71,470,266	846,732	1.18
		0.01	14,691,953,097	677,084,716	4.61	14,690,970,229	598,512,951	4.07	14,691,195,814	121,611,500	0.83
		0.005	16,607,865,682	396,039,223	2.38	16,607,673,063	351,134,963	2.11	16,607,436,867	40,398,152	0.24
		0.001	17,911,485,062	88,015,564	0.49	17,911,789,392	78,225,120	0.44	17,911,349,219	2,021,773	0.01

Table 7.5: Statistics on Sketches Saved.

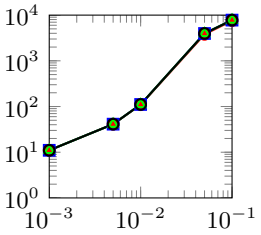
Table 7.5 provides statistics on the number of sketches taken by the IM algorithms and the number of sketches saved in the hypergraph. We present this statistics for two graphs, though we collected it for all the graphs we tested. It is rather surprising that, for example, CTT2 can drop 99.99% of sketches (for the DBLP graph, when $p = 0.001$), and compute a good set of seeds from just the 0.01% longest sketches. Note, that other RIS method implementations save all of the sketches; this explains the enormous savings in memory consumption by our algorithms.



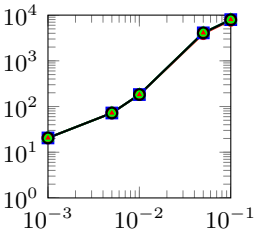
(1) Caida spread; $k = 5$



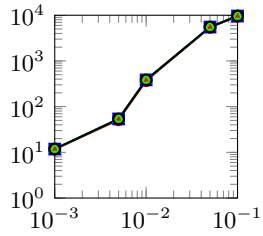
(2) Caida spread; $k = 10$



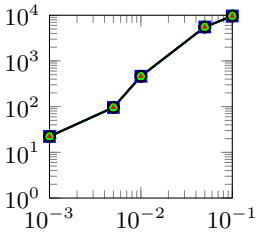
(3) EnronD spread; $k = 5$



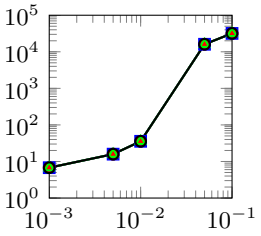
(4) EnronD spread; $k = 10$



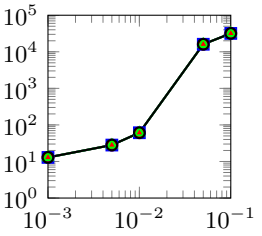
(5) Enron spread; $k = 5$



(6) Enron spread; $k = 10$

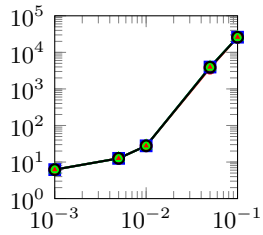


(7) DZR spread; $k = 5$

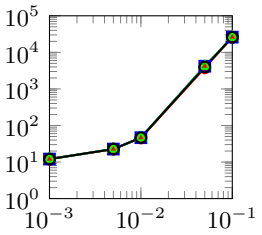


(8) DZR spread; $k = 10$

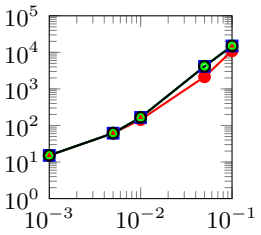
Figure 7.2: Smaller graphs quality, varying p; TopDegree, NS, CTT1, and CTT2.



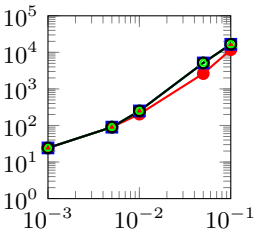
(1) DBLP spread; $k = 5$



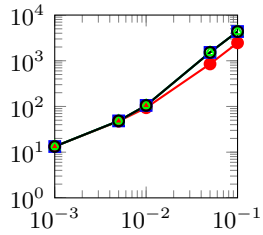
(2) DBLP spread; $k = 10$



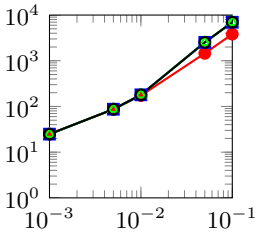
(3) UK spread; $k = 5$



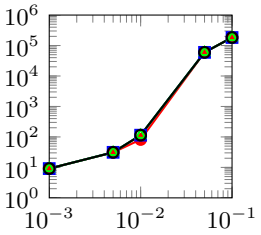
(4) UK spread; $k = 10$



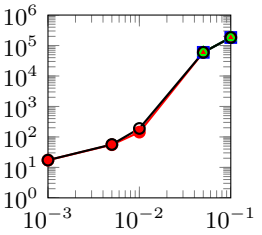
(5) CNR spread; $k = 5$



(6) CNR spread; $k = 10$



(7) DBLP2011 spread; $k = 5$



(8) DBLP2011 spread; $k = 10$

Figure 7.3: Larger graphs quality, varying p; TopDegree, NS, CTT1, and CTT2.

7.3.4 Quality

We tested the quality of IM solutions computed by CTT1 and CTT2 by comparing the spreads of their seeds with the spread of NoSingles seeds. NoSingles is proven (Theorem 11) to have the Borgs *et al.* theoretical guarantee for the approximation to optimal, so it can serve as a yardstick. For a fair comparison, we calculated all the spreads using our benchmark, Monte Carlo simulation.

Additionally, we tested the quality of a simple IM solution that is often used by marketing teams: pick up the top-degree nodes in the network as the seeds. We found the top-5 and top-10 nodes in each of tested graphs and used the benchmark to calculate their spread.

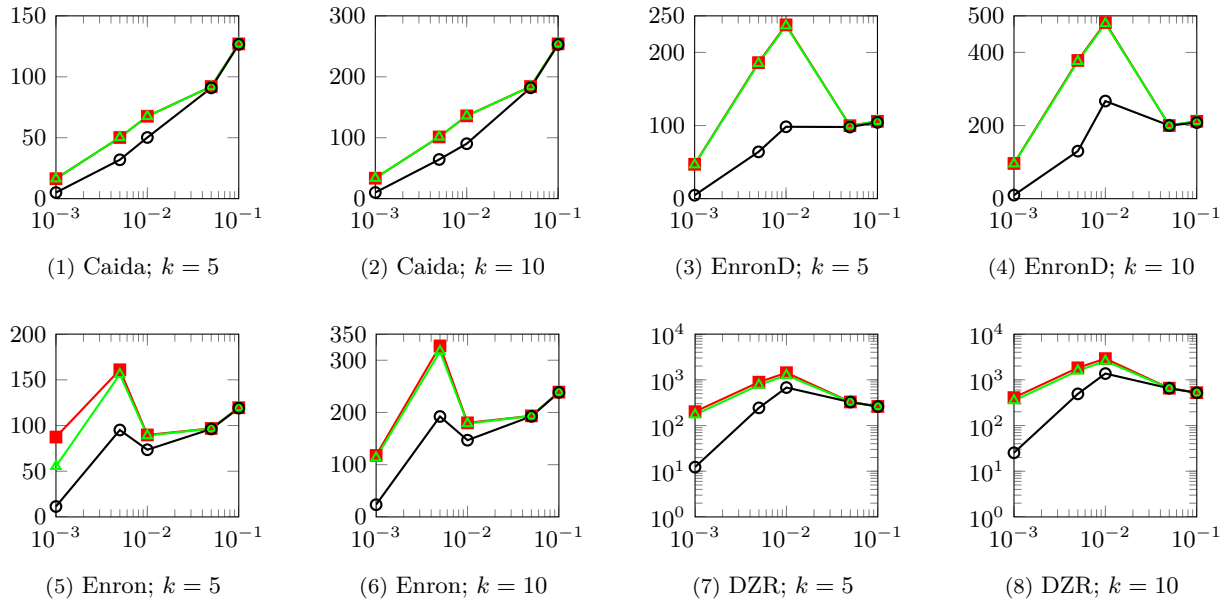


Figure 7.4: Smaller graphs space, MB, varying p; NS, CTT1, and CTT2.

The charts in Figure 7.2 and 7.3 show the spreads achieved by NoSingles (NS), CutTheTail1 (CTT1), CutTheTail2 (CTT2) algorithms, and Top Degrees (TopDeg) nodes. Analysis of the test results shows that the quality of the CTT1 and CTT2 solutions is very close to the quality of the NoSingles solution for all tested graphs. The largest difference, detected while testing UK100K graph, is within 2% of the spread value.

The quality of Top Degree solution varies. The Top Degree spread is never larger than the spread achieved by our algorithms, and often smaller. The largest difference was detected while testing Caida graph: the Top Degree spread included only 33% of the number of nodes

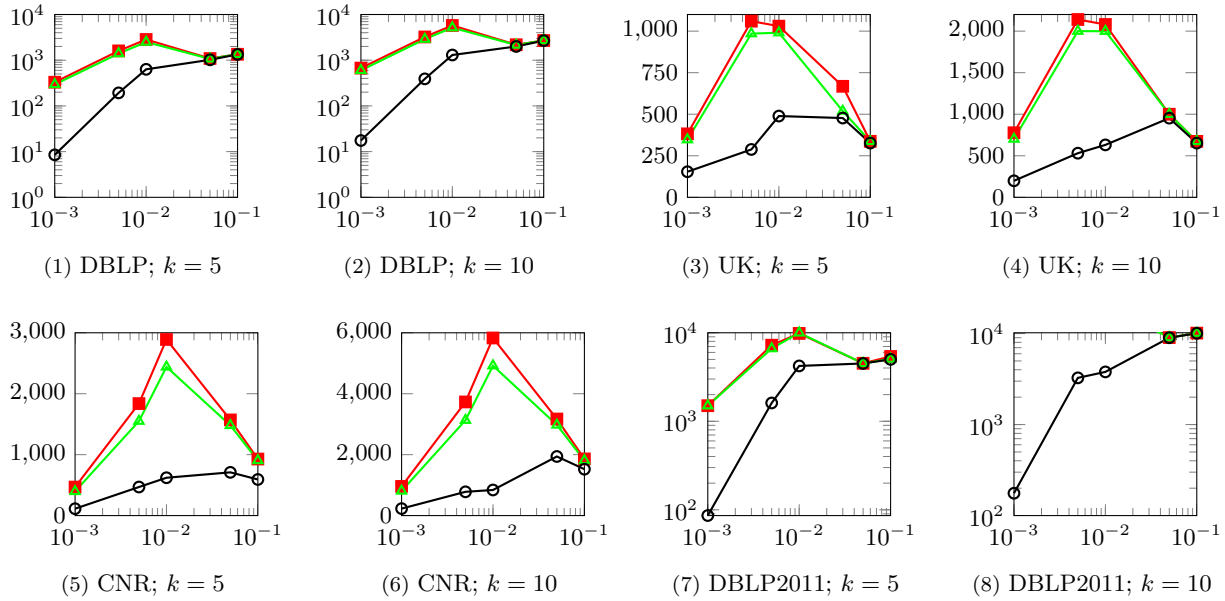


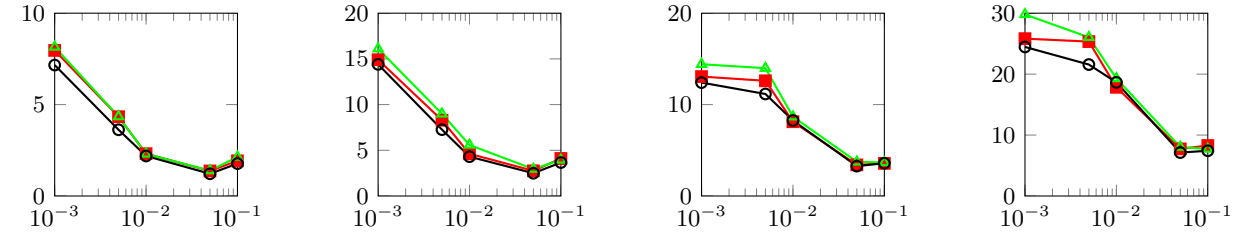
Figure 7.5: Larger graphs space, MB, varying p; NS, CTT1, and CTT2.

reached by the seeds computed by NoSingles, CTT1, or CTT2 (Figure 7.21, $p = 0.005$). But for several graphs, Top Degree provides a comparable quality solution, which explains its broad usage in marketing.

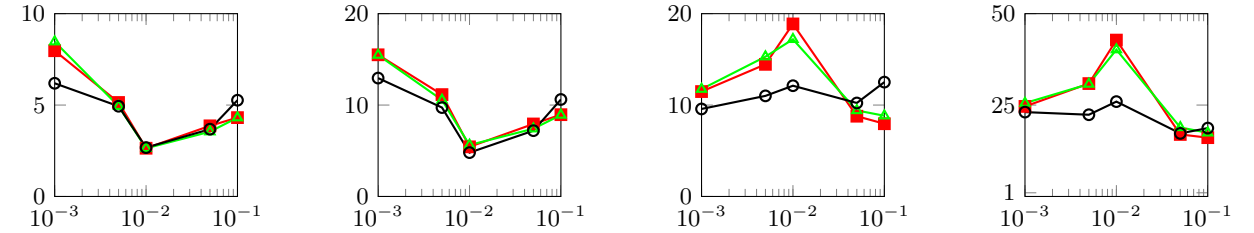
7.3.5 Space

Figure 7.4 and 7.5 show the space required by CTT1, CTT2, and NoSingles. For all tested graphs, NoSingles takes most space to store IM intermediate results, CTT1 follows closely, and CTT2 takes much less space. Space requirements vary from graph to graph. For some, with probability of edge existence p decreasing, less and less space is required for storing the hypergraph. For other graphs, the picture is more complex: the most space is taken when the probability $p = 0.01$ or $p = 0.05$. It depends on the graph structure and the distribution of degrees over the nodes.

The best by space/memory requirements is CTT2. With the quality of solution close to NoSingles and CTT1 (as shown in Section 7.3.4), it makes sense to use CTT2 for computing IM, especially for larger graphs.

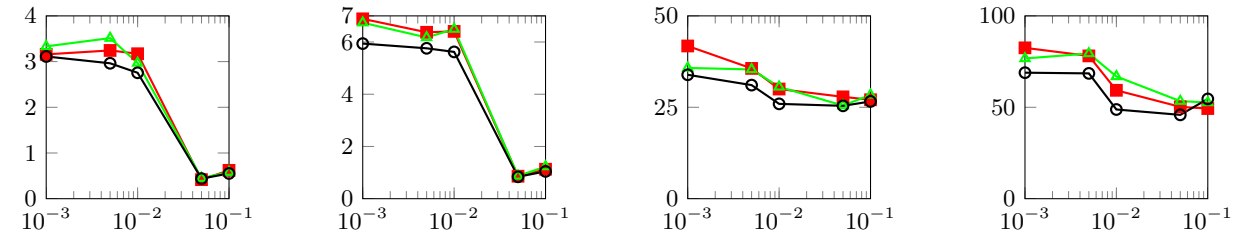


(1) Caida time, min; $k = 5$ (2) Caida time, min; $k = 10$ (3) EnronD time, min; $k = 5$ (4) EnronD time, min; $k = 10$

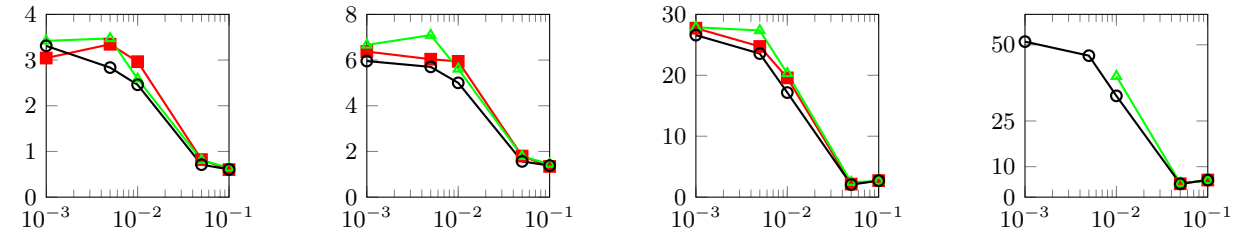


(5) Enron time, min; $k = 5$ (6) Enron time, min; $k = 10$ (7) DZR time, min; $k = 5$ (8) DZR time, min; $k = 10$

Figure 7.6: Smaller graphs runtime, varying p; NS, CTT1, and CTT2.



(1) DBLP time, hrs; $k = 5$ (2) DBLP time, hrs; $k = 10$ (3) UK time, min; $k = 5$ (4) UK time, min; $k = 10$



(5) CNR time, hrs; $k = 5$ (6) CNR time, hrs; $k = 10$ (7) DBLP2011 time, hrs; $k = 5$ (8) DBLP2011 time, hrs; $k = 10$

Figure 7.7: Larger graphs runtime, varying p; NS, CTT1, and CTT2.

7.3.6 Runtime

Figure 7.6 and 7.7 show the time taken by CTT1, CTT2, and NoSingles for computing a seed set. As a rule, NoSingles takes the most time to complete the task, CTT1 – a little less time, and CTT2 - the least time. However, as can be seen in Figure 7.6, CTT2 can be slower than other algorithms for $p = 0.1$ and dense graphs (*e.g.*, DZR), because CTT2 has to take at least 10,000 additional samples for calculating the sk_tail, as explained in Section 7.2.4.

7.3.7 CTT1 and CTT2 *vs.* DIM

p	spread NS	spread CTT	spread CTT2	spread DIM
0.1	689.256	689.713	687.236	690.172
0.05	28.7223	28.6297	28.5145	28.7223
0.01	6.8527	6.83805	6.85405	6.8534
0.005	5.85685	5.8395	5.84275	5.8584
0.001	5.13855	5.1189	5.13525	5.14325

(1) WordAsn spreads; $k = 5$.

p	spread NS	spread CTT	spread CTT2	spread DIM
0.1	3055.5	3055.15	3055.36	3055.44
0.05	2202.17	2202.17	2202.55	2201.59
0.01	269.85	269.85	269.85	269.85
0.005	35.2356	35.2356	35.2356	35.2356
0.001	8.66075	8.6597	8.6473	8.66075

(2) FB spreads; $k = 5$.

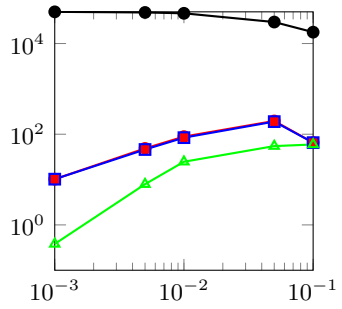
Figure 7.8: Quality: DIM, CTT1, CTT2, NoSingles.

Quality comparison

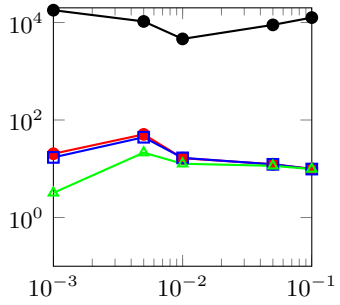
In this subsection, we present the results of testing the DIM algorithm [32] on two of our graphs. DIM code was downloaded from GitHub
(`https://github.com/todo314/dynamic-influence-analysis`)
and tuned to compute IM solutions using the Borgs *et al.* formula for the number of samples from Theorem 3.1 in [7]. The quality of IM solutions by DIM and our algorithms is so close, that we present, instead of the charts, the actual values of spreads in the table format (Figure 7.8).

Space comparison

Figure 7.9 shows the memory required for a successful run of DIM and our algorithms. DIM needs up to 5 orders of magnitude more memory for keeping the intermediate results of IM computing (WordAsn graph, $p = 0.001$).

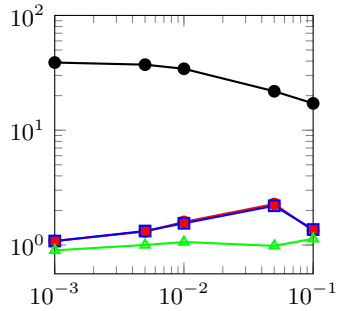


(1) WordAsn; $k = 5$

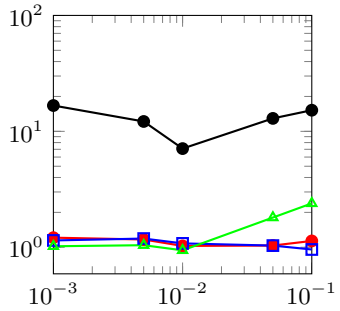


(2) FB; $k = 5$

Figure 7.9: Space, MB, varying p; NS, CTT1, CTT2, and DIM.



(1) WordAsn; $k = 5$



(2) FB; $k = 5$

Figure 7.10: Time, min, varying p; NS, CTT1, CTT2, and DIM.

Runtime comparison

Figure 7.10 shows the difference in the runtime between DIM, CTT1, CTT2, and NoSingles. It must be noted that these charts are shown for information only; the comparison of DIM runtime to our algorithms' can be seen as unfair, because DIM builds the hypergraph sequentially, while CTT1, CTT2, and NoSingles utilize parallel processing.

7.3.8 CTT2 on Arabic-2005

R	sk_tail	sk taken	sk saved	H space	CTT2 spread	MC spead
6.4 T	100	2.5B	1.4M	437MB	8,256	8,235

Table 7.6: CTT2 on Arabic-2005.

We successfully ran CTT2 on a large graph, Arabic-2005 ($n = 23$ M, $m = 640$ M). We set $k = 5$, $\epsilon = 0.2$, and $p = 0.001$. The results are presented in Table 7.6, where R is

the hypergraph weight, *sk* means sketch, and *H* means the hypergraph. Table 7.6 shows that CTT2 took 2.5 billion sketches and saved 1.4 million of them, which took only 437 MB of space. This small hypergraph allowed CTT2 to calculate the seeds with an estimated spread of 8,256 vertices. Monte Carlo simulation for the seeds shows that the spread is 8,235 vertices. The CTT2 estimation is inflated by 21 vertices, which is 0.25%. The accuracy of CTT2 is high.

7.3.9 Scalability

The quality of IM solution is the most important characteristic of IM algorithms, but the scalability is what defines the utilitarian possibilities in practical scenarios. Imagine a campaign manager deciding to spread a piece of positive information about her candidate (or a piece of negative information about a rival). With a model of the constituency in the graph form, she can calculate the most influential individuals to share the information with, and be reasonably certain about a broad spread of information in the whole community. If it is possible to do on a consumer-grade laptop, it makes the usage of a scalable IM algorithm both practical and desirable.

A fair evaluation of the scalability of IM algorithms is complex.
(1) The algorithms might use different sets of input parameters, sometimes overlapping, sometimes disjoint.
(2) The scalability achieved by the research teams that they describe in the papers can vary significantly from the results of reproducing the tests.
Tests of eleven different IM algorithms by Arora *et al.* [1] proved both points. According to [1, Table 3], under IC model, the only IM algorithm that successfully tested on a billion-size graph, was PMC [31]. It took almost 300 GB of memory to complete this test.

We did all the testing on a laptop with 16 GB RAM, keeping in mind a possible practical usage of IM algorithms. We used the Borgs et al. bound from [7, Theorem 3.1], which stipulates a large number of samples to take for a guaranteed quality of solution. For a comparison, we tested DIM algorithm (Section 7.3.7). The large memory required for a successful run of DIM is the reason for presenting only two smaller graphs in that section

(the larger one, FB, has $m = 176$ K). These were the only graphs DIM computed IM solution for. We tried to test other graphs, but 16 GB memory of our laptop was not enough: the process was "killed" by our machine, after it ran out of memory.

The largest real-world graph successfully processed on our laptop (Section 7.3.8) is Arabic-2005 – a 2005 crawl of the web pages written in Arabic performed by UbiCrawler. The directed graph has $n = 22.7$ M, and $m = 640$ M, thus exceeding half-billion size. Arabic-2005 was processed by CTT2 using the Borgs *et al.* lower bound ([7, Theorem 3.1]) on the number of samples. This bound required taking 2.5 billion samples. No other IM algorithm could take this number of samples and save the intermediate results for the seed calculation using less than 0.5 GB of space. Note that our algorithms can be modified for a tighter bound on the number of samples, with the corresponding increase in scalability.

Chapter 8

Conclusions

Graphs are used to model complex and interlinked phenomena, both natural and man-made: social, web, and computer networks, bioinformatics data, genetic profiles, product co-purchases, protein interactions, and many others, because graphs can successfully represent imprecise, uncertain, noisy data. A simple looking graph can reflect complex intricate connections and dependencies between entities. This explains a broad interest in scientific research into graph analytics. Among these, analysis of graph structure has been shown to be highly beneficial in practical applications.

In this dissertation, we described research on two important and actively researched problems in graph analytics: (1) finding influential communities, and (2) computing the most influential nodes. Both can be seen as a sophisticated analysis of graph structure that facilitates the discovery of graph characteristics important for understanding the natural world and human society. Our work concentrated on researching data structures used by the algorithms for keeping the intermediate results of computations in main memory. The intermediate results were further used for computing the most influential communities or nodes.

(1) Finding communities of nodes that have close ties with each other is of great importance in sociology, biology, computer science, and other disciplines. Many works on the subject implicitly or explicitly assume that structural communities represent groups of nodes with similar non-topological properties or functions. In reality, and it was proven by other researchers, it is far from certain that the communities reflecting only the graph structure are an adequate means to discover the "ground truth communities" (metadata groups). This questions the usefulness of the purely topological community detection algorithms to extrapolate the hidden (non-topological) features of the nodes.

To capture a non-topological property of communities, our algorithms are using a k-influential community model. This model embeds the node importance/influence into the process of community discovery.

We proposed new data structures for keeping the intermediate results and developed fast new algorithms for computing top-r k-influential communities. Our algorithms achieved orders of magnitude speed-up compared to the other existing algorithms.

We proved that the calculated k-influential communities uniquely reflect the graph structure; the computing is exact and deterministic.

We conducted extensive experiments on large and massive graphs. Our biggest tested graph was Clueweb with about 1 billion nodes and 75 billion edges. Our results showed that we were able to compute communities for every combination of k and r in a large range of values. With our implementations, we showed that we could efficiently handle massive networks using a single consumer-grade machine within a reasonable amount of time.

(2) Another problem in graph structure discovery researched in this dissertation is the problem of influence maximization: in an arbitrary graph, given the size of a subset S of nodes, find S that maximizes the probabilistic influence spread defined as the number of reachable nodes for a given edge probability. For decades, researchers have been chipping away, step by step, from the unsurmountable complexity, both time and space, of the influence maximization problem. We proposed a new approach to influence maximization complexity: by minimizing the memory footprint of influence maximization algorithms, we significantly increased the size of successfully processed graphs.

We used the Independent Cascade model of influence spread for our work, and the Reverse Influence Sampling method for calculating the number of samples.

We developed data structures that drastically shrunk the memory footprint, while preserving the intermediate results of the Influence Maximization computation necessary for calculating the seed set.

We designed and implemented new accurate and space-efficient influence maximization algorithms: NoSinges, NoSinglesTopNodes, CutTheTail1, and CutTheTail2.

We conducted experiments on a number of real-world, different type graphs, with statistical analysis of the results. Experimental comparison of the quality of solution and space

performance of our algorithms *vs.* DIM and D-SSA algorithms showed that memory required for our algorithms is orders of magnitude smaller than the one for DIM (up to 50,000 times smaller) or D-SSA (up to 3000 times smaller), for the same graph and quality of solution.

We presented a thorough analysis of our algorithms concluding that using our algorithms and data structures, it is practical to compute influence maximization for large networks on a laptop, and keep the intermediate results for future use.

However, modern networks are massive, with billions of nodes and hundreds of billions of edges. In our future research, we plan to use the results of our k-influential communities discovery for developing new heuristic algorithms for influence maximization computation on modern massive graphs.

Bibliography

[1] A. Arora, S. Galhotra, and S. Ranu. Debunking the myths of influence maximization: An in-depth benchmarking study. In *Proceedings of the 43rd ACM SIGMOD International Conference on Management of Data*, pages 651–666, 2017.

[2] V. Batagelj and M. Zaversnik. An O (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[3] F. Bi, L. Chang, X. Lin, and W. Zhang. An optimal and progressive approach to online search of top-k influential communities. *Proc. VLDB Endow.*, 11(9):1056–1068, May 2018.

[4] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596, 2011.

[5] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, pages 595–602, 2004.

[6] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. Influence maximization in social networks: Towards an optimal algorithmic solution, 2012.

[7] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 946–957, 2014.

[8] S. Chen, R. Wei, D. Popova, and A. 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*, CIKM '16, pages 1553–1562, New York, NY, USA, 2016.

[9] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1029–1038, 2010.

[10] W. Chen, Y. Wang, and S. Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 199–208, 2009.

[11] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Sketch-based influence maximization and computation: Scaling up with guarantees. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 629–638, 2014.

[12] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 991–1002, New York, NY, USA, 2014.

[13] N. Du, L. Song, M. G. Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *Proceedings of the Advances in Neural Information Processing Systems 26*, pages 3147–3155, 2013.

[14] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[15] J. Goldenberg, B. Libai, and E. Muller. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters*, 12(3):211–223, 2001.

[16] A. Goyal, W. Lu, and L. Lakshmanan. CELF++: Optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th International Conference on World Wide Web*, pages 47–48, 2011.

[17] E. Gregori, L. Lenzini, and C. Orsini. k-dense communities in the internet AS-level topology graph. *Computer Networks*, 57(1):213–227, 2013.

[18] K. Huang, S. Wang, G. S. Bevilacqua, X. Xiao, and L. V. S. Lakshmanan. Revisiting the Stop-and-Stare algorithms for influence maximization. *PVLDB*, 10:913–924, 2017.

[19] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng. Approximate closest community search in networks. *PVLDB*, 9(4), 2015.

[20] K. Jung, W. Heo, and W. Chen. IRIE: Scalable and robust influence maximization in social networks. In *Proceedings of the 12th IEEE International Conference on Data Mining*, pages 918–923, 2012.

[21] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[22] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single PC. *PVLDB*, 9(1):13–23, 2015.

[23] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 420–429, 2007.

[24] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential community search in large networks. *PVLDB*, 8(5):509–520, 2015.

[25] R.-H. Li, J. X. Yu, and R. Mao. Efficient core maintenance in large dynamic graphs. *TKDE*, 26(10):2453–2465, 2014.

[26] B. Lucier, J. Oren, and Y. Singer. Influence at scale: Distributed computation of complex contagion in networks. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 735–744, 2015.

[27] M. Minoux. Accelerated greedy algorithms for maximizing submodular set functions. *Optimization Techniques*, 7:234–243, 1978.

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

[29] H. T. Nguyen, T. P. Nguyen, T. N. Vu, and T. N. Dinh. Outward influence and cascade size estimation in billion-scale networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):20:1–20:30, 2017.

[30] H. T. Nguyen, M. T. Thai, and T. N. Dinh. Stop-and-Stare: Optimal sampling algorithms for viral marketing in billion-scale networks. In *Proceedings of the 42nd ACM SIGMOD International Conference on Management of Data*, pages 695–710, 2016.

[31] N. Ohsaka, T. Akiba, Y. Yoshida, and K. Kawarabayashi. Fast and accurate influence maximization on large networks with pruned Monte-Carlo simulations. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 138–144, 2014.

[32] N. Ohsaka, T. Akiba, Y. Yoshida, and K. Kawarabayashi. Dynamic influence analysis in evolving networks. *Proceedings of the VLDB Endowment*, 9(12):1077–1088, 2016.

[33] D. Popova, K.-i. Kawarabayashi, and A. Thomo. CutTheTail: An accurate and space-efficient algorithm for influence maximization. *Proceedings of the VLDB'19*, 2019.

[34] D. Popova, A. Knot, and A. Thomo. Data structures for efficient computation of influence maximization and influence estimation. In *Proceedings of the 21st International Conference on Extending Database Technology*, EDBT. OpenProceedings.org, 2018.

[35] D. Popova, N. Ohsaka, K.-i. Kawarabayashi, and A. Thomo. NoSingles: A space-efficient algorithm for influence maximization. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, SSDBM '18, pages 18:1–18:12, New York, NY, USA, 2018.

[36] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 61–70. ACM, 2002.

[37] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

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

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

[40] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 939–948, New York, NY, USA, 2010.

[41] J. Stallings, E. Vance, J. Yang, M. W. Vannier, J. Liang, L. Pang, L. Dai, I. Ye, and G. Wang. Determining scientific impact using a collaboration index. *Proceedings of the National Academy of Sciences*, 110(24):9680–9685, 2013.

[42] Y. Tang, Y. Shi, and X. Xiao. Influence maximization in near-linear time: A martingale approach. In *Proceedings of the 41st ACM SIGMOD International Conference on Management of Data*, pages 1539–1554, 2015.

[43] Y. Tang, X. Xiao, and Y. Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 40th ACM SIGMOD International Conference on Management of Data*, pages 75–86, 2014.

[44] B. Tootoonchi, V. Srinivasan, and A. Thomo. Efficient implementation of anchored 2-core algorithm. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages 1009–1016. ACM, 2017.

[45] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo. K-truss decomposition of large networks on a single consumer-grade machine. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 873–880. IEEE, 2018.

[46] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, MDS '12, pages 3:1–3:8, New York, NY, USA, 2012.