NoSingles: a Space-Efficient Algorithm for Influence Maximization

Diana Popova
University of Victoria
British Columbia, Canada
dpopova@uvic.ca

Naoto Ohsaka
NEC Corporation
Japan
n-ohsaka@ak.jp.nec.com

Ken-ichi Kawarabayashi*
National Institute of Informatics
Japan
k_keniti@nii.ac.jp

Alex Thomo
University of Victoria
British Columbia, Canada
thomo@uvic.ca

ABSTRACT

Algorithmic problems of computing *influence estimation* and *influence maximization* have been actively researched for decades. We developed a novel algorithm, NoSingles, based on the Reverse Influence Sampling method proposed by Borgs *et al.* in 2013. NoSingles solves the problem of influence maximization in large graphs using much smaller space than the existing state-of-the-art algorithms while preserving the theoretical guarantee of the approximation of $(1 - 1/e - \epsilon)$ of the optimum, for any $\epsilon > 0$. The NoSingles data structure is saved on the hard drive of the machine, and can be used repeatedly for playing out "what if" scenarios (e.g. trying different combination of seeds and calculating the influence spread). We also introduce a variation of NoSingles algorithm, which further decreases the running time, while preserving the approximation guarantee. We support our claims with extensive experiments on large real-world graphs. Savings in required space allow to successfully run NoSingles on a consumer-grade laptop for graphs with tens of millions of vertices and hundreds of millions of edges.

CCS CONCEPTS

• **Information systems** → **Data structures**; *Social networks*; • **Theory of computation** → **Data structures design and analysis**; • **Computing methodologies** → *Optimization algorithms*;

ACM Reference Format:
Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. 2018. NoSingles: a Space-Efficient Algorithm for Influence Maximization. In *SSDBM '18: 30th International Conference on Scientific and Statistical Database Management, July 9–11, 2018, Bozen-Bolzano, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3221269.3221291

*Also with JST, ERATO, Kawarabayashi Large Graph Project.

ACM ISBN 978-1-4503-6505-5/18/07…$15.00
https://doi.org/10.1145/3221269.3221291

1 INTRODUCTION

One of the actively researched problems in graph structure discovery is the problem of *influence maximization* (IM): in an arbitrary graph, given the size of a subset S of vertices, find S that maximizes some *influence function*. The most popular influence function is the reachability [9, 12, 22]: the network is modelled as a directed graph where entities correspond to vertices; a vertex influence is calculated as a number of other vertices reachable from it. Given a set of *seeds* (initial vertices), *influence estimation* (IE) is calculated as the total number of vertices reachable from all the seeds in the set S. Note that influence is seen as spreading from vertex to vertex via graph paths with some probability: the fact that vertices are "closely related" does not guarantee the influence spread. That is, we are dealing with *probabilistic* reachability.

To capture the probabilistic influence spread, Kempe *et al.* [12] introduced the *Independent Cascade* (IC) model [9]: an independent random variable is assigned to each directed edge (u, v); the variable reflects the level of influence from u to v. Starting from a vertex, in each step, information spreads to the vertex neighbours with the probability equal to the level of influence of the vertex over each neighbour. Kempe *et al.* showed that IM on the IC model encodes the classic maximum coverage problem and therefore is NP-hard [12]. Chen *et al.* [5] showed for the IE problem that computing the exact influence of a single seed is #P-hard. Kempe *et al.* [12] showed that IM on the IC model is *monotone* and *submodular*, and therefore, a Greedy algorithm produces provably-good quality solutions. More precisely, the influence of the *approximate* Greedy solution with given number of seeds is $(1 - 1/e - \epsilon)$ of the optimal solution, for any $\epsilon > 0$ [16]. IC became a standard model of influence spread, and we are using it for our algorithms.

Building on the Kempe *et al.* results, substantial research has been done on developing an *approximation* algorithm for IM [6–8, 12] and IE [14, 17]. However in spite of successful speed-up techniques, modern massive networks require even faster algorithms.

In 2013, a different approach was proposed by Borgs *et al.* [4]: Reverse Influence Sampling (RIS) method[1]. The idea in RIS is to select a vertex v uniformly at random, and determine the set of vertices that *would have influenced* v. If a certain vertex u appears often as influential for different randomly selected vertices, then u is a good candidate for the most influential vertex. This can be done

[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

by simulating the influence process using the IC model in the graph with the directions of edges reversed. RIS is a fast algorithm for IM, obtaining the near-optimal 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 vertices, m is the number of edges, and k is the number of seeds. Borgs *et al.* showed RIS runtime to be optimal (up to a logarithmic factor). But, just as in case of all other IM algorithms, the required main memory is so large that to successfully run RIS, large and expensive computers are needed even for medium-size graphs.

Algorithm in this paper. In this paper, we propose a space-efficient algorithm NoSingles, which allows to scale-up computing of IM and IE to massive graphs with billions of edges. The NoSingles algorithm upholds the theoretical guarantee of Borgs *et al.*, and can work under other theoretical guarantee bounds, with minor modifications. NoSingles can successfully run on consumer-grade machines unlike other IM and IE algorithms with theoretical guarantee that require expensive computers with vast amount of main memory. We compared the time and space performance of NoSingles with state-of-the-art algorithms DIM [20] and D-SSA [18]. NoSingles takes less time and much less space for the same graph processing than either DIM or D-SSA.

To achieve this result, we used several techniques all aiming at reducing the data structure size and therefore, memory footprint.

(a) We used Webgraph, a highly efficient, and actively maintained graph compression framework [3].
(b) We coded in Java 8, taking advantage of its parallel processing capabilities - streams, lambda expressions.
(c) We developed a new data structure for storing the intermediate results of IM computing.
(d) We designed a novel way of processing the graph that greatly decreases the required space, without affecting the theoretical guarantee of the approximation.

More specifically, our contributions are as follows. We present

(1) NoSingles - a space-efficient algorithm for computing IM and IE, with theoretical guarantee on the solution accuracy.
(2) A version of NoSingles algorithm, NoSinglesTopNodes, that offers a possibility for reducing the running time, with some increase in space consumption compared to NoSingles.
(3) Experimental comparison of time and space performance of NoSingles *vs.* state-of-the-art algorithms DIM [20] and D-SSA [18]. Memory required for NoSingles is orders of magnitude smaller than the one for either DIM (up to 5000 times smaller) or D-SSA (up to 3000 times smaller), for the same number of samples.
(4) Experiments on large graphs conducted on a consumer-grade laptop, with statistical analysis of the results.

Related Work. A number of IM computing algorithms have been developed in recent years, both heuristic ([5, 6, 11]) and with theoretical guarantee ([10, 18–20, 23, 24]). However, as Arora *et al.* demonstrated in [1], none of the existing algorithms satisfies the triad: quality of spread, runtime efficiency, and low memory footprint. For an IM solution on an arbitrary graph, quality of spread is the most important. Heuristic methods might work well for a certain category of graphs, but their solution quality is not guaranteed. That is why we concentrate on a method with theoretical guarantee, namely Borgs *et al.* RIS method.

Most of research teams employing RIS method ([10, 18, 23, 24]) 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.

Work [21] reports research on three distinct data structures for IM. In the present paper, we introduce two more data structures. As we show, the most scalable one, the NoSingles hypergraph, allows to drastically cut the memory consumption.

Organization. The remainder of this paper is organized as follows. Section 2 introduces several definitions and the reverse influence sampling algorithm. Section 3 proposes to use the Webgraph framework for storing the hypergraph. Section 4 describes how to parallelize the hypergraph construction. In Section 5, we present our proposed NoSingles and NoSinglesTopNodes algorithms and discuss their advantages. Section 6 reports experimental results on real-world graphs. Section 7 concludes the paper.

2 PRELIMINARIES

2.1 Notations

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

2.2 Independent Cascade

Independent Cascade (IC) model [9]: Starting from a set S of *seeds*, influence spreads in rounds/steps: each vertex after getting infected has one possibility to infect its neighbours. IC selects edges from the seed neighbourhood with *independent* probabilities. Infected neighbours, in their turn, have one possibility to infect their neighbours forming a *cascade* of information propagation. Hence, the name – Independent Cascade. The *influence spread* of a seed set S, denoted by $\sigma(S)$, is defined as the expected total number of reachable vertices for S.

2.3 IM and IE Problems

PROBLEM 1 (INFLUENCE ESTIMATION PROBLEM (IE)). *Given a graph $G = (V, E, p)$ and a vertex set $S \subseteq V$, compute the influence spread $\sigma(S)$ of S.*

PROBLEM 2 (INFLUENCE MAXIMIZATION PROBLEM (IM)). *Given a graph $G = (V, E, p)$ and an integer k, find a vertex set $S \subseteq V$ of size k that maximizes $\sigma(S)$.*

2.4 RIS Algorithm

In Reverse Influence Sampling (RIS) algorithm, Borgs *et al.* [4] employed a "polling" process on a graph: select (uniformly at random, with replacement) a vertex and find a set of vertices that *would have* *influenced* it. Repeat the polling many times. The intuition behind this process is that if a vertex appears often in sets of "influencers" then this vertex is a good candidate for the most influential vertex in the graph.

2.4.1 Hypergraph Building.

To find the "influencers", Borgs *et al.* propose to repeatedly run the graph search on the *transpose* (with the directions of edges reversed) graph. This is a randomized approach, with all the coins tossed beforehand: the probability of each edge existence in a poll is defined by a given probability function p. The resulting list of vertices reached by 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 polls, the algorithm creates a structure called by Borgs *et al. hypergraph*. In order to determine the time at which we stop polling, we define the *weight* of the hypergraph. The weight R of the hypergraph is defined as the number of edges "touched" during the graph search, in other words, it is equal to the sum of the in-degree taken over each vertex in the hypergraph.

2.4.2 Approximating IE with Hypergraphs.

We now explain how to use hypergraphs to approximately estimate the influence spread. Relying on the nature of the polling process, Borgs *et al.* have proven the following observation.

Lemma 2.1. *[4, Observation 3.2] Any vertex set S intersects a (random) set of influencers with probability $\sigma(S)/n$. In particular, the probability of a vertex 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 vertices (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 vertex 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 probability at least $1 - \delta$.

2.4.3 Solving IM with Hypergraphs.

To find the set of *seeds* (most influential vertices), 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 vertex. 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 vertex that participated in the same sketches as the already found seed. The calculation of the most influential vertex 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 2.2. *[4, 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 is shown in [4] to be runtime-optimal (up to a logarithmic factor) with respect to network size. Despite these strong results in theory, RIS and its extensions suffer from practical inefficiency as described below.

2.4.4 Practical Challenges of RIS.

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 vertices 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 vertices 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 H. Consider the "polling" process: a vertex v is picked up at random; each outgoing edge of v is "touched". With probability p, an edge is selected to follow to a vertex u, while with probability $(1 - p)$, the edge is ignored. p is a parameter for RIS run. Since the edge probability is usually picked up in the range $0.1 - 0.001$ for social networks (trivalency model; see, for example, [11]), it is much more probable that an edge will be ignored than followed. With each sketch taken, the weight R will increase much faster than the number of edges in the hypergraph. This is good news: we do not need to keep R integers, which would exceed main memory capacity for most machines. Still, the hypergraph is large: for a graph with 100K vertices 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.

3 DATA STRUCTURES FOR HYPERGRAPH

We designed three different data structures for hypergraph and the corresponding algorithms for IM and IE ([21]). The best performing data structure described in [21] is compressed flat, one-dimensional, arrays. The custom compression we used in [21] proved to save space, while not slowing down the algorithm. This result led to the next idea: to use Webgraph framework for building and storing the hypergraph.

3.1 Webgraph

Webgraph framework is a highly efficient, and actively maintained graph compression framework [3]. It based on a compression technique often allowing to get the graph size down to 10% of its edge list size. It also includes multiple algorithms implemented in Java allowing a quick and easy manipulation of compressed graphs. We used Webgraph in [21] for compressing the input graphs, but this is the first time we decided to build and store the *output* of our algorithm as a Webgraph.

The Borgs *et al.* hypergraph exists in the main memory for a short time needed for seeds 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 store the hypergraph on a secondary memory medium, with an intention to re-use the hypergraph: for example, we can re-use the hypergraph for computing the information spread of different sets of seeds.

We implemented and tested two algorithms building the hypergraph as Webgraph.

Algorithm 1 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 vertex of $G$
 5:     $sketch \leftarrow$ BFS[2] 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 vertexID
11: convert $edge\_list$ on disk into $H$ (Webgraph)
```

Algorithm 1, TextHypergraph. TextHypergraph algorithm builds a text file of hypergraph edges and then converts the edge list to Webgraph. For each sketch, we place the hypergraph edge (vertexID, sketchID) on a separate line in a text file. The text file of edges is saved on, *e.g.*, disk, then sorted by the vertexID and converted to Webgraph format. We used a custom implementation of merge sort on disk allowing to 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[3]. This build allows to format the hypergraph as compressed adjacency lists {vertexID: sketchIDs}, that is, we get the Borgs *et al.* hypergraph format, but compressed into a Webgraph.

Algorithm 2 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 vertex of $G$
 5:     $sketch \leftarrow$ reachable vertices in $G$ starting from $v$
 6:     append $sketch$ to $H$
 7:     $H\_weight \leftarrow + = sketch\_weight$
 8: return $H$
```

Algorithm 2, BuildHypergraph. Webgraph is built directly in main memory and then saved on disk. There are no ancillary programs/scripts/commands, the build is coded as part of Java program implementing the whole process of computing an IM or IE solution. Webgraph is built by adding, sequentially, each calculated sketch to the stack of the previously saved sketches[4]. We end up with the hypergraph as the compressed adjacency lists in the format {sketchID: vertexIDs}.

We tested the TextHypergraph and BuildHypergraph algorithms on several graphs (Table 1), but for brevity, present here only the results for cnr2000. The results for the other graphs are similar. We tested the algorithms using Borgs *et al.* formula from [4, Theorem 4.1] with coefficient β varied. Fig. 1 shows that Algorithm 2, BuildHypergraph achieves the overall time performance and space consumption that are much better than Algorithm 1.

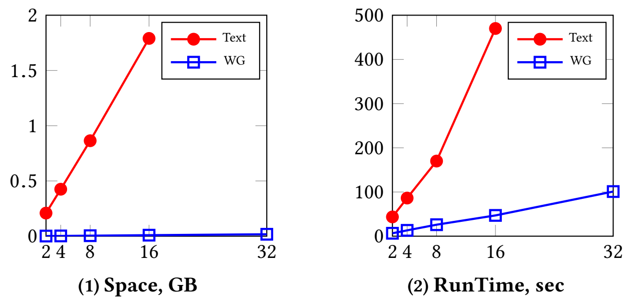


Figure 1: TextFile (Text) *vs.* Webgraph (WG), varying β

We use Algorithm 2, BuildHypergraph, for our implementation of Webgraph data structure.

4 PARALLEL BUILD OF HYPERGRAPH

We researched parallel processing applicability to random sampling. 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 1). The weight for the hypergraphs was defined by Borgs *et al.* formula[5], 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	613 M

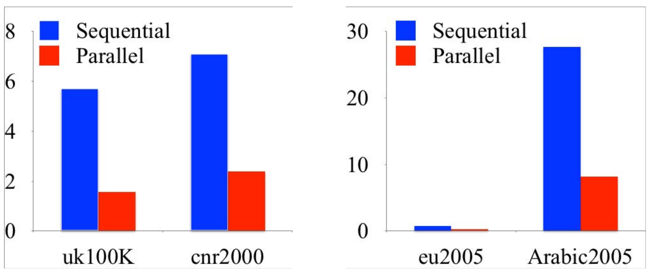
Table 1: Datasets for hypergraph build.

As we can see in Fig. 2, time performance of the sequential and parallel hypergraph builds shows a significant advantage of the parallel sampling. For all the tested graphs, the overall time is significantly lower when processing is done in parallel, in spite of the overhead of separate storing and then combining the partial hypergraphs. We use the parallel sampling for our implementations. Note, that the number of samples calculated by a formula can be

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

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

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



(1) UK, CNR **(2) EU, Arabic**

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

replaced by the input number of samples; it is an easy modification of our code that could be used, *e.g.* for Monte Carlo sampling.

5 THE NO_SINGLES ALGORITHM

Algorithm 3 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 vertex 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:             $count(u) \leftarrow count(u) + 1$
14:             $sk\_degree \leftarrow sk\_degree + G^T.outdegree(u)$
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:         $v_i \leftarrow \text{argmax}_v\{count(v)\}$
24:         $S.insert(v_i)$
25:         $\sigma(S) \leftarrow \sigma(S) + count(v_i) * n/sk\_num$
26:         scan $H$
27:         if $v_i \in sk_j$ then
28:             for each $u \in sk_j$ do
29:                 $count(u) \leftarrow count(u) - 1$
30:         $count(v_i) \leftarrow 0$
31:     output $S$, $\sigma(S)$
```

In this section, we present our main algorithm, NoSingles, which significantly reduces the space for storing the RIS hypergraph, while fully preserving the approximation guarantee. We show the pseudocode in Algorithm 3, and describe its main features next.

The idea behind the NoSingles algorithm is simple: why keep the single-vertex sketches? If random sampling picked up a vertex v, attempted 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-vertex sketches, but not the single-vertex sketches. It must be noted that as we are using Borgs *et al.* bound for calculating the required number of samples, we have to include the single-vertex sketches into the marginal influence calculation. We do it by increasing by 1 the count of sketches for the vertex.

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

(1) After computing BFS, we store the sketch in the format {sketch ID: vertices reached by BFS}. A stack of the sketches will form the *NoSingles hypergraph.*
(2) We do not store all the sketches taken. Instead, while counting *all* the sketches, the algorithm stores only the sketches with two or more vertices (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 vertex. Note that while we do not store single-vertex sketches, we do count them and store the count of sketches for each vertex.
(5) For computing the seed set, the algorithm loads the NoSingles hypergraph and *node_cover*, and processes them. This way, we scan only the stored two-or-more-vertices sketches. But we use the saved total number of sketches for calculating the marginal influence.

5.1 Analysis of NoSingles algorithm

Observation 1. *NoSingles hypergraph can be seen as the transpose Borgs* et al. *hypergraph.*

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

Theorem 5.1. *Algorithm NoSingles correctly computes a set of seeds preserving the approximation guarantee proved in [4, Theorem 3.1].*

Proof. Let us describe the process of finding the seeds by NoSingles algorithm and compare it to [4, Algorithm 1] .

(1) NoSingles is taking the same number of samples, R, as mandated in [4]. 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, makes a decision of taking more samples, if $H\_weight < R$.

(2) Each sketch is adding +1 to the count of sketches for each vertex participating in it. *for* loop in lines 10 − 12 scans the sketch and updates the counts. These counts exactly correspond to the degrees of vertices in Borgs *et al.* hypergraph.
(3) Line 13 (*if* statement) checks the number of vertices in the sketch and line 14 appends the two-or-more-vertex sketches to the hypergraph.
(4) In line 21, the next seed is found as the vertex with the 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 the vertex with the maximum degree in 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 vertices 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 vertices.
(6) NoSingles sets the count for the most recently found seed to zero in line 28. After that step, we have got the updated number of sketches for each vertex. The same result is achieved in Borgs *et al.* Algorithm 1 by deleting all the sketches from the hypergraph row for the seed.
(7) NoSingles recurses to Step 4, to find the next seed, exactly like Borgs *et al.* Algorithm 1 does.

The NoSingles hypergraph contains only the sketches with more than one vertex. Will this affect any step in the above description? Yes, Step 5: after Step 5 completed, the sketch count for the seed could be not zero, because the seed could participate in single-vertex sketches and they were included in the count, but NoSingles hypergraph did not store them. So, after updating the seed count by scanning the NoSingles hypergraph, the count for the seed was not decreased by the number of single-vertex sketches it participated in. In 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 transpose updated Borgs *et al.* algorithm.

Thus, NoSingles solves IM problem by following all the steps of the RIS algorithm (subsection 2.4), and the approximation guarantee proved in [4, Theorem 3.1] holds for NoSingles algorithm. ■

5.2 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: vertex 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 vertex (the root for BFS) is random, the order of sketch numeration 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. It is not easy to do for updating the two-dimensional (2D) list of vertices {vertex: sketch IDs} (Borgs *et al.* hypergraph), as sketches must be uniquely numbered. If we create several 2D lists of vertices, one in each core, the problem of merging them is not trivial: each node, in each core, with 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 vertices by each core, we run into the problems of locks, deadlocks, and collision.
(2) **Significant savings in space** by not storing single-vertex sketches, while preserving the approximation guarantee, as proved in Theorem 5.1. It is not possible to do for the Borgs *et al.* hypergraph: if we do not include the IDs of one-vertex sketches, these vertices degrees in Borgs *et al.* hypergraph will be wrong (too low) and their influence will be calculated incorrectly.
(3) NoSingles allows to **scale up** and compute the IM and IE solution for the graphs of millions of vertices and hundreds of millions of edges using a consumer-grade laptop.
(4) Furthermore, it is possible to build and store 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).

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	613 M	web graph

Table 2: Datasets for statistics.

5.3 Statistics of sketch cardinality

We gathered statistics on the proportion of single-vertex sketches. For this purpose, we sampled 1M sketches and recorded their cardinality. Tables 3, 4 show some of the statistics gathered for the graphs listed in Table 2.

Dataset	min	max	**median**	**1vertex 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 3: Sketch Cardinality Statistics ($p = 0.1$).

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

Dataset	min	max	**median**	**1vertex 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 4: Sketch Cardinality Statistics ($p = 0.01$).

(1) in all the tests, **median** value of sketch cardinality is 1; that is, at least 50% of the sketches include just one vertex;
(2) in the last column of the Tables 3, 4, we see how many single-vertex sketches are 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 subsection 6.1.

5.4 The NoSinglesTopNodes algorithm

A consequence of NoSingles hypergraph format, {sketch: vertex IDs}, is the necessity to scan the full hypergraph after each seed is computed, as shown in Algorithm 3. The scan finds all the seed sketches, deletes the sketches, and decreases by one the counts for all the vertices in each found sketch. This takes a lot of time as is shown in the Experimental Results, subsection 6.2: as the number of seeds to compute grows, the seeds calculation time is growing significantly.

For the hypergraph in the Borgs *et al.* format, {vertex: sketch IDs}, it is often advantageous to use an accelerated version of the greedy algorithm called *Lazy Greedy* [13, 15], where only highly influential vertices are getting updated after each seed. To use this technique on NoSingles hypergraph, we propose NoSinglesTopNodes algorithm, with the following enhancements to procedure *GetSeeds*:

(1) put the vertices into a priority queue in the order of their influence; the head of the queue is the first seed;
(2) for the top vertices in the priority queue, create a partial hypergraph in the format {vertex: sketch IDs};
(3) use Lazy Greedy acceleration on the partial hypergraph.

The pseudocode for the updated *GetSeeds* procedure is shown in Algorithm 4. **Analysis of NoSinglesTopNodes algorithm.** We used Pareto principle for defining the top nodes as 20% of all the graph vertices, which have the highest positions in the priority queue. The priority queue vertices are constantly shifting their positions (lines 13 – 15 in Algorithm 4). When the next head of the queue is pulled, its influence is decreased, if it has any sketches in common with already found seeds. Then the vertex 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 was already recalculated. This head becomes the next seed.

As the hash map is holding only 20% of the graph vertices, it might happen that the priority queue head is in the rest 80% of the vertices. In this case, the algorithm will re-build the hash map using the current positions of vertices in the queue. This way, we will always have only 20% of the vertices converted into format {vertex: sketch IDs}. However, it increases the required memory space for the algorithm completion, compared to NoSingles. Here we are dealing with the tradeoff between the 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 subsection 6.2.

Algorithm 4 GetSeeds_topNodes

```
Input: hypergraph $H$, array $node\_cover$ with vertex counts
Output: seeds set $S \subseteq V$ of size $k$, spread $\sigma(S)$
1: $S \leftarrow \emptyset$
2: $\sigma(S) \leftarrow 0$
3: priority queue $pq \leftarrow$ vertices in order of influence
4: MAP: hash map $top\_map \leftarrow$ top 20% vertices 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:            $\sigma(S) \leftarrow \sigma(S) + count(s)$
11:            $count(s) \leftarrow 0$
12:        else
13:            update $count(pq\_head)$ 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 $S$, $\sigma(S)$
```

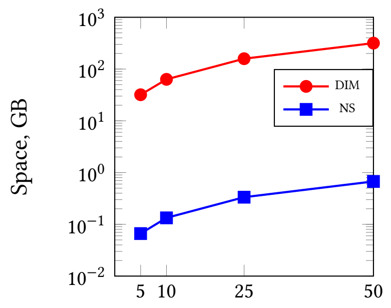
6 EXPERIMENTAL RESULTS

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	631.2 M	web graph

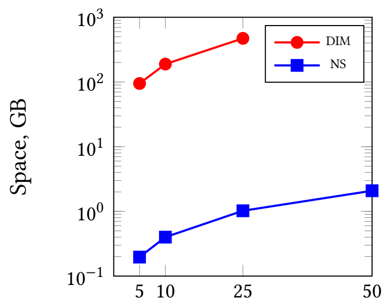
Table 5: Test datasets ordered by the number of edges m.

We implemented all the algorithms in Java 8 and used Webgraph [3] as a graph compression framework (http://webgraph.di.unimi.it).

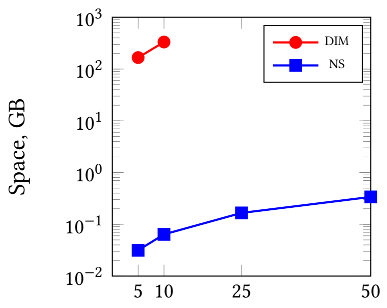
Datasets. The graphs we used were obtained from the Laboratory for Web Algorithmics [2, 3] (http://law.di.unimi.it/datasets.php). They vary from smaller to medium to larger sizes (Table 5). We picked up graphs of different types, to test our algorithms performance on an arbitrary graph.



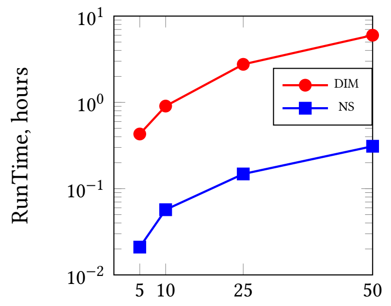
(1) $p = 0.1$



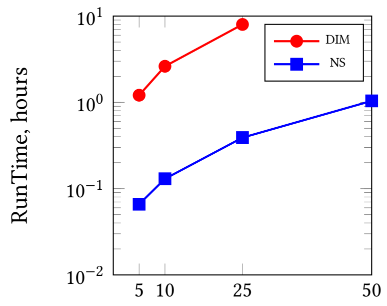
(2) $p = 0.01$



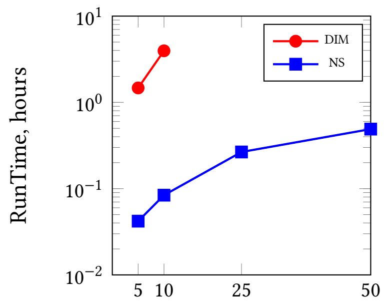
(3) $p = 0.001$



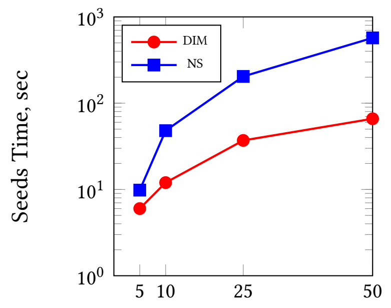
(4) $p = 0.1$



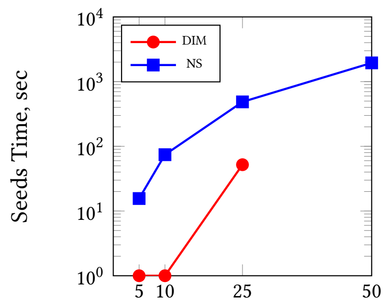
(5) $p = 0.01$



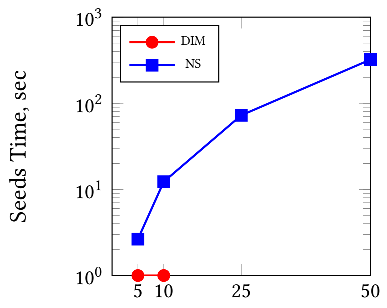
(6) $p = 0.001$



(7) $p = 0.1$



(8) $p = 0.01$



(9) $p = 0.001$

Figure 3: NoSingles *vs.* DIM, varying k

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 subsection 6.1 were conducted on a different machine, as described below.

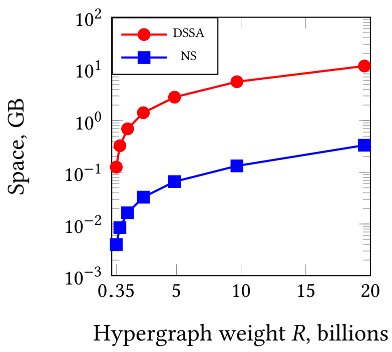
6.1 NoSingles *vs.* DIM and D-SSA

A large comprehensive review and testing of the existing state-of-the-art IM algorithms was conducted by Arora *et al.* [1]. Arora *et al.* tested several renown IM algorithms, among them CELF++ [10], TIM [24], IRIE [11], PMC [19], 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 [20] and D-SSA [18], and perform a comparative analysis of the results.

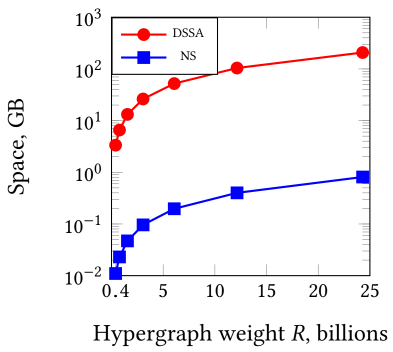
Comparison of time and space performance of NoSingles with DIM and D-SSA algorithms was done on an expensive and powerful machine with the following characteristics: CPU=Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, running OS CentOS, with RAM=1TB. We could not test these algorithms on the laptop, because both DIM and D-SSA require large memory to run IM and IE even on medium-size graphs. In [21], we show how quickly DIM consumes all the available memory on our laptop (RAM=16 GB).

6.1.1 NoSingles vs. *DIM.* Fig. 3 shows testing results when NoSingles was compared to DIM ([20]) while processing IM for graph *cnr*2000 - a medium-size web graph with 326K vertices and 3.2M edges.

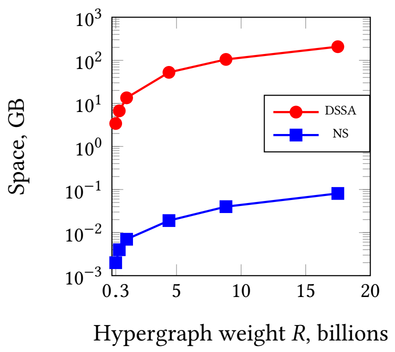
Both algorithms use RIS method, with the lower bound on the hypergraph weight as defined in Theorem 4.1 in [4], version 5, updated June 22, 2016. Parameters used by both algorithms were identical throughout testing: Borgs *et al.* coefficient $\beta = 32$, number



(1) $p = 0.1$



(2) $p = 0.01$



(3) $p = 0.001$

Figure 4: NoSingles *vs.* D-SSA

of calculated seeds varied: $k = 5, 10, 25, 50$, and the probability of edge existence was taken as $p = 0.1, 0.01, 0.001$. This ensures that the IM solutions computed by NoSingles and DIM have the same approximation guarantees.

NoSingles uses parallel sampling described in subsection **??** and builds NoSingles hypergraph (Section 5), while DIM implements RIS by sequential sampling and builds Borgs *et al.* hypergraph (Subsection 2.4).

Charts in Fig. 3 demonstrate that:

(1) space consumed by NoSingles is orders of magnitude smaller: roughly, 450 times smaller for $p = 0.1$, 500 times smaller for $p = 0.01$, and 5000 times smaller for $p = 0.001$,
(2) time taken by the whole processing (Total Time), is lower for NoSingles, for all tested k, but
(3) time taken by the seed calculation (Seeds Time), is lower for DIM.

Space. Space consumption is shown in Fig. 3 (1), (2), and (3). In our testing, we assigned the same p for all graph edges. Missing values for DIM in Fig. 3 (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, 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 the 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 NoSingles great advantage in space performance: NoSingles does not keep the lightest-weight sketches, *e.g.*, single-vertex sketches, while DIM keeps them all. 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 Fig. 3 (2) and (3) for $k = 25, 50$.

RunTime. RunTime consists, mostly, of the time for building the hypergraph and the time for calculating the seeds. Fig. 3 (4), (5), and (6) shows that NoSingles spends overall less time for calculating IM solutions, for all k. The main reason for this advantage is parallel processing (8 cores) of sampling employed by NoSingles. It also helps that NoSingles hypergraph is much smaller, and not much time is spent by memory manager for allocating additional space as hypergraph grows.

Seeds Time. Fig. 3 (7), (8), and (9) shows that the time for calculating seeds is much lower for DIM than for NoSingles. This happens because of different format of each hyperedge in the hypergraph: DIM creates Borgs *et al.* hypergraph {vertex: sketch IDs}, and NoSingles builds NoSingles hypergraph {sketch: vertex IDs}. DIM evaluates the marginal influence as the degree of the vertex' hyperedge, and can quickly update the hypergraph using Lazy Greedy acceleration [15]. NoSingles has to scan the whole hypergraph after each computed seed, and cannot use Lazy Greedy technique for update.

6.1.2 *NoSingles* vs. *D-SSA*.

Fig. 4 shows testing results when NoSingles was compared to D-SSA ([18]) while processing IM for graph *cnr*2000 - a medium-size web graph with 326K vertices and 3.2M edges.

Both algorithms use RIS method, but different techniques for calculating the hypergraph weight to ensure the approximation guarantee. The weight of the hypergraph is the number of edges considered by the algorithm. While NoSingles uses a pre-calculated weight and builds the hypergraph till 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 6.1.1.

This is how we overcame the challenge of fair comparison between NoSingles and D-SSA: We tested D-SSA with different parameters, and recorded the weight of its hypergraph and the memory used for its storage. Then we tuned the parameters for NoSingles in such a way that the NoSingles hypergraph weight would be close (within +20%) to D-SSA hypergraph weight. In all the tests, we picked up the higher weight for NoSingles giving some advantage to D-SSA. In the charts presented on Fig. 4, 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.

Charts in Fig. 4 demonstrate that for all hypergraph weights, the space consumed by NoSingles is orders of magnitude smaller than D-SSA required space: roughly, 35 times smaller for $p = 0.1$, 300 times smaller for $p = 0.01$, and 3000 times smaller for $p = 0.001$. Charts show that NoSingles space advantage grows with lower p.

Testing D-SSA algorithm clearly shows that NoSingles space advantage does not depend on a particular method used for calculating the weight of the hypergraph. For any hypergraph weight, NoSingles lowers the required space to store the hypergraph to the orders of magnitude smaller value.

6.2 NoSingles *vs.* NoSinglesTopNodes performance

NoSinglesTopNodes design (subsection 5.4) is aiming at cutting down the time for seeds calculation. NoSingles hypergraph format, {sketch: vertex 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 {vertex: sketch IDs}, which makes it possible to use Lazy Greedy acceleration [13, 15] and significantly speed up the seeds calculation. We ran NoSingles and NoSinglesTopNodes algorithms on five graphs (Table 5) and compared their time performance. We picked up the graphs of different types, with different density and vastly different structures: for example, amazon2008 depicts similarities between books, while in2004 is a web graph of the `.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 density. Fig. 5 shows the total time for completion, and Fig. 6 shows the seeds calculation time, 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

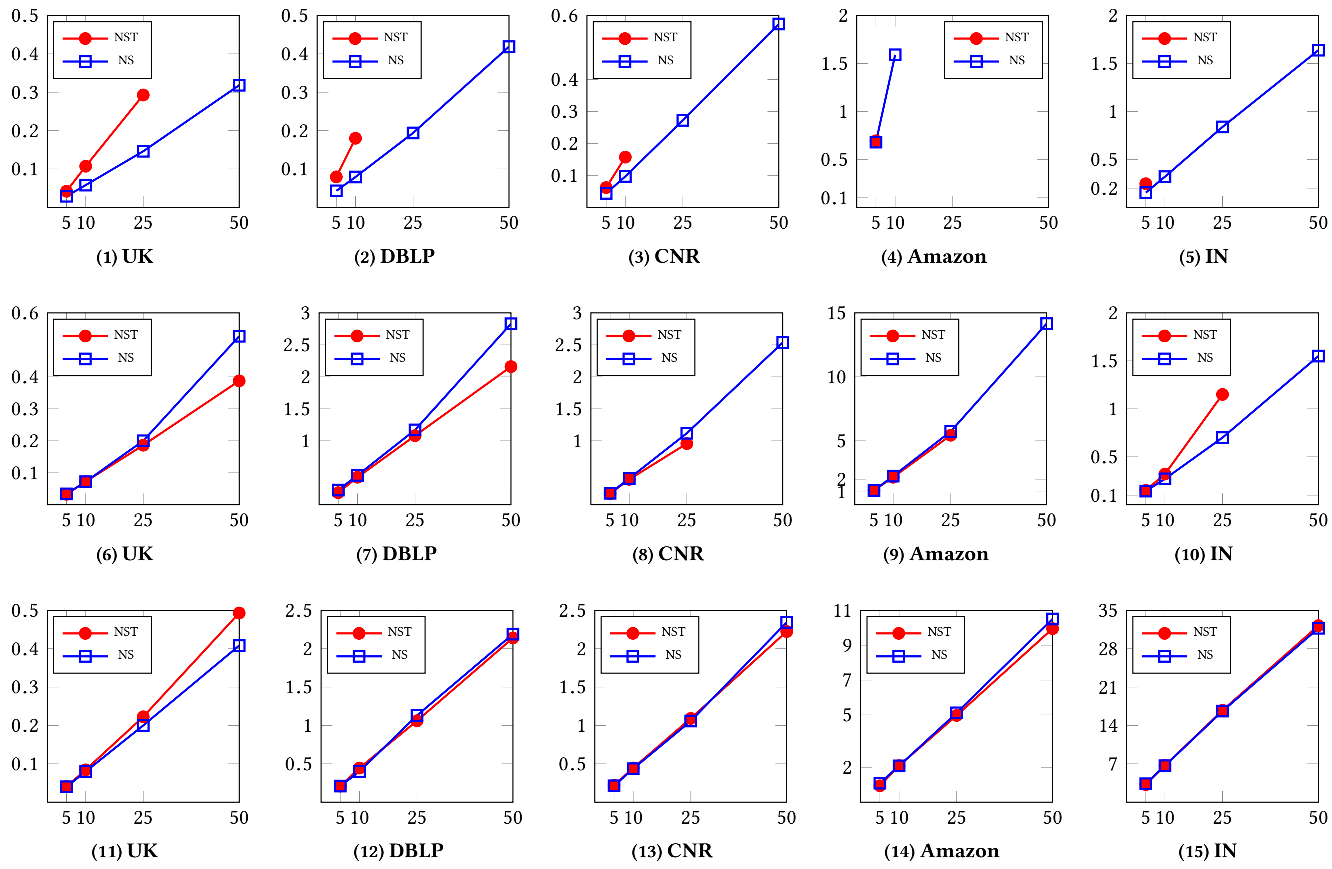


Figure 5: RunTime (hrs) NoSingles *vs.* NoSinglesTopNodes; 1st row $p = 0.1$, 2nd $p = 0.01$, 3rd $p = 0.001$.

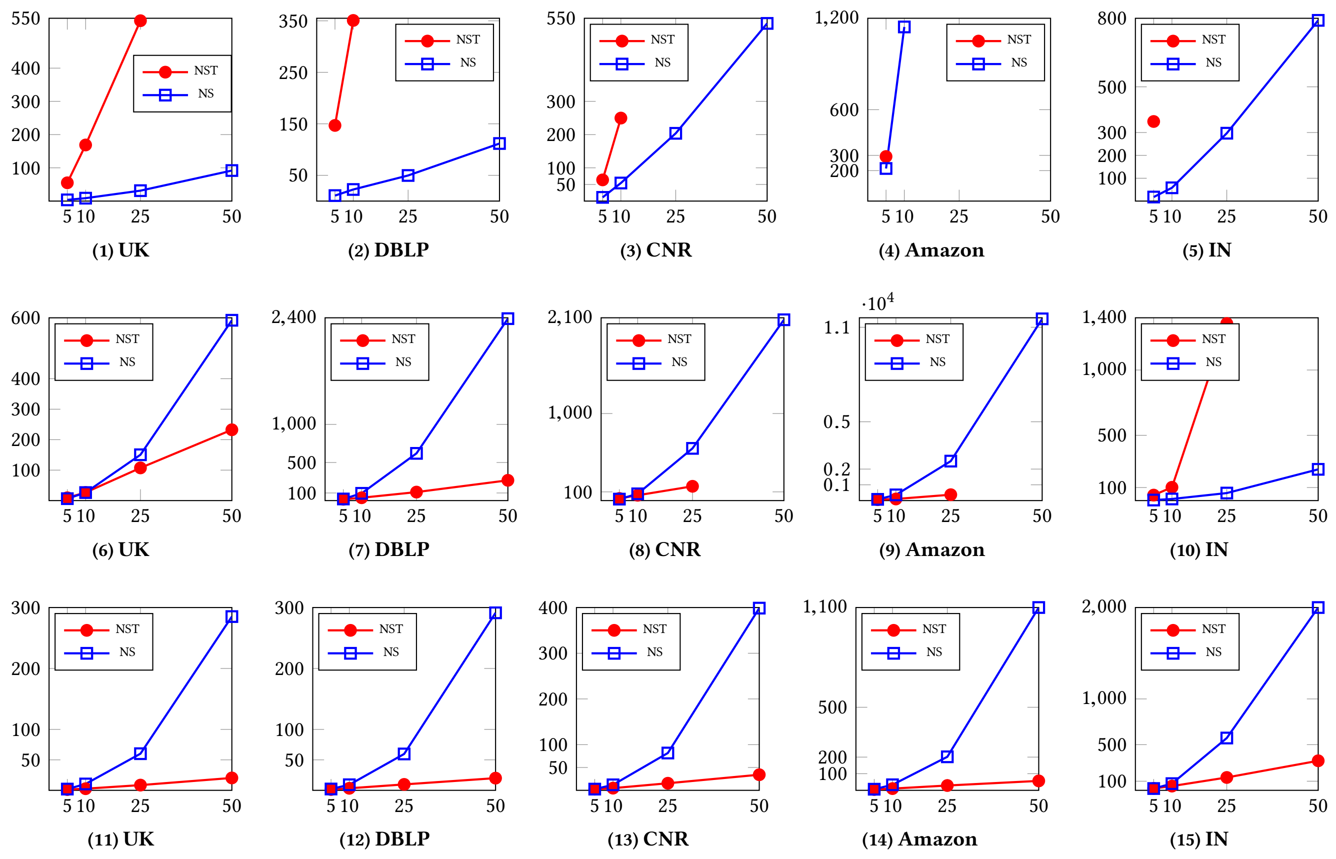


Figure 6: Seeds Time (sec) NoSingles *vs.* NoSinglesTopNodes; 1st row $p = 0.1$, 2nd $p = 0.01$, 3rd $p = 0.001$.

NoSingles are shown in charts 5.4 and 6.4. In these charts for amazon2008, the NoSingles values are missing for $k = 25$ and $k = 50$. amazon2008 structure proved rather difficult for IM calculation: the average degree of vertices is low and most vertices have the same degree. The graph appears almost homogeneous, which is natural for the depiction of similarities between books. For our algorithms, it means to process very large hypergraphs with most sketches containing $2-3$ vertices. 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 number of seeds (and, correspondingly, smaller hypergraphs). From the presented 60 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). In Fig. 5, 6, the first row shows results for $p = 0.1$, the second row – for $p = 0.01$, and the third row – for $p = 0.001$. The value of p proved to significantly impact the test result: the lower p makes the sketches contain fewer vertices, 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 seeds calculation. The explanation for this effect is the growing number of single-vertex sketches that we do not save. It is true for both algorithms, NoSingles and NoSinglesTopNodes.

Time performance. The motivation for NoSinglesTopNodes development was to significantly decrease the seed calculation time, especially when we need a large number of seeds. **a.** The tests show that the best time performance for NoSinglesTopNodes compared to NoSingles is achieved for larger k. If we follow a chart in Fig. 6, *e.g.* Fig. 6(7), we see how the advantage of NoSinglesTopNodes grows with the growing k. The same effect is observed in any chart of the last row. **b.** However, when $p = 0.1$, the effect of using NoSinglesTopNodes is negative: the overhead of building Map containing the transpose hypergraph exceeds the possible gain in speed for seed calculation. In most tests with this p we saw the Map being re-build several times (for the updated top 20% vertices), and this process slows down the seed calculation to the point of making NoSinglesTopNodes useless.

Bottomline. We recommend to use NoSinglesTopNodes algorithm when you need large number of seeds calculated for a social network with low probability of information diffusion (for example, 0.001). You also need sufficient memory on your machine for building the partial hypergraph in the format {vertex: sketch IDs}.

6.3 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 ([4, Theorem 3.1]). Parameters, intermediate values, and results are presented in Tables 6 – 8, where n is the number of graph vertices, m – the number of edges, ϵ – the allowed error, p – the edge existence probability, k – the number of seeds to compute, R – the calculated weight of the hypergraph, sk – an abbreviation for "sketches", H – an abbreviation for "hypergraph".

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

Table 6: Parameters.

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

Table 7: Intermediate results.

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

Table 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 vertices and hundreds of millions of edges was processed by a customer-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 the main memory and re-processed, for example, for evaluating the information spread for a set of given seeds.

7 CONCLUSIONS

For decades, researchers have been chipping away, step by step, from the unsurmountable complexity, both time and space, of the influence maximization problem. We presented NoSingles - a novel algorithm for computing influence estimation and influence maximization on large graphs. With this algorithm, and its variant NoSinglesTopNodes, we were able to compute influence maximization in large graphs using much smaller space than the existing state-of-the-art algorithms, while preserving the Borgs et. al. theoretical guarantee of the approximation. Furthermore, the NoSingles data structure is saved on external storage, and as such, can be used repeatedly for playing out "what if" scenarios. We presented extensive experiments, comparing our algorithm versus state-of-the-art and achieve drastic improvement in space consumption, making possible to compute influence maximization even on a consumer-grade laptop.

The source code for this paper can be found at: https://github.com/dianapopova/InfluenceMax

REFERENCES

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] D. Popova, A. Khot, and A. Thomo. Data structures for efficient computation of influence maximization and influence estimation. *CoRR*, abs/1602.05240, 2017.

[22] 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.

[23] 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.

[24] 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.