Fast and Error-Adaptive Influence Maximization based on Count-Distinct Sketches

Gökhan Göktürk and Kamer Kaya

Abstract—In this work, we describe a fast, error-adaptive approach that leverages Count-Distinct sketches and hash-based fused sampling to estimate the number of influenced vertices throughout a diffusion. We use per-vertex Flajolet-Martin sketches where each sketch corresponds to a sampled subgraph. To efficiently simulate the diffusions, the reach-set cardinalities of a single vertex are stored in memory in a consecutive fashion. This allows the proposed algorithm to estimate the number of influenced vertices in a single step for simulations at once. For a faster IM kernel, we rebuild the sketches in parallel only after observing estimation errors above a given threshold. Our experimental results show that the proposed algorithm yields comparable seed sets while being up to $3,337\times$ faster than a state-of-the-art, high-quality influence maximization algorithm. In addition, it is up to $63\times$ faster than a sketch-based approach while producing seed sets with 2%–10% better influence scores.

1 Introduction

Efficient information/influence dissemination in a network is an important research area with several applications in various fields, such as viral marketing [20], [29], social media analysis [24], [30], and recommendation systems [22]. As the study of these networks is imperative for educational, political, economic, and social purposes, a high-quality seed set to initiate the diffusion may have vital importance. Furthermore, since the diffusion analysis may be time-critical, or increasing the influence coverage may be too expensive, novel and efficient approaches to find good vertex sets that propagate the information effectively are essential.

Influence maximization is the problem of finding a subset $S \subset V$ of K vertices in a graph $G = (V, E)$ with the vertex set V and edge set E such that S reaches the maximum reachability, i.e., influences the maximum expected number of vertices, under some diffusion model. Kempe et al. [16] introduced the IM problem, proved it to be NP-hard, and provided a greedy Monte-Carlo approach that has a constant approximation ratio over the optimal solution. This greedy approach is one of the most frequently applied algorithms for IM. The time complexity of the greedy algorithm, with an influence score estimate σ, running R simulations, and selecting K seed vertices is $\mathcal{O}(KRn\sigma)$ for a graph with n vertices. Although they perform well in terms of seed-set quality, the greedy Monte-Carlo solutions are impractical for real-life networks featuring millions of vertices as a consequence of their expensive simulation costs. Due to this reason, many heuristics and proxy methods have been proposed in the literature [3], [4], [5], [6], [8], [12], [13], [15], [17], [18], [21], [25].

• *G. Göktürk and K. Kaya are with Computer Science and Engineering, Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey.*

Simulating a greedy algorithm in parallel is a straightforward workaround to reduce the execution time of IM kernels and make them scalable for large-scale networks. However, for large networks, a parallel, greedy approach with a good approximation guarantee does not come cheap on networks with billions of vertices and edges even if a large number of processing units/cores are available. Following similar attempts in the literature, we propose a parallel, sketch-based approach that approximates the Monte-Carlo processes. To boost the performance, the proposed approach does not exactly count the number of influenced vertices. Instead, it leverages Count-Distinct sketches. Below is a summary of our contributions:

- We propose HyperFuseR[1], an open-source, blazing-fast, sketch-based and accurate Influence Maximization algorithm. The proposed scheme samples the edges as they are traversed across several simulations. Thus, sampling, diffusion, and count-distinct processes are fused for all simulations.
- Running concurrent simulations on per-vertex Count-Distinct sketches reduces the number of memory accesses which is the main bottleneck for many graph kernels in the literature. While traversing an edge, HyperFuseR concurrently performs multiple diffusion simulations while using only a single (8-bit) value per vertex for each simulation.
- HyperFuseR can process large-scale graphs with millions of vertices and hundreds of millions of edges under a minute without compromising the quality of results. Furthermore, the performance scales near linearly with the number of threads available. In addition, while processing a large-scale graph, only a few GBs of memory is used, where most of the memory is spent for storing the graph itself.
- Once it is read from the memory, HyperFuseR processes all samples of a single edge together. The suggested approach, therefore, decreases the pressure on the memory subsystem. Furthermore, it employs vector compute units

1. https://github.com/ggokturk/infuser

to its near maximum efficiency to regularize memory accesses.

- We evaluate the runtime performance, memory consumption, and influence score of sketch- and approximation-based state-of-the-art influence maximization algorithms, namely SKIM [8], SSA [27], TIM+ [28] and IMM [23], to accurately position the performance of HYPERFUSER within the IM literature. The experiments show that HYPERFUSER can be 63× and 3337× faster than a state-of-the-art sketch-based and high-quality approximation algorithm, respectively while reaching the influence quality of the accurate algorithms with less memory.

2 NOTATION AND BACKGROUND

Let $G = (V, E)$ be a directed graph where the n vertices in V represent the agents, and m edges in E represent the relations among them. An edge $(u, v) \in E$ is an *incoming* edge for v and an *outgoing* edge of u. The *incoming* neighborhood of a vertex $v \in V$ is denoted as $\Gamma^-_G(v) = \{u : (u, v) \in E\}$. Similarly, the *outgoing* neighborhood of a vertex $v \in V$ is denoted as $\Gamma^+_G(v) = \{u : (v, u) \in E\}$. A graph $G' = (V', E')$ is a sub-graph of G if $V' \subseteq V$ and $E' \subseteq E$. The diffusion probability on the edge $(u, v) \in G$ is noted as $w_{u,v}$, where $w_{u,v}$ can be determined either by the diffusion model or according to the strength of u and v's relationship.

TABLE 1
TABLE OF NOTATIONS

Variable	Definition
$G = (V, E)$	Graph G with vertices V and edges E
N	Number of vertices
M	Number of edges
T	Depth of the graph
$\Gamma^+_G(u)$	The set of vertices v where $(u, v) \in E$
$\Gamma^-_G(u)$	The set of vertices v where $(v, u) \in E$
$w_{u,v}$	Probability of u directly influencing v
$R_G(v)$	Reachability set of vertex v on graph G
S	Seed set to maximize influence
K	Size of the seed set
$\mathcal{J}$	Number of Monte-Carlo simulations performed
$\sigma_G(S)$	The influence score of S in G, i.e., expected number of vertices reached from S in G
$w_{u,v}$	Sampling probability for the edge (u, v)
$P(s, v)_r$	Random probability generated for selecting edge vertices s to v in simulation r
$h(u, v)$	Hash function for edge $\{u, v\}$
h_{max}	Maximum value hash function h can return
e	Estimated reachability set size
$M_u[j]$	jth sketch register for vertex u
ς	Influence gained before last sketch build
σ	Influence Score
δ	Marginal gain after last sketch build
err_l	Local estimation error of the sketch
err_g	Global estimation error of the sketch
ϵ_g	Global estimation error threshold
ϵ_l	Local estimation error threshold
ϵ_c	Non-convergenced vertex threshold

2.1 Influence Maximization

Influence Maximization aims to find a seed set $S \subseteq V$ among all possible size K subsets of V that maximizes an *influence spread function* σ when the diffusion process is initiated from S. In the literature, *independent* and *weighted cascade* (IC and WC), and *linear threshold* (LT) [16] are three widely recognized diffusion models for IM.

The **Independent Cascade** model works in rounds and activates a vertex v in the current round if one of v's incoming edges (u, v) is used during the diffusion round, which happens with the activation probability $w_{u,v}$, given that u has already been influenced in the previous rounds. The activation probabilities are independent (from each other and previous activations) in the *independent cascade* model, which we focus on in this paper. A toy graph with activation probabilities on the edges is shown in Figure 1. The complexity analysis stays consistent for many diffusion models, including *Independent Cascade*, *Weighted Cascade*, and *Linear Threshold* models; the time complexity of the greedy algorithm, estimating the σ influence score, running R simulations, and selecting K seed vertices is $\mathcal{O}(KRn\sigma)$ for a graph with n vertices. We concentrate on the IC model in this paper, but although their adaptation requires some work, the proposed methods are also relevant to other models in the literature.

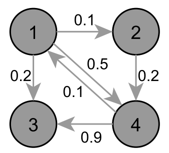


Fig. 1. The directed graph $G = (V, E)$ for IC with independent diffusion probabilities.

2.2 Count-Distinct Sketches

The *distinct element count* problem focuses on finding the number of distinct elements in a stream where the elements are coming from a universal set $\mathcal{U}$. Finding the number of vertices to be influenced of a candidate seed vertex u, i.e., the cardinality of u's *reachability set*, is a similar problem. For each sample subgraph, the number of visited vertices is found while traversing the subgraphs starting from u. Note that an exact, linear-time computation of stream cardinality requires memory proportional to the universal set cardinality, i.e., $\Theta(|\mathcal{U}|)$.

The reachability set of a vertex is the union of all its connected vertices (via outgoing edges). Many IM kernels exploit this property to some degree. The methods based on *reverse reachability* [2] utilize this property directly to merge the reachability sets of connected vertices to estimate the number of vertices influenced. *MixGreedy* [4] goes one step further; it utilizes the fact that for an undirected graph, all vertices in a connected component have the same reachability set. Therefore, all the reachability sets within a single sample subgraph can be found via a single graph traversal.

For directed graphs, storing reachability sets for all vertices and merging these sets are infeasible for nontrivial graphs. If one-hot vectors are used to store the reachability sets for constant insertion time, $\Theta(n^2\mathcal{R})$ bits of memory is required where each merge operation has $\Theta(n)$ time complexity. If disjoints sets are used for storing reachability sets; $\mathcal{O}(n\sigma\mathcal{R})$ memory is required to store all reachability sets, and each merge operation has $\mathcal{O}(\mathtt{Ack}(\sigma))$ complexity where $\mathtt{Ack}$ is the Ackermann [1] function.

Count-Distinct Sketches can be leveraged to estimate a reachability set's cardinality efficiently; for instance, the Flajolet–Martin (FM) sketch [11] uses a constant number, $\mathcal{J}$, of registers where $\mathcal{J}$ can be increased for better estimates.

Furthermore, the union of two sketches can be computed in constant time which is a necessary property for an efficient estimation of reachability set cardinalities. The essence of an FM sketch is that it keeps the track of how rare the elements are in a stream. The rarity of the stream is estimated by counting the maximum number of leading zeros in the stream elements' hash values. Initially, each register is initialized with zero. The items are hashed one by one, and the length of the longest all-zero prefix is stored in the register. With a single register, the cardinality estimation can be done by computing the power 2^r where r is the value in the register.

In practice, multiple, $\mathcal{J}$, registers and hash values, $M[j]$ and h_j, $1 \leq j \leq \mathcal{J}$, are used to reduce the variance of the estimation. For a sketch with multiple registers, the impact of adding an item $x \in \mathcal{U}$ is shown in (1):

$$M[j] = \max(M[j], clz(h_j(x)), \ 1 \leq j \leq \mathcal{J} \tag{1}$$

where $clz(y)$ returns the number of leading zeros in y and $\mathcal{J}$ is the number of sketch registers. With multiple registers, the average of the register values can be used to estimate the cardinality, and the result is divided to a correction factor $\phi \approx 0.77351$ to fix the error due to hash collisions. That is the estimated cardinality e is computed as

$$e = 2^{\bar{M}}/\phi \tag{2}$$

where $\bar{M} = \texttt{avg}_j\{M[j]\}$ is the mean of the register values.

In this work, we utilize a variant of Flajolet–Martin sketch; since multiple Monte-Carlo simulations are performed to calculate the estimated influence, we use one register per simulation and take the average length of the longest leading zeros. Two given FM sketches M_u and M_v can be merged, i.e., their union M_{uv} can be computed by taking the pairwise maximums of their registers. Formally;

$$M_{uv}[j] = \max(M_u[j], M_v[j]), \ 1 \leq j \leq \mathcal{J}. \tag{3}$$

In our implementation, the merge operations are performed if and only if there is a sampled edge between the vertices.

3 Error-Adaptive Influence Maximization

HyperFuser uses *directed* fused sampling, utilizes *sketches* for both influence estimation and seed selection, and a novel error-adaptive sketch rebuilding strategy. On the other hand, InFuseR [14] employs *undirected* hash-based fused sampling and CELF optimizations for seed selection.

Most IM algorithms have the same few steps to find the best seed vertex set; sampling, building the influence oracle, verifying the impact of new candidates, and removing the latent seed set's residual reachability set. Following the idea proposed in [14], HyperFuser fuses the sampling step with other steps to avoid reading the graph multiple times.

Throughout the process, each register is used only for a single sample/simulation. For an edge (u, v) in simulation j where v is a live vertex, an update, i.e., a merge operation, on u's register is performed on the corresponding register $M_u[j]$, i.e., $M_u[j] = max(M_u[j], M_v[j])$. The toy example in Fig. 2 shows how the sketch operations are done on a toy problem. In Fig. 2(a), the sketches, in a toy problem with 4 samples and 4-bit hash functions, are initialized; the id of vertex u is hashed and leading zeros of the hash are written to registers. In Fig. 2(a), the merge operation is illustrated on registers of vertex u, M_u, and vertex v, M_v; merge operation takes element-wise maximum of registers to find union of the reachability sets. Fig. 2 shows writing the merged reachability sets back to registers w.r.t sampling using a conditional move operation.

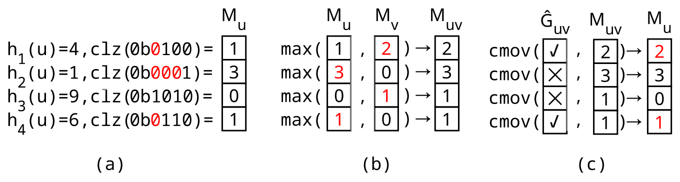


Fig. 2. (a) Toy sketch initialization of vertex x with four registers and 4-bit hash functions. (b) Example of union operation on sketches of vertices x and y with four registers. (c) Diffusion operation from vertex v to vertex u w.r.t sampling in $\hat{G}$.

That is, at each iteration, vertices (outgoing) neighbors' reachability sets are added to their sketches. This recursive formulation of the influence iteratively relays the reachability information among the vertices, allowing us to estimate the marginal influence for all vertices very fast.

After estimating the reachability set cardinalities, HyperFuser picks the vertex v with the largest cardinality by evaluating the sketches. Then it finds the (actual) reachability set of the latent seed set, which is the union of the reachability sets of v and the vertices in the seed set, by performing Monte-Carlo simulations. The vertices in this reachability set are removed from the live set L. Hence, in later iterations, these vertices will not contribute to the marginal gain. Finally, the algorithm checks if rebuilding is necessary for the sketches based on the difference between the sketch estimate and Monte-Carlo estimate. The proposed approach and the algorithm will be described in this section in more detail starting with fused sampling.

3.1 Hash-based Fused Sampling

The probabilistic nature of cascade models requires sampling subgraphs from $G = (V, E)$ to simulate the diffusion process. If performed individually as a preprocessing step, as the literature traditionally does, sampling can be an expensive stage, furthermore, a time-wise dominating one for the overall IM kernel. That is first, sampling multiple sub-graphs may demand multiple passes on the graph, which can be very large and expensive to stream to the computational cores, and second, if samples are memoized, the memory requirement can be a multiple of the graph size. In this work, we borrow the fused-sampling technique from InFuseR [14] which eliminates the necessity of creation and storage of the sample subgraphs in memory. In Fig. 3, we illustrate fused-sampling; instead of processing the samples independently as in Fig. 3a, fused-sampling processes each edge concurrently for multiple simulations as shown in Fig. 3b. This allows us to process each edge only a few times instead of once per simulation.

In HyperFuser, when an edge of the original graph is being processed, it is processed for all possible samples. Then, it is decided to be *sampled* or *skipped* depending on the outcome of the hash-based random value for each sample.

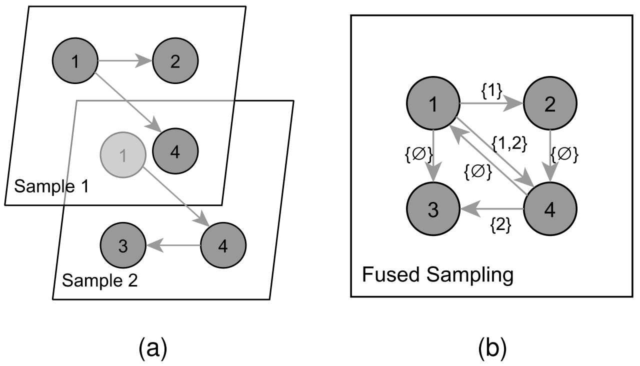


Fig. 3. (a) Two sampled subgraphs of the toy graph from Figure 1 with 4 vertices and 6 edges. (b) The simulations performed are fused with sampling. Each edge is labeled with the corresponding sample/simulation IDs.

Given a graph $G = (V, E)$, for an edge $(u, v) \in E$, the hash function used is given below:

$$h(u, v) = \textsc{Hash}(u||v) \; mod \; 2^{31} \tag{4}$$

where $||$ is the concatenation operator. An important bottleneck of hash-based fused sampling is the need of the hash value for an edge whenever it is read from the memory. To mitigate its overhead, using $\Theta(|E|)$ more memory, we precompute and memoize the hash values (and read them when we need them). We have also tried recomputing the hash values every time they are needed; this incurred around 5% slowdown. Since HyperFuseR is designed for multi-core servers, extra $\Theta(|E|)$ memory is a practical overhead. However, considering its low overhead, the recomputation strategy can be promising for memory-restricted accelerators and devices such as GPUs.

A set of uniformly randomly chosen numbers $X_r \in_R [0, h_{max}]$ associated with each sample r are generated for each edge to generate a unique sampling probabilities for all samples. Then, the sampling probability of (u, v) for simulation r, $P(u, v)_r$, is computed: the hash value, $h(u, v)$, is XOR'ed with X_r and the result is normalized by dividing the value to the upper limit of the hash value h_{max}. Formally,

$$P(u, v)_r = \frac{X_r \oplus h(u, v)}{h_{max}}. \tag{5}$$

The edge (u, v) exists in the sample r if and only if $P(u, v)_r$ is smaller than the edge threshold $w_{u,v}$. One of this approach's benefits is that an edge can be sampled using a single XOR and compare-greater-than operation which is crucial since HyperFuseR recomputes this probability value each time the corresponding edge is read from the memory. In practice, we do not apply the division operation since h_{max} is a constant value and the same for every edge and simulation. Moreover, the corresponding control flow branch overhead can be removed using *conditional move* instructions.

3.2 Estimating Reachability Set Cardinality

A greedy solution to IM requires finding a vertex with a maximum marginal influence at each step until the seed set size reaches K. An exact computation of marginal influences, i.e., the additional influence when a vertex v is added to the seed set, requires the computation of all candidate vertices' reachability sets within all the samples. This involves many graph traversals and is expensive even with various algorithmic optimizations and a scalable parallelized implementation, e.g., see [14]. In this work, we pursue the idea of using Count-Distinct sketches to estimate marginal influence scores. Here we propose an efficient and effective IM algorithm, HyperFuseR, that utilizes Flajolet–Martin sketches described in Section 2.2 to estimate the averages of distinct elements in the sampled subgraphs. Algorithm 1 shows the steps taken by HyperFuseR.

Algorithm 1 HyperFuseR$(G, K, \mathcal{J})$

```
Input: $G = (V, E)$: the influence graph
       $K$: number of seed vertices
       $\mathcal{J}$: number of Monte-Carlo simulations
Output: $S$: a seed set that maximizes the influence on $G$
 1: $S \leftarrow \{\emptyset\}$
 2: for $v \in V$ do in parallel
 3:     for $j \in \{1, \ldots, \mathcal{J}\}$ do
 4:         $M_v[j] \leftarrow clz(hash(v) \oplus hash(j))$
 5: $M \leftarrow$Simulate$(G, M, \mathcal{J}, \emptyset)$
 6: $M_{S'} \leftarrow zeros(\mathcal{J})$
 7: $\varsigma \leftarrow 0$
 8: for $k = 1 \ldots K$ do
 9:     $s \leftarrow \operatorname{argmax}_{v \in V}\{$Estimate$($Merge$(M_{S'}, M_v))\}$
10:     $S \leftarrow S \cup \{s\}$
11:     $e \leftarrow$Estimate$($Merge$(M_{S'}, M_s))$
12:     $R_G(S) \leftarrow$ reachability set of $S$ (for all simulations)
13:     $\sigma \leftarrow$ Monte-Carlo-based (actual) influence of $S$
14:     $\delta = \sigma - \varsigma$
15:     $err_l = |(e - \delta)/\delta|$
16:     $err_g = |(e - \delta)/\sigma|$
17:     if $err_l < \epsilon_l \vee err_g < \epsilon_g$ then
18:         $M_{S'} \leftarrow$ Merge$(M_{S'}, M_s)$
19:     else
20:         for $v \in V$ do in parallel
21:             for $j \in \{1, \ldots, \mathcal{J}\}$ do
22:                 $M_v[j] \leftarrow clz(hash(v) \oplus hash(j))$
23:         $M \leftarrow$Simulate$(G, M, \mathcal{J}, R_G(S))$
24:         $M_{S'} \leftarrow zeros(\mathcal{J})$
25:         $\varsigma \leftarrow \sigma$
26: return $S$
```

Algorithm 1 first initializes the reachability sets of all vertices by adding the vertices themselves. That is for all vertices u, its jth register is set to $M_u[j] = clz(h_j(u))$ meaning $R_{G_j}(u) = \{u\}$ where G_j is the jth sampled graph. Then, we perform the diffusion process on the sketch registers whose pseudocode is given in Algorithm 2. The diffusion starts by adding all the vertices to the *live vertex set* L. Then at each step, the incoming edges of the live vertices are processed. For a vertex u, its sketch, M_u, is updated by merging the sketches M_v of all live outgoing neighbors vertices $v \in L \cap \Gamma^+_G(u)$. For each such vertex v and simulation j, the operation $M_u[j] = max(M_u[j], M_v[j])$ is performed. This approach can be seen as a bottom-up, i.e., reversed, diffusion process where at each iteration, the cardinality information is pulled from vertices neighbors.

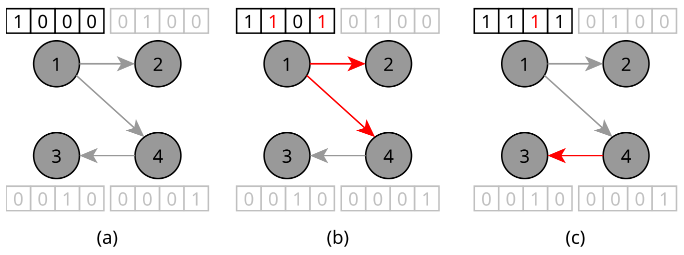


Fig. 4. InFuseR diffusion process illustrated for only vertex 1 on toy sample graph. (a) Toy graph InFuseR initialization. (b) First level of diffusion using *push-based* method for vertex 1. (c) Second level of diffusion following Fig.4(a).

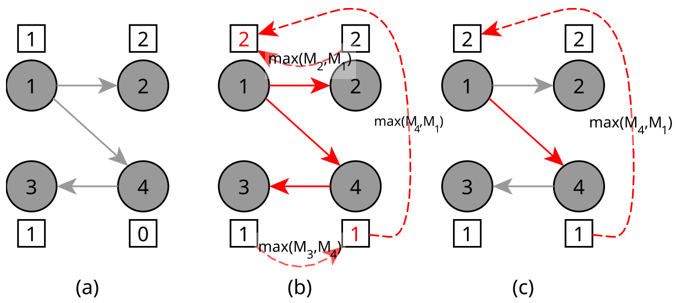


Fig. 5. HYPERFUSER diffusion process illustrated for *all* vertices on toy sample graph. (a) Toy sketch initialization. (b) First level of diffusion using *Pull-based* method for *all* vertices. (c) Second level diffusion following Fig.5(a), only updated vertices' sketches are pulled.

If any of u's sketch registers changes during this operation, it is added to the live vertex set L' of the next iteration. Once the incoming edges of all live vertices are processed, the iteration ends.

Algorithm 2 SIMULATE$(G, M, \mathcal{J}, R_S)$

```
Input: $G = (V, E)$: the influence graph
       $M$: sketch vectors of vertices
       $\mathcal{J}$: number of MC simulations
       $R_S$: reachability set of the seed set
Output: $M$: updated Sketch vectors
 1: $L \leftarrow V$
 2: $L' \leftarrow \emptyset$
 3: while $|L|/|V| > \epsilon_c$ do
 4:     for $u \in \Gamma(L)$ do in parallel
 5:         for $e_{u,v} \in A(u)$ do
 6:             for $j \in (0, \mathcal{J}]$ do
 7:                 if $P(u,v)_j < w_{u,v} \wedge u \notin R_S[j]$ then
 8:                     $M_u[j] \leftarrow max(M_u[j], M_v[j])$
 9:             if $M_u$ changed then
10:                 $L' \leftarrow L' \cup u$
11:     $L \leftarrow L'$
12:     $L' \leftarrow \{\emptyset\}$
13: return $M$
```

The traditional Greedy algorithm [16] processes the simulations one-by-one and computes the vertices' reachability sets. On the other hand, HYPERFUSER efficiently performs multiple simulations at once in a single-step iteration (which processes distance-1 neighborhoods). Since each iteration relays one level of cardinality information, this step requires at most d iterations where d is the diameter of G. Note that when processed individually (as the Greedy algorithm), the jth simulation over the sampled subgraph G_j would require only at most d_j iterations, which is the diameter of G_j, and probably much smaller than d considering the sampled subgraphs being highly sparse. However, d is also a loose upper bound for HYPERFUSER. A better one is $\max\{d_j : 1 \leq j \leq \mathcal{J}\} + 1$ where d_j is the diameter of G_j. We have observed that during these simulations later iterations have only a minor impact on the marginal influence scores. Hence, to further reduce the overhead of concurrent simulation processing and avoid bottleneck simulations due to remaining perimeter structures (e.g., paths on the perimeter), we employ an early-exit threshold ϵ_c over the remaining live vertices ratio, which is expected to be very small when only one or two (bottleneck) simulations remain. That is if $|L'| \leq |V| \times \epsilon_c$ the diffusion process in Algorithm 2 stops. Otherwise, L is set to L', L' is cleared, and the next iteration starts. We used $\epsilon_c = 0.02$ to make HYPERFUSER faster while keeping its quality almost the same. Figure 5 shows a toy problem diffusion process on 4 vertices and a single sample, the process initializes sketches using respective hashes' leading zeros and bigger numbers are pulled from all recently changed neighbors. Whereas Infuser works vertices one by one, labeling visited vertices for the processed vertex. The process is repeated for multiple vertices until a seed vertex is found. Figure 4 shows an example in the same settings as Figure 5 to emphasize the difference between the two algorithms.

After the diffusion process, the following steps are repeated until K vertices are added to the seed set S. First, for each $v \in V$, the cardinality of the reachability set, $R_G(S \cup \{v\})$, is estimated by merging $M_{S'}$ and v's sketch registers where $M_{S'}$ is the set of sketch registers for the seed set S used to estimate the number of already influenced vertices by S[2]. Before the kernel, these registers are initialized with zeros. Second, a vertex s with the maximum cardinality estimation is selected and added to S. Third, the actual simulations are performed to compute the reachability set of S. Having an actual $R_G(S)$ allows us to calculate the estimation errors and find the blocked vertices for all simulations, which is vital since these blocked vertices can be skipped during the next diffusion steps. Besides, we leverage the actual influence to have an *error-adaptive kernel*, i.e., to compute the actual sketch error and rebuild the sketches when the accumulated error reaches a critical level which can deteriorate the quality for the following seed vertex selections.

3.2.1 *Error-adaptive sketch rebuilding*

Sketches are fast. However, each sketch operation, including update and merge, can decrease their estimation quality below a desired threshold. Our preliminary experiments revealed that sketches are highly competent at finding the first few seed vertices for influence maximization. Unfortunately,

2. In fact, the definition is exact only if the error-adaptive, smart sketch rebuilding is disabled. As it will be described in the following subsection, when HYPERFUSER's error-adaptive mechanism is enabled, $M_{S'}$ is periodically rebuilt to estimate the cardinality of reachability sets over the remaining, unblocked vertices. This is why S' is used instead of S.

after a few seed vertices, the values inside the sketch registers $M_{S'}$, which are updated at line 18 of Algorithm 1 via merging with new seed vertex s's reachability set, become large. The saturation of $M_{S'}$ registers is important since HYPERFUSER uses them to select the best seed candidate at line 9. When they lose their sensitivity for seed selection, a significant drop in the quality is observed.

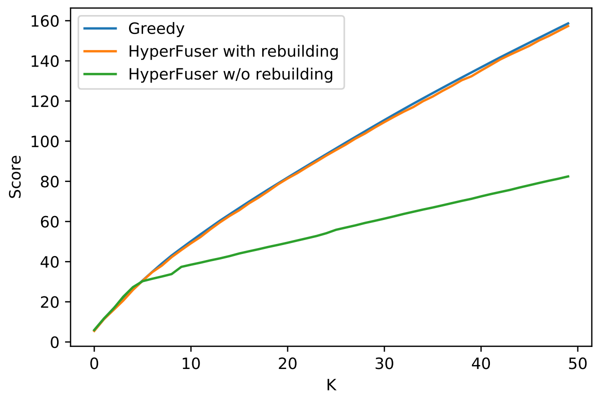


Fig. 6. Effect of register saturation on `Amazon0302` dataset using HYPERFUSER ($\mathcal{J} = 256$) without rebuilding against Greedy($\mathcal{R} = 20000$) method [16].

Figure 6 shows the effect of register saturation by comparing two HYPERFUSER variants; the first one rebuilds a new sketch to choose each seed vertex s, i.e., the **else** part in lines 20–25 of Algorithm 1 is executed for every iteration of the **for** loop at line 8. This sketch is built on the residual graph $G \setminus R_G(S)$, which remains after the current seed set's reachability is removed. The second variant builds a sketch only once at the beginning and employs it through the IM kernel, i.e., the **else** part is never executed. The figure shows that the latter's seed selection quality is comparable to that of the former for the first few seed vertices. However, a significant reduction in the quality is observed for the later vertices. Furthermore, the former approach's quality is on par with the expensive Greedy algorithm's quality, which computes actual reachability sets. This shows that sketch-based estimation can perform as well as the accurate but expensive approach. Note that rebuilding also allows HYPERFUSER to work on a smaller problem for the following seed vertex selection since we remove the already influenced vertices from sample subgraphs and work on the remaining subgraphs.

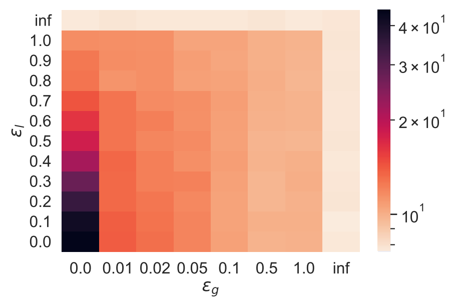
Although its quality is on par with the traditional algorithm, the variant which rebuilds a sketch for all the seed vertex selections can be expensive. Here, we leverage an error-adaptive approach by rebuilding them when a significant cardinality estimation error is observed. The estimation error is calculated as follows; we store the influence score after each sketch rebuild in ς (line 25 of Algorithm 1). Let σ be the real influence for the seed set S including the selected vertex. We first compute the marginal influence gain $\delta = \sigma - \varsigma$, which is the additional influence obtained since the last sketch rebuilt. Note that e, computed at line 11 is the sketch estimate for this value. HYPERFUSER computes the local estimation error $err_l = |(e - \delta)/\delta|$ and the global error $err_g = |(e - \delta)/\sigma|$. The sketches are assumed to be fresh if the local estimation error err_l is smaller than a local threshold ϵ_l or the global error $err_g = |(e - \delta)/\sigma|$ is smaller than a global threshold ϵ_g.

The use of two different, local and global, thresholds allows the algorithm to rebuild the sketches after significant local errors and skip this expensive process if the estimation error is insignificant compared to the total influence. As explained above, when the rebuilding is skipped, HYPERFUSER only updates $M_{S'}$ by merging it with the candidate vertex's sketch. Hence, the selected threshold values, ϵ_l and ϵ_g, have a significant impact on the performance. In the setting $\epsilon_l = \epsilon_g = 0$, the algorithm always rebuilds. Conversely, setting $\epsilon_l = \epsilon_g = \infty$ will make HYPERFUSER fast since sketches are built only once. However, the influence scores will suffer, which is already shown by Fig. 6. To evaluate the interplay and find the thresholds that yield a nice tradeoff, we conducted a grid search in which HYPERFUSER's execution time and influence quality are measured for different parameters. The results of this preliminary experiment are shown in Figure 7. We found that the parameters $\epsilon_l = 0.3$ and $\epsilon_g = 0.01$ perform well on many datasets, both in terms of speed and quality.

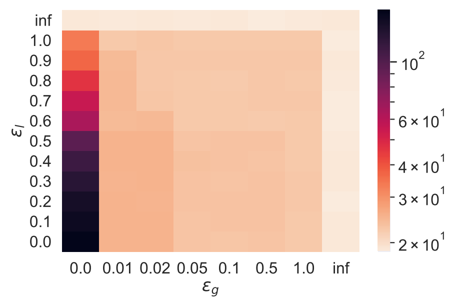
3.3 Runtime Analysis

The seed selection process requires K iterations to select K seed vertices. For a single iteration, we have multiple steps each processing a single depth of information, e.g., neighborhood. The propagation process is actually a reverse influence process that starts with whole graph as frontier and diffuses register values with respect to the sample graphs. In the worst case, i.e., assuming all edges are processed and all the sketch registers are updated at each step, each iteration of the seed-set construction has $\mathcal{O}(d \times (|V| \times \mathcal{J} + |E|))$ time complexity where d is the diameter of G which is the number of steps at each iteration. However, on average, a single iteration of HYPERFUSER performs much faster. First, as explained above, d is a loose upper bound on the number of steps. Second, since only the recently updated registers are propagated to the next step, and only the edges in a sample are processed, a small fraction of the graph is processed for each step. Last, the complexity reduces for later iterations, since the nodes and their edges, which are already covered by the reachability set of the current seed set, are not processed.

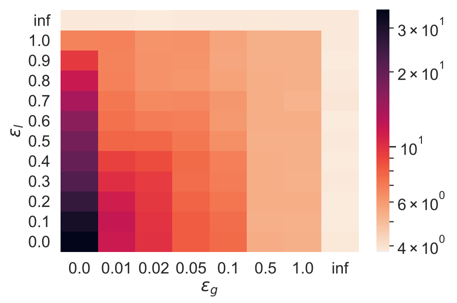
These analysis and observations tell us that when the probability of being influenced, i.e., the average probability of an edge being in a sampled subgraph, is high, the samples will be denser and hence, the actual number of steps HYPERFUSER performs will be smaller. Furthermore, depending on the topology of G, the marginal influence scores of the first few vertices can be much higher with larger probability values and adding these vertices to the seed set will leave less number of vertices to be processed for later iterations. On the other hand, increasing the influence probabilities also increase the number of edges to be processed on the samples. Note that other factors that has an impact on the overall performance exist such as the number of iterations skipping the sketch rebuilding. This also depends on G's



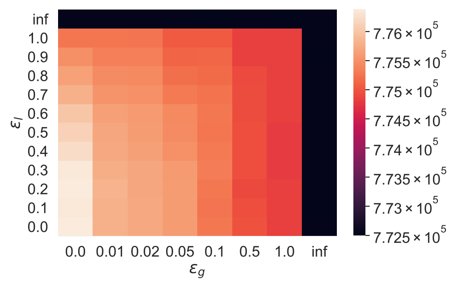
(a) `LiveJournal` execution time in seconds



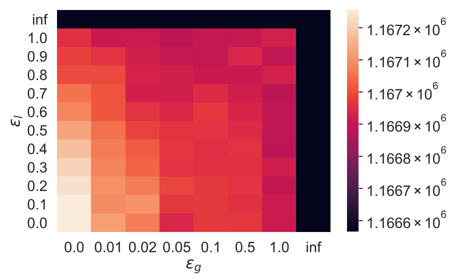
(b) `Orkut` execution time in seconds



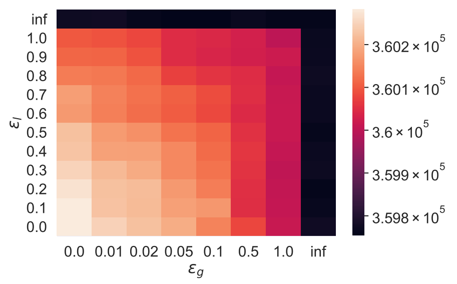
(c) `Pokec` execution time in seconds



(d) `LiveJournal` influence score



(e) `Orkut` influence score



(f) `Pokec` influence score

Fig. 7. Effect of ϵ parameters on HYPERFUSER ($\mathcal{J} = 256$, $\epsilon_c = 0.02$) performance, using $\tau = 16$ threads. Lighter shades are better.

structure as well as the influence probabilities. Still, as the experiments also will show, HYPERFUSER's execution time tends to decrease for many graphs when the influence probabilities increase.

3.4 Approximation Ratio of HYPERFUSER

The influence function is non-negative, monotone, and submodular; hence, adding a single vertex to the current seed set can only increase the overall influence and decreases the marginal influence scores for the remaining vertices that are not in the set. A greedy solution of final size K is at least $1 - (1 - 1/K)^K \approx 63.5\%$ of the optimal solution [26]. Since it uses sketches, HYPERFUSER's approximation guarantee is not straightforward to derive. This being said, based on the average approximation performance of the sketches, we can argue the following. HYPERFUSER is a greedy algorithm; vertices with a maximum reachability which is estimated via sketches. At each iteration, HYPERFUSER can choose a vertex worse than the local optimum due to its estimation errors. Since the standard error for the FM sketch is $1.30/\sqrt{\mathcal{J}}$ [10], we can expect $(1 - 1.3/\sqrt{256}) \times (1 - (1 - 1/50)^{50}) \approx 58.4\%$ of the optimal solution for $K = 50$ and $\mathcal{J} = 256$. However, as the greedy algorithm, the practical performance is better. Furthermore, the mistakes in previous iterations can be rectified in later iterations due to submodularity.

3.5 Implementation Details

To efficiently process real-life graphs, HYPERFUSER uses the Compressed Sparse Row (CSR) graph data structure. In CSR, an array, $ptrs$, holds the starting indices of the adjacency lists for each vertex, while another array, adj, holds the actual adjacency lists (i.e., the outgoing neighbors) one after another. Hence, the adjacency list of vertex i is located in adj at locations $adj[ptrs[i], \ldots, ptrs[i+1] - 1]$.

The traditional two-step (sample-then-diffuse) computation model stores the (graph) data in a loosely coupled fashion. While designing HYPERFUSER, we fine-tuned it to be vectorization friendly, including its data layout and computation patterns. These design choices allow us to perform multiple operations, i.e., the same operations but on different data, at once. For instance, we keep all the memory registers of a single vertex from different simulations adjacent, and this allows the efficient use of vectorized computation hardware while performing lines 6–8 of Algorithm 2. Also, random number generation, fused sampling, and sketch merging are vectorizable operations when the data are stored in a coupled way as in HYPERFUSER.

We allow a single easy-to-predict branch after the sample edges are generated to exit early without merging the registers. In our experiments, sorting random values X_r significantly increases the performance by clustering similar simulations together while preserving branch locality. Since sorted random values are XOR'ed with the same edge hash values, it is more likely to non-taken edges are sampled together. For example, the naive approach has only 13% vector units filled while processing Amazon dataset with 0.01 edge probabilities. Sorting the X_r values increases the fill rate 32%. This shows that, the early-exit branch is taken $2.4\times$ more, reducing the total number of arithmetic operations from 23 to 9 per edge traversal.

Multi-thread parallelization is achieved using OpenMP pragmas; the sketch computation and Monte-Carlo simulations are performed in per-vertex tasks. This approach allow us to preserve both temporal and spatial locality. The OpenMP tasks are distributed to the threads via dynamic scheduling with large enough chunks (8192) to mitigate scheduling costs. Even though dynamic scheduling has a higher cost than the static scheduling, it is preferable due to uneven edge distribution among the vertices of G (as well as of the sampled subgraphs).

HYPERFUSER computes the actual influence via an iterative processing of the vertices' neighborhoods starting with a queue containing the seed set. This approach allows multiple threads to process different vertices at the same time. The visited vertices are labeled in parallel without any synchronization since the operations are idempotent, and they can be done simultaneously in any order. In addition, we utilize SIMD vectors to process multiple samples simultaneously for each thread. This approach is wasteful for small influence graphs, but the error-adaptive rebuilding strategy allows us to mitigate the cost in such conditions.

4 EXPERIMENTAL RESULTS

We performed the experiments on a server with an 16-core `Intel Xeon E5-2620 v4` CPU, running at 2.10GHz, and 192GB memory. The Operating System on the server is `Ubuntu 20.04.2 LTS` with 5.4.0-65 kernel. The algorithms are implemented using `C++20`, and compiled with `GCC 9.3.0` with `"-Ofast"` and `"-march=native"` optimization flags. Multi-thread parallelization was achieved with `OpenMP` pragmas. `AVX2` instructions are utilized by handcrafted code with vector intrinsics.

TABLE 2
PROPERTIES OF NETWORKS USED IN THE EXPERIMENTS

	Dataset	No. of Vertices	No. of Edges	Avg. Degree	Diameter
Undirected	`Amazon0302`	262,113	1,234,878	4.71	32
	`DBLP`	317,081	1,049,867	3.31	21
	`Orkut`	3,072,441	117,185,083	38.14	9
	`Youtube`	1,134,891	2,987,625	2.63	20
Directed	`Epinions`	75,880	508,838	6.71	14
	`Friendster`	65,608,366	1,806,067,135	27.53	32
	`LiveJournal`	4,847,571	68,993,773	14.23	16
	`Pokec`	1,632,803	30,622,564	18.75	11
	`Slashdot0811`	77,360	905,468	11.70	10
	`Slashdot0902`	82,168	948,464	11.54	11

4.1 Experiment Settings

We performed the experiments on ten graphs (four undirected, six directed). For comparability, graphs that have been frequently used within the Influence Maximization literature are selected. The properties of these graphs are given in Table 2: `Amazon0302` is an Amazon co-purchase network, `DBLP` is DBLP collaboration network, `Orkut` is an online social network where users form friendship, `Youtube` is a video social video sharing platform that allows friendship relations, `Epinions` is a consumer review trust network, `Friendster` is a social gaming website where users can form friendship edge each other, `LiveJournal` is a blogging site that allows users to add friends, `Pokec` is the Slovakian poker game site friend network, and `Slashdot` networks are friend-foe networks (08-11, 09-11).

Five diffusion settings are simulated for a comprehensive experimental evaluation; for each network, we use

1) constant edge weights $w = 0.005$,
2) constant edge weights $w = 0.01$ (as in [16] and [4]),
3) constant edge weights $w = 0.1$ (as in [16]),
4) normally distributed edge weights $w \in N(0.05, 0.025)$,
5) uniform distributed edge weights $w \in U(0, 0.1)$.

We selected $w = 0.005$ as a setting to challenge HYPERFUSER. As explained in Section 3.3, due to the nature of its diffusion algorithm, HYPERFUSER is expected to perform slower on sparser samples. The later two settings, i.e., $w = 0.01$ and $w = 0.1$, are selected to emulate the experiments of [16] and [4]. In addition to constant edge weights, we performed experiments on two different statistical distributed edge weights to test how algorithms behave on mixed (large and small edge weights together) edge weights. The normal distribution setting incurs a small number of extreme sketch values and uniform distribution setting incurs a large number of large sketch values.

4.2 Performance Metrics

The algorithms are evaluated based on (1) execution time, (2) influence score, and (2) maximum memory used. For Influence Maximization, there is a trade-off among these performance metrics; in one extreme, it is trivial to select random vertices as the seed set. In another, one can compute the reachability sets of every possible seed set of size K and choose the best one. In all our experiments, the execution times are the wall times reported by the programs. All the methods we benchmarked exclude the time spent on reading files and preprocessing. We allowed all methods to utilize the given number of CPU cores in all benchmarks, except TIM+, which is a single-threaded algorithm. The memory use reported in this paper is the *maximum resident set sizes* (RSS), which are measured using GNU `time` command.

Since the algorithms may use different methods to measure the influence, the reported influence scores may not be suitable for comparison purposes with high precision. Due to this reason, we implemented an independent oracle with a straightforward, sample-then-diffuse algorithm without any optimizations. For sampling, the random values are generated by the 32-bit Mersenne Twister pseudorandom generator `mt19937` from `C++` standard library. This algorithm-independent oracle obtains all the influence scores in this paper.

4.3 Algorithms evaluated in the experiments

We evaluated our method against three other state-of-the-art influence maximization algorithms, TIM+, SKIM, IMM, and SSA. The first algorithm focuses on the influence score, whereas the second is a sketch-based algorithm that takes the execution time into account. The third one is a scalable, parallel approximation algorithm with a parameter to control the influence quality, and the last one leverages a Stop-and-Stare strategy.

- The Two-phased Influence Maximization (TIM+) runs in two phases: *Parameter Estimation* which estimates the maximum expected influence and a parameter θ and *Node*

Selection which randomly samples θ reverse reachability sets from G and then derives a size-K vertex-set S that covers a large number of these sets [28]. The algorithm has a parameter ϵ which allows a trade-off between the seed set quality and execution time. In our experiments, we set $\epsilon = 0.2$ to have a high-quality influence maximization baseline as suggested by the software. We also experimented with $\epsilon = 1.0$ as authors' suggested, which gives around $8\times$ speedup on average but a reduction on the influence score up to 9%.

- The Sketch-based Influence Maximization (SKIM) uses a combined bottom-k min-hash reachability sketch [9] to estimate the influence scores of the seed sets [8]. As suggested by the authors, in this work, we employ SKIM with $k = 64$ and $\ell = 64$ sampled subgraphs. The implementation (from the authors) is partially parallelized and leverages multicore processors. However, it assumes a constant w value; hence, we only used SKIM for the first three settings given in Section 4.1.
- Minutoli et al.'s IMM is a high-performance, parallel algorithm that efficiently produces accurate seed sets [23]. It is an approximation method that improves the Reverse Influence Sampling (RIS) [2] algorithm by eliminating the need for the threshold to be used. We have used $\epsilon = 0.5$ as suggested in the original paper, where ϵ is a user-defined parameter to control the approximation boundaries.
- Nguyen et al. proposed SSA and D-SSA [27], two novel sampling approaches for Influence Maximization problem. SSA and its dynamic network variant D-SSA are fast methods compared to previously mentioned approaches while providing similar approximation guarantees. The approach employs a novel Stop-and-Stare strategy. It stops at exponential checkpoints to verify (stare) if there is adequate statistical evidence on the solution quality. We used SSA for our benchmarks since our networks are static.

4.4 Comparing HYPERFUSER with State-of-the-Art

To compare the run time, memory use, and quality of HYPERFUSER with those of the state-of-the-art, we perform experiments using the following parameters controlling the quality of the seed sets: TIM+ ($\epsilon = 0.2$), IMM ($\epsilon = 0.5$), SKIM ($l = 64, k = 64$), and SSA($\epsilon = 0.1$). In fact, one of the drawbacks of HYPERFUSER is that it does not have a direct control over the approximation factor, whereas TIM+ and IMM have one. Still, HYPERFUSER can control the quality indirectly by tuning the number of Monte-Carlo simulations $\mathcal{J}$ which also increases the number of sketches used per vertex. In the experiments, we have used $\mathcal{J} = 256$ for HYPERFUSER. In addition, as explained in the previous section, we use a global error threshold $\epsilon_g = 0.01$, a local error threshold as $\epsilon_l = 0.3$, and the early-exit ratio as $\epsilon_c = 0.02$.

We present the results in Tables 3, 4, 5, 6 and 7 for constant edge weights $w = 0.005$, 0.01, 0.1, normally distributed edge weights $w \in N(0.05, 0.025)$ and uniformly distributed $w \in U(0, 0.1)$, respectively. The top part of each table shows the results for the networks, and the bottom four rows are the arithmetic mean, geometric mean, maximum and minimum, respectively, of the scores after they are normalized w.r.t. those of HYPERFUSER's scores. In all tables, for the execution time and memory columns, lower values are better. For the influence score columns, higher values are better.

HYPERFUSER's memory consumption is less compared to those of others. Furthermore, it stays the same for all experimental settings with different w values. This is partly due to fused sampling; the memory consumption is linearly dependent only on the number of vertices in G. Both sketch-registers and *visited* information are stored per vertex. Hence, HYPERFUSER's memory consumption stays constant for any simulation parameters or any number of edges. That is given $\mathcal{J}$, HYPERFUSER's memory use is predictable for any graph. On the other hand, the other methods' memory consumption tend to increase with w, and their behaviours change with different parameters and graphs.

Overall, the performance characteristics of the proposed algorithm are different from its state-of-the-art competitors. HYPERFUSER's performance is highly affected by G's diameter. For instance, for `Pokec` with $w = 0.01$, the average diameter of the samples is 43, which makes HYPERFUSER to lose its edge against its fastest competitor. On the other hand, with $w = 0.1$, the average diameter is only around 17, and HYPERFUSER is six times faster than its nearest competitor. Indeed, its execution time decreases as the samples and the influence graph G get denser. On the other hand, the other methods tend to get relatively slower under these changes. For instance, when the influence probability increases from $w = 0.005$ to $w = 0.01$, the average speedups of HYPERFUSER over the baselines, i.e., TIM+, IMM, SKIM, and SSA, increase by $29\times$, $3.1\times$, $3.8\times$, and $27\times$, respectively.

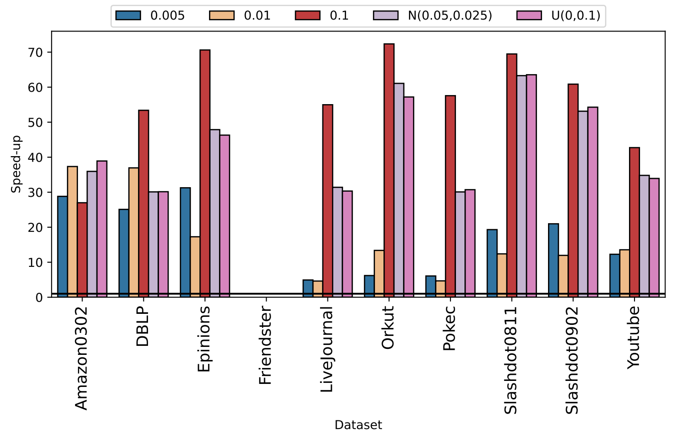


Fig. 8. Speed-up achieved by HYPERFUSER ($\mathcal{J} = 256$) over IMM (ϵ=0.5) using $\tau = 16$ threads.

In terms of influence scores, as expected, the average influence probability seems to have a significant impact on the expected number of influenced vertices. For $w \in \{0.005, 0.01, 0.1\}$ this is clear; for instance, on the graph `LiveJournal`, the influences obtained by HYPERFUSER are 5892, 48182, and 1698955, respectively. The uniform and normal distribution with mean $w_{mean} = 0.05$ also generate influence values in between the ones generated for $w = 0.01$ and $w = 0.1$. On the other hand, the changes on the influence vary for different graph; for instance, when w is increased from 0.01 to 0.1, for `Slashdot` graphs, the

TABLE 3
EXECUTION TIMES (IN SECS), INFLUENCE SCORES, AND MEMORY USE (IN GBS) WITH $K = 50$ SEEDS, $\tau = 16$ THREADS, AND $w = 0.005$. INFLUENCE SCORES ARE GIVEN RELATIVE TO HYPERFUSER. THE RUNS THAT DID NOT FINISH IN 5 HOURS OR DUE TO HIGH MEMORY USE SHOWN AS "-".

	Time					Score					Memory				
Method Dataset	HYP. FUS.	TIM+	IMM	SKIM	SSA	HYP. FUS.	TIM+	IMM	SKIM	SSA	HYP. FUS.	TIM+	IMM	SKIM	SSA
Amazon0302	2.0	295.3	64.5	12.7	11.4	96.5	1.047×	1.021×	0.571×	1.014×	0.09	45.03	1.01	1.51	1.35
DBLP	2.5	480.2	64.9	13.2	28.4	107.7	1.046×	1.008×	1.010×	1.014×	0.14	66.98	1.05	1.38	2.68
Friendster	1054.0	-	-	4055.9	-	4.94M	-	-	1.000×	-	62.0	-	-	112.3	-
Orkut	37.3	-	253.4	49.0	1245.3	0.16M	-	0.997×	1.001×	0.995×	2.61	-	7.97	1.77	5.81
Youtube	5.4	82.1	63.7	5.7	3.8	1829.5	1.031×	1.013×	1.012×	1.016×	0.37	7.41	0.63	0.53	0.37
Epinions	1.3	13.6	47.9	1.9	0.6	245.0	1.013×	0.974×	0.988×	0.962×	0.03	3.70	0.24	0.17	0.10
LiveJournal	12.3	684.5	74.6	22.3	17.0	5892.2	1.046×	1.034×	1.030×	1.029×	1.88	17.06	2.19	2.05	1.38
Pokec	15.3	312.2	78.6	23.2	16.9	593.6	1.011×	0.988×	0.982×	0.978×	0.69	33.18	1.80	1.78	1.09
Slashdot0811	2.1	12.8	38.5	1.3	1.0	323.3	1.013×	1.001×	0.986×	0.990×	0.03	2.47	0.20	0.13	0.10
Slashdot0902	1.7	14.9	38.2	1.4	0.5	341.8	1.023×	0.985×	0.989×	0.978×	0.03	2.66	0.21	0.13	0.06
Avg. Perf.	1.0×	57.0×	18.3×	2.4×	6.1×	1.00×	1.03×	1.00×	0.96×	1.00×	1.00×	166.16×	5.28×	4.77×	5.32×
Max. Perf.	1.0×	191.3×	35.8×	6.4×	33.4×	1.00×	1.05×	1.03×	1.03×	1.03×	1.00×	493.46×	11.06×	16.57×	19.17×
Geo. Mean	1.0×	26.8×	14.6×	1.8×	1.7×	1.00×	1.03×	1.00×	0.95×	1.00×	1.00×	78.94×	4.19×	3.05×	2.88×

TABLE 4
EXECUTION TIMES (IN SECS), INFLUENCE SCORES, AND MEMORY USE (IN GBS) WITH $K = 50$ SEEDS, $\tau = 16$ THREADS, AND $w = 0.01$. INFLUENCE SCORES ARE GIVEN RELATIVE TO HYPERFUSER. THE RUNS THAT DID NOT FINISH IN 5 HOURS OR DUE TO HIGH MEMORY USE SHOWN AS "-".

	Time					Score					Memory				
Method Dataset	HYP. FUS.	TIM+	IMM	SKIM	SSA	HYP. FUS.	TIM+	IMM	SKIM	SSA	HYP. FUS.	TIM+	IMM	SKIM	SSA
Amazon0302	1.2	211.2	52.7	12.4	7.3	153.3	1.034×	1.001×	0.402×	1.014×	0.09	35.22	0.67	1.43	0.70
DBLP	1.6	145.5	59.2	7.3	13.9	236.8	1.027×	0.994×	0.981×	1.003×	0.14	24.53	0.61	0.82	1.38
Friendster	871.7	-	-	2641.5	-	11.50M	-	-	1.000×	-	62.03	-	-	78.17	-
Orkut	45.2	-	606.0	125.1	5613.0	0.65M	-	1.000×	1.000×	1.000×	2.61	-	20.48	3.33	27.19
Youtube	3.5	1056.4	50.4	8.5	19.8	9098.2	1.007×	0.993×	1.004×	0.982×	0.37	14.15	0.62	0.73	0.41
Epinions	2.1	13.1	37.4	1.1	0.3	673.6	1.037×	1.007×	1.026×	0.996×	0.03	1.51	0.17	0.11	0.04
LiveJournal	19.0	-	88.3	18.3	195.2	48182.6	-	1.000×	1.004×	0.995×	1.88	-	2.96	1.47	2.52
Pokec	16.2	145.3	59.3	7.5	7.2	1842.6	1.015×	0.990×	0.983×	0.987×	0.69	11.43	1.12	0.76	0.61
Slashdot0811	2.7	39.8	35.3	1.0	1.6	1194.0	1.023×	1.005×	0.993×	0.997×	0.03	1.29	0.16	0.08	0.06
Slashdot0902	2.8	44.5	35.2	1.1	1.0	1270.4	1.027×	0.998×	1.006×	0.984×	0.03	1.43	0.16	0.08	0.04
Avg. Perf.	1.0×	84.9×	17.3×	2.5×	17.3×	1.00×	1.02×	1.00×	0.94×	1.00×	1.00×	106.97×	4.44×	3.64×	3.95×
Max. Perf.	1.0×	295.9×	41.2×	9.8×	124.1×	1.00×	1.04×	1.01×	1.03×	1.01×	1.00×	385.78×	7.84×	15.71×	10.40×
Geo. Mean	1.0×	34.1×	13.3×	1.4×	2.6×	1.00×	1.02×	1.00×	0.91×	1.00×	1.00×	62.23×	3.75×	2.36×	2.42×

TABLE 5
EXECUTION TIMES (IN SECS), INFLUENCE SCORES, AND MEMORY USE (IN GBS) WITH $K = 50$ SEEDS, $\tau = 16$ THREADS, AND $w = 0.1$. INFLUENCE SCORES ARE GIVEN RELATIVE TO HYPERFUSER. THE RUNS THAT DID NOT FINISH IN 5 HOURS OR DUE TO HIGH MEMORY USE SHOWN AS "-".

	Time					Score					Memory				
Method Dataset	HYP. FUS.	TIM+	IMM	SKIM	SSA	HYP. FUS.	TIM+	IMM	SKIM	SSA	HYP. FUS.	TIM+	IMM	SKIM	SSA
Amazon0302	1.9	297.1	46.8	4.9	6.7	12387.2	1.006x	0.985x	0.823x	0.953x	0.09	11.26	0.37	0.54	0.20
DBLP	0.9	2904.0	45.2	3.8	55.2	48809.6	1.001x	0.992x	1.000x	0.990x	0.14	69.72	1.18	0.27	1.10
Friendster	455.3	-	-	5660.2	-	36.99M	-	-	1.000x	-	62.03	-	-	90.95	-
Orkut	16.4	-	1148.1	490.1	-	2.69M	-	1.000x	1.000x	-	2.61	-	71.76	10.08	-
Youtube	1.7	-	71.5	15.8	352.5	0.17	-	0.999x	1.001x	0.999x	0.37	-	4.29	1.11	5.15
Epinions	0.5	436.9	34.4	2.1	16.0	10479.6	1.005x	0.997x	1.005x	0.996x	0.03	14.77	0.37	0.18	0.36
LiveJournal	9.8	-	553.4	71.0	-	1.69M	-	0.999x	1.002x	-	1.88	-	46.07	1.35	-
Pokec	4.5	-	255.8	37.1	3256.6	0.77M	-	1.000x	1.001x	1.000x	0.69	-	19.60	0.99	35.56
Slashdot0811	0.5	994.8	33.5	3.1	32.0	23216.4	1.000x	0.998x	1.001x	0.998x	0.03	27.16	0.51	0.24	0.63
Slashdot0902	0.5	897.3	33.1	2.9	23.6	24259.3	1.001x	0.996x	1.001x	0.995x	0.03	26.39	0.53	0.23	0.63
Avg. Perf.	1.0x	1654.0x	56.3x	9.1x	164.0x	1.00x	1.00x	1.00x	0.98x	0.99x	1.00x	545.28x	16.46x	3.83x	17.82x
Max. Perf.	1.0x	3337.9x	74.1x	29.9x	728.5x	1.00x	1.01x	1.00x	1.01x	1.00x	1.00x	832.20x	28.35x	7.40x	51.41x
Geo. Mean	1.0x	1110.5x	53.8x	7.2x	61.5x	1.00x	1.00x	1.00x	0.98x	0.99x	1.00x	456.57x	14.18x	2.99x	12.61x

TABLE 6
EXECUTION TIMES (IN SECS), INF. SCORES, AND MEMORY USE (IN GBS) WITH $K = 50$ SEEDS, $\tau = 16$ THREADS, AND $w \in N(0.05, 0.025)$. INFLUENCE SCORES ARE GIVEN RELATIVE TO HYPERFUSER. THE RUNS THAT DID NOT FINISH IN 5 HOURS OR DUE TO HIGH MEMORY USE SHOWN AS "-".

	Time				Score				Memory			
Method Dataset	HYP. FUS.	TIM+	IMM	SSA	HYP. FUS.	TIM+	IMM	SSA	HYP. FUS.	TIM+	IMM	SSA
Amazon0302	1.4	27.0	43.3	1.2	1151.8	1.020x	1.007x	0.973x	0.09	2.71	0.26	0.09
DBLP	1.4	472.6	41.1	10.7	10010.3	1.004x	0.989x	0.972x	0.14	13.37	0.40	0.25
Friendster	492.1	-	-	-	29.45M	-	-	-	62.03	-	-	-
Orkut	18.3	-	1158.9	-	2.32M	-	1.000x	-	2.61	-	62.48	-
Youtube	1.9	-	65.2	231.6	86486.2	-	0.998x	0.998x	0.37	-	2.43	2.31
Epinions	0.8	273.4	37.0	8.2	5869.9	1.007x	0.984x	0.983x	0.03	7.92	0.27	0.20
LiveJournal	13.5	-	402.5	-	0.93M	-	0.997x	-	1.88	-	26.52	-
Pokec	6.4	-	186.9	1418.0	0.46M	-	0.999x	0.998x	0.69	-	12.00	14.09
Slashdot0811	0.6	549.8	39.2	22.5	13942.5	1.001x	0.994x	0.995x	0.03	14.79	0.39	0.46
Slashdot0902	0.7	608.2	38.4	20.2	14522.0	1.002x	0.992x	0.991x	0.03	16.58	0.40	0.45
Avg. Perf.	1.0x	483.8x	42.2x	60.8x	1.00x	1.01x	1.00x	0.99x	1.00x	266.05x	11.15x	9.05x
Max. Perf.	1.0x	872.7x	63.5x	222.3x	1.00x	1.02x	1.01x	1.00x	1.00x	482.03x	23.90x	20.38x
Geo. Mean	1.0x	279.3x	40.2x	21.1x	1.00x	1.01x	1.00x	0.99x	1.00x	175.72x	9.04x	6.01x

TABLE 7
EXECUTION TIMES (IN SECS), INFLUENCE SCORES, AND MEMORY USE (IN GBS) WITH $K = 50$ SEEDS, $\tau = 16$ THREADS AND $w \in U(0, 0.1)$. INFLUENCE SCORES ARE GIVEN RELATIVE TO HYPERFUSER. THE RUNS THAT DID NOT FINISH IN 5 HOURS OR DUE TO HIGH MEMORY USE SHOWN AS "-".

	Time				Score				Memory			
Method	HYP.	TIM+	IMM	SSA	HYP.	TIM+	IMM	SSA	HYP.	TIM+	IMM	SSA
Dataset	FUS.				FUS.				FUS.			
`Amazon0302`	1.4	24.3	43.9	1.2	1129.6	1.004x	0.991x	0.969x	0.09	2.74	0.27	0.09
`DBLP`	1.3	431.5	41.1	18.1	9971.0	1.008x	0.986x	0.982x	0.14	12.64	0.39	0.38
`Friendster`	519.7	-	-	-	29.41M	-	-	-	62.03	-	-	-
`Orkut`	18.9	-	1163.9	-	2.32M	-	1.000x	-	2.61	-	62.30	-
`Youtube`	1.9	-	64.2	152.2	86133.5	-	0.998x	0.998x	0.37	-	2.35	2.10
`Epinions`	0.8	297.2	36.9	8.1	5851.2	1.007x	0.987x	0.979x	0.03	8.63	0.27	0.18
`LiveJournal`	14.0	-	402.0	-	924652.4	-	0.997x	-	1.88	-	26.39	-
`Pokec`	6.3	-	186.2	1220.5	462030.1	-	0.998x	0.998x	0.69	-	12.19	13.49
`Slashdot0811`	0.6	533.3	38.8	23.3	13895.8	1.001x	0.994x	0.994x	0.03	12.63	0.39	0.45
`Slashdot0902`	0.7	637.8	38.6	19.9	14484.7	1.002x	0.992x	0.992x	0.03	16.90	0.40	0.45
Avg. Perf.	1.0x	527.0x	43.4x	52.6x	1.00x	1.00x	0.99x	0.99x	1.00x	258.70x	11.14x	8.85x
Max. Perf.	1.0x	966.3x	68.0x	192.5x	1.00x	1.01x	1.00x	1.00x	1.00x	492.75x	23.83x	19.50x
Geo. Mean	1.0x	289.5x	41.1x	21.8x	1.00x	1.00x	0.99x	0.99x	1.00x	172.40x	9.00x	6.16x

influence increases from around 1200 to 24000, whereas it increases from around 1850 to 775000 for `Pokec`.

Figure 8 shows the speedups of HYPERFUSER over IMM for all the graphs and all w values. As described above, the relatively sparser setting $w = 0.005$ is especially challenging due to the high diameter of the influence graph and low vector unit utilization. Even with this w value, HYPERFUSER is only slower by a few seconds and only when the influence is small. For larger graphs with larger influences HYPERFUSER is much faster than IMM. As explained before, for larger w, HYPERFUSER's execution-time performance is usually better, and its influence quality is on par with that of IMM.

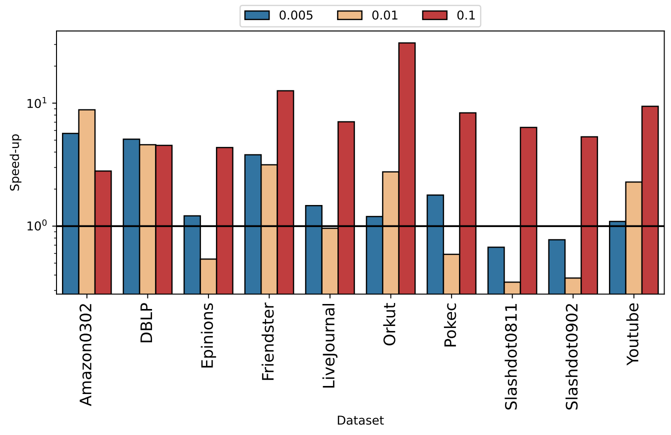


Fig. 9. Speed-up achieved by HYPERFUSER ($\mathcal{J} = 256$) over SKIM ($r{=}64, l{=}64$) using $\tau = 16$ threads.

Figure 9 compares HYPERFUSER's execution-time performance with that of SKIM. As the figure shows, the proposed approach performs much better, both in terms of quality and speed in almost all settings. For the notorious `Pokec` dataset, HYPERFUSER performs better than SKIM, except for $w = 0.01$. The diameter of G does not affect SKIM's performance as much as HYPERFUSER. SKIM is faster in this setting, but it has worse influence quality. In some settings such as `Amazon` and $w = 0.01$, SKIM performs very poorly; only 40% of the influence is achieved with respect to HYPERFUSER. In addition, under the same setting, SKIM spends 12.48 seconds whereas HYPERFUSER finishes in 1.2 seconds.

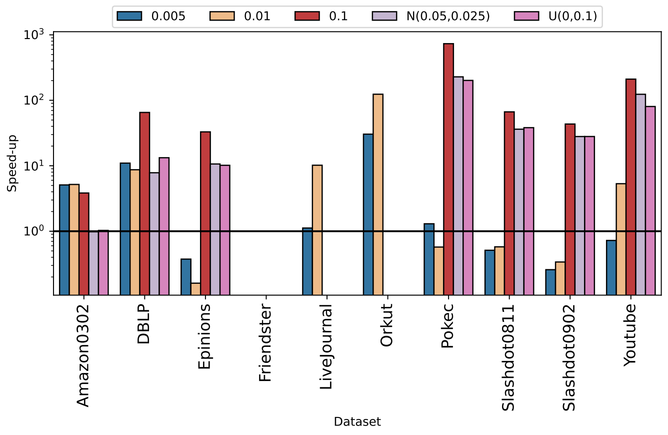


Fig. 10. Speed-up achieved by HYPERFUSER ($\mathcal{J} = 256$) over SSA ($\epsilon = 0.2$) using $\tau = 16$ threads.

Figure 10 compares HYPERFUSER's execution-time performance with SSA. The figure shows that the proposed approach can process much larger datasets efficiently. For smaller datasets, SSA can be competitive with our approach. However, when the influence graph is denser, HYPERFUSER is a few orders of magnitude faster.

4.5 Scalability with multi-threaded parallelism

In our implementation, we used a *pull-based* approach in which the vertices (processed at line 4 of Algorithm 2) pull the influence, i.e., reachability set cardinality estimations, from their outgoing neighbors. A classical *push-based* approach, in which the vertices relay their influence to their outgoing neighbors could also be leveraged. However, the push-based approach makes a (target) vertex register potentially updated at the same time in different computation units. Specifically, the update operation (corresponding to the one at line 8 of Algorithm 2) of the *pull-based* approach will be the cause of race conditions. One can easily argue that since we are already using sketches, and not computing exact cardinalities, such race conditions are acceptable and they will not reduce the quality of influence estimations. However, the performance may suffer due to false sharing. Even though the pull-based diffusion shows a nice parallel performance, it is possible to implement HYPERFUSER using other approaches such as the *queue-based* approach which may improve performance by only processing live vertices. The pull-based diffusion method is chosen due to its simplicity and scalability to many threads.

5 Related Work

HyperFuseR borrows much from InFuseR [14], including hash-based fused sampling. InFuseR computes influence by memoizing connected components for all vertices and only can work on *undirected* datasets. It also employs CELF optimization to reduce cardinality computations. On the other hand, HyperFuseR can process both directed and undirected graphs and uses the Flajolet–Martin sketches in a novel way to estimate cardinality and choose seed candidates.

Sketch-based IM methods are cheaper compared to simulation-based methods. A popular method for sketch-based IM is Skim [8] by Cohen et al. Skim uses combined bottom-k min-hash reachability sketches [7], [9], built on ℓ sampled subgraphs, to estimate the influence scores of the seed sets. It is parallel in the sense that it uses `OpenMP` parallelization during sketch utilization. However, the sketch building step is single-threaded. In this work, we choose Flajolet-Martin sketches [11] for their simplicity, suitability for vectorization and fused sampling, and hence, execution-time performance. Skim treats vertex/sample pairs as distinct elements and reduces edge traversals via their smallest ranks in bottom-k ketches. On the other hand, HyperFuseR sees vertices as shared elements among the samples, and builds a sketch for each instance. This allows independent parallel processing of vertices and samples, fused sampling, and a better memory layout than the former. In addition, HyperFuseR does not require removing the reachable vertices from the samples. Instead, it uses a rebuilding strategy to improve the result quality.

Two-phased Influence Maximization (Tim+) borrows ideas from RIS but overcomes its limitations with a novel algorithm [28]. Its first phase computes a lower bound of the maximum expected influence over all size-K node sets. It then uses this bound to derive a parameter θ. In the second phase, it samples θ random RR sets from G, and then derives a size-K node-set that covers a large number of RR sets.

The stop-and-stare algorithm proposed by Nguyen et al. [27] utilizes an exponential checkpoint strategy for the Influence Maximization problem to verify statistical evidence on the quality of its solution. At each iteration, it utilizes the RIS framework to generate RR sets to find a candidate solution. If the quality of the solution is inferior to the expectation, the sample size is doubled.

Kumar and Calders [19] proposed the Time Constrained Information Cascade Model and a kernel that works on the model using versioned HyperLogLog sketches. The algorithm computes the influence for all vertices in G while performing a single pass over the data. The sketches are used for each time window to estimate active edges. On the other hand, HyperFuseR uses $\mathcal{J}$ sketches for each vertex to estimate the marginal influence and employs a rebuilding strategy for fast processing. HyperFuseR also utilizes fused sampling and error-adaptive rebuilding of sketches.

6 Conclusion and Future Work

In this work, we propose a sketch-based Influence Maximization algorithm that employs fused sampling and a novel error-adaptive rebuilding strategy. We provide a fast implementation of the algorithm that utilizes multi-threading to exploit multiple cores. Also, we present a performance comparison with state-of-the-art IM algorithms on real-world datasets and show that HyperFuseR can be an order of magnitude faster while obtaining the same influence. In the future, we will extend our work to a distributed GPGPU setting to process graphs with billions of vertices and edges under a minute.

Acknowledgements

This work was supported by Scientific and Technological Research Council of Turkey (TÜBİTAK) and EuroHPC Joint Undertaking through grant agreement No 220N254 and grant agreement No 956213 (SparCity), respectively. The numerical calculations reported in this paper were partially performed at TÜBİTAK ULAKBİM, High Performance and Grid Computing Center (TRUBA resources).

References

[1] W. Ackermann, "Zum hilbertschen aufbau der reellen zahlen," *Math. Ann.*, no. 99, p. 118–133, 1928.
[2] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier, "Maximizing social influence in nearly optimal time," in *Proc. of the 25th annual ACM-SIAM symposium on Discrete Alg.* SIAM, 2014, pp. 946–957.
[3] W. Chen, C. Wang, and Y. Wang, "Scalable influence maximization for prevalent viral marketing in large-scale social networks," in *Proc. of the 16th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining.* ACM, 2010, pp. 1029–1038.
[4] W. Chen, Y. Wang, and S. Yang, "Efficient influence maximization in social networks," in *Proc. of the 15th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining.* ACM, 2009, pp. 199–208.
[5] W. Chen, Y. Yuan, and L. Zhang, "Scalable influence maximization in social networks under the linear threshold model," in *2010 IEEE Int. Conf. on data mining.* IEEE, 2010, pp. 88–97.
[6] S. Cheng, H. Shen, J. Huang, W. Chen, and X. Cheng, "Imrank: influence maximization via finding self-consistent ranking," in *Proc. of the 37th Int. ACM SIGIR Conf. on Research & Development in Information Retrieval.* ACM, 2014, pp. 475–484.
[7] E. Cohen, "All-distances sketches, revisited: Hip estimators for massive graphs analysis," *IEEE Trans. on Knowledge and Data Eng.*, vol. 27, no. 9, pp. 2320–2334, 2015.
[8] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck, "Sketch-based influence maximization and computation: Scaling up with guarantees," in *Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management*, 2014, pp. 629–638.
[9] E. Cohen and H. Kaplan, "Summarizing data using bottom-k sketches," in *Proc. of the 26th Symp. on Principles of Distributed Comp.*, ser. PODC '07. New York, NY, USA: ACM, 2007, p. 225–234.
[10] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *European Symposium on Algorithms.* Springer, 2003, pp. 605–617.
[11] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.
[12] S. Galhotra, A. Arora, and S. Roy, "Holistic influence maximization: Combining scalability and efficiency with opinion-aware models," in *Proc. of the 2016 Int. Conf. on Management of Data.* ACM, 2016, pp. 743–758.
[13] A. Goyal, W. Lu, and L. V. Lakshmanan, "Simpath: An efficient algorithm for influence maximization under the linear threshold model," in *11th Int. Conf. on Data Mining*, 2011, pp. 211–220.
[14] G. Göktürk and K. Kaya, "Boosting parallel influence-maximization kernels for undirected networks with fusing and vectorization," *IEEE Trans. on Parallel and Dist. Systems*, vol. 32, no. 5, pp. 1001–1013, 2021.

[15] K. Jung, W. Heo, and W. Chen, "Irie: Scalable and robust influence maximization in social networks," in *2012 IEEE 12th Int. Conf. on Data Mining*. IEEE, 2012, pp. 918–923.

[16] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proc. of the ninth ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. ACM, 2003, pp. 137–146.

[17] J. Kim, S.-K. Kim, and H. Yu, "Scalable and parallelizable processing of influence maximization for large-scale social networks?" in *29th Int. Conf. on Data Eng. (ICDE)*, 2013, pp. 266–277.

[18] M. Kimura, K. Saito, and R. Nakano, "Extracting influential nodes for information diffusion on a social network," in *AAAI*, vol. 7, 2007, pp. 1371–1376.

[19] R. Kumar and T. Calders, "Information propagation in interaction networks," *Advances in Database Technology-EDBT*, vol. 2017, pp. 270–281, 2017.

[20] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Trans. on the Web (TWEB)*, vol. 1, no. 1, p. 5, 2007.

[21] Q. Liu, B. Xiang, E. Chen, H. Xiong, F. Tang, and J. X. Yu, "Influence maximization over large-scale social networks: A bounded linear approach," in *Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management*. ACM, 2014, pp. 171–180.

[22] L. Lü, M. Medo, C. H. Yeung, Y.-C. Zhang, Z.-K. Zhang, and T. Zhou, "Recommender systems," *Physics reports*, vol. 519, no. 1, pp. 1–49, 2012.

[23] M. Minutoli, M. Halappanavar, A. Kalyanaraman, A. Sathanur, R. Mcclure, and J. McDermott, "Fast and scalable implementations of influence maximization algorithms," in *2019 IEEE Int. Conf. on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[24] Y. Moreno, M. Nekovee, and A. F. Pacheco, "Dynamics of rumor spreading in complex networks," *Physical Review E*, vol. 69, no. 6, p. 066130, 2004.

[25] R. Narayanam and Y. Narahari, "A shapley value-based approach to discover influential nodes in social networks," *IEEE Trans. on Automation Science and Eng.*, vol. 8, no. 1, pp. 130–147, 2010.

[26] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions—i," *Mathematical programming*, vol. 14, no. 1, pp. 265–294, 1978.

[27] 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 2016 international conference on management of data*, 2016, pp. 695–710.

[28] Y. Tang, X. Xiao, and Y. Shi, "Influence maximization: Near-optimal time complexity meets practical efficiency," *CoRR*, vol. abs/1404.0900, 2014.

[29] M. Trusov, R. E. Bucklin, and K. Pauwels, "Effects of word-of-mouth versus traditional marketing: findings from an internet social networking site," *J. of Marketing*, vol. 73, no. 5, pp. 90–102, 2009.

[30] D. Zeng, H. Chen, R. Lusch, and S.-H. Li, "Social media analytics and intelligence," *IEEE Intel. Sys.*, vol. 25, no. 6, pp. 13–16, 2010.