# Fast and Space-Efficient Parallel Algorithms for Influence Maximization

Letong Wang
UC Riverside
lwang323@ucr.edu

Xiangyun Ding
UC Riverside
xding047@ucr.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

## Abstract

Influence Maximization (IM) is a crucial problem in data science. The goal is to find a fixed-size set of highly-influential *seed* vertices on a network to maximize the influence spread along the edges. While IM is NP-hard on commonly-used diffusion models, a greedy algorithm can achieve $(1-1/e)$-approximation, repeatedly selecting the vertex with the highest *marginal gain* in influence as the seed. Due to theoretical guarantees, rich literature focuses on improving the performance of the greedy algorithm. To estimate the marginal gain, existing work either runs Monte Carlo (MC) simulations of influence spread or pre-stores hundreds of *sketches* (usually per-vertex information). However, these approaches can be inefficient in time (MC simulation) or space (storing sketches), preventing the ideas from scaling to today's large-scale graphs.

This paper significantly improves the scalability of IM using two key techniques. The first is a *sketch-compression* technique for the independent cascading model on undirected graphs. It allows combining the simulation and sketching approaches to achieve a time-space tradeoff. The second technique includes new data structures for parallel seed selection. Using our new approaches, we implemented *PaC-IM*: Parallel and Compressed IM.

We compare *PaC-IM* with state-of-the-art parallel IM systems on a 96-core machine with 1.5TB memory. *PaC-IM* can process large-scale graphs with up to 900M vertices and 74B edges in about 2 hours. On average across all tested graphs, our uncompressed version is 5–50× faster and about 1.5× more space-efficient than existing parallel IM systems. Using compression further saves 3.8× space with only 70% overhead in time on average.

## 1 Introduction

Influence Maximization (IM) is a crucial problem in data science. The goal is to find a fixed-size set of highly-influential *seed* vertices on a network to maximize the spread of influence along the edges. For example, in viral marketing, the company may choose to send free samples to a small set of users, in the hope of triggering a large cascade of further adoptions through the "word-of-mouth" effects. Given a network $G = (V, E)$ and a stochastic *diffusion model* to specify how influence spreads along edges, we use $n = |V|$, $m = |E|$, and $\sigma(S)$ to denote the expected influence spread on $G$ using seed set $S \subseteq V$. The IM problem aims to find a seed set $S$ with fixed size $k$ to maximize $\sigma(S)$. Given its importance, IM is widely-studied, and we refer the audience to a list of surveys [5, 7, 82] that reviews the numerous applications and a few hundred papers on this topic.

Among various diffusion models, Independent Cascade (IC) [32] (defined in Sec. 2) is one of the earliest and most widely-used models. In this model, only seeds are ***active*** initially. In each timestamp, each vertex $v$ that is newly activated in the last timestamp will activate its neighbors $u$ with a probability $p_{vu}$. Although IM is NP-hard on the IC model [41], the monotone and submodular properties of IC allow for a greedy algorithm with $(1 - 1/e)$-approximation [41]. Given the current seed set $S$, the greedy algorithm selects the next seed as the vertex with the highest *marginal gain*, i.e., $\arg\max_{v \in V}\{\sigma(S \cup \{v\}) - \sigma(S)\}$. Due to the theoretical guarantee, this greedy strategy generally gives better solution quality than other heuristics [49]. However, the challenge lies in estimating the influence $\sigma(S)$ of a seed set $S$. Early work uses Monte-Carlo (MC) experiments by averaging $R'$ rounds of influence diffusion simulation [41, 47]. However, the solution quality relies on the value of $R'$, which is usually around $10^4$. Later work uses ***sketch-based*** approaches [17, 21–23, 56, 70, 71] to avoid MC experiments. Such algorithms pre-store $R$ ***sketches***, where each of them represents a sampled graph—each edge is chosen with the probability in which influence can spread along this edge. When estimating the influence, the diffusion is only simulated on the sampled graphs, which act as the results of the MC experiments. In an existing study [22], using $R \approx 200$ sketches can achieve similar solution quality to $R' = 10^4$ MC experiments, which greatly improves efficiency. The sketches can either be the sampled graphs, and/or *memoizing* more information from the sampled graphs to accelerate influence computation, such as connectivity [21, 31] or strong connectivity [56].

While numerous sketch-based solutions have been developed, we observed great challenges in scaling them to today's large-scale graphs. In a SIGMOD'17 benchmark paper [3] on nine state-of-the-art (sequential) IM solutions, none of them can process the Friendster graph (65M vertices and 3.6B edges) [45] due to either timeout (more than 40 hours) or out-of-memory. Even the recent parallel algorithms [31, 53, 54, 60] need more than half an hour to process Friendster on a 96-core machine (See Table 4). There are two major challenges in scaling sketch-based approaches to today's billion-scale graphs. The first is the *space issue*. Storing each sketch usually needs some per-vertex information. This indicates $O(Rn)$ space, which is expensive on large graphs (empirically $R$ is a few hundred). The second is insufficient parallelism. In particular, many state-of-the-art IM solutions uses the *CELF* [47] optimization for seed selection (see details below), which is inherently sequential.

In this paper, we take a significant step to ***improve the scalability of sketch-based IM solutions***, and test our algorithms on ***real-world billion-scale graphs***. We propose two new techniques to improve both space and time for the IM problem. Our first solution is a ***sketch compression*** technique for *IC model on undirected graphs*, which limits the auxiliary space by a user-defined capacity and reduces space usage. Our second technique is ***parallel data structures*** for greedy seed selection that improves parallelism and reduces running time. Our new data structures work on general graphs and any diffusion model with submodularity. Combining the new ideas, we implemented ***PaC-IM*: Parallel and Compressed IM**. We present a heatmap to overview our results in Fig. 1. On the aforementioned Friendster graph ( "FT" in Fig. 1), *PaC-IM* only uses 128 seconds to finish without compression (using 2.5× auxiliary space on top of the input graph), or 609 seconds when limiting

auxiliary space in 0.45× input size, using a 96-core machine. This is at least 15× faster than existing parallel solutions, while being much more space efficient, and achieving the same solution quality (see Tab. 4). Below, we overview the key contributions of this paper.

Our first contribution is a **novel compression scheme to store sketches** on undirected graphs and the IC model, which allows for user-defined compression ratios (details in Sec. 3). Similar with existing algorithms [21, 31], our algorithm memoizes connectivity information of the sketches, but *carefully avoids the $O(Rn)$ space to store per-vertex information.* Our technique is a combination (and thus a tradeoff) of (partial) memoization with (partial) simulations. The high-level idea is to memoize the connectivity information only for $\rho = \alpha n$ *centers* $C \subseteq V$, where $0 \leq \alpha \leq 1$ is a user-defined parameter. The influence information of the non-center vertices will be retrieved by a local simulation to find a connected center. Theoretically, we show that we can limit the auxiliary space by a factor of $\alpha$ by increasing the time by a factor of $O(1/\alpha)$. Experimentally, such tradeoff is studied in Fig. 8.

Our second contribution includes **two new parallel data structures for seed selection**. Recall that many state-of-the-art IM solutions [21, 22, 31, 42, 54] rely on the CELF optimization [47] (details in Sec. 2) for seed selection, utilizing the *submodularity* of the problem. In a nutshell, CELF is an iterative approach that lazily evaluates the marginal gain of vertices in seed selection, one at a time. While laziness reduces the number of vertices to evaluate, CELF is inherently sequential, which limits the parallelism of existing IM systems. We proposed two novel solutions that achieve high parallelism without introducing much additional work. The key algorithmic challenge is to identify more vertices to evaluate in parallel, while remaining lazy such that most of the "unpromising" vertices remain untouched as in CELF. Our first solution is based on a binary search tree (BST) called *P-tree* [11, 13, 69] (Sec. 4.1). We highlight its *theoretical efficiency* (Thm. 4.1 to 4.3 and their analysis). Our second solution is referred to as *Win-Tree* (Sec. 4.2), which is simpler and has lower space usage, leading to slightly better overall performance. The new data structures are general and work on both directed and undirected graphs and any diffusion model with submodularity. More importantly, we believe they apply to general optimization problems with submodular objective functions. We leave this as future work, and discuss it briefly in Sec. 7.

Our final contribution is an in-depth experimental study of the proposed techniques, and the performance comparison among *PaC-IM* and state-of-the-art parallel IM systems. Our experiment is on four types of a total of 17 graphs, five of which have more than 1B edges. Besides social networks, we also tested other real-world graphs including web graphs, road graphs and $k$-NN graphs. One can consider IM on such graphs as studying the influence diffusion between webpages, geologically or geometrically close objects. In all settings, *PaC-IM* achieves the best running time and space usage on all tested graphs, while guaranteeing comparable or better solution quality to all baselines. Compared to the best baseline, *PaC-IM* with no compression is 5.6× faster and is 1.5× more space-efficient on average (geometric mean across tested graphs), and is 3.2× faster and 6× more space-efficient using compression with $\alpha = 0.1$. Due to space- and time-efficiency, *PaC-IM* is the only system that can process the largest graph ClueWeb [52] with 978M vertices and 75B edges. We believe *PaC-IM* is the first IM solution that scales to tens of billions of edges and close to a billion vertices.

| | | Ours$_1$ | | Ours$_{0.1}$ | | InfuserMG | | Ripples | |
|---|---|---|---|---|---|---|---|---|---|
| | | Time | Space | Time | Space | Time | Space | Time | Space |
| Scale-free (Social and Web) | EP | 1.00 | 1.00 | 1.67 | 0.39 | 1.28 | 1.18 | 14.1 | 1.53 |
| | SLDT | 1.00 | 1.00 | 1.76 | 0.38 | 1.60 | 1.17 | 86.3 | 1.99 |
| | DBLP | 1.00 | 1.00 | 1.06 | 0.21 | 2.5 | 1.40 | 121 | 0.60 |
| | YT | 1.00 | 1.00 | 2.00 | 0.17 | 5.09 | 1.44 | 58.4 | 0.94 |
| | OK | 1.00 | 1.00 | 4.50 | 0.40 | 9.13 | 1.3 | 37.4 | 12.8 |
| | LJ | 1.00 | 1.00 | 3.42 | 0.27 | 10.2 | 1.76 | 20.5 | 3.41 |
| | TW* | 1.00 | 1.00 | 4.05 | 0.41 | 6.84 | 1.63 | 67.0 | 11.6 |
| | FT* | 1.00 | 1.00 | 4.75 | 0.41 | 15.4 | 1.64 | - | - |
| | SD* | 1.00 | 1.00 | 4.18 | 0.36 | 11.2 | 1.69 | - | - |
| | **MEAN** | 1.00 | 1.00 | 2.70 | 0.32 | 5.21 | 1.66 | 46.1 | 2.63 |
| Sparse (Road and kNN) | GER | 1.00 | 1.00 | 0.91 | 0.19 | 2.86 | 1.89 | 250 | 1.73 |
| | USA | 1.00 | 1.00 | 0.94 | 0.20 | 3.63 | 1.90 | 933 | 1.80 |
| | HT5 | 1.00 | 1.00 | 0.95 | 0.18 | 5.90 | 1.41 | 66.2 | 0.61 |
| | HH5 | 1.00 | 1.00 | 0.93 | 0.17 | 3.62 | 1.42 | 23.7 | 0.41 |
| | CH5 | 1.00 | 1.00 | 1.51 | 0.22 | 62.2 | 1.80 | 11.1 | 0.36 |
| | GL5 | 1.00 | 1.00 | 0.92 | 0.22 | 5.96 | 1.87 | 32.7 | 0.32 |
| | COS5* | 1.00 | 1.00 | 0.82 | 0.19 | 6.66 | 1.88 | - | - |
| | **MEAN** | 1.00 | 1.00 | 0.98 | 0.19 | 6.60 | 1.73 | 71.4 | 0.67 |
| **MEAN** | | 1.00 | 1.00 | 1.76 | 0.26 | 5.66 | 1.56 | 55.6 | 1.46 |

0 .2 .5 1 2 4 8 16 >16 MEAN = geometric mean

**Figure 1: Heatmap of relative running time and space usage, normalized to Ours$_1$.** Ours$_1$: *PaC-IM* with no compression. Ours$_{0.1}$: *PaC-IM* with 10× sketch compression. *InfuserMG* [31] and *Ripples* [54]: existing parallel IM systems. Lower/green is better. The graph information is in Tab. 3. The absolute running times are in Tab. 4. ∗: graphs with more than a billion edges. (Colors are adjusted to be distinguishable in gray-scale.)

We publish our code at [2].

## 2 Preliminaries

**Basic Notations**. For graph $G = (V, E)$, we use $n = |V|$ and $m = |E|$. Since our sketch compression is designed for undirected graphs and IC model, for simple description, throughout the paper, we assume $G$ is undirected and consider the IC model unless otherwise specified. On undirected graphs, a ***connected component (CC)*** is a maximal subset in $V$ s.t. every two vertices in it are connected by a path. In a max-priority-queue, we use ***top*** to refer to the element with the largest key, and use function ***pop*** to find and remove the top element. We use $\tilde{O}(f(n))$ to denote $O(f(n) \cdot \text{polylog}(n))$.

**Computational Model**. We use the standard fork-join parallelism [14, 24], and the work-span analysis [16, 37]. We assume a set of threads that access a shared memory. A process can `fork` two child software threads to work in parallel. When both children complete, the parent process continues. A parallel for-loop can be simulated by recursive `fork`s in logarithmic levels. The ***work*** of an algorithm is the total number of instructions and the ***span*** is the length of the longest sequence of dependent instructions. We can execute the computation using a randomized work-stealing scheduler [4, 16] in practice. We use *atomic* operation WriteMax$(t, v_{new})$, which atomically reads the memory location pointed to by $t$, and write value $v_{new}$ to it if $v_{new}$ is larger than the current value in $t$. We use compare-and-swap to implement WriteMax.

### The Influence Maximization (IM) Problem

Given a graph $G = (V, E)$, an influence diffusion model $M$ specifies how influence spreads from a set of current ***active*** vertices to ***activate*** more vertices in $V$. Given a ***seed*** set $S \subseteq V$, we use $\sigma_{G,M}(S)$ to denote the expected number of vertices that $S$ can activate (in-

| | |
|---|---|
| $G = (V, E)$ : The input graph. | $k$ : number of seeds. |
| $\sigma(S)$ **or** $\sigma_{G,M}(S)$ : The influence spread of seed set $S \subseteq V$ on graph $G$ and diffusion model $M$. | |
| $\Delta(v \mid S)$ : The ***true score*** (marginal gain) of $v$ on top of $S$. $\Delta(v \mid S) = \sigma(S \cup \{v\}) - \sigma(S)$. We omit $S$ and use $\Delta(v)$ with clear context. | |
| $\bar{\Delta}[v]$ : The ***stale score*** (lazily-evaluated marginal gain) of $v$ stored in an array. It is an upper bound of $\Delta(v \mid S)$ for the current seed set $S$. | |
| $\Phi_{1..R}$ : the sketches. Formally defined in Sec. 3. | |
| $\rho$ **and** $\alpha$ : $\rho = \alpha n$ is the number of centers. | |
| **Function names:** | |
| $\textsc{Sketch}(G, r)$ : Compute the $r$-th sketch from graph $G$ | |
| $\textsc{Marginal}(S, v, \Phi_{1..R})$ : The marginal gain of vertex $v$ given the current seed set $S$ estimated from $R$ sketches $\Phi_{1..R}$ | |
| $\textsc{NextSeed}(S, \Phi_{1..R})$ : Greedily determine the next seed based on sketches $\Phi_{1..R}$ given the current seed set $S$. | |

**Table 1: Notations in the paper.**

cluding $S$) under diffusion model $M$ on graph $G$. The ***Influence Maximization (IM)*** problem is to find a subset $S^* \subseteq V$ with size $k$, s.t. $S^*$ maximizes the influence spread function $\sigma_{G,M}$. With clear context, we omit $M$ and $G$, and use $\sigma(\cdot)$. Several propagation models have been proposed in the literature, including the Independent Cascade (IC) model [32], Linear Threshold (LT) model [34, 64], and more [19, 41, 50, 62]. Since our sketch compression focuses on the IC model, we briefly introduce the IC model here. In the IC model, influence spreads in rounds. Initially, only the seed vertices are active. In round $i$, each vertex $u$ that was newly activated in round $i-1$ attempts spread the influence via all incident edges $e$, and activate the other endpoint $v$ with probability $p_e$.

Kempe et al. [41] proved that IM under the IC model is NP-hard, and that the influence spread function has the following properties: for every $X, Y \subseteq V$ where $X \subseteq Y$, and $v \in V \setminus Y$, we have:

$$\textbf{Monotonicity:} \quad \sigma(Y \cup \{v\}) \geq \sigma(Y) \tag{1}$$

$$\textbf{Submodularity:} \ \sigma(X \cup \{v\}) - \sigma(X) \geq \sigma(Y \cup \{v\}) - \sigma(Y) \tag{2}$$

These two properties allow the following **greedy algorithm** (later referred to as *GeneralGreedy*) to give a $(1 - 1/e)$-approximation. The algorithm starts with $S = \emptyset$ and repeatedly adds the vertex with the highest ***marginal gain*** to $S$, until $|S| = k$. The marginal gain $\Delta(v \mid S)$ of a vertex $v$ given the current seed set $S$ is defined as:

$$\Delta(v \mid S) = \sigma(S \cup \{v\}) - \sigma(S) \tag{3}$$

With clear context, we omit $S$ and use $\Delta(v)$, and also call it the ***score*** or the ***true score*** of $v$. We call the process to compute the true score of a vertex an ***evaluation***. To estimate $\sigma(\cdot)$, early solutions average $R'$ rounds of Monte Carlo (MC) experiments of influence diffusion simulation. However, on real-world graphs, this approach requires a large value of $R'$ to converge, which can be expensive.

**Sketch-Based Algorithms.** Sketch-based algorithms are proposed to accelerate influence spread simulation. Instead of running independent MC experiments in every evaluation, sketch-based algorithms statically sample $R$ graphs, each reflecting an MC experiment, and always simulate and average the result on the $R$ sampled graphs for any evaluation. As mentioned in Sec. 1, using sketches allows for much faster convergence, with the number of simulations (i.e., sketches) $R$ smaller than that in MC experiments. Therefore, sketch-based algorithms are widely studied [17, 21–23, 56, 70, 71]. We summarize sketch-based algorithms in two steps (see Alg. 1): *sketch construction* and *seed selection*. Next, we briefly introduce

**Algorithm 1:** Sketch-based IM algorithm

**Notations:** $G = (V, E)$: the input graph. $k$: the number of seed vertices. $R$: the number of sampled graphs
**Output:** $S$: a set of $K$ seeds that maximizes influence on $G$
**Notes:** $\Phi_{1..R}$: $R$ sketches computed from $R$ sampled graphs

*// Step 1: Sketch construction*
**ParallelForEach** $r \leftarrow 1 \ldots R$ **do**
  $\Phi_r \leftarrow \textsc{Sketch}(r)$ *// Compute the r-th sketch*
*// Step 2: Seed selection using CELF*
$S \leftarrow \emptyset$
**while** $|S| < k$ **do**
  $s^* \leftarrow \textsc{NextSeed}(S, \Phi_{1..R})$ *// Find* $\arg\max_{v \in V} \textsc{Marginal}(S, v, \Phi_{1..R})$
  $\textsc{MarkSeed}(s^*, \Phi_{1..R}, S)$ *// Mark $s^*$ as a seed in the sketches*
  $S \leftarrow S \cup \{s^*\}$
**return** $S$

**Algorithm 2:** Sequential Seed Selection with CELF

**Notes:** $Q$: max-priority-queue on all vertices $v \in V$ with key $\bar{\Delta}[v]$
Initially $\bar{\Delta}[v] = \textsc{Marginal}(\emptyset, v, \Phi_{1..R})$

**Function** $\textsc{NextSeed}(S, \Phi_{1..R})$ *// S: current seed set; $\Phi_{1..R}$: R sketches*
  **Repeat**
    $s^* \leftarrow Q.\textsc{Pop}()$ *// Pop: find and remove the top*
    $\bar{\Delta}[s^*] \leftarrow \textsc{Marginal}(S, s^*, \Phi_{1..R})$
    **if** $\bar{\Delta}[s^*] < Q.\textsc{Top}()$ **then return** $s^*$
    **else** $Q.\textsc{Insert}(s^*)$ *// insert $s^*$ back with new score*

both steps, and some useful techniques from previous work. We summarize some related approaches in Tab. 2, and review more existing work in Sec. 6. Some useful notations are given in Tab. 1.

**Sketch Construction.** In the earliest sketch-based algorithm *StaticGreedy*, $R$ sampled graphs [22] are explicitly stored as sketches. In the IC model, the $r$-th sketch corresponds to a sampled graph $G'_r = (V, E'_r)$, where $E'_r \subseteq E$, such that each edge $e \in E$ is sampled with probability $p_e$, meaning a successful activation. An evaluation will average the number of reachable vertices on all sampled graphs from the seed set $S$. A later paper *Infuser* proposed the ***fusion*** optimization [31], which uses hash functions to avoid explicitly storing the sampled graphs $G'_r$. They sample an edge $e$ in a sketch $G'_r$ with a random number generated from seed $\langle e, r \rangle$, such that whether an edge is selected in a certain sketch is always deterministic, and a sampled graph $G'_r$ can be fully reconstructed from the sketch id $r$. We also use this idea in our sketch compression algorithm.

Many existing algorithms also use ***memoization*** to avoid influence spread simulation on sketches. On undirected graphs and the IC model, the *MixGreedy* paper [21] first observed that a vertex $v$'s influence on a sketch is all vertices in the same connected component (CC) as $v$, but they only used this idea to select the first seed. The *Infuser* system adopts this idea to select all seeds, and proposes *InfuserMG*, which memoizes the CC information of each sampled graph as the sketch. A vertex $v$'s score is then the average of the (inactivated) CC sizes on all $R$ sketches, which can be obtained in $O(R)$ cost. This approach avoids simulation, but leads to $O(Rn)$ space that is expensive for large graphs. Sec. 3 presents how our sketch compression approach reduces this high space usage.

**Seed Selection with CELF Optimization.** Another useful op-

| | Randomization | Compute Influence | Select Seed | #vertics per seed | Space | Parallel |
|---|---|---|---|---|---|---|
| ***GeneralGreedy* [41]** | $R'$ Monte Carlo experiments | Simulation | Evaluate all | $O(nR'T)$ | $O(n)$ | no |
| ***MixGreedy* [21] 1st Seed** | Fixed $R$ sampled graphs | Memoization | Evaluate all | $O(nR)$ | $O(n)$ | no |
| ***MixGreedy* [21] Others** | $R'$ Monte Carlo experiments | Simulation | CELF | $O(n_cR'T)$ | $O(n)$ | no |
| ***StaticGreedy* [22]** | Fixed $R$ sampled graphs | Simulation | CELF | $O(n_cRT)$ | $O(n)$ | no |
| ***InfuserMG* [31]** | Fixed $R$ sampled graphs | Memoization | CELF | $O(n_cR)$ | $O(nR)$ | yes |
| ***PaC-IM* (this work)** | Fixed $R$ sampled graphs | Simulation + Memoization | CELF | $O(n_cR \cdot \min(T, 1/\alpha))$ | $O((1+\alpha R)n)$ | yes |

**Table 2: Existing and our new approaches.** *MixGreedy* uses different approaches to select the first seed and the other seeds, so we list them separately. "#vertices per seed": number of vertices to visit in all re-evaluations involved to find a seed. For *StaticGreedy*, we assume the *fusion* optimization in [31] to avoid explicitly storing sampled graphs. $n$: number of vertices. $n_c$: number of re-evaluations needed in CELF. $T$: the average number of reachable vertices in a simulation (or a sketch). Empirically, $T$ is large, and $n_c \ll n$. To achieve similar quality, $R \ll R'$.

timization for the greedy algorithm is CELF [47], which avoids evaluating all vertices to select the next seed. CELF uses *lazy* evaluation for *submodular* functions (defined in (2) ). In CELF, a vertex $u$ is evaluated only if it becomes "promising" as a candidate of the next seed. We show the CELF-based seed selection algorithm in Alg. 2. The algorithm uses a priority queue $Q$ to maintain all vertices with their scores as the key. Due to submodularity, $\Delta(v)$ is non-increasing with the expansion of the seed set $S$. With lazy evaluation, the scores in $Q$ may be stale, but are always upper bounds of the true score. To distinguish it from the true score, we call this lazily-evaluated score the ***stale score*** of $v$, and denote it as $\bar{\Delta}[v]$ stored in an array. To select the next seed, CELF keeps popping the top element $v$ from $Q$, re-evaluating its true score $\Delta(v)$, and inserting it back unless $\Delta(v)$ is greater than the current top $\bar{\Delta}[\cdot]$ value in $Q$ (Line 3-Line 6). In this case, we can set $v$ as the next seed without further examining the remaining vertices in $Q$, since their true score can only be lower than the values in $Q$. In many cases, CELF can reduce the number of re-evaluations, but it is essentially sequential and evaluates all vertices one by one. In Sec. 4, we present our new data structures that allow for parallel re-evaluations in CELF.

## 3 Space-Efficient Sketches

This section presents our new technique to construct compressed sketches in parallel for the IC model on undirected graphs. In this setting, as discussed in Sec. 2, a vertex $v$ can activate all vertices in the same CC on a certain sketch. Thus, *memoizing* per-vertex CC information in sketches [31] can accelerate the influence evaluation, but requires $O(Rn)$ space, which does not scale to large input graphs. Alternatively, one can avoid memoization and run a *simulation* by traversing the sampled graph to find the CC when needed. This requires no auxiliary space, but can take significant time.

Our approach combines the benefit of both, using bounded-size auxiliary space while allowing for efficiency. Our key idea is to only *partially memoize* CC information in each sketch, and retrieve this information by *partial simulations*. In *PaC-IM*, we only store the CC information for $\rho = \alpha n$ ***center*** vertices $C \subseteq V$, where $0 \le \alpha \le 1$ is a user-defined parameter. Each sketch $\Phi_r$ is a triple $\langle r, label[\cdot], size[\cdot]\rangle$ corresponding to an implicit sampled graph $G'_r$ from $G$, where each edge $e = (u, v)$ is retained with probability $p_e$.

- $r$: the sketch id. Similar to the fusion idea mentioned in Sec. 2, the sketch id fully represents the sampled graph.
- $label[1..\rho]$: the CC label of center $c_i$ on this sketch. It is the smallest center id $j$ where $c_j$ is in the same CC as $c_i$.
- $size[1..\rho]$: if $label[i] = i$ (i.e., $i$ represents the label of its CC), $size[i]$ is the influence of center $c_i$ on this sketch. It is initially the CC size of $c_i$, and becomes 0 when any vertex in this CC is selected as a seed.

**Algorithm 3:** Our sketch algorithm with compression

**Global Variables:** $G = (V, E)$: the input graph
$C = \{c_1, c_2, \ldots, c_\rho\} \subseteq V$: randomly selected *centers*. $\rho = |C|$
$R$: the number of sketches

**Notes:** A sketch $\Phi_r$ is a triple $\langle r, size[1..\rho], label[1..\rho]\rangle$, defined at the beginning of Sec. 3.

```
1  Function Sketch(r)                                          // r: sketch id
2      Compute the connected components of graph G' = (V, E'), where
         E = {(u,v) | (u,v) ∈ E, Sample(u,v,r) = true}
3      ParallelForEach c_i ∈ C do
4          label[i] ← min_j {c_j is in the same CC as c_i}
5      ParallelForEach c_i ∈ C do
6          if label[i] = i then size[i] ← the CC size of center c_i
7      return ⟨r, label[1..ρ], size[1..ρ]⟩
   // sample an edge e with probability p_e for sketch r
8  Function Sample(e, r)                          // e: edge identifier; r: sketch identifier
9      p ← random(e, r)          // Generate p ∈ [0,1] from random seed e, r
10     return p ≤ p_e
   // δ: marginal influence of v on sketch Φ_r. l: label of v's CC in sketch Φ_r if v
      is connected to any center; otherwise l = −1. S: the current seed set.
11 Function ⟨δ, l⟩ = GetCenter(Φ_r, v, S)
12     Start BFS from v on a sampled subgraph of G, where an edge e ∈ E
         exists if Sample(e, r) = true. Count the #reached-vertices as n'
13     if a center c_i ∈ C is encountered during BFS then
14         l ← Φ_r.label[i]                                    // Find the label of the center
15         return ⟨Φ_r.size[l], l⟩          // the CC size and label of the center
16     if any v' ∈ S has been visited by BFS then return ⟨0, −1⟩
17     else return ⟨n', −1⟩                  // influence is #visited vertices in BFS
   // Marginal gain of v given seed set S on sketches Φ_{1..R}
18 Function Marginal(S, v, Φ_{1..R})
19     ParallelForEach r ← 1..R do
20         ⟨δ_r, ·⟩ ← GetCenter(Φ_r, v, S)
21     return (Σ_{r=1}^{R} δ_r)/R          // the sum can be computed in parallel
22 Function MarkSeed(v, Φ_{1..r}, S)
23     ParallelForEach r ← 1..R do                             // For each sketch Φ_r
24         ⟨δ, l⟩ ← GetCenter(Φ_r, v, S)
25         if l > 0 then Φ_r.size[l] ← 0   // Clear the corresponding CC size
```

**Algorithm Overview**. We first present the high-level idea of our sketch compression algorithm. An illustration is presented in Fig. 2. As mentioned, we select $\rho = \alpha n$ center vertices *uniformly at random*. We only store the CC information (label and size) for the centers in sketches. We use a global flag array to indicate if a vertex is a center, and thus the total space usage is $O((1+\alpha R)n)$. To query the CC size

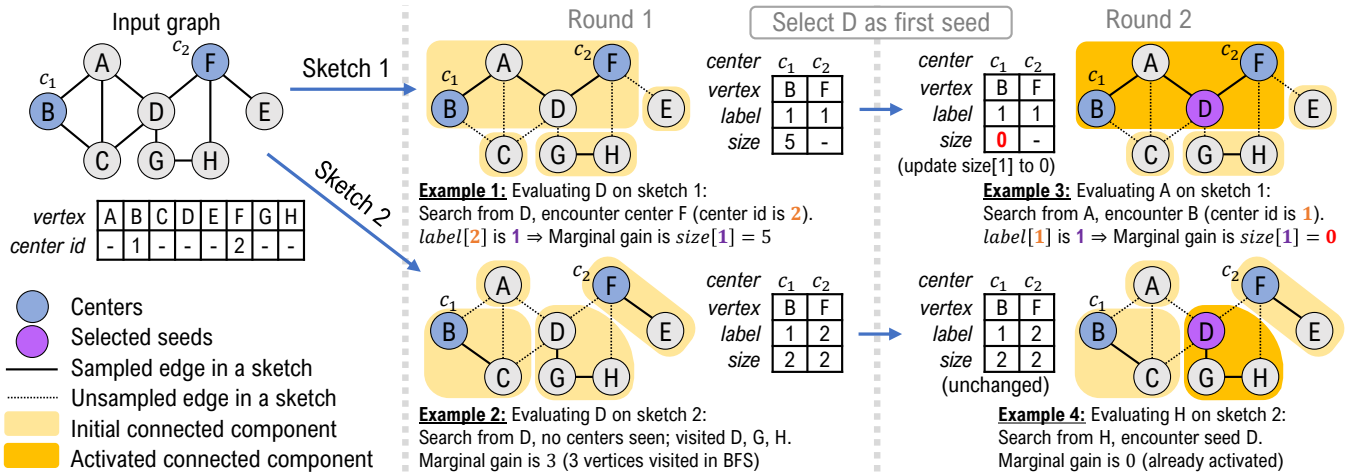


**Figure 2: An example of our sketch compression** on a graph with 8 vertices (B and F as centers) and $R = 2$ sampled graphs.

of an arbitrary vertex $v$ on sketch $\Phi_r$, we will start a breadth-first search (BFS) from $v$. When any center $c_i$ is encountered, we know that $v$ should activate the same set of vertices as $c_i$ on this sketch. As such, we can stop searching and use the influence (CC size) of $c_i$ as the influence for $v$. If $v$ is not connected with any center on the sampled graph, the CC containing $v$ is likely small, and the BFS can visit all of them quickly. In either case, the number of vertices visited by the BFS can be bounded. We show in Thm. 3.1 that limiting the auxiliary space by a factor of $\alpha$ roughly increases the evaluation time by a factor of $O(1/\alpha)$. By controlling the number of centers $\rho = \alpha n$, we can achieve a tradeoff between the time to query the score of each vertex and the space usage.

We present our algorithm in Alg. 3. Next, we will elaborate on the three functions needed in Alg. 1: SKETCH$(G, r)$, which constructs the $r$-th sketch from input graph $G$, MARGINAL$(S, v, \Phi_{1..R})$, which computes the score (marginal gain) of a vertex $v$ on top of $S$ using sketches $\Phi_{1..R}$, and MARKSEED$(s^*, \Phi_{1..R})$, which adjusts the sketches $\Phi_{1..R}$ when $s^*$ is selected as a seed.

**Sketch Construction (SKETCH$(G, r)$).** Recall that we maintain CC information for $\rho = \alpha n$ centers $C = \{c_1, c_2, \ldots, c_\rho\}$ in sketches. To construct a sketch $\Phi_r$, we first compute the CC information of the sampled graph $G'_r$, which can be performed by any parallel connectivity algorithm [26]. We store the CC information for all centers in two arrays. $\Phi_r.label[i]$ records the label of CC of the $i$-th center. For multiple centers in the same CC, we simply use the smallest CC id as the label for all of them to represent this CC, so all such centers refer to the label to find the CC information. For a center $c_i$, If $i$ is the label of its CC, we use $\Phi_r.size[i]$ to record the size of this CC. An example of these arrays is given in Fig. 2.

**Computing the Marginal Gain (MARGINAL$(S, v, \Phi_{1..R})$).** Given the sketches $\Phi_{1..R}$ and the current seed set $S$, the function MARGINAL$(S, v, \Phi_{1..R})$ computes the marginal gain of a vertex $v$. Our algorithm loops over all sketches and sums up the marginal gains of $v$ on all of them. We use a helper function $\langle \delta, l \rangle$ = GETCENTER$(\Phi_r, v, S)$, which returns $\delta$ as the marginal gain of $v$ on sketch $\Phi_r$, and $l$ as the label of centers connected to $v$ ($l = -1$ if no center is connected to $v$). This function will run a breadth-first search (BFS) from $v$ on $G'_r$ (i.e., only using edges $e \in E$ s.t. SAMPLE$(e, r)$ is *true*). If any center is encountered during this BFS (Examples 1 and 3 in Fig. 2), then the influence of $v$ is the same as $c_i$ on this sketch. The information of $c_i$ is retrieved by its label $l = \Phi_r.label[i]$, and thus $\delta = \Phi_r.size[l]$. The influence $\delta$ is either the size of the CC containing $v$ when no vertices in this CC are seeds (Example 1 in Fig. 2), or 0 otherwise, as is updated in MARKSEED (Example 3 in Fig. 2). Otherwise, if the BFS terminates without visiting any centers, the returned label $l$ will be $-1$. If any seed is visited during BFS, then the marginal gain is $\delta = 0$ (Line 16, Example 4 in Fig. 2). Otherwise $\delta$ is the number of vertices $n'$ visited during BFS, which is also the size of CC containing $v$ (Example 2 in Fig. 2). Using the GETCENTER function, the marginal influence of $v$ on sketch $\Phi_r$ can be obtained as the first return value $\delta$ of GETCENTER$(\Phi_r, v, S)$.

**Marking a Seed (MARKSEED$(s^*, \Phi_{1..R})$).** This function updates the sketches when $s^* \in V$ is selected as a seed. For each sketch $\Phi_r$, the CC label of $s^*$ is the second return value $l$ of GETCENTER$(\Phi_r, s^*, S)$. If $l \neq -1$, we set $\Phi[r].size[l]$ as 0—since $s^*$ is selected, all other vertices in this CC will get no marginal gain on this sketch.

Our sketch compression scheme allows for a tradeoff between the space and the query time in MARGINAL(): a smaller $\rho$ (fewer centers) leads to less space, but increases the cost to evaluate vertices, as it may take longer to find a center. It is worth noting that *PaC-IM* unifies and is a hybrid of *StaticGreedy* and *InfuserMG*. Theoretically, using $\alpha = 1$, our sketch is equivalent to *InfuserMG* where the CC information for all vertices on all sketches are memoized; using $\alpha = 0$, our sketch is equivalent to *StaticGreedy* where no CC information is maintained, and evaluations are done by traversing the sampled graph. In practice, *PaC-IM* is much faster than *StaticGreedy* and *InfuserMG* even when with no compression due to better parallelism and the techniques in Sec. 4. We summarize the theoretical guarantees in Tab. 2, and state them in Thm. 3.1.

THEOREM 3.1. *PaC-IM with parameter $\alpha$ requires $O((1 + \alpha R)n)$ space to maintain $R$ sketches, and visits $O(R \cdot \min(1/\alpha, T))$ vertices to re-evaluate the marginal gain of one vertex $v$, where $T$ is the average CC size of $v$ on all sketches.*

*Proof.* Assume that no connectivity is stored in the sketches, then running a simulation for a vertex will visit $RT$ vertices, based on the definition of $T$. This is also what the baseline algorithm *StaticGreedy*

does. We now consider the case that $\rho = \alpha n$ centers are selected. Note that all centers are picked independently and uniformly at random. This means that each visited vertex in the BFS has the probability of $\alpha$ that is a center. Let us first focus on a search on a specific sketch $r$, and assume the CC size of $v$ is $T_r$. The search terminates either 1) all $T_r$ vertices have been visited, or 2) a center is encountered. Therefore, when visiting the $T'$ vertices in the BFS order, each of them stops the search with a probability of $\alpha$. The expected number of visited vertices in the BFS is therefore also bounded by $\sum_{i=1}^{\infty}(1-\alpha)^{i-1}\alpha \cdot i$, which solves to $1/\alpha$. The total number of visited vertices in all BFS is $\sum_r \min(T_r, 1/\alpha) = \min(RT, R/\alpha) = R \cdot \min(T, 1/\alpha)$. □

black

## 4 Parallel Priority Queues for Seed Selection

We now present the parallel seed selection process in *PaC-IM*. We call each iteration in seed selection (selecting one seed) a ***round***. Recall that prior solutions use the CELF optimization (see Tab. 2 and Sec. 2), which maintains (possibly stale) scores of all vertices in a priority queue $Q$, and updates them lazily. In each round, CELF pops the vertex with the highest (stale) score from $Q$, and re-evaluates it. The process terminates when a newly evaluated score is higher than all scores in $Q$; otherwise, the new score will be inserted back to $Q$. For simplicity, we assume no tie between scores. In the cases when ties exist, both our approach and CELF break the tie by vertex id. Let $F_i = \{v \mid \bar{\Delta}_{i-1}[v] \geq \Delta^*\}$ be the set of vertices re-evaluated by CELF in round $i$, where $\bar{\Delta}_{i-1}[v]$ is the stale score of $v$ after round $i-1$, and $\Delta^*$ is the maximum true score in round $i$ (i.e., the score of the chosen seed in round $i$). black However, the CELF process is inherently sequential. Hence, some existing parallel implementations (e.g., [31, 54]) only parallelize the re-evaluation function MARGINAL (Line 4 in Alg. 2), but leave the CELF process sequential, i.e., all $|F_i|$ evaluations are performed one by one. When $F_i$ is large, this results in low parallelism.

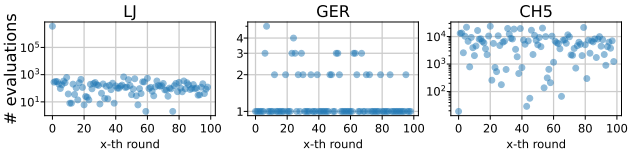


**Figure 3:** black

**Prior Work on Parallel Priority Queue**. As a fundamental data type, parallel priority queues are widely studied [10, 13, 27, 63, 68, 69, 76]. However, as far as we know, all these algorithms/interfaces require knowing the batch of operations (e.g., the threshold to extract keys, or the number of keys to extract) *ahead of time*. This is not true in CELF—the set $F_i$ is only known during the execution. We need different approaches to tackle this challenge.

**Overview of Our New Approaches**. We first formalize the interface of the parallel priority queue needed in the IM framework in Alg. 1. The data structure maintains an array $\bar{\Delta}[\cdot]$ of (stale) scores for all vertices. It is allowed to call MARGINAL function to re-evaluate and obtain the true score of any vertex. The interface needs to support NEXTSEED function, which returns the vertex id with the highest true score.

**Algorithm 4:** Seed Selection based on *P-tree*

**Maintains:** A parallel binary search tree $T$ for all $\bar{\Delta}[v]$.

**Function** NEXTSEED($S, \Phi_{1..R}$)
  $s^* \leftarrow \perp$ *// The best seed so far*
  $j \leftarrow 0$
  **repeat**
    *// Extract (remove & output) the top $2^j$ elements in $T$ into array $B_j$*
    $B_j[1..2^j] \leftarrow T$.SPLITANDREMOVE($2^j$)
    **ParallelForEach** $v \in B_j$ **do** *// Get the true score for each $v \in B_j$*
      $\bar{\Delta}[v] \leftarrow$ MARGINAL($S, v, \Phi_{1..R}$)
    $t \leftarrow \arg\max_{v \in B_j} \bar{\Delta}[v]$ *// can be computed in parallel*
    **if** ($s^* = \perp$) OR ($\bar{\Delta}[t] > \bar{\Delta}[s^*]$) **then** $s^* \leftarrow t$
    $j \leftarrow j + 1$
  **until** $\bar{\Delta}[s^*] > T$.MAX() *// $\bar{\Delta}[s^*]$ is better than the top in $T$*
  $T$.BATCHINSERT($\bigcup_{j'=0}^{j-1} B_{j'} \setminus \{s^*\}$)
  **return** $s^*$

We present two parallel data structures to maintain the lazily evaluated scores in parallel. Our first approach is based on a parallel binary search tree (BST), the ***P-tree*** [11, 13, 69]. We prove that using *P-tree*s, our approach has work (number of re-evaluations) asymptotically the same as that in CELF, while is highly parallel: the $i$-th seed selection finishes in rough $\log |F_i|$ iterations of re-evaluations (each iteration evaluates multiple vertices in parallel), instead of $|F_i|$ iterations (each iteration evaluate one vertex) in CELF. We also propose a new data structure based on parallel winning trees (aka. tournament trees), and refer to it as ***Win-Tree***. *Win-Tree* does not maintain the total order of the scores, and is simpler and potentially more practical than *P-tree*s. We introduce the *P-tree*-based approach in Sec. 4.1 and *Win-Tree*-based approach in Sec. 4.2, and compare their performance in Sec. 5.3. *These two approaches are **independent of the sketching algorithm** and apply to seed selection on all submodular diffusion models* (not necessarily the IC model and/or on undirected graphs). Note that most diffusion models for IM are submodular (e.g., Independent Cascade (IC), Linear Threshold (LT), Triggering (TR [41]), and more [41, 72, 79]).

### 4.1 Parallel Priority Queue Based on *P-tree*

Our first approach to parallelizing CELF is to maintain the total (decreasing) order of the scores of all vertices, using a parallel binary search tree (BST) called *P-tree* [11, 13]. We will use two functions on a *P-tree* $T$: 1) $T$.SPLITANDREMOVE($k$), which extracts (removes and outputs to an array) the first $k$ tree nodes ($k$ largest scores) from $T$, and 2) $T$.BATCHINSERT($B$), which inserts a set of keys $B$ to $T$. Both algorithms are parallel with a polylogarithmic span.

Different from many priority queue implementations (e.g., binary heap) that extract the top element one at a time, *P-tree* maintains the total order of all vertices, and thus we can extract a batch of vertices with top (stale) scores and evaluate them in parallel. The key challenge here is to design a similar stop condition as CELF to avoid evaluating too many vertices, since we are unaware of the number of "useful" re-evaluations, i.e., $|F_i|$, ahead of time. Our idea is to use *prefix doubling* [15, 25, 35, 36, 65] to achieve both work efficiency and high parallelism. The pseudocode is given in Alg. 4 with an illustration in Fig. 4. To find the next seed, the *P-tree* $T$ starts with the stale scores $\bar{\Delta}[\cdot]$ from the previous round. It will then extract the top (largest stale score) nodes in batches of size

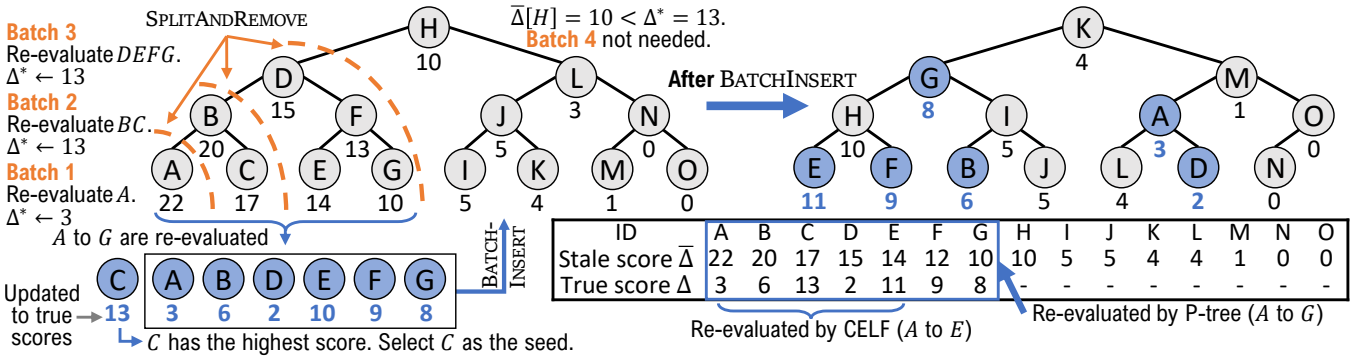


**Figure 4: Example of *P-tree*-Based Seed Selection.** The letters in the tree nodes represent vertices, and the numbers below them are their stale scores. *P-tree* maintains decreasing order of the (stale) scores. By prefix-doubling, we extract batches of 1, 2, 4 vertices and evaluate each batch in parallel. After batch 3, the highest true score (13) is higher than the current best in the tree (10), and the algorithm stops. We will select the node with the highest true score and insert the rest back to the tree with their new score. *P-tree* may evaluate more vertices than CELF, but the extra work can be bounded (Thm. 4.2).

1, 2, 4, 8, ... from $T$ (Line 5). Within each batch $B_j$, we re-evaluate all vertices in parallel (Line 7). These new scores are used to update the current best seed $s^*$. The loop terminates when the score of $s^*$ is better than the best score in $T$ (Line 11). Finally, the algorithm selects $s^*$ as the seed and inserts the rest of the new true scores $\bigcup B_j \setminus \{s^*\}$ back to $T$.

Due to prefix doubling, each seed selection finishes in at most $O(\log n)$ rounds. Note that our approach evaluates more vertices than CELF, but due to the stop condition (Line 11), the extra work is bounded by a constant factor (proved in Thm. 4.2). Next, we prove the correctness and efficiency of the algorithm.

Theorem 4.1 (*P-tree* Correctness). *Alg. 4 always selects the next seed with the largest marginal gain, i.e.,* $\Delta(s^*) = \max_{v \in V} \{\Delta(v)\}$.

*Proof.* Let $s^*$ be the vertex selected by the algorithm, and we will show $\Delta(s^*) \geq \Delta(v)$ for all $v \in V$ when the stop condition on Line 11 is triggered. We first show that $\Delta(s^*) \geq \Delta(v)$ for all $v \in V$ that has been split from the tree. This is because all such vertices have been re-evaluated, and $s^*$, by definition, has the highest true score among them. We then show $s^*$ has a higher true score than any other vertices still in $T$. For any $u \in T$, the stop condition indicates $\Delta(s^*) = \bar{\Delta}[s^*] > \bar{\Delta}[u] \geq \Delta(u)$ (due to submodularity). Therefore, $\Delta(s^*)$ is the highest true score among all vertices. □

Theorem 4.2 (*P-tree* Efficiency). *Alg. 4 has the total number of re-evaluations at most twice that of CELF.*

*Proof.* Recall that $F_i$ is the set of re-evaluated vertices by CELF when selecting the $i$-th seed. Let $F'_i$ be the set of re-evaluated vertices by *P-trees* in Alg. 4. We first show a simple case—if both CELF and Alg. 4 start with the same stale scores $\bar{\Delta}_{i-1}[\cdot]$, then $|F'_i| \leq 2|F_i|$. Let us reorder vertices in $V$ as $v_1, v_2, \ldots, v_n$ by the decreasing order of their stale score $\bar{\Delta}_{i-1}[\cdot]$. Assume $v_l$ is the last vertex evaluated by CELF, so $|F_i| = l$. $v_l$ must be in the last batch in *P-tree*. Assume the last batch is batch $j$ with $2^j$ vertices. This indicates that all $2^j - 1$ vertices in the previous $j-1$ batches are before $v_l$. Therefore $2^j - 1 < l = |F_i|$, and $|F'_i| = 2^{j+1} - 1$, which proves $|F'_i| \leq 2|F_i|$.

We now consider the general case. We first focus on a specific seed selection round $i$. Due to different sets of vertices re-evaluated in each round, at the beginning of round $i$, CELF and *P-tree* may not see exactly the same stale scores. We denote the stale score *at the beginning of round i* in CELF as $\bar{\Delta}_{CELF}[\cdot]$ and that for *P-tree* as $\bar{\Delta}_{BST}[\cdot]$. We reorder vertices by decreasing order of $\bar{\Delta}_{CELF}[\cdot]$ as $v_1, v_2, \ldots, v_n$, and similarly for $\bar{\Delta}_{BST}[\cdot]$ as $u_1, u_2, \ldots, u_n$. Denote $\Delta^*$ as the highest true score in round $i$. Let $v_{x_i}$ be the last vertex in $v_{1..n}$ such that $\bar{\Delta}_{CELF}[v_{x_i}] \geq \Delta^*$, and $u_{y_i}$ the last vertex in $u_{1..n}$ such that $\bar{\Delta}_{BST}[u_{y_i}] \geq \Delta^*$. Namely, $x_i$ is the rank of $\Delta^*$ in $v_{1..n}$, and $y_i$ is the rank of $\Delta^*$ in $u_{1..n}$. By definition, $F_i$ is exactly the first $x_i$ vertices in sequence $v$, and thus $x_i = |F_i|$; $F'_i$ contains all vertices smaller than $u_{y_i}$ and possibly some more in the same batch with $u_{y_i}$, so $|F'_i| \leq 2y_i$. In the simple case discussed above where we assume $\bar{\Delta}_{CELF}[\cdot] = \bar{\Delta}_{BST}[\cdot]$, we always have $x_i = y_i$.

We will use amortized analysis to show that $\sum y_i \leq \sum x_i$, which further indicates $\sum |F'_i| \leq 2 \cdot \sum |F_i|$. We can partition $u_{1..y_i}$ into two categories: $U_1$ as the intersection of $u_{1..y_i}$ and $v_{1..x_i}$, and $U_2$ as the rest. Note that for $t \in U_2$, $\bar{\Delta}_{BST}[t] \geq \Delta^* > \bar{\Delta}_{CELF}[t]$. This happens iff. $t$ is evaluated by CELF in a previous round, but not by *P-tree*. Namely, there exists a round $j$, such that $t \in F_j$ but $t \notin (F'_j \cup F'_{j+1} \cup \ldots \cup F'_{i-1})$. Let $v^*_{1..n}$ and $u^*_{1..n}$ be the $u$ and $v$ sequences in round $j$, respectively. Note that $F_j = v^*_{1..x_j}$, and $u^*_{1..y_j} \subseteq F'_j$. Since $t \in F_j$, then vertex $t$ was counted in $x_j$. Since $t \notin F'_j$, $t$ is not counted in any $y_j$. Therefore, we can save a token for such $t$ when it is evaluated by CELF in round $j$ but not by *P-tree*, such that when later $t$ is counted in $u_{1..y_i}$ in round $i$, we will use the token to count $t$ for free. Note that $t \notin (F'_j \cup F'_{j+1} \cup \ldots \cup F'_{i-1})$, for the same reason, $t$ is not counted in any $y_{j'}$ for $j \leq j' < i$, so the token must still available in round $i$. In summary, all vertices in $U_1$ are counted in $x_i$, and all vertices in $U_2$ can be counted by the saved tokens (charged to some previous $x_j$). Therefore, using amortized analysis, we have $\sum y_i \leq \sum x_i$.

Recall that $x_i = |F_i|$, and $|F'_i| \leq 2y_i$. Since $\sum y_i \leq \sum x_i$, we proved that $\sum |F'_i| \leq \sum 2y_i \leq 2 \sum x_i = 2 \sum |F_i|$. □

Theorem 4.3 (*P-tree* Cost Bound). *Given the same sketches* $\Phi_{1..R}$*, our seed selection based on P-tree will select the same seed set as CELF with* $O(n \log n + W_{CELF})$ *work and* $\tilde{O}(kD_\Delta)$ *span, where* $k$ *is the number of seeds,* $W_{CELF}$ *is the work (time complexity) by CELF, and* $D_\Delta$ *is the span to re-evaluate one vertex.*

*Proof.* Using the analysis of *P-tree* [13], SplitAndRemove have $O(\log n)$ work and span, where $n$ is the tree size. BatchInsert has $O(n' \log n)$ work and $O(\log n' \log n)$ span, where $n'$ is the size of unordered insertion batch. In addition, constructing the *P-tree* uses $O(n \log n)$ work and $O(\log n)$ span.

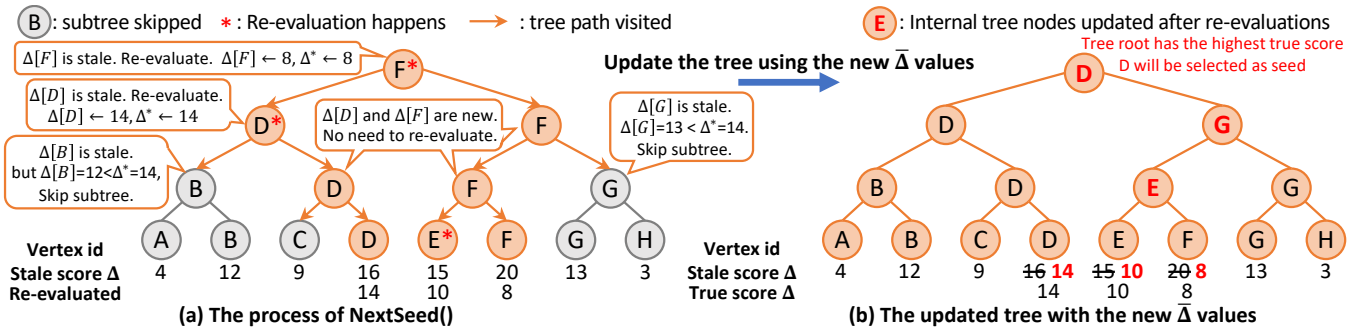


**Figure 5: Seed selection based on *Win-Tree*.** Each leaf stores a vertex id, and each internal node stores the vertex in its subtree with the highest (stale) score. (a) An example of finding the maximum score. For illustration purposes, we assume the parallel threads work at the same speed and all tree nodes on the same level are processed in parallel (in reality the threads run asynchronously in fork-join parallelism). Therefore, the subtree at $G$ will see $\Delta^* = 14$ updated by $D$, and this subtree will be skipped. (b) Updating the internal nodes with the new $\bar{\Delta}[\cdot]$ values. Finally, the root of the tree has the highest true score.

For each seed selection round that evaluates a set of vertices $F'_i$, Alg. 4 will first split them out from the tree, re-evaluate them, and insert all but one back to tree. Based on the aforementioned cost bounds, the work per element in $F'_i$ is $O(\log n)$, the same as in CELF per element in $F_i$. Thm. 4.2 indicates that $\sum |F'_i| = O(\sum |F_i|)$, so the total work for P-tree is asymptotically the same as the sequential CELF except for the $O(n \log n)$ preprocsssing cost.

For the span, note that *P-tree* needs $O(\log n)$ batches per round. Each batch requires a split, and evaluates multiple vertices in parallel. Therefore, the span is $O(D_\Delta + \text{polylog}(n))$ in each round. After all the batches are processed, finally a batch-insertion adds all but one vertices back to the tree with polylogarithmic span. Combine all pieces together, the span is $\tilde{O}(D_\Delta)$ per round, and $\tilde{O}(kD_\Delta)$ for the entire Alg. 4. □

## 4.2 Parallel Priority Queue: *Win-Tree*

While *P-tree* provides theoretical efficiency for seed selection, it maintains the total order of all vertices, which is not needed in many implementations for priority queues (e.g., a binary heap) and may cause performance overhead. Also, *P-tree* is an explicit tree that maintains the parent-child pointers, which causes additional space usage. We now propose a more practical data structure based on a winning tree that overcomes these two challenges, though it does not have the same bounds as in Thm. 4.3.

A classic *winning tree* (aka. tournament tree) is a complete binary tree with $n$ leaf nodes and $n-1$ interior nodes. The data are stored in the leaves. Each interior node records the larger key of its two children. Since a winning tree is a complete binary tree, it can be stored *implicitly* in an array $T[1..2n-1]$, which consumes smaller space. In our case, each tree node stores a vertex id (noted as $t.id$). The key of a node $t$ is the (stale) score of this vertex, i.e., $\bar{\Delta}[t.id]$. Each interior node stores the id of its children with a larger score. As a result, the vertex at the root has the highest (stale) score.

To support CELF seed selection efficiently, we use the information in the internal nodes of *Win-Tree* to prune the search process. Suppose the best true score we evaluated so far is $\Delta^*$, then if we see a subtree root $t$ with a stale score smaller than $\Delta^*$, we can skip the *entire subtree*. This is because all nodes in this subtree must have smaller stale scores than $t.id$, which indicates even smaller true scores. Although this idea is simple, we must also carefully maintain the *Win-Tree* structure, with the newly evaluated true scores. We presented our algorithm in Alg. 5 with an illustration in Fig. 5, and elaborate on more details below.

**Algorithm 5:** Seed Selection based on *Win-Tree*

**Maintains:** Global variable $\Delta^*$: the highest true score evaluated so far
A winning tree $T$ with $n$ leaf nodes each storing a record.
For a tree node $t \in T$, we use the following notations:
$t.id$: the id of the vertex stored in this node
$t.parent$ / $t.left$ / $t.right$ : the parent/left child/right child of node $t$
The *Win-Tree* is a max-priority-queue based on (stale) score $\bar{\Delta}[t.id]$ for each vertex.

```
1  Function FindMax(tree node t, current seed set S, sketches Φ_{1..R})
2      if t.id = t.parent.id then stale ← false        // evaluated by parent
3      else stale ← true
       // Skip a subtree if the max is stale and is smaller than the current best
          score; no re-evaluation needed for the entire subtree
4      if stale = true and Δ̄[t.id] < Δ* then return
5      if stale = true then      // Current value is stale, re-evaluation needed
6          Δ̄[t.id] ← MarginalGain(S, t.id, Φ_{1..R})      // Re-evaluate
7          WriteMax(Δ*, Δ̄[t.id])      // update the best score so far
8      if t is a leaf then return
9      In Parallel:
10         FindMax(t.left, S, Φ_{1..R})
11         FindMax(t.right, S, Φ_{1..R})
       // compare two branches and reset max
12     if Δ̄[t.left.id] > Δ̄[t.right.id] then t.id ← t.left.id
13     else t.id ← t.right.id
14 Function NextSeed(S, Φ_{1..R})
15     Δ* ← 0
16     FindMax(T.root, S, Φ_{1..R})
17     return T.root.id
```

To implement NextSeed, we keep a global variable $\Delta^*$ to track the largest true score obtained so far, initialized to 0. The algorithm calls the FindMax$(t, \dots)$ subroutine starting from the root, which explores the subtree rooted at $t$. We first determine if the score of the vertex at $t$ is stale: when $t$'s id is the same as its parent, this vertex has been re-evaluated and has the true score ready (Line 2). Based on the node's status, there are three cases. First, if the score is stale, and is already lower than $\Delta^*$, accordingly to the discussion above, we can skip the entire subtree and terminate the function (Line 4). Second, if the score is stale, but is higher than $\Delta^*$, we have to re-evaluate the vertex $t.id$, since it may be a candidate for the seed (Line 6). We then use the atomic operation WriteMax to update

$\Delta^*$ by this true score if it is better. The third case is when the score is not stale. Although no re-evaluation is needed on $t.id$, we still have to further explore the subtree, since with the newly evaluated score, the subtree structure (i.e., the ids of the internal nodes) may change. Therefore, in both cases 2 and 3, we recursively explore the two subtrees in parallel (Lines 9 to 11). After the recursive call returns, we re-compute the vertex id by comparing its left and right children, and use the one with a higher score.

black

Unlike *P-tree*s, we cannot prove strong bounds for the number of re-evaluations in *Win-Tree*—since the parallel threads are highly asynchronous, the progress of updating $\Delta^*$ and pruning the search cannot be guaranteed. However, we expect *Win-Tree* to be more practical than *P-tree* for a few reasons. First, *Win-Tree* is a complete binary and can be maintained in an array, which requires smaller space (no need to store metadata such as pointers in *P-tree*s). Second, the *P-tree* algorithm requires $O(\log n)$ batches and synchronizing all threads between batches. Such synchronization may result in scheduling overhead, while the *Win-Tree* algorithm is highly asynchronous. Most importantly, *Win-Tree* does not maintain the total order, and the construction time is $O(n)$ instead of $O(n \log n)$. In Sec. 5.3, we experimentally verify that although *Win-Tree* incurs more re-evaluations than *P-tree*s, it is faster in most tests.

## 5 Experiments

**Setup.** We implemented *PaC-IM* in C++. We run our experiments on a 96-core (192 hyperthreads) machine with four Intel Xeon Gold 6252 CPUs, and 1.5 TB of main memory. We use `numactl -i all` in experiments with more than one thread to spread the memory pages across CPUs in a round-robin fashion. We run each test for 4 times and report the average of the last 3 runs.

We tested 17 graphs with information shown in Tab. 3. To understand the generality of our approaches, we include real-world graphs with a wide range of sizes and distributions, including **five billion-scale or larger graphs**. In addition to the commonly-used benchmarks of social networks, we also include web graphs, road networks, and $k$-NN graphs (each vertex is a multi-dimensional data point connecting to its $k$-nearest neighbors [75]). Solving IM on such graphs simulates the influence spread between websites (web graphs), geologically connected objects (road networks), and geometrically close objects ($k$-NN graphs). Based on graph patterns, we call social and web graphs ***scale-free*** graphs, and the rest ***sparse*** graphs. We symmetrize the directed graphs to make them undirected. *PaC-IM* (with compression) is the only tested system that can process the largest graph ClueWeb [52] with 978M vertices and 74B edges. Even with 1.5TB memory, *PaC-IM* can process Clueweb only when $\alpha \leq 0.25$ (4× or more compression ratio in sketches), which shows the necessity of compression.

We select $k = 100$ seeds in all tests. We use the IC model with a constant propagation probability $p$ in the same graph. black When comparing the ***average*** numbers across multiple graphs, we use the ***geometric mean***. Since we picked many graphs with different edge-to-vertex ratios, our experiment covers a wide range of cases.

**Software Libraries.** Our implementation uses ParlayLib [12] for fork-join parallelism and some parallel primitives (e.g., sorting). We use the *P-tree* implementation from the PAM library [67, 69].

| | | $n$ | $m$ | Influence | Notes |
|---|---|---|---|---|---|
| Social | **EP** | 0.08M | 0.81M | 5332 | Epinions1 [61] |
| | **SLDT** | 0.08M | 0.94M | 6342 | Slashdot [48] |
| | **DBLP** | 0.32M | 2.10M | 1057 | DBLP [78] |
| | **YT** | 1.14M | 5.98M | 29614 | com-Youtube [78] |
| | **OK** | 3.07M | 234M | 1460000 | com-orkut [78] |
| | **LJ** | 4.85M | 85.7M | 376701 | soc-LiveJournal1 [6] |
| | **TW** | 41.7M | 2.40B | 11776629 | Twitter [45] |
| | **FT** | 65.6M | 3.61B | 19198744 | Friendster [78] |
| Web | **SD** | 89.2M | 3.88B | 15559737 | sd_arc [52] |
| | **CW** | 978M | 74.7B | - | ClueWeb [52] |
| Road | **GER** | 12.3M | 32.3M | 384 | Germany [1] |
| | **USA** | 23.9M | 57.7M | 370 | RoadUSA [1] |
| k-NN | **HT5** | 2.05M | 13.0M | 1018 | HT [28, 75], $k$=5 |
| | **HH5** | 2.05M | 13.0M | 2827 | Household [28, 75], $k$=5 |
| | **CH5** | 4.21M | 29.7M | 355065 | CHEM [30, 75], $k$=5 |
| | **GL5** | 24.9M | 157M | 11632 | GeoLife [75, 80], $k$=5 |
| | **COS5** | 321M | 1.96B | 4753 | Cosmo50 [46, 75], $k$=5 |

**Table 3: Graph Information.** $n$: number of vertices. $m$: number of edges. Influence: the maximum influence spread of 100 seeds selected by *PaC-IM* ($R = 256$), *InfuserMG* ($R = 256$), and *Ripples* ($\epsilon = 0.5$), rounded to integers. Most influence values are evaluated by 20000 simulations. The underline numbers on very large graphs are evaluated by 2000 simulations. "-": unable to evaluate within 50 hours using 2000 simulations.

When computing connectivity in sketch construction, we use the union-find implementation UniteRemCAS from ConnectIt [26].

**Tested Algorithms.** We tested both *P-tree* and *Win-Tree* for seed selection. In most cases, *Win-Tree* is more efficient in both time and space, so use *Win-Tree* as the default option in *PaC-IM*. We present more results comparing the two options in Sec. 5.3.

We compare to three existing parallel IM systems: *InfuserMG* [31], *No-Singles* [60] and *Ripples* [53, 54], and call them the ***baselines***. As introduced in Sec. 2, *InfuserMG* uses a similar sketch-based approach as *PaC-IM* but does not support compression or parallel priority queues. *No-Singles* and *Ripples* both use the idea of Reverse Influence Sampling (RIS) [17]. In our experiments, black We note that *InfuserMG* and our *PaC-IM* require undirected input graphs while *Ripples* make no such assumptions. We observe that *InfuserMG* and *Ripples* have *severe scalability issue* when the number of threads increase (see examples in Fig. 7). Hence, we report *their shortest time among all the tested numbers of threads*.

Each algorithm has a parameter that controls the solution quality: $R$ for *InfuserMG* and our algorithms and $\epsilon$ ($0 < \epsilon \leq 0.5$) for *Ripples*. The solution qualities increase with larger $R$ or smaller $\epsilon$. To ensure fairness, when comparing running time, we guarantee that *PaC-IM always gives a better solution* than the baselines. For *InfuserMG* and *PaC-IM*, we set the number of sketches $R = 256$. Our solution using $R = 256$ is on average more than 99% of the quality when using $R = 2^{15}$, which is consistent with the observation in the *StaticGreedy* paper [22] (see our full version for quality analysis). We independently verified this and shown it in Fig. 6 For *Ripples*, smaller $\epsilon$ means better accuracy but more computation. We tested *Ripples* under $\epsilon$ in range 0.13–0.5 as used in their paper. *PaC-IM* under $R = 256$ yields about the same solution quality as $\epsilon = 0.13$ (the best tested setting in their paper). When reporting time, we use $\epsilon = 0.5$ since this gives the fastest running time, and the quality

| | | Relative Influence | | | Total Running Time (second) | | | | Memory Usage (GB) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Ours | *InfuserMG* | *Ripples* | Ours$_1$ | Ours$_{0.1}$ | *InfuserMG* | *Ripples* | CSR | Ours$_1$ | Ours$_{0.1}$ | *InfuserMG* | *Ripples* |
| **Social** | **EP** | 100% | 99.9% | 98.9% | **0.29** | 0.49 | 0.38 | 4.14 | 0.01 | 0.14 | **0.06** | 0.17 | 0.22 |
| | **SLDT** | 100% | 99.8% | 99.1% | **0.30** | 0.53 | 0.48 | 26 | 0.01 | 0.15 | **0.05** | 0.17 | 0.29 |
| | **DBLP** | 100% | 99.0% | 98.3% | **0.35** | 0.37 | 0.86 | 41.9 | 0.02 | 0.48 | **0.10** | 0.67 | 0.29 |
| | **YT** | 100% | 99.9% | 98.2% | **1.22** | 2.44 | 6.20 | 71.1 | 0.05 | 1.63 | **0.28** | 2.36 | 1.54 |
| | **OK** | 100% | 100% | 99.8% | **8.79** | 39.6 | 80.3 | 329 | 1.77 | 6.13 | **2.44** | 8.07 | 78.5 |
| | **LJ** | 100% | 99.9% | 99.4% | **6.00** | 20.5 | 61.3 | 123 | 0.68 | 5.98 | **1.63** | 10.5 | 20.4 |
| | **TW** | 100% | 100% | - | **93.4** | 378 | 639 | 6258 | 18.2 | 63.0 | **25.9** | 103 | 729 |
| | **FT** | 100% | 100% | - | **128** | 609 | 1973 | - | 27.4 | 97.9 | **39.9** | 161 | - |
| **Web** | **SD** | 100% | 97.1% | - | **150** | 627 | 1684 | - | 29.6 | 125 | **44.8** | 211 | - |
| | **CW** | 100% | - | - | - | 9776 | - | - | 564 | - | 738 | - | - |
| **Road** | **USA** | 100% | 74.7% | 92.8% | 14.6 | **13.7** | 53.1 | 13628 | 0.61 | 25.8 | **5.05** | 49.0 | 46.4 |
| | **GER** | 100% | 70.4% | 94.4% | 9.35 | **8.53** | 26.7 | 2334 | 0.33 | 13.3 | **2.58** | 25.1 | 23.0 |
| ***k*-NN** | **HT5** | 100% | 85.7% | 94.8% | 0.72 | **0.68** | 2.73 | 47.5 | 0.11 | 1.37 | **0.24** | 1.93 | 0.83 |
| | **HH5** | 100% | 79.5% | 97.6% | 2.28 | **2.11** | 8.26 | 53.9 | 0.11 | 3.00 | **0.51** | 4.26 | 1.22 |
| | **CH5** | 100% | 92.3% | 97.8% | 3.52 | **5.32** | 124 | 39.1 | 0.25 | 4.87 | **1.05** | 8.77 | 1.77 |
| | **GL5** | 100% | 76.9% | 98.9% | 19.5 | **17.9** | 116 | 637 | 1.36 | 27.6 | **6.02** | 51.6 | 8.71 |
| | **COS5** | 100% | 37.7% | - | 348 | **284** | 2319 | - | 17.0 | 355 | **66.1** | 666 | - |

**Table 4: Running time, memory usage, and influence spread (normalized to the maximum) of all tested systems on a machine with 96 cores (192 hyperthreads).** black "-": out of memory (1.5 TB) or time limit (3 hours). Ours$_1$ is our implementation with *Win-Tree* without compression. Ours$_{0.1}$ is our implementation with $\alpha = 0.1$ (10× compression for sketches). *InfuserMG* [31] and *Ripples* [53, 54] are baselines. We report the *best time* of *InfuserMG* and *Ripples* by varied core counts (the scalability issue of *InfuserMG* and *Ripples* are shown in Fig. 7). CSR is the memory used to store the graph in CSR format (see more in Sec. 5.1). The bold numbers are the fastest time/smallest memory among all implementations on each graph. The underlined numbers in memory usage are the smallest memory among systems that do not use compression (Ours$_1$, *Ripples* and *InfuserMG*). A heatmap of the full result is in Fig. 1.

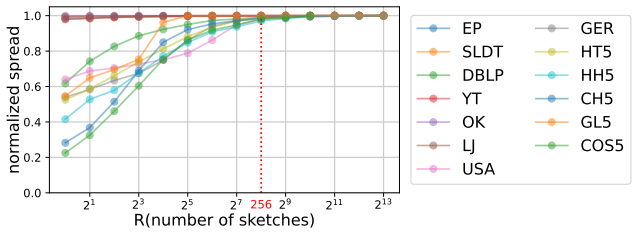


**Figure 6: Spread curves for our algorithm.** In each plot, $x$-axis is the number of sketches (R) and $y$-axis is the spread normalized to the maximum spread of each graph. Higher is better.

is still reasonably high (at least 93% of our best influence).

We observe that on sparse graphs, the influence spread of *InfuserMG* is only 38–92% of the best achieved by *PaC-IM* and *Ripples*. Although theoretically *PaC-IM* and *InfuserMG* should give the same output (assuming a fixed seed for the random number generator), we note that *InfuserMG* uses many optimizations that sacrifice solution quality. We tried to increase $R$ and various other attempts in *InfuserMG*, but they did not improve the solution quality. Therefore, we keep the same value $R = 256$ for both *PaC-IM* and *InfuserMG*.

## 5.1 Overall Time and Space

Tab. 4 shows the running time, memory usage, and normalized influence spread of all systems. Ours$_1$ and Ours$_{0.1}$ are our *PaC-IM* with $\alpha = 1$ (no compression) and $\alpha = 0.1$ (10× sketch compression), respectively. To better illustrate the tradeoff between time and space for all systems, we present a heatmap in Fig. 1, where all the numbers (time and space) are normalized to Ours$_1$. Ours$_{0.1}$ is the *only algorithm that can process the largest graph CW* [52]. With similar spread quality, Ours$_1$ is *faster than all baselines on all graphs*, and Ours$_{0.1}$ is just slower than *InfuserMG* on the two smallest graphs. Meanwhile, Ours$_{0.1}$ has the smallest space usage on *all* graphs. The advantage of *PaC-IM* is more significant on larger graphs, both in time and space.

**Running Time.** *PaC-IM* is significantly faster than the baseline algorithms on almost all graphs. As mentioned, we report the best running time among all tested core counts for *InfuserMG* and *Ripples*, since they may not scale to 192 threads (see Fig. 7 and Sec. 5.2). Even so, *PaC-IM* is still faster on all graphs. On average, Ours$_1$ is 5.7× faster than *InfuserMG* and 56× faster than *Ripples*. With compression, Ours$_{0.1}$ is slightly slower than Ours$_1$, but is still 3.2× faster than *InfuserMG* and 32× faster than *Ripples*.

In general, the compression in *PaC-IM* saves space by a tradeoff of increasing running time. When $\alpha = 1$, the CC sizes for all vertices are stored in the sketches, and a re-evaluation only needs a constant time to query on each sketch. When $\alpha = 0.1$, each query involves a search to either find a center, or visit all connected vertices, which roughly costs $O(1/\alpha)$ on each sketch. Indeed, on all scale-free graphs, Ours$_{0.1}$ takes a longer time than Ours$_1$. Interestingly, on most of the sparse graphs, we observe that Ours$_{0.1}$ is also faster than Ours$_1$. This is because seed selection only takes a small fraction of the total running time (except for CH5), so the slow-down in the seed selection step is negligible for the overall performance. Meanwhile, avoiding storing $O(Rn)$ connectivity sizes reduces memory footprint and makes the sketching step slightly faster, which overall speeds up the running time.

**Memory Usage.** We show space usage of *PaC-IM* and baselines in Fig. 1 and 8 and Tab. 4. We also show the size of representing the graph in standard Compressed Sparse Row (CSR) format as a reference in Tab. 4 and Fig. 8, which roughly indicates the space to store the input graph. CSR uses 8 bytes for each vertex and each edge. Using $\alpha = 0.1$, *PaC-IM* uses the least memory on all graphs. Even without compression, our algorithm uses less memory than the baselines on 10 out of 16 graphs. Note that although the

compression rate for sketches is 10× in $\text{Ours}_{0.1}$, the total space also includes the input graph and the data structure for seed selection. Therefore, we cannot directly achieve a 10× improvement in space. In most cases, the total memory usage is about 5× smaller.

**Summary.** Overall, *PaC-IM* has better performance than the baselines in both time and space. We note that the space usage of *Ripples* can be (up to 3×) better than $\text{Ours}_1$ on certain graphs (but still worse than $\text{Ours}_{0.1}$), but in these cases the running time is also much longer (by 10× or more). On scale-free graphs, $\text{Ours}_{0.1}$ is 2.5× slower than $\text{Ours}_1$ on average, but uses 3× less space. On sparse graphs, $\text{Ours}_{0.1}$ is almost always better in both time and space.

## 5.2 Scalability

We also study the scalability of all systems. We present the performance on six representative graphs in Fig. 7 with varying core counts $P$. We separate the time for sketch construction (the "sketch time") and seed selection (the "selection time") to study the two components independently. For *PaC-IM*, both sketch and selection time decrease with more cores and achieve almost linear speedup. black black

The scalability curves also *indicate the necessity of our new parallel data structures for seed selection* proposed in Sec. 4. black For both *InfuserMG* and *Ripples*, the sketch time parallelizes better than the selection time. This is because both algorithms use well-parallelized algorithms to construct sketches. For example, *InfuserMG* uses a standard coloring [57, 66] idea for parallel connectivity. However, for seed selection, both systems use sequential CELF, and only parallelize the re-evaluation on $R$ sketches. When the core count becomes comparable to $R$, such an approach may cause scheduling overhead and decrease the performance.

## 5.3 Analysis of the Proposed Techniques

Next, we evaluate the two proposed techniques in this paper: sketch compression and data structures for seed selection.

**Compression.** We evaluate the time and space usage of different compression ratios by controlling the parameter $\alpha$, and present the results in Fig. 8. The gray dashed line represents the CSR size of each graph, which is the space to store the input. Compression always reduces memory usage, but may affect running time differently. The memory usage always decreases with the value $\alpha$ decreases. As mentioned previously, the actual compression rate can be lower than $1/\alpha$, since our compression only controls the memory for storing sketches, and there are other space usage in the algorithm. Roughly speaking, using $\alpha = 0.05$ can save space on graphs by up to 8× and shrink the space very close to the input graph size.

Compression affects the running time differently for sketch construction and seed selection. Smaller $\alpha$ indicates longer running time in seed selection, but may improve the sketch time slightly due to the reduced memory footprint. The effect of compression on total running time depends on the ratio of sketching and selection time, where scale-free and sparse graphs exhibit different patterns. In the sampled graphs, scale-free graphs usually have one or several large CCs since they are dense, while the sparse graphs usually have many small CCs. This can also be seen by the overall influence in Tab. 3: even we use smaller $p$ on scale-free networks, the total influence is still much larger than the sparse graphs. On scale-free graphs, due to large CCs on the sampled graph, selecting any seed in this CC may significantly lower the score of other vertices, leading to much large total re-evaluations than sparse graphs. As a result, smaller $\alpha$ causes a clear time increase for seed selection on scale-free networks since each re-evaluation becomes slower, while the compression only has a small impact on most of the sparse graphs. We provide the number of re-evaluations on each graph in Tab. 5 as a reference.

| | | $n$ | CELF | P-Tree | Win-Tree |
|---|---|---|---|---|---|
| social | HP | 12008 | 7936 | 7964 | 8322 |
| | EP | 75879 | 57980 | 58010 | 58398 |
| | SLDT | 77360 | 66618 | 66649 | 67035 |
| | DBLP | 317080 | 3255 | 3269 | 4075 |
| | YT | 1134890 | 656014 | 656068 | 656727 |
| | OK | 3072627 | 3066983 | 3066995 | 3067514 |
| | LJ | 4847571 | 3609995 | 3610105 | 3611583 |
| | TW | 41652231 | 41200415 | 41200437 | 41200774 |
| | FT | 65608366 | 61946405 | 61946434 | 61947064 |
| web | SD | 89247739 | 84554009 | 84576755 | 84576236 |
| | CW | 978408098 | | 873230222 | 873256302 |
| road | USA | 23947348 | 143 | 159 | 465 |
| | GER | 12277375 | 139 | 155 | 416 |
| k-NN | HT5 | 928991 | 516 | 536 | 1007 |
| | HH5 | 2049280 | 3931 | 3943 | 4717 |
| | CH5 | 4208261 | 628918 | 629501 | 653157 |
| | GL5 | 24876978 | 11042 | 11430 | 12996 |
| | COS5 | 321065547 | 431 | 449 | 1264 |

**Table 5:** Numbers of re-evaluations for each graph instances using different algorithms.

***P-tree* vs. *Win-Tree*.** We now experimentally study both data structures for seed selection. Recall that our goal is to achieve high parallelism without introducing much overhead on work. In particular, *P-tree* has the theoretical guarantee that the total number of re-evaluations is no more than twice of the sequential CELF. To evaluate the work overhead is small, we plotted the number of re-evaluations for both data structures compared to CELF Fig. 9(a). For each graph, we count the number of re-evaluations by CELF as $x$, the number by *P-tree* as $y_1$, and the number by *Win-Tree* as $y_2$, and draw all $(x, y_1/x)$ as blue round points and all $(x, y_2/x)$ as orange triangle points. The number of re-evaluations by *P-tree*s is very close to that by CELF (1.03× on average), while *Win-Tree*s require slightly more (1.7× on average). The result is consistent with our theoretical analysis.

As mentioned in Sec. 4.2, we expect *Win-Tree*s to perform better in practice due to various reasons. To study this, we plotted Fig. 9(b) as a comparison for selection time between *P-tree* and *Win-Tree* under different compression ratios. Each data point represents a graph, and the value is the ratio of selection time between *P-tree* and *Win-Tree*. The average ratios are always larger than 1, indicating better performance for *Win-Tree*, but the advantage decreases as $\alpha$ becomes smaller (higher compression). This is because when $\alpha$ is large, the re-evaluation is fast, and the major time is on the tree operations, where *Win-Tree*s is more advantageous due to the reasons mentioned in Sec. 4.2. With higher compression (small $\alpha$), the re-evaluation becomes more expensive. Since the *P-tree* evaluates fewer vertices, the overall selection time is more likely to be better.

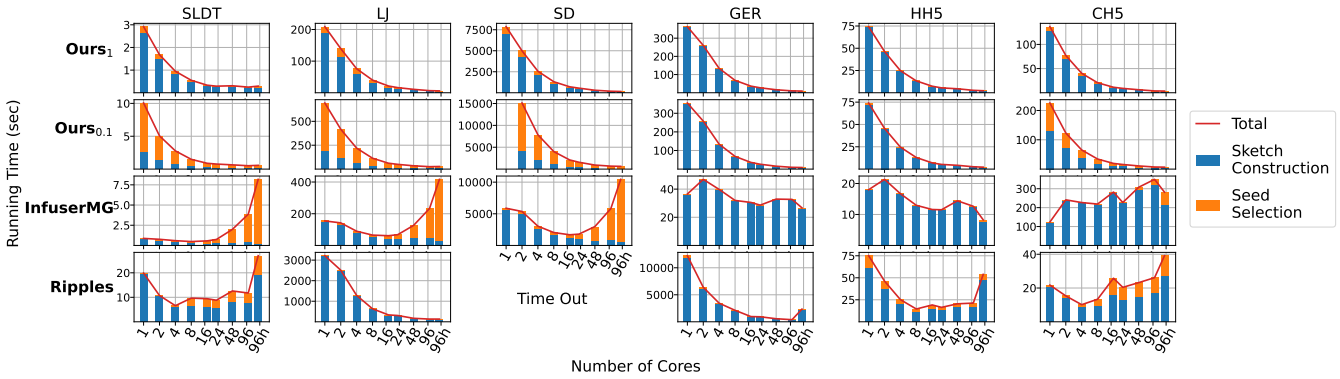


**Figure 7: Running time using different core counts for different IM algorithms.** In each plot, the $x$-axis shows core counts (96h means 96 cores with hyperthreading) and the $y$-axis is running time in seconds.

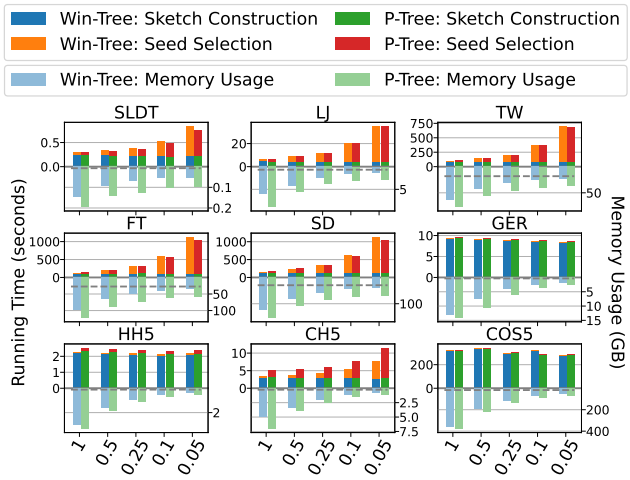


**Figure 8: Running time and memory with different values of $\alpha$.** The $x$-axis represents the compression rate. The growing up $y$-axis represents the running time (in seconds) of the sketching and selecting process. The growing down $y$-axis represents the total memory usage (in GB). The gray horizontal line represents the CSR size of each graph, which is the basic memory we need to load the graph.

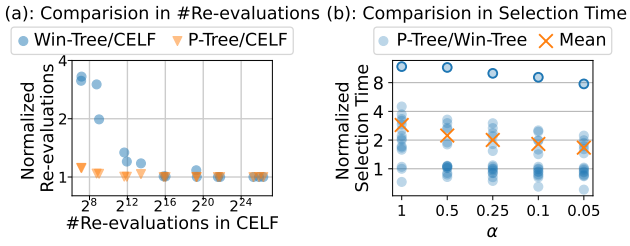


**Figure 9: Compare *P-tree* and *Win-Tree* in the number of re-evaluations and selection time. (a)**: Let $\#CELF$, $\#P\text{-}tree$ and $\#Win\text{-}Tree$ represent the total number of re-evaluations needed in the corresponding method. Each point represents a tested graph. Each blue circle is a data point $(\#CELF, \#Win\text{-}Tree/\#CELF)$. Each orange triangle is a data point $(\#CELF, \#P\text{-}tree/\#CELF)$. **(b)**: Each point represents a tested graph under different values of $\alpha$. The y-axis is *P-tree* selection time/*Win-Tree* selection time. The orange cross is the average value under each $\alpha$.

To further understand this, we focus on the five topmost points (circled) in Fig. 9(b): those data points are from COS5, where *P-tree* is 8–12× slower than *Win-Tree*. For COS5, the number of vertices $n$ is large, but only hundreds of vertices are re-evaluated in total. Thus, the seed selection time is dominated by constructing the data structures, i.e., $O(n \log n)$ work for *P-tree* to maintain total ordering, and $O(n)$ for *Win-Tree* that is a lot faster. Since the number of re-evaluations is small, most of the vertices in the tree are never touched, and thus maintaining their order wastes the work.

In Fig. 8, we also compare the time and space between *P-tree* and *Win-Tree*. Similar to the discussions above, when $\alpha$ is small, *Win-Tree* is almost always faster than *P-tree*, but *P-tree* may perform better when $\alpha$ is large. However, the advantage is quite small, since the number of re-evaluations of *Win-Tree* is still close to *P-tree* (see Fig. 9(a)). *Win-Tree* also uses smaller memory than *P-tree*. As discussed in Sec. 4.2, this is because *P-tree* needs to explicitly maintain tree pointers and balancing criteria, while each *Win-Tree* node only needs to store the vertex id ($2n$ integers in total).

In summary, *P-tree* almost always incurs fewer re-evaluations than *Win-Tree*, but is worse in running time in most cases, especially with low compression ($\alpha$ close to 1). When higher compression (smaller $\alpha$), *P-tree* can perform slightly better. *Win-Tree* is also more space-efficient than *P-tree*. Based on these observations, we always use *Win-Tree* as the default data structure in *PaC-IM*, but also provides the interface for users to choose *P-trees*.

## 6 Related Work

**Influence Maximization (IM).** IM has been widely studied for decades with a list of excellent surveys [5, 7, 49, 59, 82] that review the applications and papers on this topic. IM is also of high relevance of the data management community, and many excellent papers published recently regarding benchmarking [3, 55], new algorithms [38], new propagation models and applications [9, 39, 72, 77], and interdisciplinary extensions [79, 83].

According to [49], IM algorithms can roughly be categorized into three methodologies: simulation-based, proxy-based, and sketch-based. Among them, (Monte Carlo) simulation-based approaches (e.g., [33, 41, 47, 74, 81]) are the most general and apply to the most settings (i.e., graph types and diffusion models); however, they do not take advantage of the specific settings, so generally their performance is limited. Proxy-based approaches (e.g., [20, 40, 42, 43, 51, 58]) use simpler algorithms/problems (e.g., PageRank or shortest-paths) to solve IM. While the solutions can be fast in practice, their solution quality has no theoretical guarantees.

Sketch-based solutions, as mentioned in Sec. 1 and 2, generally have good performance and theoretical guarantees, but can be specific to diffusion models. As a sketch-based solution, *PaC-IM* mainly focuses on the IC model. We note that the parallel data structures in Sec. 4 are general to submodular diffusion models such as linear threshold [41], and more [72, 79].

Sketch-based algorithms can further be categorized into forward (influence) sketches and reverse (reachable) sketches [49]. As the names suggest, forward sketches record the influence that each vertex can propagate to in sampled graphs. Most algorithms [21–23, 31, 41, 56] mentioned in this paper, including *PaC-IM*, use forward sketches. Reverse sketches find a sample of vertices $T$ and keep the sets of vertices that can reach them. Many IM algorithms (e.g., [17, 70, 71, 73]), including *No-Singles* [60] and *Ripples* [53, 54] that we compared to, use reverse sketches. These algorithms can trade off (lower) solution quality for (better) performance/space, by adjusting the size of $T$. black

**Space-Efficient Connectivity.** We are aware of a few algorithms that can compute graph connectivity using $o(n)$ space [8, 18, 29, 44], which share a similar motivation with *PaC-IM*. However, *PaC-IM* does not require computing connecitivity in $o(n)$ space, but only requires storing it in $O(n)$ space. Hence, the goal here is essentially different, although these approaches are inspiring.

## 7 Conclusion and Discussions

In this paper, we address the scalability issues in existing IM systems by novel techniques including sketch compression and parallel CELF. Our sketch compression (Sec. 3) applies to the IC model and undirected graphs, and avoids the $O(Rn)$ space usage in existing systems, which allows *PaC-IM* to run on much larger graphs without sacrificing much performance. To the best of our knowledge, our new data structures, *P-tree* and *Win-Tree* (in Sec. 4), are the first parallel version of the CELF seed selection, which is general to any submodular diffusion models.

In addition to new algorithms, our techniques are carefully analyzed (Thm. 3.1 and 4.3) and have good theoretical guarantees regarding work, parallelism, and space. These analyses not only lead to good practical performance (see Fig. 1), but also help to understand how the techniques interplay. The techniques are also experimentally verified in Fig. 7 to 9. The theory and our careful implementation also lead to stable speedup with increasing core counts (see Fig. 7).

black

## References


[1] 2010. OpenStreetMap © OpenStreetMap contributors. https://www.openstreetmap.org/.

[2] 2023. https://github.com/ucrparlay/Influence-Maximization.

[3] Akhil Arora, Sainyam Galhotra, and Sayan Ranu. 2017. Debunking the myths of influence maximization: An in-depth benchmarking study. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 651–666.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (01 Apr 2001).

[5] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. 2010. Flash in a DBMS: Where and How? *IEEE Data Engineering Bulletin* 33, 4 (2010), 28–34. Special issue on data management using modern storage hardware.

[6] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.

[7] Suman Banerjee, Mamata Jenamani, and Dilip Kumar Pratihar. 2020. A survey on influence maximization in a social network. *Knowledge and Information Systems* 62, 9 (2020), 3417–3455.

[8] Naama Ben-David, Guy E. Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. Implicit Decomposition for Write-Efficient Connectivity Algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[9] Song Bian, Qintian Guo, Sibo Wang, and Jeffrey Xu Yu. 2020. Efficient algorithms for budgeted influence maximization on massive social networks. *Proceedings of the VLDB Endowment (PVLDB)* 13, 9 (2020), 1498–1510.

[10] Timo Bingmann, Thomas Keh, and Peter Sanders. 2015. A bulk-parallel priority queue in external memory with STXXL. In *International Symposium on Experimental Algorithms (SEA)*. Springer, 28–40.

[11] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Transactions on Parallel Computing (TOPC)* 9, 2 (2022), 1–41.

[12] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[13] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[14] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[15] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Parallelism in Randomized Incremental Algorithms. *J. ACM* 67, 5 (2020), 1–27.

[16] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[17] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing social influence in nearly optimal time. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 946–957.

[18] Andrei Z Broder, Anna R Karlin, Prabhakar Raghavan, and Eli Upfal. 1994. Trading space for time in undirected s-t connectivity. *SIAM J. Comput.* 23, 2 (1994), 324–334.

[19] Wei Chen, Wei Lu, and Ning Zhang. 2012. Time-critical influence maximization in social networks with time-delayed diffusion process. In *AAAI Conference on Artificial Intelligence*, Vol. 26. 591–598.

[20] Wei Chen, Chi Wang, and Yajun Wang. 2010. 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*. 1029–1038.

[21] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient influence maximization in social networks. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 199–208.

[22] Suqi Cheng, Huawei Shen, Junming Huang, Guoqing Zhang, and Xueqi Cheng. 2013. Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 509–518.

[23] Edith Cohen, Daniel Delling, Thomas Pajor, and Renato F Werneck. 2014. 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*. 629–638.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[25] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[26] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)* 14, 4 (2020), 653–667.

[27] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 184–197.

[28] Dheeru Dua and Casey Graf. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[29] Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. 1999. Tight lower bounds for st-connectivity on the NNJAG model. *SIAM J. Comput.* 28, 6 (1999), 2257–2284.

[30] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[31] Gökhan Göktürk and Kamer Kaya. 2020. Boosting parallel influence-maximization kernels for undirected networks with fusing and vectorization. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1001–1013.

[32] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing letters* 12 (2001), 211–223.

[33] Amit Goyal, Wei Lu, and Laks VS Lakshmanan. 2011. Celf++ optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th international conference companion on World wide web*. 47–48.

[34] Mark Granovetter. 1978. Threshold models of collective behavior. *American journal of sociology* 83, 6 (1978), 1420–1443.

[35] Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[36] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–272.

[37] Yan Gu, Omar Obeya, and Julian Shun. 2021. Parallel In-Place Algorithms: Theory and Practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 114–128.

[38] Qintian Guo, Sibo Wang, Zhewei Wei, and Ming Chen. 2020. Influence maximization revisited: Efficient reverse reachable set generation with bound tightened. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2167–2181.

[39] Shixun Huang, Wenqing Lin, Zhifeng Bao, and Jiachen Sun. 2022. Influence maximization in real-world closed social networks. *Proceedings of the VLDB Endowment (PVLDB)* 16, 2 (2022), 180–192.

[40] Kyomin Jung, Wooram Heo, and Wei Chen. 2012. Irie: Scalable and robust influence maximization in social networks. In *2012 IEEE 12th international conference on data mining*. IEEE, 918–923.

[41] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 137–146.

[42] Jinha Kim, Seung-Keol Kim, and Hwanjo Yu. 2013. Scalable and parallelizable processing of influence maximization for large-scale social networks?. In *2013 IEEE 29th international conference on data engineering (ICDE)*. IEEE, 266–277.

[43] Masahiro Kimura and Kazumi Saito. 2006. Tractable models for information diffusion in social networks. In *Knowledge Discovery in Databases: PKDD 2006: 10th European Conference on Principles and Practice of Knowledge Discovery in Databases Berlin, Germany, September 18-22, 2006 Proceedings 10*. Springer, 259–271.

[44] Adrian Kosowski. 2013. Faster walks in graphs: a $\tilde{O}(n^2)$ time-space trade-off for undirected st connectivity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1873–1883.

[45] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.

[46] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*. Springer, 132–150.

[47] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 420–429.

[48] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[49] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. 2018. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 30, 10 (2018), 1852–1872.

[50] Bo Liu, Gao Cong, Dong Xu, and Yifeng Zeng. 2012. Time constrained influence maximization in social networks. In *International Symposium on Algorithms and Computation (ISAAC)*. IEEE, 439–448.

[51] Qi Liu, Biao Xiang, Enhong Chen, Hui Xiong, Fangshuang Tang, and Jeffrey Xu

Yu. 2014. Influence maximization over large-scale social networks: A bounded linear approach. In *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*. 171–180.
[52] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons — Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.
[53] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. cuRipples: Influence maximization on multi-GPU systems. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 1–11.
[54] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. 2019. Fast and scalable implementations of influence maximization algorithms. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.
[55] Naoto Ohsaka. 2020. The solution distribution of influence maximization: A high-level experimental study on three algorithmic approaches. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2151–2166.
[56] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-ichi Kawarabayashi. 2014. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI Conference on Artificial Intelligence*, Vol. 28.
[57] Simona Mihaela Orzan. 2004. On distributed verification and verified distribution. (2004).
[58] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.
[59] Sancheng Peng, Yongmei Zhou, Lihong Cao, Shui Yu, Jianwei Niu, and Weijia Jia. 2018. Influence analysis in social networks: A survey. *Journal of Network and Computer Applications* 106 (2018), 17–32.
[60] Diana Popova, Naoto Ohsaka, Ken-ichi Kawarabayashi, and Alex Thomo. 2018. Nosingles: a space-efficient algorithm for influence maximization. In *International Conference on Scientific and Statistical Database Management*. 1–12.
[61] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. 2003. Trust management for the semantic web. In *The Semantic Web-ISWC 2003: Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003. Proceedings 2*. Springer, 351–368.
[62] Manuel Gomez Rodriguez, David Balduzzi, and Bernhard Schölkopf. 2011. Uncovering the temporal dynamics of diffusion networks. (2011), 561–568.
[63] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. [n.d.]. *Sequential and Parallel Algorithms and Data Structures.* Springer.
[64] Thomas C Schelling. 2006. *Micromotives and macrobehavior.* WW Norton & Company.
[65] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[66] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 550–559.
[67] Yihan Sun and Guy Blelloch. 2019. Implementing Parallel and Concurrent Tree Structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 447–450.
[68] Yihan Sun and Guy E Blelloch. 2019. Parallel Range, Segment and Rectangle Queries with Augmented Maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*. 159–173.
[69] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.
[70] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence maximization in near-linear time: A martingale approach. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1539–1554.
[71] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 75–86.
[72] Dimitris Tsaras, George Trimponias, Lefteris Ntaflos, and Dimitris Papadias. 2021. Collective influence maximization for multiple competing products with an awareness-to-influence model. *Proceedings of the VLDB Endowment (PVLDB)* 14, 7 (2021), 1124–1136.
[73] Xiaoyang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Chen Chen. 2016. Bring order into the samples: A novel scalable method for influence maximization. *IEEE Transactions on Knowledge and Data Engineering* 29, 2 (2016), 243–256.
[74] Yu Wang, Gao Cong, Guojie Song, and Kunqing Xie. 2010. Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1039–1048.
[75] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.
[76] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem. In *ACM Symposium on Computational Geometry (SoCG)*.
[77] Jiadong Xie. 2022. Hindering Influence Diffusion of Community. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2518–2520.
[78] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.
[79] Wentao Zhang, Zhi Yang, Yexin Wang, Yu Shen, Yang Li, Liang Wang, and Bin Cui. 2021. GRAIN: improving data efficiency of gra ph neural networks via diversified in fluence maximization. *Proceedings of the VLDB Endowment (PVLDB)* 14, 11 (2021), 2473–2482.
[80] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.
[81] Chuan Zhou, Peng Zhang, Wenyu Zang, and Li Guo. 2015. On the upper bounds of spread for greedy algorithms in social network influence maximization. *IEEE Transactions on Knowledge and Data Engineering* 27, 10 (2015), 2770–2783.
[82] Fan Zhou, Xovee Xu, Goce Trajcevski, and Kunpeng Zhang. 2021. A survey of information cascade analysis: Models, predictions, and recent advances. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.
[83] Yuqing Zhu, Jing Tang, Xueyan Tang, and Lei Chen. 2021. Analysis of influence contribution in social advertising. *Proceedings of the VLDB Endowment (PVLDB)* 15, 2 (2021), 348–360.

# A Performance Evaluation under Different Edge Probability Distributions

To study how different edge probability distributions affect our algorithm and baseline algorithms, we also tested two more edge probability distributions. The first one uses $U(0, 0.1)$ for each edge in social and web graphs, and $U(0.1, 0.3)$ for road and $k$-NN graphs, where $U(x, y)$ means to draw a uniform random number from $x$ to $y$. This probability assignment is also commonly used in previous work [31, 54]. The other probability distribution is taking $p_{uv} = \frac{2}{d_u+d_v}$, where $p_{uv}$ is the probability to sample edge $(u, v)$, $d_u$ and $d_v$ are degrees of vertex $u$ and $v$. It is similar to the Weighted IC (WIC) model on the directed graph, where $p_{uv} = \frac{1}{d_{in}(v)}$, where $p_{u,v}$ is the probability the directed edge $u \to v$ is sampled, $d_{in}(v)$ is the in-degree of vertex $v$. We will use *Uniform* and *WIC* to refer to these two edge probability assignments respectively, and use *Consistent* to refer to the assignment mentioned in the main context of the paper.

Tab. 6 and Tab. 7 show the influence scores, running time and memory usage of all systems with Uniform and WIC edge probability assignments respectively. Ours$_1$ and Ours$_{0.1}$ are our *PaC-IM* with $\alpha = 1$ (no compression) and $\alpha = 0.1$ (10× sketch compression), respectively. As mentioned in the main context, *InfuserMG* and *Ripples* has scalability issue. We tested them with both 192 hyper-threads and the same core counts to get their best performance in Tab. 4 for each graph. In Tab. 6 and Tab. 7, we report the smaller one as the running time.

**Influence Score.** With $R = 256$ for *PaC-IM* and *InfuserMG*, and $\epsilon = 0.5$ for *Ripples*, *PaC-IM* has the largest influence score on all 16 tested graphs with *Uniform* edge probability assignment, and is smaller than *Ripples* by 3.5% and 1.0% on graph OK and LJ with *WIC* edge probability assignment. The influence scores of *Ripples* and Ours differ less than 6%, which indicates $R = 256$ and $\epsilon = 0.5$ is a fair setting for baselines to compare their running time and memory. We observe that the influence score of *InfuserMG* is only $41 - 89\%$ of the best-achieved score by *Ripples* and Ours on sparse graphs with *Uniform* edge assignment and 51% of the best influence on TW with *WIC* edge assignment. The observation is the same with the *Consistent* assignment in the main context of the paper.

| | | Relative Influence | | | Total Running Time (second) | | | | Memory Usage (GB) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Ours | *InfuserMG* | *Ripples* | $\text{Ours}_1$ | $\text{Ours}_{0.1}$ | *InfuserMG* | *Ripples* | CSR | $\text{Ours}_1$ | $\text{Ours}_{0.1}$ | *InfuserMG* | *Ripples* |
| **Social & Web** | **EP** | **11.3K** | 11.3K | 11.2K | 0.36 | 0.49 | **0.34** | 4.30 | 0.01 | 0.14 | **0.05** | 0.16 | 0.46 |
| | **SLDT** | **15.4K** | 15.3K | 15.3K | **0.35** | 0.52 | 0.39 | 29.5 | 0.01 | 0.13 | **0.05** | 0.17 | 0.60 |
| | **DBLP** | **10.4K** | 10.3K | 10.2K | **0.52** | 0.79 | 2.15 | 38.6 | 0.02 | 0.47 | **0.08** | 0.67 | 0.54 |
| | **YT** | **86.6K** | 86.6K | 86.1K | **1.60** | 2.95 | 5.21 | 62.3 | 0.05 | 1.68 | **0.27** | 2.37 | 4.36 |
| | **OK** | **2.32M** | 2.32M | 2.32M | **9.93** | 29.0 | 52.7 | 290 | 1.77 | 6.09 | **2.43** | 8.09 | 129 |
| | **LJ** | **1.19M** | 1.19M | 1.19M | **7.30** | 22.3 | 54.2 | 179 | 0.68 | 6.08 | **1.62** | 10.5 | 77.5 |
| | **TW** | **20.1M** | 20.1M | - | **122** | 327 | 571 | - | 18.2 | 63.2 | **26.5** | 103 | - |
| | **FT** | **29.4M** | 29.4M | - | **150** | 492 | 1213 | - | 27.4 | 97.6 | **39.4** | 161 | - |
| | **SD** | **29.2M** | 29.0M | - | **187** | 570 | 1523 | - | 29.6 | 125 | **44.5** | 211 | - |
| **Road** | **USA** | **0.39K** | 0.30K | 0.36K | **14.4** | 13.9 | 64.3 | 12752 | 0.33 | 25.8 | **5.06** | 49.1 | 45.1 |
| | **GER** | **0.40K** | 0.29K | 0.38K | 9.26 | **8.44** | 27.6 | 2221 | 0.61 | 13.3 | **2.59** | 25.2 | 22.1 |
| **k-NN** | **HT5** | **1.07K** | 0.89K | 1.01K | 0.74 | **0.71** | 2.71 | 40.0 | 0.11 | 1.37 | **0.23** | 1.94 | 0.94 |
| | **HH5** | **2.93K** | 2.31K | 2.84K | 2.28 | **2.10** | 9.92 | 51.3 | 0.11 | 3.04 | **0.51** | 4.27 | 1.29 |
| | **CH5** | **360K** | 320K | 354K | **3.61** | 5.58 | 118.24 | 38.1 | 0.25 | 4.88 | **1.04** | 8.80 | 1.86 |
| | **GL5** | **11.6K** | 8.5K | 11.6K | 19.3 | **17.7** | 94.5 | 631 | 1.36 | 27.6 | **6.02** | 51.7 | 8.84 |
| | **COS5** | **5.0K** | 2.1K | - | **310** | 322 | 2024 | - | 17 | 355 | **66** | 666 | - |

**Table 6: Under uniform edge distribution: running time, memory usage, and the influence spread of all tested systems on a machine with 96 cores (192 hyperthreads).** For social and web graphs, the edges are sampled under the uniform distribution $U_{[0,0.1]}$; for road and k-NN graphs, the edges are sampled under $U_{[0.1,0.3]}$. "-": out of memory (1.5 TB) or time limit (3 hours). $\text{Ours}_1$ is our implementation with *Win-Tree* without compression. $\text{Ours}_{0.1}$ is our implementation with $\alpha = 0.1$ (10× compression for sketches). *InfuserMG* [31] and *Ripples* [53, 54] are baselines. We report the *best time* of *InfuserMG* and *Ripples* by varied core counts (the scalability issue of *InfuserMG* and *Ripples* are shown in Fig. 7). CSR is the memory used to store the graph in CSR format (see more in Sec. 5.1). The bold numbers are the highest influence spread/fastest time/smallest memory among all implementations on each graph. The underlined numbers in memory usage are the smallest memory among systems that do not use compression ($\text{Ours}_1$, *Ripples* and *InfuserMG*).

| | | Relative Influence | | | Total Running Time (second) | | | | Memory Usage (GB) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Ours | *InfuserMG* | *Ripples* | $\text{Ours}_1$ | $\text{Ours}_{0.1}$ | *InfuserMG* | *Ripples* | CSR | $\text{Ours}_1$ | $\text{Ours}_{0.1}$ | *InfuserMG* | *Ripples* |
| **Social & Web** | **EP** | **615** | 605 | 594 | 0.30 | 0.41 | **0.28** | 1.59 | 0.01 | 0.13 | **0.04** | 0.16 | 0.10 |
| | **SLDT** | **645** | 628 | 618 | 0.29 | 0.35 | **0.28** | 28.0 | 0.01 | 0.13 | **0.04** | 0.17 | 0.14 |
| | **DBLP** | **738** | 636 | 718 | **0.37** | 0.42 | 1.04 | 44.4 | 0.02 | 0.47 | **0.08** | 0.67 | 0.38 |
| | **YT** | **678** | 652 | 654 | **1.00** | 1.08 | 3.34 | 58.6 | 0.05 | 1.68 | **0.27** | 2.37 | 1.44 |
| | **OK** | 1747 | 1622 | **1810** | 6.81 | **6.77** | 80.3 | 65.5 | 1.77 | 6.21 | **2.43** | 8.10 | 5.62 |
| | **LJ** | 1199 | 1054 | **1211** | **5.23** | 5.23 | 32.9 | 75.6 | 0.68 | 6.04 | **1.63** | 10.5 | 4.98 |
| | **TW** | **990** | 509 | - | **73.2** | 75.6 | 416 | - | 18.2 | 62.8 | **25.6** | 103 | - |
| | **FT** | **1425** | 1324 | - | 95.7 | **93.2** | 1352 | - | 27.4 | 97.6 | **39.2** | 161 | - |
| | **SD** | **3762** | 3675 | - | **117** | 121 | 1924 | - | 29.6 | 125 | **44.4** | 211 | - |
| **Road** | **USA** | **422** | 405 | - | 13.9 | **12.9** | 86.0 | 1770 | 0.33 | 25.80 | **4.99** | 49.1 | 21.1 |
| | **GER** | **430** | 385 | - | 10.9 | **9.87** | 35.3 | 10137 | 0.61 | 13.26 | **2.62** | 25.2 | 43.3 |
| **k-NN** | **HT5** | **440** | 397 | 405 | 0.73 | **0.68** | 2.50 | 57.1 | 0.11 | 1.35 | **0.23** | 1.94 | 1.49 |
| | **HH5** | **518** | 453 | 493 | 1.97 | **1.83** | 7.55 | 69.8 | 0.11 | 3.05 | **0.49** | 4.27 | 3.13 |
| | **CH5** | **755** | 596 | 733 | 3.03 | **2.95** | 16.4 | 88.1 | 0.25 | 4.83 | **1.05** | 8.80 | 4.87 |
| | **GL5** | **489** | 406 | 468 | 18.2 | **17.0** | 74.1 | 10262 | 1.36 | 27.6 | **6.05** | 51.7 | 49.4 |
| | **COS5** | **519** | 453 | - | 294 | **267** | 1729 | - | 17 | 355 | **66.0** | 666 | - |

**Table 7: Under vertex-related edge distribution: running time, memory usage, and the influence spread of all tested systems on a machine with 96 cores (192 hyperthreads).** For an undirected-edge $(u, v)$, the probability it is sampled is $\frac{2}{d_u+d_v}$, where $d_u$ and $d_v$ are the degrees of vertex $u$ and $v$. "-": out of memory (1.5 TB) or time limit (3 hours). $\text{Ours}_1$ is our implementation with *Win-Tree* without compression. $\text{Ours}_{0.1}$ is our implementation with $\alpha = 0.1$ (10× compression for sketches). *InfuserMG* [31] and *Ripples* [53, 54] are baselines. We report the *best time* of *InfuserMG* and *Ripples* by varied core counts (the scalability issue of *InfuserMG* and *Ripples* are shown in Fig. 7). CSR is the memory used to store the graph in CSR format (see more in Sec. 5.1). The bold numbers are the highest influence spread/fastest time/smallest memory among all implementations on each graph. The underlined numbers in memory usage are the smallest memory among systems that do not use compression ($\text{Ours}_1$, *Ripples* and *InfuserMG*).

**Running Time.** *PaC-IM* is significantly faster than the baseline algorithms on almost all graphs. As mentioned, we report the running time with the best core counts setting for *InfuserMG* and *Ripples*, because they may not scale to 192 threads. With both *Uniform* and *WIC* assignment, $\text{Ours}_1$ is faster than all baselines on 15 graphs. $\text{Ours}_1$ is just slower than *InfuserMG* on the smallest graph by less than 0.02 seconds and $\text{Ours}_{0.1}$ is just slower than *InfuserMG* on the two smallest graphs. On average, with *Uniform* assignment, $\text{Ours}_1$ is 4.6× faster than *InfuserMG* and 17× faster than *Ripples*, and $\text{Ours}_{0.1}$ is 2.9× faster than *InfuserMG* and 11× faster than *Ripples*. With *WIC* assignment, $\text{Ours}_1$ is 4.5× faster than *InfuserMG* and 22× faster than *Ripples*, and $\text{Ours}_{0.1}$ is 4.4× faster than *InfuserMG*

and 22× faster than *Ripples*.

**Memory Usage.** Note that *PaC-IM* and *InfuserMG* have almost exactly the same memory usage under different edge probability assignments. Recall that *PaC-IM* and *InfuserMG* are forward-reachable sketch-based IM algorithms, their memory usage for the same graph only depends on the number of sketches $R$. Therefore, with the same $R = 256$, they will have almost the same memory usage. Different from *PaC-IM* and *InfuserMG*, *Ripples* is a reverse-reachable sketch-based IM algorithm, which dynamically samples sketches according to the $\epsilon$ and graphs. *Ripples* memory usage varies on different edge probability assignments and graphs. One observation is that on social graphs YT, OK and LJ, the memory usage of *Ripples* with WIC is much smaller than that with *Uniform* and *Consistent* edge probability assignments (10× fewer than *Uniform* and 30× fewer than *Consistent* assignment on average on these three graphs). With $\alpha = 0.1$, *PaC-IM* uses the least memory on all graphs with all assignments.

**Summary.** Overall, *PaC-IM* has better performance than the baselines in both time and space on all tested edge probability assignments. The performance comparison between *PaC-IM* and the baseline algorithms are similar on *Consistent* and *Uniform* assignments. *Ripples* exhibits slightly different performance on *WIC* to *Consistent*, because it samples reverse-reachable sketches dynamically. In summary, the relative performance is fairly consistent on these three edge probability distributions. Therefore, in the main body of the paper, we simply use the *Consistent* assignment (fixing $p$ for all edges) to demonstrate the performance.