Efficient GPU-Accelerated Adaptive Minimum Cost Seed Selection

Gongyao Guo
gong-yao.guo@connect.polyu.hk
The Hong Kong Polytechnic University

Chen Feng*
chenfeng@polyu.edu.hk
The Hong Kong Polytechnic University

Yiran Li
one.li@utoronto.ca
University of Toronto

Jieming Shi
jieming.shi@polyu.edu.hk
The Hong Kong Polytechnic University

ABSTRACT

Efficient influence estimation and seed selection are crucial to social network advertising and are widely studied in data management. We focus on adaptive minimum cost seed selection (AMCSS), which selects *seed nodes* adaptively over multiple rounds, to reach a target number η of influenced users while minimizing total seed cost. Prior work, notably ASTI with multi-root reverse reachable sets (mRR-sets), remains computationally expensive on large graphs, often taking hours to days even with CPU parallelism.

We present GAAS, a GPU-accelerated method that integrates novel algorithmic and GPU-aware system designs to solve AMCSS in minutes. Processing mRR-sets involves irregular access patterns and variable-size samples, mismatching the GPU parallel architecture. Hence, we first develop a *GPU-tailored mRR-set structure*, GmRR, that assigns each thread block exclusive ownership of an equal-size *segment* with a *circular* layout, enabling efficient parallel mRR-set management while minimizing write contention. With GmRR, we design a GPU kernel ParallelGen to generate mRR-sets. Unlike prior work that regenerates mRR-sets from scratch in each round, we propose to update and reuse those from previous round, improving efficiency while requiring GPU-aware designs and rigorous theoretical analysis. Specifically, we design a ParallelUpdate kernel with *theoretically grounded update rules* that uses *circular segment updates* on GmRR for efficient mRR-set updates, together with a load-balancing scheme. We further devise a Select kernel for parallel seed selection. Integrating these together, GAAS efficiently solves AMCSS on GPUs with guarantees. Extensive experiments on large real-world graphs under different diffusion models show that GAAS is over an order of magnitude faster (up to 68.9×) than parallel CPU and GPU baselines, while the seed cost is among the lowest.

PVLDB Reference Format:
Gongyao Guo, Chen Feng, Yiran Li, and Jieming Shi. Efficient GPU-Accelerated Adaptive Minimum Cost Seed Selection. PVLDB, 19(11): 3427 - 3439, 2026.
doi:10.14778/3836663.3836699

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gongyguo/GAAS.

*Corresponding Author.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836699

1 INTRODUCTION

Social networks are a dominant medium for information dissemination, where users share content and influence with each other. As of October 2025, over 5.6 billion people use social networks [10], driving a social advertising market exceeding $136 billion [3].

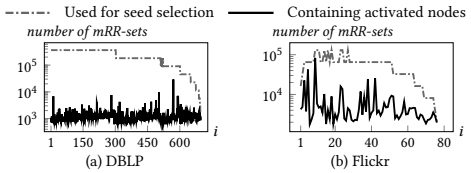
Advertisers select a set of influential users as *seeds*, each with a *cost*, to initiate diffusion process to influence a large number of users in the social network, to achieve a large *influence spread*. Identifying influential seeds requires accurate influence estimation, which is computationally expensive. This has motivated extensive work on scalable influence estimation and seed selection in data management [16, 17, 22–24, 26, 33, 34, 41, 42, 45, 47].

A typical formulation specifies a target number of influenced users η (*influence threshold*) and seeks a seed set of minimum cost whose influence spread reaches η, known as the minimum cost seed selection (MCSS) problem [12, 18, 20]. MCSS is non-adaptive: seeds are selected in a single batch without observing actual diffusion, so the realized spread can deviate substantially from η [12, 18].

Therefore, *adaptive* MCSS (AMCSS) has been proposed to select seeds iteratively over multiple rounds, where each round's choices depend on observed diffusion outcomes [13, 14, 39, 44], *i.e.*, the nodes activated in previous rounds. AMCSS is NP-hard, motivating efficient approximate algorithms with theoretical guarantees. ASTI [39] is state of the art that introduces multi-root reverse reachable sets (mRR-sets) to estimate user influence via stochastic breadth-first-search (BFS) from randomly selected root nodes. However, solving AMCSS on large graphs remains computationally prohibitive. ASTI often takes tens of thousands of seconds—more than one *day* for large η on large graphs—while a parallel CPU version of ASTI still needs around half a day. Such latency limits the practicality of AMCSS in time-critical applications.

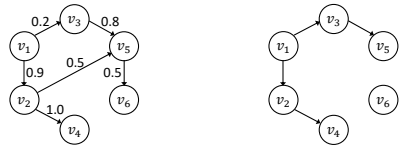
We aim to improve the processing time to *minutes* (hundreds of seconds) while preserving result quality. This is challenging and existing methods largely ignore system design aspects. We achieve this by proposing a GPU-Accelerated AMCSS method, GAAS, which integrates novel algorithmic designs with GPU-aware optimizations to exploit the massive parallelism of modern GPUs.

GPUs are increasingly important for general-purpose parallel computing [21, 36, 38, 48–50]. Compared with CPUs, GPUs have a fundamentally different architecture. They execute massive numbers of lightweight threads in a SIMT (Single Instruction Multiple



(a) DBLP (b) Flickr

Figure 1: mRR-sets Statistics.



(a) Social network G (b) Realization ϕ

Figure 2: A social graph and a possible realization.

Thread) model, which suits data-parallel tasks with simple control flow and regular memory access but performs poorly under complex branching and irregular access patterns. The memory hierarchy also differs. Therefore, directly porting CPU-native algorithms to GPUs often results in suboptimal performance.

Specifically, the main cost in AMCSS is sampling and processing mRR-sets in each adaptive round for influence estimation. This involves irregular graph traversals and variable-size samples: the size of each mRR-set is unknown a priori and the required number of mRR-sets also changes by round, complicating GPU data structures and parallelization. Moreover, existing methods regenerate all mRR-sets from scratch in every round, wasting computation despite the close relationship of mRR-sets between successive rounds.

Figure 1 shows the number of mRR-sets (log-scale) used in round i (dot-dash line) and the number of mRR-sets polluted by the activated nodes when round i finishes (solid line). The gap between the two lines indicates the mRR-sets that contain no activated nodes, i.e., not polluted. The gap is large in most rounds, e.g., in Figure 1(a) for DBLP. This suggests that many mRR-sets in round i have potential to be reused in round $i+1$ after proper updating. It motivates GAAS to update and reuse mRR-sets. This is efficient but requires careful GPU-aware designs and theoretical analysis to ensure the updated mRR-sets are equivalent to newly generated ones.

To this end, we first design a *GPU-tailored mRR-set structure*, GmRR, that supports efficient storage, parallel generation, and update/reuse of mRR-sets across rounds in GAAS. GmRR partitions a pre-allocated global array into disjoint equal-size *segments*, each exclusively owned by one GPU thread block, minimizing write contention in parallel. Within each segment, a *circular* layout allows variable-length mRR-sets to be appended and updated with little data movement. All components in GmRR are flat arrays with regular access patterns, enabling efficient GPU execution. With GmRR, we design GPU kernels to parallelize mRR-set generation, update, and seed selection in GAAS. The `ParallelGen` kernel generates mRR-sets by evenly partitioning them across thread blocks and performing warp-centric parallel BFS within each block. We then develop the `ParallelUpdate` kernel to update mRR-sets from the previous round. We first derive *theoretically sound update rules* for reusing mRR-sets under changes in the graph and influence threshold, and then parallelize the update by letting each block update the mRR-sets in its own segment using a *circular segment update* on GmRR that enables in-segment updates and appends without excessive data movement, together with a load-balancing technique. With the mRR-sets ready, we devise a GPU kernel `Select` that selects each seed batch in parallel and updates data structures for next round. Integrating these components, GAAS solves AMCSS with an approximation ratio of $\ln \eta / [(1-1/e)(1-\epsilon)]$, improving over existing study, and achieves high GPU efficiency.

Extensive experiments on real-world data under different diffusion models show that GAAS is typically over an order of magnitude faster than the baselines, while yielding seeds with comparable or lower total cost. Our contributions are summarized as follows.

- We design a GPU-accelerated method GAAS for solving AMCSS that significantly improves efficiency by integrating novel algorithmic designs with GPU-specific optimizations.
- We develop a GPU-tailored mRR-set structure GmRR that supports efficient storage, generation, and update of mRR-sets on GPUs, while reducing write contention and improving parallelism.
- We propose theoretically grounded mRR-set update rules across rounds, and GPU kernels with circular segment update on GmRR for efficient mRR-set update, generation, and seed selection.
- Extensive experiments show GAAS is often over an order of magnitude faster than baselines while maintaining low seed cost.

2 PRELIMINARIES

2.1 Influence Diffusion Models

The social network is a directed graph $G = (V, E)$ with $|V| = n$ users and $|E| = m$ directed edges. Each edge $e = (u, v) \in E$ represents a social tie from u to v along which influence may diffuse. The in- and out-neighbor sets of u are denoted by $N_{\text{in}}(u)$ and $N_{\text{out}}(u)$. Each edge e has a *diffusion probability* $p(e) = p(u, v) \in [0, 1]$, and each node $u \in V$ has a positive *seed cost* $c(u)$, normalized to $(0, 1]$ by dividing all costs by the maximum. Figure 2(a) shows a graph G, where the number on each edge indicates its diffusion probability.

On graph G, given a *seed set* $S \subseteq V$, influence diffusion follows the widely adopted models, *independent cascade* (IC) or *linear threshold* (LT) [4, 39–41], in discrete steps. At step 0, only nodes in S are active, while all other nodes are inactive. When a node u becomes active at step t, it has one chance in step $t+1$ to activate each inactive out-neighbor; afterwards, u makes no further attempts. The two models differ in how u activates an out-neighbor v. In the IC model, u activates v independently with probability $p(u, v)$. In the LT model, each node v has a threshold $\lambda_v \sim U(0, 1)$, and $p(u, v)$ is treated as a weight; v becomes active if the total weight from its active in-neighbors is no less than λ_v, i.e., $\sum_{u \in N_{\text{in}}(v)} \mathbb{I}_{\{u \text{ is active}\}} p(u, v) \geq \lambda_v$, where $\mathbb{I}_{\{\cdot\}}$ is the indicator function. The diffusion process stops when no new node is activated, and the total number of activated nodes, denoted by $I(S)$, is the *influence spread* of S.

Note that $I(S)$ is a random variable with multiple possible outcomes due to the randomness in the diffusion process. The IC propagation process can be equivalently described by the live-edge procedure [25]. Specifically, for each edge $e = (u, v) \in E$, we independently flip a coin with head probability $p(e)$ to decide whether e is *live* or *blocked*. All blocked edges are removed and the remaining graph is referred to as a *realization* of the graph G, denoted by ϕ. Under a realization ϕ, the influence spread of a seed set S, denoted as $I_\phi(S)$, is the number of nodes reachable from S via live edges. For example, Figure 2(b) shows a realization ϕ with live edges marked while blocked edges removed in the IC model. Diffusing from seed

v_2 over ϕ yields $I(\{v_2\}) = 2$, since v_2 reaches v_4 and itself. In the LT model, a realization ϕ is obtained by sampling a threshold λ_v for each node. The influence spread of S under ϕ is then the number of nodes activated from S under the sampled thresholds.

Let Ω be the set of all possible realizations and $p(\phi)$ denote the probability of a realization ϕ. For ϕ, the influence spread $I_\phi(S)$ is deterministic. Thus, the expected spread is $\mathbb{E}_\Phi[I(S)] = \sum_{\phi\in\Omega} I_\phi(S) \cdot p_\phi$, which is the weighted average spread over all realizations in Ω.

2.2 Adaptive Minimum Cost Seed Selection

Non-adaptive MCSS identifies a seed set S with minimum cost such that $\mathbb{E}_\Phi[I(S)] \geq \eta$, without knowing the actual realization in advance [12, 18, 51]. As a result, the realized number of activated users may be far fewer than η in some realizations, or more than η in others. In contrast, the adaptive setting AMCSS selects seeds over multiple rounds, with each round's choices based on the observed diffusion outcomes so far. This feedback enables more cost-effective decisions and keeps the realized spread close to the target η, making AMCSS more practical in real deployments. Specifically, in round i, we select a batch of b seeds S_i on G_i, trigger diffusion from S_i, and observe the activated nodes A_i (including S_i). We then update the remaining influence threshold to $\eta_{i+1} = \eta_i - |A_i|$ and remove A_i and their incident edges to obtain the residual graph $G_{i+1} = (V_{i+1}, E_{i+1})$. Initially, $G_1 = G, V_1 = V, E_1 = E$, and $\eta_1 = \eta$. The process stops once at least η users are activated, and the final seed set is $S = \bigcup_i S_i$.

An adaptive policy π specifies how seeds are selected in each round. For a realization ϕ, let $S(\pi, \phi)$ be the seeds chosen by π, and let $C(\pi, \phi) = \sum_{v\in S(\pi,\phi)} c(v)$ be their total cost. The AMCSS problem is formally in Definition 2.1; it is NP-hard by reduction from adaptive seed minimization (ASM) with unit costs $c(v) = 1$ for all v, which is NP-hard [39].

Definition 2.1 (The AMCSS Problem). Given a graph $G = (V, E)$ with propagation probability $p(e)$ for each $e \in E$, cost $c(v)$ for each $v \in V$, and an influence threshold η, AMCSS seeks a policy π^* that minimizes the expected cost of selected seeds while ensuring the influence spread is at least η on any realization $\phi \in \Omega$. That is,

$$\pi^* = \text{argmin}_\pi \ \mathbb{E}_\Phi[C(\pi, \Phi)] \quad \text{s.t. } |I(S(\pi, \phi))| \geq \eta, \forall\, \phi \in \Omega.$$

Since AMCSS only requires reaching η, we use the *truncated influence spread* $\Gamma_\phi(S) = \min\{I_\phi(S), \eta\}$ for each realization ϕ [39]. In round i, given previous seeds $\mathcal{S}_{i-1} = \bigcup_{j=1}^{i-1} S_j$ that activated $I_\phi(\mathcal{S}_{i-1})$ nodes, the *marginal truncated influence* of $S_i \subseteq V_i$ is $\Gamma_\phi(S_i|\mathcal{S}_{i-1}) = \Gamma_\phi(S_i \cup \mathcal{S}_{i-1}) - \Gamma_\phi(\mathcal{S}_{i-1})$. Over all realizations, the expected versions are $\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] = \sum_\phi p_\phi \Gamma_\phi(S|\mathcal{S}_{i-1})$ and $\mathbb{E}[\Gamma(S)] = \sum_\phi p_\phi \Gamma_\phi(S)$.

2.3 Graphics Processing Units

GPUs are widely used for general-purpose parallel computing [5, 28, 29, 36, 38]. The most popular GPU programming framework is Nvidia CUDA, which provides a C/C++-like programming interface.

When a GPU *kernel* is launched, the GPU creates a grid of threads organized in a hierarchical structure: the *grid* partitions threads into thread *blocks* of equal size (e.g., 1024 threads per block), and each block further divides into *warps*, containing 32 threads each. We employ one-dimensional grid with grid size and block size denoted by $gridDim.x$ and $blockDim.x$, respectively. Grid size is the number of blocks, and block size is the number of threads in the block. GPUs follow the SIMT model, where all threads within a warp execute the same instruction in lockstep. This makes GPUs particularly efficient for data-parallel tasks with simple control flow, in contrast to CPU threads which can handle complex branching logic independently.

GPUs also feature a hierarchical memory architecture. The *global memory* is the largest memory accessible by all threads, while *shared memory* is a smaller memory shared among threads within the same block. Dynamic memory allocation within kernels is supported through the device heap. However, device heap allocations incur substantial overhead and cause fragmented memory layouts. Since multiple threads may update the same memory location concurrently, GPU programs must use atomic operations judiciously.

Therefore, GPUs excel at processing simple data structures (e.g., arrays) with regular access patterns (e.g., sequential or strided), but perform poorly on complex data structures and irregular random accesses. Straightforward parallelization of CPU algorithms on GPUs often yields suboptimal performance. GPU-accelerated algorithms should be carefully redesigned to leverage simple data layouts and regular memory access patterns for efficient execution.

2.4 mRR-set and ASTI

Here we review the most related studies, leaving other related work to Section 3. The work of [13, 14] first studies AMCSS and proposes a greedy policy that selects the seed with the maximum influence-to-cost ratio in each round, achieving an approximation ratio of $(\ln \eta + 1)^2$. However, this policy assumes exact truncated influence, which is #P-hard to compute [7]. To address this, Tang et al. introduce the *multi-root Reverse Reachable* set (mRR-set) to efficiently estimate truncated influence and design ASTI method [39] outlined below.

In round i, an mRR-set on residual graph G_i is generated as follows: (i) sample $k_i = n_i/\eta_i$ roots uniformly at random from V_i; (ii) run a stochastic reverse BFS from these roots, where each edge (u, v) is traversed with probability $p(u, v)$; and (iii) take all visited nodes (including the roots) as the mRR-set R. Since k_i may be non-integer, we realize it in expectation by sampling $\lceil k_i \rceil$ roots with probability $k_i - \lfloor k_i \rfloor$ and $\lfloor k_i \rfloor$ roots otherwise. A node set S *covers* an mRR-set R if $S \cap R \neq \emptyset$, and its *coverage* $\Lambda(S)$ is the number of mRR-sets it covers. Given θ mRR-sets, we estimate $\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})]$ by $\hat{\Gamma}(S|\mathcal{S}_{i-1}) = \eta_i \Lambda(S)/\theta$ to approximate $\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})]$ with bounded error [39]: $(1 - 1/e)\mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})] \leq \mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})] \leq \mathbb{E}[\Gamma(S|\mathcal{S}_{i-1})]$, where expectation $\mathbb{E}[\hat{\Gamma}(S|\mathcal{S}_{i-1})]$ is over mRR-set randomness.

The original ASTI assumes uniform seed cost, i.e., $c(v) = 1$ for all $v \in V$ [39]. Algorithm 1 extends ASTI to non-uniform costs. It iteratively selects seeds and triggers diffusion until the influence threshold η is reached (Line 3). In each round i, ASTI selects a batch of b seeds S_i that approximates the optimal batch S_i^o within a factor $\rho_b(1 - 1/e)(1 - \epsilon)$, where $\rho_b = (1 - 1/b)^b$ is initialized at Line 1 and ϵ is the estimation error parameter. Initially, ASTI generates a small number θ_i of mRR-sets. Using these sets, it greedily selects b seeds with the largest coverage-to-cost ratio (Lines 8–10), removing the mRR-sets they cover after each selection. It then computes a lower bound $\Lambda^l(S_i)$ on the coverage of S_i and an upper bound $\Lambda^u(S_i^o)$ on the optimal coverage (Lines 11–12). If $\Lambda^l(S_i)$ approximates $\Lambda^u(S_i^o)$ within the required ratio, S_i is accepted, added to S (Line 15), and influence diffusion is triggered (Line 16), updating the residual graph and influence target for the next round at Line 17. Otherwise, θ_i is doubled (Line 14), and the process from Lines 7–14 is repeated;

Algorithm 1: ASTI

Input: Graph G, target η, estimation error ϵ, batch size b, cost $c(\cdot)$.
Output: Seed set S.

$i \leftarrow 1, G_i \leftarrow G, \eta_i \leftarrow \eta, \rho_b \leftarrow 1-(1-1/b)^b, S \leftarrow \emptyset$;
while $\eta_i > 0$ **do**
  $\delta \leftarrow 1/n_i$; $\hat{\epsilon} \leftarrow (\epsilon-\delta)(1-\delta)$; $S_i \leftarrow \emptyset$;
  Estimate θ and $\theta_{\max}$; $\theta_i \leftarrow \theta$; $H \leftarrow \lceil \log_2(\theta_{\max}/\theta) \rceil + 1$;
  $a_1 \leftarrow \ln(3H/\delta) + \ln\binom{n_i}{b}$; $a_2 \leftarrow \ln(3H/\delta)$;
  while $\theta_i < \theta_{\max}$ **do**
    Generate additional mRR-sets until total reaches θ_i;
    while $|S_i| < b$ **do**
      $u \leftarrow \text{argmax}_{v \in V_i} \frac{\Lambda(S_i \cup \{v\}) - \Lambda(S_i)}{c(v)}$;
      $S_i \leftarrow S_i \cup \{u\}$;
    $\Lambda^l(S_i) \leftarrow (\sqrt{\Lambda(S_i) + 2a_1/9} - \sqrt{a_1/2})^2 - a_1/18$;
    $\Lambda^u(S_i^o) \leftarrow (\sqrt{\Lambda(S_i)/\rho_b + a_2/2} + \sqrt{a_2/2})^2$;
    if $\Lambda^l(S_i) > \rho_b(1-\hat{\epsilon})\Lambda^u(S_i^o)$ **then break**;
    $\theta_i \leftarrow 2\theta_i$;
  $S \leftarrow S \cup S_i$;
  Diffuse from S_i, and obtain the activated nodes A_i;
  $G_{i+1} \leftarrow G_i \backslash A_i$; $\eta_{i+1} \leftarrow \eta_i - |A_i|$; $i \leftarrow i+1$;

if θ_i reaches the maximum $\theta_{\max}$, S_i is also accepted. Overall, ASTI achieves an approximation ratio of $(\ln\eta+1)^2/[(1-1/e)(1-\epsilon)]$.

ASTI is a milestone for solving the AMCSS problem, but regenerates mRR-sets in each round , incurring substantial overhead. Even with CPU parallelization, it still needs thousands of seconds on real graphs. We aim to reduce this to hundreds or even tens of seconds using GPUs. As discussed in Section 2.3, GPUs have different compute and memory architectures, so achieving this requires new algorithmic designs and GPU-specific optimizations for AMCSS.

3 RELATED WORK

Minimum Cost Seed Selection (MCSS). The MCSS problem aims to find a seed set that minimizes the cost to achieve η. The problem shares the submodularity and monotonicity properties with the classic Influence Maximization (IM) problem [30]. Existing approaches can be divided into two categories. The first category focuses on reaching η only in expectation [12, 15, 18, 30], where selected seeds can guarantee the coverage meets η. However, the actual influence spread of the selected seed often falls significantly short of η. Therefore, they are not feasible for AMCSS that requires the selected nodes must influence at least η nodes in every realization. The other category addresses this practical limitation by providing probabilistic guarantees [12, 51], ensuring that with a predefined probability p the influence achieved by the selected seeds reaches η in a random realization. Thus, we can set a high probability that the selected seeds meet the target influence in practice.

Adaptive Seed Minimization (ASM). The adaptive setting in IM and MCSS receives considerable attention [6, 9, 11, 13, 14, 16, 19, 22, 35, 39, 43, 44, 52]. In the adaptive setting, nodes are seeded in batches at different rounds, and after each batch, an actual diffusion process unfolds and its outcomes are observed before selecting the next batch. Golovin and Krause [13] pioneered the extension of the classic submodularity and monotonicity to the adaptive setting, establishing an important theoretical foundation. However, they did not provide a practical algorithm for ASM. Tang et al. [39] first introduced truncated influence, recognizing that influence beyond a threshold η provides no additional value, and they devised mRR-sets to estimate marginal truncated influence targeted for ASM.

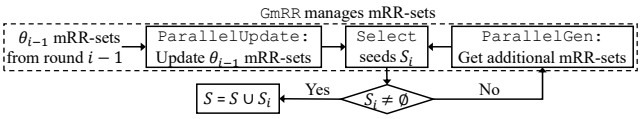


Figure 3: Overview of GAAS for round i.

However, ASTI assumes uniform node costs and overlooks their real-world heterogeneity. Moreover, ASTI is a single-threaded CPU-based method and regenerates all mRR-sets in each round.

Parallel RR-set Approaches. Several methods have been proposed to solve the IM problem in parallel [31, 32, 37] using the Reverse Reachable set (RR-set), which uses only one root. While the RR-set generation phase can be adapted for mRR-set generation in AMCSS, these approaches face notable efficiency challenges. Ripples and cuRipples [31, 32] utilize thread-level parallelization for RR-set generation. However, these methods suffer from workload imbalance due to skewed RR-set sizes and suboptimal thread utilization when RR-sets are fewer than the available threads. These issues remain when it is extended to mRR-set generation in adaptive rounds. For the block-level parallel approach [37], its implementation necessitates temporary storage and frequent atomic operations to manage RR-sets in global memory, also degrading performance.

4 GPU-ACCELERATED METHOD GAAS

4.1 Overview

GAAS is a GPU-centric framework for AMCSS, tightly coupling novel algorithmic designs with GPU-friendly mRR-set processing.

The dominant cost lies in sampling mRR-sets for influence estimation. On GPUs, efficiently managing these irregular, dynamically evolving samples is challenging. Further, to avoid regenerating mRR-sets from scratch in every round, we aim to reuse and update them across rounds, which introduces additional challenges in theoretical analysis, algorithm design, and GPU-parallel optimization.

Figure 3 outlines the GAAS method. After round $i-1$, GAAS first invokes ParallelUpdate to update the θ_{i-1} mRR-sets from round $i-1$ so they can be reused for round i, equivalent to generating new mRR-sets on the residual graph G_i. Then, Select attempts to select the seeds S_i, based on the updated mRR-sets. If S_i is not empty, GAAS returns S_i for round i without generating new mRR-sets. Otherwise, GAAS invokes ParallelGen to get additional mRR-sets, and then invokes Select again until S_i is not empty. Throughout, mRR-sets are managed by the GmRR structure, which interacts with the GPU kernels for both updating and generating mRR-sets.

The GPU-tailored GmRR structure stores, manages, and updates mRR-sets (Section 4.2). It reconciles the irregular, evolving nature of mRR-sets with massive GPU parallelism via a specialized global memory layout. Specifically, GmRR utilizes a pre-allocated global array partitioned into disjoint *segments* of equal size, each mapped exclusively to a thread block. This ownership enables blocks to generate and update mRR-sets locally, eliminating expensive dynamic memory allocation and excessive atomic contention. Internally, each segment functions as a *circular* buffer, allowing mRR-sets to be appended and updated efficiently, supporting the update of mRR-sets across rounds without frequent costly data shifting.

To efficiently generate mRR-sets on GPU, with GmRR, we propose the ParallelGen kernel (Section 4.3). At the inter-block level, it evenly partitions mRR-set generation across thread blocks, which

write mRR-sets into their own segments; at the intra-block level, it employs a GPU kernel `Probe` to perform stochastic BFS, where threads cooperatively traverse the nodes to construct an mRR-set.

To enable reuse of mRR-sets across rounds, we develop the `ParallelUpdate` kernel with its supporting theory (Section 4.4). An mRR-set from round i–1 reused in round i is *polluted* if it contains any node activated in round $i-1$, and *clean* otherwise. Both types must be updated to preserve the theoretical properties of mRR-sets for accurate influence estimation: clean mRR-sets are augmented with additional roots to satisfy the sampling requirements on the residual graph G_i, whereas polluted mRR-sets are repaired by removing activated nodes and resampling roots together with their visited nodes. Building on these theoretical insights, we design the `ParallelUpdate` kernel over `GmRR` for efficient mRR-set updates on GPUs. A key technique is *circular segment update* that treats each segment as a circular buffer, enabling updates and appends in the segment, and eliminating costly cumulative element shifts. Moreover, we design a load-balancing scheme to mitigate inter-block workload skew from uneven numbers of polluted mRR-sets across segments, further improving the efficiency of `ParallelUpdate`.

Besides, we devise a GPU `Select` kernel (Section 4.5) for seed selection. It uses a parallel reduction to find the node with the maximum coverage-to-cost ratio, and marks all mRR-sets containing it as covered while decrementing the coverage of nodes in them. After selecting b seeds, `Select` evaluates confidence bounds to decide whether the batch is accepted or more mRR-sets are required.

4.2 GPU-Tailored mRR-set Structure `GmRR`

A naive idea to parallelize the algorithms is assigning each GPU thread the task of generating a full mRR-set, including root sampling and stochastic BFS. However, this results in highly irregular and heavy workloads per thread, leading to poor GPU utilization.

Another approach is block-level parallelization, where all threads in a block construct one mRR-set. However, this still incurs substantial overhead: the mRR-set size is unknown until completion, multiple blocks may contend for a global write pointer. Accordingly, the mRR-sets need to be buffered via device-heap allocations before being copied to global memory, which is inefficient on GPUs.

Furthermore, we aim to update and reuse existing mRR-sets from the previous round $i-1$ when selecting seeds in round i, improving overall efficiency. Hence, we design `GmRR`, a GPU-tailored structure for storing mRR-sets. `GmRR` uses a *segment*-based layout where each thread block is assigned a dedicated, non-overlapping segment in global memory to store its generated mRR-sets. Each block stores its mRR-sets contiguously in its own segment, and segments are mutually disjoint. This removes both the global write pointer (each block writes only to its segment) and temporary buffer allocations, since all elements are written directly to global memory. In addition, `GmRR` is implemented as flat *arrays* for efficient GPU access. The per-block segments further enable dynamic updates of mRR-sets via a *circular layout within each segment*, detailed in Section 4.4.

Figure 4 shows a `GmRR` instance with two segments, each owned by one thread block. `GmRR` consists of three arrays: *mrr* stores all mRR-sets, $offset$ records start positions of mRR-sets, and $flag$ marks whether an mRR-set is covered/polluted. The size of *mrr* is estimated as follows. In practice, given a new graph, we first get

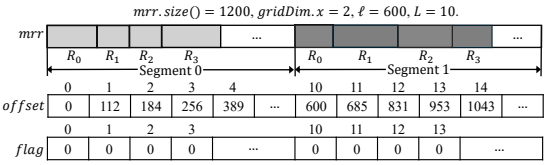


Figure 4: Example of `GmRR` with *mrr*, $offset$ and $flag$ arrays.

the theoretical upper bound on the number of mRR-sets needed in the first round, typically the largest across all rounds. We then generate a small sample of mRR-sets and measure their average size. Multiplying the two quantities gives the pre-allocated memory. This provides a practical safety margin without excessive overhead, so overflow rarely occurs. If it happens in a round, we simply allocate more memory and regenerate mRR-sets for that round.

Moreover, mRR-set memory usage across rounds is complex, where the peak level is similar but the memory also increases and drops between peaks (see Figure 17 in the technical report [2]). Therefore, we do not resize the pre-allocated memory across rounds.

We partition *mrr* into $gridDim.x$ non-overlapping segments, where $gridDim.x$ is the number of launched thread blocks. Each segment has size ℓ that is the size of *mrr* divided by $gridDim.x$. For example, the size of *mrr* is 1200 and ℓ is 600 in Figure 4. Segment j stores the mRR-sets generated by Block j, starting at the position $j \cdot \ell$ and ending at $(j+1) \cdot \ell - 1$ in *mrr*. As explained in Section 2.4, in round i each mRR-set has k_i roots and a variable number of nodes reached by reverse BFS. We store it as a contiguous sequence of node IDs: the first k_i entries are the roots, followed by the visited nodes. Each block uses all its threads to generate mRR-sets and stores them within its segment. In Figure 4, Segment 0 (gray) stores the four mRR-sets R_0–R_3 generated by Block 0, and Segment 1 (dark) stores those generated by Block 1.

In `GmRR`, mRR-sets in a segment are stored contiguously. Thus, an mRR-set is identified by the start positions of two contiguous mRR-sets in $offset$. For example, in Segment 0, R_2 and R_3 start at $offset[2] = 184$ and $offset[3] = 256$, so R_2 occupies $mrr[184]$ to $mrr[255]$. The last mRR-set has its end position stored in $offset[r+1]$, where r is the number of mRR-sets in the segment. For example, the end position of R_3 in Segment 0 is at $offset[4]$, which is also the start of free space in the segment for future mRR-sets.

Given the required number of mRR-sets per round, we evenly assign them to segments to simplify lookup and balance workload across blocks. The remaining question is how many values to store in $offset$ for each segment. Since the number of mRR-sets per segment may vary across rounds, we avoid reconstructing $offset$ whenever it changes. As the global maximum number of mRR-sets over all rounds has an upper bound and the number of segments is fixed, we predefine a sufficiently large per-segment capacity of L entries by dividing the upper bound by $gridDim.x$ and rounding up to the nearest power of two. The size of $offset$ is then $L \cdot gridDim.x$, since each segment corresponds one-to-one with a thread block. Every L entries in $offset$ record the start positions of mRR-sets in the corresponding segment. The start position of the r-th mRR-set in segment j is given by $offset[r + j \cdot L]$. For example, in Figure 4, $L = 10$, so R_1 in Segment 1 starts at $offset[1 + 1 \cdot L] = offset[11]$.

Accordingly, the $flag$ array also has $L \cdot gridDim.x$ entries; entry $(r + j \cdot L)$ marks whether the r-th mRR-set in segment j is covered/polluted or not. Section 4.5 details how $flag$ is updated.

Algorithm 2: The `ParallelGen` Kernel

Input: Graph G_i, coverage counter cnt, mrr, $offset$, θ_{i-1}, θ_i.

1 $r \leftarrow \theta_{i-1}/gridDim.x + \mathbb{I}_{\{(\theta_{i-1} \,\%\, gridDim.x) > block.id\}}$;
2 $r' \leftarrow \theta_i/gridDim.x + \mathbb{I}_{\{(\theta_i \,\%\, gridDim.x) > block.id\}}$;
3 $b\_w \leftarrow offset[r + block.id \cdot L]$; $b\_r \leftarrow b\_w$;
4 **while** $r < r'$ **do**
5 $\quad \forall u \in V_i$, $visited[u] \leftarrow false$ in **parallel**;
6 $\quad$ **parallel for** $j = thread.id$; $j < k_i$; j += $blockDim.x$ **do**
7 $\quad\quad$ Atomically sample $u \in V_i$ with $visited[u] == flase$ and set $visited[u] \leftarrow true$;
8 $\quad\quad$ $mrr[(b\_w + j) \,\%\, \ell + block.id \cdot \ell] \leftarrow u$;
9 $\quad\quad$ atomicAdd($cnt[u]$, 1);
10 $\quad$ __syncthreads();
11 $\quad$ `Probe`(G_i, k_i, b_r, b_w, mrr, $visited$, cnt);
12 $\quad$ $r \leftarrow r + 1$; $offset[r + block.id \cdot L] \leftarrow b\_w$; $b\_r \leftarrow b\_w$;

Algorithm 3: The `Probe` Procedure

Input: Graph G_i, root number k, b_r, b_w, mrr, $visited$, cnt

1 $b\_w \leftarrow b\_r + k$; $head \leftarrow 0$; $tail \leftarrow k$; $lane.id \leftarrow thread.id \,\%\, 32$;
2 **while** $head < tail$ **do**
3 $\quad$ **if** $lane.id == 0$ **then** $idx \leftarrow$ atomicAdd($head$, 1); __syncwarp();
4 $\quad$ Broadcast idx to threads in the same warp;
5 $\quad$ **if** $idx < tail$ **then**
6 $\quad\quad$ $u \leftarrow mrr[(b\_r + idx) \,\%\, \ell + block.id \cdot \ell]$;
7 $\quad\quad$ **parallel for** $j = lane.id$; $j < |N_{in}(u)|$; j += 32 **do**
8 $\quad\quad\quad$ Get the j-th node $v \in N_{\text{in}}(u)$;
9 $\quad\quad\quad$ **if** $p(v, u)$ *meets the condition in IC or LT* **then**
10 $\quad\quad\quad\quad$ **if** $visited[v] == flase$ *and set* $visited[v] \leftarrow true$ **then**
11 $\quad\quad\quad\quad\quad$ $mrr[\text{atomicAdd}(b\_w, 1) \,\%\, \ell + block.id \cdot \ell] \leftarrow v$;
12 $\quad\quad\quad\quad\quad$ atomicAdd($cnt[v]$, 1); atomicAdd($tail$, 1);
13 $\quad\quad$ __syncwarp();

4.3 Parallel mRR-set Generation

In round i, `GAAS` reuses and updates the θ_{i-1} mRR-sets from round $i-1$. (Section 4.4). If the required number θ_i exceeds θ_{i-1}, we generate $\theta_i - \theta_{i-1}$ additional mRR-sets. Especially, in round 1, we generate θ_1 mRR-sets from scratch, by applying the GPU kernel `ParallelGen`, where each block uses all its threads to construct new mRR-sets and write them into its own segment in `GmRR`.

The `ParallelGen` Kernel. Algorithm 2 presents the `ParallelGen` kernel for mRR-sets generation in a block $block.id$. In GPUs, block IDs are assigned contiguously from 0 to $gridDim.x - 1$, which we also use to index the segments in `GmRR`. As mentioned, when θ mRR-sets are needed, we let each segment store on average $\bar{\theta} = \theta/gridDim.x$ mRR-sets. Often, $\bar{\theta}$ is not an integer. In this case, we require the first $\theta \% gridDim.x$ segments to store $\lceil \bar{\theta} \rceil$ mRR-sets, and the remaining segments to store $\lfloor \bar{\theta} \rfloor$ mRR-sets. Therefore, in Line 1 we compute the number of existing mRR-sets r in the segment owned by block $block.id$ as $\theta_{i-1}/gridDim.x$, plus one if $block.id < \theta_{i-1} \% gridDim.x$. Likewise, in round i with θ_i mRR-sets required, we set the target number r' of mRR-sets in this segment to $\theta_i/gridDim.x + \mathbb{I}_{\{\theta_i \% gridDim.x > block.id\}}$. The difference $r' - r$ is the number of new mRR-sets that block $block.id$ generates in round i and stores in its segment. In Line 3, we set the starting write position b_w for the next mRR-set as the end of the last existing mRR-set in the segment, *i.e.*, $offset[r + block.id \cdot L]$. We also initialize $b\_r \leftarrow b\_w$, where b_r is the read pointer used by `Probe` (Algorithm 3) for stochastic BFS within the current mRR-set.

Each iteration of the while loop (Lines 4–12) constructs one mRR-set using all threads in the block. We first mark all nodes in the residual graph G_i as unvisited in parallel (Line 5). Next, we uniformly and parallelly sample k_i distinct roots from V_i (Line 7): thread j samples a root u and writes it to mrr at position $(b\_w + j) \% \ell + block.id \cdot \ell$ (Line 8). This modulo-based address within the segment enables circular updates later. Line 9 atomically increments $cnt[u]$, the *coverage count* of node u (the number of mRR-sets containing u). We say that u covers an mRR-set if it appears in that set. After all k_i roots are written, `Probe` (Line 11) performs reverse BFS from these roots, appending newly visited nodes to the mRR-set and updating b_w, mrr, and cnt. Finally, we advance r, record the end position of the new mRR-set in $offset[r + block.id \cdot L] \leftarrow b\_w$, and set $b\_r \leftarrow b\_w$ to start the next mRR-set generation (Line 12).

The `Probe` Kernel. Given arbitrary k roots, `Probe` performs reverse BFS from the roots to complete the mRR-set. b_r denotes the start of the current mRR-set and is used to index frontier nodes whose in-neighbors are probed. b_w denotes the next write position in mrr within the segment of $block.id$. We maintain a BFS queue indexed by $head$ and $tail$, initialized to 0 and k respectively, so that $tail - head$ equals the number of frontier nodes pending expansion. Each warp cooperatively processes the in-neighbors of a single frontier node, while different warps handle different frontier nodes. The lane identifier $lane.id$ of each thread is $thread.id$ mod 32. The loop in Line 2 proceeds while the BFS queue is non-empty. In each iteration, the first thread in the warp ($lane.id = 0$) obtains the frontier index idx (Line 3), which is then broadcast to all threads in the warp (Line 4). If idx is valid, i.e., smaller than $tail$ (Line 5), each warp retrieves the frontier node u (Line 6). The threads in warp iterate over a u's in-neighbors v (Line 8), and check whether edge (v, u) satisfies the IC or LT activation condition in Section 2.1 (Line 9). If so, unvisited v is appended to the mRR-set at the next circular position in mrr by atomically incrementing b_w (Line 11), and we update $visited[v]$, $cnt[v]$, and the BFS queue $tail$ (Line 12).

4.4 Parallel mRR-set Update

In round i, we reuse mRR-sets from round i–1. After selecting seeds S_{i-1} and simulating diffusion, we obtain activated nodes A_{i-1} and form the residual graph G_i by removing A_{i-1} and their incident edges from G_{i-1}. An mRR-set from round $i-1$ is *polluted* if it contains any node in A_{i-1}; otherwise it is *clean*.

Such reuse is not straightforward, but requires proper updates to *both* clean and polluted mRR-sets, as analyzed in Lemma 4.1 and 4.3 later, so that each updated mRR-set has exactly the same distribution as an mRR-set generated directly on G_i, to preserve the theoretical properties needed for accurate influence estimation. In round i, each mRR-set must contain $k_i = n_i/\eta_i = (n_{i-1} - |A_{i-1}|)/(\eta_{i-1} - |A_{i-1}|)$ roots, and k_i is monotonically non-decreasing over rounds (*i.e.*, $k_i \geq k_{i-1}$), since $|A_{i-1}| \geq 1$ and $n_i \geq \eta_i$. Hence, even clean mRR-sets must be updated to satisfy the new root requirement. Moreover, updating mRR-sets in `GmRR` requires dedicated designs to avoid excessive data movement on the GPU. Below, we first analyze updates of clean and polluted mRR-sets, then present a *circular segment update* strategy over `GmRR`, and finally introduce the `ParallelUpdate` kernel.

Given the activated node set A_{i-1} from round $i-1$, we first identify polluted mRR-sets in `GmRR` by invoking the `Discover` kernel (Algorithm 6) with A_{i-1} as input: every mRR-set containing a node in A_{i-1} is marked as polluted by setting its entry in the $flag$ array to true, and the coverage counts of nodes in these polluted mRR-sets

are decremented to remove their contribution in round i. The $flag$ array is reused in Section 4.5 for seed selection. Thus, we defer the details of `Discover` to there and focus here on updating mRR-sets.

Theoretical Analysis on Updating a Clean mRR-set. Let R be a clean mRR-set from round $i-1$ with root set K of size k_{i-1}. Because R contains no node in A_{i-1}, its reverse BFS and root distribution remain valid on G_i. Since $k_i \geq k_{i-1}$, we update R by keeping all existing nodes and appending $k_i - k_{i-1}$ additional roots and their reverse BFS expansions to meet the root requirement in round i.

Specifically, in Lemma 4.1, we show that when adding $k_i - k_{i-1}$ new roots, instead of sampling them all from the residual graph G_i, we can split the new roots into two subsets. First, we can sample k_∇ nodes uniformly from $R \backslash K$ and promote them to roots. Since these promoted roots already exist in R, they do not require additional reverse BFS, saving computation. The remaining $k_\Delta = k_i - k_{i-1} - k_\nabla$ new roots are sampled uniformly at random from $V_i \backslash R$ in G_i and expanded via reverse BFS by the `Probe` kernel. Lemma 4.1 establishes the theoretical correctness of this update strategy. We provide proof sketches here and all proofs are in technical report [2].

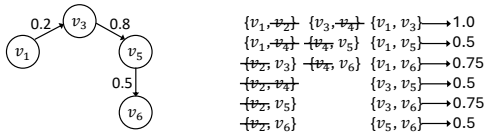
Lemma 4.1. *If the previous mRR-set R with k_{i-1} roots K is clean, the nodes in R can be preserved. Additionally, we can select $k_\Delta = (k_i - k_{i-1}) \cdot \frac{n_i - |R|}{n_i - k_{i-1}}$ roots from $V_i \backslash R$ and the rest $k_\nabla = k_i - k_{i-1} - k_\Delta$ roots from $R \backslash K$ uniformly at random. The resultant root set has the same distribution as one directly generated from V_i.*

Proof sketch. To prove that R can be preserved, we show that the distribution of nodes already in R matches that in a newly generated mRR-set. Our root-addition strategy is correct because, even if a new root coincides with a node already in R, the content of R remains unchanged, and the additional roots selected from V_i follow the same distribution as those in a fresh mRR-set. □

Theoretical Analysis on Updating a Polluted mRR-set. For a polluted mRR-set R from round $i-1$, since it contains at least one node in A_{i-1}, its reverse BFS generated on G_{i-1} is no longer valid on G_i. A naive remedy is to discard all polluted mRR-sets and regenerate them by sampling fresh roots from G_i. However, this alters the global distribution of root sets over all mRR-sets and thus breaks the approximation guarantee, as illustrated in Example 4.2.

Example 4.2. Assume the algorithm starts with $\eta = 4$ on graph G under the IC model in Figure 2(a). We sample sufficiently many mRR-sets, each with $\lceil n/\eta \rceil = 2$ roots. Figure 5(b) enumerates all $\binom{6}{2} = 15$ possible root sets, each with probability $1/15$. In the first round, v_2 is selected and, under the realization in Figure 2(b), activates v_2 and v_4. In the next round, we work on the residual graph G_2 in Figure 5(a) with $\eta_2 = 2$, still requiring mRR-sets with $\lceil n_2/\eta_2 \rceil = 2$ roots. If, during update, we regenerate every polluted mRR-set by resampling both roots uniformly from the remaining nodes, the root distribution becomes biased. For instance, $\frac{3}{5}$ of the mRR-sets whose root set contains v_2 or v_4 are regenerated, while mRR-sets such as $\{v_1, v_3\}$ never include v_2. In total, $\frac{11}{15}$ mRR-sets are regenerated, and the final probability of sampling root set $\{v_1, v_3\}$ becomes $\frac{1}{15} + \frac{11}{15} \cdot \frac{1}{6} = \frac{17}{90}$, deviating from the uniform probability $\frac{1}{6}$ and thus introducing sampling bias.

To address this, Lemma 4.3 shows that we should retain all non-activated roots in polluted mRR-sets and resample only the activated ones. For a polluted mRR-set R with root set K of size k_{i-1},



(a) The residual graph G_2 (b) Root sets

Figure 5: A residual graph and root sets (activated nodes are struck through).

let $K_1 = K \setminus A_{i-1}$ and $K_2 = K \cap A_{i-1}$. We discard the roots in K_2 and sample $|K_2| + k_i - k_{i-1}$ new roots uniformly at random from $V_i \setminus K_1$; by Lemma 4.3, the resulting root set has the same distribution as one generated directly from V_i. We then invoke the `Probe` kernel to perform reverse BFS from these roots to update the mRR-set.

Lemma 4.3. *Let K be the set of roots of a random mRR-set R from round $i-1$. We write $K = K_1 \cup K_2$, where K_1 is not activated while K_2 is. Then, in round i, we delete K_2 from K and select $|K_2| + k_i - k_{i-1}$ nodes from $V_i \backslash K_1$ uniformly at random. The distribution of the new root set is the same as the one generated from V_i directly.*

Proof sketch. Let K' be a set of k_i nodes sampled at random from V_i, and let Z be any k_i-subset of V_i. For a freshly generated mRR-set on G_i, $\Pr[K' = Z] = 1/C_{n_i}^{k_i}$. We then show that our method also selects Z with probability $1/C_{n_i}^{k_i}$. Hence, it induces the same root-set distribution as fresh generation. □

Example 4.4. To illustrate Lemma 4.3, we revisit Example 4.2. We keep all clean mRR-sets and retain non-activated roots in polluted ones. For root set $\{v_1, v_3\}$, $\frac{2}{15}$ of mRR-sets already contain v_1 or v_3. The resampling of the other root results in $\{v_1, v_3\}$ with probability $\frac{1}{3}$, contributing $\frac{2}{15} \cdot 2 \cdot \frac{1}{3}$. Among the $\frac{1}{15}$ mRR-sets with roots $\{v_2, v_4\}$, regeneration yields $\{v_1, v_3\}$ with probability $\frac{1}{6}$, contributing $\frac{1}{15} \cdot \frac{1}{6}$. The original $\frac{1}{15}$ mRR-sets with roots $\{v_1, v_3\}$ remain unchanged. Thus the final probability of $\{v_1, v_3\}$ is $\frac{2}{15} \cdot 2 \cdot \frac{1}{3} + \frac{1}{15} \cdot \frac{1}{6} + \frac{1}{15} = \frac{1}{6}$. A symmetric argument shows every root set has probability $\frac{1}{6}$, confirming unbiased root-set sampling.

Circular Segment Update of mRR-sets on `GmRR`. With the theoretical analysis, we now describe how to efficiently update mRR-sets within a `GmRR` segment. Recall that each thread block exclusively owns one segment and is responsible for generating and updating its mRR-sets. The top-left part of Figure 6 illustrates Segment 0 with four mRR-sets $R_{0,1,2,3}$ generated by Block 0 in round $i-1$, where R_0 and R_3 are clean, while R_1 and R_2 are polluted as indicated in the $flag$ array. In round i, all four mRR-sets must be updated.

A straightforward approach is to update mRR-sets sequentially from R_0 to R_3. Since R_0 is clean, we add new roots and perform reverse BFS, increasing its size and forcing us to shift subsequent mRR-sets R_1–R_3 backward to free space for the expanded R_0. Updating the polluted mRR-set R_1 likewise changes its size after root resampling and reverse BFS, so all subsequent mRR-sets must be shifted again. Repeating this process for all mRR-sets requires moving each set multiple times, which is inefficient when many mRR-sets reside in a segment, and the unknown final size of each updated mRR-set also requires temporary GPU buffers.

Instead, we design a circular segment update strategy that moves each mRR-set at most once, and writes new content circularly within the segment, so no subsequent mRR-sets are shifted. Steps ❶–❹ in Figure 6, together with the $offset$ array, illustrate this

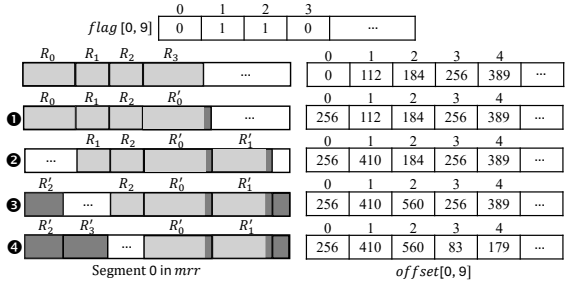


Figure 6: Circular segment update of a segment in GmRR.

process. As explained in Section 4.2, each GmRR segment stores mRR-sets sequentially in mrr, with $offset$ recording the start of their positions and the free space. Especially, we first update the last mRR-set in the segment in place. In Figure 6, the last mRR-set R_3 is updated to R'_0, which becomes the first mRR-set in the segment for round i (Step ❶); its light-gray region is the reused content, and its dark-gray region is the part newly probed from added roots. Accordingly, we set $offset[0]=256$, the start position of R'_0, which also indicates that R'_0 is now the first mRR-set in the segment.

Next, we update R_0 from round $i-1$ to R'_1. Instead of modifying R_0 in place (which would require shifting subsequent mRR-sets), we write R'_1 immediately after R'_0, leaving the original space of R_0 free (white in Step ❷). Originally the first mRR-set, R_0 becomes the second after the update, starting at $offset[1] = 410$. We then update R_1 to R'_2 and place it after R'_1, where R'_2 has start position $offset[2] = 560$, making it the third mRR-set in the segment (Step ❸). When the segment end is reached, the remaining free space lies at its beginning. Using circular addressing within the segment (mapping each physical index modulo the segment length ℓ), the portion of R'_2 exceeding the segment end is wrapped to the segment front (dark gray in Step ❸). The free space after R'_2 is then used to update R_2 to R'_3, starting at $offset[3] = 83$; as the last mRR-set, its end position is stored at $offset[4] = 179$ (Step ❹).

The `ParallelUpdate` Kernel. Algorithm 4 presents the kernel `ParallelUpdate`, which updates mRR-sets in each segment by a thread block. The input is the θ_{i-1} mRR-sets in GmRR from round $i-1$, represented by mrr and $offset$, and the array $flag$ marking each mRR-set as polluted or clean. For each block, we first compute its local mRR-set count r (Line 1), set r_1 to index the updated mRR-sets and r_2 to index the mRR-set being processed (Line 2), where $r_2 = r-1$ so that we start updating from the last mRR-set. ℓ_b is the base address of the block's segment. We then read the $start$ and end of the r_2-th mRR-set and set b_w to $start$ as the write position for its updated version (Lines 3–4). In each iteration (Line 5), we mark all nodes in V_i as unvisited in parallel, then set both b_r and b_w to the new start position of the mRR-set pending update (Line 6), and compute its original length len under the circular layout (Line 7).

From Lines 8 to 18, if the r_2-th mRR-set is clean, we first obtain k_Δ and k_∇ according to Lemma 4.1 (Line 9). We then retain all original nodes of this mRR-set by copying them to their new position and marking them as visited in the updated mRR-set (Lines 10–12), with all operations executed in parallel. Next, we sample k_Δ new roots and write each new root to position $(b\_w+len+j)\%\ell + \ell_b$ of mrr, updating the $visited$ and cnt arrays accordingly (Lines 13–15). We then perform reverse BFS from these k_Δ new roots by invoking the `Probe` kernel (Line 17). Note that the k_Δ new roots and k_∇ promoted

Algorithm 4: The `ParallelUpdate` Kernel

Input: Graph G_i, cnt, mrr, $offset$, $flag$, θ_{i-1}.

1 $r \leftarrow \theta_{i-1}/gridDim.x + \mathbb{I}_{\{(\theta_{i-1}\ \%\ gridDim.x) > block.id\}}$;
2 $r_1 \leftarrow 0$; $r_2 \leftarrow r-1$; $\ell_b \leftarrow block.id \cdot \ell$;
3 $start \leftarrow offset[r_2 + block.id \cdot L]$; $b\_w \leftarrow start$;
4 $end \leftarrow offset[r_2 + 1 + block.id \cdot L]$;
5 **while** $r_1 < r$ **do**
6 $\forall u \in V_i, visited[u] \leftarrow false$ in **parallel**; $b\_r \leftarrow b\_w$;
7 $len \leftarrow (end > start)\ ?\ (end - start) : \ell - (start - end)$;
8 **if** $flag[r_2 + block.id \cdot L] == false$ **then** `// clean`
9 Obtain k_Δ and k_∇ *w.r.t.* Lemma 4.1;
10 **parallel for** $j = thread.id; j < len; j \mathrel{+}= blockDim.x$ **do**
11 $u \leftarrow mrr[(start + j)\ \%\ \ell + \ell_b]$; $visited[u] \leftarrow true$;
12 $mrr[(b\_w + j)\ \%\ \ell + \ell_b] \leftarrow u$;
13 **parallel for** $j = thread.id; j < k_\Delta; j \mathrel{+}= blockDim.x$ **do**
14 Atomically sample $u \in V_i$ with $visited[u] == flase$ and set $visited[u] \leftarrow true$;
15 $mrr[(b\_w+len+j)\%\ell+\ell_b] \leftarrow u$; atomicAdd($cnt[u]$, 1);
16 __syncthreads();
17 `Probe`($G_i, k_\Delta, b\_r + len, b\_w, mrr, visited, cnt$);
18 **parallel** swap k_Δ and k_∇ roots to after previous k_{i-1} roots;
19 **else** `// polluted`
20 **parallel for** $j = thread.id; j < k_i; j \mathrel{+}= blockDim.x$ **do**
21 **if** $j < k_{i-1}$ **then**
22 $u \leftarrow mrr[(start+j)\%\ell+\ell_b]$; $visited[u] \leftarrow true$;
23 __syncthreads();
24 **if** $u \notin V_i$ *or* $j \geq k_{i-1}$ **then**
25 Atomically sample $u \in V_i$ with $visited[u] == flase$ and set $visited[u] \leftarrow true$;
26 $mrr[(b\_w+j)\%\ell+\ell_b] \leftarrow u$; atomicAdd($cnt[u]$, 1);
27 __syncthreads();
28 `Probe`($G_i, k_i, b\_r, b\_w, mrr, visited, cnt$);
29 $r_2 \leftarrow (r_2 + 1)\ \%\ r$; $start \leftarrow offset[r_2 + block.id \cdot L]$;
30 $end \leftarrow offset[r_2 + 1 + block.id \cdot L]$;
31 $offset[r_1 + block.id \cdot L] \leftarrow b\_r$; $r_1 \leftarrow r_1 + 1$;
32 $offset[r_1 + block.id \cdot L] \leftarrow b\_w$;

roots are not necessarily contiguous with the original roots of this mRR-set. As described in Section 4.2, each mRR-set in mrr stores all roots first, followed by nodes probed by reverse BFS. Thus, in Line 18 we swap the positions of the k_Δ and k_∇ roots so that the first k_i entries of the updated mRR-set are its roots.

From Lines 20 to 26, if the r_2-th mRR-set is polluted, we first retrieve its previous roots at their previous positions in the existing mRR-set (Lines 21–22) and, for each root u, test whether $u \in V_i$. If $u \notin V_i$ (i.e., it was activated in the previous round) or fewer than k_i valid roots remain, we sample replacement roots so that the updated mRR-set still has exactly k_i roots (Lines 24–25). Finally, we write all roots to their new positions in mrr and atomically increase the coverage of each root by 1 (Line 26). Then `Probe` is invoked from these k_i roots to get the mRR-set (Line 28). Once the r_2-th mRR-set is updated, we move to the next one by updating r_2, recomputing its original $start$ and end, recording the start position of the updated mRR-set in $offset$, and incrementing r_1 (Lines 29–31). After all mRR-sets are processed, we set b_w as end of last updated mRR-set, which is also the start of the free space in the segment (Line 32).

Update Workload Balancing. In `ParallelUpdate`, the main cost comes from updating polluted mRR-sets, which requires more BFS traversals than clean ones. Thus, the workload per segment in GmRR is largely determined by its number of polluted mRR-sets. This can be highly skewed across segments, causing severe inter-block imbalance. To alleviate this, we adopt a lightweight load-balancing scheme. Using the $flag$ array, we first count in parallel the polluted

Algorithm 5: The `Select` Kernel

Input: Batch size b, cnt, mrr, $offset$, $flag$, θ_i.
1 $S_i \leftarrow \emptyset$; $\Lambda(S_i) \leftarrow 0$; $flag[\cdot] \leftarrow false$; $cnt'[\cdot] \leftarrow cnt[\cdot]$;
2 **while** $|S_i| < b$ **do**
3 $u \leftarrow \text{argmax}_{v \in V_i} \frac{cnt'[v]}{c(v)}$ by **parallel** reduction on GPU;
4 $S_i \leftarrow S_i \cup \{u\}$, $\Lambda(S_i) \leftarrow \Lambda(S_i) + cnt'[u]$;
5 Kernel `Discover`(cnt', mrr, $offset$, $flag$, θ_i, $\{u\}$);
6 Derive $\Lambda^l(S_i)$ and $\Lambda^u(S_i^o)$;
7 **if** $\Lambda^l(S_i) > \rho_b(1-\hat{\epsilon})\Lambda^u(S_i^o)$ **then** {$cnt[\cdot] \leftarrow cnt'[\cdot]$; **return** S_i;}
8 **else return** $\emptyset$;

Algorithm 6: The `Discover` Kernel

Input: cnt, mrr, $offset$, $flag$, θ_i, node set A.
1 $r \leftarrow \theta_i / gridDim.x + \mathbb{I}_{\{(\theta_i \,\%\, gridDim.x) > block.id\}}$; $r_1 \leftarrow 0$;
2 **while** $r_1 < r$ **do**
3 $start \leftarrow offset[r_1 + block.id \cdot L]$;
4 $end \leftarrow offset[r_1 + 1 + block.id \cdot L]$;
5 $len \leftarrow (end > start)\,?\,(end - start) : \ell - (start - end)$;
6 **if** $flag[r_1 + block.id \cdot L] == false$ **then**
7 **parallel for** $j = thread.id$; $j < len$; j+=$blockDim.x$ **do**
8 **if** $mrr[(start + j) \,\%\, \ell + block.id \cdot \ell] \in A$ **then**
9 $flag[r_1 + block.id \cdot L] \leftarrow true$;
10 **if** $flag[r_1 + block.id \cdot L] == true$ **then break**;
11 __syncthreads();
12 **parallel for** $j = thread.id$; $j < len$; j+=$blockDim.x$ **do**
13 $u \leftarrow mrr[(start + j) \,\%\, \ell + block.id \cdot \ell]$;
14 atomicSub($cnt[u]$, 1);
15 __syncthreads();
16 $r_1 \leftarrow r_1 + 1$;

mRR-sets in each segment, sort all segment IDs in decreasing order of this count and partition them into β groups, each containing as many segments as blocks. Segments in the same group have similar workloads, and `ParallelUpdate` processes groups one by one over β iterations, which reduces workload imbalance among blocks.

The time breakdown in Figure 12 of Section 5.4 shows that, in `GAAS`, the `ParallelUpdate` kernel significantly reduces the time spent on mRR-set generation compared with a regeneration-based variant GAAS_{gen} with its time dominated by mRR-set generation. Consequently, the overall runtime of `GAAS` is much lower, demonstrating the substantial benefit of reusing and updating mRR-sets.

4.5 Parallel Seed Selection

In round i with θ_i mRR-sets, we greedily select a batch S_i of b seeds that maximizes the coverage-to-cost ratio. We use the GPU kernel `Select` (Algorithm 5) to perform one seed-selection attempt. We initialize $S_i = \emptyset$ with $\Lambda(S_i) = 0$, copy cnt to the working array cnt' (since this attempt may be rejected), and set all entries of $flag$ to false, as no mRR-set is covered before the first selection. In each iteration (Lines 2–5), we perform a GPU parallel reduction to select the node $u \in G_i$ with the maximum coverage-to-cost ratio, add u to S_i, and update $\Lambda(S_i)$. We then invoke the `Discover` kernel (Algorithm 6) to mark mRR-sets containing u as covered and decrement the corresponding coverage counts cnt'. After b seeds are selected, we test if S_i is qualified by computing the lower bound $\Lambda^l(S_i)$ of its coverage and the upper bound $\Lambda^u(S_i^o)$ for the optimal seed batch. If $\Lambda^l(S_i) > \rho_b(1-\hat{\epsilon})\Lambda^u(S_i^o)$ holds, S_i is accepted, we overwrite cnt with cnt', and return S_i (Line 7); otherwise, we return empty (Line 8), indicating that additional mRR-sets are needed.

Algorithm 7: `GAAS`

Input: Graph G, target η, estimation error ϵ, batch size b, cost $c(\cdot)$.
Output: Seed set S.
1 $i \leftarrow 1, G_i \leftarrow G, \eta_i \leftarrow \eta, \rho_b \leftarrow 1 - (1 - 1/b)^b, \theta_0 \leftarrow 0, S \leftarrow \emptyset$;
2 Allocate and initialize arrays mrr, $offset$, $flag$, and cnt on GPU;
3 **while** $\eta_i > 0$ **do**
4 Get θ_i and $\theta_{\max}$;
5 **if** $\theta_{i-1} > 0$ **then**
6 `ParallelUpdate`(G_i, cnt, mrr, $offset$, θ_{i-1}, $flag$);
7 $S_i \leftarrow$ Kernel `Select`(b, cnt, mrr, $offset$, θ_{i-1}, $flag$);
8 **if** $S_i \neq \emptyset$ **then** { $\theta_i \leftarrow \theta_{i-1}$; **goto** Line 15; }
9 **else** $\theta_i \leftarrow 2\theta_{i-1}$;
10 **while** $\theta_i < \theta_{\max}$ **do**
11 `ParallelGen`(G_i, cnt, mrr, $offset$, θ_{i-1}, θ_i);
12 $S_i \leftarrow$ Kernel `Select`(b, cnt, mrr, $offset$, θ_i, $flag$);
13 **if** $S_i \neq \emptyset$ **then break**;
14 $\theta_{i-1} \leftarrow \theta_i$; $\theta_i \leftarrow 2\theta_i$;
15 Diffuse from S_i, and obtain the activated nodes A_i;
16 Kernel `Discover`(cnt, mrr, $offset$, $flag$, θ_i, A_i);
17 $S \leftarrow S \cup S_i$; $G_{i+1} \leftarrow G_i \backslash A_i$; $\eta_{i+1} \leftarrow \eta_i - |A_i|$; $i \leftarrow i + 1$;

In Algorithm 6, given a node set A (equal to $\{u\}$ in the `Select` kernel), `Discover` identifies all mRR-sets covered by A and updates the coverage counts of the nodes in those mRR-sets. In Section 4.4, we have also used `Discover` to identify the mRR-sets covered, i.e., polluted by the activated nodes in A_{i-1}. In each thread block, we first obtain the number of mRR-sets r in its segment (Line 1), then iterate over them with r_1 from 0 to $r-1$ (Lines 2–16). For each mRR-set indexed by r_1, we first obtain its start and end positions from the $offset$ array and its length using the `GmRR` layout in Section 4.2 (Lines 3–5). If its flag is false (Line 6), meaning it is not yet covered, all threads in the block (Line 7) scan this mRR-set in parallel to check whether any node is in A (Line 8); once such a node is found, they set the flag to true and terminate the scan (Lines 9–10). All threads in the block then collaboratively scan the nodes in this mRR-set and decrement coverage counts in cnt (Lines 12–14). The loop repeats until all mRR-sets in the segment have been processed.

4.6 The Complete `GAAS` Algorithm and Analysis

Algorithm 7 describes `GAAS`. Given G, node costs $c(\cdot)$, target η, estimation error ϵ, and batch size b (seeds per round), we initialize $i = 1$, G_i, η_i, S, and `GmRR` with arrays mrr, $offset$, $flag$, and cnt.

While $\eta_i > 0$, we first compute the required number of mRR-sets θ_i for round i and its upper bound $\theta_{\max}$. If mRR-sets from the previous round exist (i.e., $\theta_{i-1} > 0$), we update and reuse them for seed selection: we call `ParallelUpdate` to update all existing mRR-sets in parallel (Line 6), and then invoke `Select` to obtain b seeds S_i (Line 7). If S_i is non-empty (Line 8), the current mRR-sets suffice and we proceed directly to Line 15 without further generation; otherwise, we set $\theta_i \leftarrow 2\theta_{i-1}$ (Line 9) and continue. In this loop, we progressively enlarge the mRR-set collection until either $S_i \neq \emptyset$ or θ_i reaches $\theta_{\max}$. In each iteration, we invoke `ParallelGen` to get additional mRR-sets (Line 11), then run `Select` on the enlarged collection (Line 12); if S_i remains empty, we double θ_i and repeat.

Once a non-empty batch S_i is obtained, we simulate diffusion from S_i to obtain the activated nodes A_i (Line 15). We call `Discover` (Line 16) to mark polluted mRR-sets in $flag$ for the next round and to decrement cnt so their contributions are removed. Finally, we update $S \leftarrow S \cup S_i$, construct G_{i+1}, derive η_{i+1} (Line 17), and proceed to the next round until the overall influence target is met.

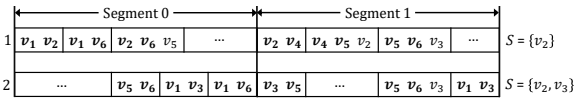


Figure 7: A running example of GAAS from round $i = 1$ to $i = 2$.

In practice, the required number of mRR-sets usually decreases over rounds, so we periodically clear all segments and regenerate mRR-sets for a round when the remaining influence threshold is halved compared with previous regeneration round. This occasional regeneration discards excess mRR-sets, and its cost is amortized by the computational savings from updates in most rounds.

Example 4.5. Figure 7 shows, from input graph G in Figure 2(a), how mRR-sets are stored, updated, and used in GmRR across rounds. GmRR has two segments with size $\ell = 10$. In round $i = 1$, ParallelGen generates six mRR-sets using two thread blocks on $G_1 = G$, with segment 0 and 1 each storing three mRR-sets. The roots of the mRR-sets are in bold. In round 1, Select chooses $S = \{v_2\}$ as seed, which activates v_2 and v_4 under the realization in Figure 2(b). In round $i = 2$, ParallelUpdate updates the mRR-sets from round $i = 1$ on G_2 in Figure 5(a), using the update techniques in Section 4.4. For example, the mRR-set containing $\{v_5, v_6, v_3\}$ in round $i = 2$ is inherited from round $i = 1$. These updated mRR-sets are then used by Select to choose v_3 as seed, resulting in seed set $S = \{v_2, v_3\}$. Thus, we update and reuse mRR-sets across rounds to reduce overhead.

The theoretical guarantee of GAAS is given in Theorem 4.6, where π^o is the optimal policy that incurs the same per-round seed cost as GAAS. The approximation factor scales linearly with $\ln \eta$, improving the guarantees of [14, 39] by a factor of $\ln \eta$.

Theorem 4.6. *The cost of* GAAS *reaching η approximates that of π^o in the form:* $\mathbb{E}[c(\text{GAAS})] \leq \frac{2 \ln \eta}{(1-\epsilon)(1-1/e)\rho_b} \mathbb{E}[c(\pi^o)] + b + 1$.

Proof sketch. Following the idea of [11], we consider a dual problem of AMCSS and analyze the approximation ratio of GAAS for the truncated influence achieved under an arbitrary budget B. By setting B in terms of $c(\pi^o)$, we show that the truncated influence is at least η with probability $1 - 1/\eta$. In the remaining case, which occurs with probability $1/\eta$, the cost to reach η is at most η. This yields the stated bound on the expected cost. □

We show that the time of updating and reusing θ_{i-1} mRR-sets in round i is asymptotically smaller than that of generating them from scratch, and derive the space complexity of GAAS.

Theorem 4.7. *Let α be the ratio of the estimated influence of A_{i-1} to the optimal influence under the same cost $c(S_{i-1})$. Updating θ_{i-1} mRR-sets in round i takes $O\left(\frac{\alpha b(m_i+n_i)\ln n_{i-1}}{\epsilon^2} \cdot \frac{\mathbb{E}[\hat{\Gamma}(A_{i-1}|\mathcal{S}_{i-1})]}{\eta_i}\right)$ time, where $\mathbb{E}[\hat{\Gamma}(A_{i-1}|\mathcal{S}_{i-1})]/\eta_i < 1$, lower than generating the same number of mRR-sets from scratch that takes $O\left(\frac{\alpha b(m_i+n_i)\ln n_i}{\epsilon^2}\right)$ time.*

Proof sketch. Updating clean mRR-sets only requires new reverse BFS from new roots, which is equivalent to generating partial mRR-sets, while updating polluted mRR-sets is nearly as costly as generating new ones. Thus, we compute the expected numbers of clean and polluted mRR-sets, multiply by their respective update costs, and sum to obtain the total time complexity in round i. □

Note that GPU parallelism is a practical factor and is thus omitted from the time complexity analysis, but evaluated in experiments.

Theorem 4.8. GAAS *has expected space $O(m + \max_i \frac{\alpha b n_i \ln n_i}{\epsilon^2})$.*

Table 1: Datasets information. ($K = 10^3, M = 10^6$)

Name	n	m	Avg. deg	Type
DBLP	317K	1.05M	6.62	undirected
YouTube	1.13M	2.99M	5.27	undirected
Flickr	2.30M	33.1M	14.4	directed
LiveJournal	4.85M	68.5M	14.1	directed

Proof sketch. In round i, the dominant cost is storing θ_i mRR-sets. Hence, the expected space is $O(\theta_i \cdot \mathbb{E}[|R|])$, where $|R|$ denotes the size of a random mRR-set in round i. Taking the maximum over rounds and adding graph storage gives the space complexity. □

Discussion. When GPU memory becomes a bottleneck, GAAS can be extended to multi-GPUs. The main memory overhead comes from storing mRR-sets, besides the graph G. The GmRR uses a segment-based layout, where each GPU thread block operates on its own dedicated segment. This design can be naturally extended to multi-GPUs by maintaining a GmRR instance per GPU to generate and update mRR-sets locally. For seed selection, only statistics derived from the GmRR instances need to be synchronized, rather than the mRR-sets. If G is small enough, it can be replicated across GPUs to reduce communication. If G is large, it can be partitioned and distributed across GPUs, and reverse BFS for mRR-sets is performed distributively. We leave a detailed investigation to future work.

5 EXPERIMENTS

We evaluate GAAS and baselines on a server with an Intel Xeon Gold 6226R 2.90GHz CPU with 16 physical cores, 384 GB RAM, and an NVIDIA RTX 3090 GPU with 24 GB of memory.

5.1 Experimental Setup

Datasets. We adopt four datasets used in previous work [12, 19, 22, 39, 46]. The statistics are in Table 1. These datasets represent diverse social networks from real world and are available at [27]. The large dataset, LiveJournal, contains over 4.85 million nodes and 68.5 million edges, which is the largest network evaluated in [39].
Baselines. ASTI originally handles uniform seed costs and runs single-threaded on CPU [39]. We extend it to non-uniform costs by selecting seeds with the largest influence-to-cost ratios in each round. We also implement a CPU-parallel version, PASTI-CPU, parallelizing all applicable computations in ASTI using OpenMP [8] across all CPU cores. Moreover, we include a GPU-parallel ASTI version, PASTI-GPU, which adapts the GPU-based techniques of [37] (designed for single-root RR-set generation in Influence Maximization). PASTI-GPU stores mRR-sets contiguously in global GPU memory. Finally, we compare with SCORE, a non-adaptive MCSS method providing a probabilistic guarantee on η in any realization. We set SCORE with confidence ($p = 0.99$) to align with the stopping criterion of adaptive methods reaching η with probability 1.0.
Parameters. We use consistent parameters across all baselines and GAAS. Following [12, 19, 22, 39], we set the mRR-set estimation error $\epsilon = 0.5$ and failure probability $\delta = 1/n_i$ per round. The diffusion probability of any edge (u, v) is $p(u, v) = 1/|N_{\text{in}}(v)|$, and the cost of node u is $c(u) = c_0 + 0.01 \cdot |N_{\text{out}}(u)|$ with $c_0 = 0.01$ to avoid zero-cost nodes. We evaluate all methods under the IC and LT models, averaging results over 10 realizations. Under IC, the batch size b is 4 on DBLP and 16 on larger datasets, ensuring that the single-threaded ASTI completes within two days. The threshold η ranges from 1K to 5K on DBLP and YouTube, 2K to 10K on Flickr, and 5K to 25K on LiveJournal, reflecting their different scales. Under LT, we

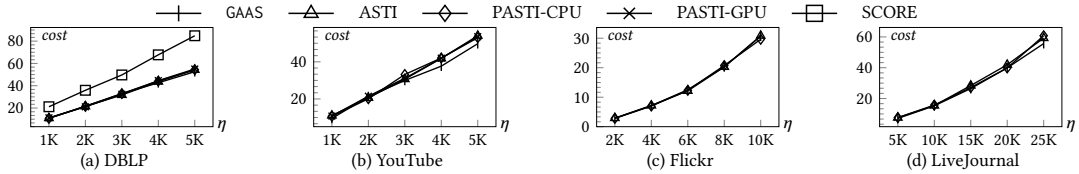


Figure 8: Seed cost of algorithms under the IC model.

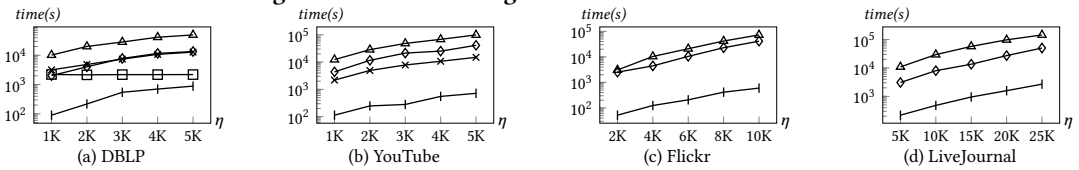


Figure 9: Running time in seconds of algorithms under the IC model.

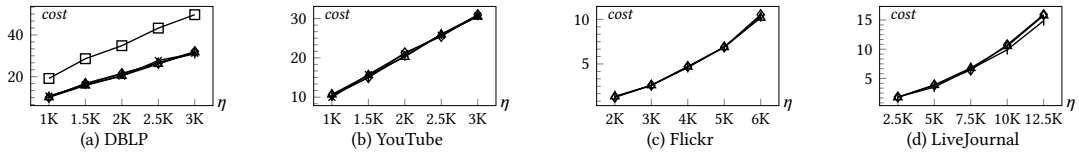


Figure 10: Seed cost of algorithms under the LT model.

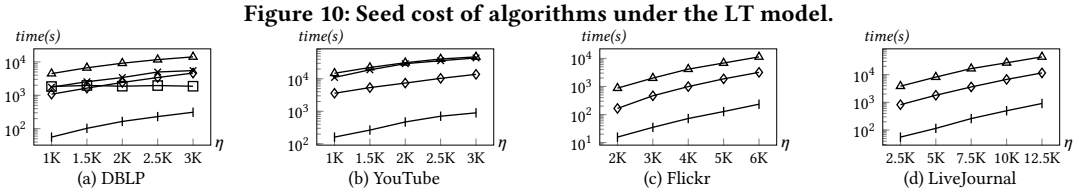


Figure 11: Running time in seconds of algorithms under the LT model.

fix $b=4$, with η ranging from 1K to 3K on DBLP and YouTube, 2K to 6K on Flickr, and 2.5K to 12.5K on LiveJournal. In GAAS, we set the number of mRR-set update groups to $\beta=8$ for load balancing, as determined by ablation study in Section 5.4. We configure GAAS and parallel methods to utilize all available CPU and GPU resources.

5.2 Seed Cost and Efficiency under the IC model

AMCSS aims to minimize seed cost while reaching the target influence η. Thus, efficiency is comparable only when methods reliably attain η with low seed cost. By the theoretical guarantees of GAAS and the adaptive baselines, all adaptive methods always attain η (100% *attainment*). We therefore omit attainment plots (all equal to 1.0) and focus on comparing the *seed cost* and *running time*.

Seed Cost. Figure 8 reports the seed cost to reach various η under the IC model. All adaptive methods yield similar low costs and require substantially less cost than the non-adaptive SCORE on DBLP. Because GAAS and the adaptive ASTI baselines leverage the mRR-set estimation framework, they have similar estimation accuracy and thus comparable seed cost. Nonetheless, GAAS achieves slightly lower cost, especially on YouTube and LiveJournal. This is because GAAS deliberately retains all previous mRR-sets during adaptive updates, although the number of required mRR-sets typically decreases over rounds, allowing it to exploit more mRR-sets than regeneration-based methods and make better seeding decisions. The higher cost of SCORE arises from its non-adaptive nature: it cannot use intermediate results and thus selects more nodes to reach η with the required probability. SCORE runs out of memory (OOM) on the other three larger datasets, since it depends on graph sketches, and the sketch size grows rapidly as the graph scales.

Running Time. Given the minimal and similar seed costs, we compare efficiency. Figure 9 reports the running time (y-axis in log scale). GAAS consistently outperforms all baselines, often by more than an order of magnitude. For all methods, running time increases with larger η since higher thresholds require more rounds. Compared with the strongest GPU-based baseline PASTI-GPU, GAAS achieves speedups of up to 33.9× and 24.2× on DBLP and YouTube, respectively. These gains stem from our GmRR structure, which enables in-segment mRR-set generation and update directly on GPU. In contrast, PASTI-GPU runs OOM on the larger Flickr and LiveJournal for two main reasons. First, it relies heavily on dynamic memory allocation, which leads to severe memory fragmentation. Second, it temporarily buffers mRR-sets being generated in global memory, effectively doubling peak memory usage in the worst case. GAAS avoids these issues by combining GmRR with carefully designed parallel mRR-set generation and update kernels, which use global memory and thread blocks more efficiently. On the two largest datasets, GAAS achieves substantial speedups over the strongest CPU baseline PASTI-CPU. On Flickr, GAAS runs up to 68.9× faster than PASTI-CPU when η = 10K, and on LiveJournal it attains more than 10× speedup. For SCORE on DBLP, its running time in Figure 9(a) is less informative because it incurs significantly higher seed cost in Figure 8(a). Even so, GAAS remains much faster than SCORE with lower seed cost. SCORE runs OOM on other datasets.

5.3 Seed Cost and Efficiency under the LT model

Under the LT model, GAAS is also faster than existing methods, often by an order of magnitude, while maintaining low seed cost.

Seed Cost. Figure 10 reports the seed cost under the LT model. The trends are consistent with those in Figure 8. For each dataset and influence threshold η, all adaptive methods incur similar costs, whereas the non-adaptive SCORE requires substantially higher cost on DBLP and runs out of memory on larger datasets. Overall,

GAAS achieves comparable or lower cost than all baselines, with its advantage increasing for larger η, where additional rounds allow it to reuse and update existing mRR-sets instead of regenerating them, leading to more accurate truncated influence estimation.

Time Efficiency. Figure 11 reports the running time under the LT model. The results show that GAAS consistently outperforms all baseline methods across all evaluated datasets. Note that SCORE and PASTI-GPU run OOM on the two largest datasets and thus do not produce results there. Across all influence targets η where the baselines succeed, GAAS achieves more than 10× speedup over the strongest competitor. In particular, GAAS attains speedups of up to 19.2× on DBLP, 22.3× on YouTube, 14.6× on Flickr, and 15.7× on LiveJournal compared to the fastest baseline. These substantial performance gains arise from our carefully designed GPU optimizations, including independent inter-block mRR-set generation and the efficient selective reuse of clean mRR-sets, which together minimize redundant computation and fully exploit GPU parallelism.

5.4 Experimental Analysis

Time Breakdown and the Effect of `ParallelUpdate`. A key design in GAAS is to update and reuse mRR-sets from previous rounds by the GPU kernel `ParallelUpdate` instead of regenerating. To quantify the efficiency benefit, we compare GAAS with a variant GAAS_{gen} that regenerates mRR-sets from scratch in every round on GPU. For each method, we decompose the running time into three components: mRR-set generation, mRR-set update, and others (including seed selection). In Figure 12, we report the time breakdown of GAAS and GAAS_{gen} under the IC model. The bar with ⋆ represents GAAS, while the bar without ⋆ represents GAAS_{gen}. It shows that GAAS with `ParallelUpdate` significantly reduces time spent on mRR-set generation: in GAAS_{gen}, mRR-set generation dominates the runtime, whereas in GAAS it becomes negligible and the total running time is much lower. This confirms that GAAS effectively reuses existing mRR-sets by updating mRR-sets in parallel on the GPU. The results under the LT model are similar and deferred to the technical report [2]. GAAS outperforms GAAS_{gen} in efficiency, confirming the power of our proposed mRR-set update kernel.

GPU Profiling. We use NVIDIA Nsight Compute [1] to quantify the efficiency of our GPU method via standard GPU performance metrics. Figure 13 reports the GPU occupancy level, instructions executed per cycle (IPC), and streaming multiprocessor (SM) active percentage—where higher values indicate better performance—as well as the number of atomic operations, where lower values are preferred for efficiency, comparing GAAS with PASTI-GPU on DBLP and YouTube under the IC model. Clearly, GAAS achieves better GPU occupancy, higher IPC, and higher SM throughput while requiring fewer atomic operations than PASTI-GPU. For example, GAAS attains 1.8× and 2.1× higher occupancy than PASTI-GPU. These demonstrate that GAAS utilizes GPU resources more efficiently, owing to our algorithmic designs and performance optimizations.

GPU Memory Usage. We compare GAAS with PASTI-GPU in Figure 14 under IC and LT models. It shows that GAAS uses less GPU memory than PASTI-GPU on DBLP and Youtube, while PASTI-GPU runs out of memory on the larger Flickr and LiveJournal, with reasons discussed in Section 5.2. GAAS uses the proposed GmRR structure

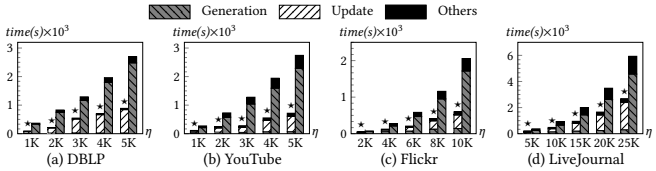


Figure 12: Runtime breakdown under the IC model; for each η, the left bar with ⋆ is GAAS and the right bar is GAAS_{gen}.

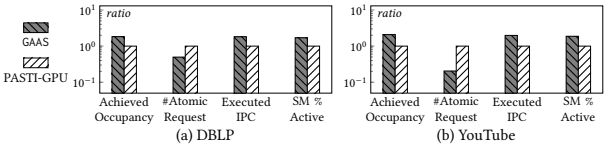


Figure 13: GPU profiling analysis relative to PASTI-GPU.

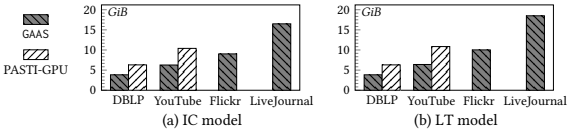


Figure 14: GPU memory comparison with PASTI-GPU.

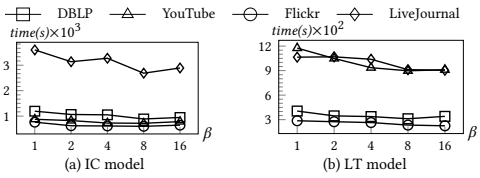


Figure 15: Vary β under largest tested η across datasets.

with carefully designed parallel mRR-set generation and update kernels, resulting in efficient memory usage on GPU.

Vary β for load balancing. In the `ParallelUpdate` kernel, we employ a load-balancing strategy controlled by a parameter β, which specifies the number of groups used for mRR-set updates on the GPU. We vary $\beta \in \{1, 2, 4, 8, 16\}$ to study its impact under both the IC and LT models. When $\beta = 1$, no load balancing is applied. In Figure 15, observe that the running time generally decreases as β increases from 1 to 8. However, when β increases from 8 to 16, the running time starts to increase, because the overhead of finer-grained grouping and block-level synchronization outweighs the benefits. Therefore, we set $\beta = 8$.

6 CONCLUSION

We present a GPU-accelerated AMCSS method GAAS by integrating novel algorithmic ideas with GPU-aware system design. We design a GPU-tailored mRR-set structure GmRR, and build a `ParallelGen` kernel for mRR-set generation, a `ParallelUpdate` kernel that reuses and updates previous mRR-sets via theoretically sound rules and circular in-segment updates with load balancing, and a `Select` kernel for parallel seed selection. Experiments on real-world graphs show that GAAS is over an order of magnitude faster than the baselines, while achieving comparable or lower seed cost. We plan to extend GAAS to distributed or multi-GPU settings as future work.

ACKNOWLEDGMENTS

This work is supported by grants from the Research Grants Council of Hong Kong SAR, China (No. PolyU 15205224), ByteDance (P0063921), and the Postdoc Matching Fund (PolyU, No. 1-W33U).

REFERENCES

[1] 2023. INSIGHT COMPUTE COMMAND LINE INTERFACE. https://docs.nvidia.com/nsight-compute/pdf/NsightComputeCli.pdf.

[2] 2026. Technical Report. https://github.com/gongyguo/GAAS/blob/main/GAAS_Technical_Report.pdf.

[3] 360iResearch. 2026. Social Media Advertising Market – Global Forecast 2026-2032. (2026), 1–187.

[4] Song Bian, Qintian Guo, Sibo Wang, and Jeffrey Xu Yu. 2020. Efficient Algorithms for Budgeted Influence Maximization on Massive Social Networks. *VLDB* 13, 9 (2020), 1498–1510.

[5] Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.

[6] Bogdan Cautis, Silviu Maniu, and Nikolaos Tziortziotis. 2019. Adaptive influence maximization. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3185–3186.

[7] 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.

[8] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[9] Gianlorenzo D'Angelo, Debashmita Poddar, and Cosimo Vinci. 2021. Improved Approximation Factor for Adaptive Influence Maximization via Simple Greedy Strategies. In *48th International Colloquium on Automata, Languages, and Programming*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 59–1.

[10] Datareportal. 2025. Global social media statistics. https://datareportal.com/social-media-users.

[11] Hossein Esfandiari, Amin Karbasi, and Vahab S. Mirrokni. 2021. Adaptivity in Adaptive Submodularity. In *COLT*, Vol. 134. 1823–1846.

[12] Chen Feng, Xingguang Chen, Qintian Guo, Fangyuan Zhang, and Sibo Wang. 2024. Efficient Approximation Algorithms for Minimum Cost Seed Selection with Probabilistic Coverage Guarantee. *Proc. ACM Manag. Data* 2, 4 (2024), 1–26.

[13] Daniel Golovin and Andreas Krause. 2011. Adaptive submodularity: Theory and applications in active learning and stochastic optimization. *J. Artif. Intell. Res.* 42 (2011), 427–486.

[14] Daniel Golovin and Andreas Krause. 2017. Adaptive Submodularity: Theory and Applications in Active Learning and Stochastic Optimization. arXiv:1003.3967 [cs.LG] https://arxiv.org/abs/1003.3967

[15] Amit Goyal, Francesco Bonchi, Laks VS Lakshmanan, and Suresh Venkatasubramanian. 2013. On minimizing budget and time in influence propagation over social networks. *Social Netw. Anal. Mining* 3 (2013), 179–192.

[16] Qintian Guo, Chen Feng, Fangyuan Zhang, and Sibo Wang. 2023. Efficient Algorithm for Budgeted Adaptive Influence Maximization: An Incremental RR-set Update Approach. *Proc. ACM Manag. Data* 1, 3 (2023), 1–26.

[17] Qintian Guo, Sibo Wang, Zhewei Wei, and Ming Chen. 2020. Influence maximization revisited: Efficient reverse reachable set generation with bound tightened. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2167–2181.

[18] Kai Han, Yuntian He, Keke Huang, Xiaokui Xiao, Shaojie Tang, Jingxin Xu, and Liusheng Huang. 2019. Best bang for the buck: Cost-effective seed selection for online social networks. 32, 12 (2019), 2297–2309.

[19] Kai Han, Keke Huang, Xiaokui Xiao, Jing Tang, Aixin Sun, and Xueyan Tang. 2018. Efficient algorithms for adaptive influence maximization. *VLDB* 11, 9 (2018), 1029–1040.

[20] Wenjing Hong, Chao Qian, and Ke Tang. 2020. Efficient minimum cost seed selection with theoretical guarantees for competitive influence maximization. *IEEE transactions on cybernetics* 51, 12 (2020), 6091–6104.

[21] Lin Hu, Yinnian Lin, Lei Zou, and M Tamer Özsu. 2025. A graph pattern mining framework for large graphs on GPU. *The VLDB Journal* 34, 1 (2025), 6.

[22] Keke Huang, Jing Tang, Kai Han, Xiaokui Xiao, Wei Chen, Aixin Sun, Xueyan Tang, and Andrew Lim. 2020. Efficient approximation algorithms for adaptive influence maximization. *The VLDB Journal* 29, 6 (2020), 1385–1406.

[23] Keke Huang, Sibo Wang, Glenn Bevilacqua, Xiaokui Xiao, and Laks VS Lakshmanan. 2017. Revisiting the stop-and-stare algorithms for influence maximization. (2017).

[24] Yiqian Huang, Shiqi Zhang, Laks V. S. Lakshmanan, Wenqing Lin, Xiaokui Xiao, and Bo Tang. 2024. Efficient and Effective Algorithms for A Family of Influence Maximization Problems with A Matroid Constraint. *Proc. VLDB Endow.* 18, 2 (2024), 117–129.

[25] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *KDD*. 137–146.

[26] Andreas Krause and Carlos Guestrin. 2005. *A note on the budgeted maximization of submodular functions.* Carnegie Mellon University. Center for Automated Learning and Discovery .

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Yinnian Lin, Lei Zou, and Xunbin Su. 2024. Towards Sufficient GPU-accelerated Dynamic Graph Management: Survey and Experiment. *Proc. VLDB Endow.* 18, 3 (2024), 599–612.

[29] Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *SIGMOD Conference*. ACM, 403–416.

[30] Cheng Long and Raymond Chi-Wing Wong. 2011. Minimizing seed set for viral marketing. In *2011 IEEE 11th International Conference on Data Mining*. IEEE, 427–436.

[31] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. 2020. cuRipples: Influence maximization on multi-GPU systems. In *Proceedings of the 34th ACM international conference on supercomputing*. 1–11.

[32] 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*. 1–12.

[33] Hung T Nguyen, Tri P Nguyen, NhatHai Phan, and Thang N Dinh. 2017. Importance sketching of influence dynamics in billion-scale networks. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 337–346.

[34] Hung T Nguyen, My T Thai, and Thang N Dinh. 2016. Stop-and-stare: Optimal sampling algorithms for viral marketing in billion-scale networks. In *Proceedings of the 2016 international conference on management of data*. 695–710.

[35] Binghui Peng and Wei Chen. 2019. Adaptive influence maximization with myopic feedback. *Advances in Neural Information Processing Systems* 32 (2019).

[36] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3204–3216.

[37] Soheil Shahrouz, Saber Salehkaleybar, and Matin Hashemi. 2021. gim: Gpu accelerated ris-based influence maximization algorithm. *IEEE Transactions on Parallel and Distributed Systems* 32, 10 (2021), 2386–2399.

[38] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Realtime Top-k Personalized PageRank over Large Graphs on GPUs. *Proc. VLDB Endow.* 13, 1 (2019), 15–28.

[39] Jing Tang, Keke Huang, Xiaokui Xiao, Laks VS Lakshmanan, Xueyan Tang, Aixin Sun, and Andrew Lim. 2019. Efficient approximation algorithms for adaptive seed minimization. In *SIGMOD*. 1096–1113.

[40] Jing Tang, Xueyan Tang, Xiaokui Xiao, and Junsong Yuan. 2018. Online Processing Algorithms for Influence Maximization. In *SIGMOD*. 991–1005.

[41] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence maximization in near-linear time: A martingale approach. In *SIGMOD*. 1539–1554.

[42] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: near-optimal time complexity meets practical efficiency. In *SIGMOD*. 75–86.

[43] Guangmo Tong, Weili Wu, Shaojie Tang, and Ding-Zhu Du. 2016. Adaptive influence maximization in dynamic social networks. *IEEE/ACM Transactions on Networking* 25, 1 (2016), 112–125.

[44] Sharan Vaswani and Laks VS Lakshmanan. 2016. Adaptive influence maximization in social networks: Why commit when you can adapt? *arXiv preprint arXiv:1604.08171* (2016).

[45] Jinghao Wang, Yanping Wu, Xiaoyang Wang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. Efficient Influence Minimization via Node Blocking. *Proc. VLDB Endow.* 17, 10 (2024), 2501–2513.

[46] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. 2023. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. *VLDB* 17, 3 (2023), 400–413.

[47] Yexin Wang, Zhi Yang, Junqi Liu, Wentao Zhang, and Bin Cui. 2023. Scapin: Scalable Graph Structure Perturbation by Augmented Influence Maximization. *Proc. ACM Manag. Data* 1, 2 (2023), 146:1–146:21.

[48] Yifei Xia, Feng Zhang, Qingyu Xu, Mingde Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, Bingsheng He, and Siqi Ma. 2024. GPU-based butterfly counting. *The VLDB Journal* 33, 5 (2024), 1543–1567.

[49] Qingyu Xu, Feng Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, and Bingsheng He. 2022. Efficient Load-Balanced Butterfly Counting on GPU. *Proc. VLDB Endow.* 15, 11 (2022), 2450–2462.

[50] Yichao Yuan, Haojie Ye, Sanketh Vedula, Wynn Kaza, and Nishil Talati. 2023. Everest: GPU-Accelerated System For Mining Temporal Motifs. *Proc. VLDB Endow.* 17, 2 (2023), 162–174.

[51] Peng Zhang, Wei Chen, Xiaoming Sun, Yajun Wang, and Jialin Zhang. 2014. Minimizing seed set selection with probabilistic coverage guarantee in a social network. In *KDD*. 1306–1315.

[52] Yapu Zhang, Shengminjie Chen, Wenqing Xu, and Zhenning Zhang. 2022. Adaptive influence maximization under fixed observation time-step. *Theoretical Computer Science* 928 (2022), 104–114.