Gem: Scalable Monotonic Graph Processing Beyond Billion-Scale on a Single Machine

CHENGYING HUAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENGYI YANG, University of New South Wales, Australia
HAOSHEN YANG, Rutgers, The State University of New Jersey, United States
SHAONAN MA, Qiyuan Lab, China
RONG GU*, State Key Laboratory for Novel Software Technology, Nanjing University, China
FANG XI, Qiyuan Lab, China
YONGCHAO LIU, Ant Group, China
GUIHAI CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHEN TIAN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Monotonic graph algorithms, such as shortest path, BFS, and reachability, are fundamental to graph analytics and are widely used across domains. Recent systems employ pruning techniques to accelerate the processing of these algorithms. However, state-of-the-art monotonic graph engines are restricted to in-memory execution and cannot scale to graphs that exceed main memory capacity. In contrast, existing out-of-core graph engines are designed for general-purpose workloads and lack effective pruning mechanisms tailored to monotonic graph algorithms. To bridge this gap, we present Gem, an out-of-core graph engine designed for monotonic graph algorithms. Gem introduces a PageRank-based graph sketch that captures key topological features in-memory with minimal preprocessing overhead. Building on this sketch, we propose a novel graph abstraction that enables the direct derivation of tight bounds for monotonic graph algorithms, supporting effective pruning at both the vertex and partition levels. Comprehensive evaluations on six real-world datasets, including the 42.5-billion-edge ClueWeb graph, show that Gem significantly outperforms existing systems. It achieves up to **135.40**× speedup over GridGraph and **12.58**× over Wonderland in out-of-core settings, and also delivers substantial improvements in other modes: up to **10.41**× over RisGraph in memory and **20.64**× over CGgraph out-of-GPU memory.

CCS Concepts: • **Information systems → Hierarchical data models**.

Additional Key Words and Phrases: Graph Processing; Graph Abstraction; Out-of-Core;

*Rong Gu is the corresponding author of this paper.

Authors' Contact Information: Chengying Huan, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, huanchengying@nju.edu.cn; Zhengyi Yang, University of New South Wales, Sydney, Australia, zhengyi.yang@unsw.edu.au; Haoshen Yang, Rutgers, The State University of New Jersey, New Brunswick, United States, hy482@scarletmail.rutgers.edu; Shaonan Ma, Qiyuan Lab, Beijing, China, mashaonan@qiyuanlab.com; Rong Gu, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, gurong@nju.edu.cn; Fang Xi, Qiyuan Lab, Beijing, China, xifang@qiyuanlab.com; Yongchao Liu, Ant Group, Hang Zhou, China, yongchao.ly@antgroup.com; Guihai Chen, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, gchen@nju.edu.cn; Chen Tian, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, tianchen@nju.edu.cn.

ACM 2836-6573/2025/12-ART330
https://doi.org/10.1145/3769795

Type	Name	Pruning	Sketch	Out-of-Core
Monotonic Graph Engine	RisGraph [33]	×	×	×
	Tripoline [47]	Δ	×	×
	SGraph [22]	Δ	×	×
General Purpose Engine	GridGraph [111]	×	×	✓
	Core Graph [45]	×	×	✓
	CGgraph [31]	×	×	✓
	Wonderland [103]	×	Δ	✓
Our System	**Gem**	✓	✓	✓

Table 1. Comparison of pruning, sketch, and out-of-core execution capabilities across representative systems and Gem. Here, × indicates the absence of a capability, Δ denotes a weak capability, and ✓ represents a strong capability.

ACM Reference Format:
Chengying Huan, Zhengyi Yang, Haoshen Yang, Shaonan Ma, Rong Gu, Fang Xi, Yongchao Liu, Guihai Chen, and Chen Tian. 2025. Gem: Scalable Monotonic Graph Processing Beyond Billion-Scale on a Single Machine. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 330 (December 2025), 30 pages. https://doi.org/10.1145/3769795

1 Introduction

Graphs are a natural representation for numerous real-world applications [20, 21, 23, 24, 26, 39, 42, 66, 68, 96, 109], such as social networks [7, 17, 43, 81, 82, 89], e-commerce [41, 74, 77], and deep learning [12, 40, 71, 78, 80, 86, 88]. In graph analytics, many widely used algorithms exhibit a key property known as *monotonicity*, where intermediate results are updated in a monotonically convergent manner. Notable algorithms include shortest path, breadth-first search (BFS), reachability, and connected component detection. For instance, in a shortest-path query from src to $dest$, the intermediate distance $dist[src \rightarrow dest]$ decreases monotonically until it converges to the final shortest distance $DIST(src \rightarrow dest)$. This class of algorithms are known as *monotonic graph algorithms* [33, 47]. Given the broad applicability of monotonic graph algorithms, they have attracted considerable attention in recent years. Several systems have been specifically designed to support the efficient processing of monotonic graph algorithms, including SGraph [22], Tripoline [47], MergeGraph [28], and RisGraph [33]. However, real-world monotonic graph applications often operate on massive graphs beyond billion-scale that can reach sizes of hundreds of gigabytes. To scale these applications to graphs that exceed a machine's memory capacity, a typical approach is to utilize external memory (i.e., disk) for out-of-core graph processing [49, 51, 53, 58, 59, 72, 83, 85, 92, 99, 105]. Compared to alternatives such as distributed graph processing, out-of-core processing on a single machine is usually more cost-effective and demands fewer hardware resources [67, 69].

We observe that none of the existing monotonic graph engines support out-of-core execution, which limits their scalability on large graphs. Pruning [22, 33, 47, 84] is a common technique used to avoid unnecessary computations by identifying and skipping parts of the graph that cannot affect the final result. It is widely adopted by state-of-the-art monotonic graph engines such as Tripoline [47] and SGraph [33]. These systems estimate a range of possible values, referred to as lower and upper bounds, for certain metrics between vertices, and discard updates that fall outside this range. However, such engines are designed for in-memory execution only and cannot scale to graphs that exceed memory capacity. On the other hand, out-of-core graph engines like GridGraph [111], Core Graph [45], CGgraph [31], and Wonderland [103] target general-purpose workloads at scale. Some of these, such as Wonderland, use graph sketches to cache key topological structures in memory, accelerating convergence and reducing disk I/O. Nevertheless, they lack pruning mechanisms tailored to monotonic graph algorithms, limiting their computational efficiency on such tasks.

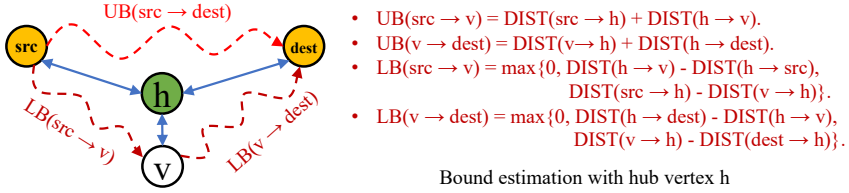


Fig. 1. Bounds and pruning example. The blue arrows are precalculated exact distance (i.e., *DIST*) based on the hub vertex *h*.

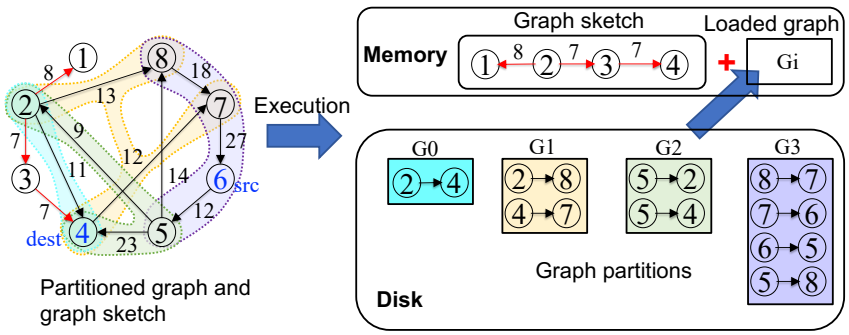


Fig. 2. Using graph sketch for single machine graph processing in Wonderland [103].

Table 1 highlights the gap: monotonic graph engines use pruning but require in-memory execution, while out-of-core systems handle large graphs but do not support pruning for monotonic algorithms. Our proposed system Gem bridges this gap by combining bound-based pruning with a graph abstraction inspired by graph sketching, enabling efficient out-of-core processing for monotonic workloads.

In summary, designing an efficient out-of-core engine for monotonic graph processing requires tackling two main challenges: *Ineffective Memory-Constrained Bound Estimation* and *Ineffective Graph Sketches for Monotonic Algorithms*. We first outline these challenges and discuss them in detail in Section 3.

Challenge I: Ineffective Memory-Constrained Bound Estimation. Monotonic graph algorithms often rely on precomputed bounds (e.g., lower/upper distance bounds) to prune unnecessary work. These bounds are typically derived from a small set of hub vertices, where each hub stores distances to all other nodes. While effective in-memory, this approach becomes infeasible out-of-core: each hub consumes $O(|V|)$ space, and only a limited number can fit in memory. As a result, pruning degrades sharply due to loose or missing bounds. Moreover, common bounding strategies such as triangle inequality often yield conservative estimates, further limiting effectiveness. The key problem is how to achieve tight, global bounds without large memory footprints.

Challenge II: Ineffective Graph Sketches. To reduce I/O overhead, out-of-core systems use in-memory graph sketches which compact subgraphs that retain high-impact edges to guide navigation across partitions. These sketches improve convergence and minimize disk access. However, traditional sketches often lack structural coverage and offer no formal guarantees for preserving global graph properties, limiting their effectiveness for navigation and bound estimation for pruning in monotonic algorithms. Enhancing sketch quality with techniques such as high-centrality node selection is computationally expensive and memory-intensive, making it impractical in out-of-core environments. The fundamental challenge lies in designing lightweight sketches that preserve sufficient global connectivity to enable effective propagation and pruning.

Contributions. To address the limitations in existing works, this paper presents Gem (short for **G**raph processing with **E**xternal memory for **M**onotonic algorithms), an efficient monotonic graph

processing engine on a single machine. Specifically, Gem (i) constructs a compact and navigable graph sketch (GS) tailored for large graphs, which effectively captures the graph's underlying topological structure using PageRank, and (ii) builds upon this sketch to construct a novel graph abstraction (GA) that enables the derivation of high-quality (i.e., tight) bounds, thereby minimizing redundant computations and unnecessary data accesses. In our evaluation, Gem achieves speedups of up to **12.58**× and **135.40**× over Wonderland and GridGraph, respectively, under out-of-core settings. Although designed for out-of-core execution, Gem also delivers significant performance improvements across other execution modes. It achieves up to **4.79**× and **10.41**× speedups over SGraph [22] and RisGraph [33] in the in-memory setting, and up to **16.69**× and **20.64**× over Core Graph [45] and CGgraph [31] in the out-of-GPU-memory setting, respectively. The major contributions of Gem are summarized as follows:

- **Novel PageRank-Based Graph Abstraction.** We propose a PageRank-based graph sketch that effectively captures key topological information with significantly better performance than existing methods [45, 103, 112], while maintaining low preprocessing overhead for large-scale graphs. Compared to the graph sketch used in state-of-the-art systems, our navigable graph sketch yields up to a 3.86× speedup. Building on this, we introduce a novel graph abstraction (GA) that incorporates both real edges within the GS and bound-aware virtual edges that connect sketch vertices to abstract vertices (each representing a graph partition).
- **High-Quality Bounds with Two-Level Pruning.** Based on the GA, we design an efficient algorithm that extracts bounds directly. These bounds are tight and more accurately capture distances from the source vertex (*src*) to all other vertices, and from each vertex to the destination (*dest*). While the bound extraction process accounts for less than 13% of the total execution time, it enables speedups of up to two orders of magnitude. Additionally, we introduce novel vertex-level and partition-level pruning techniques to further reduce redundant computation and I/O overhead.
- **Comprehensive Experimental Evaluation.** We conduct an extensive experimental study to evaluate the effectiveness of our GS, bound construction, and pruning strategies across six widely used real-world datasets, including the 317GB ClueWeb graph with 42.5 billion edges, and six graph algorithms, under various configurations. As a result, Gem consistently and significantly outperforms state-of-the-art graph processing systems.

2 Background

2.1 Bounds and Pruning

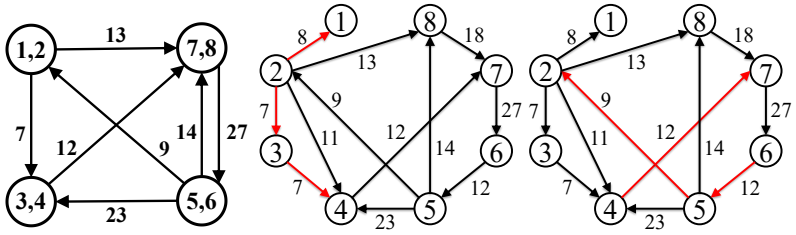
While pruning is a key technique for improving the efficiency of monotonic graph algorithms [22, 33, 47, 84], its effectiveness largely depends on the quality of the bounds. Recent approaches estimate upper bounds (UB) and lower bounds (LB) to skip unnecessary edge updates and reduce redundant computation. The bounds from a source vertex src to a vertex v, denoted $UB(src \rightarrow v)$ and $LB(src \rightarrow v)$, are defined as follows.

Definition 1 (Upper Bound)**.** The upper bound $UB(src \rightarrow v)$ is a close estimate that is greater than or equal to $DIST(src \rightarrow v)$.

Definition 2 (Lower Bound)**.** The lower bound $LB(src \rightarrow v)$ is a close estimate that is less than or equal to $DIST(src \rightarrow v)$.

Using Figure 1 as an example, we illustrate how the bounds can be applied to enable pruning during shortest-path computation.

(i) If $LB(src \rightarrow v) + LB(v \rightarrow dest) > UB(src \rightarrow dest)$, we can assert v will never appear on the shortest path $src \rightarrow dest$.



(a) GraphQ. (b) Wonderland. (c) Gem.

Fig. 3. The graph sketches of running example obtained by (a) GraphQ, (b) Wonderland, and (c) our Gem.

(ii) Otherwise, if $DIST(src \rightarrow v) + LB(v \rightarrow dest) > UB(src \rightarrow dest)$, we can safely remove v from the computation.

(iii) If the previous two cases failed, we derive $DIST(v \rightarrow dest)$. In this case, if $DIST(src \rightarrow v) + DIST(v \rightarrow dest) > UB(src \rightarrow dest)$, we can still remove v from the computation.

To estimate bounds, existing approaches like Triplion [47] apply the triangle inequality to compute both upper and lower bounds.

Definition 3 (Triangle Inequality)**.** Given three vertices src, h, and $dest$, $DIST(src \rightarrow dest) \leq DIST(src \rightarrow h) + DIST(h \rightarrow dest)$.

Triplion uses the triangle inequality to compute the upper bound as $UB(src \rightarrow dest) = DIST(src \rightarrow h) + DIST(h \rightarrow dest)$, based on precomputed distances to hub vertices. SGraph [22] applies a similar approach for lower bounds, using $LB(src \rightarrow v) = DIST(h \rightarrow src) - DIST(v \rightarrow h)$, and similarly for $LB(v \rightarrow dest)$, treating each v as a hub. To improve pruning, multiple hubs are typically selected.

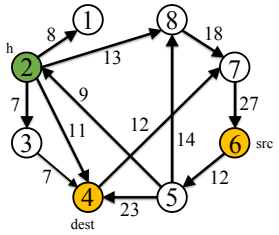
2.2 Graph Sketch

As shown in Table 1, existing out-of-core engines such as GridGraph [111] do not provide a graph sketch with navigation capabilities, which are essential for propagating information across graph partitions and reducing disk I/O. To address this limitation, GraphQ [87] introduced the concept of a graph sketch (**GS**) to guide point-to-point queries in out-of-core settings [29]. The sketch acts as a compact contraction of the original graph, preserving key structural properties while reducing its size. Similar ideas also appear in classical graph partitioning and coarsening techniques, such as those used in METIS [13, 48]. In GraphQ, nearby vertices are grouped into abstract vertices. An abstract edge is created between two abstract vertices A and B if any vertex in A is connected to a vertex in B (see Figure 3a). This allows GraphQ to avoid loading partitions when no abstract connections exist, thus reducing unnecessary I/O. However, GraphQ's sketch lacks real edges and therefore cannot support effective navigation.

Wonderland [103] uses a graph sketch built from real edges, unlike GraphQ, which may introduce abstract edges not present in the original graph. For shortest path queries, Wonderland selects the top-k lightest edges to form the sketch, helping guide graph traversal. In each iteration, a graph partition is loaded from disk and merged with the in-memory sketch to create a working subgraph. Since the sketch is kept in memory across iterations, it participates in every update. This **weight-based** sketch effectively accelerates out-of-core graph processing.

3 Motivation and Challenges

Although prior work has explored bound-based pruning for monotonic graph algorithms and graph sketching for large-scale single-machine processing, two key challenges remain: (1) limited



(a) Running example.

DIST(2→1)	8
DIST(2→2)	0
DIST(2→3)	7
DIST(2→4)	11
DIST(2→5)	62
DIST(2→6)	50
DIST(2→7)	23
DIST(2→8)	13

(b)

DIST(1→2)	∞
DIST(2→2)	0
DIST(3→2)	67
DIST(4→2)	60
DIST(5→2)	9
DIST(6→2)	21
DIST(7→2)	48
DIST(8→2)	66

(c)

LB(6→1)	0
LB(6→2)	21
LB(6→3)	0
LB(6→4)	0
LB(6→5)	12
LB(6→6)	0
LB(6→7)	0
LB(6→8)	0

(d)

LB(6→1)	28
LB(6→2)	21
LB(6→3)	21
LB(6→4)	28
LB(6→5)	12
LB(6→6)	0
LB(6→7)	26
LB(6→8)	12

(e)

Fig. 4. Running example with the distances from vertex $h = 2$ to others and comparison of lower bounds (LBs) in SGraph and Gem: (b) Distance from 2 to v; (c) Distance from v to 2; (d) SGraph LB from src to v; (e) Gem LB from src to v.

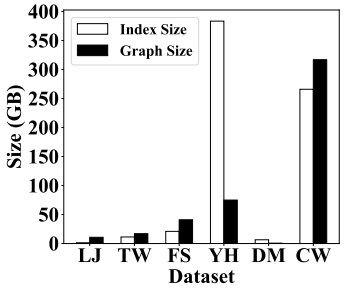


Fig. 5. Index size of SGraph.

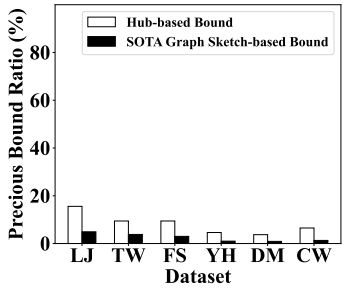


Fig. 6. Bound analysis.

pruning effectiveness due to current loose bound estimations, and (2) insufficient navigational capabilities in existing graph sketches, which restrict efficient query traversal. Below, we elaborate on the challenges outlined in Section 1, as detailed in the following:

Challenge I. Ineffective Memory-Constrained Bound Estimation. As discussed above and shown in Table 1, existing pruning methods for monotonic algorithms such as the hub-based strategy in SGraph [22] are inefficient under memory constraints. These approaches rely on precomputed distances from hub vertices, which consume large amounts of memory and often produce imprecise bounds, limiting their pruning effectiveness on large-scale graphs.

High Memory Usage. For each hub vertex h, the method computes and stores both forward and reverse distances to each other vertex, i.e., $\text{DIST}(h \rightarrow v)$ and $\text{DIST}(v \rightarrow h)$, resulting in a memory cost of $2|V|$ per hub. This leads to a total space complexity of $2 \cdot \text{Hub} \cdot |V|$. When using 16 hubs, which is the default setting in SGraph, the memory overhead becomes substantial. For relatively sparse datasets (i.e., those with a high vertex-to-edge ratio), such as LiveJournal (LJ) [4], Yahoo (YH) [2], and Dimacs (DM) [5], the index size can exceed the size of the raw graph itself, as illustrated in Figure 5. For other datasets used in the experiments, the index size often exceeds 50% of the dataset size, making this approach impractical under single-machine.

Imprecise Bounds. To meet the memory limit, when only a small number of hubs can be used, the selected hub vertices may fail to capture long-range connectivity, leading to imprecise lower bounds. Figure 4 shows an example where the shortest path is computed from vertex 6 to vertex 4. SGraph selects vertex 2, which is the highest-degree node, as the hub vertex h, and precomputes distances from and to h for all vertices, as shown in Figures 4b and 4c. The resulting lower bounds, presented in Figure 4d, include many invalid or zero-valued entries, which are ineffective for pruning. Figure 6 shows the proportion of vertices for which the hub-based method achieves the

same bound precision as our approach. The results highlight a clear gap in bound tightness, with fewer than 20% of vertices matching our precision, while the majority exhibit looser estimates.

Key Insight I. Building on Challenge I, our key insight is that *graph sketches enable more efficient bound estimation than hub-based methods under memory constraints*, making them better suited for single-machine out-of-core settings, particularly on large-scale graphs. However, this introduces a new challenge: conventional graph sketches remain inefficient when applied to out-of-core monotonic graph algorithms.

Challenge II. Ineffective Graph Sketches. Existing state-of-the-art graph sketches, such as the weight-based graph sketch used in Wonderland [103], lack global topological awareness, limiting their ability to compute tight bounds and hindering effective navigation. Both of them are essential for efficient out-of-core graph processing. Specifically, we identify two key limitations:

Ineffective Bound Estimation. As observed, bounds computed directly from existing graph sketches such as Wonderland [103] are often even looser than those from hub-based methods. This is mainly because such sketches fail to preserve graph connectivity and topological structure, resulting in a high proportion of invalid (zero-valued) or overly loose bounds. Consequently, pruning performance degrades significantly. As shown in Figure 6, fewer than 6.5% of vertices yield bounds as precise as those from our method.

Poor Navigation Capability. GraphQ [87] builds abstract sketches that lack a direct mapping to original graph edges (Figure 3a), limiting their ability to reduce data access or support effective information propagation across partitions. This severely slows convergence, leading to significantly lower performance compared to more recent systems like Wonderland [103]. Wonderland improves by using actual edges but relies solely on edge weights, failing to capture global topology. As shown in Figure 3b, this can result in selecting rarely traversed edges (e.g., red edges), which offer poor guidance for inter-partition propagation and delay convergence. For example, under an 8 GB memory constraint on the 17 GB Twitter graph, GraphQ takes 291.55 s to compute shortest paths, Wonderland takes 24.37 s, while our approach completes in just 4.48 s.

Key Insight II. Building on the above challenge, our second key insight is that *constructing graph sketches using both edge weights and the likelihood of edges appearing in monotonic query results (e.g., shortest paths) helps preserve query-relevant topological structure.* This joint design improves navigational capability and enhances bound accuracy on graph sketches, enabling more effective pruning in out-of-core settings.

4 Overview

Building on the above key insights, our core idea is to *leverage PageRank as a metric to construct a navigable graph sketch that facilitates information propagation across graph partitions in out-of-core settings.* Our adoption of PageRank is motivated by the key observation that *the likelihood of an edge appearing in the result path of a monotonic query correlates with its visitation probability during a random walk*, a property that PageRank effectively captures. By leveraging this insight, the PageRank-guided graph sketch preserves essential topological structures and enables efficient navigation across graph partitions. This design achieves a favorable balance between low preprocessing overhead and high-performance query execution (see Section 7.14).

Based on the above idea, we propose Gem, a novel monotonic graph processing engine for single-machine execution which uses a navigable PageRank-based graph sketch to efficiently propagate information between graph partitions. We also design a graph abstraction that combines the graph sketch with graph partitions to estimate tighter and more accurate bounds for pruning monotonic graph queries. Gem 's workflow has two main stages: offline preprocessing and online query processing. In the out-of-core setting, the workflow is as follows:

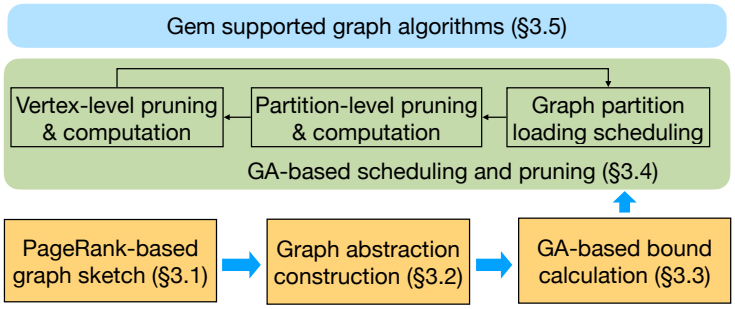


Fig. 7. The architecture of Gem.

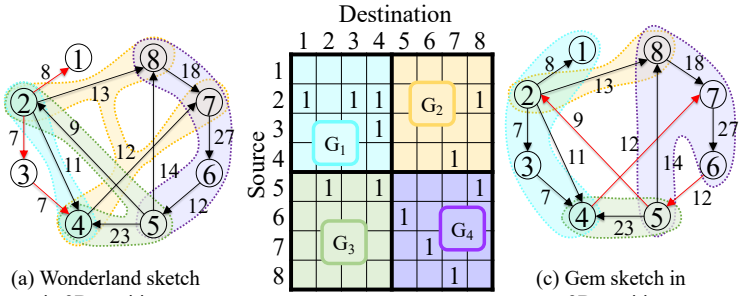
- **Offline Preprocessing Stage.** In this stage, Gem employs PageRank to guide the construction of a navigable graph sketch (see Section 5.1). It first computes PageRank scores for all vertices and ranks edges based on a combination of their weights and the PageRank values of their endpoints. The top-ranked edges are then selected to form the graph sketch. Subsequently, Gem constructs a graph abstraction (GA) by representing each graph partition as a vertex and incorporating both graph sketch edges and inter-partition edges (see Section 5.2).
- **Online Graph Query Stage.** During query execution, Gem first estimates pruning bounds using the GA, which is compact enough to reside entirely in memory with minimal overhead (Section 5.3). Then, for each iteration of the monotonic graph algorithm, Gem first updates the in-memory graph sketch, followed by updating each partition using a combination of vertex-level and partition-level pruning strategies (Section 5.4).

Supported Execution Modes. Gem supports single-machine graph processing across multiple execution modes, including in-memory, out-of-core, and out-of-GPU-memory, as detailed below:

- **In-Memory Execution.** In this mode, both the graph sketch and the original graph are fully stored in memory. Since there is no disk I/O involved, the navigation benefits of the graph sketch are diminished. However, the graph abstraction-based pruning remains effective by reducing redundant computations and search space.
- **Out-of-Core Execution.** In the out-of-core setting, the graph sketch is retained in memory while the partitioned graph resides on disk. The PageRank-based graph sketch efficiently guides information propagation across partitions, reducing the number of iterations and lowering disk I/O overhead. In addition, the proposed pruning strategies further reduce the search space, leading to improvements in both I/O efficiency and computation time.
- **Out-of-GPU-Memory Execution.** Similar to the out-of-core setting, in the out-of-GPU-memory mode, Gem stores the graph sketch in GPU HBM, keeps the partitioned graph in CPU DRAM, and loads graph partitions into GPU memory sequentially until convergence. This design leverages the PageRank-based graph sketch to reduce CPU-to-GPU data transfers and uses graph abstraction to prune redundant search space.

5 Gem System Design

Figure 7 illustrates the main components of Gem, which include graph sketch and abstraction construction, bound calculation, and runtime execution. For simplicity, we use the shortest path algorithm as the running example throughout the paper. However, the proposed techniques are applicable to a broad class of monotonic graph algorithms (Section 5.5).



(a) Wonderland sketch in 2D partition. (b) 2D graph partition. (c) Gem sketch in 2D partition.

Fig. 8. (c) Gem selects a superior graph sketch as a "bridge" between partitions compared to (a) Wonderland.

5.1 PageRank-based Graph Sketch (GS)

We first present our PageRank-based graph sketch, designed to enable effective navigation across graph partitions.

Gem's Metric for Sketch Construction. We introduce the following metric for graph sketch construction, for each $e_{(v,u)}$:

$$P_{e_{(v,u)}} = min\{R(v), R(u)\}, \tag{1}$$

$$S_{e_{(v,u)}} = P_{e_{(v,u)}} \oplus w_{e_{(v,u)}}, \tag{2}$$

where $R(v)$, $R(u)$, $w_{e_{(v,u)}}$ are the PageRank values of source v and destination u of edge $e_{(v,u)}$, and edge weight for edge $e_{(v,u)}$. $P_{e_{(v,u)}}$ is defined as the minimum PageRank value of the vertices at each edge $e_{(v,u)}$, respectively. Then, $S_{e_{(v,u)}}$ denotes the PageRank value of each edge $e_{(v,u)}$ which is derived with $P_{e_{(v,u)}}$ and $w_{e_{(v,u)}}$ following operator $\oplus$. Edges with higher PageRank values are selected in the graph sketch. In practice, we do not wait for the convergence of the PageRank calculation, as it can be too time-consuming. We use the PageRank values after 10 iterations (Section 7.7).

For the shortest path, $\oplus$ is defined as a divide function, that is, $\frac{P_{e_{(v,u)}}}{w_{e_{(v,u)}}}$. This implies that edges with smaller weights, and whose source and destination vertices have higher PageRank values, will have a higher probability of efficiently propagating information between different graph partitions. We chose such edges for our graph sketch. Other graph algorithms are defined in Section 5.5.

Gem offers a better graph sketch than the other related works mainly because PageRank selects the edges that are topologically important.. Figure 8 exemplifies this fact. For the same toy example in Figure 8a, we partition it into four partitions in Figure 8b. One can see that none of the three edges in the Wonderland sketch connects different partitions in Figure 8a. In contrast, Gem offers a better sketch, as shown in Figure 8c, edge $e_{(4,7)}$ bridges the partitions G_1, G_3 and G_4, $e_{(5,2)}$ bridges all partitions, and $e_{(6,5)}$ links G_3 to G_4. In addition, we incorporate local edge weight information into our metric to strike a balance between local and global information.

It is worth noting that our running example is not a special case that PageRank is more effective at identifying global bridge edges than the weight-based method employed by Wonderland. Let the adjacency matrix of the graph be A, and the diagonal degree matrix be D. We arrive at the original stochastic matrix P as $P = (D^{-1}A)^T$. We update the stochastic matrix with damping factor c:

$$P' = (1-c)(P + \frac{\sum_j \mathbf{1}_j}{n}) + c\frac{I}{n}, \qquad \forall j \text{ s.t. } \sum_{i=1}^{n} p_{ij} = 0, \tag{3}$$

where I is an identity matrix, and $\mathbf{1}_j$ means a $n \times n$ matrix with the elements in column j are 1, and others are 0. P' is a column stochastic matrix, which means that P' must have an eigenvalue $\lambda_1 = 1$. Since every element $p'_{ij} > 0$, P' is a primitive matrix. According to the Perron-Frobenius

theorem, the Markov chain has a unique stationary distribution π and converges to it from any initial distribution. So, the PageRank vector is defined as π, more precisely as $P'\pi = \pi$, where $\pi > 0$ and $\mathbf{1}^T\pi = 1$.

PageRank π_i can be interpreted in the context of web surfers. Starting from any node of the graph, the PageRank value of each node is the potential probability that a web surfer will arrive at that particular node. Clearly, edges with higher PageRank values are topologically critical edges.

5.2 Graph Abstraction (GA) Construction

Building on the PageRank-based graph sketch, we introduce a graph abstraction (GA) that enables efficient computation of tight bounds. Gem maintains two GAs: a forward GA and a backward GA. For example, in the shortest path algorithm, the forward GA estimates $LB(src \rightarrow v)$ and the backward GA estimates $LB(v \rightarrow dest)$. Other monotonic algorithms, like the widest path, also rely on both GAs to compute upper bounds in each direction.

Each GA is a graph containing a *vertex set*, an *edge set*, and *edge weights*. The vertex set contains the source and destination vertices of the graph sketch and the abstract vertices. Here, each graph partition is treated as an abstract vertex. Notably, the vertex set is the same for both forward and backward GAs.

```
1  func ForwardGA(GA, Sketch, 𝔾)
2      foreach G_i ∈ 𝔾: //Green edges
3          foreach G_j ∈ 𝔾:
4              if G_i.in_vertex ∩ G_j.out_vertex != ∅:
5                  GA.add_edge(G_i, G_j, G_i.min_edge)
6      foreach v ∈ Sketch: //Blue edges
7          foreach G_i ∈ 𝔾:
8              if v ∈ G_i.out_vertex:
9                  GA.add_edge(v, G_i, 0)
10             if v ∈ G_i.in_vertex:
11                 GA.add_edge(G_i, v, G_i.min_edge)
```

Fig. 9. Algorithm for the forward GA construction.

The edge and weight sets construction are slightly more complex: forward (backward) GA contains (i) the (inverse of) red sketch edges (**real edges**); (ii) the green edges between graph partitions (**bound-aware virtual edges**); (iii) the blue edges connecting vertices in the graph sketch with the graph partitions (**bound-aware virtual edges**). Since the red edges are either the sketch or simply the transpose of the sketch for forward and backward GAs, respectively, we only explain how to build the green and blue edges.

Forward GA: Construction Rule of Green Edges Between Graph Partitions. Figure 9 shows the pseudocode for constructing green edges in the forward GA. For each partition pair G_i and G_j, if they share a vertex that has an in-neighbor in G_i and an out-neighbor in G_j, we add a green edge $G_i \rightarrow G_j$. The edge weight depends on the algorithm: the minimum edge weight in G_i for shortest path, the maximum for widest path, and 1 for reachability. For example, in Figure 10a, edge $(G_2 \rightarrow G_4, 13)$ is created by vertex 8, where 13 is the minimum edge weight in G_2. This edge is valid because: (i) Vertex 8 links G_2 to G_4 through its in- and out-edges, (ii) The minimum weight in G_2 gives a valid bound for transitions to G_4, and (iii) The weight can be precomputed and cached, avoiding reloading G_2. Note that only the source partition contributes to the bound in forward GA, so G_4's weights are not used.

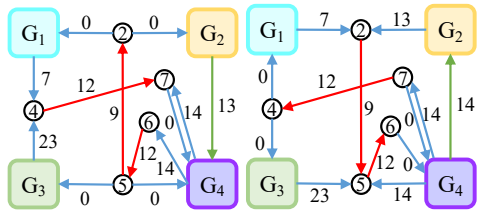


Fig. 10. The (a) Forward GA and (b) Backward GA example.

Forward GA: Construction Rule of Blue Edges Between Graph Sketch and Graph Partitions. Blue edges link sketch vertices and graph partitions to ensure connectivity, as each sketch vertex must appear in at least one partition. As shown in Figure 9, there are two types of blue edges: (i) from sketch vertices to partitions, and (ii) from partitions to sketch vertices. For type (i), if a sketch vertex has an outgoing edge in a partition, we add a blue edge from the sketch vertex to that partition. This edge is assigned a weight of 0 to indicate that the vertex belongs to the partition. For example, in Figure 10a, we create edges from vertex 2 to G_1 and G_2, from vertex 5 to G_3 and G_4, and from vertex 7 to G_4. For type (ii), if a partition contains an edge pointing to a sketch vertex, we create an edge from the partition to that sketch vertex. The edge weight is set to the minimum edge weight in the partition to provide a valid bound for all transitions. For instance, since partition G_4 connects to vertices 6 and 7, we add edges $G_4 \longrightarrow 6$ and $G_4 \longrightarrow 7$, both with weight 14, the minimum weight in G_4.

```
1  func BackwardGA(GA, Sketch^T, 𝔾)
2      foreach G_i ∈ 𝔾: //Green edges
3          foreach G_j ∈ 𝔾:
4              if G_i.out_vertex ∩ G_j.in_vertex != ∅:
5                  GA.add_edge(G_i, G_j, G_i.min_edge)
6      foreach v ∈ Sketch^T: //Blue edges
7          foreach G_i ∈ 𝔾:
8              if v ∈ G_i.in_vertex:
9                  GA.add_edge(v, G_i, 0)
10             if v ∈ G_i.out_vertex:
11                 GA.add_edge(G_i, v, G_i.min_edge)
```

Fig. 11. Algorithm for backward GA construction.

Backward GA Construction. In the backward GA, the graph is transposed, reversing edge directions while keeping weights from the new source partitions. The pseudocode is shown in Figure 11. For green edges, if a vertex is shared between partitions G_i and G_j, and has an out-neighbor in G_i and an in-neighbor in G_j, we add an edge $G_i \longrightarrow G_j$. For example, the edge $(G_4 \longrightarrow G_2, 14)$ in Figure 10b. For blue edges, we consider two cases: (i) An edge from each sketch vertex to its partition with weight 0. (ii) An edge from a partition to a sketch vertex if it has outgoing neighbors there, using the partition's minimum edge weight. Figure 10b shows examples: edge $4 \longrightarrow G_1$ (weight 0) and edge $G_4 \longrightarrow 5$ (weight 14).

5.3 GA-based Bound Construction

Based on the proposed GA, this section presents the strategy for calculating the tight bounds.

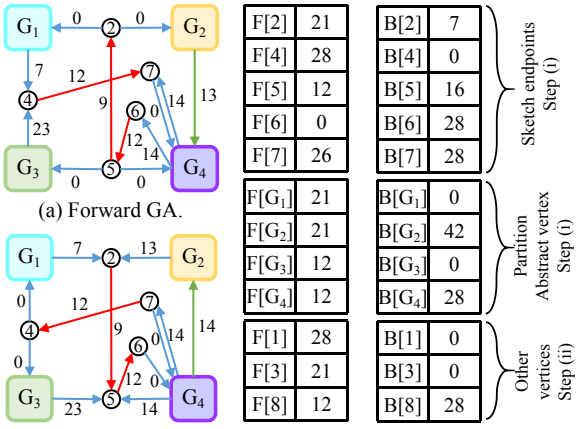


Fig. 12. Forward LB and backward LB of example graph, where for query 6 → 4.

Given a shortest-path query $src \rightarrow dest$, we derive the $LB(src \rightarrow v)$ and $LB(v \rightarrow dest)$ based on the forward and backward GAs. This process is achieved in two steps: (i) we derive the $LB(src \rightarrow v)$ and $LB(v \rightarrow dest)$ for the endpoints of the sketches and graph partitions. (ii) Based on step (i), we derive the $LB(src \rightarrow v)$ and $LB(v \rightarrow dest)$ for the other vertex v of the entire graph.

Adopting programming style, we use the forward array $F[v]$ to replace $LB(src \rightarrow v)$, and the backward array $B[v]$ to represent $LB(v \rightarrow dest)$. $F[]$ and $B[]$ are built on the forward and backward GAs, respectively.

For step (i), $F[]$ and $B[]$ for each sketch vertex are directly derived by applying the shortest path algorithm on the GAs. In particular, we use the shortest path algorithm in the forward GA from src to derive $F[]$, and that on the backward GA from $dest$ for $B[]$. Subsequently, we derive the F and B of each graph partition G_i as follows. Since we treat G_i as an abstract vertex and v which resides in G_i and has outgoing neighbors in G_i separately, $F[G_i]$ is set as $min\{F[G_i], F[v]\}$. Similarly, for the backward GA, all edges are reversed, so $B[G_i]$ is set as $min\{B[G_i], B[v]\}$ with v in the neighbors of G_i. The rationality behind this design is that the bound of each graph partition is equal to the min or max of the bound of each out vertex inside the graph partition.

Figure 12 exemplifies the process with a graph query from 6 to 4. We focus on the graph partition G_4. We first use the shortest path to get $F[G_4] = 12$. Subsequently, since the sketch vertex 5 and 7 are the vertices with outgoing neighbors in G_4, we arrive at $F[G_4] = \min\{F[G_4], F[5], F[7]\} = 12$.

Once step (i) is done, step (ii) falls into three cases:

- **Case 1:** If v is in the graph sketch, we directly use the calculated $F[v]$ for $LB(src \rightarrow v)$ and $B[v]$ for $LB(v \rightarrow dest)$, respectively.
- **Case 2:** If v is not in the graph sketch and has outgoing neighbors in several graph partitions, G_i's, $F[v] = min\{F[G_i]\}$ for all G_i's. Further, (i) If v holds in neighbors from graph partitions G_j's, we have $B[v] = min\{B[G_j]\}$. (ii) If v does not have any in neighbors from any graph partitions, which means the $LB(v, dest)$ is at least the distance of the shortest edge in this G_i plus $LB(v, G_i)$, so $B[v] = min\{B[G_i] + G_i.min\_edge\}$.
- **Case 3:** If v is not in the graph sketch and only has in neighbors from several graph partitions G_i's, this means $LB(src, v)$ is at least the distance of smallest edge weight in each G_i plus $LB(src, G_i)$. That is, $F[v] = min\{F[G_i] + G_i.min\_edge\}$, and $B[v] = min\{B[G_i]\}$.

Figure 12 shows three cases: **Case 1**: Vertices {2, 4, 5, 6, 7}; **Case 2**: Vertex 3 has one out-neighbor and one in-neighbor in G_1, so $F[3] = F[G_1] = 21$ and $B[3] = B[G_1] = 0$. Also, $B[7]$ is updated

as $\min\{B[7], B[G_4]\}$; **Case 3**: Vertex 1 only has an in-neighbor from G_1, so $F[1] = F[G_1] + G_1.\text{min\_edge} = 28$ and $B[1] = B[G_1] = 0$.

5.4 GA-based Scheduling and Pruning

Using the proposed bounds, we design effective scheduling and pruning strategies for both graph partitions and vertices to reduce redundant computations and minimize disk I/O.

Graph Partition Scheduling. Gem loads graph partitions based on their priority, defined using bounds. For each partition $G_i \in \mathbb{G}$, the priority is `Combine`$(F[G_i], B[G_i])$, e.g., $F[G_i] + B[G_i]$ for shortest paths. After the first iteration, $F[G_i]$ is updated using distances from vertices in G_i, and the final priority becomes $\min\{F[G_i] + B[G_i]\}$. This approach favors loading partitions that are more likely to lead to shorter paths, thereby accelerating convergence.

Vertex-level Pruning. For a particular query $src \rightarrow dest$ and the to-be-loaded edge (v, u, w), our GA-based pruning approach comprises the following three steps:

- **Step 1:** We check if the sum of the forward and backward bounds of v exceeds the current distance (noted $dist[src \rightarrow dest]$) from the source vertex src to the destination vertex $dest$, e.g., $F[v] + B[v] \geq dist[src \rightarrow dest]$ for the shortest path. If this condition is met, v is pruned, that is, the incoming and outgoing edges of v will never be loaded or updated in subsequent iterations.
- **Step 2:** If step 1 does not hold true, we proceed to calculate $dist[src \rightarrow v] + B[v]$. If $dist[src \rightarrow v] + B[v] \geq dist[src \rightarrow dest]$, we will neither load any outgoing edges of v nor perform any v incurred distance updates at this iteration. Different from Step 1, since $dist[src \rightarrow v]$ could be updated later, this step does not completely prune the vertex v from future iterations.
- **Step 3:** If Step 2 turns out false, which implies there is a possibility that the shortest path $src \rightarrow dest$ passes v, we therefore load the edge (v, u, w). Now, we can check if $dist[src \rightarrow v] + w + B[u] \geq dist[src \rightarrow dest]$. If yes, we will neither update $dist[src \rightarrow u]$, nor activate u. Otherwise, we will update $dist[src \rightarrow u]$ and activate vertex u.

  Note that, the triangle inequality always holds for monotonic graph workloads.

Graph Partition-level Pruning. Similar to Wonderland's selective loading of graph partitions, there is a chance that we might not need to load the entire graph partition. We introduce two steps to determine the inactive partitions as follows.

- **Step 4:** Similar to Step 1, we treat each graph partition as a single vertex. For each partition G_i, we check whether the sum of its forward and backward bounds exceeds the current distance from t src to $dest$. If so, we mark G_i as inactive.
- **Step 5:** Otherwise, i.e., $F[G_i] + B[G_i] < dist[src \rightarrow dest]$, we will derive tighter bounds by assessing whether each edge within the graph partition G_i does not need to be updated. This involves checking if $min\{dist[src \rightarrow v]\} + B[G] \geq dist[src \rightarrow dest]$, where v represents the source vertex of the edges within G_i. Should this condition be satisfied, the graph partition G_i can also become inactive.

As the algorithm nears convergence, more graph partitions become inactive due to effective pruning. Thus, Steps 4–5 yield greater benefits in later iterations.

When a partition is loaded into memory, we use multithreading to update its edges in parallel and update the `dist[]` array.

5.5 Gem Supported Graph Algorithms

In addition to *Reachability* [38], *Shortest Path* [27], *Widest Path*, and *Weakly Connected Components (WCC)*, which are supported by recent systems [45, 103], Gem also supports *Breadth-First Search (BFS)*, *k-Nearest Neighbor Search (kNN)* [30], and *Minimum Cost Spanning Tree (MCST)* [57]. Since BFS is similar to the shortest path algorithm, we focus on the details of kNN and MCST evaluated

Applications	Reach/WCC	SP/BFS/kNN/MCST	WP
Method (⊕)	$P_{e_{(v,u)}}$	$P_{e_{(v,u)}}/w_{e_{(v,u)}}$	$P_{e_{(v,u)}} * w_{e_{(v,u)}}$

Table 2. PageRank value method(⊕) of each edge $e_{(v,u)}$.

Applications	**Pruning Conditions**
(a) Reachability/WCC	$UB(src \rightarrow v) \times UB(v \rightarrow dest) = 0$
(b) Shortest Path/ BFS/kNN/MCST	$LB(src \rightarrow v) + LB(v \rightarrow dest) \geq UB(src \rightarrow dest)$
(c) Widest Path	$\min\{UB(src \rightarrow v), UB(v \rightarrow dest)\} \leq LB(src \rightarrow dest)$

Table 3. LB/UB usage for various graph algorithms.

in Section 7.2. Table 2 shows the PageRank calculation strategy for different graph applications, and Table 3 shows the bound usage and pruning conditions of these algorithms.

(a) Reachability/WCC: Reachability and WCC are unweighted graph queries important for tasks like graph clustering [73] and higher-order connectivity [14]. As shown in Table 2, the PageRank score of an edge, $S_{e_{(v,u)}}$, equals $P_{e_{(v,u)}}$ since it depends only on vertex PageRank values, not edge weights. In Table 3, for both Reachability and WCC, we define the connectivity bound from the source to vertex v as $UB[src \rightarrow v]$, and from v to destination as $UB[v \rightarrow dest]$. We set $UB[src \rightarrow v] = 0$ if no path exists in the forward GA, and 1 otherwise. Similarly, $UB[v \rightarrow dest]$ is derived from the backward GA. A vertex v is pruned if $UB[src \rightarrow v] \times UB[v \rightarrow dest] = 0$; otherwise, it remains active.

(b) Shortest Path/BFS/kNN/MCST: To address the queries of the shortest path or kNN problem, the PageRank value of each edge $S_{e_{(v,u)}}$ is defined as $\frac{P_{e_{(v,u)}}}{w_{e_{(v,u)}}}$ because it is higher when the edge weight is smaller and we refine the functionality of the bounding function LB as detailed above. Here we explain the key pruning condition for it. As shown in Table 3, we can get $LB[src \rightarrow v]$ and $LB[v \rightarrow dest]$ from the forward and backward GA, separately. Then, if $LB[src \rightarrow v] + LB[v \rightarrow dest] \geq UB[src \rightarrow dest]$, then v is pruned. And note that for applications like Breadth-First Search (BFS), similar to the shortest path problem, we use the same pruning condition but treat all edge weights as 1.

(c) Widest Path: For the widest path problem, the objective is to find a path where the smallest edge weight along the path is maximized. We define the PageRank score of each edge as $S_{e_{(v,u)}} = P_{e_{(v,u)}} \cdot w_{e_{(v,u)}}$, combining PageRank and edge weight to prioritize stronger edges. Based on this, we redefine the upper bound UB. As shown in Table 3, $UB(src \rightarrow v)$ and $UB(v \rightarrow dest)$ represent the maximum possible lower limit on the path capacity from source to v (via forward GA) and from v to destination (via backward GA), respectively. During pruning, if $\min\{UB(src \rightarrow v), UB(v \rightarrow dest)\} \leq LB(src \rightarrow dest)$, then vertex v cannot contribute to a better solution and is pruned.

6 System Implementation and Optimizations

Implementation Details. Gem is implemented in 3K lines of C++ using `mmap` for out-of-core access. We use multithreads to update each edge in parallel within the in-memory graph, including both the graph sketch and the loaded graph partition. For data loading, `MAP_PRIVATE` handles static edge data, while `MADV_WILLNEED` is used to prefetch vertex data with dynamic updates. Although these flags have limited impact on performance, they simplify memory management. In each iteration, only active vertices and their outgoing edges are loaded; processing these edges may update the states of their destination vertices.

6.1 Generalizability Discussion

Generalizability of GS across Different Applications. Consistent with the prior graph sketch–based systems [45, 103], Gem allows a single graph sketch to be reused across various query types. For example, a sketch built for shortest path queries can also support kNN and MCST, as shown in Table 2. The sketch construction is divided into two steps: computing PageRank scores and selecting the top-k edges with the highest scores, as defined in Equation 2. A major benefit of this design is that PageRank needs to be computed only once. Since the edge score $P_{e_{(v,u)}}$ remains fixed across different queries, only the aggregation operator $\oplus$ varies by query type. This enables efficient sketch reuse through a single scan of the dataset during edge selection.

Generalizability of GS across Different Execution Modes. Our PageRank-based graph sketch supports two key optimizations: (i) *Sketch-based navigation*, which propagates information across graph partitions to reduce I/O and accelerate convergence under memory constraints; and (ii) *Graph abstraction–based pruning*, which estimates tight bounds to reduce the search space and avoid redundant computations. Optimization (i) is particularly effective in memory-constrained environments, such as out-of-core and out-of-GPU-memory settings. In contrast, optimization (ii) is broadly applicable across all execution modes, as it enhances efficiency regardless of the platform.

6.2 Preprocessing Design and Optimizations

Out-of-core PageRank Calculation. Gem uses the PageRank value after 10 iterations (Section 7.7), with the initial value set to $\frac{1}{|V|}$, for sketch selection. For efficient out-of-core PageRank calculation, we adopt LUMOS [83]. Note that for a specific graph dataset, once the PageRank value of each vertex is computed ($P_{e_{(v,u)}}$ in Equation 1), it does not need to be recalculated for different graph applications and the PageRank time can be averaged.

Out-of-core Top-k Edge Selection. In the edge selection stage, we select the top-k edges with the highest PageRank scores, denoted as $S_{e_{(v,u)}}$. Gem avoids the need for a full sorting step by using multithreading to find the top-k edges in an out-of-core manner. Specifically, each graph partition is loaded into memory to build the graph sketch (GS). For each edge e in the partition, we perform a binary search on the sketch to find its insertion position. This follows a bucket top-k paradigm [11]. If multiple edges need to be inserted, threads perform the search in parallel. The actual insertions, which are lightweight, can be done either serially or in parallel. This search process has a time complexity of $O\left(\frac{P\log(X)}{\text{Thread\_number}}\right)$ per partition and $O\left(\frac{|E|\log(X)}{\text{Thread\_number}}\right)$ overall, where P is the partition size and X is the GS size. In contrast, the traditional GA construction in Wonderland [103] requires sorting all edges with a time complexity of $O(|E|\log|E|)$, which introduces significant overhead.

6.3 Graph Sketch Update for Evolving Graphs

To handle evolving graphs, we use a decoupled design that separates sketch updates from PageRank recomputation, based on the observation that small updates have minimal impact on sketch quality. The graph sketch is updated online using existing PageRank scores, with an amortized cost of $O(\log|GS|)$ per edge. PageRank is recomputed only when query performance falls below a user-defined threshold α, at which point the sketch is refreshed. Formally, let the original graph be $\mathbb{G}$ and updates be $\mathbb{U}$, giving an updated graph $\mathbb{G}' = \mathbb{G} \cup \mathbb{U}$. For *insertions*, we reuse PageRank scores from $\mathbb{G}$ to compute $S_{e_{(v,u)}}$ for each new edge $e(v,u) \in \mathbb{U}$ (per Equation 2), and insert into a max-heap $\mathbb{Q}$ of size $|GS|$, costing $O(|\mathbb{U}|\log|GS|)$. For *deletions*, we remove affected edges from $\mathbb{Q}$ in $O(\log|GS|)$ time.

Graph	Vertices	Edges	Data size	Diameter
LiveJournal (LJ) [4]	4.85M	68.48M	790MB	15
Twitter (TW) [52]	41.65M	1.47B	17GB	26
Friendster (FS) [3]	65.61M	1.81B	41GB	32
Yahoo (YH) [2]	1.41B	6.64B	75GB	928
Dimacs (DM) [5]	23.94M	58.33M	668MB	8122
Clueweb (CW) [1]	978M	42.5B	317 GB	254

Table 4. Graph datasets.

To decide when to refresh PageRank, we track the average query latency. If query time on $\mathbb{G}'$ exceeds that on $\mathbb{G}$ by more than α, we incrementally update the PageRank scores $R(u)$ for vertices affected by $\mathbb{U}$, using the method from GraphBolt [65]. With $\alpha = 20\%$, for example, PageRank on Friendster is recomputed only after 22M edge changes, keeping overhead low for small updates.

7 Evaluation

7.1 Evaluation Setting

Testbed. All experiments are conducted on a server with two Intel(R) Xeon(R) E5-2640 v2 @ 2.00 GHz CPUs (16 threads total), 256 GB DRAM, and a 1 TB SATA SSD (650 MB/s read speed). To enforce memory constraints in the out-of-core setting, we use the OS-level *cgroup* feature, following prior work like Wonderland [103].

Graph Datasets. Table 4 lists the six graph datasets used in our evaluation. Yahoo and Clueweb have the most vertices and edges, while Dimacs has the longest diameters. These datasets are widely used in recent studies [22, 83], and all data is stored in CSR format.

Gem Parameter Configurations. Following the approach of Wonderland [103], we tune two key parameters for Gem: the graph sketch (GS) size X and the graph partition size P, ensuring that $X + P$ fits within the available memory. Our experiments (see Section 7.6) show that setting X to 30% of the graph size yields good performance. When $X < 30\%$, increasing X improves performance. Based on this, if available memory exceeds 60% of the graph size, we set X to 30% of the total number of edges. Otherwise, X is set to half of the available memory divided by the size of each edge. Once X is fixed, we assign the largest possible P such that $X + P$ stays within the memory limit. These values of X and P determine the graph abstraction (GA) size, which is used to estimate computational bounds.

Evaluation Methodology. For the out-of-core setting, similar to state-of-the-art engines like Wonderland [103], we set memory limits at $\frac{1}{2}$, $\frac{1}{4}$, and $\frac{1}{8}$ of the graph size. Our baselines are **Wonderland** [103], **GridGraph** [111], **Blaze** [50], **CLIP** [9], **LUMOS** [83], **RisGraph** [33], **SGraph** [22], **Core Graph** [45] and **CGgraph** [31]. For each application, we generate 20 random source-destination pairs for point-to-point queries. Gem and all baselines use the same set of query pairs for fair comparison and we verify that Gem produces the same outputs as all baselines to ensure correctness.

7.2 Gem vs. State-of-the-art

Following prior work [103, 111], when comparing with SOTA engines, we focus on execution time including bound estimation and graph query, since preprocessing can be amortized over multiple queries. Details on preprocessing and end-to-end performance are provided in Section 7.14 and Section 7.10, respectively.

Gem vs. Wonderland. Table 5 compares the execution time of Gem and Wonderland. Across all four algorithms, Gem achieves the highest speedups on DM or YH and the lowest on LJ. For Reachability, speedup ranges from 1.36× (LJ) to 12.58× (DM). For Shortest Path, Widest Path, and WCC, speedups range from 2.05×, 1.97×, and 1.92× on LJ to 9.17×, 8.63×, and 8.95× on DM or YH,

Dataset	Mem Limit	Reachability		Shortest Path		Widest Path		WCC	
		Gem	WL	**Gem**	WL	**Gem**	WL	**Gem**	WL
LJ	1/8	1.66 (**1.36**×)	2.25	3.11 (**2.05**×)	6.38	3.69 (**1.97**×)	7.27	1.42 (**1.92**×)	2.73
	1/4	0.72 (1.60×)	1.15	1.08 (4.84×)	5.23	1.91 (5.04×)	9.63	0.52 (2.31×)	1.20
	1/2	0.37 (1.94×)	0.72	0.68 (6.75×)	4.59	0.72 (6.72×)	4.84	0.22 (2.90×)	0.64
TW	1/8	3.68 (3.37×)	12.39	18.12 (2.52×)	45.66	27.12 (2.00×)	54.24	4.54 (2.60×)	11.80
	1/4	1.64 (3.75×)	6.15	8.76 (3.85×)	33.72	16.27 (2.06×)	33.52	2.76 (3.83×)	10.57
	1/2	0.56 (5.21×)	2.92	4.48 (5.44×)	24.37	7.92 (3.00×)	23.76	1.36 (5.35×)	7.28
FS	1/8	6.03 (3.73×)	22.49	24.24 (3.32×)	80.51	32.94 (2.47×)	81.47	10.92 (2.74×)	30.03
	1/4	2.65 (4.78×)	12.68	12.56 (4.33×)	54.42	17.37 (3.25×)	56.38	6.04 (4.85×)	29.33
	1/2	0.85 (4.89×)	4.15	7.30 (5.21×)	38.02	8.48 (4.52×)	38.33	3.13 (5.44×)	17.03
YH	1/8	78.24 (3.39×)	265.23	303.02 (6.42×)	1945.38	377.93 (5.43×)	2052.16	35.43 (8.43×)	298.67
	1/4	38.28 (3.44×)	131.68	194.49 (6.61×)	1285.57	207.30 (6.54×)	1355.74	18.27 (8.80)	160.77
	1/2	13.38 (4.42×)	59.22	97.13 (6.77×)	657.57	97.44 (7.09×)	690.84	10.72 (**8.95**×)	95.96
DM	1/8	14.65 (7.15×)	104.75	74.61 (7.35×)	548.38	79.20 (7.03×)	556.77	14.71 (6.93×)	101.94
	1/4	8.37 (9.35×)	78.26	40.59 (8.38×)	340.14	44.78 (8.25×)	369.43	8.33 (7.50×)	62.47
	1/2	3.43 (**12.58**×)	43.25	21.99 (**9.17**×)	201.64	27.27 (**8.63**×)	235.34	4.63 (7.87×)	36.44
CW	1/8	105.19 (3.41×)	359.30	428.91 (6.24×)	2678.21	450.28 (6.48×)	2916.87	102.66 (3.74×)	384.04
	1/4	46.43 (4.16×)	193.35	217.62 (6.31×)	1372.94	214.33 (7.03×)	1506.10	49.68 (4.21×)	209.51
	1/2	15.47 (5.81×)	89.90	103.13 (6.74×)	695.09	100.36 (7.59×)	761.86	25.20 (4.96×)	124.95

Table 5. Total execution time (seconds) and speedup of Gem (Ours) over Wonderland (WL).

respectively. The higher speedups on DM and YH stem from their long diameters, which require more iterations and benefit more from GA-based cross-partition communication. In contrast, LJ has a short diameter, limiting GA's advantage.

We observe that increasing memory capacity consistently improves Gem's speedup over Wonderland across all algorithms and datasets. More memory enables more detailed GAs, leading to tighter bounds and better navigation. As shown in Figure 13, average speedups increase significantly, e.g., from 1.83× to 4.57× on LJ and from 7.11× to 9.56× on DM, indicating that Gem performs best when sufficient memory is available to hold detailed GAs.

Gem vs. other SOTAs on Different Physical Memory Configurations. Figure 13 compares Gem with several out-of-core graph systems including GridGraph [111], Blaze [50], CLIP [9], LUMOS [83], and the in-memory SGraph [22], using the Shortest Path algorithm on the TW dataset. We vary the available memory to test scalability. While CLIP and LUMOS perform well with large memory, they degrade under tighter memory constraints. Blaze maintains stable performance in low-memory settings via binning-based scatter-gather, but lacks GA-based pruning and incurs

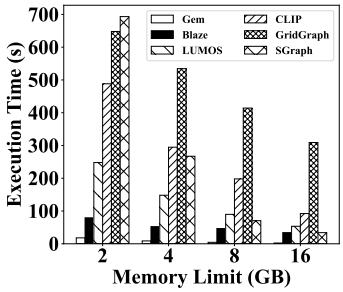


Fig. 13. Gem vs. SOTAs.

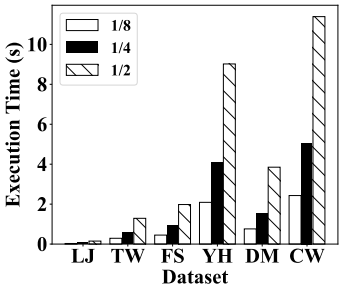


Fig. 14. Bound estimation.

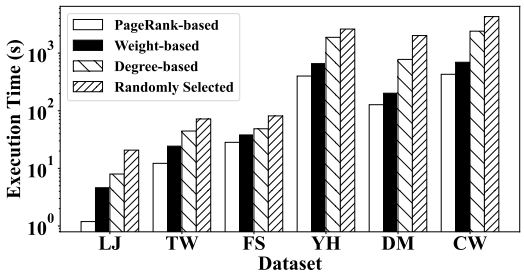


Fig. 15. Comparison of PageRank-based GS (Gem), weight-based GS (Wonderland), degree-based GS and randomly selected GS.

more I/O. Overall, Gem outperforms all baselines, achieving up to 135.40× speedup over GridGraph, 44.21× over LUMOS, 23.44× over CLIP, and 14.90× over Blaze.

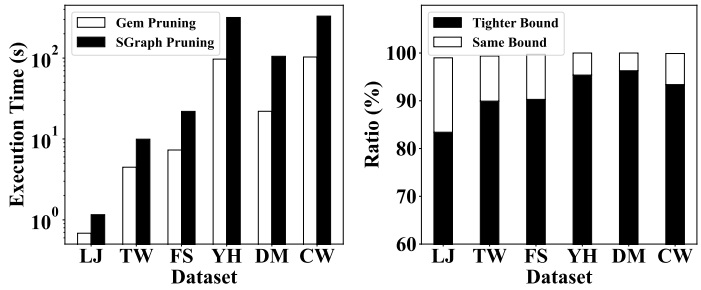
For SGraph, which was not designed as an out-of-core engine, we modified it to map its files (graph data, distance array, and bound array) directly from disk to memory using the *mmap* function. We found that SGraph performs poorly with low memory capacity, as it requires a large amount of space to calculate bounds using the triangle inequality. Its performance improves as available memory increases. Nonetheless, Gem consistently outperforms SGraph across all memory limits, achieving speedups from 14.90× to 38.26×.

Bound Estimation Time. Figure 14 shows the bound estimation time across different datasets and memory limits. It accounts for only 0.58% ∼ 12.51% of total execution time. Since the graph abstraction (GA) is much smaller than the full graph and fits in memory, fast in-memory algorithms (e.g., Dijkstra) can be used, keeping bound estimation efficient.

7.3 Efficiency of PageRank-based GS

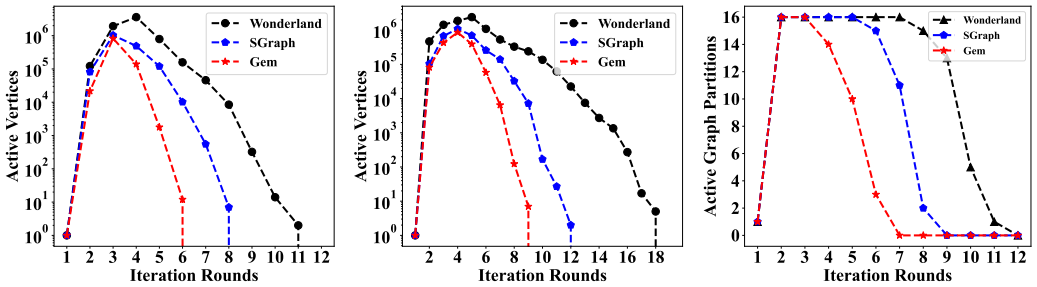
To evaluate the efficiency of Gem's navigable PageRank-based graph sketch (GS), we compare it with other GS selection strategies: the weight-based GS (Wonderland [103]), the degree-based GS (selecting vertices with the highest k degrees), and the randomly selected GS. For fairness, we implement each GS selection strategy in Gem without applying any additional optimizations (e.g, the bound-based pruning in Gem). We choose the shortest path as the application and set the memory limit as 1/2 for each dataset size.

In Figure 15, Gem's PageRank-based GS outperforms other graph sketch (GS) strategies, with speedups of 1.36×–3.86× over weight-based GS (Wonderland), 1.72×–6.69× over degree-based GS, and 2.88×–17.38× over random GS. The gains are most significant on long-diameter graphs like DM, YH, and CW, where topological information helps cross-partition propagation. Weight-based GS performs better than degree-based GS due to its alignment with shortest-path dynamics, while



(a) Performance comparison. (b) Bound tightness.

Fig. 16. Comparison of pruning efficiency and bound tightness (ratios of tighter or equal bounds).



(a) Twitter. (b) Friendster. (c) Partitions.

Fig. 17. Active vertices and graph partitions comparison.

random GS performs the worst due to a lack of graph structure information. Gem combines edge weights with global structure, leading to faster convergence.

7.4 Efficiency of GA-based Pruning

To demonstrate the efficiency of GA-based pruning using the tight bounds calculated by Gem, we compare its bound-based pruning strategy with that of SGraph [22]. For fairness, we integrate the SGraph pruning strategy into Gem and evaluate it alongside the GA-based pruning of Gem. We choose the shortest path as the application and set the memory limit as 1/2 for each dataset size.

Figure 16 shows that our GA-based pruning in Gem achieves 1.43× to 6.66× speedup over SGraph. The bounds in Gem are 83.42% to 96.28% tighter, with over 99% of bounds being equal or more precise across all datasets. This enables more effective pruning for monotonic problems like shortest path. Gem outperforms SGraph for two reasons: (i) SGraph's triangle-inequality-based bounds are less accurate than our GA-based bounds (Figure 16b); (ii) in out-of-core settings, SGraph stores large distance arrays on disk, incurring high I/O when accessing hub-related data. In contrast, Gem keeps the compact GA fully in memory, avoiding this overhead.

Gem vs. Wonderland and SGraph on Active Vertices. Figure 17 compares Gem with SGraph in terms of the number of active vertices across different iteration counts for the shortest path application on the TW and FS datasets. Overall, Gem activates fewer vertices and requires fewer iterations than both SGraph and Wonderland, leading to lower execution time and reduced disk I/O. Vertex-level pruning plays a key role in this efficiency, while graph partition-level pruning further reduces the number of iterations needed. Additionally, Gem activates significantly fewer graph partitions than SGraph and Wonderland, contributing to its overall superior performance.

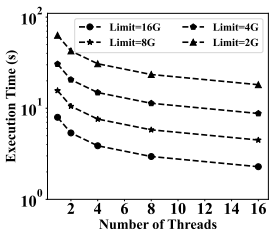


Fig. 18. Scalability.

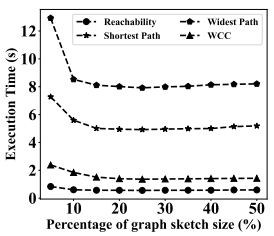


Fig. 19. GS size sensitivity.

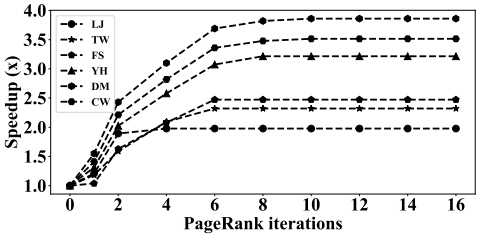


Fig. 20. PageRank iteration sensitivity.

7.5 Scalability

Figure 18 illustrates the scalability of Gem, i.e., the execution time under varying memory limits and thread counts with the shortest path algorithm on TW dataset. The speedup does not scale linearly with the number of threads, because multithreading primarily accelerates the computation phase, while disk I/O accounts for a large part of the overall execution time. However, even with these diminishing returns, Gem achieves a commendable 3.48× speedup with 16 threads over its single-threaded execution.

7.6 Graph Sketch Size Sensitivity

Because GA is constructed on top of the foundation of the graph sketch, i.e., GA is constructed by GS, a fixed GS size will lead to a fixed GA size. To test the impact of GA size on Gem, we directly study the performance of Gem under different GS sizes.

Figure 19 shows the impact of graph sketch (GS) size using the TW dataset with a memory limit of half the dataset. We evaluate Reachability, Shortest Path, Widest Path, and WCC. A larger GS does not always lead to better performance, i.e., it can tighten bounds and speed up convergence but also reduces memory available for graph partitions, increasing iterations. For monotonic algorithms like Shortest Path and Widest Path, larger partitions help convergence. Overall, the best performance occurs when GS size is between 10% and 40% of the dataset.

7.7 PageRank Iteration Sensitivity

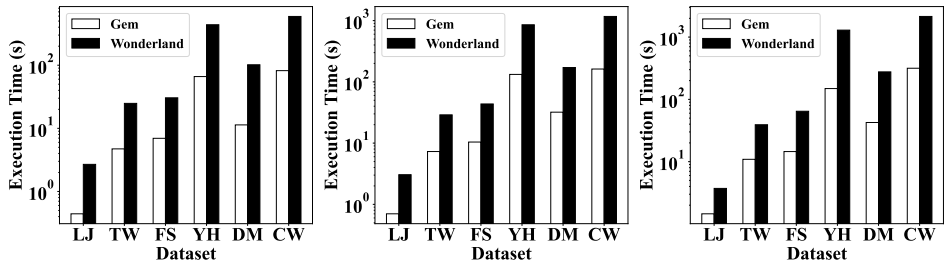
To assess the impact of different PageRank iterations on Gem's performance, we evaluate the performance of our PageRank-based graph sketch across various iterations for shortest path queries, with the memory limit set to half of the dataset size.

Figure 20 shows the speedup achieved with different numbers of PageRank iterations, using iteration 0 as the baseline. We observe that performance improvements plateau after 10 iterations across all datasets. For Dimacs, which has the longest diameter, 10 iterations are needed, while other datasets converge with fewer. This is because, as defined in Equation 2, the graph sketch scoring metric $S_{e_{(v,u)}}$ depends on both edge weights and PageRank values. After 10 iterations, further changes in vertex PageRank values have little effect on $S_{e_{(v,u)}}$, making additional iterations unnecessary. Therefore, we adopt 10 iterations as a default, with an early-stop policy.

7.8 Gem vs Wonderland on NVMe SSDs

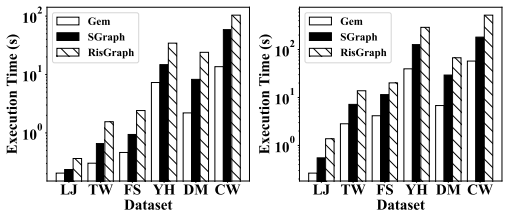
This section further evaluates Gem speedup over Wonderland on high-speed storage platforms, specifically a 512GB NVMe SSD, which offers sequential read bandwidths of 2.88GB/s. In this assessment, shown in Figure 21, we compare Gem against Wonderland with different memory limits on the shortest path.

Our results show that Gem achieves a speedup of 1.68× to 10.90× over Wonderland when using NVMe SSDs. Gem performs best on graphs with longer diameters, larger memory sizes, and larger



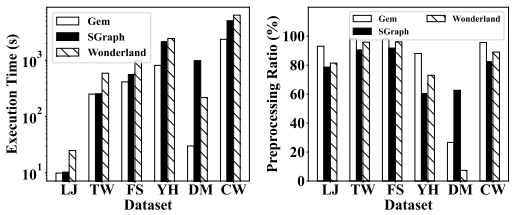
(a) 1/2. (b) 1/4. (c) 1/8.

Fig. 21. Gem vs Wonderland on NVMe SSD with the memory limit set as 1/2, 1/4, and 1/8 of the dataset size.



(a) Reachability. (b) Shortest Path.

Fig. 22. In-memory comparison with SOTAs.



(a) End-to-end time. (b) Preprocessing ratio.

Fig. 23. End-to-end time analysis.

overall sizes. For graphs of longer diameters, the navigation of the GA becomes more efficient in propagating information between different graph partitions, thereby reducing the number of iteration rounds. For larger memory sizes, the increased capacity leads to a larger GA, enabling it to obtain more extensive navigation and tighter bounds. Finally, larger graphs often involve more redundant edge updates. Our tight bound-based pruning technique can effectively reduce these redundant edge updates, thereby minimizing disk I/O operations and computational costs.

7.9 Comparisons for In-Memory Settings

Figure 22 presents the performance comparison of SGraph [22] and RisGraph [33] with Gem on a machine that contains 512 GB DRAM while the other configurations remain unchanged. Notably, both SGraph and RisGraph are inherently in-memory systems.

Figure 22 shows that Gem outperforms RisGraph with speedups from 1.78× to 10.41×, and SGraph with speedups from 1.16× to 4.79×. RisGraph does not use bounds to reduce redundant work, whereas SGraph applies triangle inequality-based bounds for in-memory pruning. However, SGraph's bounds are loose, as they rely on hub distances. In contrast, Gem uses tighter bounds from GA, resulting in stronger pruning and better in-memory performance.

This evaluation, along with Table 5 and Figure 21, demonstrates that Gem consistently outperforms state-of-the-art systems across different storage platforms. This superior performance stems from Gem's algorithm-level optimizations, which remain effective regardless of the underlying hardware configuration.

7.10 End-to-End Comparison

Here, we evaluate the end-to-end time, including preprocessing time, bound estimation time, and graph query time, for Gem, SGraph, and Wonderland, as shown in Figure 23. The memory is set to half the size of the dataset for the shortest-path queries.

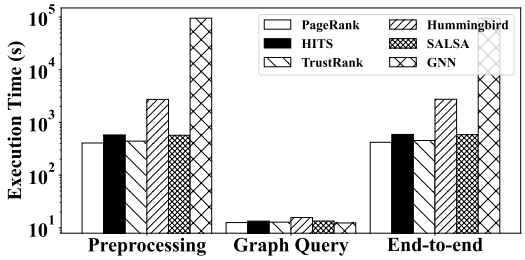


Fig. 24. PageRank vs. other GS selection algorithms.

While Gem includes extra PageRank computation during preprocessing, it achieves 1.02 ∼ 15.63× and 2.35 ∼ 7.26× speedups over SGraph and Wonderland, respectively. This is due to two main reasons: (i) For query execution (bound estimation + shortest path), Gem cuts the average runtime from 362.9s (SGraph) and 270.2s (Wonderland) to 39.1s; (ii) For preprocessing, it reduces the average time from 1138.6s (SGraph) and 1496.9s (Wonderland) to 606.9s. Overall, Gem lowers the end-to-end average from 1501.5s (SGraph) and 1767.2s (Wonderland) to 646.0s. Despite the added PageRank cost, its faster sketch selection (see Sections 6.2 and 7.14) keeps preprocessing efficient.

Role of Preprocessing Time in End-to-End Performance. As shown in Figure 23b, we report the proportion of preprocessing time in the total end-to-end runtime. For all datasets except DM (Dimacs), despite preprocessing accounting for over 90% of the total runtime, Gem significantly outperforms all baselines in end-to-end performance. This highlights the system's highly optimized query execution and efficient preprocessing design (see Section 6.2). The DM dataset has the longest diameter but the smallest size. It typically poses challenges for iteration-based algorithms, such as the shortest path. For this dataset, both Gem and Wonderland spend considerably less time on preprocessing. In Gem, PageRank converges in only a few iterations, often in as few as 10 rounds (see Section 7.7), without needing full convergence. Wonderland selects top-weighted edges, so its preprocessing cost mainly depends on the dataset size. As a result, the preprocessing ratios for both systems on DM remain below 35%. These are also lower than those of SGraph, which incurs high preprocessing costs due to computing the shortest paths from each hub to all other vertices.

7.11 PageRank vs. Other Selection Algorithms

We compared PageRank with other link analysis methods (e.g., HITS [56], TrustRank [36], Hummingbird [79], SALSA [55]) and embedding-based techniques (e.g., GNNs [37]) for graph sketch construction. Centrality-based methods, such as betweenness, were excluded because their preprocessing did not finish within three days. Figure 24 shows preprocessing time, query time, and end-to-end runtime on the Friendster dataset (41 GB) under a 10 GB memory limit.

For link-based methods, they behave similarly to PageRank on static graphs. HITS requires 1.4× more preprocessing time due to slower iterative updates, but it offers similar query performance. TrustRank adds no benefit when all nodes are seeds and reduces to PageRank. Hummingbird is 6.7× slower due to complex sorting, with no pruning gains. SALSA is similar to HITS in both cost and effectiveness. For embedding-based methods such as GNNs, they incur high costs in out-of-core settings due to backpropagation and frequent I/O. Specifically, using DiskGNN [61] with GraphSAGE [37] takes 26.4 hours on FS, while our PageRank-based sketch completes in 407s. Both achieve similar query performance (within 5%), but PageRank is 226× faster end-to-end.

In summary, PageRank offers the best trade-off between efficiency and effectiveness for out-of-core graph sketch construction, outperforming both link analysis and GNN-based methods.

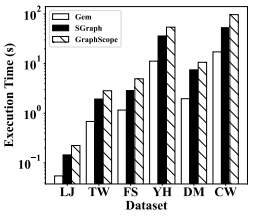


Fig. 25. Distributed test.

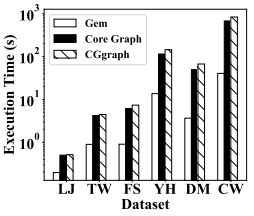
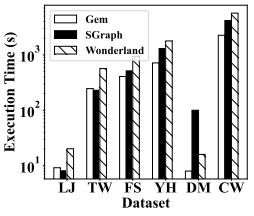
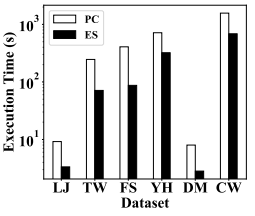


Fig. 26. Out-of-GPU test.



(a) Overall time.



(b) Breakdown.

Fig. 27. Preprocessing time analysis.

7.12 Evaluation in Distributed Settings

Gem can be naturally extended to a distributed setting, as its graph abstraction-based pruning effectively reduces redundant computation and search space without depending on a specific execution mode. To demonstrate this scalability, we extend Gem to distributed execution and present experimental results in Figure 25.

As shown in Figure 25, on an 8-node cluster connected via 200 Gbps InfiniBand, Gem achieves speedups of up to 3.81× over SGraph [22] and 5.63× over GraphScope [32]. The improvement over SGraph is due to Gem's precise GA-based bounds, which enable more effective pruning. In contrast, GraphScope does not employ efficient bound-based pruning, which limits its performance.

7.13 Out-of-GPU-memory Comparisons

In this section, we evaluate the performance of Gem in out-of-GPU-memory settings by comparing it with Core Graph [45] and CGgraph [31] on the shortest path application, using an A100 GPU equipped with 40 GB of HBM.

As shown in Figure 26, Gem achieves a speedup of 2.16× to 16.69× over Core Graph and 2.22× to 20.64× over CGgraph. The speedup is attributed to two main factors: (i) the PageRank-based GS enables Gem to efficiently propagate information between different graph partitions, speeding up convergence and significantly reducing PCIe I/O between CPU and GPU, while both Core Graph and Subway lack a navigable GS; (ii) the efficient GA-based pruning in Gem effectively reduces redundant edge updates and minimizes the I/O required for loading these edges to the GPU.

7.14 Study on Preprocessing

To assess preprocessing efficiency, Figure 27a compares Gem with SGraph and Wonderland. Figure 26b further breaks down Gem's preprocessing into PageRank computation (PC) and edge selection (ES); graph abstraction takes less than 1% of the total time. Both Gem and Wonderland build a graph sketch, while SGraph computes distances from 16 hub vertices to all others. The memory limit is set to half the dataset size.

As shown in Figure 27, Gem achieves up to 12.52× faster preprocessing than SGraph. Although SGraph skips sketch construction, it incurs high overhead by computing shortest paths from all hubs under out-of-core settings. It performs particularly poorly on long-diameter graphs like YH, DM, and CW, while Gem remains efficient. The largest gain is on the Dimacs dataset. For smaller graphs like LJ and TW, preprocessing times are similar, but Gem still outperforms overall. Compared to Wonderland, Gem is up to 3.39× faster in preprocessing, despite the added PageRank step. This is due to: (i) limiting PageRank to 10 iterations with multithreading, (ii) efficient partition access during PageRank, and (iii) our optimized top-k edge selection with complexity $O(\frac{|E|\log(X)}{\text{Thread\_number}})$, compared to Wonderland's $O(|E|\log|E|)$, which is not out-of-core friendly.

Amortized Preprocessing. Gem's preprocessing cost is quickly amortized. On the Friendster dataset, preprocessing takes around 400 seconds, while each query completes in just 10 seconds. In contrast, GridGraph requires over 200 seconds per query, so the preprocessing cost is offset after only 2–3 queries. Wonderland and SGraph also incur higher query times and greater preprocessing overhead. In real-world scenarios with frequent queries, this upfront cost is well justified.

In the preprocessing breakdown, PageRank accounts for roughly 75% of the total preprocessing time due to multiple rounds of dataset loading. This overhead is mitigated through two key strategies: (1) we avoid requiring full convergence by performing only a fixed number of iterations (e.g., 10 rounds), as described in Section 7.7; and (2) we employ a state-of-the-art out-of-core PageRank engine, such as LUMOS [83], to accelerate the computation.

8 Related Work

Pruning-based Graph Processing. KickStarter [84] pioneered pruning strategies using upper bounds to speed up streaming graph algorithms with monotonic properties by computing distances incrementally based on prior results. However, because edge deletions break this assumption, tagging strategies are used to ensure correctness. RisGraph [33] applies a similar approach to larger graphs, achieving high throughput through localized data access and parallel updates. Tripoline [47] formally introduced a triangle inequality-based method to derive upper bounds—a technique also adopted by PnP [91] and VRGQ [46]. SGraph [22] extends this by deriving both upper and lower bounds using the triangle inequality, which allows it to prune redundant vertices whose lower bounds exceed the current upper bounds. However, all these methods produce loose bounds, limiting their pruning effectiveness.

Out-of-core Graph Processing. Some out-of-core systems, like FlashGraph [108], Graphene [60], and CLIP-OPT[10], leverage NVMe SSDs and use selective I/O to load edges of active vertices for computation. Blaze [50] builds on these by introducing an online binning scatter-gather technique that efficiently utilizes SSD bandwidth and avoids atomic operations. However, none of these systems adopt GS/GA-based optimizations to further accelerate graph processing.

It is worth mentioning that other graph processing engines, such as distributed and in-memory engines [25, 34, 35, 54, 62–64, 75, 76, 90, 97, 102, 110], GPU-based engines [6, 18, 19, 31, 44, 70, 80, 93–95, 98, 100, 101, 106] and PIM-based engines [8, 15, 16, 104, 107, 112], can also potentially benefit from our GA-based pruning mechanism.

9 Conclusion

This paper introduces Gem, a new GA-based pruning system for efficient graph processing. Gem uses a PageRank-based method to generate a GS that effectively propagates information across partitions. By treating each partition as a vertex, we build a GA to derive bounds for scheduling and pruning at both the partition and vertex levels. Gem delivers up to 135.4× performance improvement over SOTA systems such as GridGraph, Wonderland, and SGraph on various algorithms and datasets.

Acknowledgments

We sincerely thank the reviewers for their insightful comments. This work is funded in part by the Nanjing "U35" Talent Cultivation Program (No. U (2024) 001), National Key Research and Development Plan of China (2023YFB4502305), National Natural Science Foundation of China (NO. 62072230, 62325205 and 62172204), the Key Program of Natural Science Foundation of Jiangsu under grant (No. BK20243053), Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University. Rong Gu is the corresponding author of this paper.

References

[1] [n. d.]. Clueweb dataset from WebGraph. https://law.di.unimi.it/webdata/clueweb12/ .

[2] [n. d.]. G2 - Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, circa 2002. http://webscope.sandbox.yahoo.com/.

[3] [n. d.]. S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/com-Friendster.html.

[4] [n. d.]. S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/soc-LiveJournal1.html.

[5] [n. d.]. The Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it /challenge9/download.shtml.

[6] Seher Acer, Erik G. Boman, Christian A. Glusa, and Sivasankaran Rajamanickam. 2021. Sphynx: A parallel multi-GPU graph partitioner for distributed-memory systems. *Parallel Comput.* 106 (2021), 102769. https://doi.org/10.1016/J.PARCO.2021.102769

[7] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 133–145. https://doi.org/10.1145/3575693.3575713

[8] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[9] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX Annual Technical Conference*. 125–137.

[10] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2019. Clip: A Disk I/O Focused Parallel Out-of-Core Graph Processing System. *IEEE Trans. Parallel Distrib. Syst.* 30, 1 (2019), 45–62.

[11] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach. 2012. Fast k-selection algorithms for graphics processing units. *ACM J. Exp. Algorithmics* 17, 1 (2012). https://doi.org/10.1145/2133803.2345676

[12] Jason Ansel, Edward Z. Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*. ACM, 929–947. https://doi.org/10.1145/3620665.3640366

[13] Vignesh Balaji, Neal Clayton Crago, Aamer Jaleel, and Stephen W. Keckler. 2023. Community-based Matrix Reordering for Sparse Linear Algebra Optimization. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2023, Raleigh, NC, USA, April 23-25, 2023*. IEEE, 214–223. https://doi.org/10.1109/ISPASS57527.2023.00029

[14] MO Ball, CJ Colbourn, and JS Provan. 1995. Chapter 11 network reliability. Network Models, volume 7 of Handbooks in Operations Research and Management Science.

[15] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, et al. 2021. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 282–297.

[16] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez-Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 282–297. https://doi.org/10.1145/3466752.3480133

[17] Maciej Besta, Cesare Miglioli, Paolo Sylos Labini, Jakub Tetek, Patrick Iff, Raghavendra Kanakagiri, Saleh Ashkboos, Kacper Janda, Michal Podstawski, Grzegorz Kwasniewski, Niels Gleinig, Flavio Vella, Onur Mutlu, and Torsten Hoefler. 2022. ProbGraph: High-Performance and High-Accuracy Graph Mining with Probabilistic Set Representations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*. IEEE, 43:1–43:17. https://doi.org/10.1109/SC41404.2022.00048

[18] Ian Bogle, George M. Slota, Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2022. Parallel graph coloring algorithms for distributed GPU environments. *Parallel Comput.* 110 (2022), 102896. https://doi.org/10.1016/J.PARCO.2022.102896

[19] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S. Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie S. Kim, Nika Mansouri-Ghiasi, Gagandeep Singh, Juan Gómez-Luna, Nour Almadhoun Alserr, Mohammed Alser, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. 2022. SeGraM: a universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*. ACM, 638–655. https://doi.org/10.1145/3470496.3527436

[20] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2023. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.* 55, 12 (2023), 253:1–253:38. https://doi.org/10.1145/3571808

[21] Joanna Che, Kasra Jamshidi, and Keval Vora. 2024. Contigra: Graph Mining with Containment Constraints. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 50–65. https://doi.org/10.1145/3627703.3629589

[22] Hongtao Chen, Mingxing Zhang, Ke Yang, Kang Chen, Albert Zomaya, Yongwei Wu, and Xuehai Qian. 2023. Achieving Sub-second Pairwise Query over Evolving Graphs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1–15.

[23] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 413–426. https://doi.org/10.1145/3575693.3575743

[24] Qihang Chen, Boyu Tian, and Mingyu Gao. 2022. FINGERS: exploiting fine-grained parallelism in graph mining accelerators. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 43–55. https://doi.org/10.1145/3503222.3507730

[25] Rong Chen and Haibo Chen. 2021. Wukong: A Distributed Framework for Fast and Concurrent Graph Querying. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 77–83. https://doi.org/10.1145/3469379.3469388

[26] Rong Chen, Youyang Yao, Peng Wang, Kaiyuan Zhang, Zhaoguo Wang, Haibing Guan, Binyu Zang, and Haibo Chen. 2018. Replication-Based Fault-Tolerance for Large-Scale Graph Processing. *IEEE Trans. Parallel Distributed Syst.* 29, 7 (2018), 1621–1635. https://doi.org/10.1109/TPDS.2017.2703904

[27] Zebin Chen, Kaiyu Chen, Dong Wen, Zhengyi Yang, Wentao Li, and Ying Zhang. 2025. Accelerating Shortest Path Counting on Road Networks. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 3508–3521. https://doi.org/10.1109/ICDE65448.2025.00262

[28] Zheng Chen, Feng Zhang, Yang Chen, Xiaokun Fang, Guanyu Feng, Xiaowei Zhu, Wenguang Chen, and Xiaoyong Du. 2024. Enabling Window-Based Monotonic Graph Analytics with Reusable Transitional Results for Pattern-Consistent Queries. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3003–3016.

[29] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 154–169.

[30] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[31] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1405–1417.

[32] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[33] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 513–527. https://doi.org/10.1145/3448016.3457263

[34] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30.

[35] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation*. 599–613.

[36] Zoltan Gyongyi, Hector Garcia-Molina, and Jan Pedersen. 2004. Combating web spam with trustrank. In *Proceedings of the 30th international conference on very large data bases (VLDB)*.

[37] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[38] Huangleshuai He, Zhengyi Yang, Dong Wen, Wenqian Zhang, Michael Yu, Wenke Yang, and Wenjie Zhang. 2025. A Survey on Efficient Graph Reachability Queries. In *Data Science: Foundations and Applications - 29th Pacific-Asia Conference on Knowledge Discovery and Data Mining, PAKDD 2025, Sydney, NSW, Australia, June 10-13, 2025, Proceedings, Part VI (Lecture Notes in Computer Science, Vol. 15875)*. Springer, 58–77. https://doi.org/10.1007/978-981-96-8295-9_4

[39] Tim Hegeman, Animesh Trivedi, and Alexandru Iosup. 2020. Grade10: A Framework for Performance Characterization of Distributed Graph Processing. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 57–68.

[40] Farzin Houshmand, Mohsen Lesani, and Keval Vora. 2021. Grafs: declarative graph analytics. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–32. https://doi.org/10.1145/3473588

[41] Chengying Huan, Shuaiwen Leon Song, Santosh Pandey, Hang Liu, Yongchao Liu, Baptiste Lepers, Changhua He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2023. TEA: A General-Purpose Temporal Graph Random Walk Engine. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 182–198. https://doi.org/10.1145/3552326.3567491

[42] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 93:1–93:13. https://doi.org/10.1145/3581784.3607106

[43] Kasra Jamshidi, Harry Xu, and Keval Vora. 2023. Accelerating Graph Mining Systems with Subgraph Morphing. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 162–181. https://doi.org/10.1145/3552326.3567489

[44] Shinnung Jeong, Yongwoo Lee, Jaeho Lee, Heelim Choi, Seungbin Song, Jinho Lee, Youngsok Kim, and Hanjun Kim. 2022. Decoupling Schedule, Topology Layout, and Algorithm to Easily Enlarge the Tuning Space of GPU Graph Processing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022*. ACM, 198–210. https://doi.org/10.1145/3559009.3569686

[45] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. 2024. Core Graph: Exploiting edge centrality to speedup the evaluation of iterative graph queries. In *2024 Proceedings of the Nineteen European Conference on Computer Systems (EuroSys' 24)*.

[46] Xiaolin Jiang, Chengshuo Xu, and Rajiv Gupta. 2021. VRGQ: Evaluating a Stream of Iterative Graph Queries via Value Reuse. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 11–20. https://doi.org/10.1145/3469379.3469382

[47] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 17–32. https://doi.org/10.1145/3447786.3456226

[48] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[49] Zuhair Khayyat, Karim Awara, Amani AlOnazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. ACM, 169–182. https://doi.org/10.1145/2465351.2465369

[50] Juno Kim and Steven Swanson. 2022. Blaze: Fast Graph Processing on Fast SSDs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*. IEEE, 44:1–44:15. https://doi.org/10.1109/SC41404.2022.00049

[51] Pradeep Kumar and H Howie Huang. 2016. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 830–841.

[52] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[53] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[54] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112. https://doi.org/10.14778/3339490.3339494

[55] Ronny Lempel and Shlomo Moran. 2001. SALSA: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)* 19, 2 (2001), 131–160.

[56] Longzhuang Li, Yi Shang, and Wei Zhang. 2002. Improvement of HITS-based algorithms on web documents. In *Proceedings of the 11th international conference on World Wide Web*. 527–535.

[57] Xue Li, Mingxing Zhang, Kang Chen, and Yongwei Wu. 2018. ReGraph: A Graph Processing Framework that Alternately Shrinks and Repartitions the Graph. In *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*. ACM, 172–183. https://doi.org/10.1145/3205289.3205292

[58] Xiaofei Liao, Jin Zhao, Yu Zhang, Bingsheng He, Ligang He, Hai Jin, and Lin Gu. 2022. A Structure-Aware Storage Optimization for Out-of-Core Concurrent Graph Processing. *IEEE Trans. Computers* 71, 7 (2022), 1612–1625. https://doi.org/10.1109/TC.2021.3098976

[59] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies*. 285–300.

[60] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*. USENIX Association, 285–300. https://www.usenix.org/conference/fast17/technical-sessions/presentation/liu

[61] Renjie Liu, Yichuan Wang, Xiao Yan, Haitian Jiang, Zhenkun Cai, Minjie Wang, Bo Tang, and Jinyang Li. 2025. DiskGNN: Bridging I/O efficiency and model accuracy for out-of-core GNN training. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[62] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[63] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*. 527–543.

[64] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[65] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 25:1–25:16.

[66] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan Nicholas Yzelman. 2023. Design and Implementation for Nonblocking Execution in GraphBLAS: Tradeoffs and Performance. *ACM Trans. Archit. Code Optim.* 20, 1 (2023), 6:1–6:23. https://doi.org/10.1145/3561652

[67] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) *(HOTOS'15)*. USENIX Association, USA, 14.

[68] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2025. Revisiting Graph Analytics Benchmark. *Proc. ACM Manag. Data* 3, 3 (2025), 208:1–208:28. https://doi.org/10.1145/3725345

[69] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2025. A Survey of Distributed Graph Algorithms on Massive Graphs. *ACM Comput. Surv.* 57, 2 (2025), 27:1–27:39. https://doi.org/10.1145/3694966

[70] S. Deepak Narayanan, Aditya Sinha, Prateek Jain, Purushottam Kar, and Sundararajan Sellamanickam. 2022. IGLU: Efficient GCN Training via Lazy Updates. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=5kq11Tl1z4

[71] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: concurrency-aware graph processing on SSDs. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[72] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[73] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer science review* 1, 1 (2007), 27–64.

[74] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. 2023. Bridging the Gap between Relational OLTP and Graph-based OLAP. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 181–196. https://www.usenix.org/conference/atc23/presentation/shen

[75] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 317–332.

[76] Jixian Su, Chiyu Hao, Shixuan Sun, Hao Zhang, Sen Gao, Jiaxin Jiang, Yao Chen, Chenyi Zhang, Bingsheng He, and Minyi Guo. 2025. Revisiting the Design of In-Memory Dynamic Graph Storage. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[77] Dahai Tang, Jiali Wang, Rong Chen, Lei Wang, Wenyuan Yu, Jingren Zhou, and Kenli Li. 2024. XGNN: Boosting Multi-GPU GNN Training via Global GNN Memory Store. *Proc. VLDB Endow.* 17, 5 (2024), 1105–1118. https://www.vldb.org/pvldb/vol17/p1105-chen.pdf

[78] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[79] Stephen Tomlinson. 2003. Robust, Web and Genomic Retrieval with Hummingbird SearchServer at TREC 2003.. In *TREC*. 254–267.

[80] Alok Tripathy, Katherine A. Yelick, and Aydin Buluç. 2020. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 70. https://doi.org/10.1109/SC41405.2020.00074

[81] Alexander van der Grinten, Elisabetta Bergamini, Oded Green, David A. Bader, and Henning Meyerhenke. 2022. Scalable Katz Ranking Computation in Large Static and Dynamic Graphs. *ACM J. Exp. Algorithmics* 27 (2022), 1.7:1–1.7:16. https://doi.org/10.1145/3524615

[82] Pourya Vaziri and Keval Vora. 2021. Controlling Memory Footprint of Stateful Streaming Graph Processing. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 269–283. https://www.usenix.org/conference/atc21/presentation/vaziri

[83] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference*. 429–442.

[84] Keval Vora, Rajiv Gupta, and Guoqing (Harry) Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 237–251.

[85] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference*.

[86] Binghui Wang, Ang Li, Meng Pang, Hai Li, and Yiran Chen. 2022. GraphFL: A Federated Learning Framework for Semi-Supervised Node Classification on Graphs. In *IEEE International Conference on Data Mining, ICDM 2022, Orlando, FL, USA, November 28 - Dec. 1, 2022*. IEEE, 498–507. https://doi.org/10.1109/ICDM54844.2022.00060

[87] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. {GraphQ}: Graph Query Processing with Abstraction {Refinement—Scalable} and Programmable Analytics over Very Large Graphs on a Single {PC}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 387–401.

[88] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin J. Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 779–795. https://www.usenix.org/conference/osdi23/presentation/wang-yuke

[89] Yiqi Wang, Long Yuan, Zi Chen, Wenjie Zhang, Xuemin Lin, and Qing Liu. 2023. Towards Efficient Shortest Path Counting on Billion-Scale Graphs. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2579–2592. https://doi.org/10.1109/ICDE55515.2023.00198

[90] Jingqi Wu, Rong Chen, and Yubin Xia. 2021. Fast and Accurate Optimizer for Query Processing over Knowledge Graphs. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*. ACM, 503–517. https://doi.org/10.1145/3472883.3486991

[91] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 587–600.

[92] Xianghao Xu, Fang Wang, Hong Jiang, Yongli Cheng, Yu Hua, Dan Feng, and Yongxuan Zhang. 2023. LOSC: A locality-optimized subgraph construction scheme for out-of-core graph processing. *J. Parallel Distributed Comput.* 172 (2023), 51–68. https://doi.org/10.1016/J.JPDC.2022.10.005

[93] Carl Yang, Aydin Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1 (2022), 1:1–1:51. https://doi.org/10.1145/3466795

[94] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 417–434. https://doi.org/10.1145/3492321.3519557

[95] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. 2023. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 103–117. https://doi.org/10.1145/3575693.3575725

[96] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2024. Seraph: Towards scalable and efficient fully-external graph computation via on-demand processing. In *22nd USENIX Conference on File and Storage Technologies*

(FAST 24). 373–387.

[97] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2049–2062. https://doi.org/10.1145/3448016.3457237

[98] Zihang Yao, Rong Chen, Binyu Zang, and Haibo Chen. 2022. Wukong+G: Fast and Concurrent RDF Query Processing Using RDMA-Assisted GPU Graph Exploration. *IEEE Trans. Parallel Distributed Syst.* 33, 7 (2022), 1619–1635. https://doi.org/10.1109/TPDS.2021.3121568

[99] Long Yuan, Xia Li, Zi Chen, Xuemin Lin, Xiang Zhao, and Wenjie Zhang. 2024. I/O Efficient Label-Constrained Reachability Queries in Large Graphs. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2590–2602.

[100] Long Yuan, Zeyu Zhou, Zi Chen, Xuemin Lin, Xiang Zhao, and Fan Zhang. 2025. $\text {GPUSCAN}ˆ{++}$: Efficient Structural Graph Clustering on GPUs. *IEEE Trans. Parallel Distributed Syst.* 36, 9 (2025), 1890–1903. https://doi.org/10.1109/TPDS.2025.3582996

[101] Heng Zhang, Lingda Li, Hang Liu, Donglin Zhuang, Rui Liu, Chengying Huan, Shuang Song, Dingwen Tao, Yongchao Liu, Charles He, Yanjun Wu, and Shuaiwen Leon Song. 2022. Bring orders into uncertainty: enabling efficient uncertain graph processing via novel path sampling on multi-accelerator systems. In *2022 International Conference on Supercomputing*. 11:1–11:14.

[102] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 285–300.

[103] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 608–621.

[104] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *IEEE International Symposium on High Performance Computer Architecture*. 544–557.

[105] Wei Zhang, Yong Chen, and Dong Dai. 2018. AKIN: A Streaming Graph Partitioning Algorithm for Distributed Graph Storage Systems. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. IEEE Computer Society, 183–192. https://doi.org/10.1109/CCGRID.2018.00033

[106] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A dual-cache training system for graph neural networks on giant graphs with the GPU. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[107] Yongxuan Zhang, Hong Jiang, Fang Wang, Yu Hua, Dan Feng, Yongli Cheng, Yuchong Hu, and Renzhi Xiao. 2021. CIC-PIM: Trading spare computing power for memory space in graph processing. *J. Parallel Distributed Comput.* 147 (2021), 152–165. https://doi.org/10.1016/j.jpdc.2020.09.008

[108] Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. USENIX Association, 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

[109] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2023. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *Proceedings of the VLDB Endowment* 17, 4 (2023), 891–903.

[110] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 301–316.

[111] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference*. 375–386.

[112] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 712–725. https://doi.org/10.1145/3352460.3358256

Received April 2025; revised July 2025; accepted August 2025