

Break Iteration Barrier: Parallelize Priority-Based Graph Processing

SIYI TENG, The Chinese University of Hong Kong, China

JEFFREY XU YU, The Hong Kong University of Science and Technology (Guangzhou), China and The Chinese University of Hong Kong, China

Many graph processing systems and graph libraries have been developed to process and analyze graph data efficiently. Among the built-in graph algorithms, priority-based graph algorithms, such as Dijkstra's algorithm and greedy algorithms for combination optimization problems, e.g., influence maximization problem, represent a significant category. They can hardly achieve higher efficiency by parallelism due to their inherent iterative dependencies in the priority queue used. Existing parallelization techniques, e.g., parallel priority queues, struggle to preserve the original processing order of these algorithms. They require larger theoretical time complexities or lose the theoretical guarantees associated with these algorithms. To address these issues, in this paper, we propose PQ^+ , a priority queue designed to facilitate parallel execution of priority-based graph processing algorithms without altering their inherent process order and preserving the same theoretical time complexity. Extensive experiments on ten real-world datasets across four representative graph processing algorithms validate the effectiveness and efficiency of our proposed approach.

CCS Concepts: • **Theory of computation** → **Data structures design and analysis**; **Parallel algorithms**; **Graph algorithms analysis**.

Additional Key Words and Phrases: priority-based graph processing, priority queues, parallel graph algorithms

ACM Reference Format:

Siyi Teng and Jeffrey Xu Yu. 2026. Break Iteration Barrier: Parallelize Priority-Based Graph Processing. *Proc. ACM Manag. Data* 4, 4 (SIGMOD), Article 261 (September 2026), 26 pages. <https://doi.org/10.1145/3837099>

1 Introduction

Graph processing is a cornerstone of modern computing systems, powering applications in social networks, knowledge graphs, business transactions, and artificial intelligence. Nowadays, real-world graphs, such as the Facebook social network, with its billions of vertices and trillions of edges, display massive scale and irregular structures, for example, power-law degree distributions among vertices [10]. It poses a serious challenge for efficiently processing large-scale graphs. Among graph processing algorithms, priority-based graph algorithms represent a significant category. These include Dijkstra's algorithm for the single-source shortest path (SSSP) problem [8], Prim's algorithm for the minimum spanning tree problem [35], IMM for influence maximization problem [14, 38, 39], along with a broad family of greedy algorithms used to solve diverse problems of graph processing [15, 28, 43, 46]. The efficiency of such graph algorithms relies heavily on the efficiency of the priority queues [13, 20]. However, the inherently sequential nature of priority queue operations limits parallelism and hinders the utilization of available computational power.

Authors' Contact Information: Siyi Teng, The Chinese University of Hong Kong, Hong Kong, China, [syten@se.cuhk.edu.hk](mailto:syteng@se.cuhk.edu.hk); Jeffrey Xu Yu, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China and The Chinese University of Hong Kong, Hong Kong, China, jxyu@cuhk.edu.hk.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART261

<https://doi.org/10.1145/3837099>

Priority-based graph algorithms operate iteratively on a graph, following a unique global progress order established by a priority queue. Specifically, in each iteration, the vertex with the highest priority is popped from the queue and utilized to update the priorities of other vertices based on the graph's topological structure, such as updating its neighbors, and then pushing these updates into the priority queue. Formally, in each iteration, it involved one pop operation and multiple push operations. For example, in Dijkstra's algorithm, the current distance from the source vertex s serves as the priority for each vertex. Every iteration involves popping a vertex with the shortest distance from s from the priority queue and using it to update the priorities of its neighbors, i.e., updating their distances from s . In the realm of combination optimization problems within graph processing, the marginal income is employed as the priority for each vertex. During each iteration, the algorithm selects the highest priority element from the queue. The inherent sequential nature of these priority-based graph processing algorithms poses challenges for parallelization due to the interdependencies between iterations. Each update in an iteration relies on the vertex popped from the queue, and the popped vertex in the next iteration depends on the updates made in the preceding iteration. The interdependencies between iterations are fundamental for maintaining the efficiency and accuracy of these algorithms. Firstly, selecting the highest priority element from queue in each iteration minimizes the total number of iterations required for achieving their theoretical time complexity [13, 20]. While it may be acceptable for a specific problem, e.g. Delta-stepping algorithm for SSSP problem [32], not to select the topmost element from the queue in every iteration, a vertex might need to be popped multiple times, resulting in a worse theoretical time complexity, which can hinder their ability to outperform sequential algorithms in certain cases, e.g., fewer threads are available. Secondly, ensuring the pop of the highest priority element guarantees correctness or provides result guarantees for problems. For instance, in combination optimization problems, such a selection can yield a $(1 - 1/e - \epsilon)$ -approximation when maximizing a submodular objective function or minimizing a supermodular objective function [23, 33]. Hence, while specific algorithms have seen proposals for parallel acceleration versions, these adaptations struggle to be generalized across other priority-based algorithms. This difficulty arises from the fact that such parallel versions disrupt the interdependencies between iterations, which are typically based on the unique properties of specific problems.

In this paper, we explore a novel way to parallelize while maintaining the inherent interdependencies between iterations for processing priority-based graph algorithms. Various existing approaches cater to parallel priority queues, with two state-of-the-art solutions. One is an industry-ready version from Intel TBB [19], while the other is an academically designed variant Skiplist [3, 17, 27, 36]. Both of these methods facilitate simultaneous pushing and popping of the queue elements. However, to ensure high efficiency, they lose the guarantee of exact top-1 element retrieval in pop operation. Hence, they still can not guarantee the interdependencies between iterations. Even with a guarantee on pop operation, given the power-law degree distributions among vertices in real-world graphs [10], solely parallelizing push operations in each iteration remains inefficient. This inefficiency arises because most vertices in the graph possess lower degrees, resulting in push operations in the majority of iterations falling below the batch size required for optimal batch push optimizations. Thus, the costs associated with parallel processing, e.g., synchronization, outweigh the time saved from parallelization, leading to suboptimal performance compared to sequential implementations in most scenarios.

In this paper, we design a priority queue PQ^+ for parallel priority-based graph processing. The main contributions of this work are summarized below. ❶ To ensure exact top-1 retrieval for pop operation in PQ^+ while enabling parallel execution, we introduce a two-queue structure. One queue is the exact priority queue S , ensuring that the top-1 element is always popped from it, while another queue P ensures the top- K elements are always popped from it. The top- K elements in PQ^+

are stored in S , with P serving as a backup containing remain elements to refill S when it becomes empty. This separated structure lays the foundation for further coordination and parallel execution mechanisms. ② Our main technical contribution is a set of coordination mechanisms between S and P that make this design both correct and efficient. Specifically, we decompose the operations in queues S and P into fine-grained actions that can be processed in parallel. (i) For pop operation in PQ^+ , most of the time, it directly pops from S swiftly. Occasionally, it pops from P , which involves returning K elements immediately to S and then adjusting P concurrently, ensuring that pop operation in PQ^+ does not need to wait for adjustments in P . (ii) For push operation in PQ^+ , as it always inserts directly into P , we design a staging area for it. This setup allows most push operations to be pushed into the staging area in a short time. Occasionally, elements in the staging area are pushed to P . Necessary adjustments are swiftly made to ensure the subsequent top- K elements popped from P , while other adjustments in P can be carried out in parallel. ③ We prove that when utilizing our priority queue PQ^+ in priority-based graph processing, it preserves the theoretical time complexity comparable to its sequential counterpart when operating in sequential execution mode. ④ Extensive experiments are conducted on 10 real-world datasets using 4 fundamental graph processing algorithms. The results demonstrate that our approach PQ^+ outperforms all existing methods, achieving up to 3.11x faster running times while maintaining the original processing order of these graph algorithms.

The article is organized as follows. We give the preliminaries in Section 2, and discuss the issues of existing approaches in Section 3. We provide the design of our two-queue approach in Section 4, and give the implementation details in Section 5. We conduct extensive experiments to confirm the effectiveness and efficiency of our approach in Section 6. We discuss related work in Section 7 and conclude this work in Section 8.

2 Preliminary and Problem Statement

In this paper, we focus on a simple weighted graph $G = (V, E)$, where V and E represent the set of vertices and edges in G , respectively. In the following, we also use $V(G)$ and $E(G)$ to denote the set of vertices and edges of a graph G , and n and m to denote the number of vertices and edges, i.e., $n = |V(G)|$ and $m = |E(G)|$. For each edge $e = (v, u) \in E$, we use $w(v, u)$ to denote its weight.

Priority Queue: A priority queue Q is a data structure supporting retrieval of the top-1 element according a definable priority order. It supports three core operations: push, pop, and empty.

- (1) $\text{push}(e)$: insert an element e into Q , and adjust Q to be ready for next operation.
- (2) $\text{pop}()$: find and return the element of highest priority in Q , delete it, and adjust Q to be ready for next operation.
- (3) $\text{empty}()$: return whether Q is empty.

Priority-Based Graph Algorithms: A priority-based graph algorithm is a computation process that executes iteratively on a graph G with a unique global progress order determined by a priority queue. Specifically, we use $\circ$ to indicate the sequential execution order between operations. A priority-based algorithm over n iterations can be modeled as:

$$\bigcirc_{i=1}^n \left(\text{pop} \circ (\text{push})^{k_i} \right) \quad (1)$$

where the i -th iteration performs one pop operation followed by k_i push operations. This model can be easily extended to allow multiple pop or $k_i = 0$ within an iteration. Such an execution order is sequential. Below, we give an example for priority-based algorithms using a priority queue $Q : \langle v, p(v) \rangle$, where v is a vertex in the graph, and $p(v)$ denotes the priority of vertex v .

Dijkstra's algorithm [8] is for the single-source shortest path (SSSP) problem. This algorithm stands out as the best in terms of theoretical time complexity among sequential algorithms for

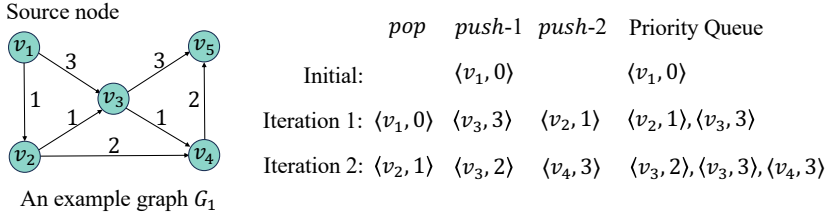


Fig. 1. An example graph G_1 and the first 2 iterations of Dijkstra's algorithm on it from source vertex v_1

SSSP problem, and based on it, extensive studies have been conducted over large graphs [9, 45, 47]. Given a graph $G = (V, E)$ and source vertex $s \in V$, the SSSP problem aims to find the shortest paths from s to all vertices $v \in V \setminus \{s\}$. In Dijkstra's algorithm, the priority $p(v)$ of each vertex v is the minimum weight of current reachable paths from s to v , originally set as $+\infty$ for $v \in V \setminus \{s\}$ and 0 for s . Initially, $\langle s, p(s) \rangle$ is pushed into Q . Next, in each iteration, a v with smallest $p(v)$ is popped by Q , and used to update priority for its neighbors u , i.e., $p'(u) = \min(p(v) + w(v, u), p(u))$ for each edge $(v, u) \in E$, and $\langle u, p'(u) \rangle$ is pushed into Q if $p'(u) < p(u)$. The algorithm terminates when Q is empty. Except Dijkstra's algorithm, many other graph processing algorithms also adhere to this execution pattern, e.g., [2, 15, 16, 22, 24, 28, 29, 35, 39, 42].

The pattern illustrated by the Dijkstra's algorithm is inherently following Eq. (1). Specifically, for any two operations in the iterations, their relative order is subject to iteration-dependent constraints. In the same i -th iteration, pop must execute before all push; and the pop in the i -th iteration must execute after all push in the $(i - 1)$ -th iteration. For push operations in the same iteration, there is no execution order constraint.

Example 2.1: Fig. 1 illustrates an example graph G_1 and the first two iterations of Dijkstra's algorithm executed on G_1 from source vertex v_1 . Initially, $\langle v_1, 0 \rangle$ is pushed into the queue Q . In the first iteration, $\langle v_1, 0 \rangle$ is popped, triggering priority updates on its neighbors v_2 and v_3 . Consequently, $\langle v_3, 3 \rangle$ and $\langle v_2, 1 \rangle$ are pushed into the queue. Note that if $\langle v_1, 0 \rangle$ is not popped first, the algorithm can not identify which neighbors require priority updates, and thus can not push the corresponding neighbors into the queue. This establishes the execution order between pop and push within an iteration, such that the pop must precede any push operations.

In the second iteration, the current top-1 element, $\langle v_2, 1 \rangle$, is popped. This pop must occur after $\langle v_2, 1 \rangle$ is pushed in the first iteration. Otherwise, if pop were to happen before that push, then $\langle v_3, 3 \rangle$ would become the top-1 element and be incorrectly popped instead. This demonstrates that the pop of the current iteration must follow all push operations from the previous iteration.

This strict execution order poses challenges to accelerating such graph algorithms, since parallelism can hardly be conducted. Below, for simplicity, we assume all elements in the priority queue have distinct priorities.

The Problem Statement: We aim at making all graph algorithms efficient if they are implemented by a priority queue by achieving three key objectives: (i) guarantees strict top-1 element retrieval in pop, (ii) preserving theoretical time complexity comparable to its classical sequential counterpart when operating in sequential execution mode, and (iii) enables parallel acceleration of algorithms.

3 The Issues of Existing Approaches

In this section, we first discuss existing works on concurrent priority queues. Next, we analyze the issues of applying these priority queues to priority-based graph algorithms.

Table 1. Comparison of Dijkstra's execution on G_1

Iteration	Exact Priority Queue		Relaxed Priority Queue	
	Queue State	Pushed	Queue State	Pushed
(1)	$(v_1, 0)$	$(v_2, 1), (v_3, 3)$	$(v_1, 0)$	$(v_2, 1), (v_3, 3)$
(2)	$(v_2, 1), (v_3, 3)$	$(v_3, 2), (v_4, 3)$	$(v_2, 1), (v_3, 3)$	$(v_4, 4), (v_5, 6)$
(3)	$(v_3, 2), (v_3, 3), (v_4, 3)$	$(v_5, 5)$	$(v_2, 1), (v_4, 4), (v_5, 6)$	$(v_3, 2), (v_4, 3)$
(4)	$(v_3, 3), (v_4, 3), (v_5, 5)$	None	$(v_3, 2), (v_4, 3), (v_4, 4), (v_5, 6)$	$(v_5, 5)$
(5)	$(v_4, 3), (v_5, 5)$	None	$(v_4, 3), (v_4, 4), (v_5, 5), (v_5, 6)$	None
(6)	$(v_5, 5)$	None	$(v_4, 4), (v_5, 5), (v_5, 6)$	None
(7)	$\emptyset$		$(v_5, 5), (v_5, 6)$	None
(8)			$(v_5, 6)$	None
(9)			$\emptyset$	

Table 2. Extra iterations in Dijkstra's algorithm with Spraylist (#thread=16).

dataset	n	avg_deg	#extra iterations
Wiki-talk	2,394,385	2.10	202
Google	875,713	5.83	1,380
Amazon	3,683,637	7.66	1,353
Topcats	1,791,489	15.92	1,187
USA	23,947,347	2.44	56,766
Livejournal	4,847,571	14.23	1,293
Dewiki	2,166,669	39.85	82,437
Yahoo	1,625,951	157.94	2,038

3.1 Existing Priority Queue Approaches

There are open-source libraries and researches that provide many implementations of parallel priority queues. We discuss two state-of-the-art among them: an industry-ready version from Intel TBB, and an academically designed version optimized for high-performance operations. The two baselines correspond to practical industrial requirements and rigorous research objectives, respectively, and are built upon two widely studied data structures in the field of priority queues: the heap and the skiplist.

TBB [19]: The Intel TBB library provides a concurrent priority queue, which allows multiple threads to concurrently push and pop queue items. Its underlying implementation uses an STL vector container to maintain a binary heap structure, which is not lock-free. It improves efficiency by doing the following optimizations. First, it allows for a shuffling of the order of operations that occur nearly simultaneously, prioritizing the operations that are easier to execute. Second, it leverages an unordered data structure S containing unheapified elements to accelerate push, while for pop, it returns an approximate top-1 element by comparing only the top of the heap and the last element of S . Such a design makes it a relaxed priority queue, i.e., it cannot guarantee the retrieval of the exact top-1 element in every pop operation.

Skiplist [3, 17, 27, 36]: Skiplist is another parallel priority queue approach, which is composed of several linked lists organized in multiple levels. It is first proposed in [36] and implemented as a lock-free priority queue [17]. [27] presents an exact priority queue design which focuses on improving the performance of conducting simultaneous pop operations. [3] proposes the state-of-the-art work of implementing a skiplist-based lock-free priority queue with high scalability, SprayList. As a relaxed priority queue, SprayList guarantees the retrieval of the top- $O(p \log^3 p)$ element in pop operation instead of guaranteeing top-1, where p is the number of threads. The experimental results in [3] indicate that although SprayList relaxes the constraint to achieve the top-1 element in the pop operation, it outperforms its baselines, e.g., [27], on graph algorithms like Dijkstra's algorithm in practice.

3.2 Issues of Existing Priority Queues

While high-performance priority queues have been proposed, their effectiveness in accelerating priority-based graph algorithms remains limited due to fundamental mismatches in design requirements and operational patterns, as discussed below.

Exactness Issue: As stated above, neither of the two existing approaches can ensure correct retrieval of the top-1 element, since they are designed to handle multiple push and pop operations simultaneously, thereby trading off this guarantee. However, in priority-based graph algorithms, each iteration requires exactly the highest-priority element, which makes the top-1 element of

crucial importance for priority-based graph algorithms. Specifically, the relaxation of execution order impacts algorithms in two ways.

Firstly, such relaxation affects the efficiency of algorithms. We show it using Dijkstra's algorithm.

Example 3.1: We illustrate the impact of relaxing the priority order using the example graph G_1 and Dijkstra's algorithm as a concrete case. Starting from source vertex v_1 , the elements held in the priority queue and updated elements at each iteration are listed in Table 1. Note that, as a common implementation, when the updated distance of v is larger than or equal to the current distance of v , it will not be added into the priority queue. Elements in the priority queue are listed according to their priority, i.e., a smaller value of distance indicates a higher priority. In each iteration, the exact priority queue pops the first element, while the relaxed priority queue may pop the second element $\langle v_3, 3 \rangle$ in iteration (2). The relaxed priority queue only fails to pop the top-1 element in one iteration, but degrades the performance of the algorithm noticeably. While the exact version terminates in 6 iteration, the relaxed version terminates after 8 iterations.

This example illustrates that the algorithm's correctness may remain intact as additional iterations and operations can be employed to ensure convergence. However, extra iterations result in heightened time complexity compared to the sequential algorithm. Thus, priority violations can significantly elevate the overall number of operations, consequently impacting the efficiency of algorithms. In Table 2, we report the number of extra iterations in Dijkstra's algorithm on 8 datasets using priority queue implemented by SprayList. The thread number is set as 16, which is a representative configuration from the range in which SprayList generally achieves its strongest performance across the evaluated graph algorithms. We also report the average degree of each dataset as a coarse structural indicator of the overhead of these extra iterations, since each extra iteration triggers one additional pop and further neighbor processing on the popped vertex.

Secondly, for many priority-based graph algorithms, relaxing the order can directly contravene the theoretical underpinnings that ensure solution quality. A prominent example is employing the greedy algorithm in problems aiming to maximize a *submodular* objective function or minimize a *supermodular* objective function under cardinality constraints, e.g., [24, 28, 29, 38, 39, 43]. The utilization of the greedy algorithm with a priority queue can achieve an $(1 - 1/e - \epsilon)$ -approximation guarantee, but this guarantee hinges crucially on the exact sequential order of greedy selections [18, 23, 33]. Any accelerated versions of priority queues that break this order in pursuit of efficiency cannot preserve the same theoretical bounds on these problems.

Workload Issue: The optimizations for conducting multiple push and pop simultaneously and efficiently, proposed by existing approaches of priority queues, would be ill-suited for priority-based graph algorithms. On one hand, the sequential priority-driven pattern cannot exploit optimizations for batch pop, because it only extracts one element per iteration. On the other hand, optimizations for batch push can be leveraged within an iteration, but need synchronization before the next iteration begins, i.e., next iteration's pop should wait for the completion of all push in the last iteration. It limits the utilization of batch push performance, and introduces synchronization overhead. Moreover, real-world graphs, such as social networks, web networks, and road networks, typically demonstrate highly skewed degree distributions following the power law. Consequently, in most iterations of a priority-based graph algorithm, if a vertex has a low degree, a small number of push operations are needed to push the updated priority of its neighbors. In such cases, the push operations in most iterations are far below the batch size needed to fully leverage batch push optimizations. This situation leads to the cost incurred from parallel processing, e.g., synchronization, outweighing the time savings from parallel processing, resulting in these existing approaches being even slower than sequential implementations in most cases, as illustrated in Exp. 1 in Section 6.

The two issues highlight that there exist mismatches between the existing design of high-performance priority queues and the requirements of graph algorithms. Such mismatches, as illustrated by experimental results in Section 6, lead to inefficiency compared to even the sequential version of graph algorithms. An effective design of a priority queue for graph algorithms must align with the strong requirement of exactness, while maintaining high-performance.

4 A Novel Two-Queue Approach: PQ^+

In this section, we present an overview of our approach, followed by its internal parallelism mechanisms and optimization techniques.

4.1 An Overview

We propose a new two-queue approach, named PQ^+ , to achieve the three key objectives. There are two priority queues inside PQ^+ , namely an exact priority queue S , which guarantees exact top-1 element retrieval, and a relaxed priority queue P , which pops unordered top- K exactly where $K > 1$. Specifically, elements in P are stored in a relaxed order, i.e., it only guarantees the K elements at one pop are the exact top- K elements, but does not guarantee the exact order among the K elements being popped.

Below, for simplicity, we assume that a smaller value of $p(v)$ indicates a higher priority. The elements kept in S is of higher priority than elements kept in P , i.e., $\max_{\langle v, p(v) \rangle \in S} p(v) \leq \min_{\langle v, p(v) \rangle \in P} p(v)$. PQ^+ will maintain this **core invariant** to provide a guarantee for the exact top-1 retrieval, i.e., if S is not empty, the top-1 element exists in S . Thus, each pop will first pop an element in S . For each push, it first tries to add a new element into S if the core invariant is maintained. Formally, a new element $\langle v, p(v) \rangle$ can be added to S if the following **check** $\mathcal{A}(\langle v, p(v) \rangle, S)$ holds, i.e., $\mathcal{A}(\langle v, p(v) \rangle, S) : p(v) < \max_{\langle u, p(u) \rangle \in S} p(u)$.

Since for each pop or push, S is needed, to improve the efficiency, we restrict the size of S as a user-defined parameter K , and the setting of K will be discussed later in Section 5.2. S contains at most K elements, much fewer than the number of total elements that can reside in the priority queue. Given the priority is updated from neighboring vertices, the queue may contain up to $O(m)$ elements. The other elements are maintained in P , kept in a relaxed priority order. P supports pushing a single element, and popping top- K elements from P in order to push the top- K elements into S when S is empty, where such pushing the top- K into S is done by an internal refill operation. Such a design of P focuses on maintaining the size of S between 0 and K , while maintaining the core invariant. Specifically, the size of S has two states that require adjustment. (1) $|S| = 0$: Since in each iteration, pop removes one elements from S , and possibly there is no new elements inserted into S due to failure to pass check $\mathcal{A}$, S would become empty. In such a case, we need to refill S with the current top- K elements in P . (2) $|S| > K$: If a newly pushed element passes $\mathcal{A}$ and is inserted to S , leading to the size of S becomes $K + 1$, it needs further adjustments to maintain its maximum size. Specifically, the element $\langle v, p(v) \rangle = \arg \max_{\langle u, p(u) \rangle \in S} p(u)$ will be removed from S , and inserted into P .

The above two adjustments of S supported by P both keep the core invariant. We next formally claim the equivalence between the original push and pop operations of a single priority queue and the collaboration between S and P . The empty operation, given its simplicity, i.e., checking whether S and P are both empty, is excluded from the subsequent descriptions for brevity. First, we denote operations on S and P as follows.

- (1) $\text{push}_S(e)$: insert an element e into S , and adjust S for next operation.
- (2) $\text{pop}_S()$: find and return the top-1 (the highest priority) element in S , delete it, and adjust S for next operation.

- (3) $\text{low}_S()$: find and return the element with the lowest priority in S , delete it, and adjust S for next operation.
- (4) $\text{refill}_S(E)$: refill the empty S with a size- K element set E . Semantically, this operation is equivalent to conduct $\text{push}_S(e)$ for each $e \in E$, while for simplicity and functional differentiation, we use a different operation here.
- (5) $\text{push}_P(e)$: insert an element e into P , and adjust P for next operation.
- (6) $\text{pop}_P()$: find and return K elements with the highest priority in P , delete them, and adjust P for next operation.

The push and pop operations for a priority queue given in Section 2 are semantically equal to the following operations on S and P with branching logic:

- (1) $\text{push}(e) \equiv$

$$\begin{cases} (a) \text{push}_S(e), & \text{if } \mathcal{A}(e, S) \text{ and } |S| < K; \\ (b) \text{push}_S(e) \circ \text{low}_S() \circ \text{push}_P(e'), & \text{if } \mathcal{A}(e, S) \text{ and } |S| = K; \\ (c) \text{push}_P(e), & \text{otherwise.} \end{cases}$$
- (2) $\text{pop}() \equiv$

$$\begin{cases} (a) \text{pop}_P() \circ \text{refill}_S(E) \circ \text{pop}_S(), & \text{if } |S| = 0; \\ (b) \text{pop}_S(), & \text{otherwise.} \end{cases}$$

Here, e' in branch push (b) is returned by low_S , E in branch pop (a) is returned by pop_P . We next prove the correctness of our priority queue consisting of S and P , with push and pop following the above branching logic.

Lemma 4.1: *The two queue approach $Q = (S, P)$, equipped with push and pop operations defined above, correctly implements a priority queue, such that every pop operation returns the element with highest priority among all elements inserted by push but not yet popped.*

Proof Sketch: First, we claim the data integrity. For each element e pushed, according to the push operation logic, e resides in $S \cup P$ until it is popped by pop. Next, we prove the exactness of top-1 retrieval for pop. An element is removed from the queue only when it is the top-1 element in S . Given the core invariant holds for any logic branch, the top-1 element in S is the global top-1 element in $S \cup P$. Thus we prove the correctness of our priority queue. $\square$

The branching logic inside push and pop allows acceleration for four reasons. ① Operation branches that containing operations on only S , namely push (a) and pop (b), are inherently fast due to the small bounded size of S . ② Operation push_P involves push (b) and (c). We employ a lazy-update strategy that defers the actual push_P operation. The workload of push_P is accumulated and then processed via batched push operations into P , thereby lowering the amortized overhead. ③ Operation pop_P is used only in pop (a), a branch that is rarely invoked because of the boundary state of S , i.e., $|S| = 0$. If the priority queue holds $O(m)$ elements, pop (a) will be invoked at most $O(\frac{m}{K})$ times. To accelerate pop (a), i.e., the pop_P operation on P , we relax the internal ordering of P . Elements are maintained in a staged order rather than a globally strict order inside P . This relaxation still guarantees exact top- K retrieval when pop_P is invoked, while allowing more efficient operations on P .

In Section 5.2, we prove that our priority queue, when executed in sequential mode, provides the same time complexity with the traditional sequential one, using an example of an algorithm with n pop and m push operations. Next, we will demonstrate the acceleration mechanisms that utilizing parallelism.

4.2 Parallel Execution via Decoupled Pipelines

We have defined our pop and push operations based on S and P . In the best case, each pop/push operation is performed only on S , thus improving the time complexity from being dependent on m to being dependent on K . However, the best case may not always occur, and we need to deal with the heavy time cost when branches push (b), push (c), and pop (a) occur, introduced by operations on P . The key idea of our parallel design is to keep the order-sensitive operations on S in the main pipeline, executed sequentially, while performing the more expensive maintenance work on P in auxiliary pipelines parallelly whenever this preserves the core invariant. This design has two benefits. First, it directly addresses the exactness issue discussed in Section 3.2. Second, since the size of S is bounded by K , keeping operations on S sequential does not become the performance bottleneck. In contrast, the heavier work is concentrated on P , which can be hid from the main pipeline by parallelism.

Specifically, we separate operations on P from the main pipeline as follows. ① For branch push (c), the operation only involves push_P . Therefore, after the branch is determined, it can be removed from the main pipeline and executed in parallel with subsequent operations on S . ② For branch push (b), after $\text{push}_S \circ \text{low}_S$ produces the overflow element e' , the remaining operation $\text{push}_P(e')$ can be removed from the main pipeline and proceed in parallel with later operations on S . ③ For branch pop (a), pop_P first returns the top- K elements required by refill_S . After that, the remaining adjustment of P can be removed from the main pipeline and proceed in parallel with $\text{refill}_S \circ \text{pop}_S$. Therefore, for the above three branches, the cost of adjusting P thus can be removed from the main pipeline by parallelism, which accounts for the majority of the overhead.

As discussed above, the original single pipeline, Eq. (1), is separable. We separate it into multiple parallel pipelines, i.e., a main pipeline for operations on S , and several pipelines for operations on P . The main pipeline is always active and contains only the sequential operations on S . In contrast, whenever an operation on P is invoked, it is assigned to a temporary auxiliary pipeline that runs in parallel with the main pipeline. Formally, Eq. (1) with branches can be written as:

$$\odot_{i=1}^n \left(\left\{ \begin{array}{c} \text{pop}_S \\ \text{pop}_P \circ \text{refill}_S \circ \text{pop}_S \end{array} \right\} \circ \left(\left\{ \begin{array}{c} \text{push}_S \\ \text{push}_P \\ \text{push}_S \circ \text{low}_S \circ \text{push}_P \end{array} \right\} \right)^{k_i} \right)$$

where $\{\}$ indicates the different branches of pop and push. After decoupling this pipeline into multiple pipelines, the main pipeline, including only operations on S , is as follows:

$$\odot_{i=1}^n \left([\text{refill}_S] \circ \text{pop}_S \circ ([\text{push}_S] \circ [\text{low}_S])^{k_i} \right)$$

where $[op]$ indicates op is an optional operation decided by different branches. Note that push_S is optional in the main pipeline because of branch push (c). For operations on P , i.e., push_P and pop_P , they are assigned to auxiliary pipelines once invoked. Pipelines are executed parallelly under certain constraints.

Next, before discussing multi-pipeline parallel execution, we give four scheduling constraints for individual branches, or decoupled pipelines, to preserve the original sequential semantics under parallel execution, denoted by $\odot_1$, $\odot_2$, $\odot_3$, and $\odot_4$. $\odot_1$ (**branch-specific**): In branch push (b), push_P must execute only after $\text{push}_S \circ \text{low}_S$ in the main pipeline finishes, because the element e' pushed by push_P is generated by $\text{push}_S \circ \text{low}_S$. $\odot_2$ (**branch-specific**): In branch pop (a), $\text{refill}_S \circ \text{pop}_S$ in the main pipeline must execute after pop_P finishes, or at least after pop_P returns the top- K elements in P , because refilling S requires these top- K elements. $\odot_3$ (**across iterations**): In iteration i , if pop uses branch pop (a), then the corresponding pop_P in its auxiliary pipeline must wait until all operations on P from previous iterations in other auxiliary pipelines have finished. This constraint

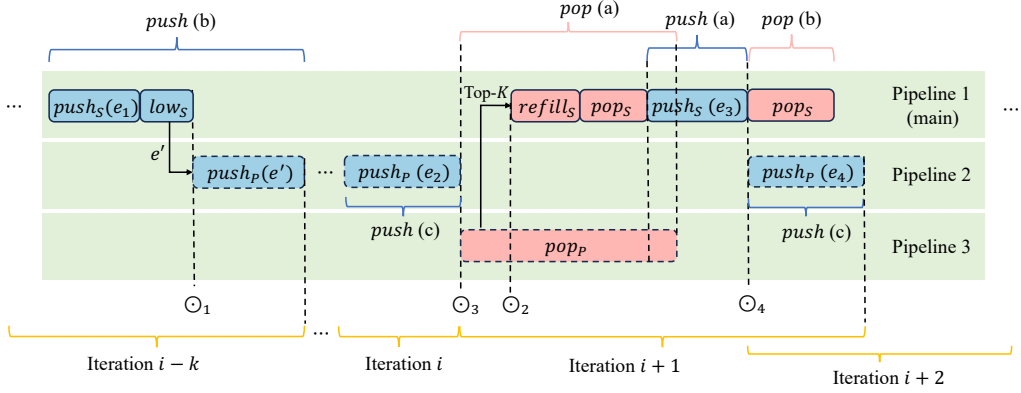


Fig. 2. Multiple pipelines under four constraints

guarantees exact top- K retrieval from P . $\odot_4$ (**branch selection**): Any branch decision for push operation must be made after all the previous operations on S finish, because the branch decision depends on evaluating $\mathcal{A}$ on the current state of S . $\odot_1, \odot_2, \odot_4$ are within-iteration constraints, while $\odot_3$ is an across-iteration constraint.

We use the following example to illustrate the multi-pipeline execution with scheduling constraints.

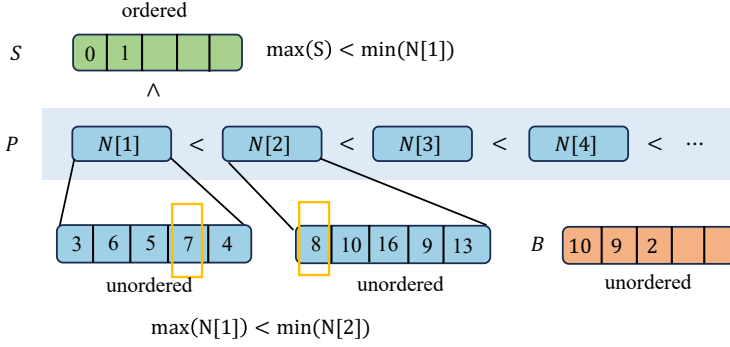
Example 4.1: Fig. 2 shows a complete iteration $i + 1$ and part of iteration $i - k$, i and $i + 2$. The iteration $i + 1$ contains one pop using branch pop (a), and two push using branch push (a) and push (c), pushes e_3 into S and e_4 into P , respectively. The pop in iteration $i + 1$ faces the situation that S is empty, thus pop_P and refill_S are called first. The pop_P operation is separated from the main pipeline into pipeline 3. The $\text{push}_P(e_4)$ operation is separated from the main pipeline into pipeline 2. The part of iteration $i - k$ contains a push operation using branch push (b), which pushes e_1 into S and causes $|S| > K$, thus further pushes e' into P . The $\text{push}_P(e')$ operation is separated from the main pipeline into pipeline 2. The scheduling constraints work as follows: First, for push in iteration $i - k$, $\odot_1$ indicates that $\text{push}_P(e')$ in pipeline 2 must execute after $\text{push}_S \circ \text{low}_S$ in pipeline 1 finishes. Second, $\odot_2$ indicates that, for pop in iteration $i + 1$, refill_S in pipeline 1 must execute after pop_P in pipeline 3 finishes, or at least after pop_P returns the top- K elements in P . Third, $\odot_3$ indicates that, between iteration i and $i + 1$, pop_P in pipeline 3 must execute after $\text{push}_P(e')$ in pipeline 2 finishes, otherwise pop_P cannot retrieve the correct top- K in P . Forth, $\odot_4$ indicates that, in iteration $i + 1$, $\text{push}_P(e_4)$ must execute after $\text{push}_S(e_3)$ finishes, otherwise $\mathcal{A}(e_4, S)$ cannot check the priority of e_3 correctly.

Note that each operation on P conceptually has its own auxiliary pipeline. For simplicity, Fig. 2 groups non-blocking operations on P into the same pipeline, i.e., pipeline 2, which is only a visualization choice and does not change the parallel execution semantics.

We next prove that under the four constraints, the parallel execution of multiple pipelines maintains the same operation ordering as a single sequential pipeline.

Theorem 4.1: *Under the four constraints, the parallel execution of the decoupled pipelines is equivalent to the original sequential execution.*

Proof Sketch: We prove the claim by induction on the iteration number. For the base case $i = 1$, the

Fig. 3. An overview of our priority queue when $K = 5$

first iteration contains one pop and k_1 push. For push/pop branches involving a single operation, i.e., push (a), push (c), and pop (b), the execution is identical to the sequential version. For branches involving multiple operations, namely push (b) and pop (a), $\odot_1$ and $\odot_2$ preserve their sequential semantics, respectively. Moreover, $\odot_4$ guarantees that the branch decision of each push is made from the correct state of S . Finally, the order between pop and the subsequent push is preserved by $\circ$. Hence, the first iteration is equivalent to the sequential execution.

Assume that the first $i - 1$ iterations are equivalent to the sequential execution. Inside iteration i , the sequential semantics is preserved for the same reasons as in the base case. Across iterations, $\odot_3$ ensures that, if the pop of iteration i needs to access P , then all operations on P from previous iterations that may affect the top- K result have already finished. Together with the sequential main pipeline, this guarantees that iteration i observes the same states of both S and P as in the sequential execution. Therefore, iteration i takes the same pop branch, returns the same result, and each subsequent push makes the same branch decision as in the sequential execution. Thus, the first i iterations are equivalent to the sequential execution. By induction, the parallel execution of the decoupled pipelines is equivalent to the sequential version. $\square$

The execution of multiple pipelines benefits efficiency for the following two reasons. ❶ The decoupled and parallel pipelines break the strict execution order between operations mentioned in Section 2. Originally, within an iteration, any push must execute after pop, while we break the barrier by leveraging branch pop (a). As shown by Fig. 2, the execution of $\text{push}_S(e_3)$ in pipeline 1 overlaps with the execution of pop_P in pipeline 3. The push operation actually executes before pop finishes. We also break the barrier between iterations by leveraging branches operated on different structures, e.g., the execution of pop_S in iteration $i + 2$, pipeline 1, overlaps with $\text{push}_P(e_4)$ in iteration $i + 1$, pipeline 3. ❷ Such decoupled pipelines allow further optimizations on pop_P and push_P , including finer-grained decomposing of pop_P , and lazy-update strategy of push_P , and we will detail them in next section. Overall, the parallelism of PQ^+ comes from two complementary sources. The first is the parallelism between S and P , as discussed in this section, and the second is the internal parallelism within P , to be presented in the next section. Together, these two aspects constitute the overall parallelism of PQ^+ .

4.3 Optimization Techniques

In this section, we first explain the efficiency bottleneck of current pipelines, and then discuss how to improve it through operation-level optimization.

As outlined above, pipelines can be executed in parallel under four constraints, which enforces the execution order between operations belonging to different pipelines, i.e., between operations on S and P . Given the size of P , if an operation op_P on P must execute before an operation op_S on S or another op_P due to the constraints, op_P will block op_S/op_P for a long time. Specifically, there are two blocking cases. **Case-1:** pop_P blocks $refill_S \circ pop_S$, caused by $\odot_2$ when using the branch pop (a) for pop . **Case-2:** $push_P$ blocks pop_P , caused by $\odot_3$ when a combination of unfinished $push_P$ in last iteration and pop_P in next iteration occurs, i.e., $push$ (b) or $push$ (c) combined with a pop (a) will cause this blocking case.

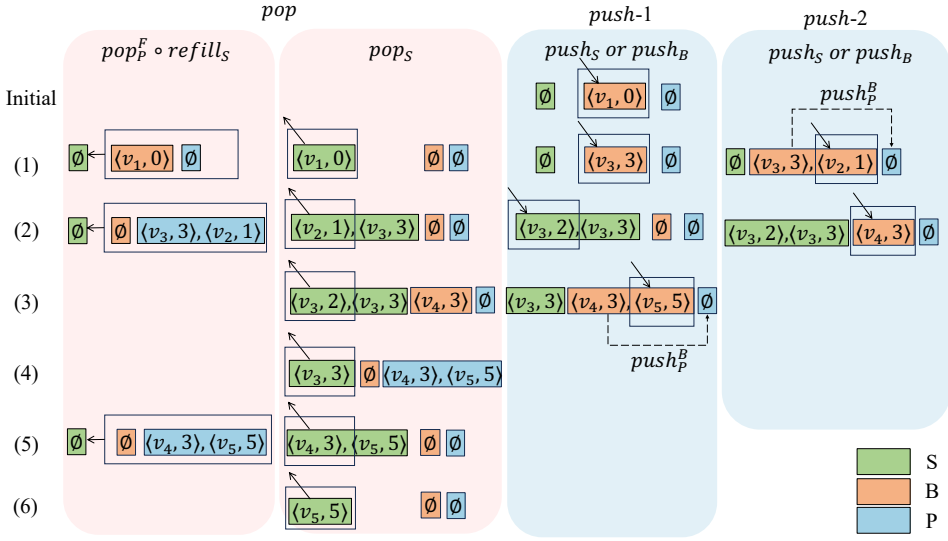
For optimization of **Case-1**, the pop_P can first find and transfer the top- K elements to $refill_S$, postponing any remaining internal adjustments of P to a later point. As a result, $refill_S$ only waits for part of pop_P . As shown by Fig. 2, for pop (a) in iteration $i + 1$, the $refill_S \circ pop_S$ in pipeline 1 can execute once pop_P returns the top- K elements in P . Formally, we decompose pop_P into two operations, i.e., $pop_P() \equiv pop_P^F \circ pop_P^D$, where pop_P^F finds and returns the top- K elements in P , and pop_P^D deletes top- K elements from P and adjusts P for next operation. Thus $refill_S \circ pop_S$ can execute after pop_P^F finishes, no need to wait for pop_P^D .

For optimization of **Case-2**, we reduce its occurrence by reducing $push_P$ operation. Specifically, we delay $push_P$ for single element, accumulate elements in a staging area and insert them into P at once. This lazy-update strategy serves a dual purpose: it reduces the frequency of $push_P$ operations while simultaneously accumulating enough elements to make batch insertion into P efficient. Formally, we decompose $push_P$ into two operations, i.e., $push_P \equiv push_B(e) \circ [push_P^B]$, where B is an unordered staging area, $push_B$ pushes a new element into B , and $push_P^B$ pushes elements in B into P at once. Since B is an unordered set, the $push_B$ operation is fast, taking $O(1)$ time. The relatively slower operation $[push_P^B]$ is optional, invoked only when B is full.

To implement above optimizations, P should support fast top- K elements retrieval and fast batch insertion, combined with a staging area storing elements waiting to be pushed into P . We denote the staging area as B , and consider partitioning P into multiple sets, each denoted by $N[i]$ and contains K elements. The sets satisfy the invariant that $\max_{(v,p(v)) \in N[i]} p(v) < \min_{(v,p(v)) \in N[i+1]} p(v)$. The set $N[1]$ contains the exact top- K elements in P . The staging area B also has capacity K , so that when it becomes full, its elements can be inserted as a new set into P .

In Fig. 3, we give an overview of our priority queue using an example of $K = 5$. For simplicity, we only show the priority of elements in the figure. In each set $N[i]$ in P , 5 elements are kept. The size of staging area B is also set as 5, and each time B is full, $push_P^B$ will push the whole B into P as new set. Both $N[i]$ and B are unordered. S has capacity of K , and will maintain priority order of elements inside it. Each pop_P^F searches for top- K elements in $N[1]$ and B , and the time complexity is related with K . It also supports fast batch insertion of K elements. When the elements in B are inserted into P as a new set, the adjustment of P will start at $N[1]$, and proceeds iteratively through the sequence of sets. At each step, it compares elements in $N[i - 1]$ and $N[i]$, retains the top- K elements in $N[i - 1] \cup N[i]$ in $N[i - 1]$, and passes the remaining elements into $N[i]$ for next comparison. When B is frequently filled, this structure allows multiple batch insertions to be processed on P simultaneously, because each $N[i]$ can be adjusted by next push after the last push finishes adjusting it.

Example 4.2: In Fig. 4, we give an example of execution inside PQ^+ according to Table 1, where K is 2. Initially, the source node with its priority 0 is inserted into B . Each push will execute on either S or B , decided by the branching constraints discussed in Section 4.1. When B is full with 2 elements, a $push_P^B$ will insert the whole B into P as a set. Each pop will first retrieve top-1 element in S . If S becomes empty (including the initial state for iteration (1)), pop_P^F will find current top- K

Fig. 4. Executions in PQ^+ when $K = 2$ according to Table 1

in $B \cup P$. If there are less than K elements inside $B \cup P$, e.g., the iteration (1), pop_P^F will return all elements in $B \cup P$. The execution terminates when S, B, P are all empty.

5 The Implementations

In this section, we first detail the underlying data structure of S and P supporting the operations mentioned in Section 4, especially pop_P and $push_P$. Next, we provide an overview of the workflows for core operations. Finally, we provide a theoretical analysis of the time complexity of PQ^+ , which is no worse than that of a sequential priority queue even under sequential execution.

5.1 Operations in Two Queues

The Queue S : As mentioned in Section 4, S is a queue supporting operations pop_S , $push_S$, $refill_S$, low_S , and it should further support $size_S$ for checking its size, and support $lowr_S$ to read the element $\langle v, p(v) \rangle = \arg \max_{\langle u, p(u) \rangle \in S} p(u)$ for the admission check $\mathcal{A}$, without deleting it from S . To attain these objectives, we utilize a min-max heap [4] for S . A min-max heap is a double-ended priority queue, enabling the retrieval of the minimum and maximum elements in $O(1)$ time, and can be constructed in linear time. Given the bounded size of S , it can support $size_S$ and $lowr_S$ operations in $O(1)$ time, support pop_S , $push_S$, low_S operations in $O(\log K)$ time, and support $refill_S$ operation in $O(K)$ time.

The Queue P : As illustrated in Section 4, P is designed as a queue in which ordering is enforced only among groups of K elements. It has a staging area B , and supports pop_P^F , pop_P^D , $push_B$, $push_P^B$ operations. The time complexity of pop_P^F is $O(K)$, and the time complexity of $push_B$ is $O(1)$, as mentioned in Section 4.3. We then discuss the design of P for pop_P^D and $push_P^B$.

Consider P contains up to $O(m)$ elements, the number of K -size sets inside it is up to $O(m/K)$. We design P as a binary tree, with each node in tree containing K elements. Each node t has a number label, denoted as $\delta(t)$. The root node of tree is labeled 1. For each non-leaf node i , its left

Algorithm 1: $P.\text{push}(u, p(u))$

Input : a vertex u and its priority $p(u)$

/ push_B */* */

1 $P.B \leftarrow P.B \cup \{ \langle u, p(u) \rangle \};$

2 **if** $|P.B| < K$ **then return**;

/ push_P^B */* */

3 $t \leftarrow 1 + \text{the number of nodes in } P; m = \lfloor \log_2(\delta(t) - 1) \rfloor + 1;$

4 $cur \leftarrow 1; C \leftarrow P.B; P.B \leftarrow \emptyset;$

5 **for** $i \leftarrow 1$ **to** m **do**

6 Assign top- K elements in $P.N[cur] \cup C$ to $P.N[cur]$ while others to $C;$

7 $cur \leftarrow 2 \cdot cur + (\lfloor \delta(t)/2^{m-i} \rfloor \bmod 2);$

8 New a node with number cur and $P.N[cur].elements \leftarrow C;$

9 **return**;

child node is labeled $2i$, and its right child node is labeled $(2i + 1)$. Below, we use $P.N[i]$ to denote a node in P labeled i . The tree satisfies the following properties. ❶ Every non-leaf node has two child nodes, a left one and a right one. The only exception allowed is the last non-leaf node, which can have only a left child node. ❷ The lowest priority of elements in a parent node is higher than the highest priority of elements in its child node(s). ❸ Inside each node, K elements are unordered. For the staging area B , since elements inside it are unordered, no specific data structure is designed for it, and it is viewed as a set.

Formally, for a node t in the tree, we denote its parent as $f(t)$. When the context is clear, we also use t to refer to the set of K elements stored inside that node. For any node t , it satisfies $\max_{\langle v, p(v) \rangle \in f(t)} p(v) \leq \min_{\langle v, p(v) \rangle \in t} p(v)$. There are no requirements of priority order inside a node, nor between sibling nodes. Because of the property ❷, it guarantees the root node of the tree contains the exact top- K elements in P , supporting fast pop_P^F . Next, we provide the details of operations push_P^B and pop_P^D in P .

Operation push_P^B : We first use the special case $K = 1$ only as intuition. In this case, P degenerates to a binary-heap-like structure, where a push operation inserts the new element at the end of the tree and then adjusts it upward by repeatedly swapping it with its parent whenever necessary. Our push_P^B operation retains this intuition, but generalizes it to $K > 1$ in two key modifications. The first modification is related to the efficiency issue. Given the adjustment progress of push is from the bottom to the top of a tree, nodes on the path from a newly added node to the root must be blocked since they may not satisfy property ❷, until the adjustment is completed. This adjustment could obstruct other operations that require the root node of P , i.e., pop_P^F and pop_P^D . Alternatively, adjusting from the top to the bottom allows execution of pop_P^F and pop_P^D once the elements in the root node are determined. Therefore, we design a top-to-bottom adjustment. Specifically, for a node t , whose binary representation is $(1t_1t_2 \dots t_b)_2$. We have $b = \lfloor \log_2(\delta(t) - 1) \rfloor + 1$, hence, $t_i = \lfloor \delta(t)/2^{b-i} \rfloor \bmod 2$. Assuming the root node has a tree depth of 1. Based on the following theorem, the process begins at the root with number 1, where, at each step, a node t at layer i conducts a swap operation with its left child node if $t_i = 0$ or with its right child node otherwise.

Theorem 5.1: *Given a node t in tree of P with a label $\delta(t)$, let $b = \lfloor \log_2(\delta(t) - 1) \rfloor + 1$ and $t_i = \lfloor \delta(t)/2^{b-i} \rfloor \bmod 2$. Let $[x_0, \dots, x_b]$ be a path satisfy x_0 is the root, $x_b = t$, and x_{i-1} is the parent node of x_i for $i \in \{1, \dots, b\}$. Let $[y_0, \dots, y_b]$ be a path satisfy y_0 is the root, $y_b = t$, and y_{i+1} is the*

Algorithm 2: $P.\text{pop}()$

```

Output : a set of top- $K$  elements in  $P$ 
/*  $\text{pop}_P^F$  */
1  $C \leftarrow P.B \cup P.N[1].\text{elements};$ 
2  $R \leftarrow$  top- $K$  elements in  $C$ ;  $C \leftarrow C \setminus R$ ;
3 return  $R$  while not return this function;
/*  $\text{pop}_P^D$  */
4  $t \leftarrow$  the number of nodes in  $P$ ;
5  $P.N[1] \leftarrow P.N[t]$ ; remove node  $P.N[t]$  from  $P$ ;  $\text{cur} \leftarrow P.N[1]$ ;
6 while  $\text{cur}$  has child nodes in  $P$  do
7   if  $\text{cur}$  has one child node  $c$  then
8     Assign top- $K$  elements in  $\text{cur} \cup c$  to  $\text{cur}$  while others to  $c$ ;  $\text{cur} \leftarrow c$ ;
9   else
10     $c_1, c_2 \leftarrow$  child nodes of  $\text{cur}$ ;
11    if  $\max_{\langle v, p(v) \rangle \in c_1} p(v) > \max_{\langle v, p(v) \rangle \in c_2} p(v)$  then
12      swap  $c_1$  and  $c_2$ ;
13    Assign top- $K$  elements in  $c_1 \cup c_2$  to  $c_1$  while others to  $c_2$ ;
14    Assign top- $K$  elements in  $\text{cur} \cup c_1$  to  $\text{cur}$  while others to  $c_1$ ;  $\text{cur} \leftarrow c_1$ ;

```

left child node of y_i if $t_i = 0$, while y_i is the right child node of y_{i-1} otherwise, for $i \in \{1, \dots, b\}$. We have the path $[x_1, x_2, \dots, x_b]$ is identical to the path $[y_1, y_2, \dots, y_b]$.

Proof Sketch: We prove this by induction. When $i = 1$, we have $x_0 = y_0$ is root. Assume for $i - 1$ that y_{i-1} satisfy $(\delta(y_{i-1}))_{10} = (1t_1 \dots t_{i-1})_2$ and $x_{i-1} = y_{i-1}$. Hence, for i , if $t_i = 0$, we have y_i is the left child node of y_{i-1} , hence, $\delta(y_i) = 2\delta(y_{i-1})$, hence $(\delta(y_i))_{10} = (1t_1 \dots t_{i-1}0)_2$; if $t_i = 1$, we have y_i is the right child node of y_{i-1} , hence, $\delta(y_i) = 2\delta(y_{i-1}) + 1$, hence $(\delta(y_i))_{10} = (1t_1 \dots t_{i-1}1)_2$. Therefore, we have $(\delta(y_i))_{10} = (1t_1 \dots t_{i-1}t_i)_2$. Since $\delta(x_i) = 2\delta(x_{i-1}) + t_i = \delta(y_i)$, $x_i = y_i$ holds. $\square$

The second modification required is in swap operation: given a node t and its parent $f(t)$, there are 2 unordered sets of K elements, and we must rearrange the elements in both nodes such that property ② holds. For the top-to-bottom adjustment of push_P^B , Theorem 5.1 establishes a path from the root to the newly added nodes in tree. With a set C containing K elements to be pushed, at each node t along the determined path, we execute a swap operation between C and t , such that t encompasses the top- k elements from $t \cup C$. Next, the set C containing new elements engages in a swap operation with the child node of t as determined by Theorem 5.1. We can find that at each node t , t will potentially have elements of higher priority, as it acquires the top- K elements from $t \cup C$. This maintains the property ② between t and its child node(s).

Specifically, for a swap operation between two sets t_1 and t_2 in P , we assign k -highest priority elements to t_1 and remains to t_2 by first compute the the K -highest priority element, median μ , in $t_1 \cup t_2$, i.e., elements e satisfying $p(e) \leq \mu$ are assigned to t_1 , while the rest are assigned to t_2 . The median in an unordered list $t \cup f(t)$ can be located in $O(|t \cup f(t)|) = O(K)$ time [7].

Algorithm 1 presents the details of the push operation in P . We first add a new element to the staging area B of P (line 1), and the push operation terminates when the size of B is less than K . Upon reaching this threshold, all elements in B are pushed to P . The algorithm computes the label for new node t (line 3) and m in Theorem 5.1 for subsequent swap operations. The process starts at the root node labeled 1, utilizing C to store the current element awaiting insertion into P (line 4). For each node, the top- K elements are retained, with the remaining K elements intended for

insertion into its child node (line 6). The child nodes' labels are determined according to Theorem 5.1 (line 7). Finally, a new node labeled t is created, and K elements are added into it (line 8). Note that each push_P^B operation processes nodes sequentially, i.e., it begins at the currently adjusted node $P.N[i]$, and once the adjustment of $P.N[i]$ is complete and its child node becomes the new target, a subsequent push_P^B operation can start working on $P.N[i]$. Although each individual push_P^B operation is executed sequentially, multiple push_P^B operations can still proceed concurrently on different nodes within P , following the order from $P.N[i]$ to $P.N[i + 1]$. This enables P to exploit inter-operation parallelism through overlapping executions of distinct push_P^B operations. Though push_P^B is triggered only under the boundary constraint $|B| = K$, this constraint can occur frequently in practice when branch push (b) and push (c) are always invoked. The main purpose of this parallelism mechanism is to shorten the response time of future operations on P , especially branch pop (a), which involves pop_P , an operation that can block the main pipeline and further degrade the overall performance of the algorithm.

Operation pop_P^D : We again use the special case $K = 1$ only to illustrate the intuition. In this case, a pop operation removes the root element, refills the root with the element from the last node, deletes the last node, and then performs a downward adjustment. Starting from the root, for each node t along the path toward a leaf, the element in t is compared with those in its children. If one child has a higher-priority element, a rotate operation is performed to move the highest-priority element to t , and the process continues at that child; otherwise, the adjustment terminates.

Our pop_P^D operation follows the same high-level intuition of downward adjustment, but generalizes it to the case $K > 1$. The sole modification required is in the rotate operation: given a node t with at most 2 child nodes c_1 and c_2 , we must rearrange the elements in the three nodes such that property ② holds. We utilize swap operation proposed for push_P^B to achieve this. Specifically, as shown in Algorithm 2, if t contains only one child c_1 (line 7), we conduct swap between c_1 and t and continues process c_1 (line 8). When t contains two child nodes c_1 and c_2 (line 10), assuming $\max_{\langle v, p(v) \rangle \in c_1} p(v) \leq \max_{\langle v, p(v) \rangle \in c_2} p(v)$ (lines 11-12), we first conduct a swap between c_1 and c_2 (line 13), assigning the top- K elements to c_1 . Since c_2 has a larger maximum element, property ② is maintained for c_2 and its child nodes. Next, we focus on c_1 and t , and conduct a swap between t and c_1 , continuing the process on c_1 (line 14). The other process for $K > 1$ is similar to that of the heap. The top- K elements are extracted from P by combining the elements in B and the root node $P.N[1]$ (lines 1-2), and the remaining elements are then pushed back into B (line 2). The results can be promptly returned (line 3), while the adjustment remains necessary in another phase (lines 4-14). The process first utilizes the last node in P to populate the root node, and then removes the last node (lines 4-5). Next, an adjustment is implemented from the top to the bottom of P , where each node is compared to its child node through a swap operation (lines 8-14).

5.2 Operations in PQ^+

In this section, as a summary, we present the details of operations in our priority queue PQ^+ . Next, we analyze the theoretical time of PQ^+ and discuss the parameter setting for K .

Algorithm 3 details the push operation in our PQ^+ . The priority of the lowest priority element in S is acquired for the check $\mathcal{A}$ (line 1), which needs $O(1)$ time. If the newly added element $\langle v, p(v) \rangle$ satisfies $\mathcal{A}$ (line 2), it is inserted into S (line 3). The insertion to S needs $O(\log |S|)$ time. When the size of S exceeds K , the lowest priority element is removed from S (lines 5-6) and inserted into P (line 7). When B is not at full capacity ($|B| < K$), this operation requires $O(1)$ time. Otherwise, when B is full and needs to be inserted into P , the insertion process into P , comprising $|P|/K$ nodes, demands $\log(|P|/K)$ adjustments, where $|P|$ denotes the number of elements in P . Each adjustment, involving swap, consumes $O(K)$ time. Thus, in this case, the total time complexity

Algorithm 3: $PQ^+.push(v, p(v))$

Input : a vertex v and its priority $p(v)$

- 1 $p_l \leftarrow$ the priority of the lowest-priority element in $Q.S$;
- 2 **if** $p(v) \leq p_l$ **then**
- 3 Insert $\langle v, p(v) \rangle$ into $Q.S$;
- 4 **if** the size of $Q.S > K$ **then**
- 5 $\langle u, p(u) \rangle \leftarrow$ the lowest-priority element in $Q.S$;
- 6 Delete $\langle u, p(u) \rangle$ from $Q.S$;
- 7 $Q.P.Push(\langle u, p(u) \rangle)$;
- 8 **else**
- 9 $Q.P.Push(\langle v, p(v) \rangle)$;

Algorithm 4: $PQ^+.pop()$

Output : The highest priority element in Q

- 1 **if** $Q.S$ is empty **then**
- 2 $C \leftarrow Q.P.pop()$;
- 3 **if** $C = \emptyset$ **then return** $\emptyset$;
- 4 Build a min-max heap of C for $Q.S$;
- 5 **return** The highest-priority element in $Q.S$;

is $O(K \cdot \log(|P|/K))$. Otherwise, when the newly added element $\langle v, p(v) \rangle$ not satisfies $\mathcal{A}$ (line 8), $\langle v, p(v) \rangle$ is directly inserted into P (line 9).

Algorithm 4 details the pop operation in our PQ^+ . If S is empty (line 1), it needs to be refilled with the top- K elements in P (line 2). Given that P comprises at most $|P|/K$ nodes, the operation $Q.P.pop$ necessitates at most $|P|/K$ adjustments. Each adjustment requires $O(K)$ time, consequently, this operation needs at most $O(K \cdot \log(|P|/K))$ time. In case C remains empty, the function returns $\emptyset$ (line 3), indicating that the current priority queue Q is empty. Otherwise, a min-max heap based on C is created for S (line 4), which needs $O(K)$ time. Finally, the highest priority element can be retrieved from S in $O(1)$ (line 5).

Next, we analyze the time complexity of the operations in our PQ^+ without parallelism for priority-based graph algorithms. Following the priority-based graph algorithms, e.g., Dijkstra [8] and Prim [35], assuming the priority-based graph algorithms require $O(n)$ iterations and each push operation in an iteration is related to the degrees of popped nodes in the pop operation, implying a total of $O(m)$ push operations throughout the algorithm.

As discussed, the best-case scenario for a push operation is insertion into S , taking $O(\log K)$ time. When inserted into P , it takes $O(1)$ time if B is not full and $O(K \cdot \log \frac{m}{K})$ time when B is full. In the worst-case scenario, every K push operation fills B , necessitating push K elements into P . Thus, the total time for the push operation in PQ^+ is $O(m \log K + \frac{m}{K} \cdot K \cdot \log \frac{m}{K}) = O(m \log m)$, which does not depend on the chosen value of K . In practice, we chose K to balance the two time costs of S -side and P -side, i.e., $m \log K = \frac{m}{K} \cdot K \cdot \log \frac{m}{K}$. Thus, we set $K = \sqrt{m}$.

For pop operation, the best case involves obtaining an element from S , requiring $O(\log K)$ time due to the necessity of adjusting S for the subsequent operation. When S is empty, push operation first involves a pop operation in P to obtain K elements. As analyzed, the pop operation in P demands $O(K \cdot \log \frac{m}{K})$ time, given that P contains at most $O(m)$ elements. Then, $O(K)$ time is needed to construct a min-max heap for S . Thus, when S is empty, the total time required is

Table 3. Statistics of datasets

Dataset	#Vertices	#Edges	Type	Directed	Weighted	d_{avg}
Wiki-talk	2,394,385	5,021,410	S	Y	N	2.1
Google	875,713	5,105,039	W	Y	N	5.8
Amazon	3,683,637	28,202,600	Ra	N	Y	7.7
Topcats	1,791,489	28,511,807	W	Y	N	15.9
USA	23,947,347	58,333,344	R	N	Y	2.4
Livejournal	4,847,571	68,993,773	S	Y	N	14.2
Dewiki	2,166,669	86,337,879	W	N	Y	39.8
Orkut	3,072,441	117,185,083	S	N	N	38.1
Yahoo	1,625,951	256,804,235	Ra	N	Y	157.9
Friendster	65,608,366	1,806,067,135	S	N	N	27.5

$O(K \cdot \log \frac{m}{K})$. However, this scenario is less common, as at least K pop operations trigger a pop in P . Thus, the overall time needed for pop in PQ^+ is $O(n \log K + \frac{n}{K} \cdot K \cdot \log \frac{m}{K}) = O(n \log m)$, which does not depend on the chosen value of K . Similarly, we set $K = \sqrt{m}$ to balance the S -side cost $n \log K$ and P -side cost $\frac{n}{K} \cdot K \cdot \log \frac{m}{K}$.

Hence, our PQ^+ ensures that the theoretical time complexity remains comparable to sequential algorithms when operating in sequential mode. The overall time cost does not depend on the choice of K due to the coordination mechanism between S and P . Furthermore, the fine-grained cost of each branch is analyzed in Exp. 4 and Exp. 5 in Section 6, and the experimental results show that the probability of triggering push and pop operations in P is significantly lower than m/K and n/K , respectively, enhancing the practical performance of our PQ^+ .

6 Experiments

Datasets: The experiments are conducted using 10 real-world datasets, where Amazon, Dewiki, and Yahoo are from KONECT [25], USA is from DIMACS Implementation Challenge [1], and others are from SNAP [26]. Table 3 shows the statistics of the datasets, ordered by the number of edges, where d_{avg} denotes the average vertex degree in the graph. Datasets cover social graphs (S), web graphs (W), road networks (R), and user-item rating networks (Ra). These datasets include both weighted and unweighted graphs, as well as directed and undirected graphs. For weighted graphs, the weight of edges in Amazon and Yahoo represents the user's ratings for items, ranging from 1.0 to 5.0 and 0 to 100, respectively. The weight of edges in USA represents the real road distance, and the weight of Dewiki represents the addition (+1) and removal (-1) of hyperlinks between web pages. For unweighted graphs, we randomly assign an integer weight between 0 and 100 to each edge.

Graph Algorithms: We evaluate different approaches on 4 fundamental priority-based graph algorithms. (1) Dijkstra [8]: an algorithm for the single-source shortest path problem, the details of Dijkstra have been introduced in Section 2. We use the graphs with weights to evaluate Dijkstra. (2) IMM [38]: a greedy algorithm for the Influence Maximization (IM) problem with $(1 - 1/e - \epsilon)$ -approximation. Since other algorithms for IM problem, e.g., [14, 39], merely differ in their sampling techniques while maintaining similar vertex selection processes by RR set. We exemplify by evaluating IMM for IM problem. We set the seed set size as $k = 10$ for evaluation. Following [14, 38, 39], we assign each edge e a weight of $1/i$, where i is the number of edges sharing the same endpoint with e . In IMM algorithm, it initially generates vertex sets $N(v)$ for each vertex v in the graph, i.e., RR sets. Next, it chooses k sets for maximum coverage. The priority $p(v)$ for each vertex v equates to the marginal gain of including v in the seed set S , defined as $p(v) = \sigma(S \cup \{v\}) - \sigma(S)$, where

$\sigma(S) = \bigcup_{v \in S} N(v)$. Initially, for each vertex v , $\langle v, p(v) \rangle$ is inserted into Q , where $p(v)$ is computed assuming $S = \emptyset$. Next, it conducts k iterations, where the vertex with the highest priority is popped to update the priorities of other vertices before pushing them back into the queue. (3) Prim [35]: an algorithm for the minimum spanning tree (MST) problem. In MST problem evaluation, we treat directed graphs as undirected. In Prim algorithm, the priority $p(v)$ of each vertex v is the minimum weight of an edge connecting it to one of the expanded vertices. The Prim algorithm randomly selects a vertex $s \in V$ for initial expansion, setting $p(s) = 0$ and $p(u) = +\infty$ for $u \in V \setminus s$, then $\langle s, 0 \rangle$ is inserted into the priority queue Q . In each iteration, a vertex v with the smallest $p(v)$ is popped from Q and utilized to update the priority for its neighbor u using $p'(u) = \min(p(u), w(v, u))$. If $p'(u) < p(u)$, $\langle u, p'(u) \rangle$ is added to Q . Prim terminates when Q is empty. (4) ST [2]: an algorithm for Minimum Steiner Tree problem with a 2-approximation. Given a graph G and a vertex set T , ST first builds a complete graph G' containing only the vertices in T . The edge between each pair of vertices (u, v) in G' is assigned the shortest path distance between u and v in G , computed using Dijkstra. Next, ST computes the minimum spanning tree of G' using Prim. Replace each edge in the minimum spanning tree of G' with its corresponding shortest path in G , and remove cycles among these edges by eliminating the edge with the maximum weight in each cycle to obtain the final result.

We include the above greedy algorithms in our evaluation for reasons as follows. First, each of them solves a fundamental problem on graphs, serving as a canonical example of a priority-based graph algorithm. Second, Dijkstra and Prim are exact algorithms, and IMM and ST are approximation algorithms with a theoretical guarantee. Many other greedy algorithms, e.g. [14, 28, 39, 43, 46], can be implemented by replacing the objective function $\sigma(\cdot)$ in IMM. We believe testing on these graph algorithms can provide a comprehensive evaluation of the proposed techniques.

Compared Algorithms: We conduct comparisons between our approach, PQ^+ , with three state-of-the-art implementations of priority queues. ❶ Seq [37]: The sequential priority queue provided by STL. For reproducibility, we use Boost library to conduct all the tested algorithms using a sequential priority queue, which uses STL containers as the underlying data structure and provides interfaces supporting Dijkstra and Prim algorithms. Specifically, we use `boost::dijkstra_shortest_paths` for Dijkstra algorithm, `boost::prim_minimum_spanning_tree` for Prim algorithm, and implement the algorithms IMM and ST using the Boost library. ❷ TBB [19]: The state-of-the-art concurrency priority queue based on a heap, provided by the open-source library Intel TBB. For TBB, it uses a thread pool, and we allow configuration of the thread number by its default settings. ❸ SprayList [3]: The state-of-the-art concurrent priority queue based on skiplist. For SprayList, the number of threads is tunable. Since the original publication of SprayList does not specify an optimal thread number, we ran the algorithm with different thread number settings among $\{4, 8, 16, 32, 48, 64\}$ and report the average result obtained. ❹ PQ^+ : Our priority queue approach proposed in this paper. For evaluation, PQ^+ employs a fixed thread pool of size 48 and set $K = \sqrt{m}$ by default.

For fair comparisons, we access all algorithms on the same single machine, employing multiple threads for TBB, SprayList, and PQ^+ , and a single thread for Seq. The implementation of all graph algorithms remains consistent across all methods, with the only variation being the use of priority queues in the algorithms, except for Dijkstra and Prim in Seq, we directly utilize the provided interface for evaluation, ensuring optimal performance. The input graphs are stored in CSR format for all evaluation. For all evaluated algorithms, we use randomly selected source vertices and report the average performance across these runs.

Experimental Environments: All experiments are conducted on a Linux server with an AMD EPYC 7443 24-Core Processor and 1024G memory. All algorithms are implemented in C++. The code is compiled with g++ 8.5 under O3 optimization.

Table 4. Performance comparisons of four graph algorithms when using 4 priority queues (ms) (Exp. 1)

Dataset	Dijkstra				IMM			
	Seq	SprayList	TBB	PQ ⁺	Seq	SprayList	TBB	PQ ⁺
W	826	939	933	613	4,640	5,926	1,940	1,488
G	340	404	359	313	674	1,077	683	394
A	2,302	9,543	2,008	1,525	12,965	38,285	9,393	7,879
T	1,468	6,065	1,544	1,315	9,005	45,639	6,723	5,182
U	3,638	4,768	3,418	3,402	1,287	9,236	648	613
L	3,380	10,738	3,899	2,570	12,011	47,145	9,436	7,415
D	4,157	21,836	4,844	3,413	11,151	56,517	8,736	7,275
O	4,479	N	3,936	3,359	31,890	N	22,980	19,927
Y	3,180	9,052	2,398	1,201	30,788	46,619	22,897	23,514
F	125,535	N	107,253	74,762	595,709	N	570,630	414,985

Dataset	Prim				ST			
	Seq	SprayList	TBB	PQ ⁺	Seq	SprayList	TBB	PQ ⁺
W	890	998	899	613	17,303	18,062	17,755	12,072
G	422	618	380	351	7,006	8,170	7,175	6,358
A	2,635	15,350	2,120	2,009	51,278	257,152	43,025	31,093
T	1,691	9,729	1,509	1,468	26,954	205,955	31,723	25,801
U	3,041	4,152	3,255	2,835	77,490	95,999	79,011	69,648
L	3,563	11,746	3,610	3,054	75,599	227,893	81,601	57,337
D	4,129	28,266	3,895	3,400	84,659	519,912	100,630	65,325
O	4,607	N	5,082	3,839	98,030	N	83,006	62,424
Y	3,098	11,028	3,100	1,930	67,152	275,883	46,363	25,362
F	130,359	N	108,689	90,315	2,533,740	N	2,125,077	1,432,463

Exp. 1: Overall performance of different approaches on four graph algorithms. As shown in Table 4, our PQ⁺ consistently achieves the best performance across all datasets and algorithms. It is worth noting that, due to the fundamental nature of the problem and its priority-based nature, Seq is fast enough that the parallel algorithm baselines cannot outperform it in most cases. For Dijkstra, Prim and ST, the suboptimal performance occurs in Seq on half or more than half of the datasets. However, PQ⁺ exhibits stable acceleration performance compared to Seq, which we attribute to its theoretical complexity being comparable to Seq when executed sequentially. PQ⁺ achieves average speedups of 1.43x, 1.78x, 1.28x and 1.49x, and maximum speedups of 2.64x, 3.11x, 1.61x and 2.65x compared to Seq, on the four algorithms respectively. Compared to TBB, PQ⁺ achieves average speedups of 1.37x, 1.26x, 1.22x and 1.40x, and maximum speedups of 1.99x, 1.73x, 1.61x and 1.83x, on the four algorithms respectively. As we can find, the sequential method, Seq, serves as a strong baseline, achieving competitive runtimes in more than half of the datasets for Dijkstra, Prim, and ST algorithms, i.e., outperforming SprayList and TBB, while our method PQ⁺ always surpasses Seq in all scenarios. SprayList performs worst among all priority queues, mainly due to its over-relaxation of top-1 retrieval, causing increased iterations which cannot be offset by the increased concurrency, as claimed in [3]. Among all results, we can find one exception: for IMM algorithm on Yahoo dataset, TBB is slightly faster than PQ⁺. This outlier can be attributed to the specific characteristics of both the dataset and the algorithm. Yahoo is a bipartite graph, which is significantly dense with an average degree of 157.9 (as shown in Table 3). The dense structure fully leverages the concurrent push performance of TBB. Moreover, since IMM itself is an approximation algorithm, in implementation for TBB, we do not increase the number of iterations to compensate for its relaxed top-1 semantics but instead fix the iteration count to a constant n . Under these specific conditions, the superior performance of TBB is expected, but only a slight advantage.

Table 5. Performance comparisons of four graph algorithms using PQ^+ when varying K (ms) (Exp. 2)

Dataset	Dijkstra					IMM				
	$\sqrt{m}/4$	$\sqrt{m}/2$	$\sqrt{m}$	$2\sqrt{m}$	$4\sqrt{m}$	$\sqrt{m}/4$	$\sqrt{m}/2$	$\sqrt{m}$	$2\sqrt{m}$	$4\sqrt{m}$
W	643	615	613	619	672	1,655	1,523	1,488	1,611	1,690
G	355	350	313	356	373	534	394	394	420	490
A	1,665	1,791	1,525	1,714	1,604	7,732	7,681	7,879	8,205	8,161
T	1,289	1,434	1,315	1,304	1,353	5,280	5,448	5,182	5,525	5,639
U	4,344	3,642	3,402	3,547	3,546	648	630	613	657	692
L	2,919	2,876	2,570	2,804	2,835	7,652	7,827	7,415	7,940	8,038
D	3,472	3,347	3,413	3,616	3,366	7,311	7,591	7,275	7,364	7,285
O	3,750	3,590	3,359	3,069	3,096	19,446	20,526	19,927	19,717	19,848
Y	1,452	1,405	1,201	1,457	1,445	24,705	25,389	23,514	23,132	25,019
F	76,343	75,918	74,762	85,833	82,554	438,235	434,602	414,985	419,760	419,252

Dataset	Prim					ST				
	$\sqrt{m}/4$	$\sqrt{m}/2$	$\sqrt{m}$	$2\sqrt{m}$	$4\sqrt{m}$	$\sqrt{m}/4$	$\sqrt{m}/2$	$\sqrt{m}$	$2\sqrt{m}$	$4\sqrt{m}$
W	636	595	613	680	692	12,219	12,525	12,072	12,823	13,103
G	376	367	351	356	387	7,555	7,069	6,358	7,012	9,202
A	2,137	2,204	2,009	2,060	2,061	31,954	32,290	31,093	33,203	31,160
T	1,404	1,542	1,468	1,389	1,403	26,479	28,364	25,801	23,979	25,228
U	2,902	3,068	2,835	3,116	3,191	80,345	71,221	69,648	69,305	70,475
L	3,164	2,958	3,054	3,061	3,958	57,940	55,282	57,337	55,372	53,692
D	3,256	3,295	3,400	3,505	3,482	70,158	67,960	65,325	68,398	68,948
O	4,530	3,956	3,839	3,726	3,765	72,683	69,526	62,424	60,947	59,079
Y	2,211	2,102	1,930	1,959	2,009	26,427	25,322	25,362	24,444	23,776
F	96,935	92,218	90,315	101,110	88,232	1,535,876	1,518,220	1,432,463	1,736,563	1,622,093

Exp. 2: Effects of varying the parameter K on PQ^+ . In Table 5, we vary the parameter K among $\{\sqrt{m}/4, \sqrt{m}/2, \sqrt{m}, 2\sqrt{m}, 4\sqrt{m}\}$ of PQ^+ to show its effect on all four graph algorithms. As we can find, setting K as $\sqrt{m}$ achieves the best performance across all algorithms on almost all datasets. Specifically, algorithms Dijkstra, IMM, Prim and ST achieve the best running time on 7, 7, 4, and 5 datasets, respectively; they achieve optimal or second-optimal performance on 7, 7, 8, and 7 datasets, respectively, validating the consistent competitiveness of this parameter choice, which aligns with our theoretical analysis for balancing the time complexity of push and pop in Section 5.2.

Exp. 3: Effects of varying thread number. Fig. 5 reports the performance of three concurrent priority queues, SprayList, TBB, and PQ^+ , under varying thread numbers on 8 datasets for Dijkstra's algorithm. Orkut and Friendster are excluded because SprayList fails to terminate correctly on both datasets. Overall, PQ^+ consistently achieves the best performance and shows stable behavior across different thread numbers. This stability mainly reflects the robustness of PQ^+ . By comparison, SprayList exhibits noticeable fluctuations across different thread counts and is overall less effective, while TBB is also relatively stable but remains generally inferior to PQ^+ . The results also suggest that the thread scalability of PQ^+ is limited. This is not unexpected for priority-based graph algorithms under strict top-1 semantics, where the available parallel slack is constrained by the order-sensitive execution. As a result, a relatively small number of threads may already exploit most of the available parallelism, and further increasing the thread number brings only marginal or no gains. Therefore, PQ^+ should be viewed as an exactness-preserving acceleration mechanism rather than a design targeting unbounded scalability with thread number.

Exp. 4: Statistics of push operations in PQ^+ . As mentioned in Section 4.1, push operation of PQ^+ has three branches, namely push_S , $\text{push}_S \circ \text{low}_S \circ \text{push}_P$ and push_P . We further decompose $\text{push}_P \equiv \text{push}_B \circ [\text{push}_P^B]$. Therefore, without considering operations on P , we have three branches for push: (i) push_S ($O(\log K)$), for successfully add a new element passed $\mathcal{A}$ into S ; (ii) $\text{push}_S \circ \text{low}_S \circ \text{push}_B$ ($O(\log K)$), for successfully add a new element passed $\mathcal{A}$ into S when $|S| = K$, and

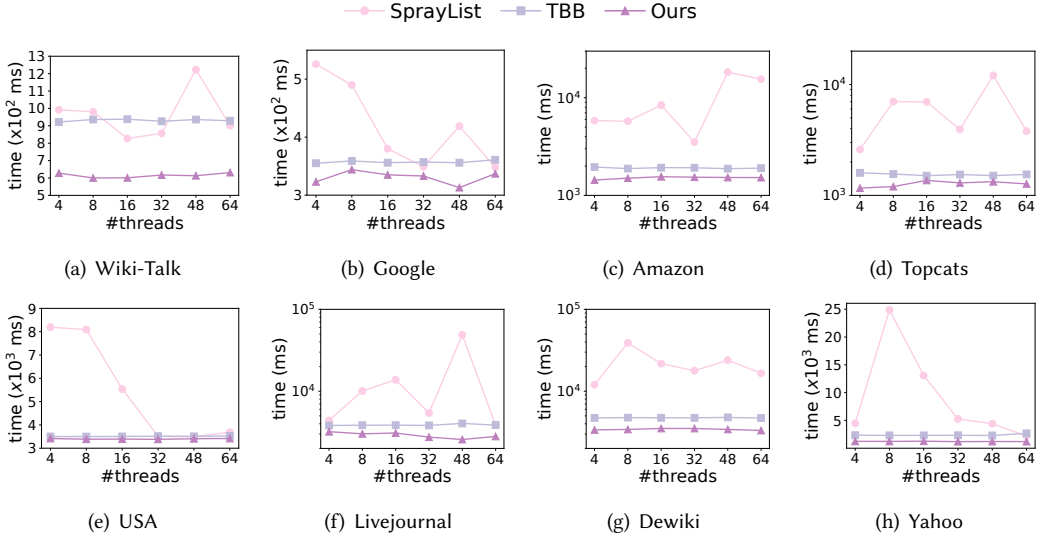


Fig. 5. Performance of three priority queues in Dijkstra's algorithm when varying the thread number (Exp. 3)

Table 6. Statistics of push/pop operations in PQ^+ of Dijkstra's algorithm (Exp. 4, Exp. 5)

Dataset	#op	push					pop		
		#(i)	#(ii)	#(iii)	#push _P ^B	#push _P ^B /#op	#(i)	#(ii)	#(ii)/#op
W	2,904,313	708	6,571	2,897,034	1,295	0.0446%	2,903,014	1,299	0.0447%
G	1,610,607	11,098	5,148	1,594,361	707	0.0439%	1,609,893	714	0.0443%
A	5,491,519	6,044	6	5,485,469	1,032	0.0188%	5,490,484	1,035	0.0188%
T	5,136,476	9,297	21,799	5,105,380	960	0.0187%	5,135,511	965	0.0188%
U	25,660,225	24,344,026	0	1,316,199	0	0.0000%	25,659,466	759	0.0030%
L	10,235,637	47,634	37,355	10,150,648	1,226	0.0120%	10,234,404	1,233	0.0120%
D	16,531,048	0	0	16,531,048	1,779	0.0108%	16,529,265	1,783	0.0108%
O	11,357,776	30,961	43,676	11,283,139	1,046	0.0092%	11,356,727	1,049	0.0092%
Y	1,961,130	77	5,535	1,955,518	122	0.0062%	1,961,006	124	0.0063%
F	179,797,478	83,605	175,423	179,538,450	4,228	0.0024%	179,793,242	4,236	0.0024%

then add an element from S into B ; (iii) push_B ($O(1)$), for successfully add a new element fails $\mathcal{A}$ into B . Among all push operations, branch (ii) and (iii) may further trigger operations on P , i.e., push_P^B ($O(K \log \frac{m}{K})$). Table 6 shows the total triggered number of each branch for push in Dijkstra algorithm, together with the number of triggered push_P^B . The operation push_P^B , with the highest time complexity related to m , is triggered at most 0.0446% of all push operations (shown in Wiki-Talk dataset), and on average 0.0166% of all push operations. It is a low probability compared to the theoretical maximum triggered number $\frac{m}{K}$, benefits from the optimizations in Section 4.3. Among all the three branches, branch (iii), with a time complexity of $O(1)$, is triggered most frequently in the vast majority of datasets, accounting for more than 90% of the total pushes on average. We notice that there exist two datasets with severe bias on branch choices, USA dataset and Dewiki dataset. USA dataset has edge weights from the real road distances, which, compared to the weighted web network and rating network, exhibits a more uneven distribution, thus most of the push operations can be completed only on S by push_S . Dewiki dataset, with uniform edge weights, fails to leverage the small-sized queue S , but the efficiency also benefits from the frequent trigger of branch (iii).

Table 7. Performance of PQ^+ in parallel and sequential modes in Dijkstra's algorithm (ms) (Exp. 6).

Dataset	W	G	A	T	U	L	D	O	Y	F
Seq	826	340	2,302	1,468	3,638	3,380	4,157	4,479	3,180	125,535
PQ_p^+	615	328	1,644	1,315	3,402	2,570	3,478	3,359	1,385	74,462
PQ_s^+	805	431	2,135	1,603	3,464	4,055	3,760	4,662	1,818	103,557

Exp. 5: Statistics of pop operations in PQ^+ . As mentioned in Section 4.1, pop operation of PQ^+ has two branches, namely (i) pop_S ($O(\log K)$), and (ii) $\text{pop}_P \circ \text{refill}_S \circ \text{pop}_S$ ($O(K \log \frac{m}{K})$). Note that we further decompose $\text{pop}_P() \equiv \text{pop}_P^F \circ \text{pop}_P^D$ in Section 4.3 for optimization, thus time complexity of branch (ii) can be considered as the time complexity of $\text{pop}_P^F \circ \text{refill}_S \circ \text{pop}_S$ for parallel version of PQ^+ , which is reduced, but still considerable compared to branch (i). Table 6 shows the total triggered number of each branch for pop operation in Dijkstra. The branch (ii), which has the higher time complexity related to m , is triggered at most 0.0447% of all pop operations (shown in Dewiki dataset), and on average 0.0170% of all pop operations, which is a low probability benefits from optimizations in Section 4.3.

Exp. 6: Comparison of performance of PQ^+ in parallel and sequential modes. To better understand the contribution of parallelism, we evaluate PQ^+ in both parallel and sequential modes in Dijkstra's algorithm, denoted by PQ_p^+ and PQ_s^+ respectively. The goal is to disentangle the gain brought by parallel execution from that brought by the separated design inside PQ^+ . The sequential variant of PQ^+ differs from the parallel one in two aspects. First, the operations on P cannot be separated from the main pipeline and then conducted parallelly to the operations on S . For example, in branch pop (a), $\text{refill}_S \cdot \text{pop}_S$ must wait until pop_P^D finishes. Second, the operations on P , i.e., push_P^B , cannot proceed parallelly. Nevertheless, the separated structure of S and P is preserved in PQ_s^+ . As shown in Table 7, we observe that, on several datasets, PQ_s^+ already outperforms the sequential baseline. However, it cannot achieve significant and consistent speedup over all datasets. This is consistent with our theoretical analysis in Section 5.2, i.e., the theoretical time complexity remains comparable to sequential algorithms when operating sequentially. PQ_p^+ further builds on this structural advantage and achieves additional as well as consistent speedup across all datasets.

7 Related Work

Existing approaches for parallelizing priority-based graph algorithms fall into two categories: specifically designed algorithms and general parallel approaches. However, both of them face limitations.

Specific Parallel Algorithms: Parallel algorithms designed for specific priority-based graph problems, e.g., [5, 6, 21, 32, 34, 41], fundamentally alter the original algorithmic semantics to achieve parallelism. We use Δ -stepping algorithm for SSSP problem as an example. In Δ -stepping algorithm, elements are grouped into buckets of width Δ according to their priority, and within the same bucket, the priority of elements is relaxed. Such specific designs sacrifice generality, since they cannot be trivially extended to other priority-based algorithms. The relaxed priority order may require vertices to be traversed multiple times, resulting in a worse theoretical time complexity. Although their performance may outperform sequential algorithms with a higher thread count, their inferior time complexity can make them less efficient than sequential algorithms when fewer threads are available in the majority of cases.

General-Purpose Parallel Approaches: Most existing parallel systems, such as [30, 31, 40, 44, 48], have been designed for graph algorithms. However, they necessitate users to reframe graph algorithms according to their specific models. For instance, when utilizing Pregel [31], graph

algorithms need to be recoded to fit the vertex-centric model. Recent systems, e.g., GraphBLAS [20], GRAPE [11, 12], enable parallel execution of existing sequential graph code, but they do so by restructuring the computation, e.g., typically through graph partitioning. Such transformation also modifies the underlying algorithmic semantics of priority-based; the global priority order is replaced by local orders. As a result, even when the output remains correct after convergence, the theoretical complexity and solution guarantees of the sequential algorithm no longer hold, as we claimed above in the exactness issue in Section 3. Moreover, graph partitioning and other mechanisms need extra workloads to preprocess the graph, e.g., GRAPE takes 20 minutes to import Friendster dataset for the first time processing.

8 Conclusion

This paper introduces PQ^+ , the first priority queue that can guarantee exactness, enable parallel acceleration of algorithms, while maintaining theoretical time complexity comparable to sequential version. All graph algorithms utilizing a priority queue can leverage PQ^+ for acceleration. Our analysis shows that PQ^+ can achieve optimal workload balance when $K = \sqrt{m}$ in practice, a finding confirmed by our experimental results. We conduct extensive experiments on 10 real-world datasets of various types, evaluate four representative graph processing algorithms, including Dijkstra's algorithm and algorithms for influence maximization. The results demonstrate that PQ^+ is the only approach that consistently speeds up algorithms on all graph types, achieving up to 2.65x speedup and 3.11x speedup compared with sequential and optimal parallel baselines, with stable performance under varied thread numbers.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions.

References

- [1] [n.d.]. 9th DIMACS Implementation Challenge - Shortest Paths. <https://www.diag.uniroma1.it/~challenge9/download.shtml>.
- [2] 2001. Vojtěch Jarník's work in combinatorial optimization. *Discrete Mathematics* 235, 1 (2001), 1–17.
- [3] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: a scalable relaxed priority queue. In *SIGPLAN*. ACM, 11–20.
- [4] M. D. Atkinson, Jörg-Rüdiger Sack, Nicola Santoro, and Thomas Strothotte. 1986. Min-Max Heaps and Generalized Priority Queues. *Commun. ACM* 29, 10 (1986), 996–1000.
- [5] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. 2023. Exponentially Faster Massively Parallel Maximal Matching. *J. ACM* 70, 5 (2023), 34:1–34:18.
- [6] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *SPAA*, Christian Scheideler and Seth Gilbert (Eds.). ACM, 443–454.
- [7] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time Bounds for Selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461.
- [8] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.
- [9] Ran Duan, Jiayi Mao, Xiao Mao, Xinkai Shu, and Longhui Yin. 2025. Breaking the Sorting Barrier for Directed Single-Source Shortest Paths. In *STOC*. ACM, 36–44.
- [10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On Power-law Relationships of the Internet Topology. In *SIGCOMM*. ACM, 251–262.
- [11] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *SIGMOD*. ACM, 495–510.
- [12] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *ACM Trans. Database Syst.* 43, 4 (2018), 18:1–18:39.
- [13] John R. Gilbert. 2018. Graph Algorithms in the Language of Linear Algebra: How Did We Get Here, and Where Do We Go Next?. In *IPDPS*. IEEE Computer Society, 268.
- [14] Qintian Guo, Chen Feng, Fangyuan Zhang, and Sibao Wang. 2023. Efficient Algorithm for Budgeted Adaptive Influence Maximization: An Incremental RR-set Update Approach. *Proc. ACM Manag. Data* 1, 3 (2023), 207:1–207:26.

- [15] Gregory Gutin and Abraham P Punnen. 2006. *The traveling salesman problem and its variations*. Vol. 12. Springer Science & Business Media.
- [16] Magnús M. Halldórsson and Jaikumar Radhakrishnan. 1994. Greed is good: approximating independent sets in sparse and bounded-degree graphs. In *STOC*. ACM, 439–448.
- [17] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.
- [18] Victor P. Il'ev and Nick Linker. 2006. Performance guarantees of a greedy algorithm for minimizing a supermodular set function on comatroid. *Eur. J. Oper. Res.* 171, 2 (2006), 648–660.
- [19] Intel oneAPI Threading Building Blocks Contributors. 2022. oneTBB Source Code Repository. <https://github.com/oneapi-src/oneTBB>. Commit hash: f1862f3 (v2022.3.0).
- [20] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Shana Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José E. Moreira, John D. Owens, Carl Yang, Marcin Zalewski, and Timothy G. Mattson. 2022. Introduction to GraphBLAS. In *Massive Graph Analytics*. Chapman and Hall/CRC, 507–524.
- [21] Jalal Khalil, Da Yan, Guimu Guo, and Lyuheng Yuan. 2022. Parallel mining of large maximal quasi-cliques. *VLDB J.* 31, 4 (2022), 649–674.
- [22] Samir Khuller, Anna Moss, and Joseph Naor. 1999. The Budgeted Maximum Coverage Problem. *Inf. Process. Lett.* 70, 1 (1999), 39–45.
- [23] Andreas Krause and Daniel Golovin. 2014. Submodular function maximization. *Tractability* 3, 71-104 (2014), 3.
- [24] Alan Kuhnle. 2019. Interlaced Greedy Algorithm for Maximization of Submodular Functions in Nearly Linear Time. In *NeurIPS*. 2371–2381.
- [25] Jérôme Kunegis. [n.d.]. The KONECT Project. <http://konect.cc/>.
- [26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>.
- [27] Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *OPDIS (Lecture Notes in Computer Science)*, Vol. 8304. Springer, 206–220.
- [28] Qingyuan Linghu, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2020. Global Reinforcement of Social Networks: The Anchored Coreness Problem. In *SIGMOD*. ACM, 2211–2226.
- [29] Yajing Liu, Edwin K. P. Chong, Ali Pezeshki, and Zhenliang Zhang. 2020. Submodular optimization problems and greedy strategies: A survey. *Discret. Event Dyn. Syst.* 30, 3 (2020), 381–412.
- [30] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727.
- [31] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. ACM, 135–146.
- [32] Ulrich Meyer and Peter Sanders. 2003. [Delta]-stepping: a parallelizable shortest path algorithm. *J. Algorithms* 49, 1 (2003), 114–152.
- [33] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. 1978. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.* 14, 1 (1978), 265–294.
- [34] You Peng, Jeffrey Xu Yu, and Sibow Wang. 2023. PSPC: Efficient Parallel Shortest Path Counting on Large-Scale Graphs. In *ICDE*. IEEE, 896–908.
- [35] R. C. Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401.
- [36] William W. Pugh. 1990. Concurrent maintenance of skip lists.
- [37] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2002. *The Boost Graph Library - User Guide and Reference Manual*. Pearson / Prentice Hall.
- [38] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence Maximization in Near-Linear Time: A Martingale Approach. In *SIGMOD*. ACM, 1539–1554.
- [39] Youze Tang, Xiaokui Xiao, and Yanchen Shi. [n.d.]. Influence maximization: near-optimal time complexity meets practical efficiency. In *SIGMOD*. ACM, 75–86.
- [40] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (2013), 193–204.
- [41] Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. 2023. Fast and Space-Efficient Parallel Algorithms for Influence Maximization. *Proc. VLDB Endow.* 17, 3 (2023), 400–413.
- [42] Jiadong Xie, Fan Zhang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2023. Minimizing the Influence of Misinformation via Vertex Blocking. In *ICDE*. IEEE, 789–801.
- [43] Jiadong Xie, Fan Zhang, Kai Wang, Jialu Liu, Xuemin Lin, and Wenjie Zhang. 2024. Influence Minimization via Blocking Strategies. (2024).
- [44] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation

- on Real-World Graphs. *Proc. VLDB Endow.* 7, 14 (2014), 1981–1992.
- [45] Weihao Yu, Dian Ouyang, Fan Zhang, Xiang Zhao, Shen Su, Xuemin Lin, and Zhihong Tian. 2025. Hops Can be Constrained: Efficient Distance Queries on Large Time-Dependent Road Networks. *Proc. ACM Manag. Data* 3, 6 (2025), 1–25.
- [46] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. Finding Critical Users for Social Network Engagement: The Collapsed k-Core Problem. In *AAAI*. AAAI Press, 245–251.
- [47] Junhua Zhang, Wentao Li, Long Yuan, Lu Qin, Ying Zhang, and Lijun Chang. 2022. Shortest-Path Queries on Complex Networks: Experiments, Analyses, and Improvement. *Proc. VLDB Endow.* 15, 11 (2022), 2640–2652.
- [48] Wenyue Zhao, Yang Cao, Peter Buneman, Jia Li, and Nikos Ntarmos. 2024. Automating Vectorized Distributed Graph Computation. *Proc. ACM Manag. Data* 2, 6 (2024), 254:1–254:27.

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026