Efficient algorithms and implementations for parallel SSSP

Xiaojun Dong[1], Yan Gu[1], Yihan Sun[1] and Yunming Zhang[2]

[1] University of California, Riverside

[2] Massachusetts Institute of Technology

Models and Background

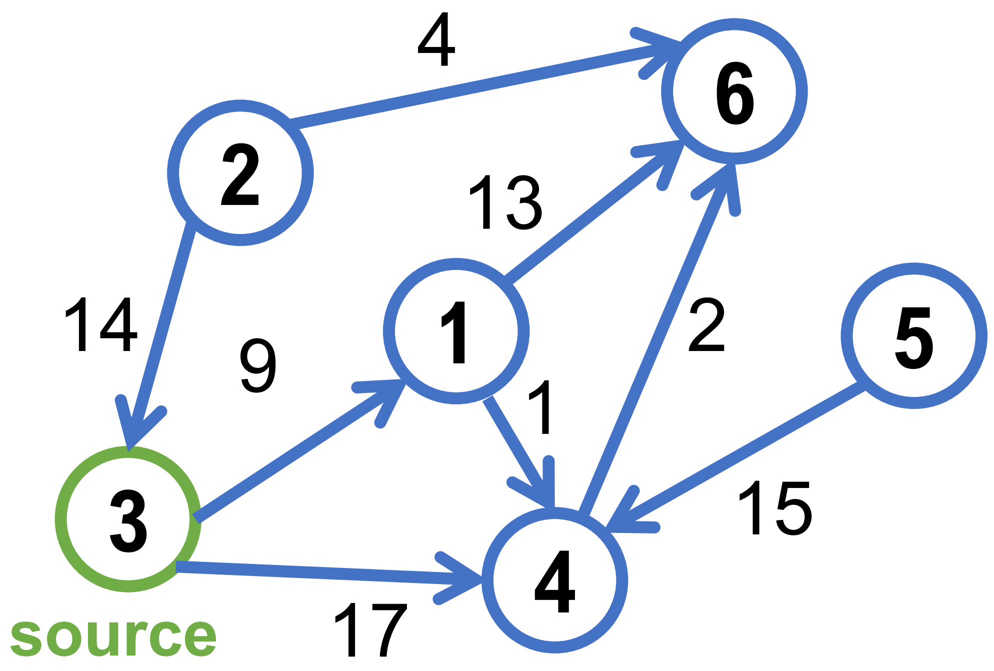
- Shared-memory multi-core setting
- Work-span model
- Work: total number of operations (sequential running time)
- Span (depth): longest dependence chain (parallel time)
- We’ll see both theoretical analysis and experimental results in this talk ☺

Single-source shortest paths (SSSP)

- **On graph $G = (V, E, w)$, with edge weight function $w: e \mapsto \mathbb{R}^+$ and a source $s \in V$, compute the shortest distances (paths) of all other vertices to s. Let $n = |V|$, $m = |E|$.**

- **Dijkstra's algorithm + priority queue**
  - Work efficient: process each vertex/edge once
  - But hard to parallelize?

- **Bellman-Ford**
  - Redundant work: process multiple times
  - But parallelism is straightforward



SSSP is notoriously hard in parallel

Theoretical algorithms:
[BGST16], [Cohen97], [Cohen00], [KS97], [Meyer01], [Meyer02], [SS99], [Spencer97], [UY91]
Approximate: [ASZ20], [CFR20], [EN19], [Li20], [MPVX15]

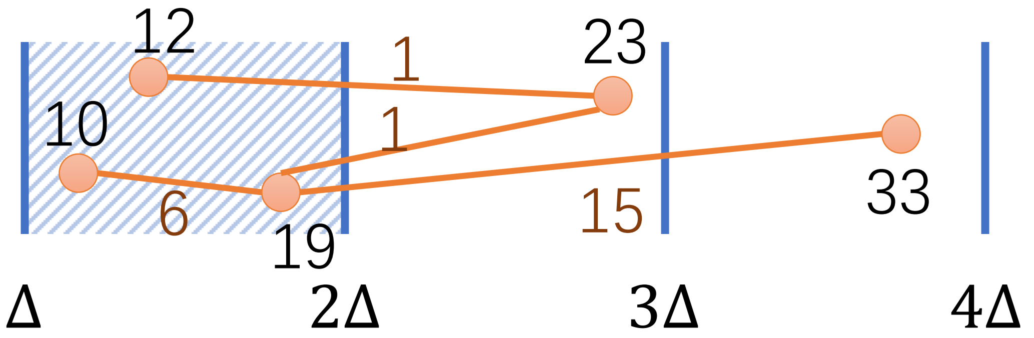
Practical implementations are based on Δ-stepping [Meyer-Sanders 03]:
Julienne [DBS17], **GAPBS** [BAP15], **Galois** [NLP13], **GraphIt** [ZBC+20]
Other platforms: [BPG+17], [DBG+14], [MAB+10], [ZCZM16], [WDY+16]

Parallel / concurrent priority queues:
PRAM [BDM+96], [CH94], [CDP96], [DPS96], [RCP+94]
Concurrent: [AKLS15], [CMH14], [HKP+13], [LJ13], [LS12], [SL20], [ST05], [ZMS19]
Others: [BKS15], [Sanders98]

Practical implementations: Δ-stepping

- Relax those close to the source, but multiple of them together in parallel

For each step
    While there remain potential unsettled vertices
        For each outgoing edge
            Relax the neighbor

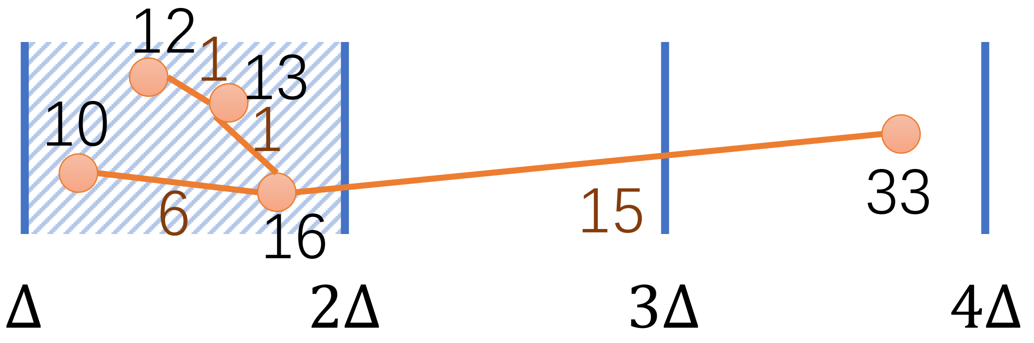


$\Delta = 10$

Practical implementations: Δ-stepping

- Relax those close to the source, but multiple of them together in parallel

For each step
  While there remain potential unsettled vertices
    For each outgoing edge
      Relax the neighbor



$\Delta = 10$

Practical implementations: Δ-stepping

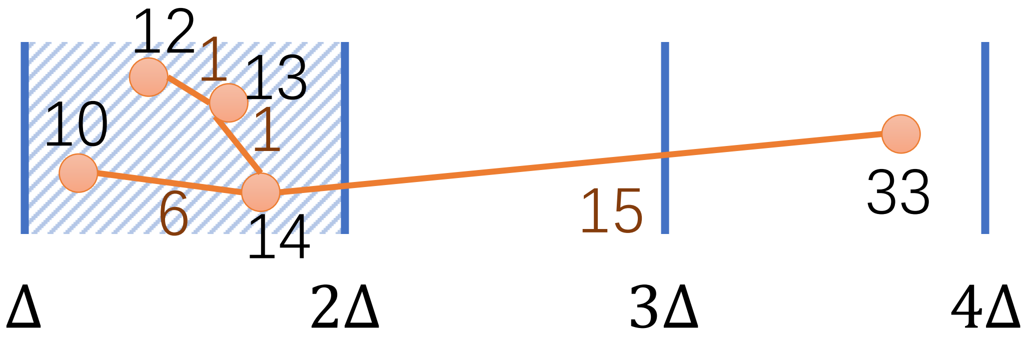
- **Relax those close to the source, but multiple of them together in parallel**

For each step

    While there remain potential unsettled vertices

        For each outgoing edge

            Relax the neighbor

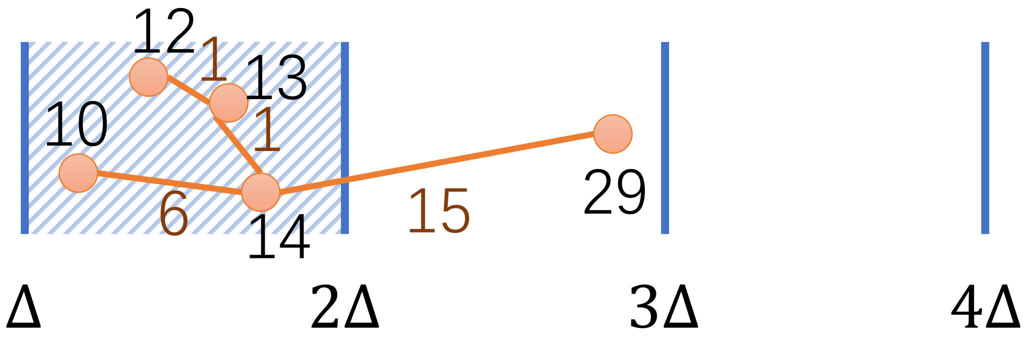


$\Delta = 10$

Practical implementations: Δ-stepping

- **Relax those close to the source, but multiple of them together in parallel**

For each step
    While there remain potential unsettled vertices
        For each outgoing edge
            Relax the neighbor

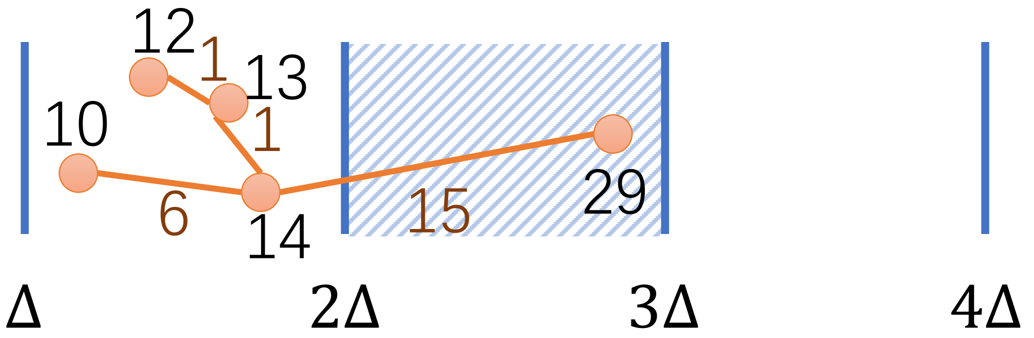


$\Delta = 10$

Practical implementations: Δ-stepping

- Relax those close to the source, but multiple of them together in parallel

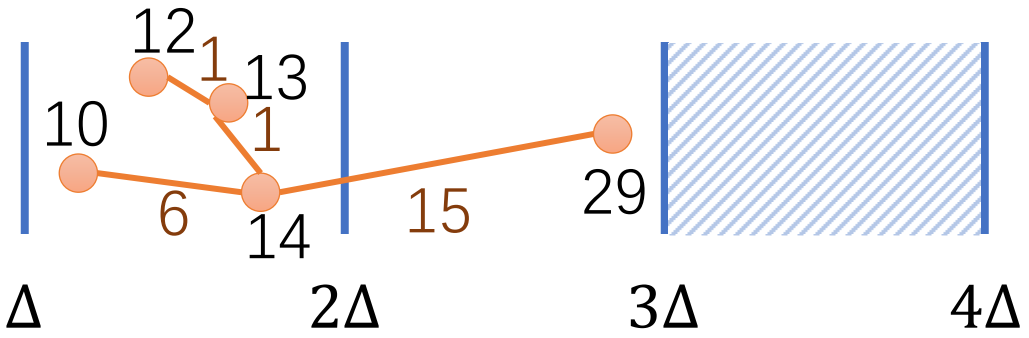
For each step
    While there remain potential unsettled vertices
        For each outgoing edge
            Relax the neighbor



$\Delta = 10$

Practical implementations: Δ-stepping

- Relax those close to the source, but multiple of them together in parallel
  - Edges crossing boundary: Dijkstra
  - Edges within a single range: Bellman-Ford
  - Try to avoid redundant work (not relaxing vertices far away), but support parallelism



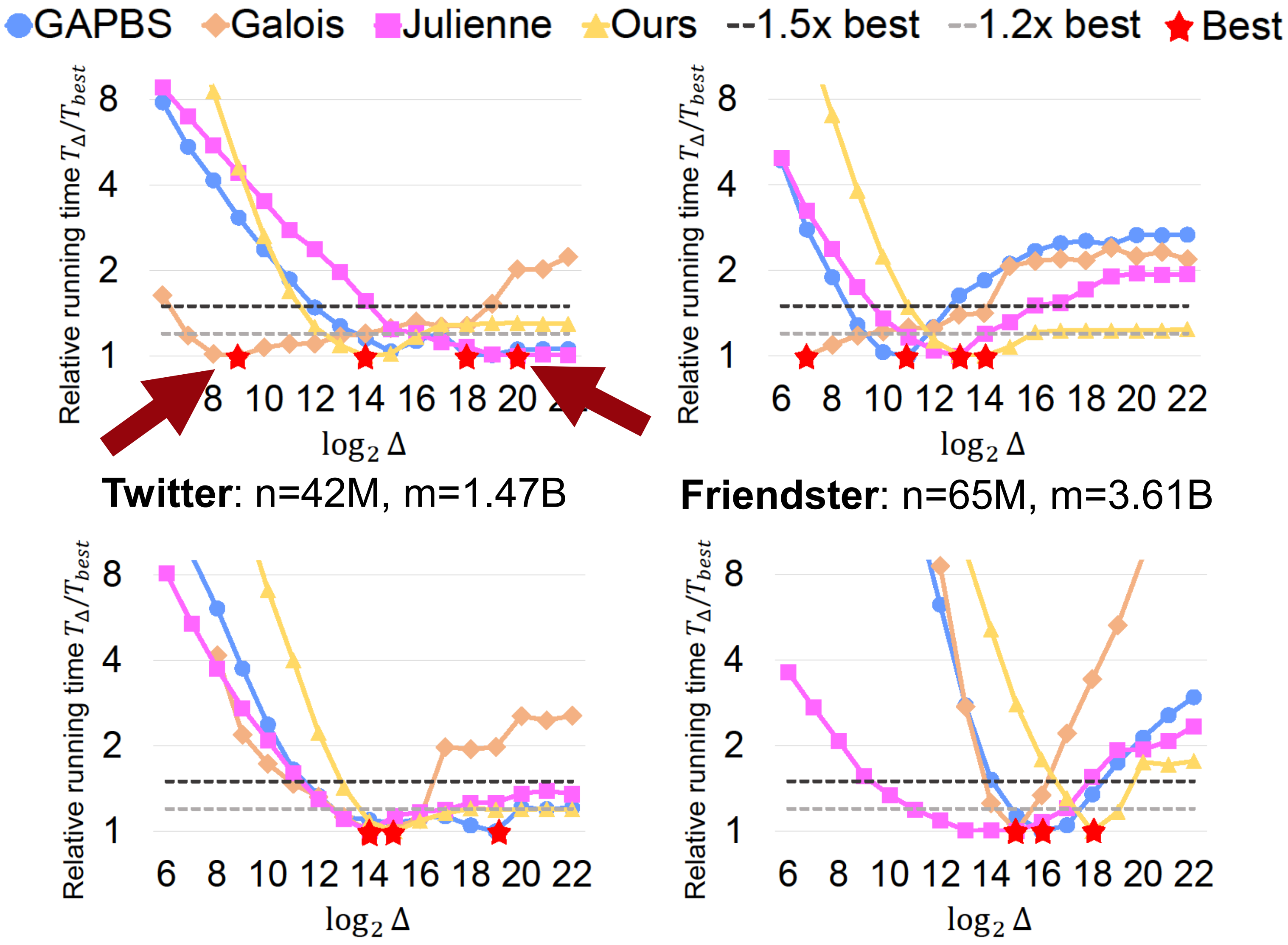
$\Delta = 10$

Δ-stepping: challenges

- **In theory, no known bounds for general graphs**
  - Has been analyzed on random graphs
  - The span can be as larger as O(n) with a shallow shortest-path tree

- **In practice, how to select the best Δ?**

Δ-stepping and Δ



Twitter: n=42M, m=1.47B

Friendster: n=65M, m=3.61B

WebGraph: n=89M, m=2.04B

RoadUSA: n=12M, m=32M

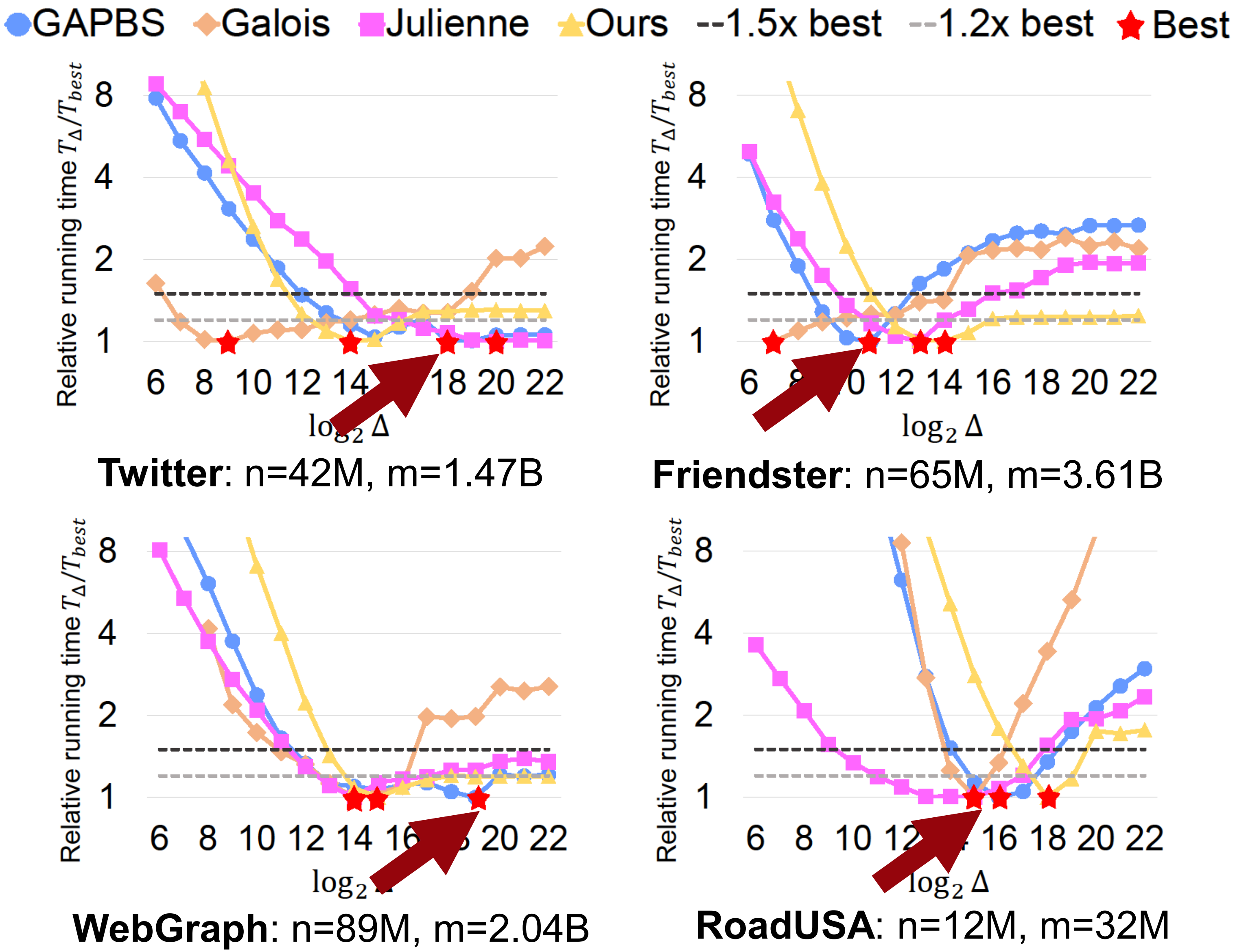
- (Relative running time)

Same graph:

- In each of the figure
- Compare the red stars for all curves

- Best Δ varies for different implementations
- 2^{11} on Twitter!

Δ-stepping and Δ



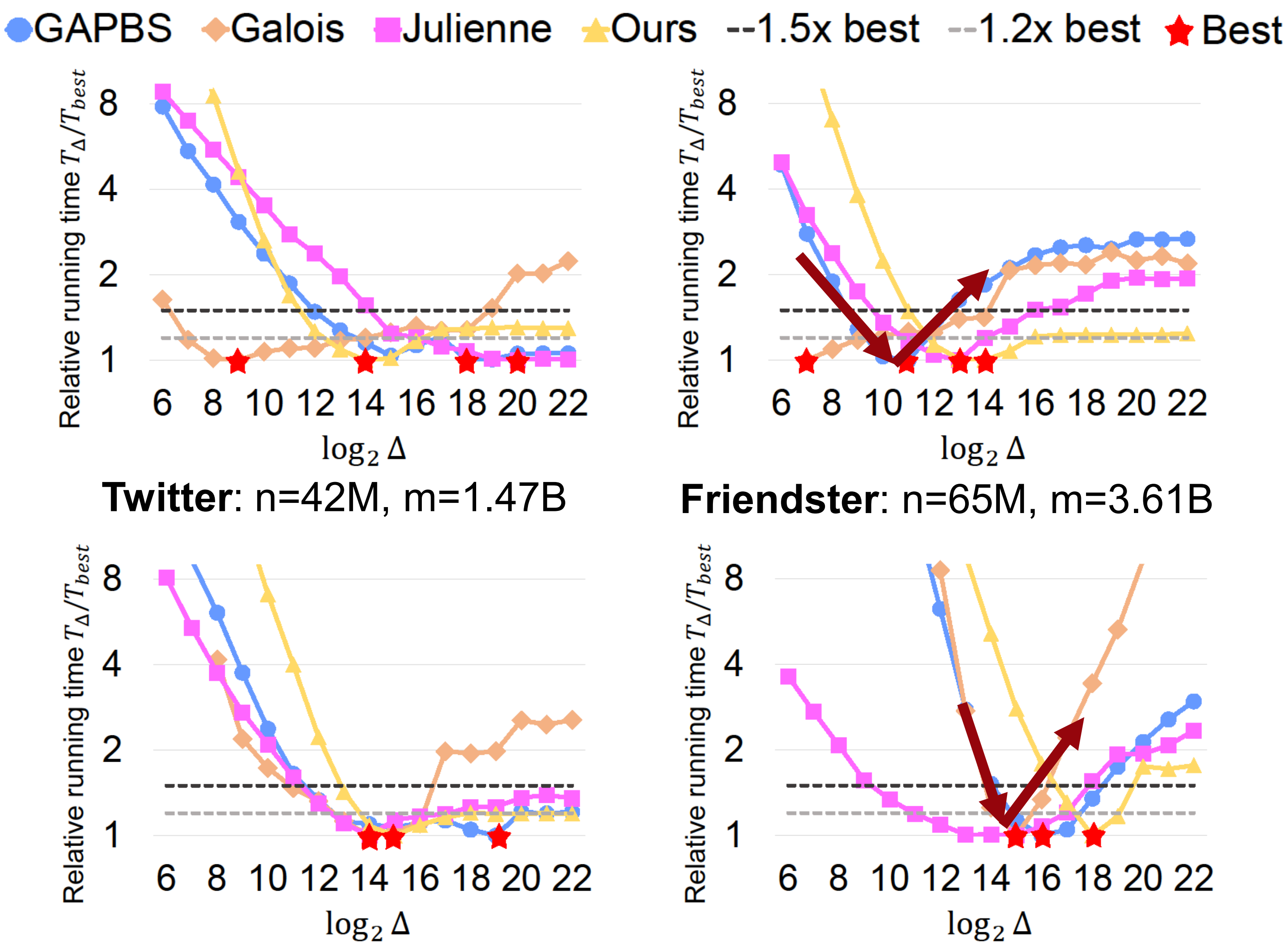
- (Relative running time)

Same Implementation:

- For four figures
- Compare the red stars of the curves with the same color

- Best Δ varies for different graphs
  - 2^9 for GAPBS!
  - First three graphs has the same edge-weight distribution

Δ-stepping and Δ



Twitter: n=42M, m=1.47B

Friendster: n=65M, m=3.61B

WebGraph: n=89M, m=2.04B

RoadUSA: n=12M, m=32M

- (Relative running time)

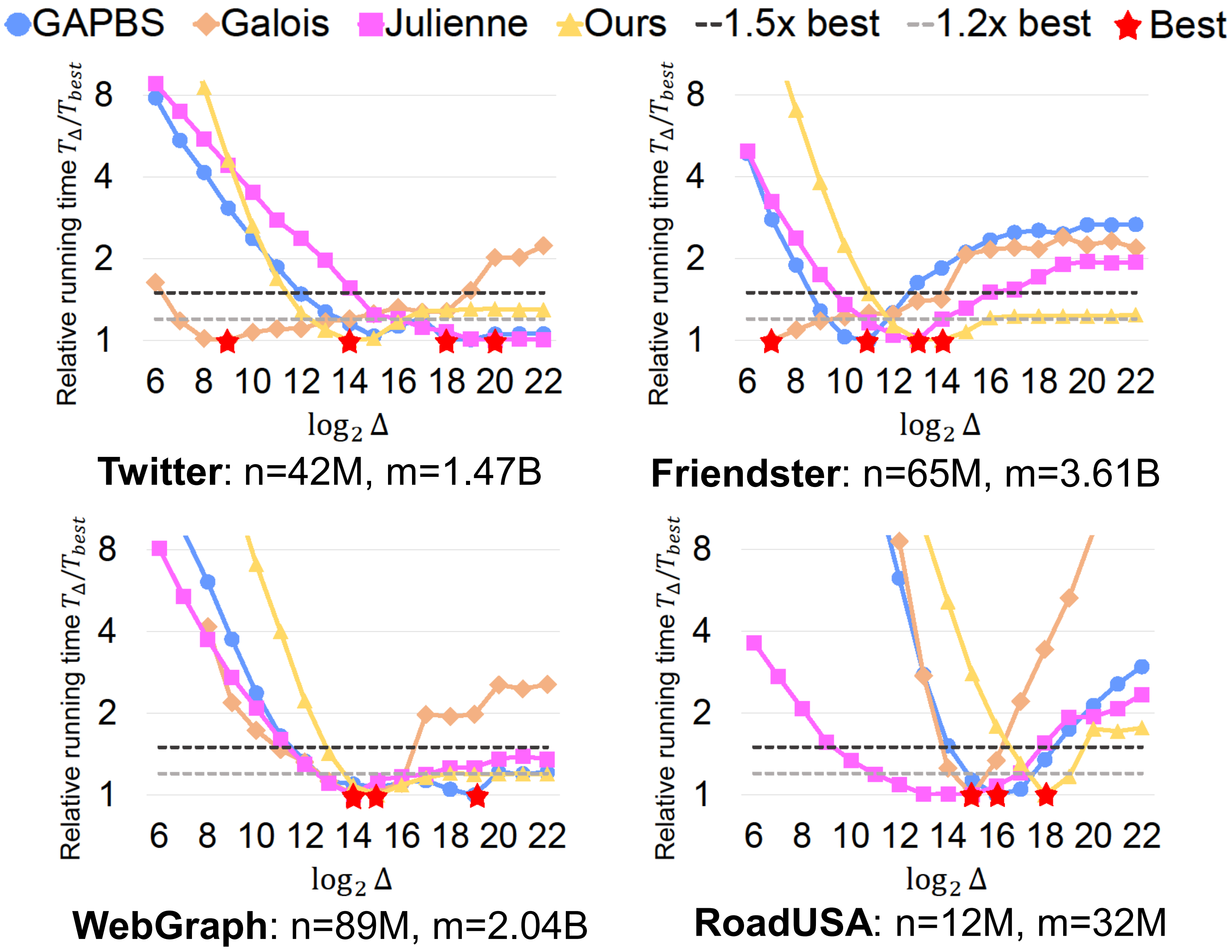
Same graph & same implementation

- Each curve

- **Sensitive** to the value of Δ

Δ-stepping and Δ

- (Relative running time)



Usually have to first exhaustively search for the best Δ for each graph-implementation

SSSP is notoriously hard in parallel

Theoretical algorithms:
[BGST16], [Cohen97], [Cohen00], [KS97], [Meyer01], [Meyer02], [SS99], [Spencer97], [UY91]
Approximate: [ASZ20], [CFR20], [EN19], [Li20], [MPVX15]

Practical implementations are based on Δ-stepping [Meyer-Sanders 03]:
Julienne [DBS17], **GAPBS** [BAP15], **Galois** [NLP13], **GraphIt** [ZBC+20]
Other platforms: [BPG+17], [DBG+14], [MAB+10], [ZCZM16], [WDY+16]

No interesting worst-case bounds

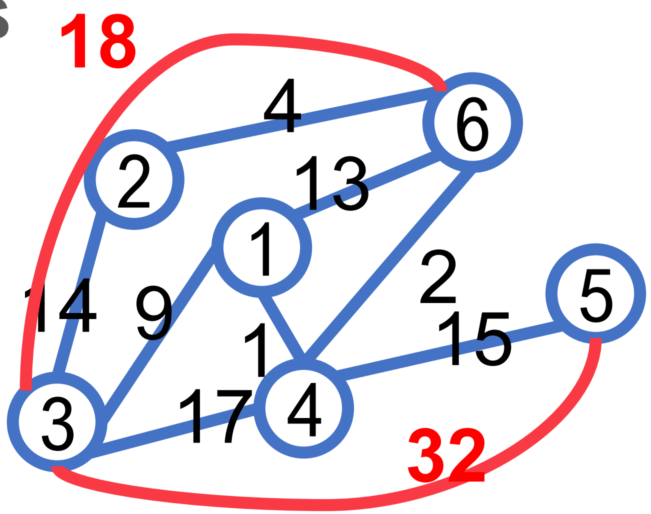
Needs tunning for parameter

Parallel / concurrent priority queues:
PRAM [BDM+96], [CH94], [CDP96], [DPS96], [RCP+94]
Concurrent: [AKLS15], [CMH14], [HKP+13], [LJ13], [LS12], [SL20], [ST05], [ZMS19]
Others: [BKS15], [Sanders98]

Theoretical parallel SSSP algorithms

- **Work-span tradeoff ("transitive closure bottleneck" [KR90])**
  - No known exact solution enables small work and span simultaneously
  - Low span is hard … consider a chain
  - Usually needs preprocessing
- **No known implementation (or slower than Δ-stepping)**

- **Need a lot shortcuts: to achieve $O(n^{1-\epsilon})$ span requires $\Omega(n^{1+\epsilon})$ shortcuts**
  - Shortcuts increase m, which means more work!



SSSP is notoriously hard in parallel

Theoretical algorithms:
[BGST16], [Cohen97], [Cohen00], [KS97], [Meyer01], [Meyer02], [SS99], [Spencer97], [UY91]
Approximate: [ASZ20], [CFR20], [EN19], [Li20], [MPVX15]

No implementations

Practical implementations are based on Δ-stepping [Meyer-Sanders 03]:
Julienne [DBS17], **GAPBS** [BAP15], **Galois** [NLP13], **GraphIt** [ZBC+20]
Other platforms: [BPG+17], [DBG+14], [MAB+10], [ZCZM16], [WDY+16]

No interesting worst-case bounds

Needs tunning for parameter

Parallel / concurrent priority queues:
PRAM [BDM+96], [CH94], [CDP96], [DPS96], [RCP+94]
Concurrent: [AKLS15], [CMH14], [HKP+13], [LJ13], [LS12], [SL20], [ST05], [ZMS19]
Others: [BKS15], [Sanders98]

Parallel / concurrent priority queues for SSSP

- **Parallelize or support concurrent operations in Dijkstra's algorithm**
  - Enable multiple updates (decreaseKey)
  - Later work allows multiple extractMin (approximately)

- **However, Dijkstra's algorithm itself is very sequential**
  - No interesting span bounds
  - Slow in practice

SSSP is notoriously hard in parallel

Theoretical algorithms:
[BGST16], [Cohen97], [Cohen00], [KS97], [Meyer01], [Meyer02], [SS99], [Spencer97], [UY91]
Approximate: [ASZ20], [CFR20], [EN19], [Li20], [MPVX15]

No implementations

Practical implementations are based on Δ-stepping [Meyer-Sanders 03]**:**
Julienne [DBS17], **GAPBS** [BAP15], **Galois** [NLP13], **GraphIt** [ZBC+20]
Other platforms: [BPG+17], [DBG+14], [MAB+10], [ZCZM16], [WDY+16]

No worst-case bounds

Needs tunning

Parallel / concurrent priority queues:
PRAM [BDM+96], [CH94], [CDP96], [DPS96], [RCP+94]
Concurrent: [AKLS15], [CMH14], [HKP+13], [LJ13], [LS12], [SL20], [ST05], [ZMS19]
Others: [BKS15], [Sanders98]

No good span (based on Dijkstra)

Not as fast as Δ-stepping

We want to have parallel SSSP with …

Theoretical Efficiency

Practicality

Good abstraction and easy implementation

Our new SSSP solutions

Theoretical Efficiency

- Worst-case work and span bounds, matching previous bounds (under assumptions)
- Avoid shortcuts

Practicality

- Competitive or faster than all existing software
- Stable and parameter-insensitive performance

New priority queue ADT

- Simple algorithms on top
- Efficient cost bounds
- Efficient implementation for multiple algorithms

Our approach

Our approach

New framework: stepping algorithm framework

New ADT: Lazy-Batched Priority Queue (LaB-PQ)

New algorithm: ρ-stepping and Δ^*-stepping

Our approach

Existing algorithms:

- Dijkstra
- Bellman-Ford
- Radius-stepping
- Δ-stepping

New framework: stepping algorithm framework

New ADT: Lazy-Batched Priority Queue (LaB-PQ)

New algorithm: ρ-stepping and Δ^*-stepping

Our approach

Existing algorithms:

- Dijkstra
- Bellman-Ford
- Radius-stepping
- Δ-stepping

New algorithm:
ρ-stepping
Δ*-stepping

New framework: stepping algorithm framework

New ADT: Lazy-Batched Priority Queue (LaB-PQ)

Our approach

Existing algorithms:

Dijkstra

Bellman-Ford

Radius-stepping

Δ-stepping

New algorithm:

ρ-stepping

Δ*-stepping

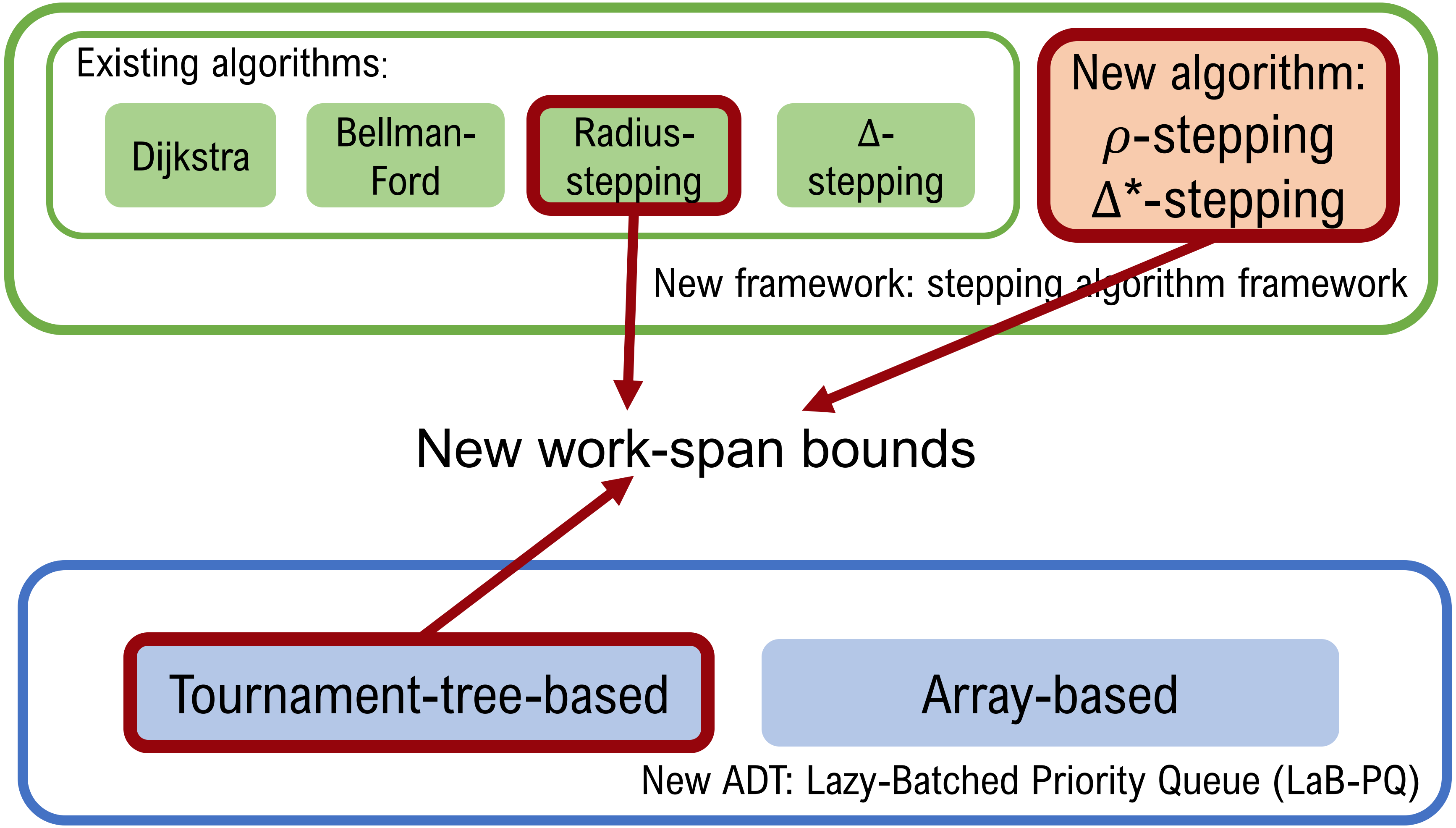
New framework: stepping algorithm framework

Tournament-tree-based

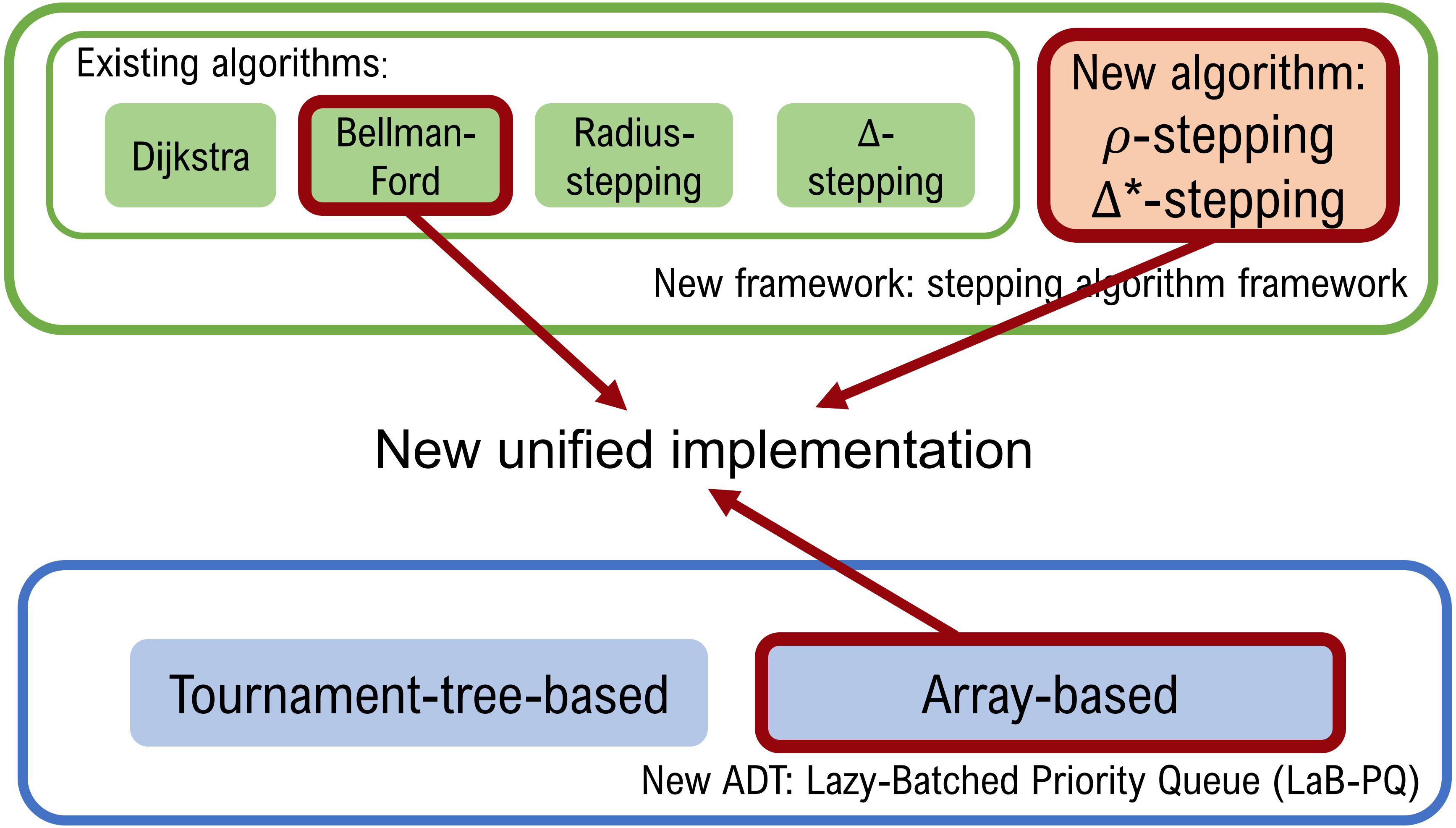
Array-based

New ADT: Lazy-Batched Priority Queue (LaB-PQ)

Our results: New or improved bounds



Our results: Efficient implmentations



Stepping algorithm framework

SSSP is notoriously hard in parallel

Radius stepping: BGST16

Δ-stepping
(also some based on Bellman-Ford)

Dijkstra

High-level similarities

- **Extract a subset of the vertices in the frontier and relax their neighbors**
  - Vertices with distances under a certain threshold

- **Repeat until all vertices are settled**

Stepping algorithm framework

tentative distance

Algorithm 1: The Stepping Algorithm Framework.

Input: A graph $G = (V, E, w)$ and a source node s.

Output: The graph distances $d(\cdot)$ from s.

$\boxed{\delta[\cdot]} \leftarrow +\infty$, associate δ to $Q \in \mathbb{PQ}$

$\delta[s] \leftarrow 0$, $Q.\textsc{Update}(s)$

while $|Q| > 0$ **do**

  ParallelForEach $u \in \boxed{Q.\textsc{Extract}(\textsc{ExtDist})}$ **do**

    ParallelForEach $v \in N(u)$ **do**

      if $\textsc{WriteMin}(\delta[v], \delta[u] + w(u, v))$ **then**

        $\boxed{Q.\textsc{Update}(v)}$

  Execute $\boxed{\textsc{FinishCheck}}$

return $\delta(\cdot)$

Compute the threshold in each "step"

Update the new distances to the PQ

Stepping algorithm framework

Dealing with substeps:
if a step does not finish, rerun

Algorithm	ExtDist	FinishCheck
Dijkstra [48]	$\theta \leftarrow \min_{v \in Q}(\delta[v])$	-
Bellman-Ford [13, 52]	$\theta \leftarrow +\infty$	-
Δ-Stepping [70]	$\theta \leftarrow i\Delta$	if no new $\delta[v] < i\Delta$, $i \leftarrow i+1$
Δ^*-Stepping (new)	$\theta \leftarrow i\Delta$	-
Radius-Stepping [26]	$\theta \leftarrow \min_{v \in Q}(\delta[v] + r_\rho(v))$	if there exists $\delta[v] < \theta$, do not recompute ExtDist
ρ-Stepping (new)	$\theta \leftarrow \rho$-th smallest $\delta[v]$ in Q	-

Stepping algorithm framework

Algorithm 1: The Stepping Algorithm Framework.

Input: A graph $G = (V, E, w)$ and a source node s.

Output: The graph distances $d(\cdot)$ from s.

$\delta[\cdot] \leftarrow +\infty$, associate δ to $Q \in \mathbb{PQ}$

$\delta[s] \leftarrow 0$, $Q.\textsc{Update}(s)$

while $|Q| > 0$ **do**

  ParallelForEach $u \in \boxed{Q.\textsc{Extract}(\textsc{ExtDist})}$ **do**

    ParallelForEach $v \in N(u)$ **do**

      if $\textsc{WriteMin}(\delta[v], \delta[u] + w(u, v))$ **then**

        $\boxed{Q.\textsc{Update}(v)}$

  Execute $\textsc{FinishCheck}$

return $\delta(\cdot)$

Compute the threshold in each “step”

Update the new distances to the PQ

Our ADT and data structure

Motivated by the “batch-dynamic” setting

- **The batch-dynamic data structures take a batch (bulk) of updates or queries and execute them in parallel**
  - Can usually design efficient parallel algorithms with low work and span
  - Some examples: hashtables **[SB14]**, search trees **[BFS16, BFS18]**, binary heap **[WYGS20]**, dynamic Euler-tour **[TDB19]**, rake-compress trees **[AABD’19]**

What operations needed for the stepping algorithms?

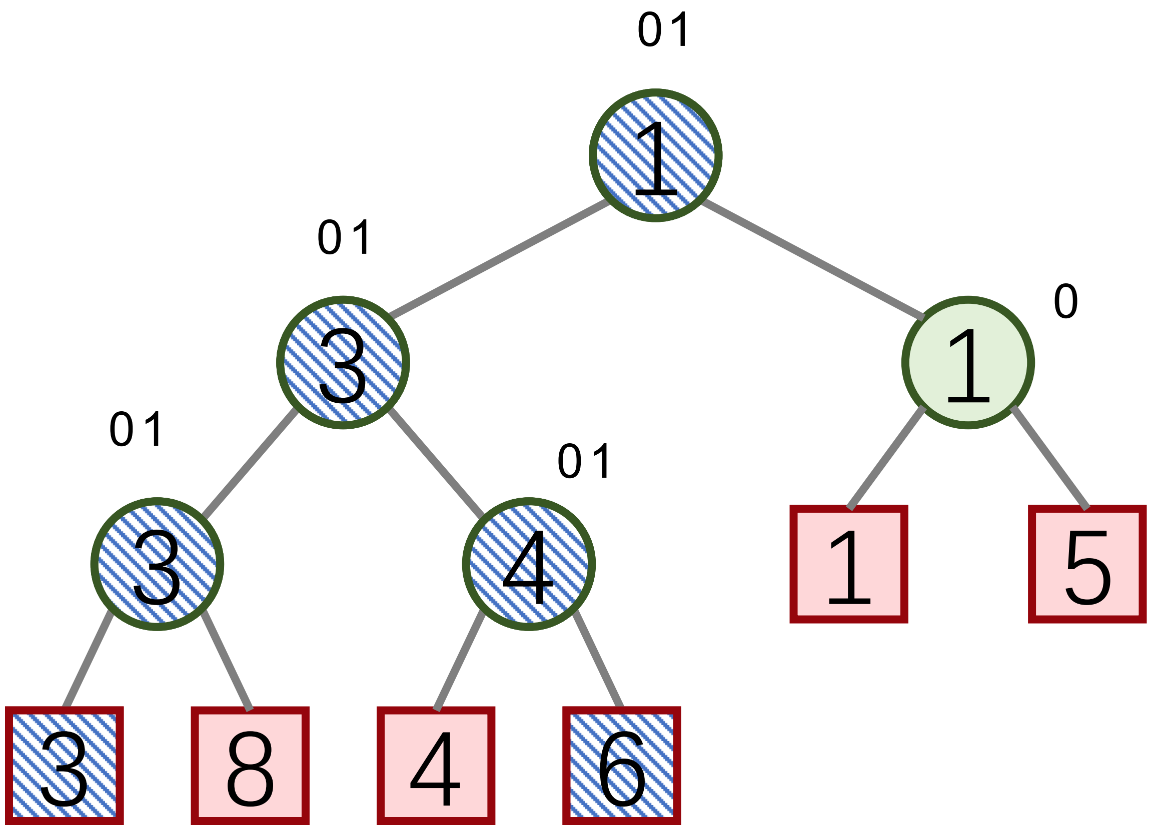
- Update a vertex's state in the priority queue (insert/decrease-key)
- Extract all vertices with keys below a certain threshold

Lab-PQ Interface

- **Update: commit an update to the data structure**
  - but **not execute immediately**
- **Extract: report and delete a batch of elements**
  - First apply all previous changes **in parallel**

A theoretically-efficient implementation

- **Use a tournament tree**
  - Since all entries are leaf nodes, dealing with concurrency issues is much easier

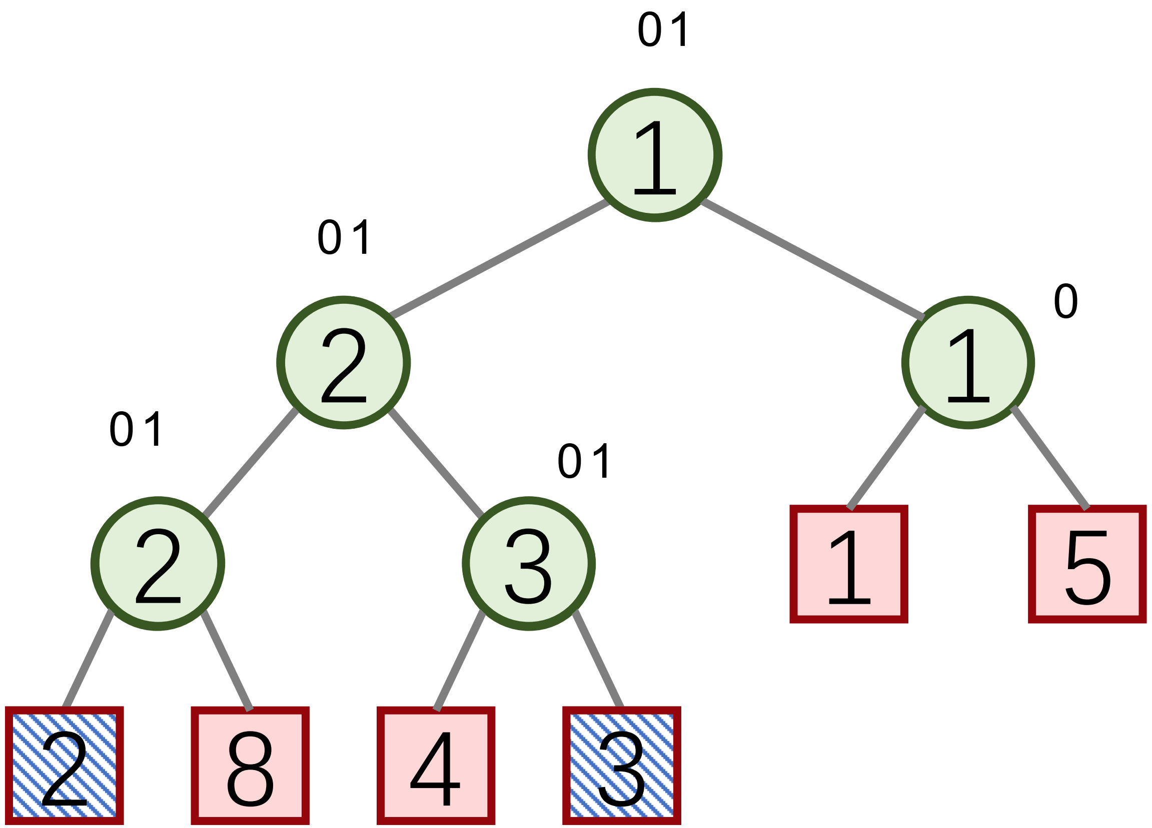


Update()

- **Mark a bit for the path from the leaf to root when update**
- **When updates are in parallel, use test-and-set**
- **Only continue when test-and-set succeed**

A theoretically-efficient implementation

- **Use a tournament tree**
  - Since all entries are leaf nodes, dealing with concurrency issues is much easier



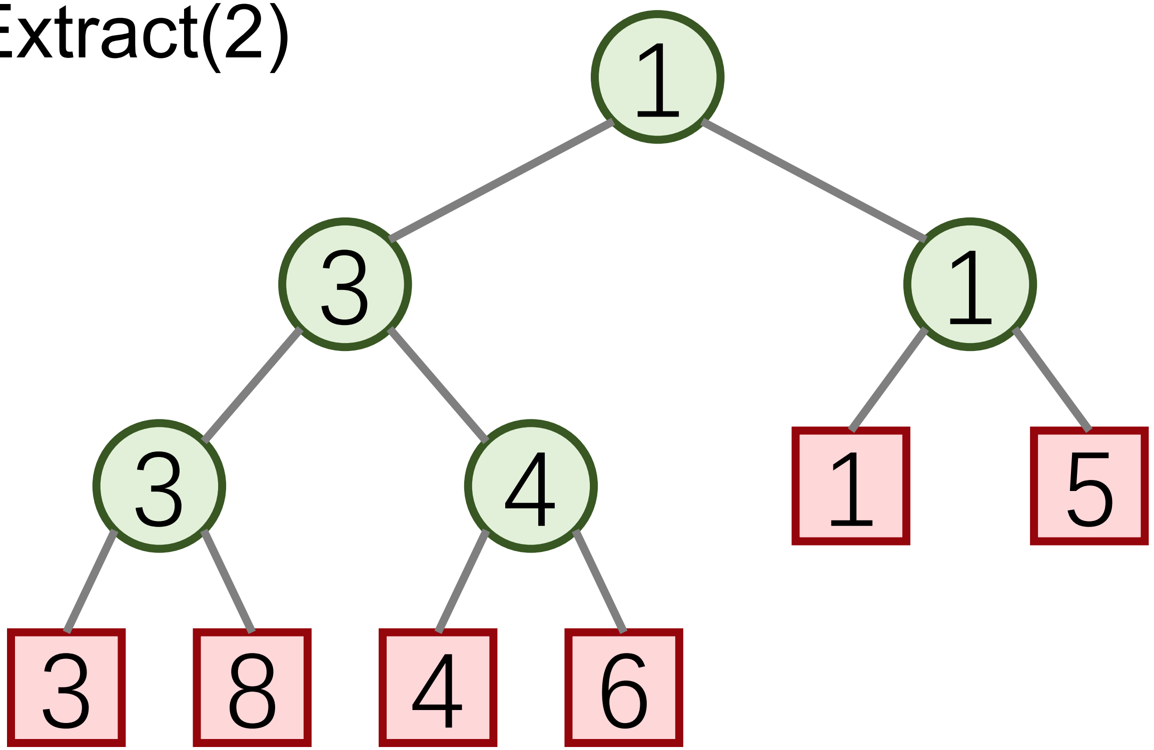
Apply all modification

- **Apply the batch using divide-and-conquer**
- **Skip a subtree if not marked**
- **Otherwise deal with two subtrees in parallel**

A theoretically-efficient implementation

- **Use a tournament tree**
  - Since all entries are leaf nodes, dealing with concurrency issues is much easier

Extract(2)

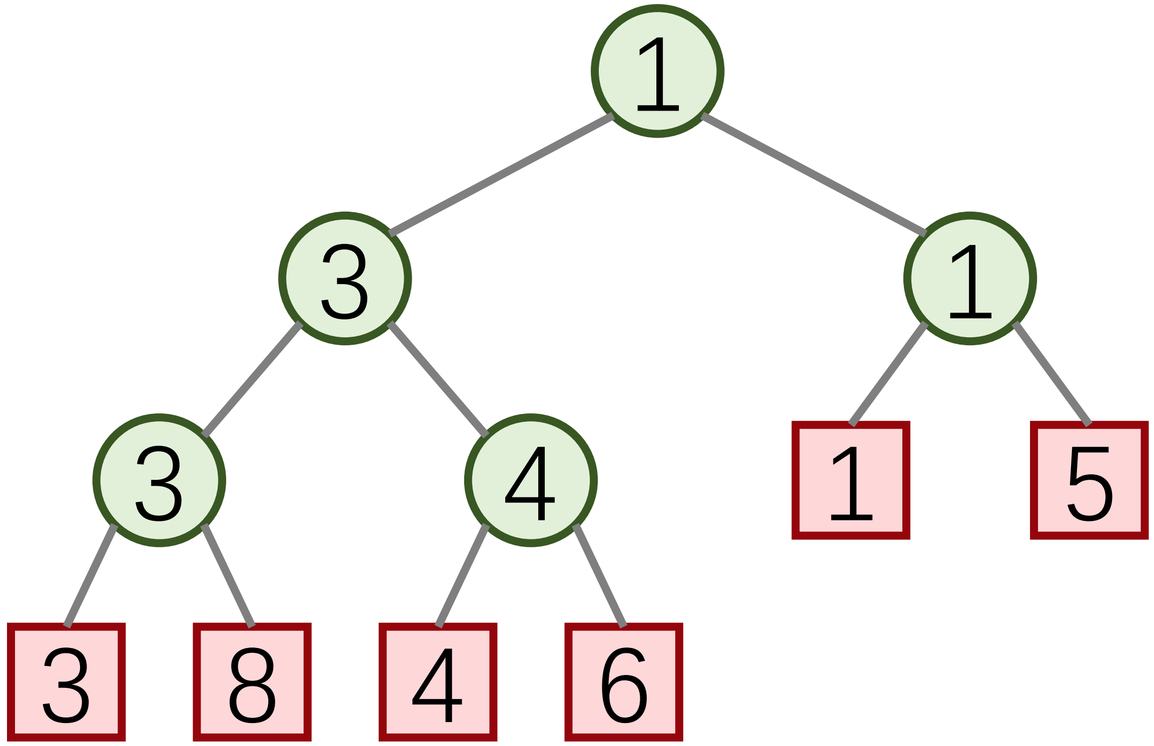


Extract($\boldsymbol{\theta}$)

- **Extract everything $\leq \theta$**
- **Skip a subtree with key $> \theta$**
- **Update internal keys and remove marks**

A theoretically-efficient implementation

- **Use a tournament tree**
  - Since all entries are leaf nodes, dealing with concurrency issues is much easier



- **If b leaves are involved in this batch, in total $O\left(b \log \frac{n}{b}\right)$ nodes are visited**
- **Each is visited a constant number of times**

Theoretical analysis

Theoretical analysis based on (k, ρ) graph

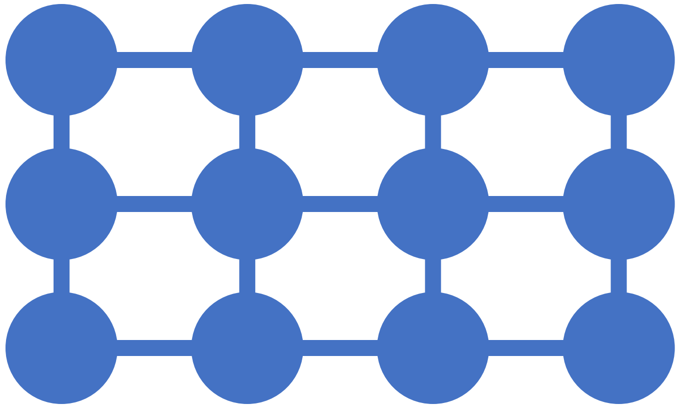
- **Without shortcut, it seems hard to show any span bounds $o(n)$ for general graphs …**

- **Consider some graph invariants?**
  - $\tilde{O}(n)$ span bound for Bellman-Ford should be understood as $\tilde{O}(d)$, where d is the shortest path tree depth

- **(k, ρ)-graph [BGST'16]: each vertex can reach its ρ closest vertices in k hops**

Theoretical analysis based on (k, ρ) graph

- **$(\boldsymbol{k}, \boldsymbol{\rho})$-graph [BGST'16]: each vertex can reach its $\boldsymbol{\rho}$ closest vertices in $\boldsymbol{k}$ hops**
  - Fix $\boldsymbol{\rho}$, use $\boldsymbol{k_\rho}$ as the smallest $\boldsymbol{k}$ to make a graph a $(\boldsymbol{k}, \boldsymbol{\rho})$-graph
  - $\boldsymbol{k_n}$ is the shortest path tree depth
  - What are the values of k_ρ for real-world graphs?

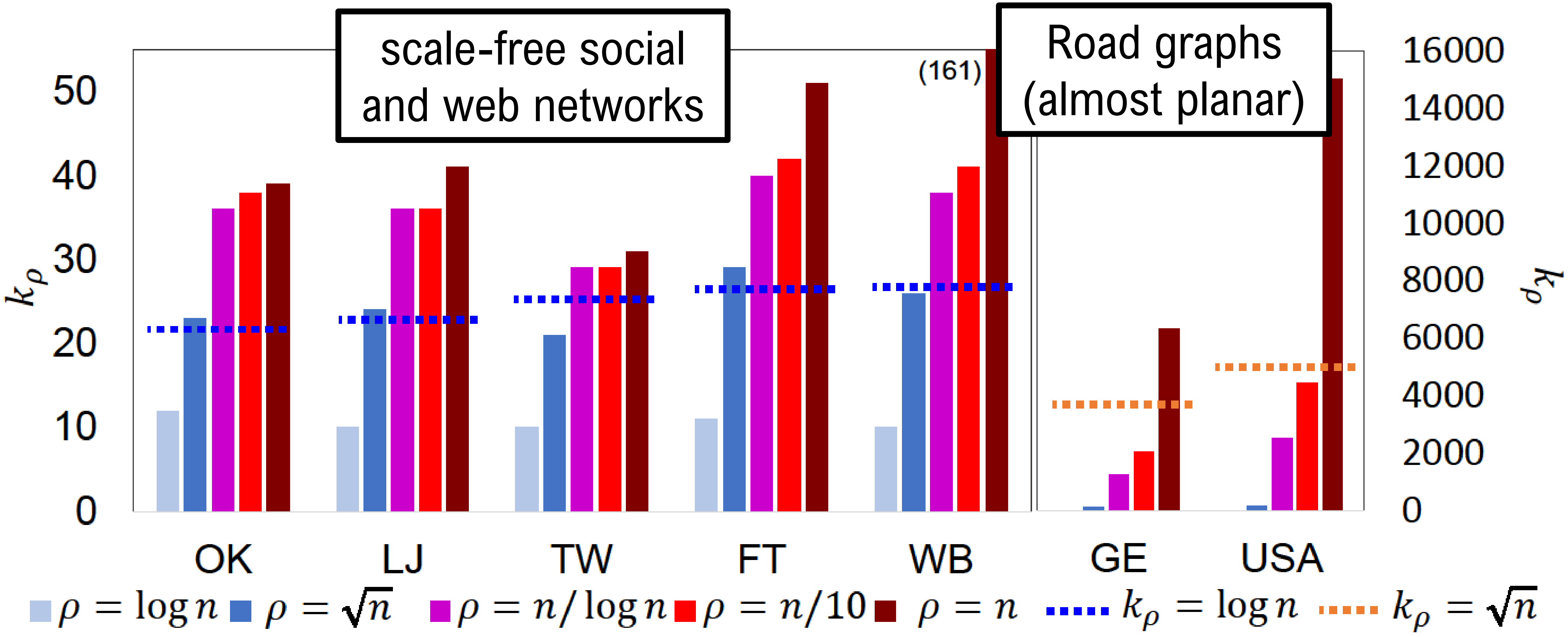


$(1,3)$-graph $\quad k_3 = 1$

$(2,4)$-graph $\quad k_4 = 2$

$(2,6)$-graph $\quad k_6 = 2$

$k - \rho$ properties for real-world graphs



- Number of vertices around 10^6 to 10^8
- For scale-free networks, k_ρ is usually small. $O(\log n)$ even for large ρ
- For road graphs, k_ρ varies significantly with ρ

Theoretical results

- **Extraction lemma, distribution lemma, data structure costs**

Algorithm	Work: Tournament-tree-based	Work: Array-based	Span	Previous Best: Work	Previous Best: Span
Dijkstra [17, 31]	$O\left(m\log\frac{n^2}{m}\right)$	$O(m+n^2)$	$O(n\log n)$	$O(m\log n)$	same
Bellman-Ford [10, 33]	$O(k_n m)$	$O(k_n m)$	$O(k_n\log n)$	same	same
Δ^*-stepping	$O\left(k_n m\log\frac{nL}{m\Delta}\right)$	$O\left(k_n m+\frac{k_n n(\Delta+L)}{\Delta}\right)$	$\boxed{O\left(\left(\frac{k_n(\Delta+L)}{\Delta}\right)\log n\right)}$	-	-
Radius-stepping† [16]	$O\left(k_\rho m\log\frac{n^2\log\rho L}{m\rho}\right)$ (U)	$O\left(k_\rho m+\frac{k_\rho n^2}{\rho}\cdot\log\rho L\right)$ (U)	$O\left(\frac{k_\rho n}{\rho}\cdot\log\rho L\log n\right)$ (U)	$O(k_\rho m\log n)$ (U)	same
Shi-Spencer† [58]	$O\left((m+n\rho)\log\frac{n^2}{m+n\rho}\right)$ (U)	$O\left(m+n\rho+\frac{n^2}{\rho}\right)$ (U)	$O\left(\frac{n\log n}{\rho}\right)$ (U)	$O((m+n\rho)\log n)$ (U)	same
ρ-Stepping	$O\left(k_n m\log\frac{n^2}{m\rho}\right)$	$O\left(k_n m+\frac{n^2k_\rho}{\rho}\right)$ (U) $O\left(k_n m+\frac{n^2k_n}{\rho}\right)$	$O\left(\frac{k_\rho n\log n}{\rho}\right)$ (U) $\boxed{O\left(\frac{k_n n\log n}{\rho}\right)}$	-	-

Assume smallest edge weight is 1, L is the largest edge weight, (U) means the bound works only on undirected graphs

Theoretical results

- **Extraction lemma, distribution lemma, data structure cost**

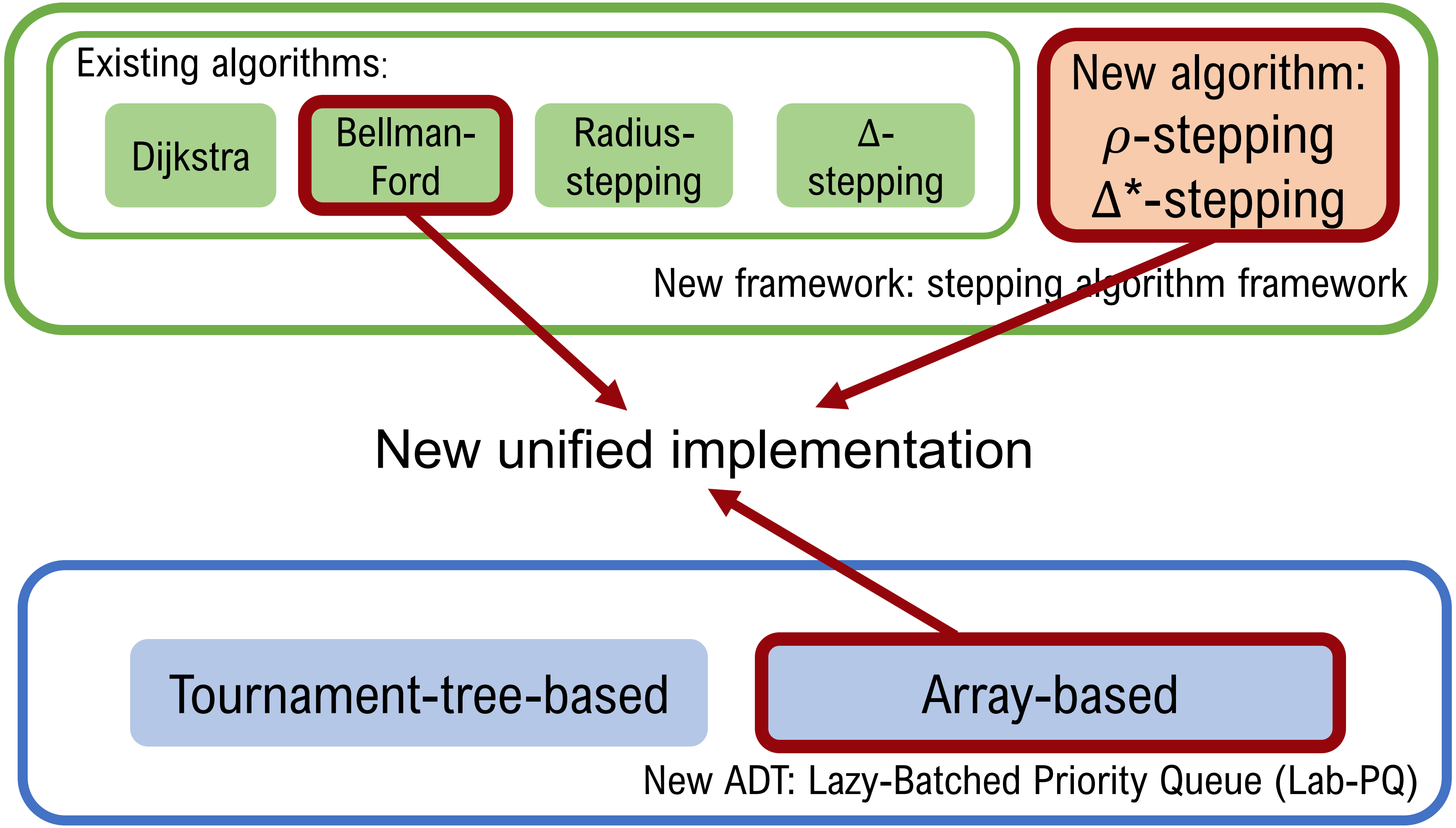
Algorithm	Work: Tournament-tree-based	Work: Array-based	Span	Previous Best: Work	Previous Best: Span
Radius-stepping† [16]	$O\left(k_\rho m \log \frac{n^2 \log \rho L}{m\rho}\right)$ (U)	$O\left(k_\rho m + \frac{k_\rho n^2}{\rho} \cdot \log \rho L\right)$ (U)	$O\left(\frac{k_\rho n}{\rho} \cdot \log \rho L \log n\right)$ (U)	$O(k_\rho m \log n)$ (U)	same
ρ-Stepping	$O\left(k_n m \log \frac{n^2}{m\rho}\right)$	$O\left(k_n m + \frac{n^2 k_\rho}{\rho}\right)$ (U) $O\left(k_n m + \frac{n^2 k_n}{\rho}\right)$	$O\left(\frac{k_\rho n \log n}{\rho}\right)$ (U) $O\left(\frac{k_n n \log n}{\rho}\right)$	-	-

Slightly worse work bound but can apply to directed case

$O(\log \rho L)$ improvement on undirected graphs
(L is the largest edge weight)

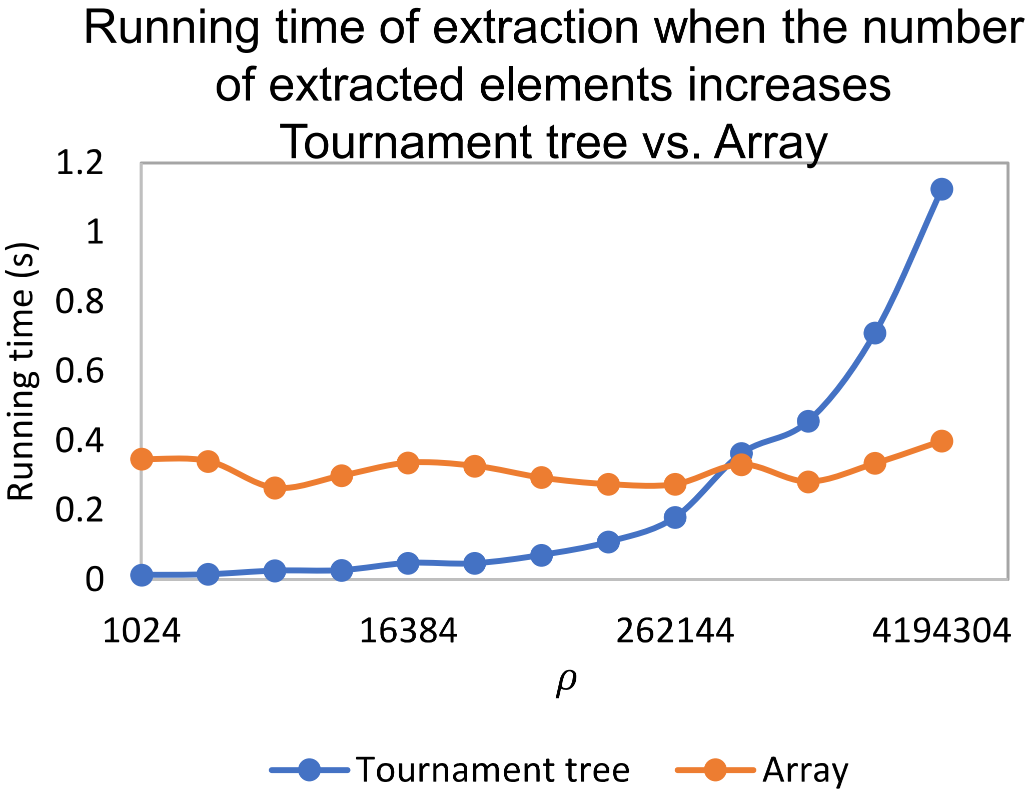
Implementation details

Recall that we implement many stepping algorithms



A practically-efficient implementation for LaB-PQ

- **JUST USE AN ARRAY**
- **Cache-friendly**
- **Easy to implement**



*We assume total #elements is 10^8, which is approximately the size of real-world graphs

Sparse/Dense optimization

- **Motivated by Ligra [SB13]**
- **Sparse: use an array to keep track of the vertices**
  - Require less space, no redundant work
- **Dense: use boolean flags to indicate if the vertices are in the frontier**
  - Better cache locality, easy to maintain

Sparse

0	2	4	5	7

Pack the vertices in frontier

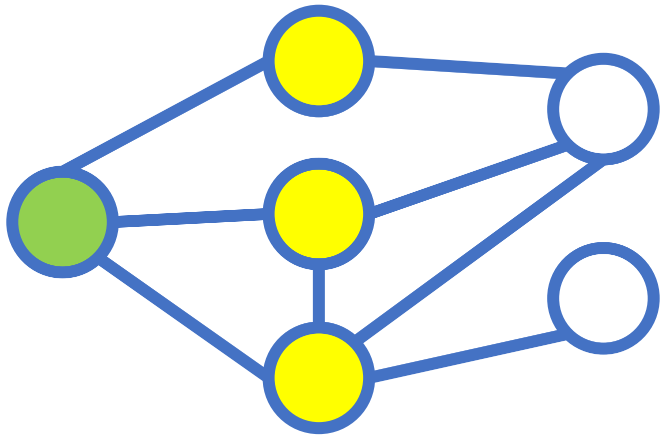
Dense

0	1	2	3	4	5	6	7
T	F	T	F	T	T	F	T

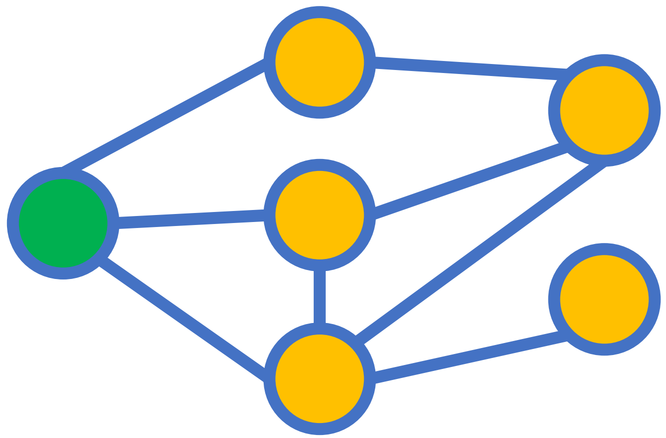
Mark the vertices in frontier as true

Bucket fusion optimization

- Motivated by GraphIt [ZBC+20]
- If the work in one round is **not sufficient**, explore **multi-hop** neighbors instead of one-hop neighbors
- **Reduce synchronization costs**
  - Critical for large-diameter graphs (e.g., road networks and grid graphs)



One-hop neighbors



Two-hop neighbors

Experiments

Set up

- **A 96-core quad-socket machine (192 hyperthreads)**
  - 1.5TB main memory and 36MB*4 L3 cache
  - C++ codes compiled with g++ 7.5.0 using CilkPlus with -O3 flag

Set up

- **7 graphs tested:**
  - 5 social and web graphs (scale-free networks): com-orkut (OK), Livejournal (LJ), Twitter (TW), Friendster (FT), and Webgraph (WB)
  - 2 road graphs: RoadUSA (USA), Germany (GE)
  - 3 to 89 million vertices, 32 million to 3.6 billion edges
  - Scale-free networks use uniformly distributed edge weight $[1, 2^{18}]$
  - Road network has edge weight provided in the dataset
- **7 implementations tested:**
  - **Δ-stepping**: GAPBS **[BAP15, ZYB+20]**, Galois **[NLP13]**, Julienne **[DBS17]**, **ours (PQ-Δ)**
  - **Bellman-Ford**: Ligra **[SB13]**, **ours (PQ-BF)**
  - **ρ-stepping**: **ours (PQ-ρ)**
- **Our code is publicly available**
  - https://github.com/ucrparlay/parallel-sssp

Heatmap: parallel running time relative to fastest on each graph

(scale-free networks)

*: ours

		Social and Web Graphs						Road Graphs		
		OK	LJ	TW	FT	WB	Ave.	GE	USA	Ave.
Δ-step.	GAPBS	1.96	1.29	2.61	1.46	1.81	1.83	1.22	1.30	1.26
	Julienne	2.18	1.75	1.96	1.36	1.92	1.83	36.74	39.61	38.18
	Galois	1.58	1.42	1.33	1.37	1.36	1.41	1.22	1.14	1.18
	*PQ-Δ	1.00	1.03	1.15	1.26	1.19	1.13	1.00	1.00	1.00
BF	Ligra	2.02	1.45	1.67	2.53	2.01	1.93	-	-	-
	*PQ-BF	1.09	1.19	1.28	1.34	1.60	1.30	1.69	1.60	1.64
ρ-step.	*PQ-ρ-fix	1.08	1.09	1.00	1.00	1.01	1.03	1.14	1.18	1.16
	*PQ-ρ-best	1.02	1.00	1.00	1.00	1.00	1.00	1.14	1.18	1.16

For all Δ-stepping we use the best Δ. PQ-ρ-best uses best ρ, and PQ-ρ-fix uses a fixed value of ρ for scale-free networks, and road graphs, respectively

Our implementations are always the fastest

(scale-free networks)

*: ours

		Social and Web Graphs						Road Graphs		
		OK	LJ	TW	FT	WB	Ave.	GE	USA	Ave.
Δ-step.	GAPBS	1.96	1.29	2.61	1.46	1.81	1.83	1.22	1.30	1.26
	Julienne	2.18	1.75	1.96	1.36	1.92	1.83	36.74	39.61	38.18
	Galois	1.58	1.42	1.33	1.37	1.36	1.41	1.22	1.14	1.18
	*PQ-Δ	1.00	1.03	1.15	1.26	1.19	1.13	1.00	1.00	1.00
BF	Ligra	2.02	1.45	1.67	2.53	2.01	1.93	-	-	-
	*PQ-BF	1.09	1.19	1.28	1.34	1.60	1.30	1.69	1.60	1.64
ρ-step.	*PQ-ρ-fix	1.08	1.09	1.00	1.00	1.01	1.03	1.14	1.18	1.16
	*PQ-ρ-best	1.02	1.00	1.00	1.00	1.00	1.00	1.14	1.18	1.16

For all Δ-stepping we use the best Δ. PQ-ρ-best uses best ρ, and PQ-ρ-fix uses a fixed value of ρ for scale-free networks, and road graphs, respectively

Scale-free networks: ρ-stepping is faster than all existing code by at least 40%

*: ours

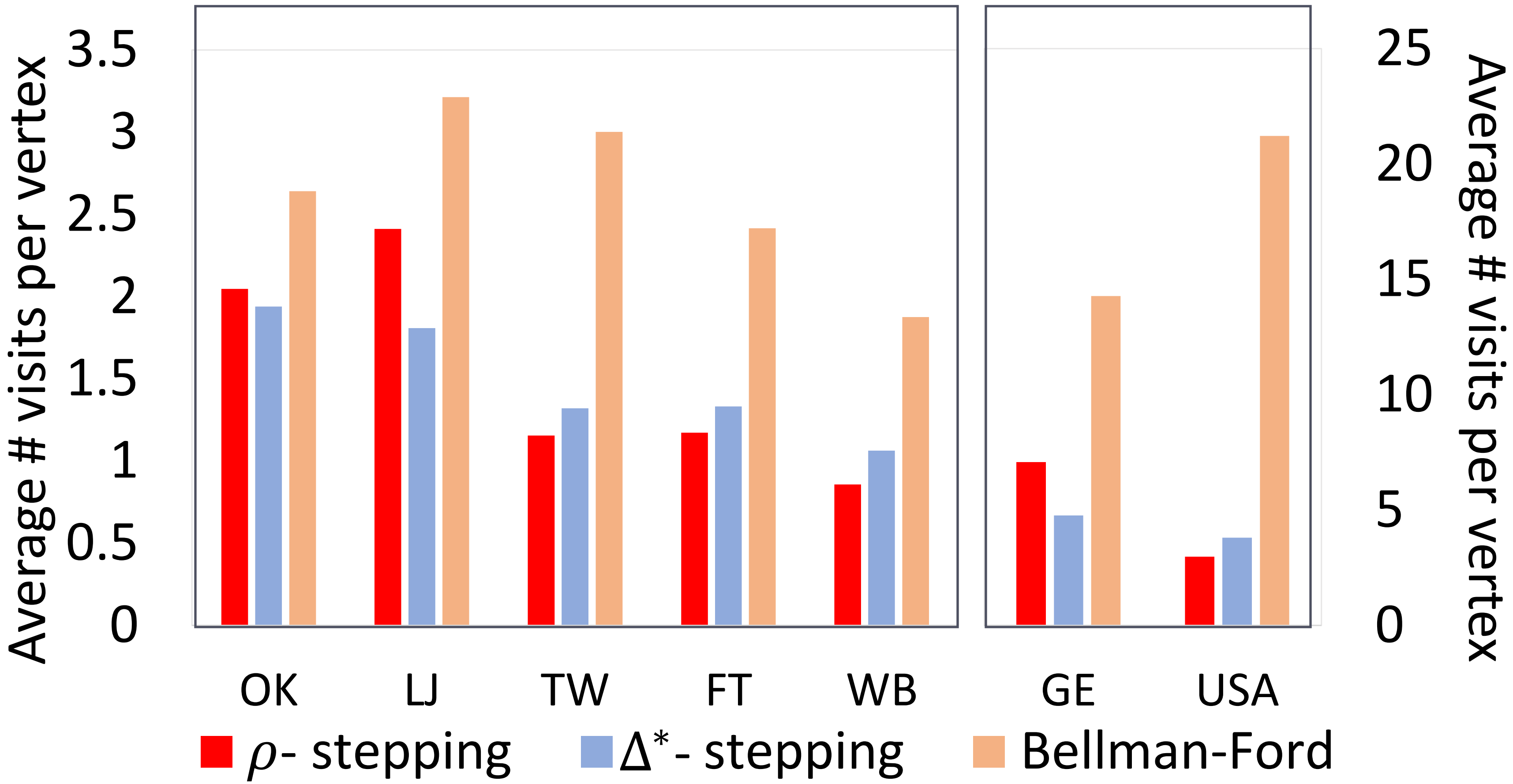
		Social and Web Graphs						Road Graphs		
		OK	LJ	TW	FT	WB	Ave.	GE	USA	Ave.
Δ-step.	GAPBS	1.96	1.29	2.61	1.46	1.81	1.83	1.22	1.30	1.26
	Julienne	2.18	1.75	1.96	1.36	1.92	1.83	36.74	39.61	38.18
	Galois	1.58	1.42	1.33	1.37	1.36	1.41	1.22	1.14	1.18
	*PQ-Δ	1.00	1.03	1.15	1.26	1.19	1.13	1.00	1.00	1.00
BF	Ligra	2.02	1.45	1.67	2.53	2.01	1.93	-	-	-
	*PQ-BF	1.09	1.19	1.28	1.34	1.60	1.30	1.69	1.60	1.64
ρ-step.	*PQ-ρ-fix	1.08	1.09	1.00	1.00	1.01	1.03	1.14	1.18	1.16
	*PQ-ρ-best	1.02	1.00	1.00	1.00	1.00	1.00	1.14	1.18	1.16

Our Δ-stepping is fastest on road graphs, and our ρ-stepping is competitive

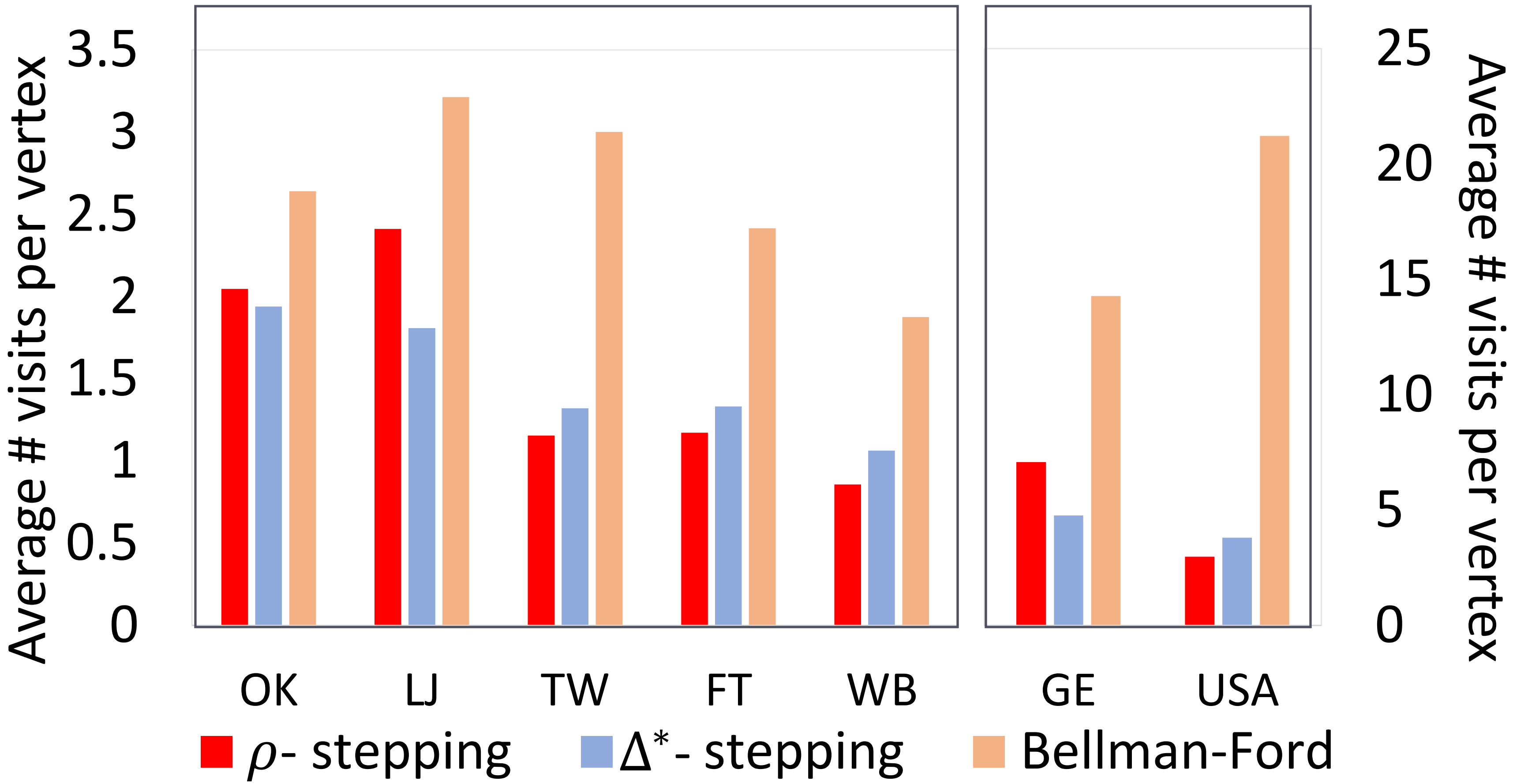
*: ours

		Social and Web Graphs						Road Graphs		
		OK	LJ	TW	FT	WB	Ave.	GE	USA	Ave.
Δ-step.	GAPBS	1.96	1.29	2.61	1.46	1.81	1.83	1.22	1.30	1.26
	Julienne	2.18	1.75	1.96	1.36	1.92	1.83	36.74	39.61	38.18
	Galois	1.58	1.42	1.33	1.37	1.36	1.41	1.22	1.14	1.18
	*PQ-Δ	1.00	1.03	1.15	1.26	1.19	1.13	1.00	1.00	1.00
BF	Ligra	2.02	1.45	1.67	2.53	2.01	1.93	-	-	-
	*PQ-BF	1.09	1.19	1.28	1.34	1.60	1.30	1.69	1.60	1.64
ρ-step.	*PQ-ρ-fix	1.08	1.09	1.00	1.00	1.01	1.03	1.14	1.18	1.16
	*PQ-ρ-best	1.02	1.00	1.00	1.00	1.00	1.00	1.14	1.18	1.16

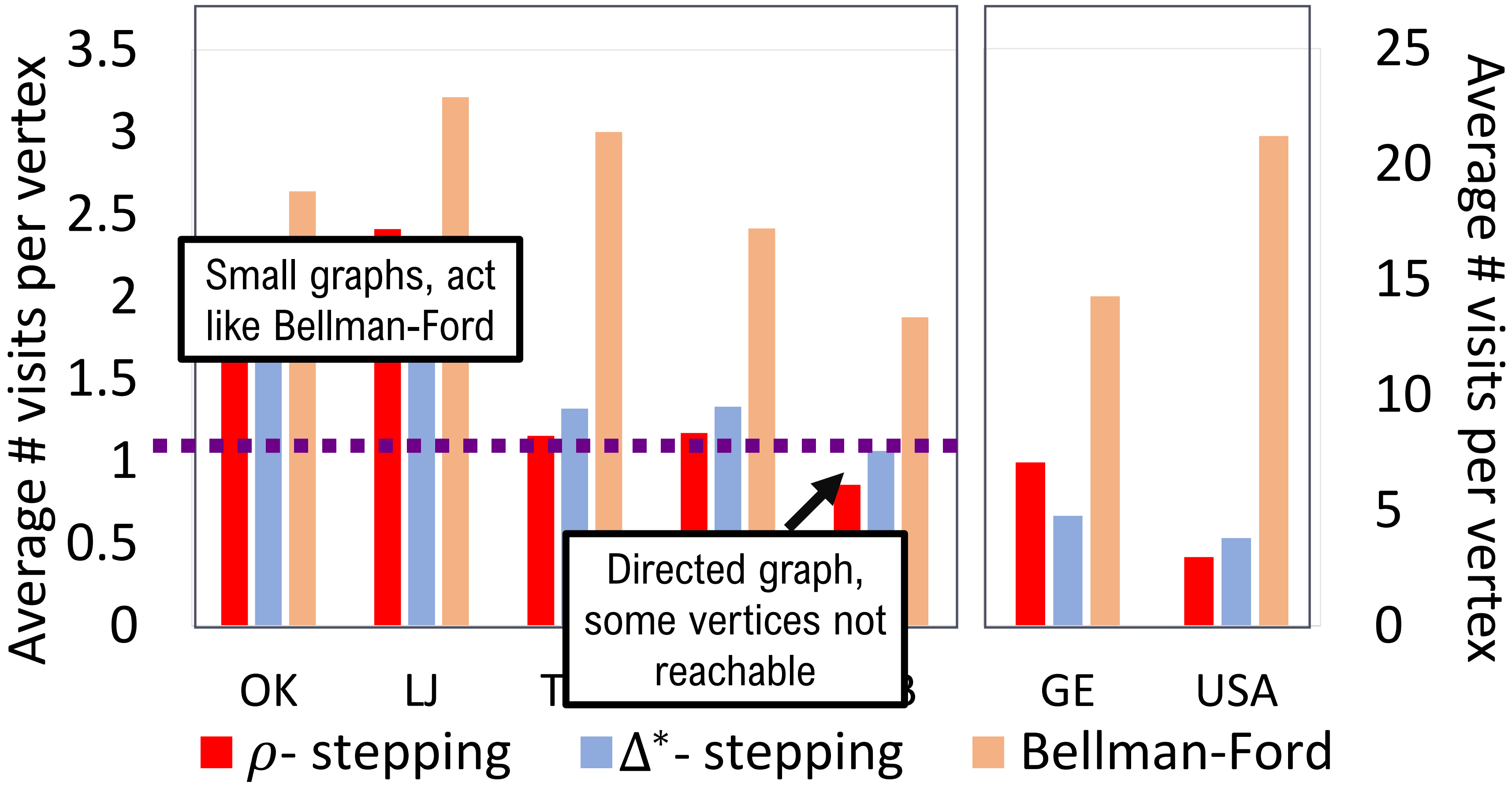
Number of visit (enqueue) per vertex



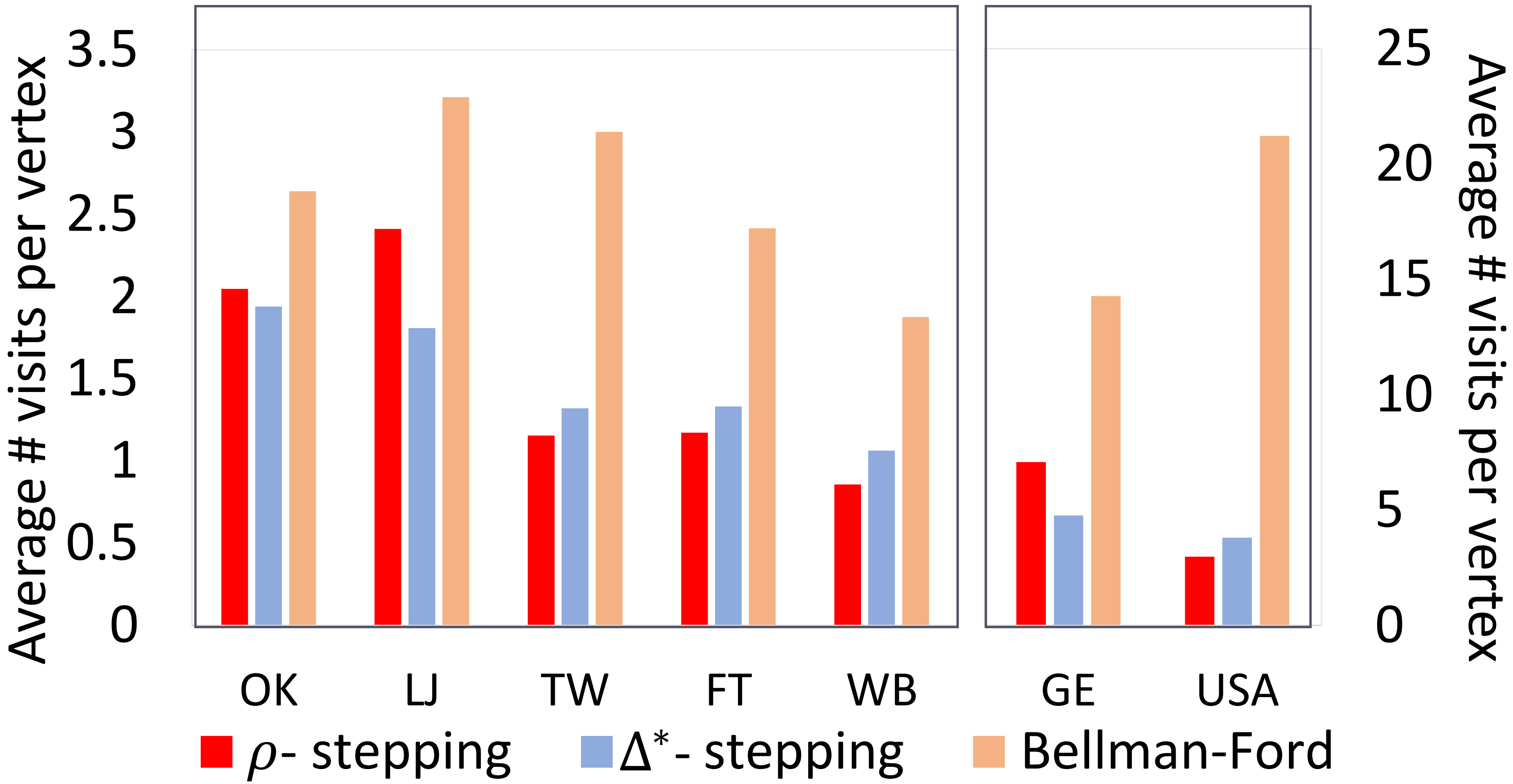
With careful coding, Bellman-Ford is already close to optimal on scale-free networks (2.5 #enqueue per vertex)



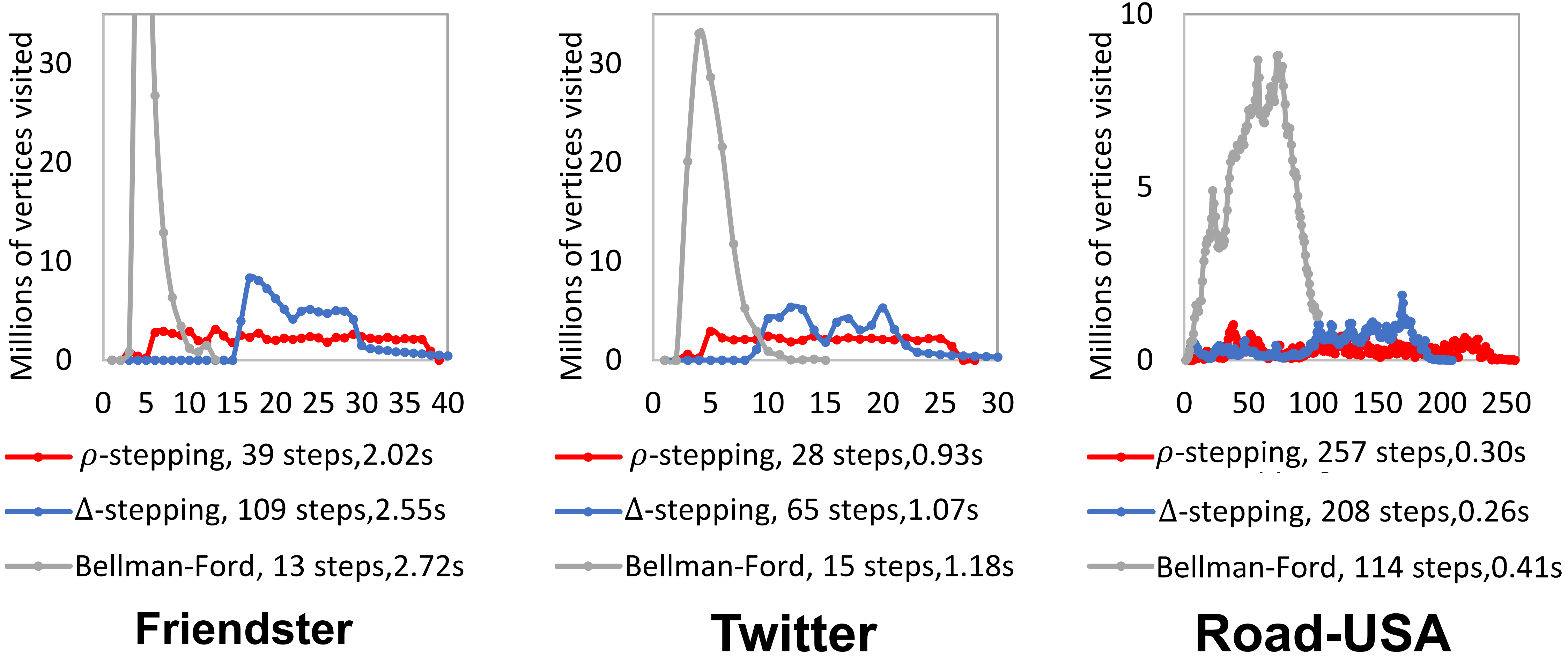
ρ-stepping is almost optimal (very close to 1)



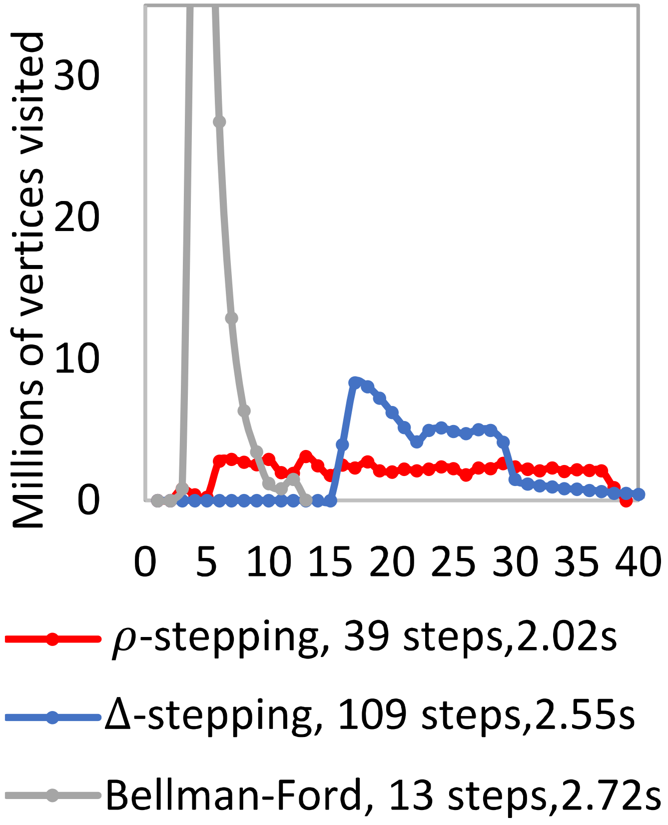
Room for improvement for road graphs
(but these graphs are relatively small)



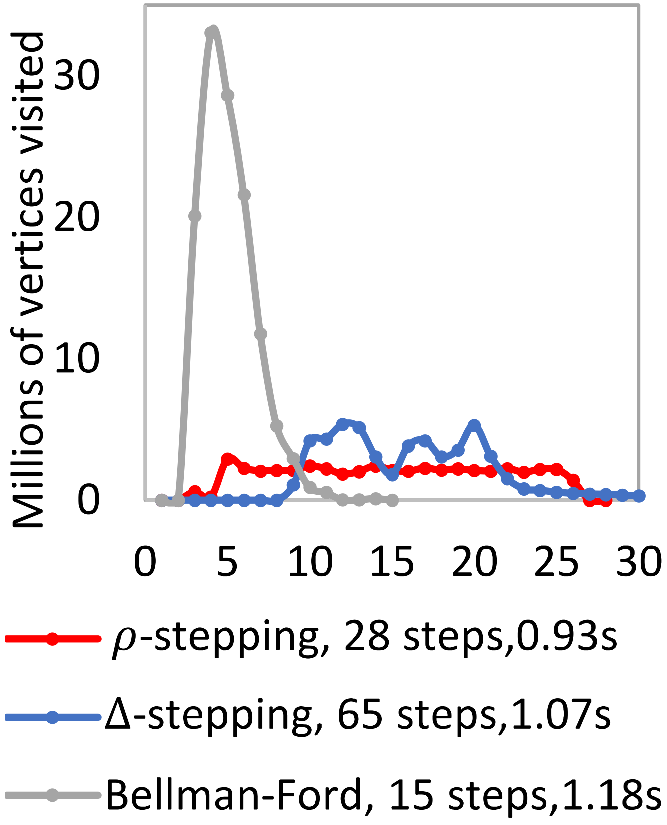
Vertices visited per step



ρ-stepping: sufficient but minimal work to saturate all processors, independent with graph properties



Friendster



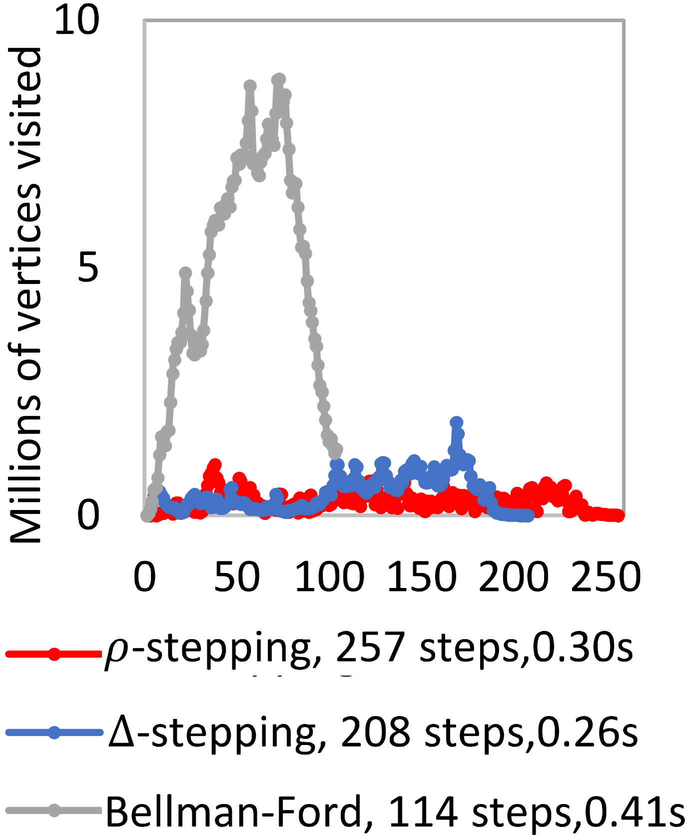
Twitter

Road-USA

ρ-stepping: can be too eager for large diameter graphs

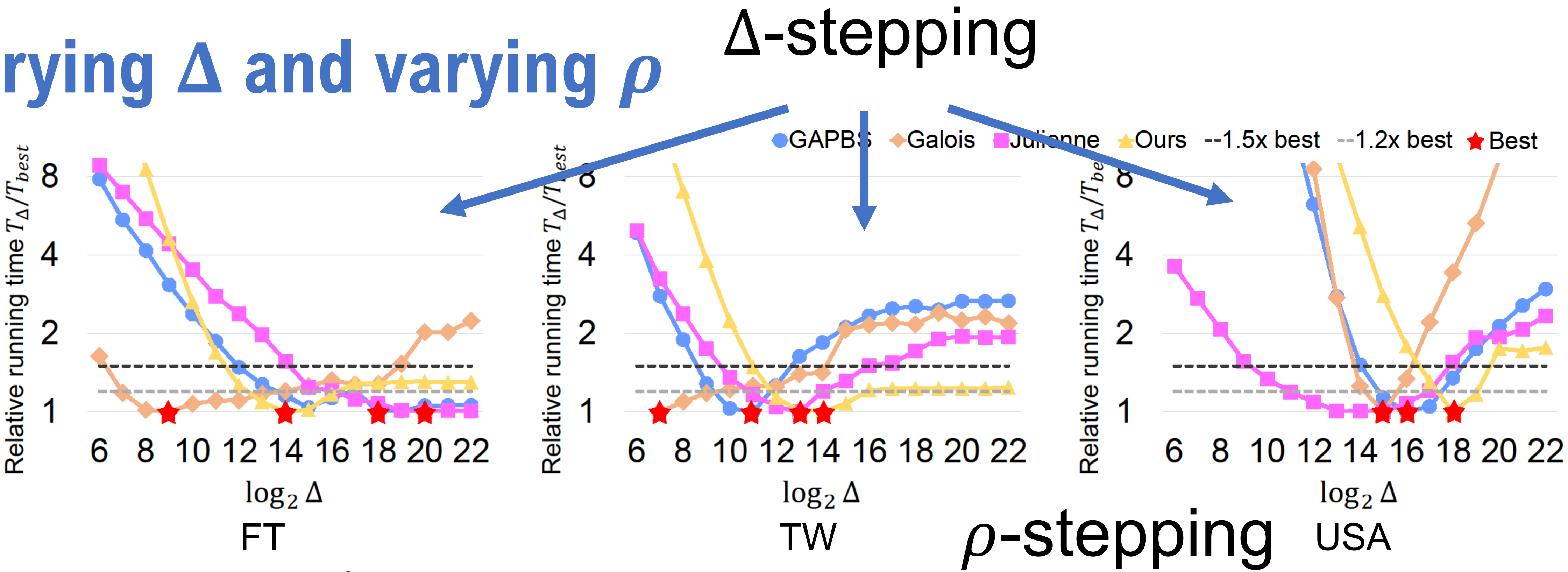
Friendster

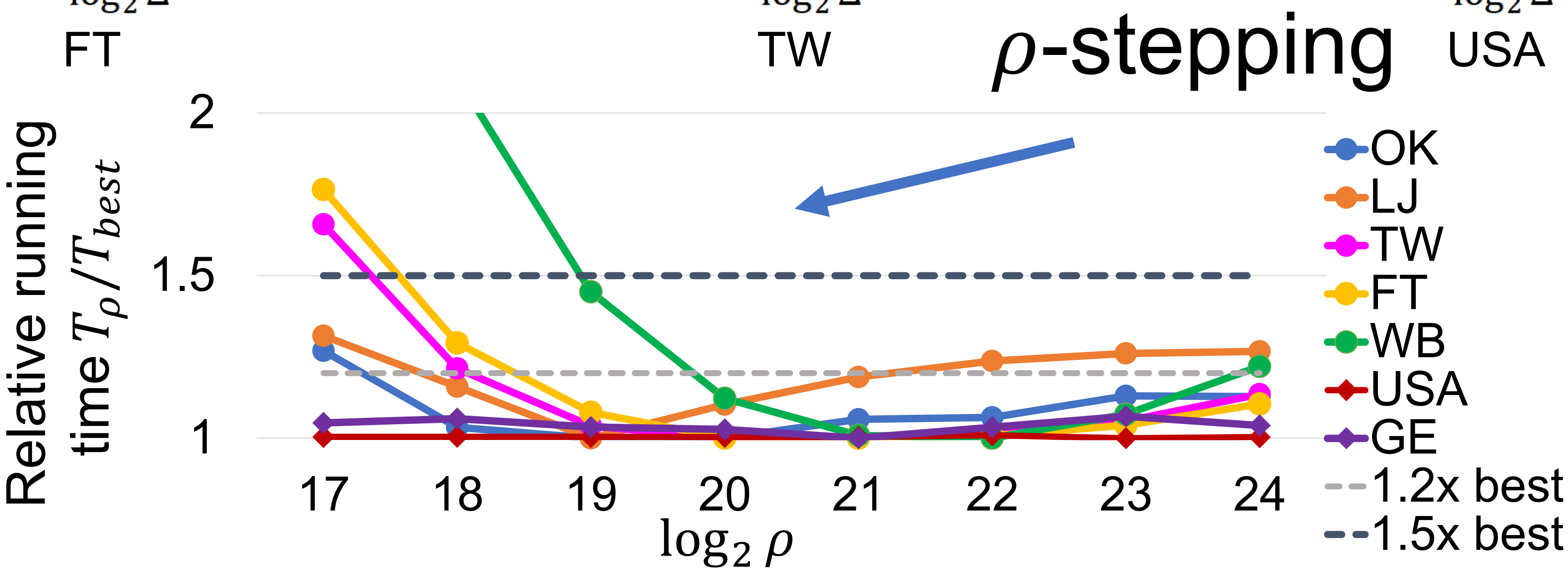
Twitter



Road-USA

Varying Δ and varying ρ





Other experiments

- **More experimental results**
  - difference source vertices
  - different machine
  - Average #visits per edge
  - given in the full version of this paper

Summary

Our approach

Existing algorithms:

Dijkstra

Bellman-Ford

Radius-stepping

Δ-stepping

New algorithm:
ρ-stepping
Δ*-stepping

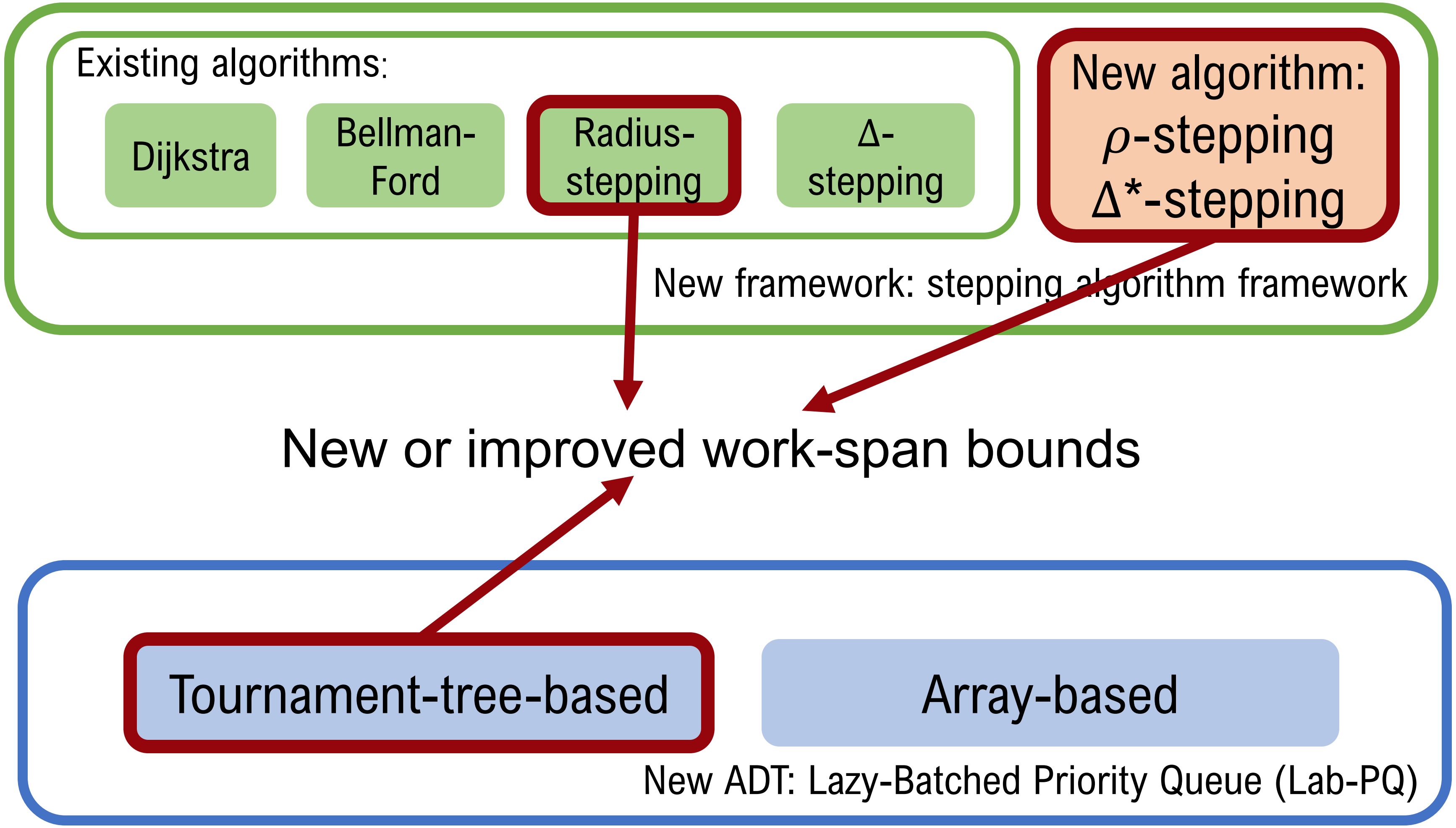
New framework: stepping algorithm framework

Tournament-tree-based

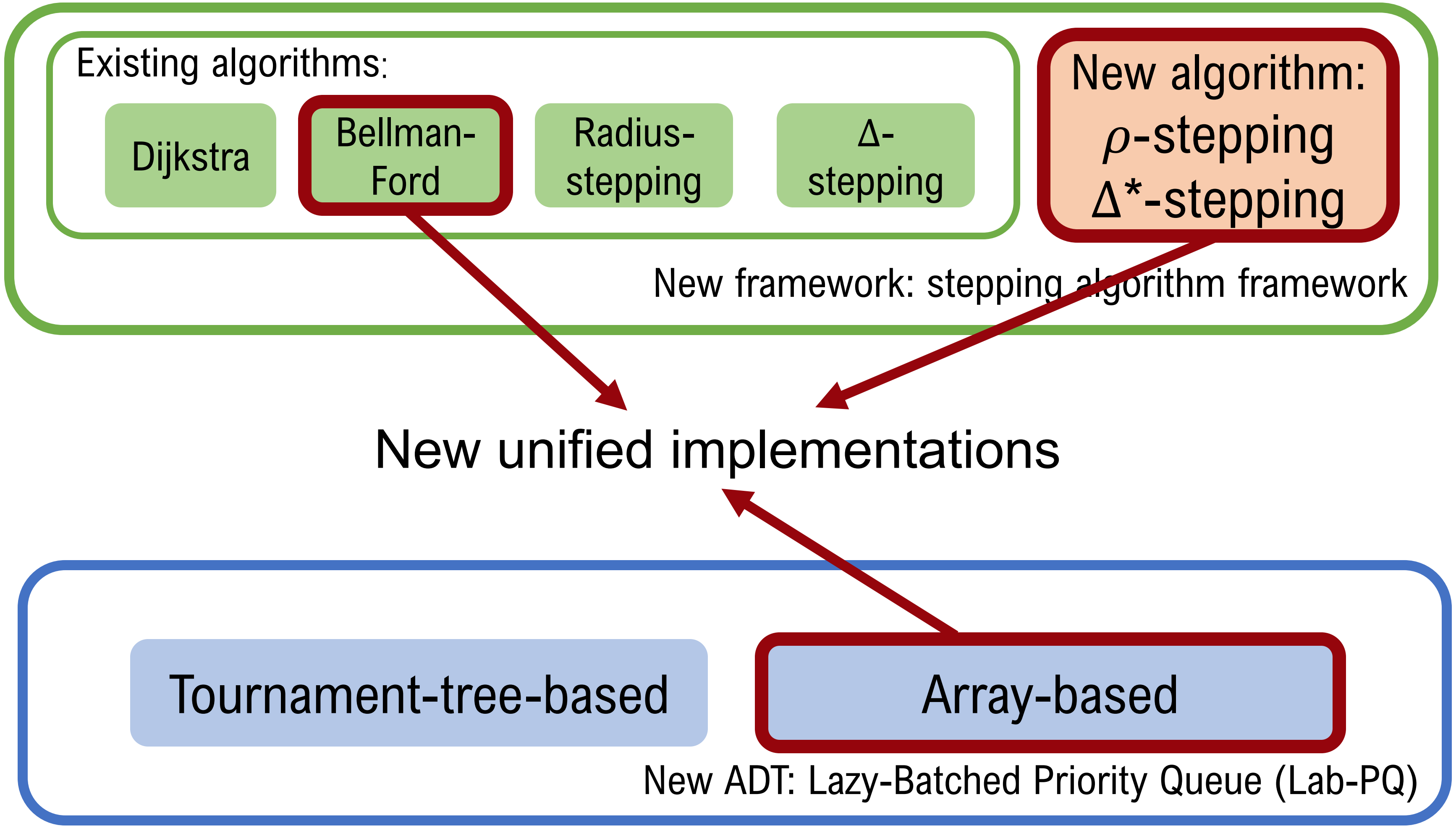
Array-based

New ADT: Lazy-Batched Priority Queue (Lab-PQ)

Our results: theoretical analysis



Our results: efficient implementations

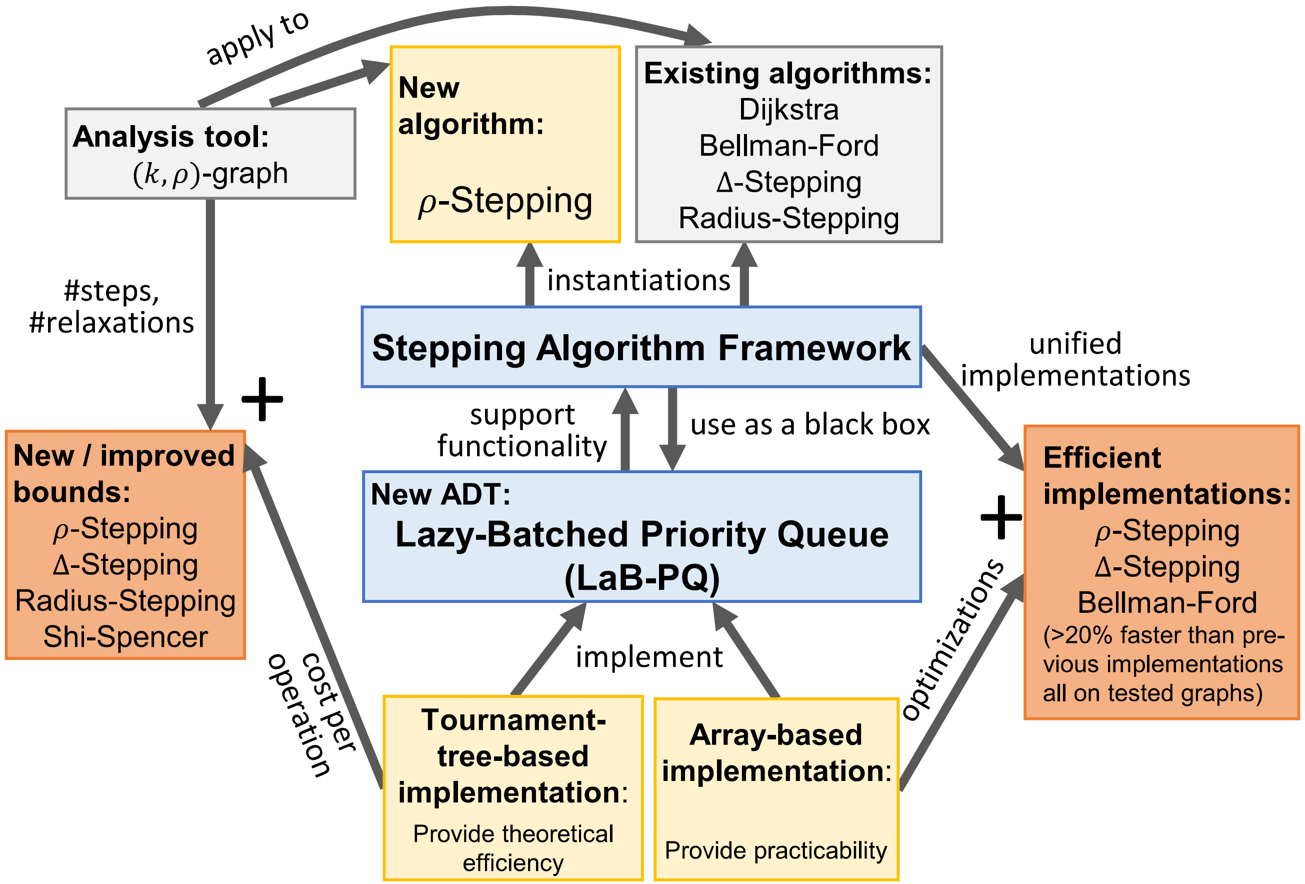


New algorithm: ρ-stepping and Δ^*-stepping

- **Extremely simple on top of the LaB-PQ**
  - Just use an array
- **Good theoretical guarantee: similar to Radius-Stepping**
- **Avoid sub-steps in Δ-stepping and Radius-Stepping**

- **ρ-stepping**
  - Insensitive to the value of $\boldsymbol{\rho}$
  - Especially good on scale-free networks

- **Δ^*-stepping**
  - Simply remove the FinishCheck in $\boldsymbol{\Delta}$-stepping
  - Especially good on road networks



- **Full version: https://arxiv.org/abs/2105.06145**
- **Code: https://github.com/ucrparlay/Parallel-SSSP**
- **Contact: Xiaojun Dong (xdong038@ucr.edu)**