KSPine: Fast Parallel K-Shortest Path Computation via Block-Cut+ Tree Decomposition and Dynamic Caching

SC26 · Chicago, IL · November 2026

Weixi Chen[1] **Runbang Hu**[2] **Yuede Ji**[2]

[1] University of Southern California [2] The University of Texas at Arlington



K-shortest path (KSP) queries are a core graph operator

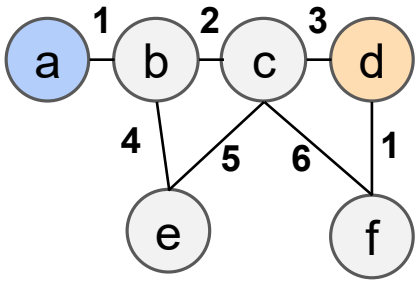
Example: the 3 shortest paths between two people in Neo4j

```
MATCH p = SHORTEST 3 (a {name:'Alice'})-[:KNOWS]-+(b {name:'Bob'})
RETURN p
```

Graph databases	Neo4j, ArangoDB, Memgraph
Navigation	alternative routes in OSRM, GraphHopper, Valhalla
Social networks	explaining how two people are connected

UTA

KSP returns the K shortest loopless paths from s to t



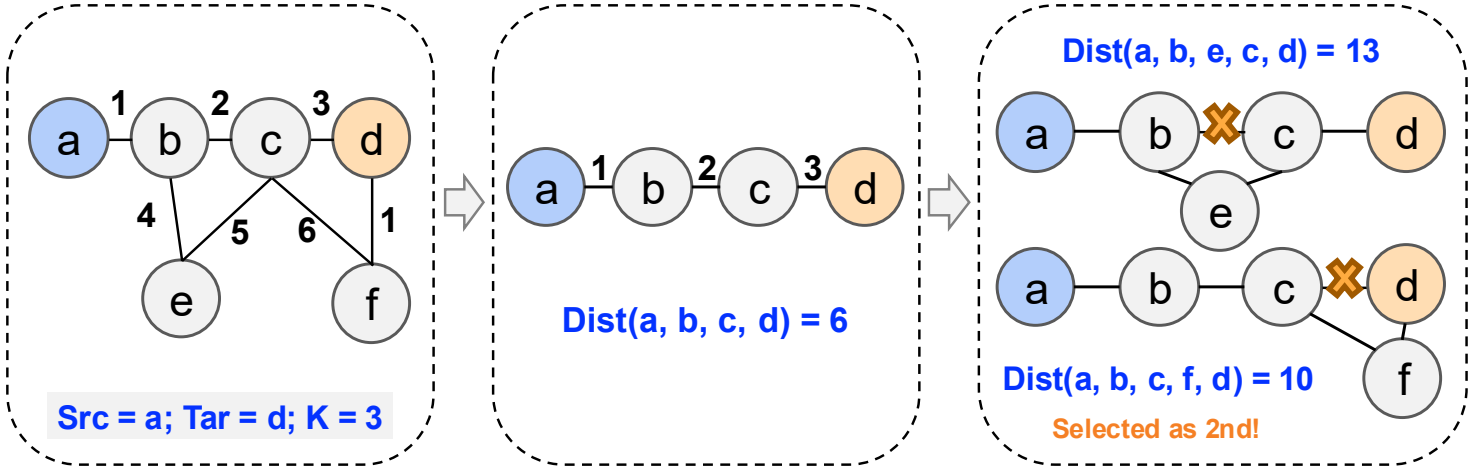
Src = a; Tar = d; K = 3

Input graph G, source s, target t, integer K

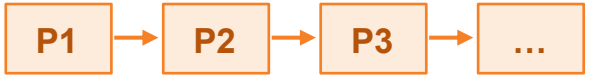
Output K distinct simple s–t paths, shortest first

K = 1 is plain shortest path

Yen's method builds each new path from an earlier one



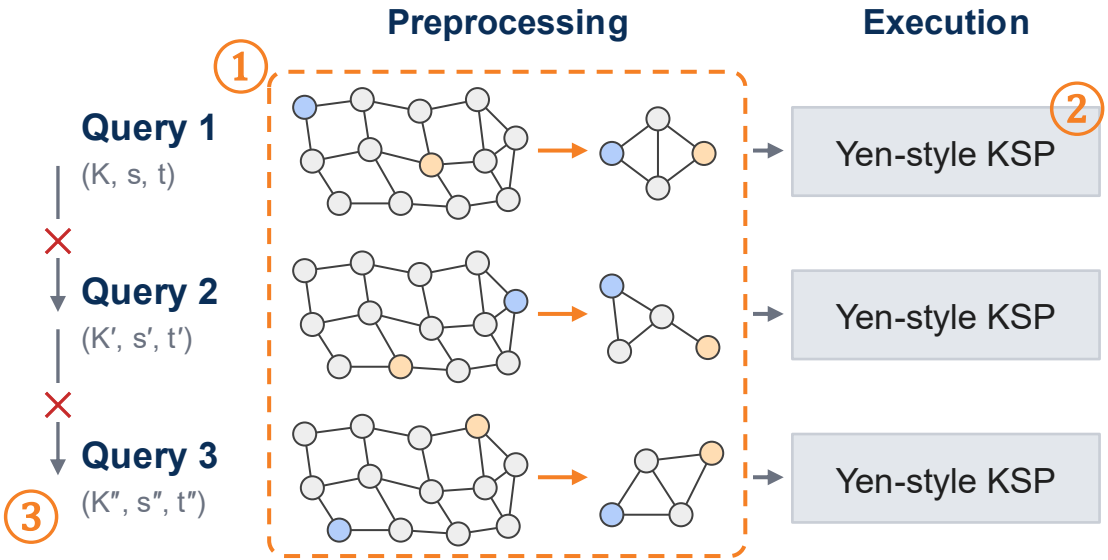
(a) An example graph (b) 1st shortest path (c) Deviation paths



Each path is derived from the previous one: sequential

$O(K \cdot n \cdot (m + n \log n))$ · basis of OptYen, Feng, KIM, and PeeK

Observation: PeeK [SC'23] still faces three challenges

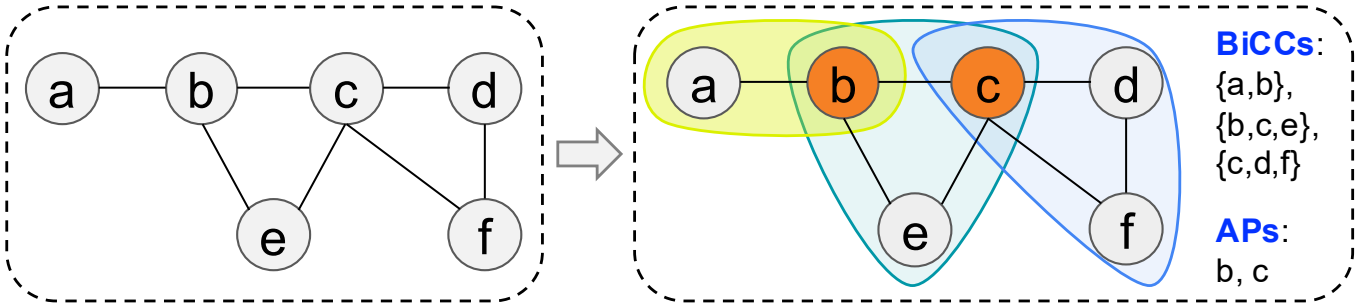


1. **Query-dependent pruning**
   Pruning depends on (K, s, t), so it is redone for every query

2. **Limited parallelism**
   Deviation-path generation takes **59%** of PeeK's runtime

3. **Neglected historical computation**
   Queries share subpaths, yet each starts from scratch

All s–t paths pass through the APs that separate s and t



Articulation point (AP): removing it disconnects the graph

Biconnected component (BiCC): a maximal subgraph that stays connected when any one of its vertices is removed

Every a→d path = a→b in {a,b} + b→c in {b,c,e} + c→d in {c,d,f}

KSP(a, d) splits into one small KSP per block, joined at the APs b and c.

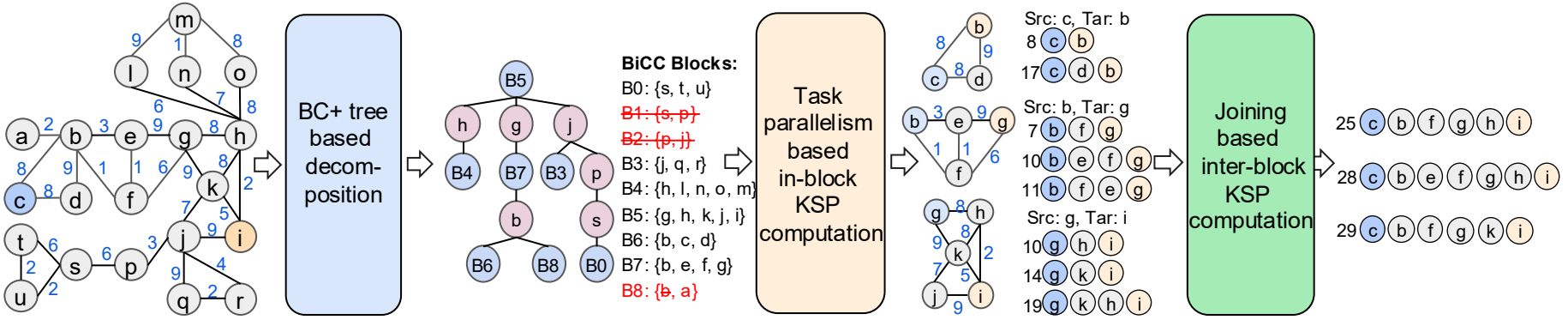
Real-world graphs split into 0.2M–7.4M biconnected blocks

Graph	|V|	# APs	# BiCCs	Largest BiCC
Wiki-Talk (WT)	2.4M	34,493	**1,769,165**	**26.0% of V**
Com-Youtube (CY)	1.1M	223,409	**673,661**	39.8% of V
As-Skitter (AS)	1.7M	111,541	**238,721**	**85.1% of V**
Socfb-A-Anon (SA)	3.1M	186,402	**1,658,933**	46.4% of V
Dblp-Author (DA)	5.6M	1,895,542	**2,209,735**	60.7% of V
Gap-Road (GR)	23.9M	6,229,607	**7,390,330**	66.9% of V

① **Many blocks:** 0.2M–7.4M BiCCs per graph

② **Shapes differ:** largest block = 26%–85% of V

KSPine splits each query into independent per-block KSPs



(a) An example graph (b) Block-Cut+ Tree (c) Sub-KSP in blocks (d) Final KSP results

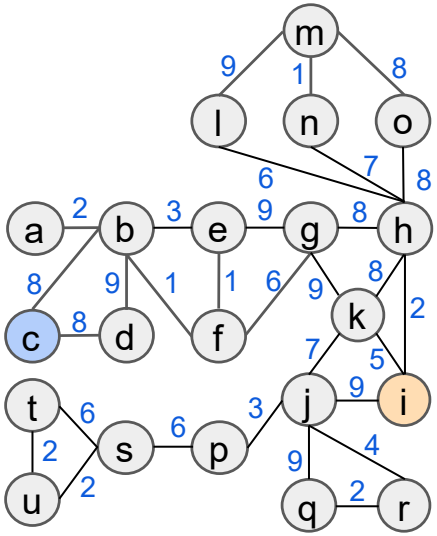
① **Build the BC+ tree**
once per graph

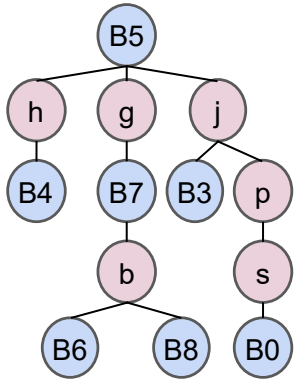
② **Find the blocks**
on the s–t tree path

③ **Solve local KSPs**
one block per thread

④ **Join local results**
into the global top-K

Block-Cut+ (BC+) Tree: drop blocks that add no alternative paths





Block-Cut+ Tree

BiCC Blocks:

B0: {s, t, u}

~~B1: {s, p}~~

~~B2: {p, j}~~

B3: {j, q, r}

B4: {h, l, n, o, m}

B5: {g, h, k, j, i}

B6: {b, c, d}

B7: {b, e, f, g}

B8: {~~b~~, a}

Bridge block (2 APs)

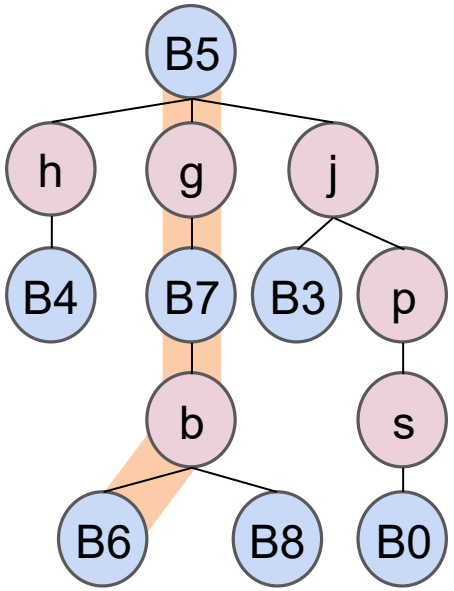
B1 {s, p}, B2 {p, j}: removed, APs linked directly

Dangling block (AP + 1 vertex)

B8 {b, a}: reduced to vertex a

Built once in O(n + m), unique (Lemma 1)

A query only touches the blocks on its BC+ tree path



Query: s = c (in B6), t = i (in B5)

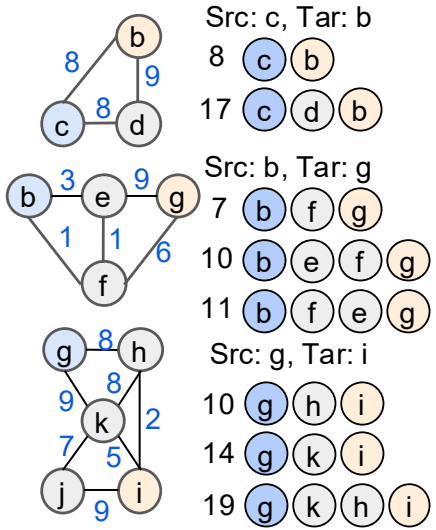
Block on path	Sub-source → sub-target
B6	c → b
B7	b → g
B5	g → i

Lemma 2: all top-K paths stay in blocks on this path

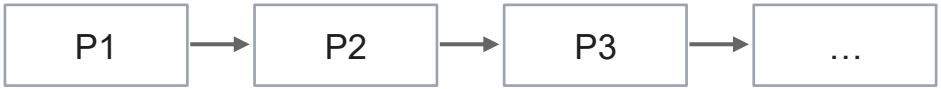
Leaving through an AP means coming back through the same AP, which forms a loop

B4, B3, B0, and B8 are never touched

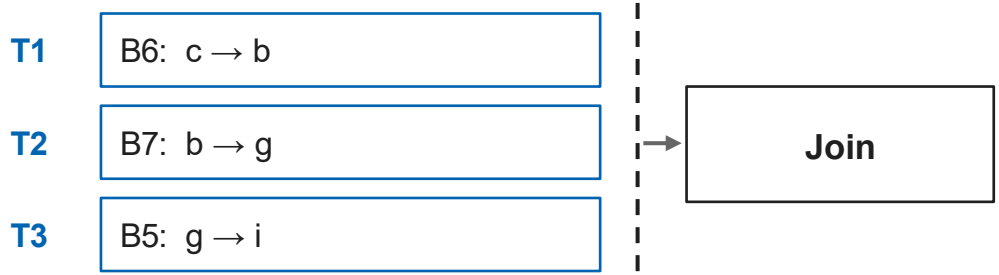
Blocks on the path are solved in parallel, without locks



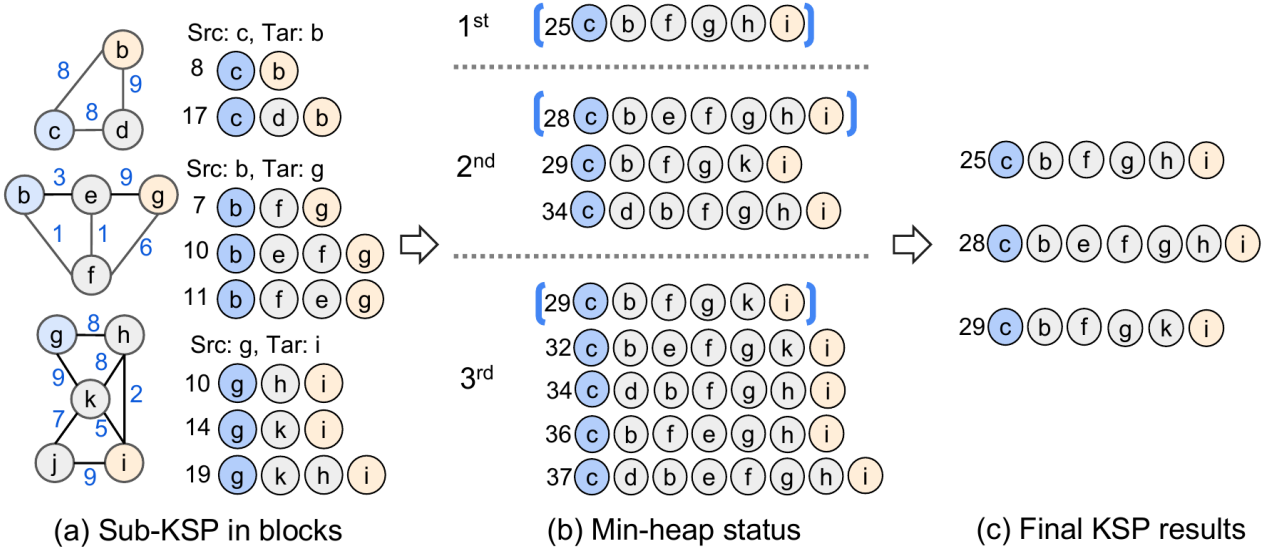
Yen-style methods: one long dependency chain



KSPine: one thread per block



A min-heap joins local lists without enumerating K^n cases



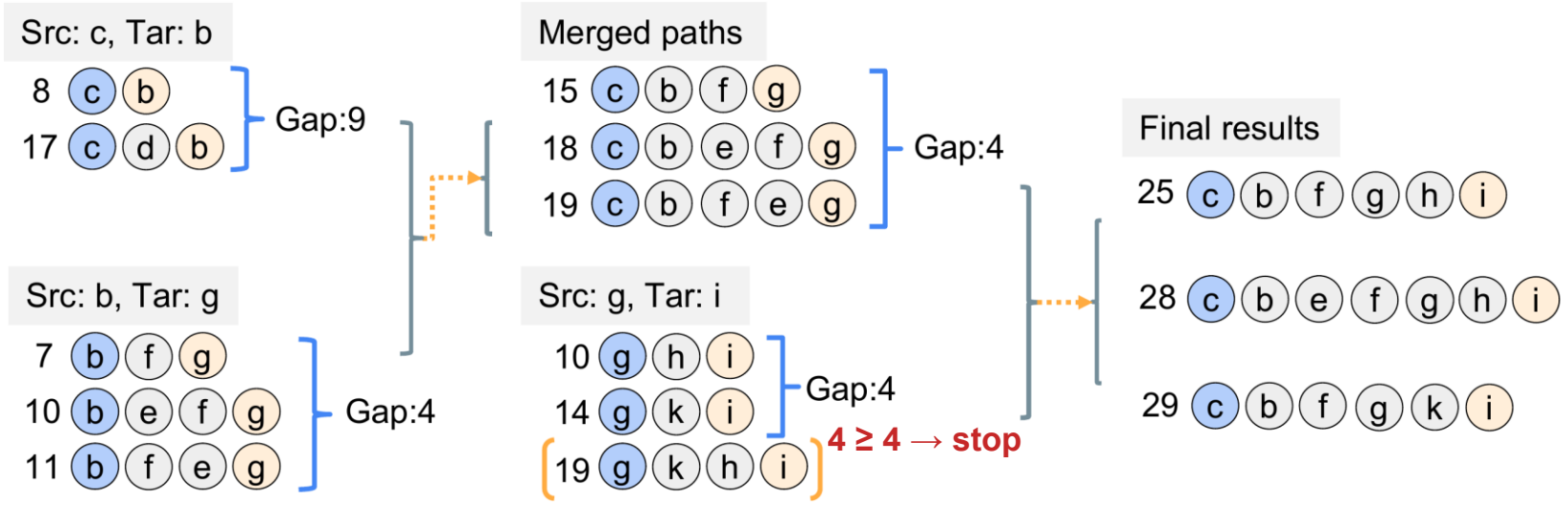
(a) Sub-KSP in blocks (b) Min-heap status (c) Final KSP results

Every global path picks one entry per block. **Lists are sorted**, so start from (0, 0, 0) and expand only the best tuple.

Lemma 4: the i-th pop is the i-th shortest path.

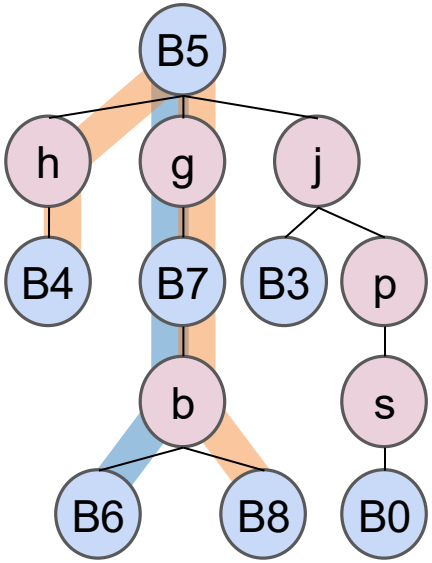
Limitation: each block must first compute its full local top-K, even paths never used (e.g., 19 in B5).

Merge joining stops a block once it cannot change the top-K



Stop when |M| = K and sub-gap ≥ gap (Lemma 5)

Queries that cross the same block reuse each other's work



Q1: c → i Q2: a → n

Block	Q1	Q2	Q2 reuses
B7	b → g	b → g	**local top-K paths**
B5	g → i	g → h	**extracted subgraph**
B4	–	h → n	computed fresh

1 **Subpath cache** key: (block, AP pair)
2 **Subgraph cache** key: block
3 **Otherwise build, solve, and insert**

Only AP-pair entries are cached: they are where queries meet

We use six real-world graphs with up to 24M vertices

Graph	Type	|V|	|E|
Com-Youtube (CY)	Community	1.1M	6.0M
As-Skitter (AS)	Autonomous system	1.7M	22.2M
Wiki-Talk (WT)	Communication	2.4M	9.3M
Socfb-A-Anon (SA)	Social	3.1M	47.3M
Dblp-Author (DA)	Authorship	5.6M	24.6M
Gap-Road (GR)	Road	23.9M	57.7M

Platform
2× AMD EPYC 9334, up to 32 threads, C++/OpenMP

Baselines
Yen, OptYen, Feng, KIM, PeeK

Queries
100 random s–t pairs (10 on GR); K = 32 unless noted

We ask: **Q1 How fast?** **Q2 Where does the speedup come from?**
Q3 How does it scale with K? **Q4 When does it work best?**

UTA

KSPine is fastest on all six graphs: 1.5× over PeeK

Q1 · How fast is KSPine?

Runtime (s), K = 32	WT	CY	SA	AS	DA	GR	KSPine is
Yen [Mgmt. Sci.'71]	449	286	1,025	535	1,659	–	4.1× faster
KIM [Networks'82]	6,026	3,478	–	8,518	–	–	43× faster
Feng [Networks'14]	715	670	3,486	2,382	6,198	–	13× faster
OptYen [ICPP'18]	284	172	996	450	1,153	–	3.2× faster
PeeK [SC'23]	**253**	**172**	**262**	**198**	**442**	**2,612**	**1.5× faster**
KSPine	**108**	**120**	**191**	**192**	**298**	**2,580**	

– : not reported · across K = 8–128, 1.4× over the best baseline

Decomposition drives the speedup

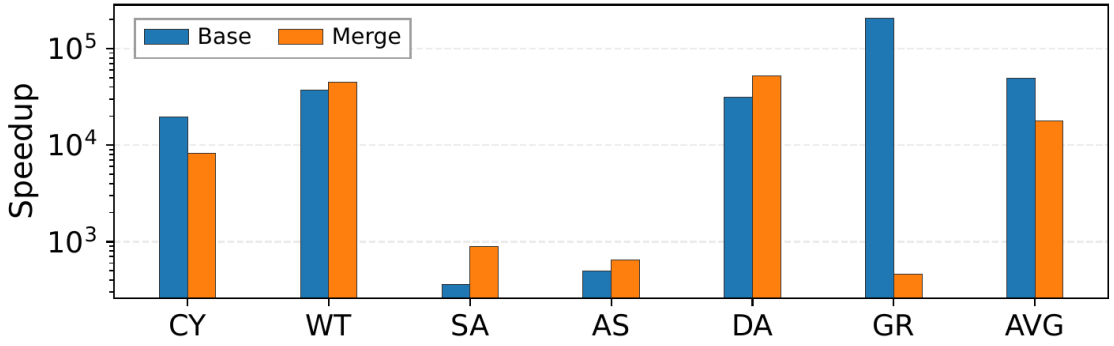
Q2 · Where does the speedup come from?

Runtime (s), K = 32	WT	CY	SA	AS	DA	GR
PeeK (state of the art)	253	172	262	198	442	2,612
KSPine_Base (heap join)	111.3	122.2	197.4	198.6	298.7	**2,553.2**
KSPine_Merge (merge join)	**106.0**	**119.6**	**188.4**	**191.3**	**289.2**	2,555.5
One-time BC+ tree preprocessing (s)						
	1.9	0.5	2.2	0.3	9.0	24.7

Merge vs. Base: 2.1 – 4.8% faster on five graphs

Subpath caching speeds up hot-path queries

Q2 · Where does the speedup come from?



Speedup from turning the subpath cache on (log scale)

Cost: peak memory

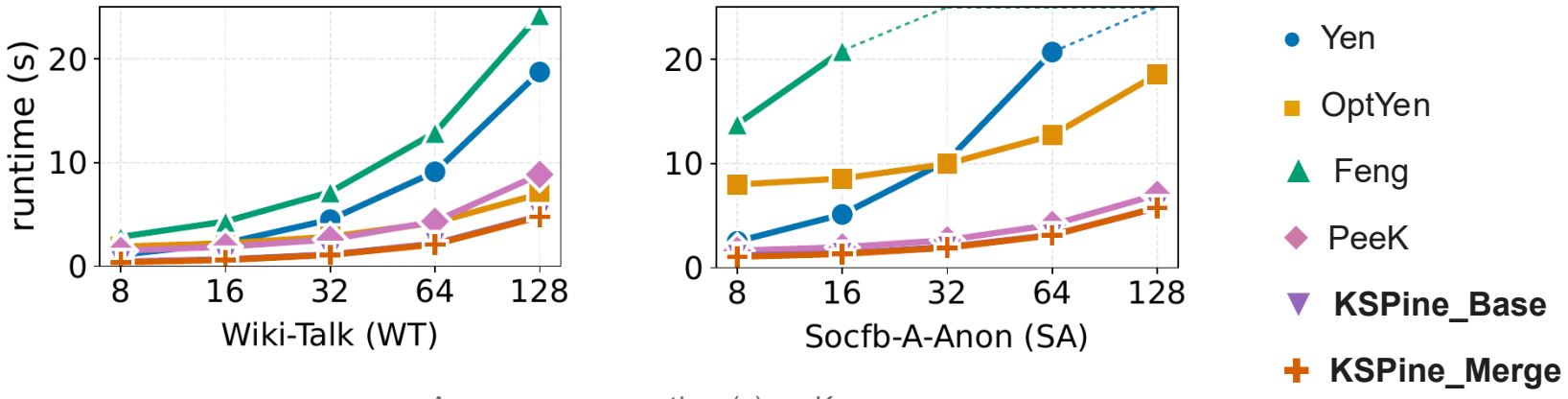
Graph	Memory (GB)	+
WT	2.6 → 3.2	21%
CY	1.4 → 1.8	27%
SA	9.7 → 12.6	31%
AS	12.6 → 13.5	7%
DA	17.8 → 19.7	11%
GR	38.2 → 43.0	13%

Hot-path workload: 10 queries that share BC+ tree segments

Cache on vs. cache off

KSPine's runtime grows near-linearly with K

Q3 · How does KSPine scale with K?

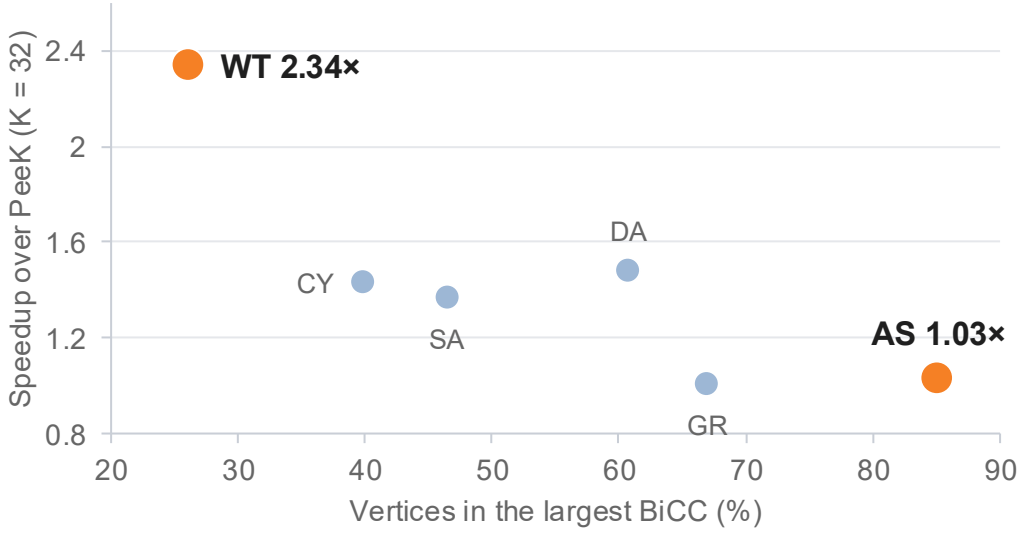


Average per-query time (s) vs. K

KSPine_Merge is lowest in every case except K = 128 on CY

Gains are largest when no single block dominates the graph

Q4 · When does KSPine work best?



r = −0.84

largest-block share vs. speedup

Wiki-Talk 26% in one block → 2.34×

As-Skitter 85% in one block → 1.03×

BiCCs associated per AP: r = +0.84; AP ratio alone: r = −0.35

KSPine: let graph structure guide KSP

Query-dependent pruning	→	**BC+ tree: one skeleton, built once, prunes every query**
Limited parallelism	→	**Independent sub-KSP per block**
Neglected historical computation	→	**Hierarchical caching of AP-pair paths and block subgraphs**

1.5× faster than PeeK on average; fastest on all six graphs

github.com/SC-Lab-Go/KSPine

Thank you! Questions?

UTA