PeeK: A Prune-Centric Approach for *K* Shortest Path Computation

Wang Feng, Shiyang Chen, Hang Liu, Yuede Ji

UNT UNIVERSITY OF NORTH TEXAS

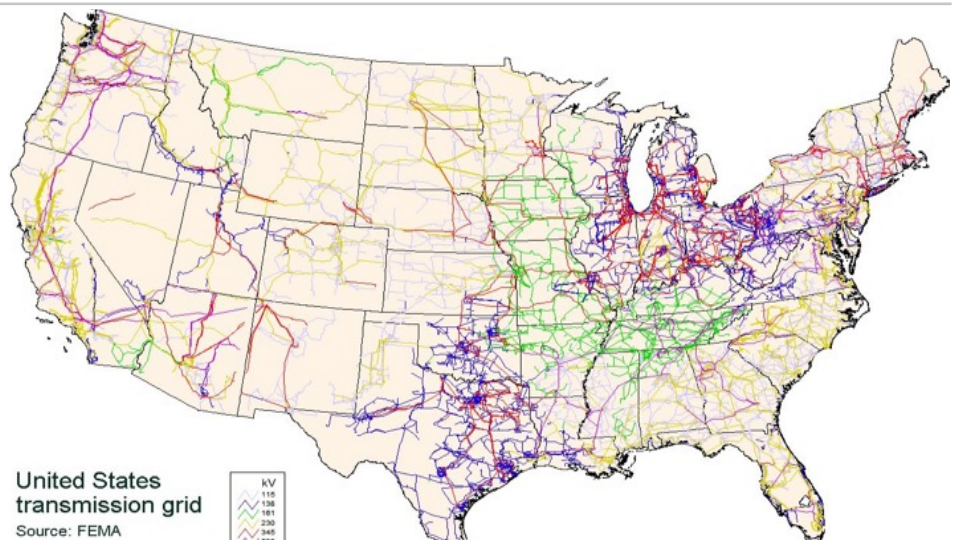
RUTGERS

RUTGERS

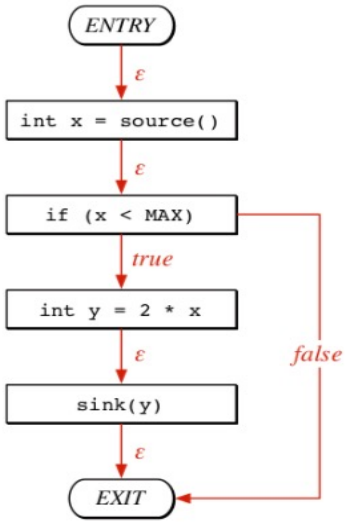
UNT UNIVERSITY OF NORTH TEXAS

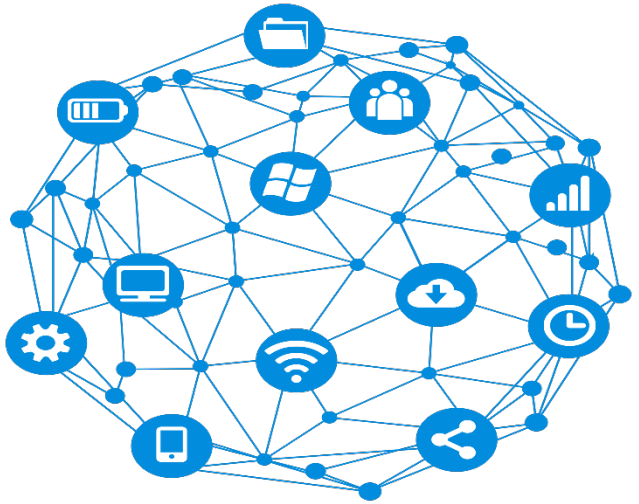
SC23 DENVER NOV 12-17

Graph is Everywhere



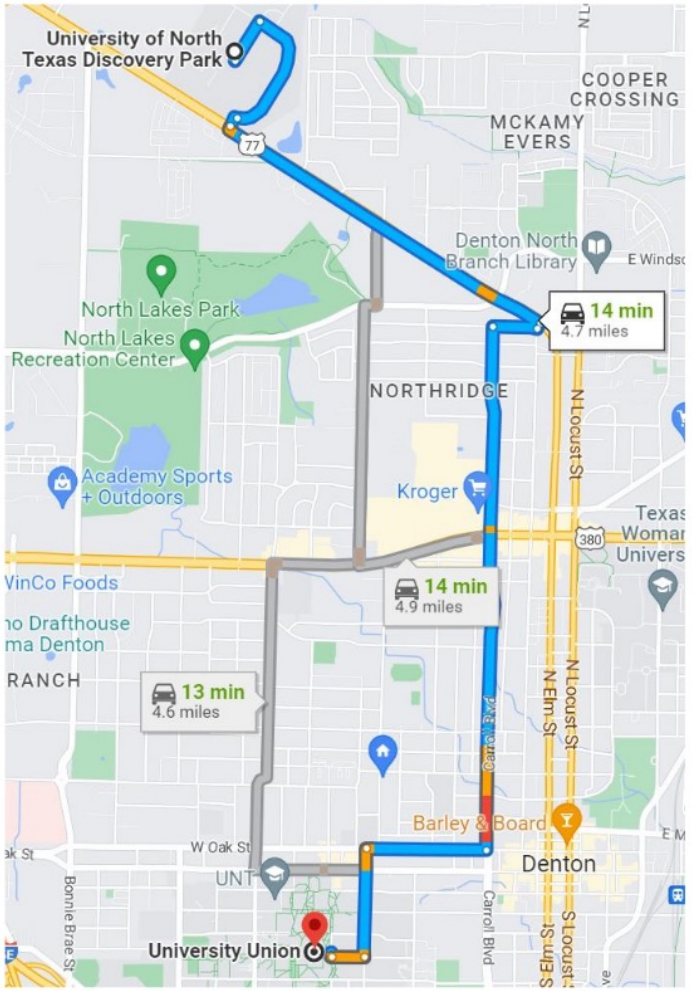

```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```





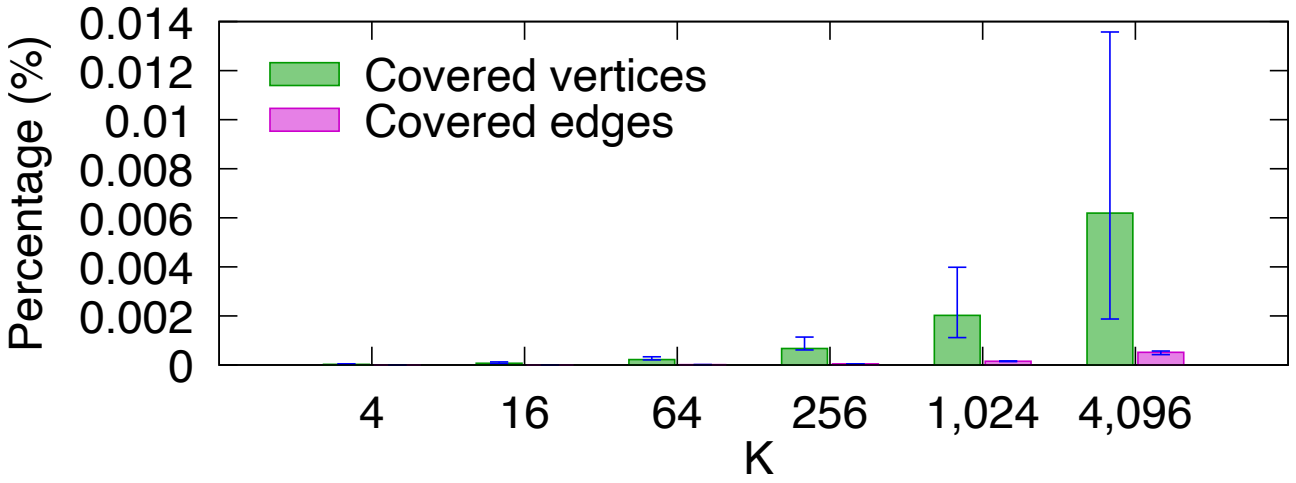
Top-*K* Shortest Path (KSP)

- *K shortest path (KSP)* algorithm aims to find the top-*K* shortest paths from the *source vertex s* to the *target vertex t*.



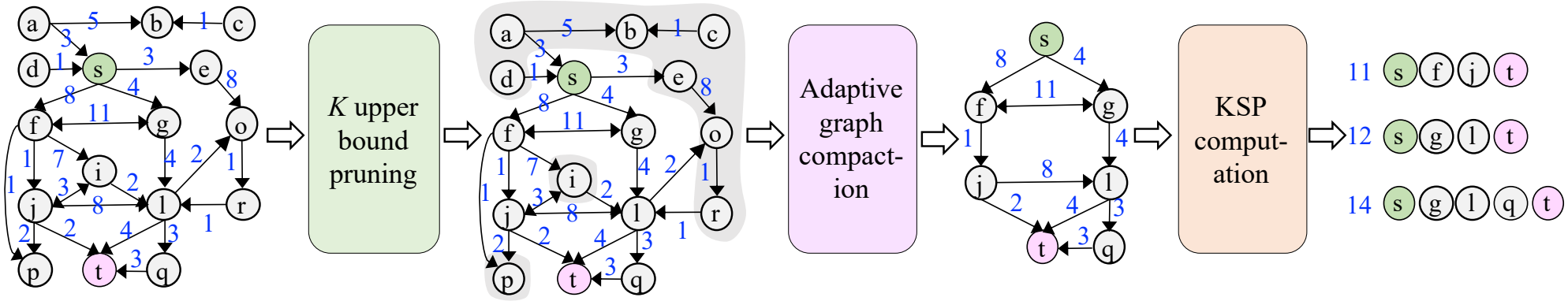
Motivation

- Key observation: Top-*K* shortest paths only cover an *extremely small portion* of the original graph, even for very large *K* values.
- Example
  - GAP-twitter with 61.6M vertices and 1.5B edges.



Overview of PeeK

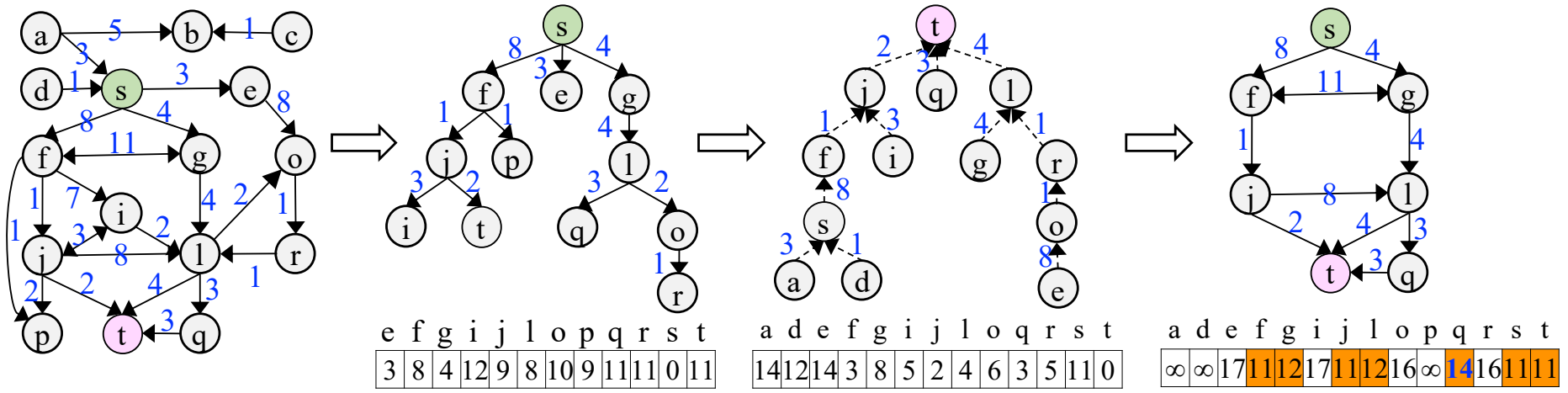
- PeeK
  - Step 1: K upper bound pruning.
  - Step 2: Adaptive graph compaction.
  - Step 3: KSP computation.



(a) An example graph (b) Identified unnecessary vertices and edges (c) Compacted graph (d) KSP results

Contribution #1: K Upper Bound Pruning

- K upper bound pruning
  - Step 1: SSSP computation based on source vertex.
  - Step 2: SSSP computation based on target vertex.
  - Step 3: Sum up two distances of each node based on two SSSP trees to find out K upper bound.



(a) An example graph.

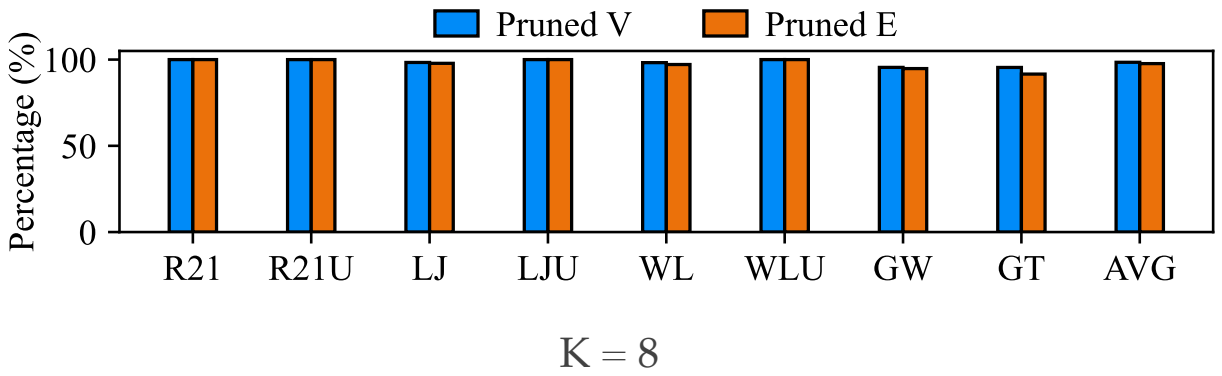
(b) The ***source*** SSSP tree and distance array ***spSrc***.

(c) The ***target*** SSSP tree and distance array ***spTgt***.

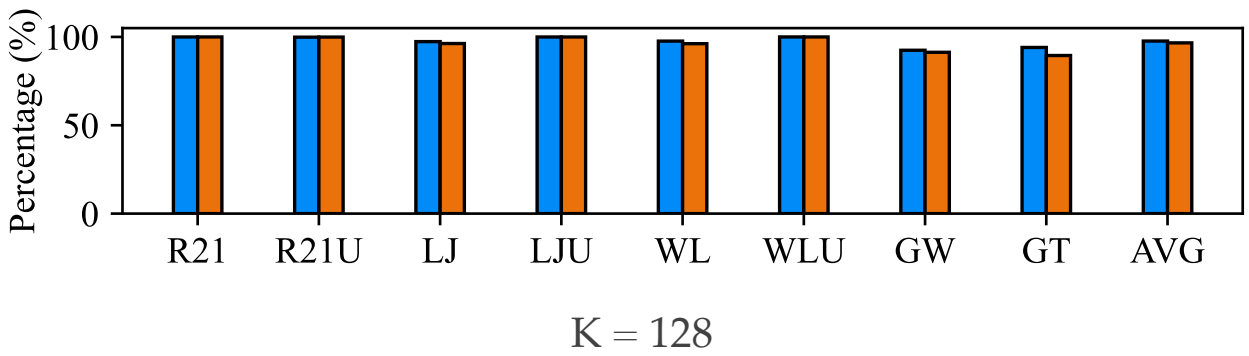
(d) The remaining graph and distance sum array ***spSum***.

Contribution #1: K Upper Bound Pruning

- Benefits:
  - Pruning 98.4% vertices and 97.7% edges on average for k equaling to 8.
  - Pruning 97.7% vertices and 96.6% edges on average for k equaling to 128.



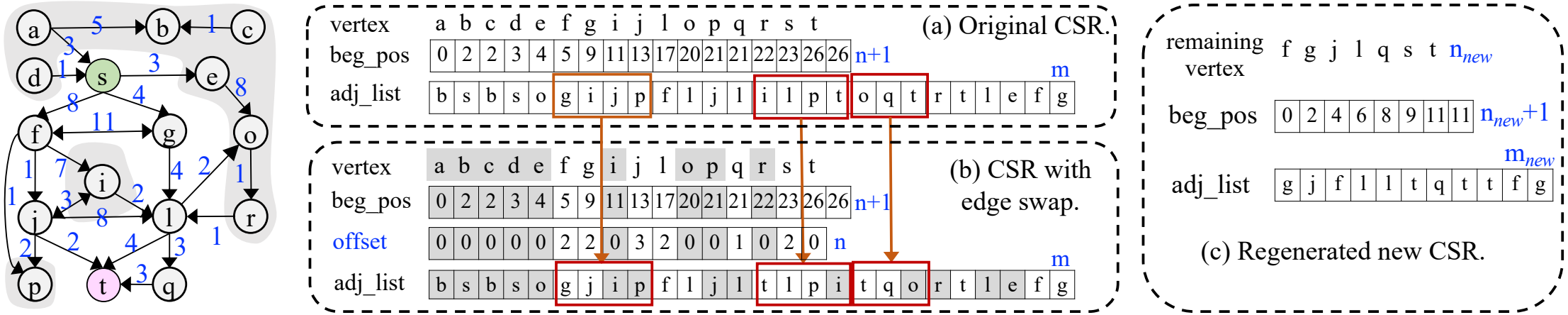
K = 8



K = 128

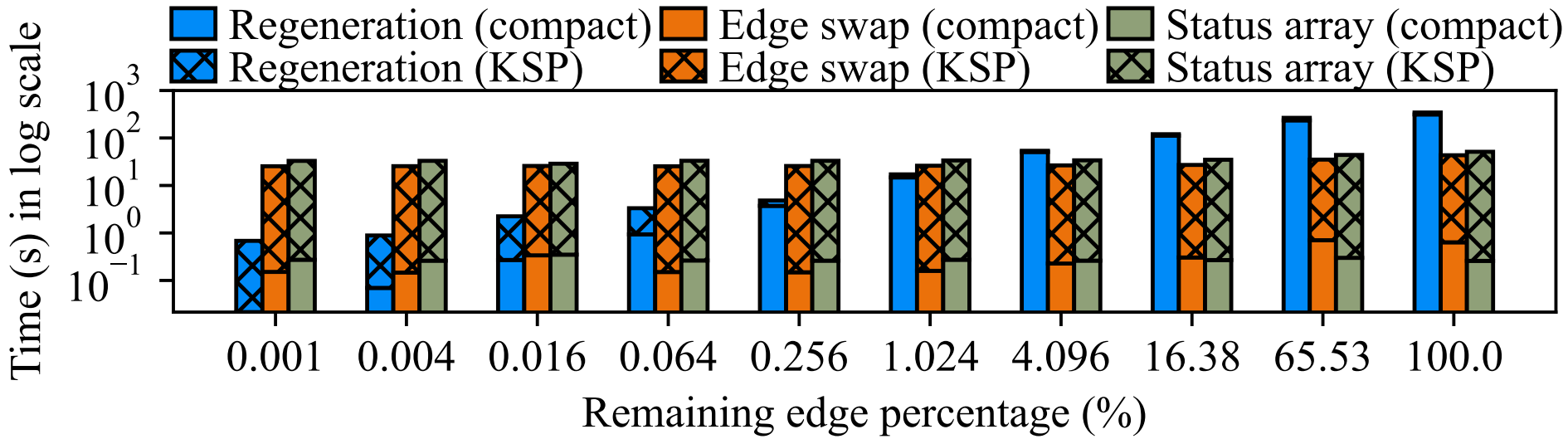
Contribution #2: Adaptive Graph Compaction

- Unique graph compaction patterns
  - Only involving vertex and edges deletion, not insertion.
  - The graph is updated three times, i.e., two SSSPs and K upper bound pruning.
- Two compaction strategies
  - Edge swap-based compaction.
  - Graph regeneration-based compaction.



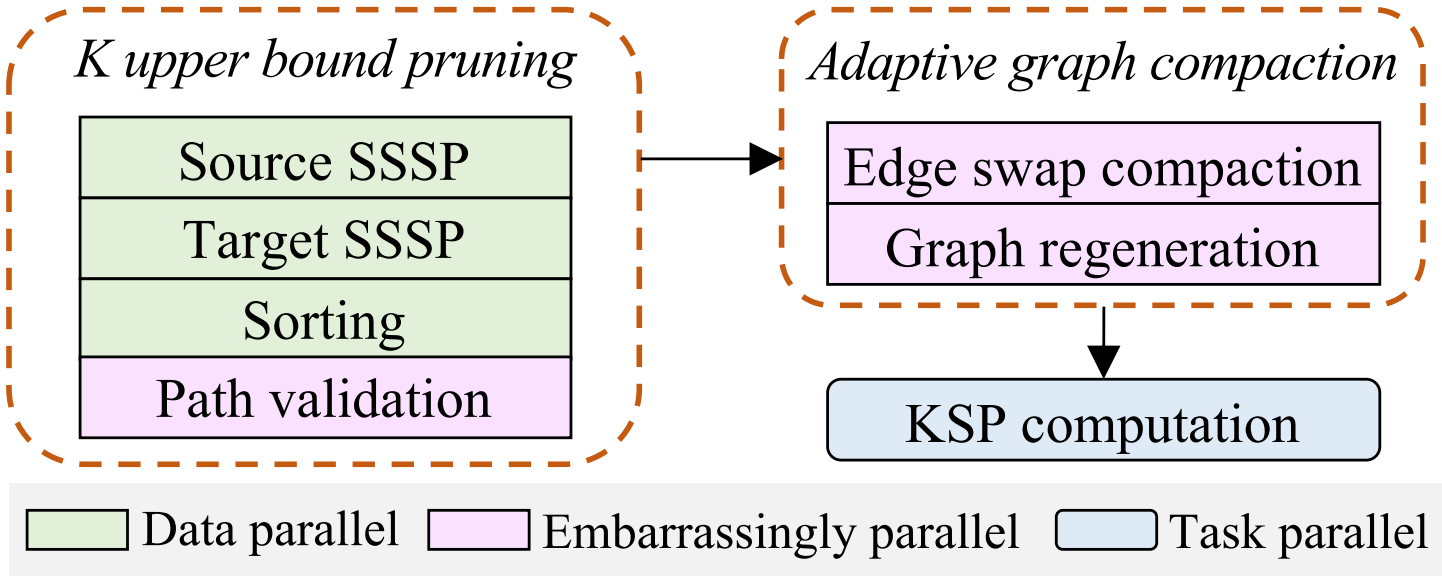
Contribution #2: Adaptive Graph Compaction

- Graph compactions comparison:
  - Status array-based (baseline)
  - Edge swap-based
  - Graph regeneration-based



Contribution #3: Parallel PeeK

- Data parallel
- Task parallel
- Shared-memory PeeK
- Distributed PeeK



Experiment

- Experiment setting

Table 1: Graph benchmarks (sorted by vertex count).

Graph	Abbr.	Graph type	# vertex	# edge	Weight
Rmat21	R21	Synthetic graph	2.1M	33.6M	random
Rmat21-U	R21U	Synthetic graph	2.1M	33.6M	1
Livejournal	LJ	Social network	4.8M	68.5M	random
Livejournal-U	LJU	Social network	4.8M	68.5M	1
Wikipedia	WL	Article network	13.6M	437.2M	random
Wikipedia-U	WLU	Article network	13.6M	437.2M	1
GAP-web	GW	Web network	50.6M	1.9B	real
GAP-twitter	GT	Social network	61.6M	1.5B	real

Performance Comparison

- PeeK outperforms all the other algorithms for both parallel and serial executions.
- PeeK achieves more speedup for larger K values.

		R21	R21U	LJ	LJU	WL	WLU	GW	GT	×
K=8	Yen	5.3	4.6	13.7	5.8	36.8	24.4	105.7	239.3	6.8
	NC	11.1	1.8	34	9.9	113.6	23.2	247.2	672.4	14.3
	OptYen	3.6	1.9	11.3	4.2	34	19.6	82.9	186.3	5.1
	PeeK	0.7	0.7	1.7	1.6	6.4	5.3	13	22.8	
		(5.1)	*(2.7)*	*(6.5)*	*(2.7)*	*(5.3)*	*(3.7)*	*(6.4)*	*(8.2)*	
K=128	Yen	68.3	61.2	196	44.1	337.1	93.6	174.8	2,168	56.7
	NC	188.6	10.6	665	192	2,405	204.8	-	-	170
	OptYen	39.6	7.8	150	10.8	272.4	24.9	132.4	1,131	28.8
	PeeK	0.8	0.7	2.5	1.6	6.9	5.4	13.6	23.1	
		(49.4)	*(11.5)*	*(60.8)*	*(6.6)*	*(39.2)*	*(4.7)*	*(9.7)*	*(48.9)*	

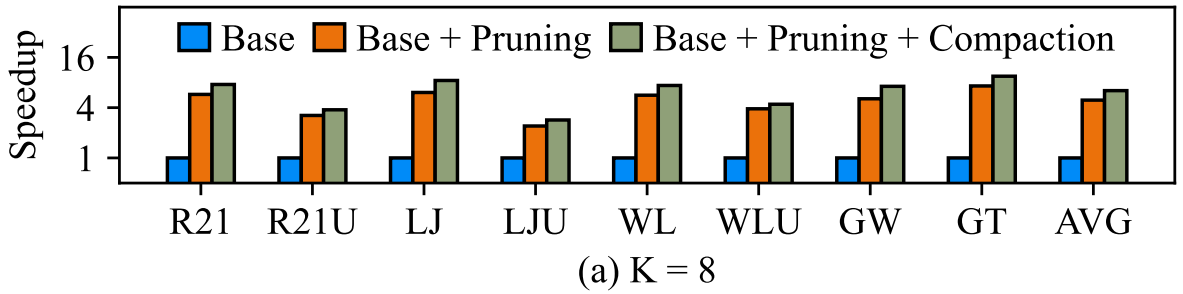
Parallel runtime (s)

		R21	R21U	LJ	LJU	WL	WLU	GW	GT	×
K=8	Yen	30.6	19.4	100.6	12.1	203	44.4	158.4	1,994	7.7
	NC	24.6	9.1	78	19.1	347.1	45	639.9	1,292	7.8
	OptYen	7.4	3.8	39	5.9	68.9	23	113.3	224.3	2.2
	SB	7.7	4.5	30.3	10.3	79.6	31.6	174.5	231.3	2.5
	SB*	7.7	4.4	17.8	8.4	69.6	29.4	196	223.7	2.2
	PeeK	3.5	3.1	7	5.6	26.3	18.6	56.5	115.4	
		(2.1)	*(1.2)*	*(2.5)*	*(1.05)*	*(2.6)*	*(1.2)*	*(2)*	*(1.9)*	
K=128	Yen	576.9	354	2,109	159.5	2,813	254.3	758.3	-	105.9
	NC	421.6	120	1,791	215.1	-	1,238.6	-	-	104
	OptYen	44.8	8.7	353	14.2	347.4	29.3	187.9	-	12.4
	SB	13.3	4.7	108.8	10.6	197.4	31.5	535.0	328.3	5.5
	SB*	9.3	4.6	31.5	8.8	86.8	30.5	430.9	291.3	3.1
	PeeK	3.5	3.1	7	5.7	26.5	18.6	58.3	118.1	
		(2.6)	*(1.5)*	*(4.5)*	*(1.5)*	*(3.3)*	*(1.6)*	*(3.2)*	*(2.5)*	

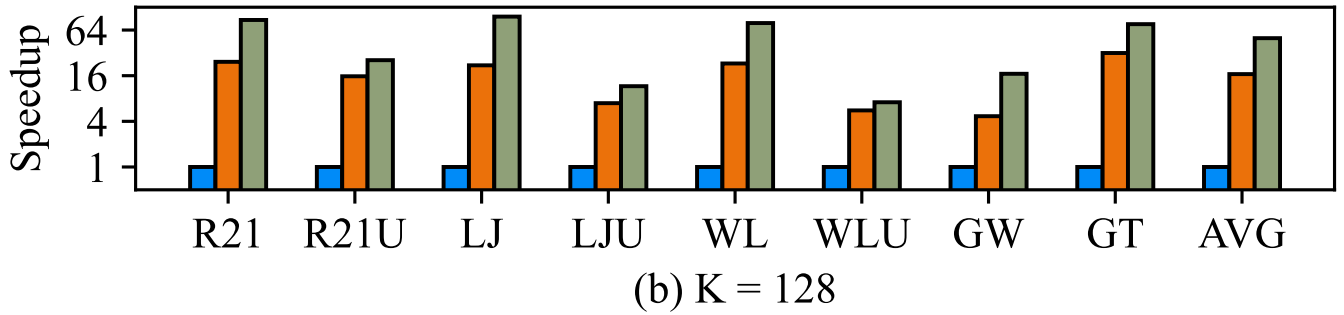
Serial runtime (s)

Technique Benefits

- Two techniques are tested against baseline (OptYen).
- K upper bound pruning technique contributes more than adaptive graph compaction.



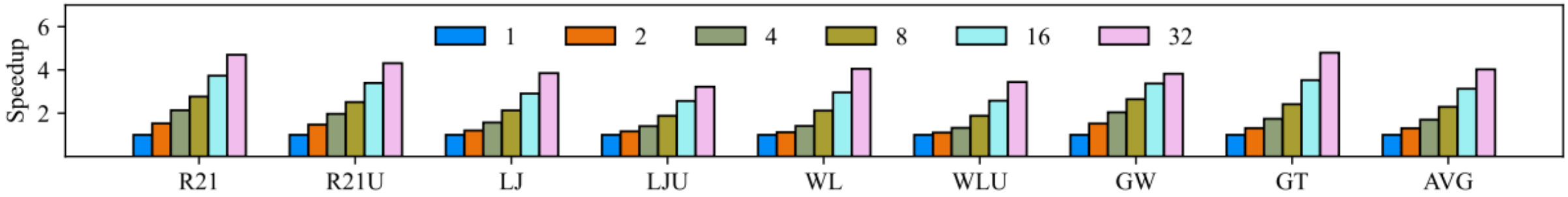
(a) K = 8



(b) K = 128

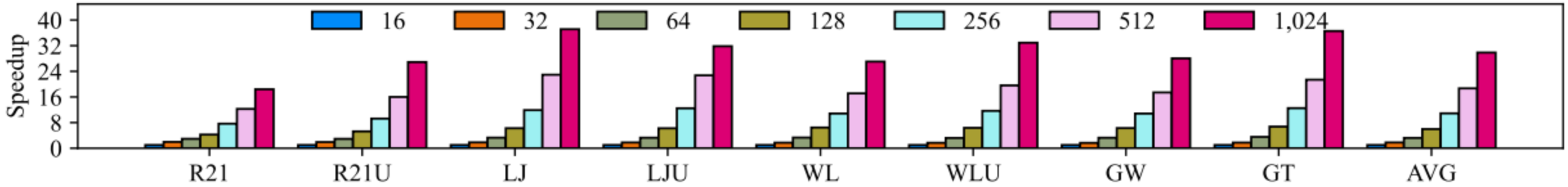
Shared-Memory Scalability

- PeeK shows a stable speedup increase when the thread count also increases.
- PeeK with 32 threads achieves 4* speedup on average, and highest speedup is 4.8* for graph GT.



Distributed Scalability

- The performance of PeeK stably improves with the increase of computing nodes and cores.



Key Takeaway

- K upper bound pruning is the key contribution, which impacts a lot to KSP computation.
- PeeK can integrate with existing KSP algorithms to boost their performance.

Acknowledgement

- Thank the anonymous reviewers for their valuable suggestions.
- Express our grateful thanks to the authors of OptYen for sharing the source code with us.

- It is not the final version.
- Source code
  - https://github.com/SC-Lab-Go/PeeK