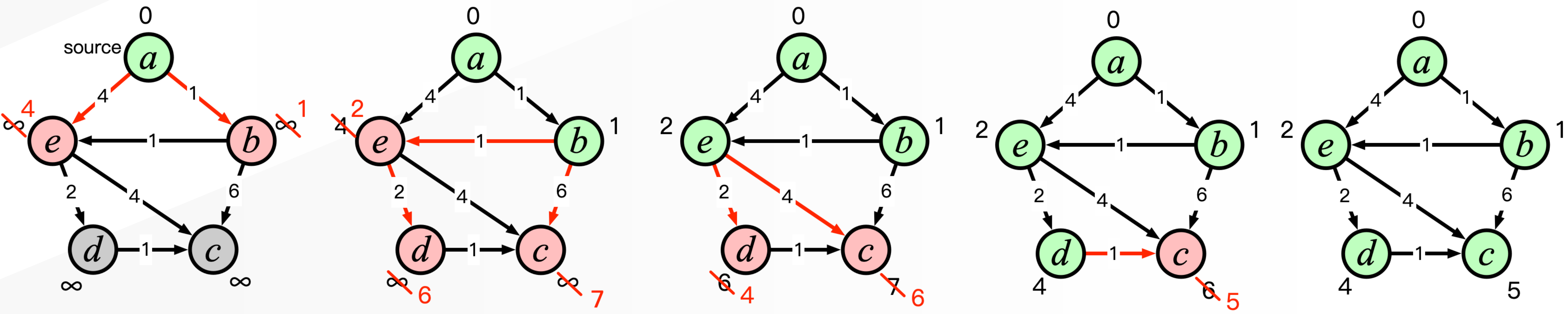
Fast Iterative Graph Computing with Updated Neighbor States

Yijie Zhou[1], **Shufeng Gong**[1], Feng Yao[1], Hanzhang Chen[1], Song Yu[1],
Pengxi Liu[1], Yanfeng Zhang[1], Ge Yu[1], Jeffrey Xu Yu[2]
Northeastern University[1], The Chinese University of Hong Kong[2]

Iterative Computation

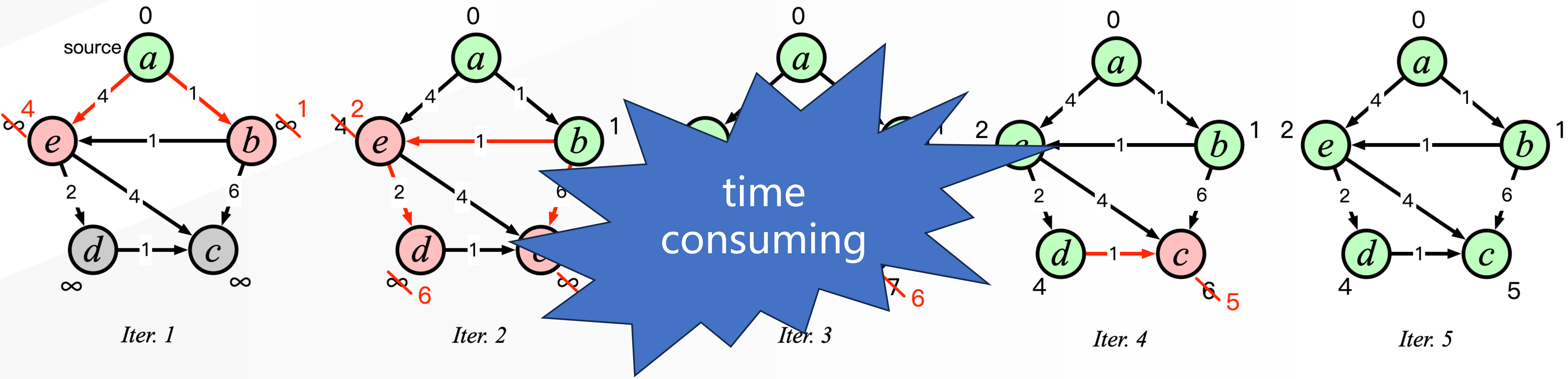
- PageRank, SSSP, BFS, ……
- Traversing the entire graph multiple times



Iter. 1 *Iter. 2* *Iter. 3* *Iter. 4* *Iter. 5*

Iterative Computation

- PageRank, SSSP, BFS, ……
- Traversing the entire graph multiple times



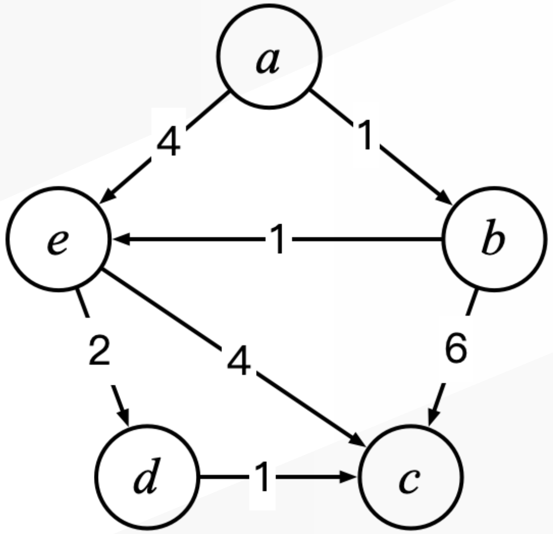
Accelerate Iterative Computation

- How to accelerate iterative computation?
  - Reduce the time per iteration
  - <u>Decrease the number of iterations</u>

Vertex State Changes in Iterative Rounds

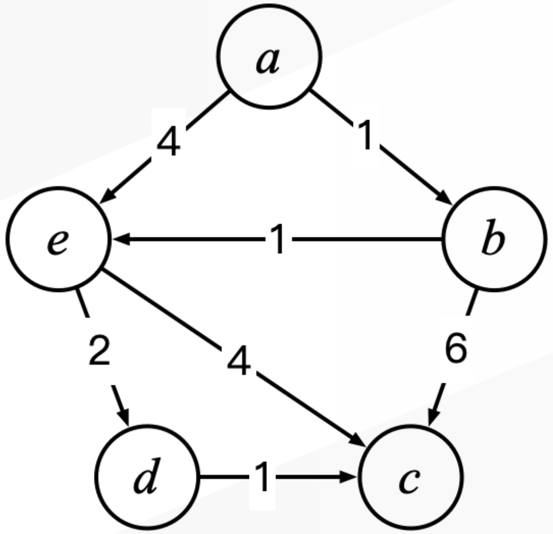
Synchronous

	a	b	c	d	e
0	0	∞	∞	∞	∞



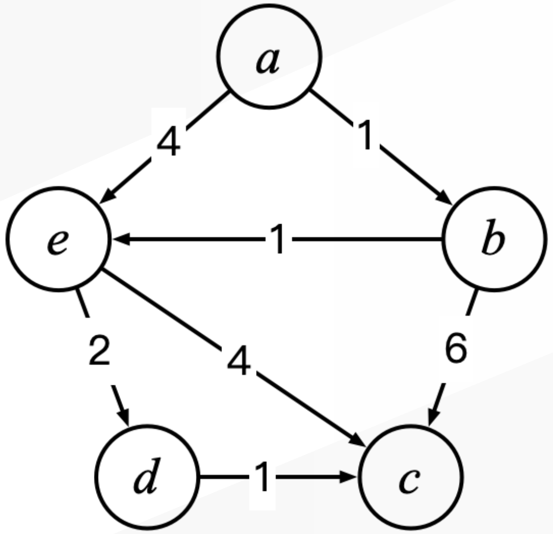
Vertex State Changes in Iterative Rounds

Synchronous

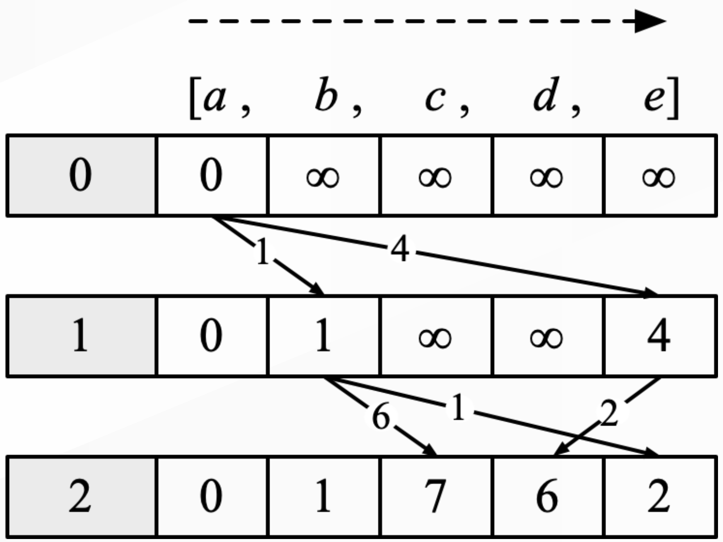


	[a ,	b ,	c ,	d ,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4

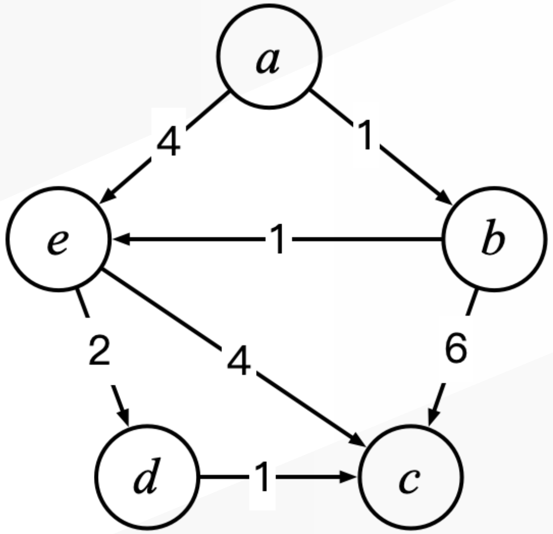
Vertex State Changes in Iterative Rounds



Synchronous



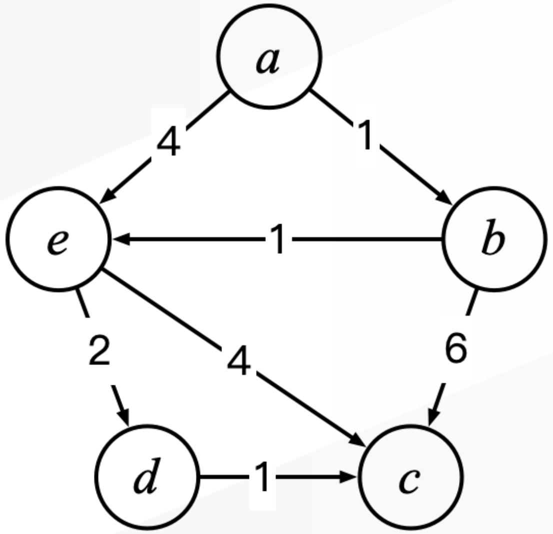
Vertex State Changes in Iterative Rounds



Synchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2

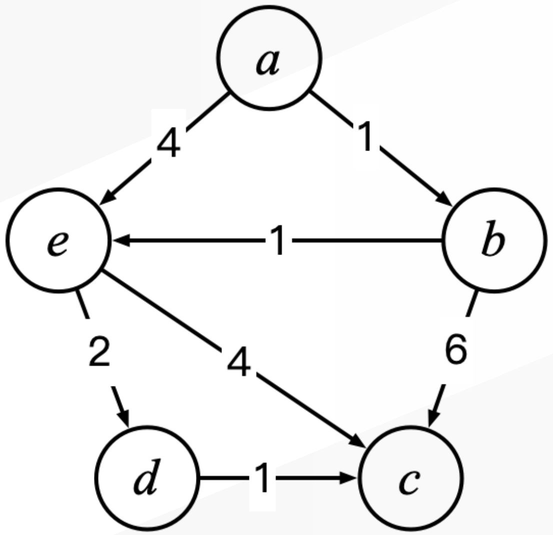
Vertex State Changes in Iterative Rounds



Synchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2
4	0	1	5	4	2

Vertex State Changes in Iterative Rounds



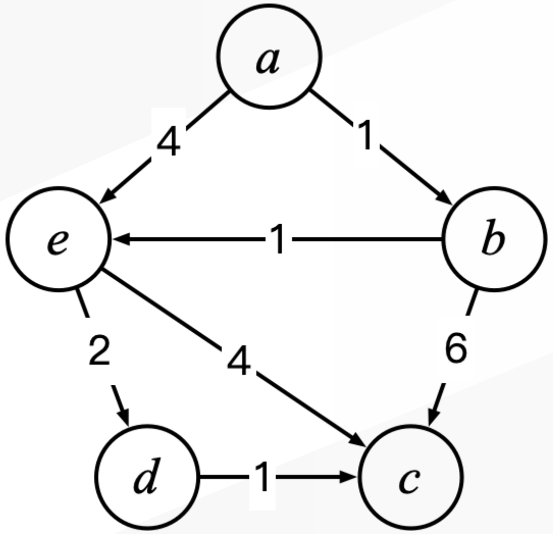
Synchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2
4	0	1	5	4	2

Asynchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞

Vertex State Changes in Iterative Rounds



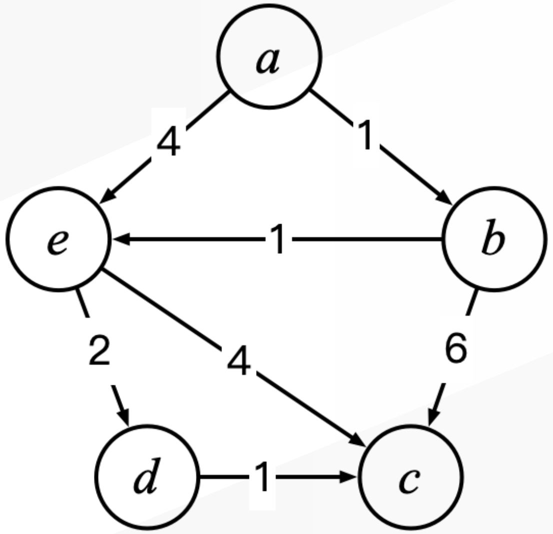
Synchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2
4	0	1	5	4	2

Asynchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	7	∞	2

Vertex State Changes in Iterative Rounds



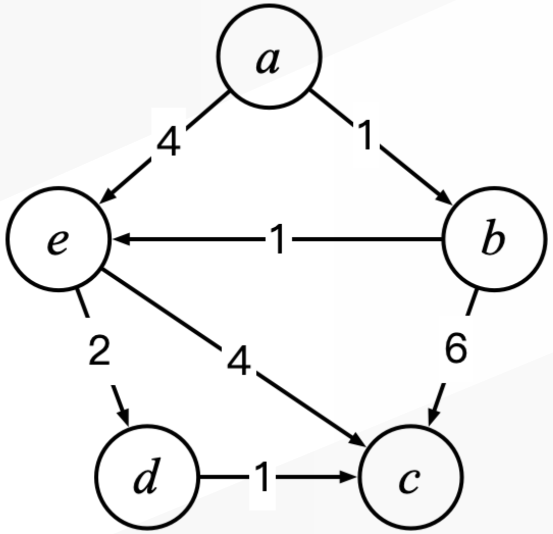
Synchronous

	[a ,	b ,	c ,	d ,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2
4	0	1	5	4	2

Asynchronous

	[a ,	b ,	c ,	d ,	e]
0	0	∞	∞	∞	∞
1	0	1	7	∞	2
2	0	1	6	4	2

Vertex State Changes in Iterative Rounds



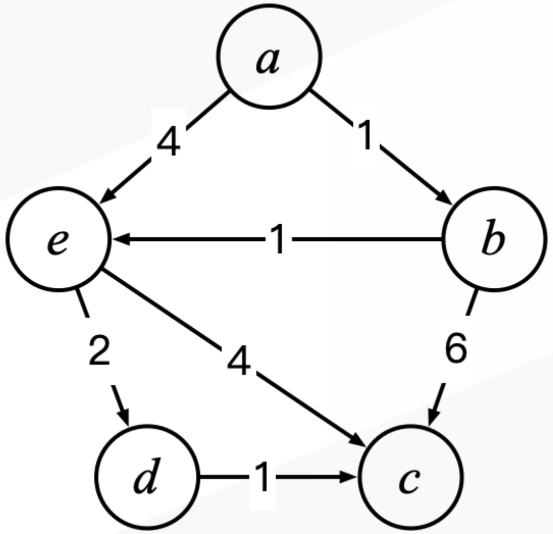
Synchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2
4	0	1	5	4	2

Asynchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	7	∞	2
2	0	1	6	4	2
3	0	1	5	4	2

Vertex State Changes in Iterative Rounds



Synchronous

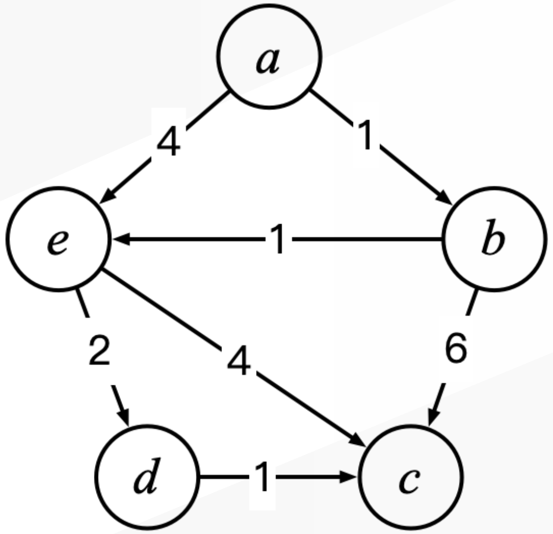
	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	1	∞	∞	4
2	0	1	7	6	2
3	0	1	6	4	2
4	0	1	5	4	2

Asynchronous

	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	1	7	∞	2
2	0	1	6	4	2
3	0	1	5	4	2

Asynchronous iteration reduces the number of rounds because each vertex can immediately use the latest state value.

Vertex State Changes in Iterative Rounds



Asynchronous

[a , b , c , d , e]

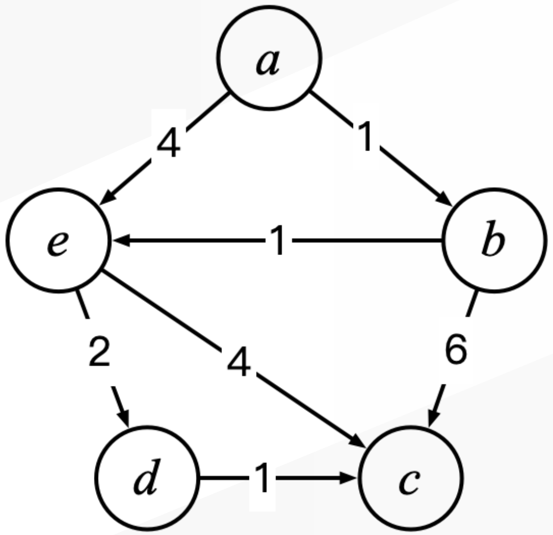
	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	1	7	∞	2
2	0	1	6	4	2
3	0	1	5	4	2

Asynchronous
with reordered order

[a , b , e , d , c]

	a	b	e	d	c
0	0	∞	∞	∞	∞

Vertex State Changes in Iterative Rounds



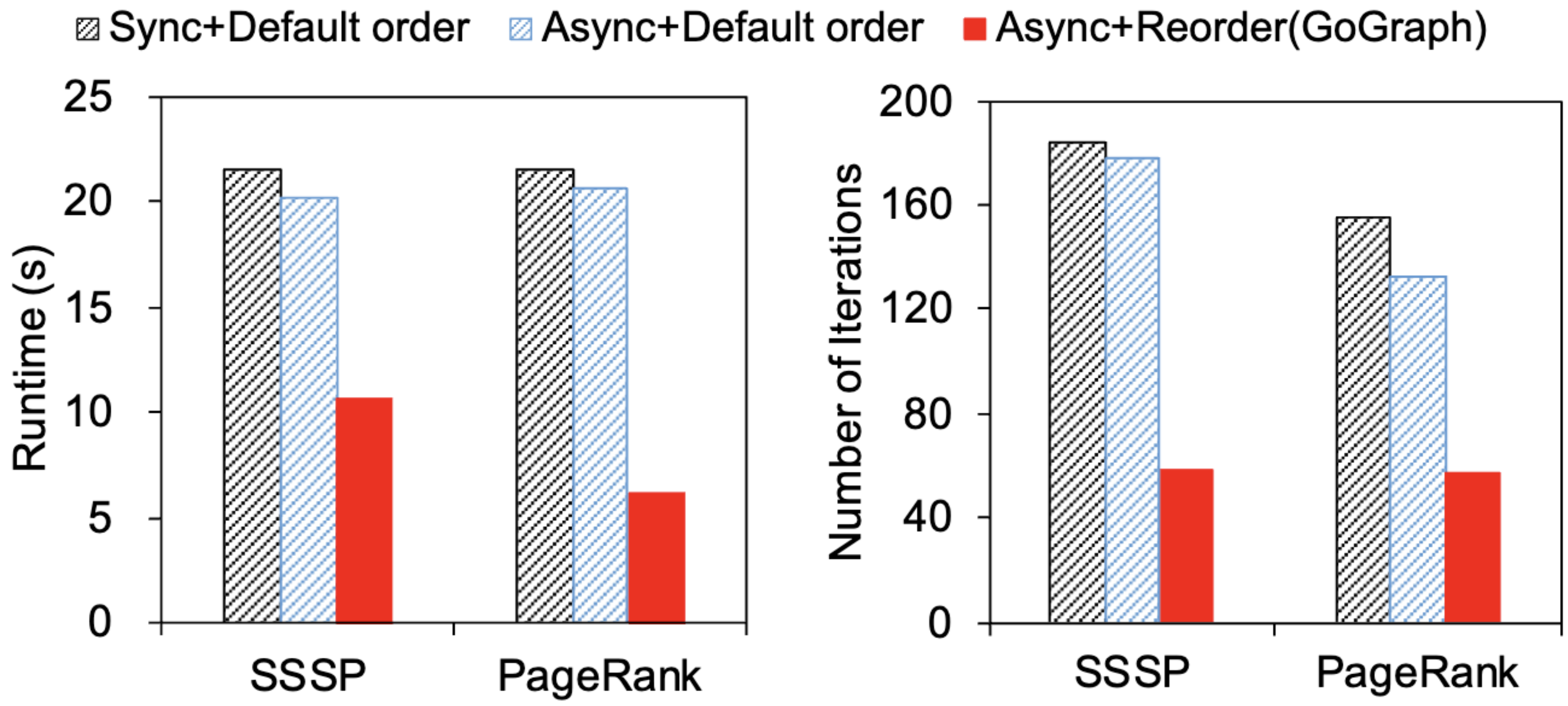
Asynchronous

	[a,	b,	c,	d,	e]
0	0	∞	∞	∞	∞
1	0	1	7	∞	2
2	0	1	6	4	2
3	0	1	5	4	2

Asynchronous with reordered order

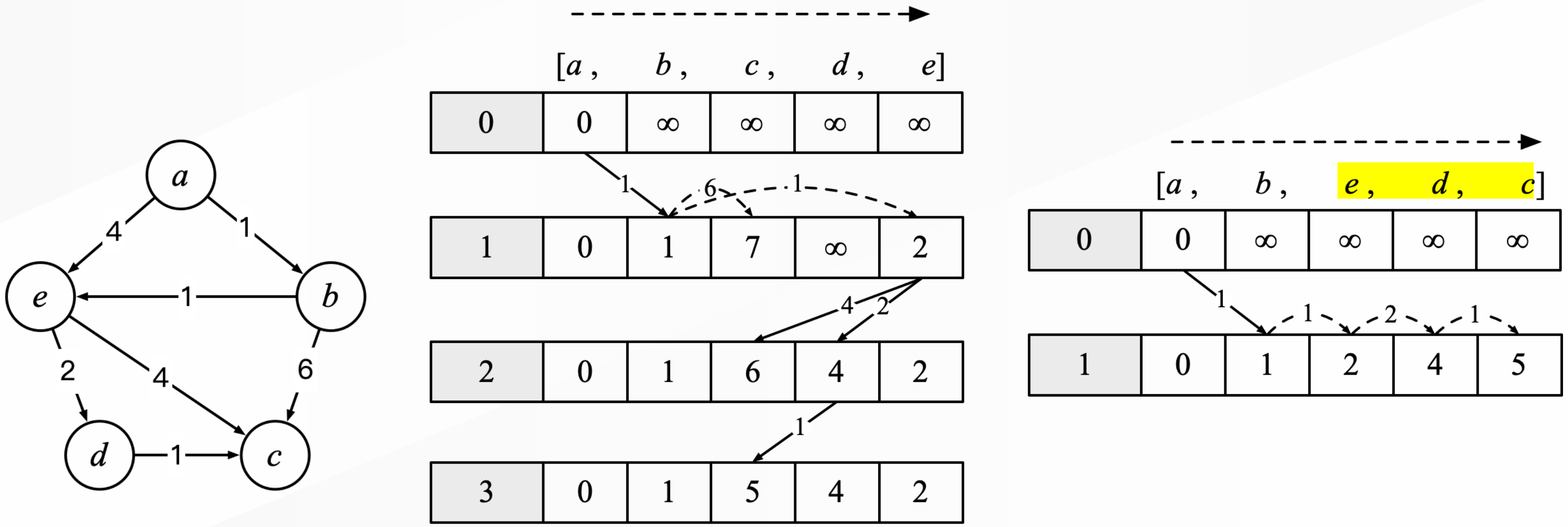
	[a,	b,	e,	d,	c]
0	0	∞	∞	∞	∞
1	0	1	2	4	5

Converging Quicker After Reordering



(a) Runtime (b) Iteration rounds

Goal



- Construct an efficient vertex processing order to accelerate the iterative computation.

Challenges

- Which processing order is better?
  - Challenge 1. Design a metric to measure the quality of the processing order.
- How to reorder the vertex to make the iterations converge faster?
  - Challenge 2. Design a vertex reordering method.

Positive/Negative Edge

For an existing edge: $u \rightarrow v$

- Processing order 1: $O_V = [a, b, \ldots, u, \ldots, v, \ldots, z]$
  - <u,v> is a positive edge, since *v* can use *u*’s latest state in the same round.
- Processing order 2: $O_V = [a, b, \ldots, v, \ldots, u, \ldots, z]$
  - <u,v> is a negative edge, since *v* can only use *u*'s latest state in the next round.

Metric Funtion

Intuition: as mentioned before, the more positive edges, the more vertex state values can be utilized per round, speeding up convergence.

- Counts the number of positive edges ($u \rightarrow v$, u is processed prior to v)

$$M(O_V) = n\left(e_{pos}\right) = \sum_{(u,v)\in E} \chi(u,v)$$

$$\chi(u,v) = \begin{cases} 1, & if\ (u,v)\ is\ positive, \\ 0, & if\ (u,v)\ is\ negative. \end{cases}$$

- The goal of our reordering method: maximize the quantity of positive edges

GoGraph

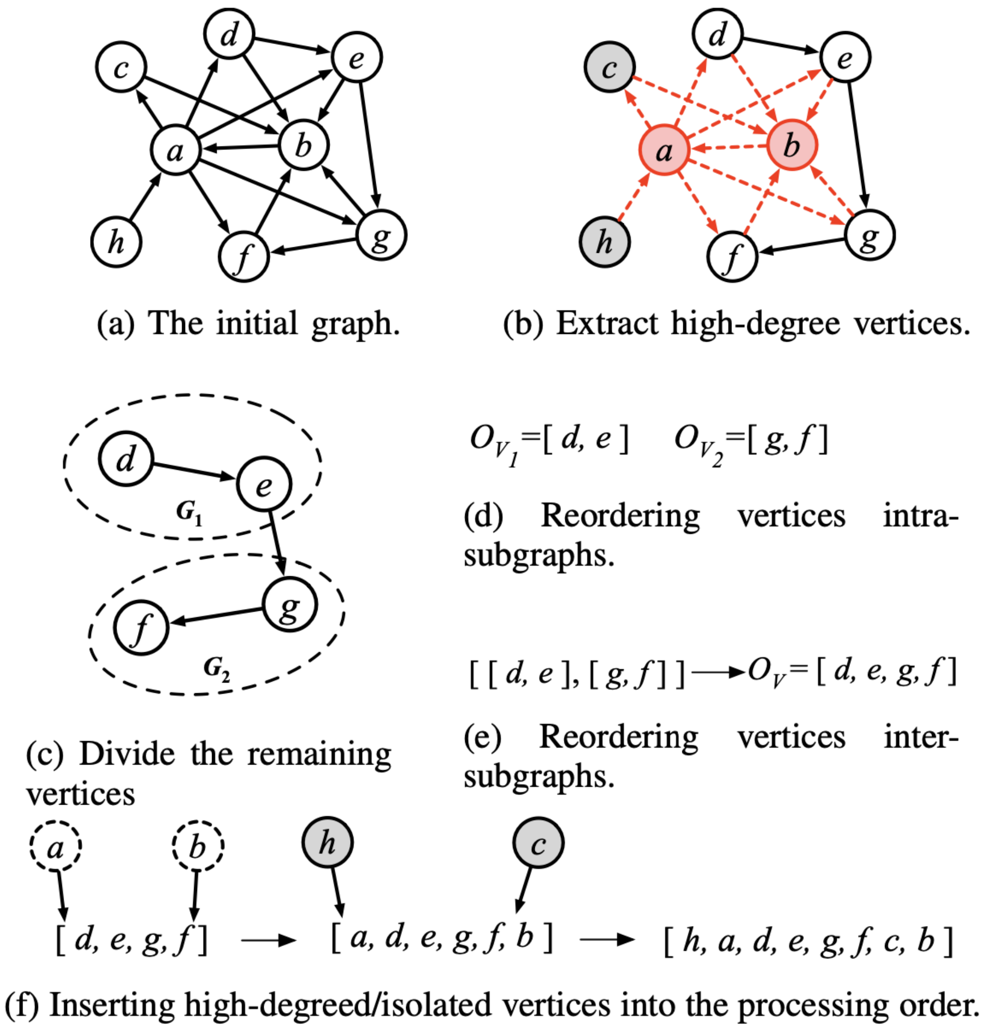
1) Extract high-degree and isolated vertices

2) Divide other vertices

3) Reorder vertices within subgraphs

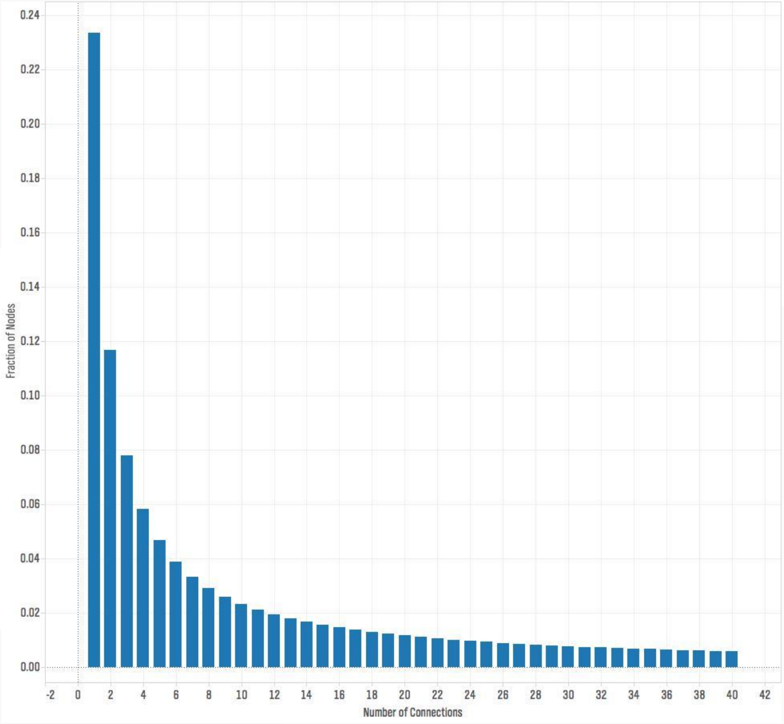
4) Reorder subgraphs

5) Insert high-degree and isolated vertices

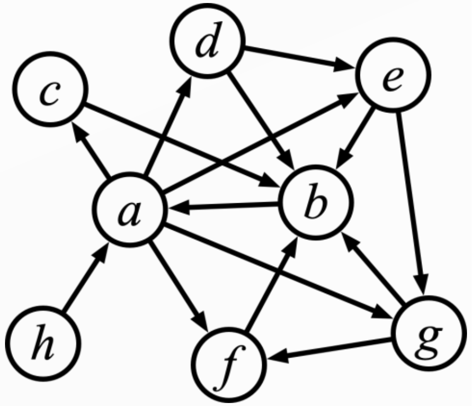


GoGraph

1) Extract high-degree and isolated vertices

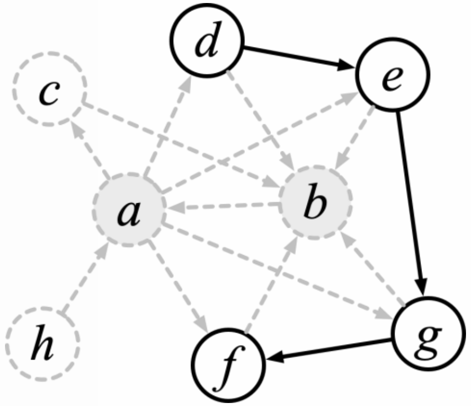


Distribution of vertices of different degrees for scale-free graphs



$O_V^1 = [d, e, c, b, h, a, g, f]$

$\mathrm{M}(O_V^1) = 10$



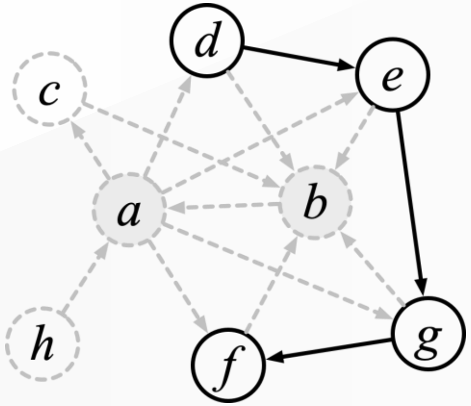
$O_V^2 = [h, a, c, d, e, g, f, b]$

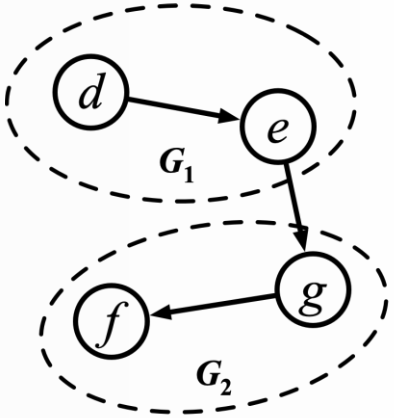
$\mathrm{M}(O_V^2) = 14$

GoGraph

2) Divide other vertices

- Louvain, Metis, …

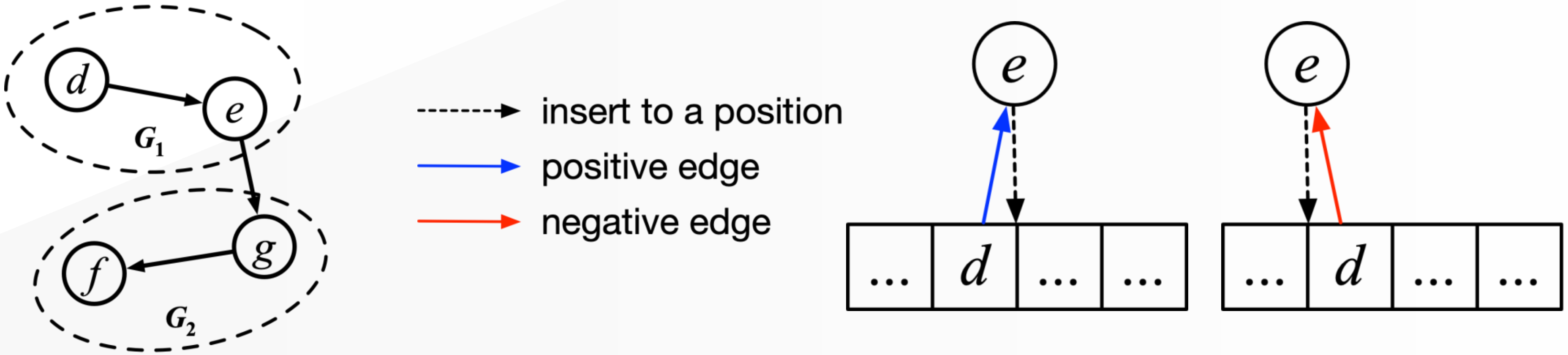




GoGraph

3) Reorder vertices within subgraphs

- Calculate the *M* value based on where the vertices are inserted;
- Find the position that maximizes *M* (maximize the number of positive edges and minimize the number of negative edges).



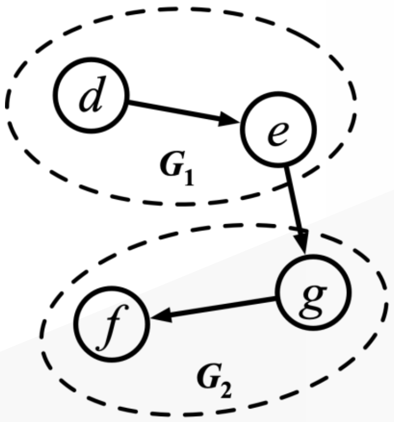
$O_{G_1} = [d, e]$

$O_{G_2} = [g, f]$

GoGraph

4) Reorder subgraphs

- Consider each subgraph as a super vertex;
- Perform the same operation in step 3.

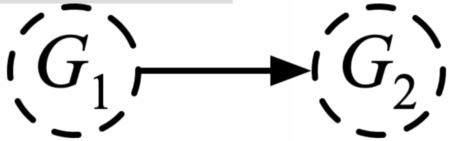


$$w_{G_1,G_2} = \left|\left\{\{u,v\} \middle| u \in G_i, v \in G_j\right\}\right|$$

$$M(O_G) = \sum_{(G_i,G_j)\in P} \chi(G_i,G_j)$$

$$\chi\left(G_i,G_j\right) = \begin{cases} w_{G_1,G_2}, & if\ e_{(G_i,G_j)}\ is\ positive, \\ 0, & if\ e_{(G_i,G_j)}\ is\ negative. \end{cases}$$

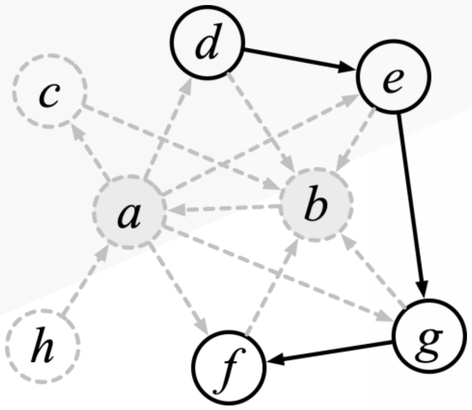
$O_{G_1} = [d,e]$ $O_{G_2} = [g,f]$

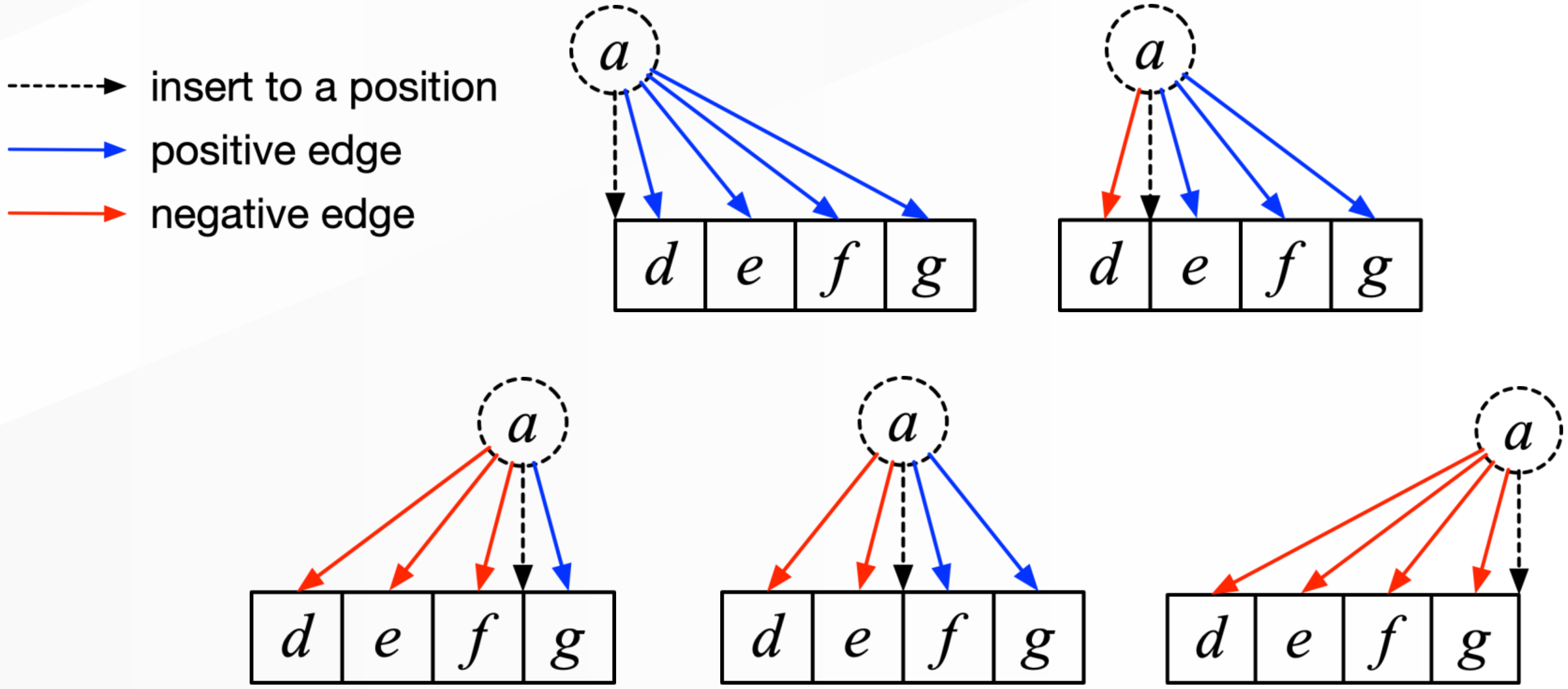


$O_G = [d,e,g,f]$

GoGraph

5) Insert high-degree and isolated vertices





[h, a, d, e, g, f, c, b]

Experiments

- Competitors
  Degree Sorting, Hub Sorting, Hub Clustering, Rabbit, Gorder
- Workloads
  PageRank, SSSP, BFS, PHP
- Environment
  Linux server, 98 GB RAM, Ubuntu 22.04 (64-bit), GCC 7.5
- Datasets

Dataset	Vertices	Edges	Abbreviation
Indochina [16]	11,358	49,138	IC
SK-2005 [16]	121,422	36,7579	SK
Google [46]	875,713	5,241,298	GL
Wiki-2009 [16]	1,864,433	4,652,358	WK
Cit-Patents [47]	3,774,768	18,204,371	CP
LiveJournal [16]	4,033,137	27,972,078	LJ

Overall performance

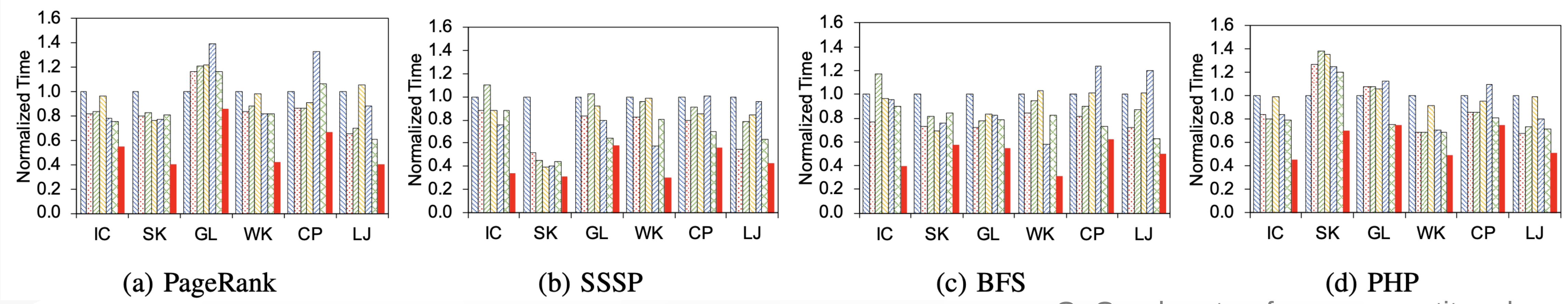


Fig. 1 The comparison of runtime

GoGraph outperforms competitors by an average of 1.83× in runtime.

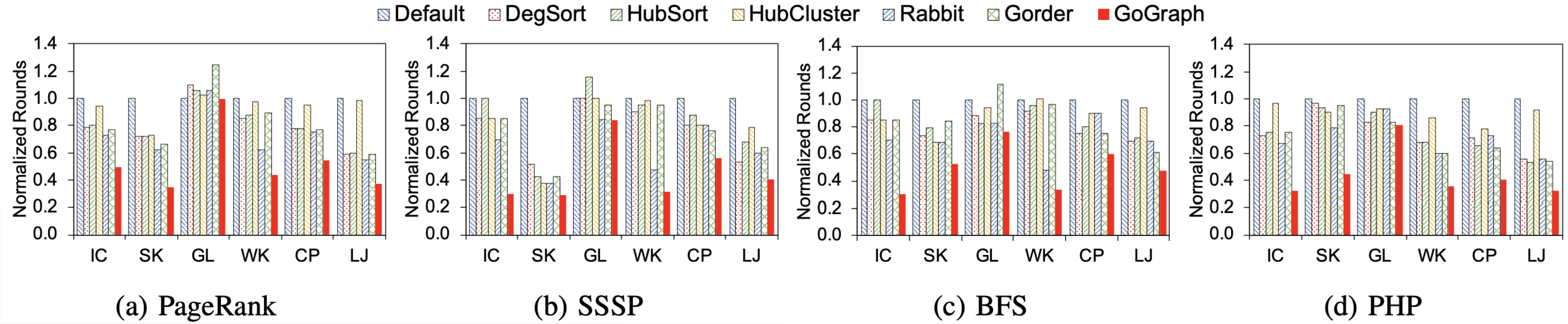


Fig. 2 The comparison of iteration rounds

GoGraph outperforms competitors by 41% on average reduction in iteration rounds.

Convergence comparison

- The distance from the state value at time t to the converged state value: $dist_t = \left|\sum_{v \in V} x^* - \sum_{v \in V} x_t\right|$

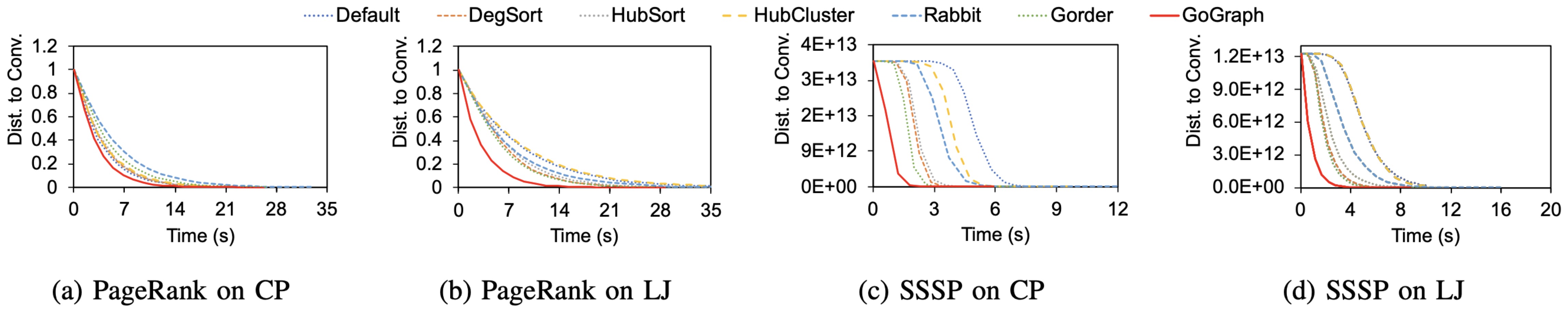


Fig. 3 The comparison of convergence speed

GoGraph algorithm consumes 59% of the average time used by competitors (with a minimum requirement of 37%) to reach convergence.

CPU cache miss

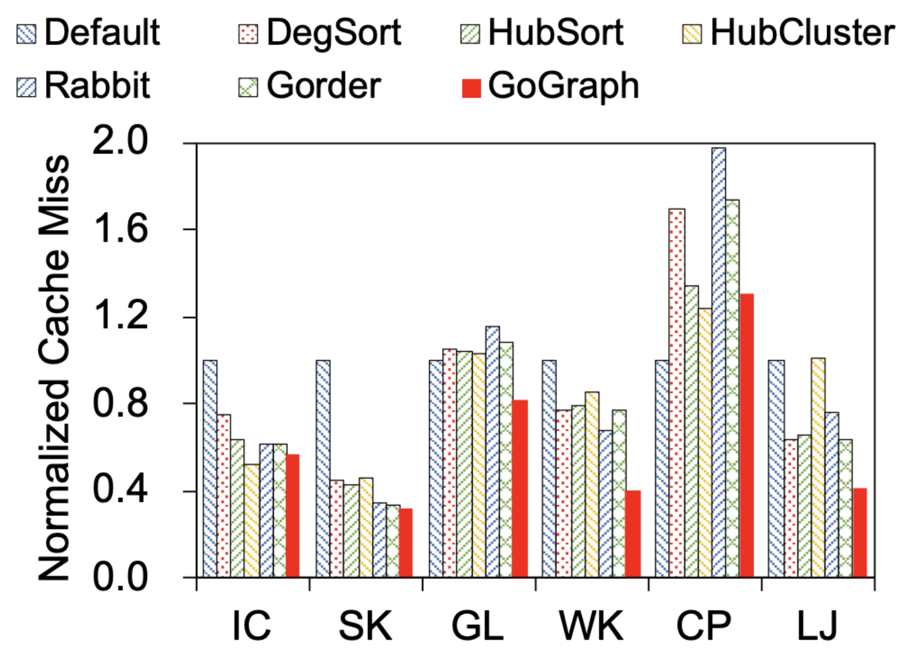


Fig. 4 The comparison of CPU cache miss

GoGraph can reduce the cache miss by 30% on average.

Conclusion

- We propose GoGraph, a graph reordering algorithm
- We propose a metric to measure the efficiency of the vertex processing order

Thank you for listening!