Research Portfolio

Chaeeun Kim

Table of Contents

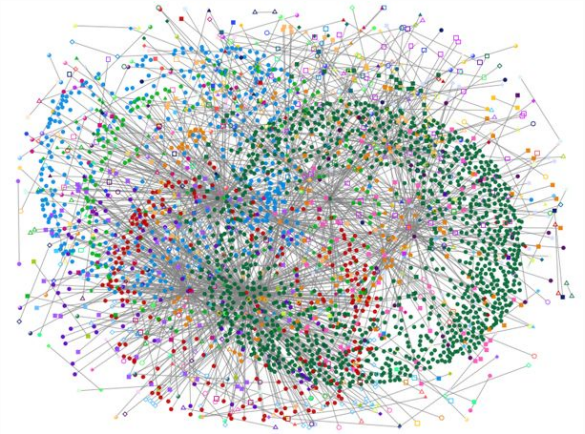
UniCon: A Unified Star-Operation to Efficiently Find Connected Components on a Cluster of Commodity Hardware

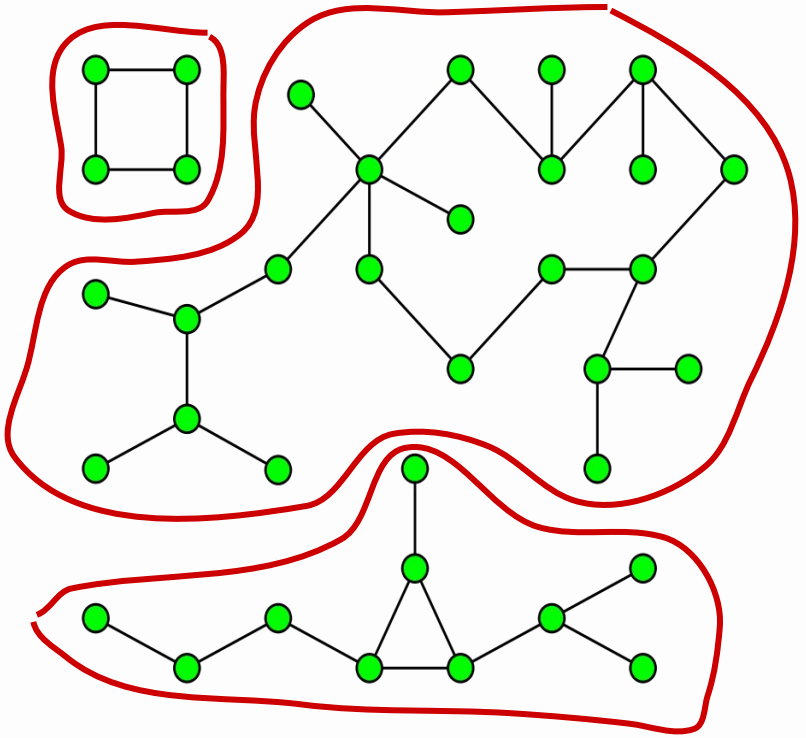
Topic Overview

Topic Overview

Finds all connected components on enormous graph using distributed machines

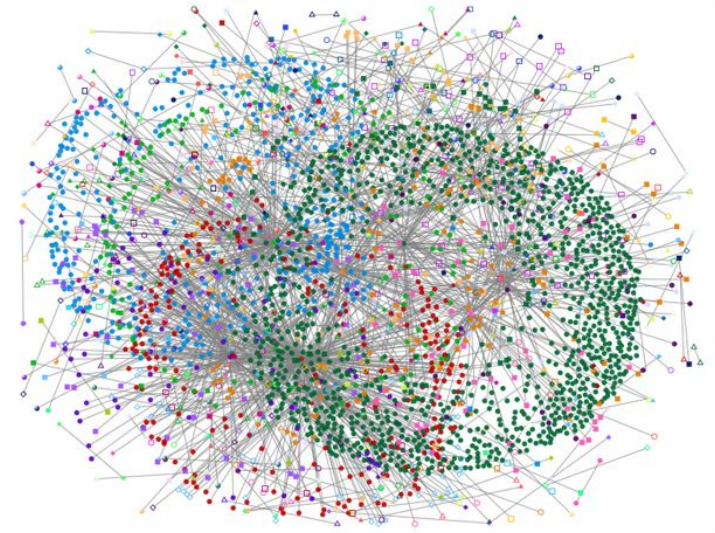
Cluster Server





3 connected components

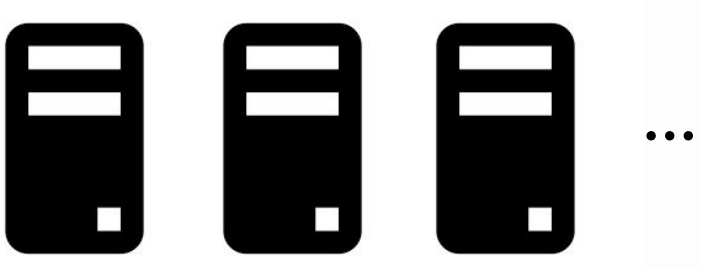
Graphs are Enormous



Twitter: 322 millions of users (2021)

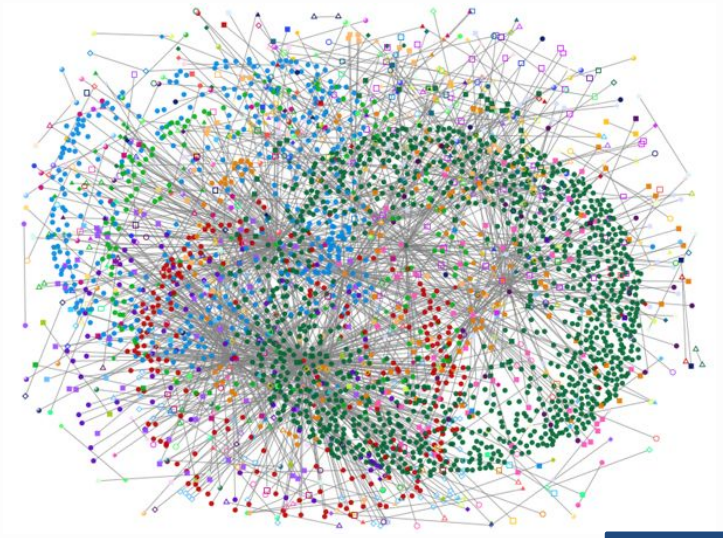
Facebook: 3 billions of active users (2021)

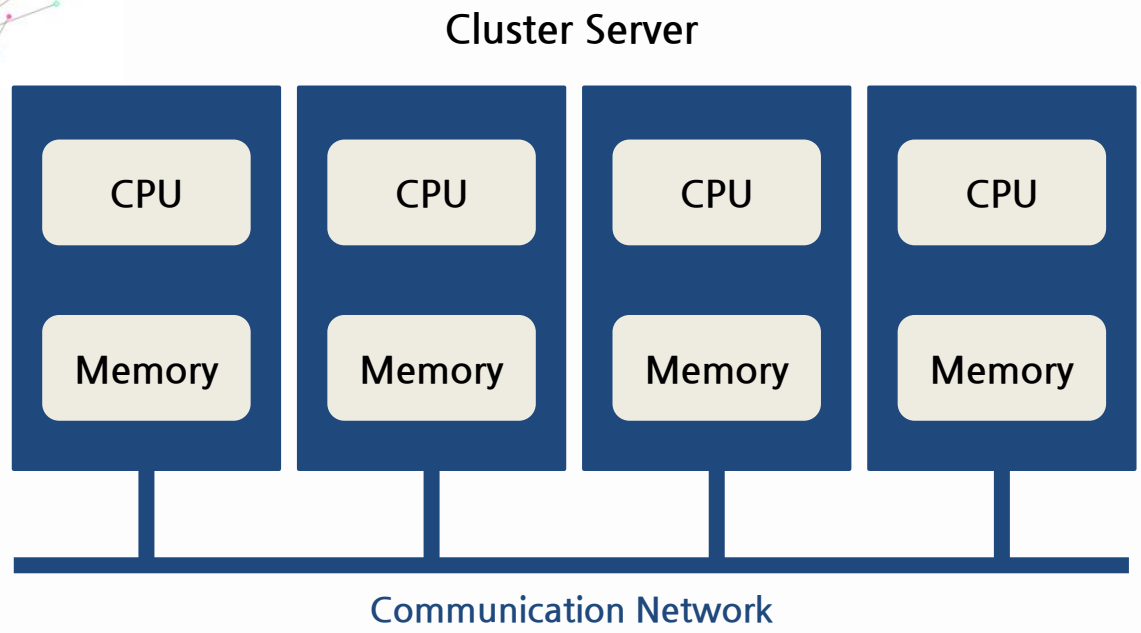
Single Machine



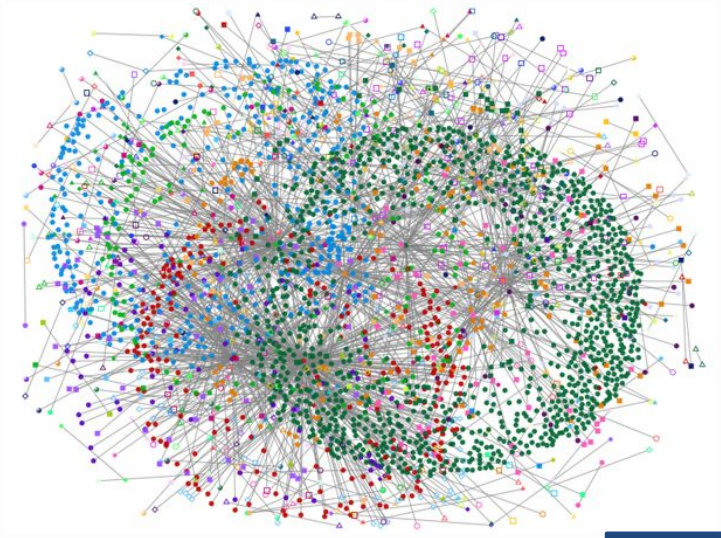
Cluster Server

MapReduce



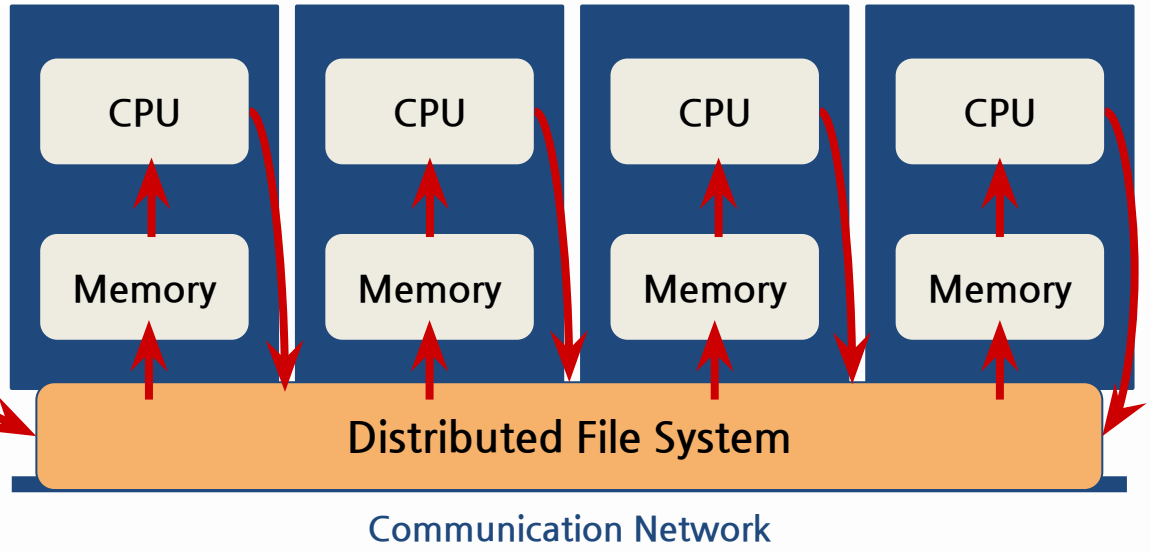

Cluster Server
CPU
CPU
CPU
CPU
Memory
Memory
Memory
Memory
Communication Network

MapReduce



edge list format

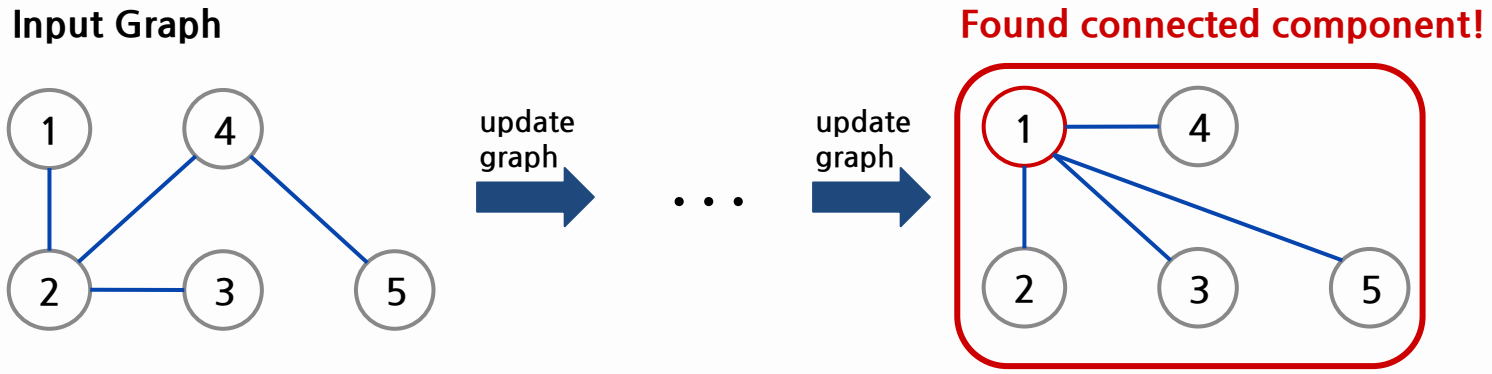
0	1
1	2
3	5
10	8
9	11
1	10



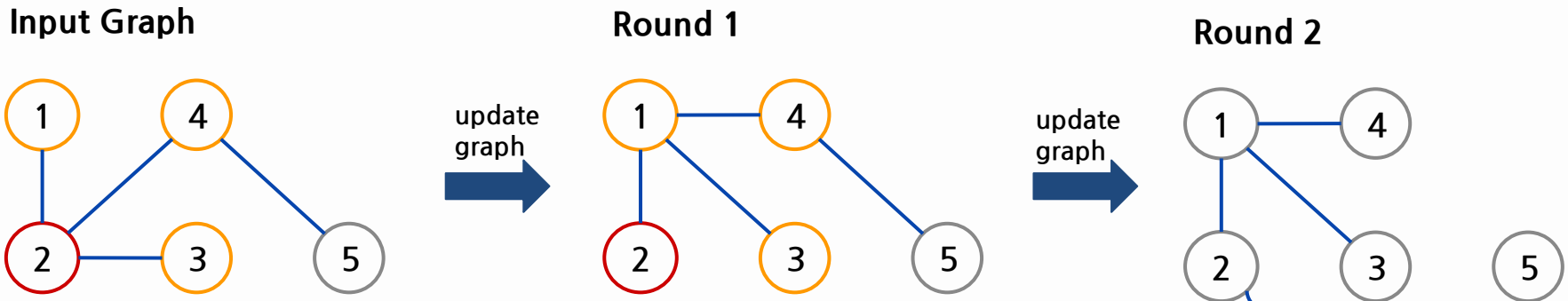
Background & Related work

Problem Definition

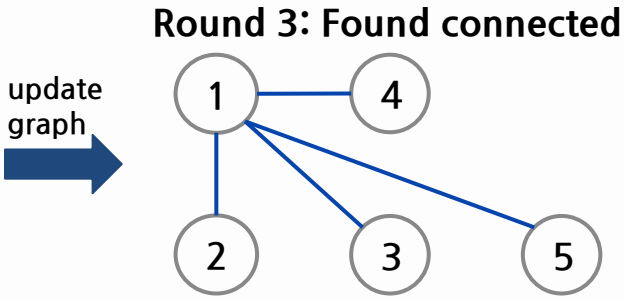
Finding connected components



Hash-to-Min: Node-centric Operation

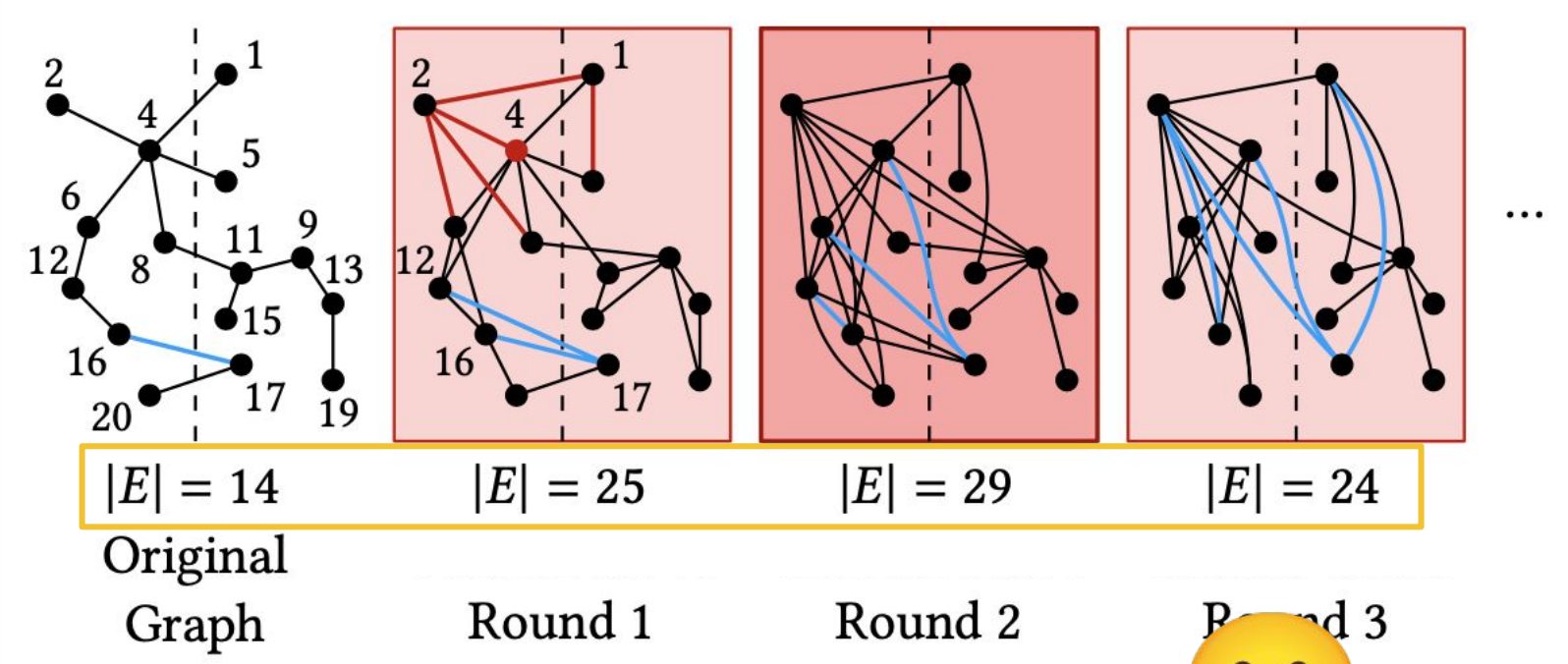


For node 2,
see all neighboring nodes {1, 2, 3, 4} → connect all to smallest node 1



Data explosion problem
& Load balancing problem

Hash-to-Min: Node-centric Operation

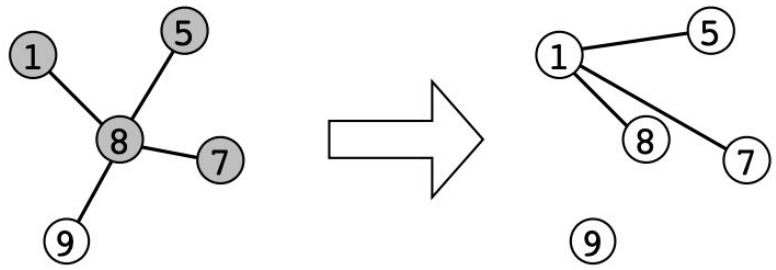


Data Explosion Problem!

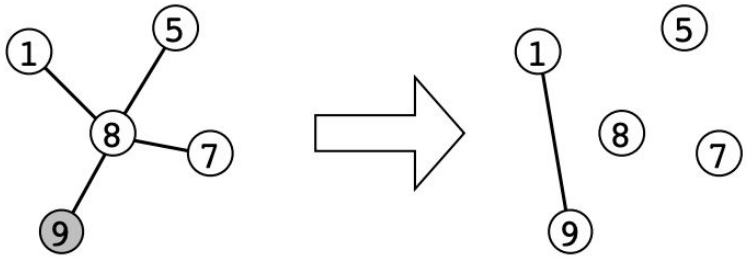
How to resolve data explosion problem?

in existing MapReduce algorithm? → Divide node-centric operation!

Ex: Alternating algorithm



(a) The small-star operation at node 8.



(b) The large-star operation at node 8.

Load balancing problem

& Two distributed operations → Disk I/O ⬆ → Long running time!

Reference: Connected Components in MapReduce and Beyond

Proposed Method
: UniStar

UniStar: A Unified Star-operation

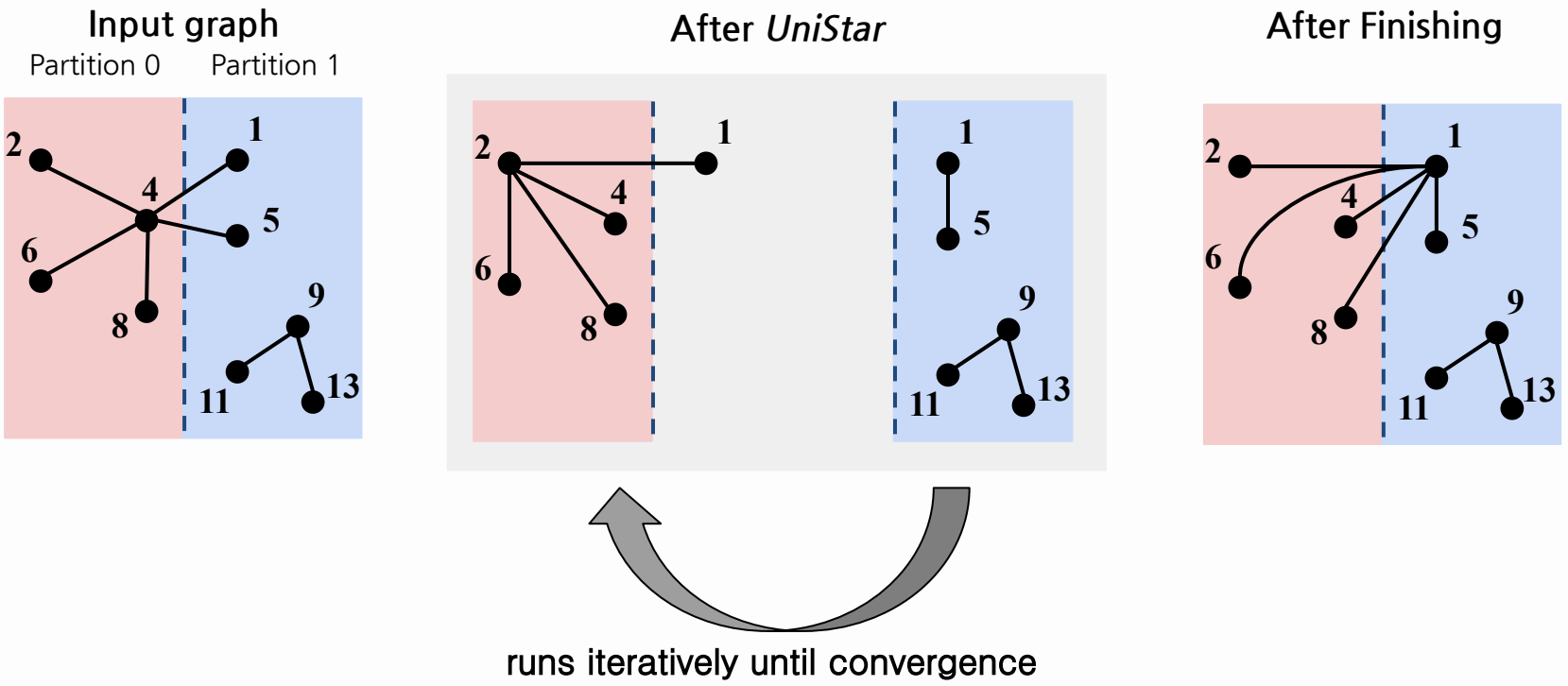
Unifies two distributed operations into one while avoiding the data explosion problem by

1. Partition-aware processing

2. Excluding “intact edge”

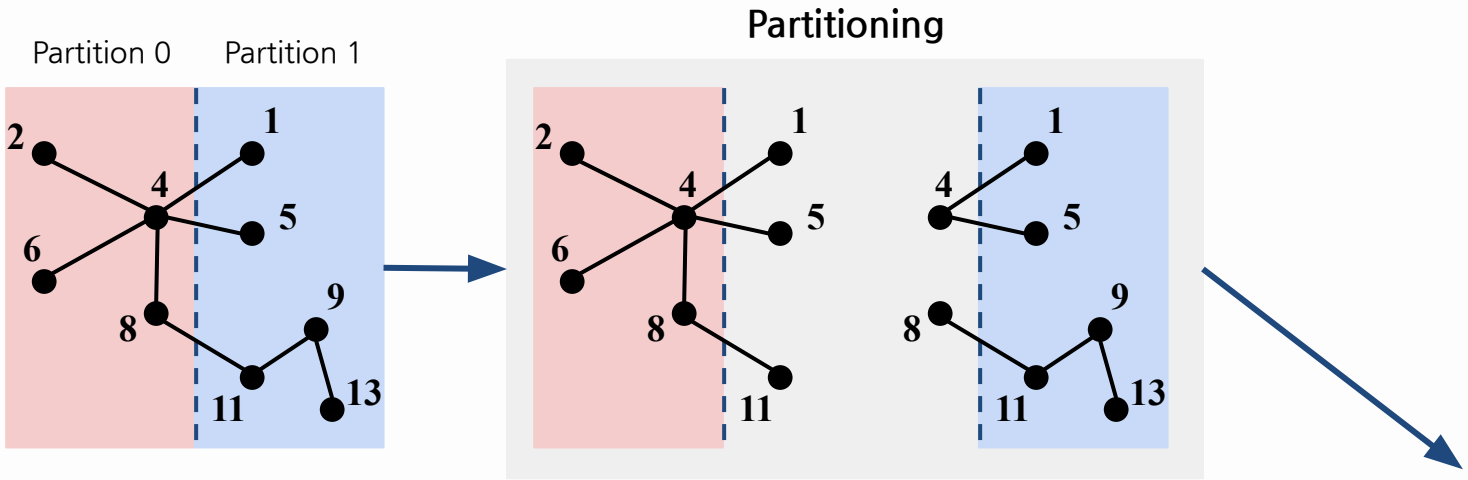
Algorithm overview

consists of 2 steps; UniStar and Finishing

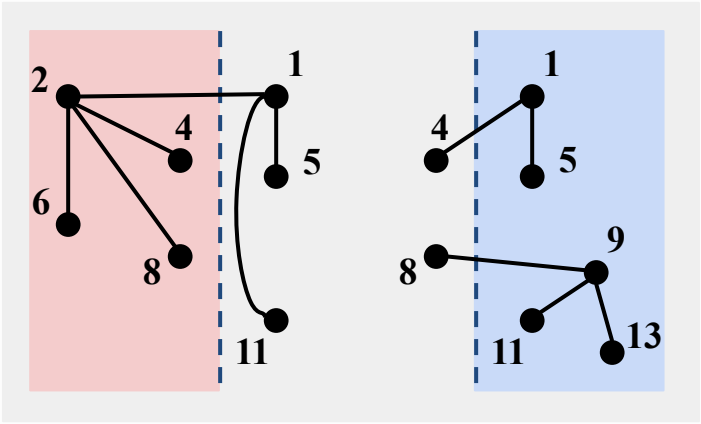


Partition-aware Processing

Partitions the nodes by a modular function and handles the nodes in one partition and their incident edges together on the same machine.



Union-Find for each subgraph



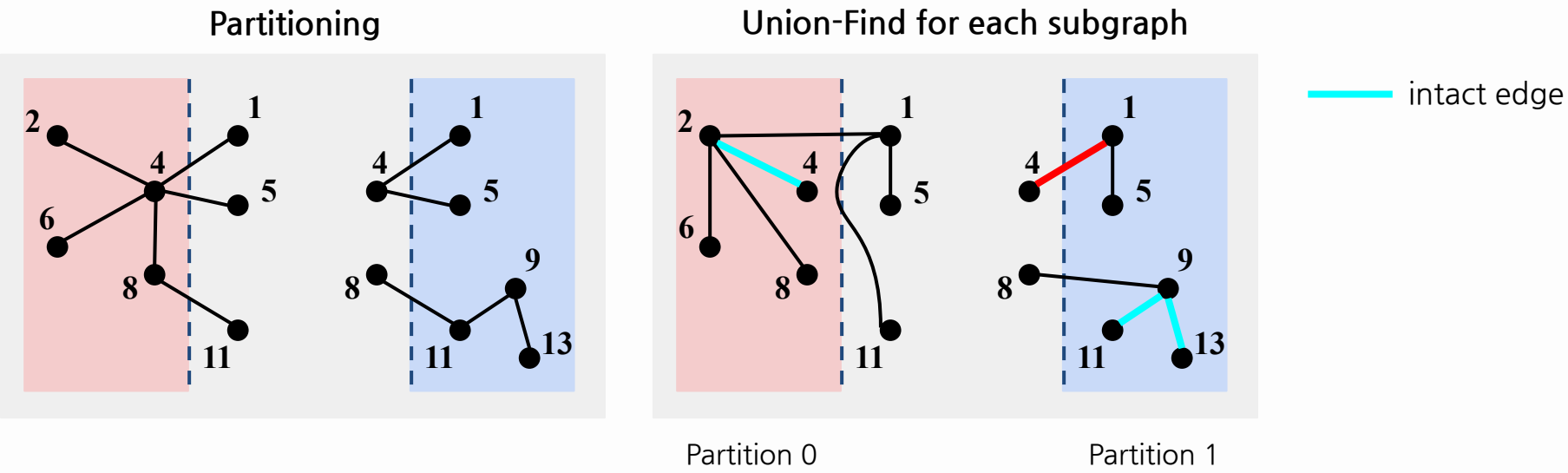
Advantages of Partition-aware Processing:

1. Reduces #edges by removing duplicated edges made in each partition.

2. Accelerates convergence by providing opportunities for each node to jump to near root node through the edges in each partition.

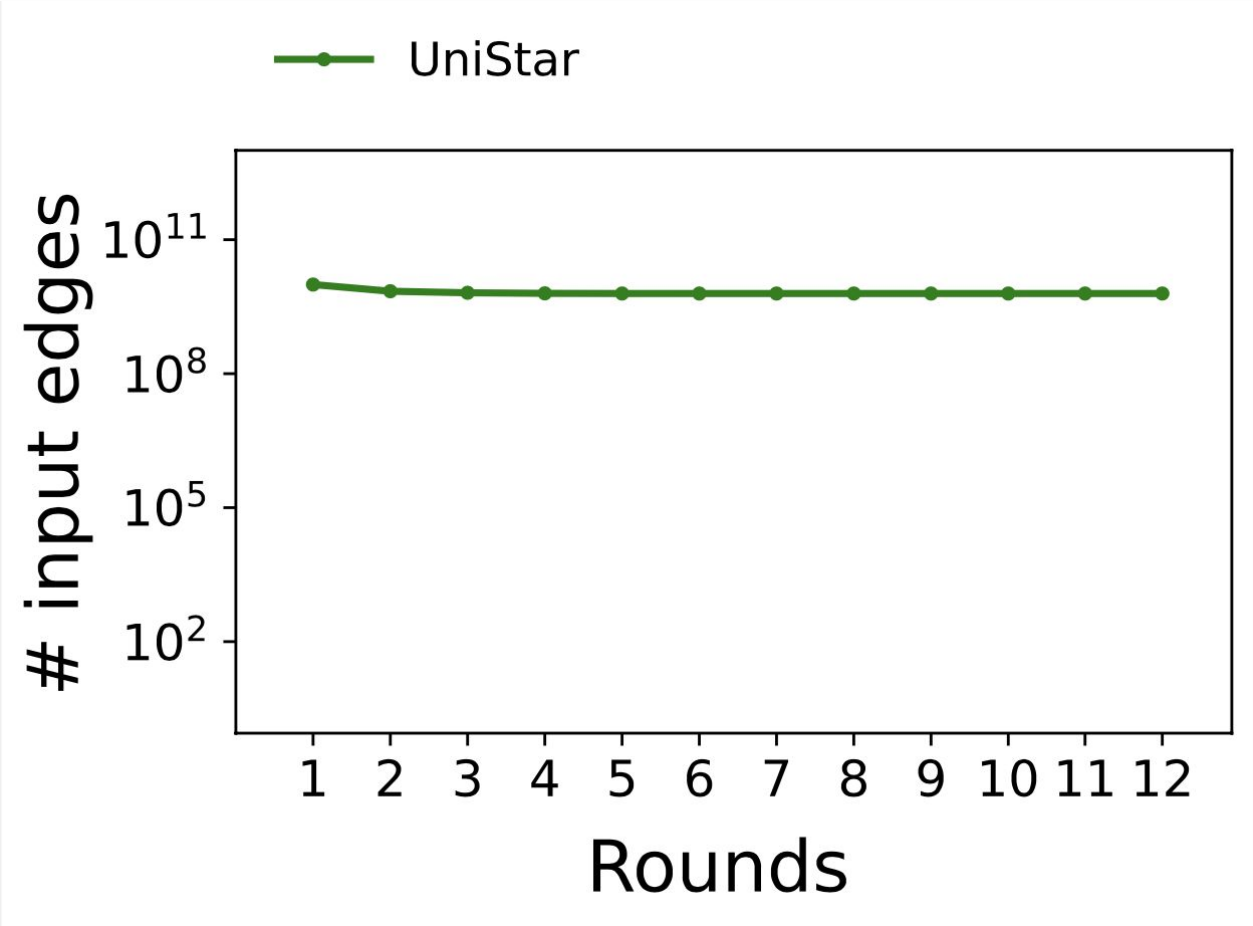
Excluding Intact Edge

We say an edge is *intact* if the edge has not changed when



UniStar-opt: Edge filtering

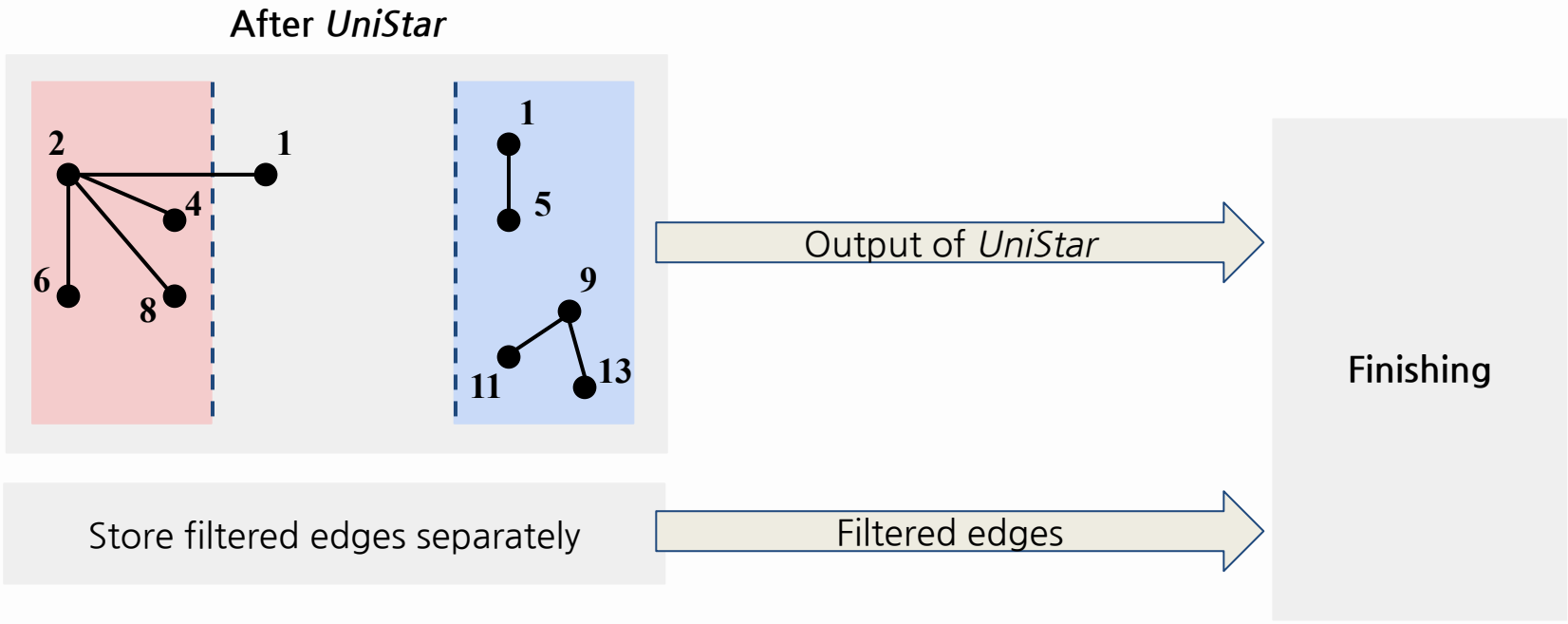
UniStar, which is not accompanied by edge filtering, takes a huge amount of input every round



Optimized UniStar
: UniStar-opt

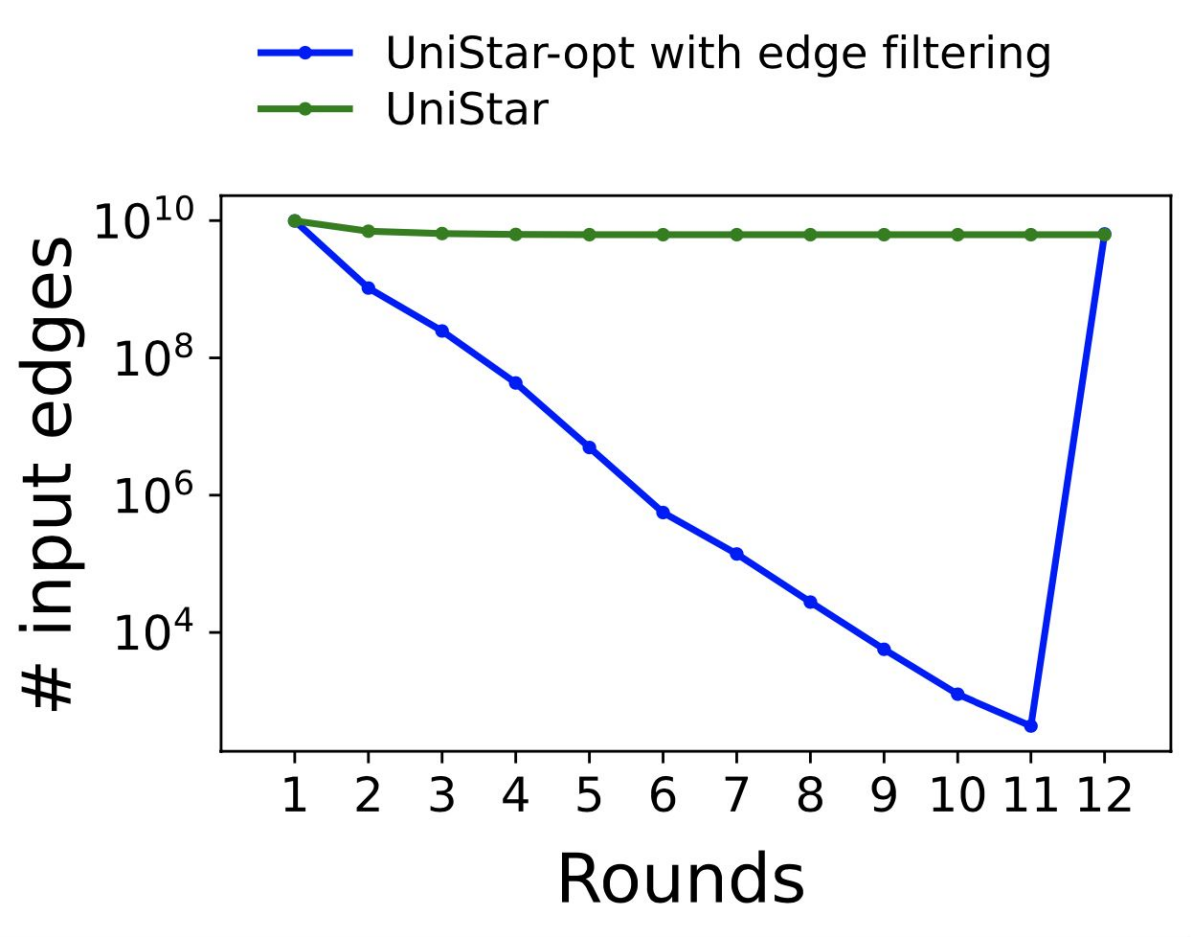
UniStar-opt: Edge filtering

We design TWO types of edges that no longer contribute to updating the graph after several rounds and filter them out



UniStar-opt: Edge filtering

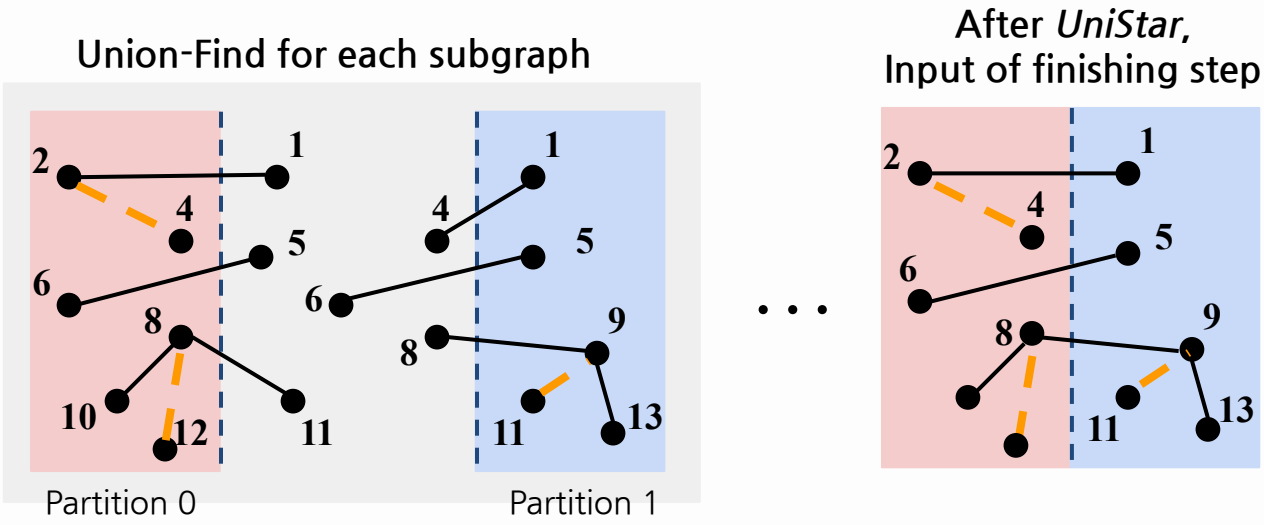
The edge filtering of UniStar-opt decreases the input size rapidly



UniStar-opt: Edge filtering

We design two types of edges that no longer contribute to updating the graph after several rounds and filter them out

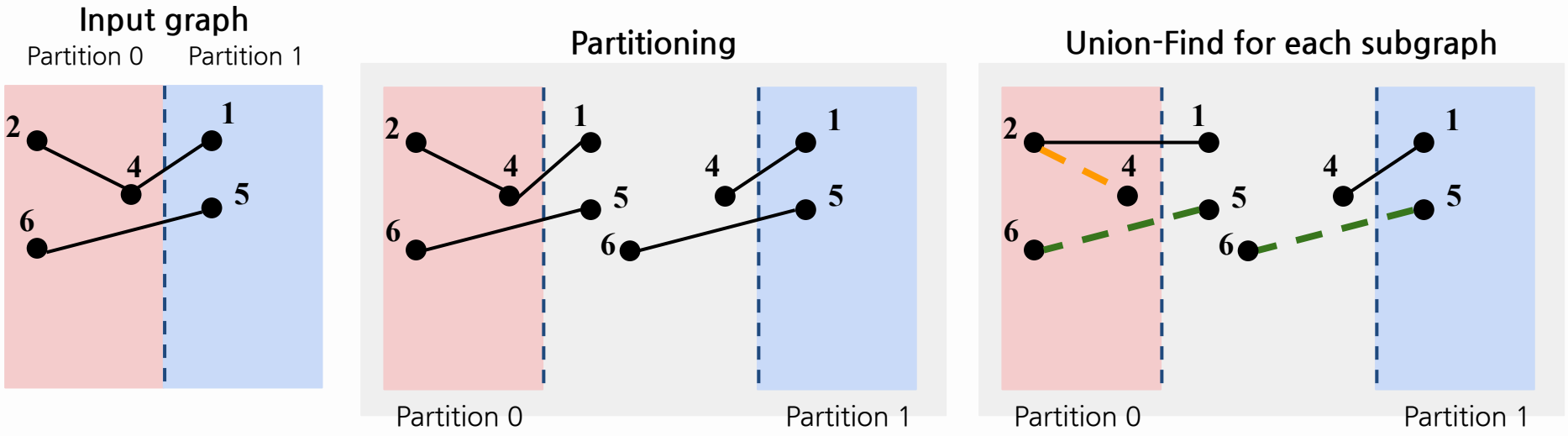
1. edge (u, v) belongs to one partition entirely



UniStar-opt: Edge filtering

We design two types of edges that no longer contribute to updating the graph after several rounds and filter them out

2. edge (u, v) that belongs to already found connected components

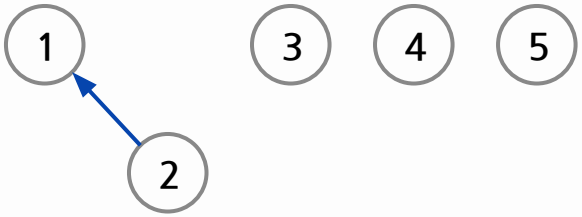


Data Structure on Union-Find

(1) Initialization: P[i] = i

i	1	2	3	4	5
P[i]	1	2	3	4	5

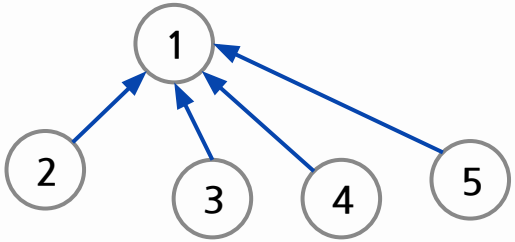
(2) union(1, 2) → find(1)=1, find(2)=2 → P[2] = 1



i	1	2	3	4	5
P[i]	1	1	3	4	5

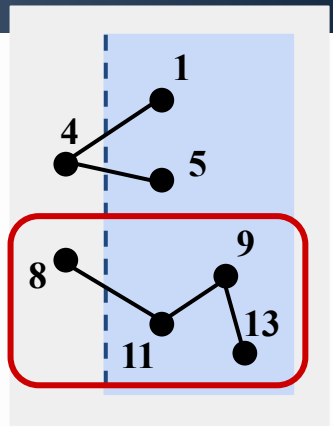
•
•
•

(6) find(*n*) for all nodes *n*



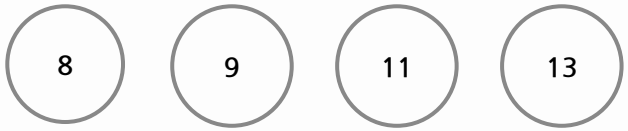
i	1	2	3	4	5
P[i]	1	1	1	1	1

Data Structure on Union-Find



When an array is used for storing parent information..

(1) Initialization: P[i] = i

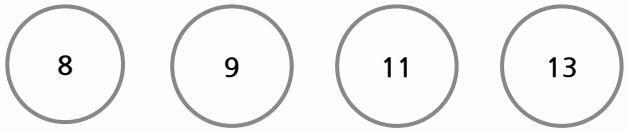


i	1	2	3	4	5	6	7	8	9	10	11	12	13
P[i]	1	2	3	4	5	6	7	8	9	10	11	12	13

→ requires too much memory space and can cause out-of-memory error

When a hash table is used for storing parent information..

(1) Initialization: P[i] = i



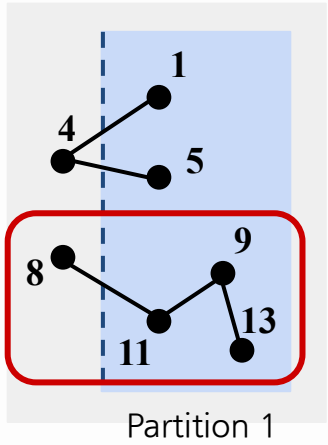
(key)	i	8	9	11	13
(value)	P[i]	8	9	11	13

→ accessing values by key from a hash table is 10 - 100 times slower than accessing values by index from an array

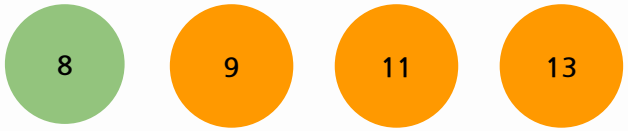
A Hybrid Map Data Structure

A new hybrid data structure that takes advantage of both an array and a hash table

Consider there are two partitions and in partition 1:



(1) Initialization: P[i] = i



Array

i	1	3	5	7	9	11	13
P[i]	1	3	5	7	9	11	13

Hash table

(key)	i	8
(value)	P[i]	8

Experiments

Settings

Datasets

real-world graphs

Dataset	# nodes	# edges
LiveJournal	4,847,571	68,993,773
Twitter	41,652,230	1,468,365,182
Friendster	65,608,366	1,806,067,135
SubDomain	89,247,739	2,043,203,933
GSH-2015	988,490,691	33,877,399,152
ClueWeb12	6,257,706,595	71,746,553,402

synthetic graphs

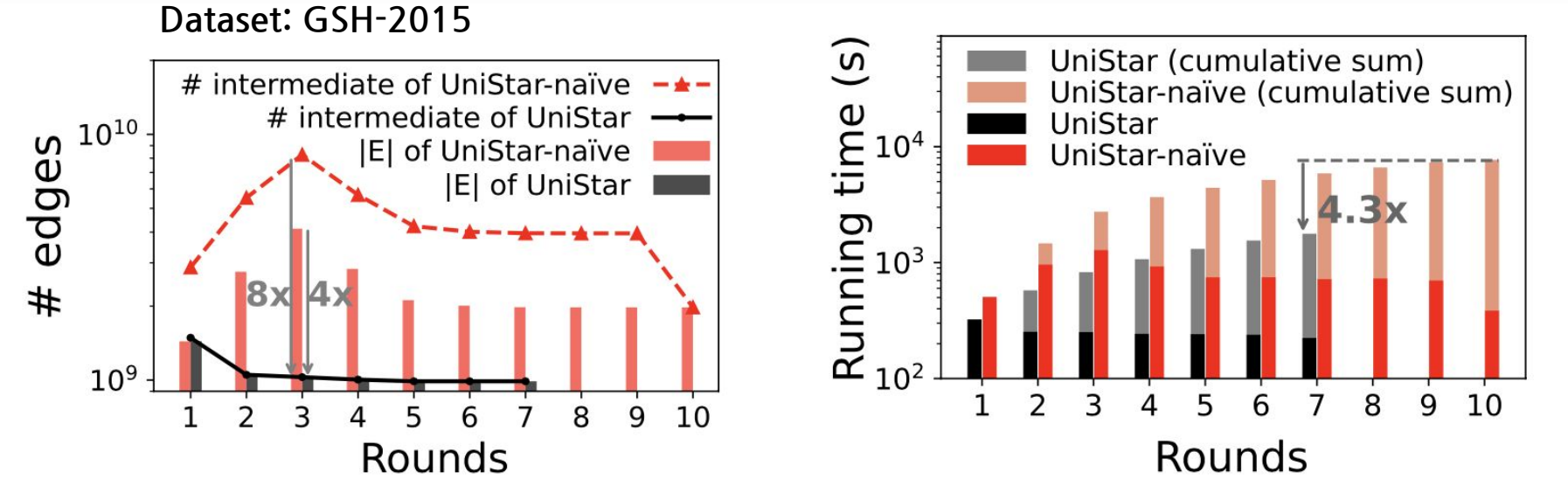
Dataset	# nodes	# edges
RMAT-21	1,114,816	31,457,280
RMAT-23	4,120,785	125,829,120
RMAT-25	15,212,447	503,316,480
RMAT-27	56,102,002	2,013,265,920
RMAT-29	207,010,037	8,053,063,680
RMAT-31	762,829,446	32,212,254,720
RMAT-33	1,090,562,291	128,849,018,880

Machines

- A cluster server (10 machines)
- CPU: Intel Xeon E3-1220 CPU (4-cores at 3.10GHz)
- RAM: 16GB
- DISK: 2 SSDs of 1TB

Efficacy of UniStar

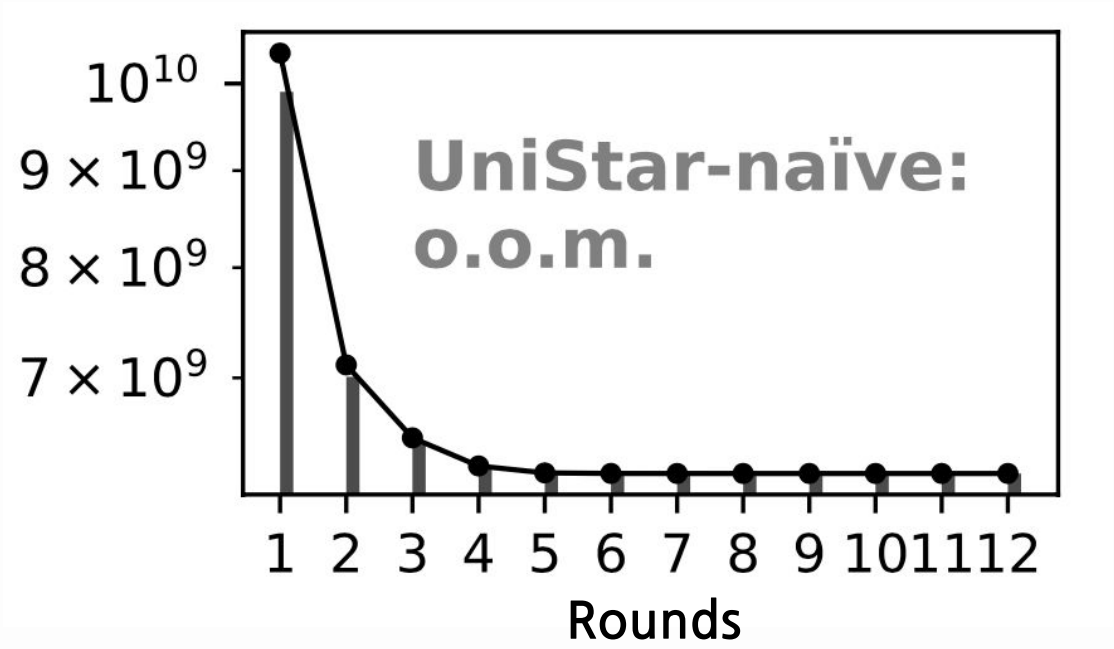
How much intermediate data does UniStar reduce to resolve the data explosion problem?



Efficacy of UniStar

How much intermediate data does UniStar reduce to resolve the data explosion problem?

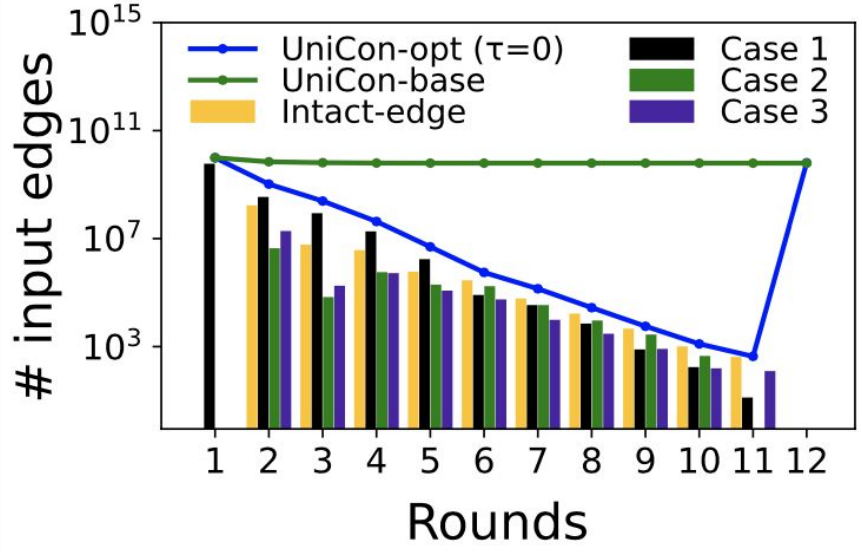
Dataset: ClueWeb12

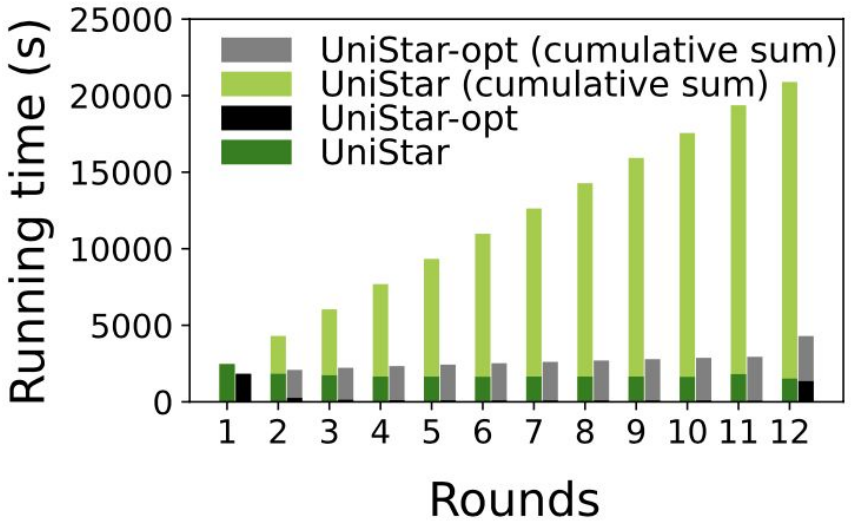


Efficacy of Edge filtering

How many edges are filtered by the edge filtering of UniStar-opt?

Dataset: ClueWeb12

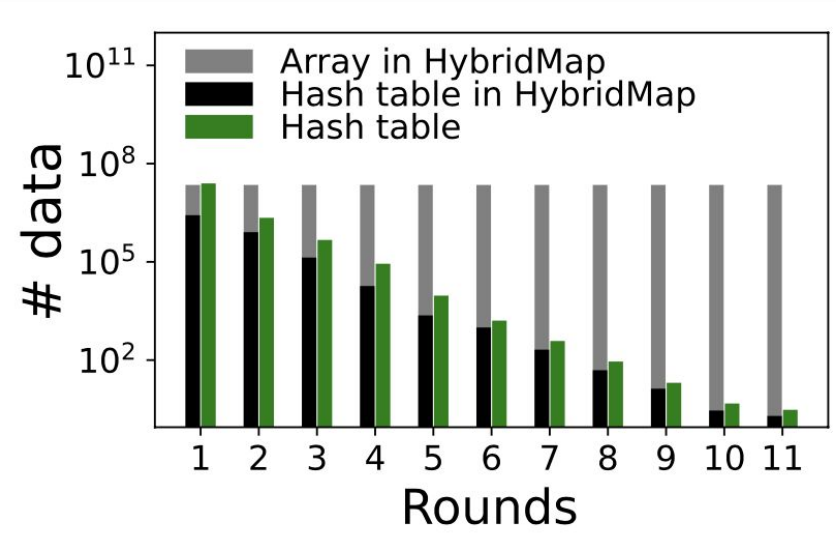


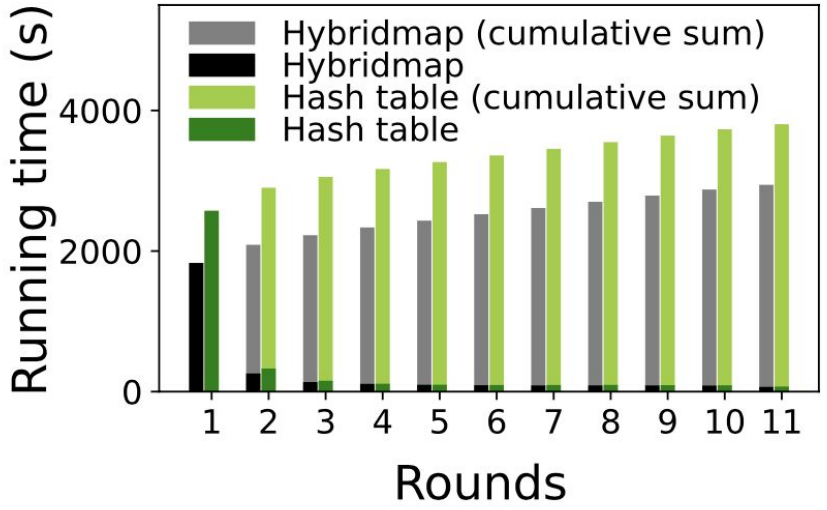


Efficacy of HybridMap

How efficient is UniStar using HybridMap compared to using an array or a hash table?

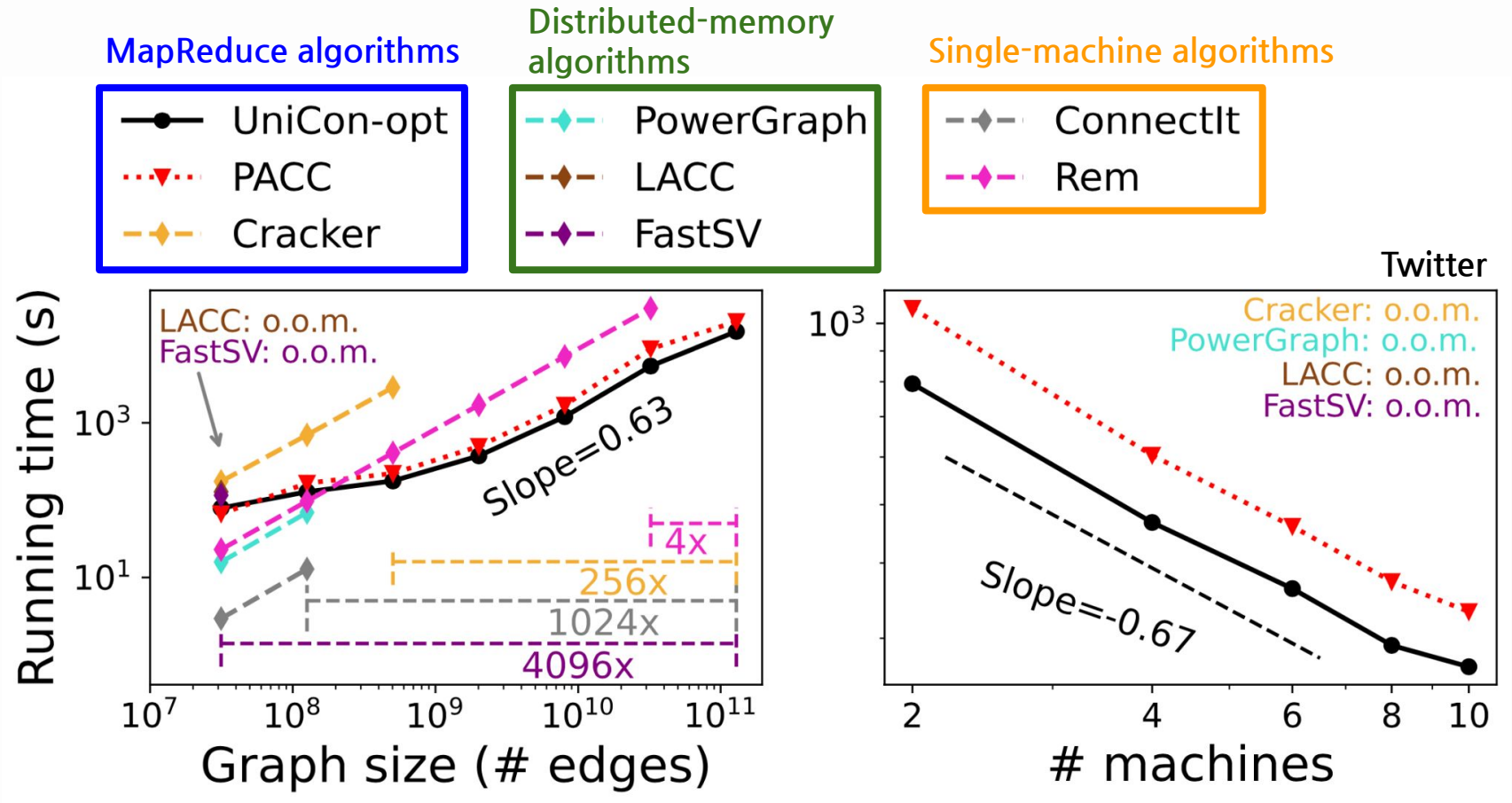
Dataset: ClueWeb12





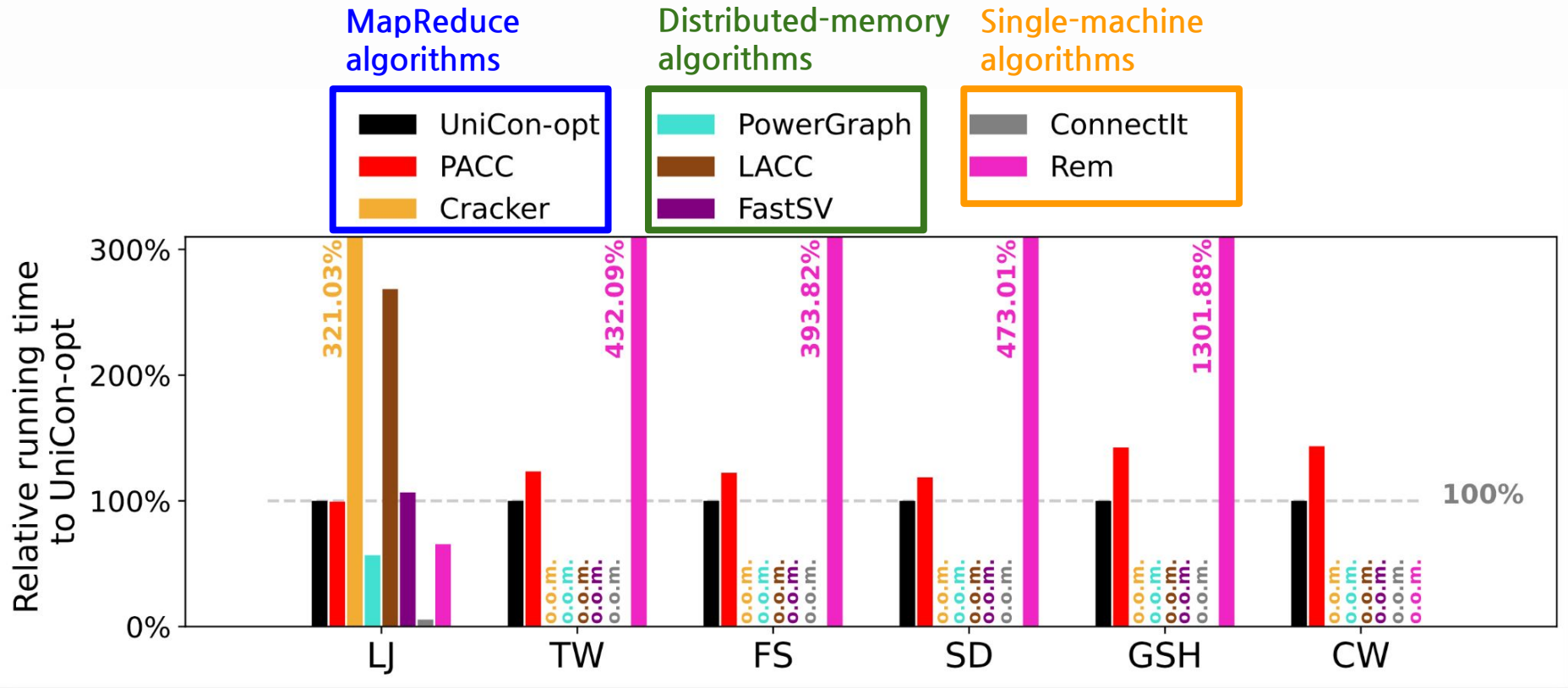
Scalability

How does our method scale up in terms of the number of machines and the data size?



Results on real-world datasets

How well does our method perform on real-world graphs compared to existing algorithms?



Conclusion

- Propose a unified star-operation, **UniStar** to minimize the number of distributed operations
- By **partition-aware processing** and **excluding intact edges**, we avoid the data explosion problem.
- **Edge filtering** of UniStar-opt shrinks the size of input data by **80.4%** on average for each round.
- The **HybridMap** data structure improves performance by **22.7%** over when using a typical hash table.

Conclusion

With a cluster of only **10 cheap machines**, we succeed in processing a graph containing **129 billion edges**, showing the **fastest** performance.

BTS: Load-Balanced Distributed Union-Find for Finding Connected Components with Balanced Tree Structures

Topic Overview

Topic Overview & Problem Definition

Implements Union-Find algorithm on a distributed-memory systems efficiently to find connected components.

Union-Find

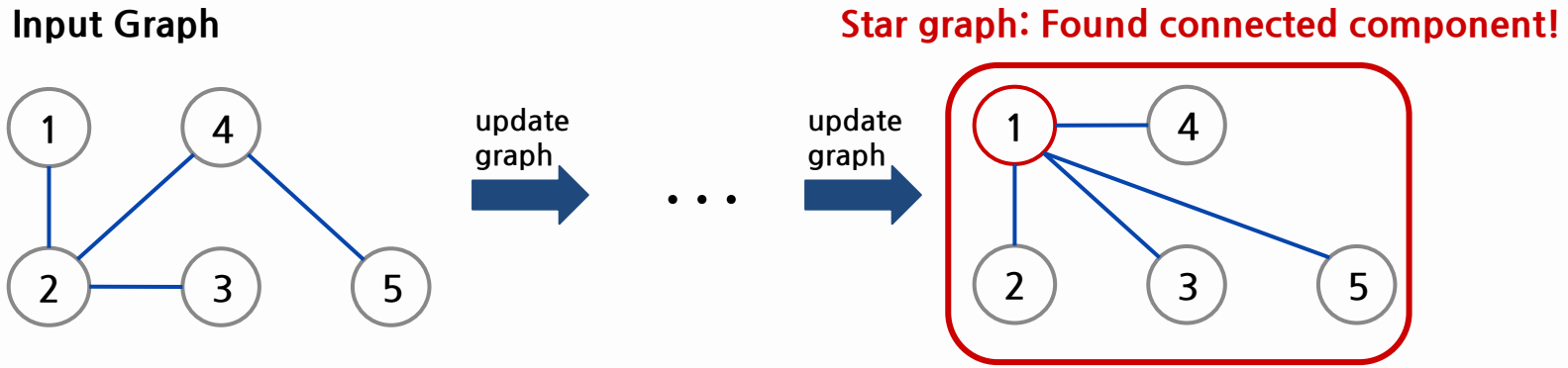
distributed-memory system

A new fast and scalable distributed Union-Find algorithm

Background & Related work

Connected components

Finding connected components



Existing connected component algorithms

TABLE I: Table of connected components algorithms

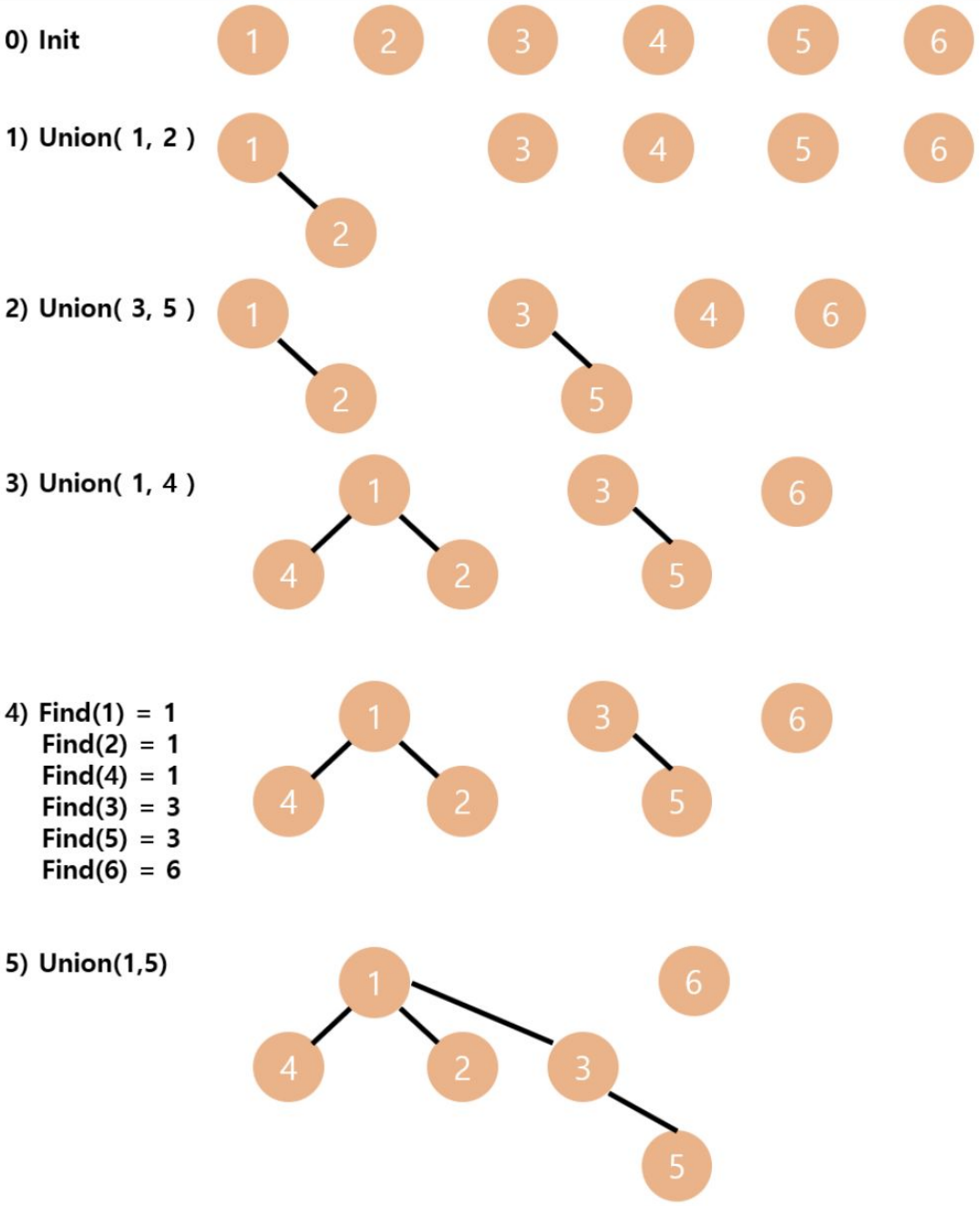
	Union-Find	Graph Traversal	Label Propagation & Shortcutting	Matrix-Vector Multiplication
Parallel, in-memory	Cybenko et al. [23], Anderson & Woll [27], Simsiri et al. [28], **ConnectIt** [29]	Bader & Cong [9], Pearce et al. [10]	Shiloach & Vishkin [13]	LACC [31], FastSV [21], [32]
Parallel, External	Agarwal et al. [30]	Pearce et al. [10]	FlashGraph [14], **Mosaic** [15]	-
Distributed, in-memory	UFM [23], DUF [24], **D-Rem** [25], ALBUF [26], **BTS (our)**	Jain et al. [11]	**D-Galois** [16]	LACC [31], **FastSV** [21], [32]
Distributed, external	-	Asokan [12]	Hash-to-Min [20], Kiveris et al. [17], **PACC** [18], [19]	Pegasus [22]

Union-Find

Union-Find is the most efficient sequential algorithm for finding connected components with low memory usage and high speed.

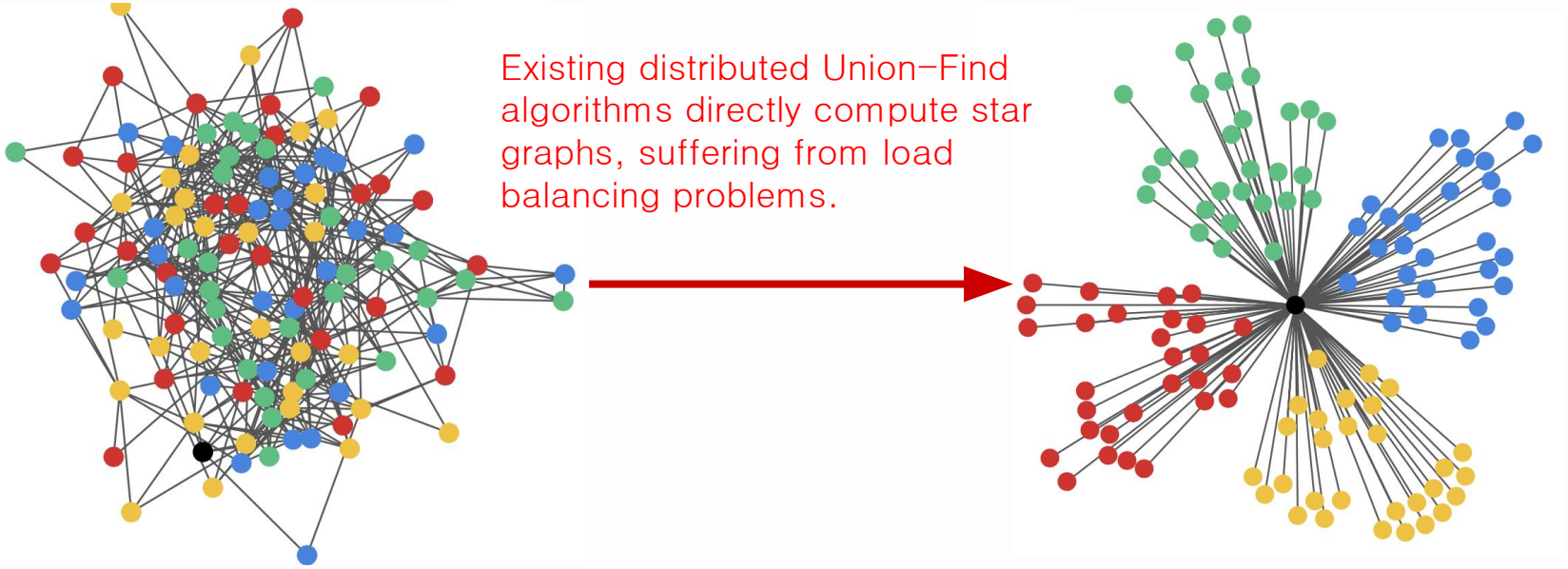
Union-Find uses two operations, *union* and *find*.

Union-Find


0) Init
1
2
3
4
5
6
1) Union(1, 2)
1
2
3
4
5
6
2) Union(3, 5)
1
2
3
5
4
6
3) Union(1, 4)
1
4
2
3
5
6
4) Find(1) = 1
Find(2) = 1
Find(4) = 1
Find(3) = 3
Find(5) = 3
Find(6) = 6
1
4
2
3
5
6
5) Union(1,5)
1
4
2
3
5
6

Load Balancing Problem

Existing vertex partitioning-based Union-Find algorithms suffer from load balancing problems due to the nature of Union-Find that gathers edges to a small number of vertices as it proceeds.



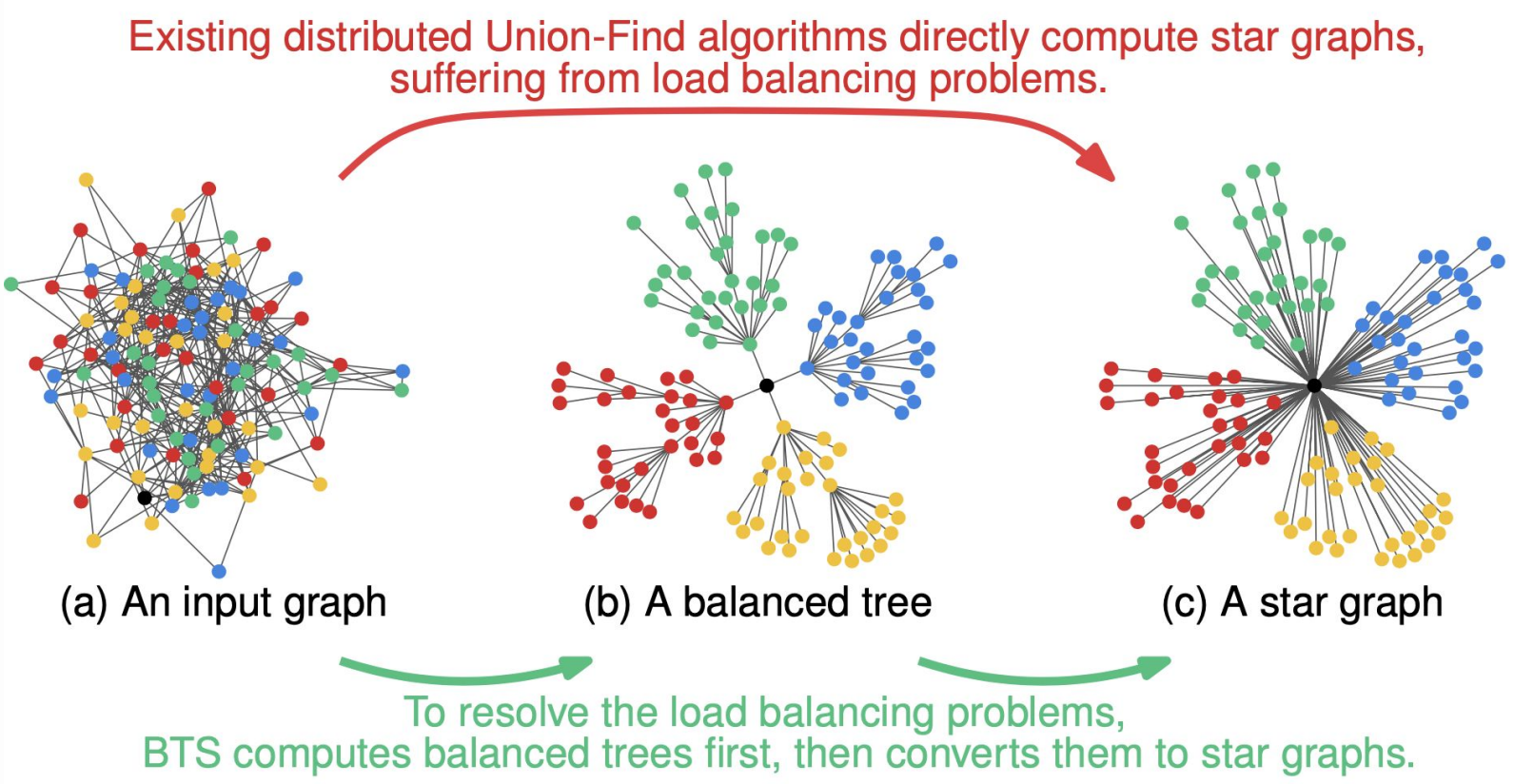
(a) An input graph

(b) A star graph

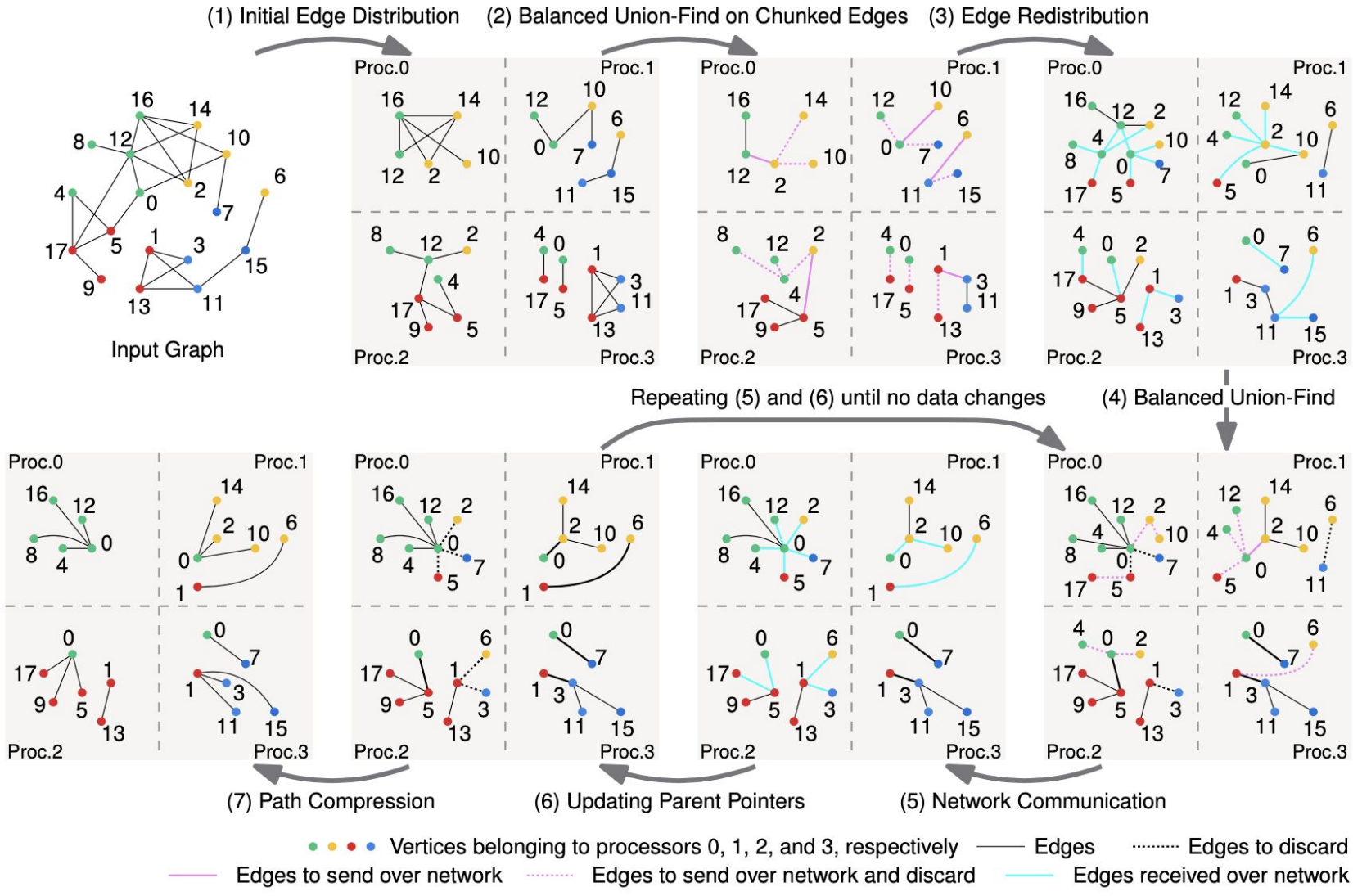
Proposed Method: BTS

BTS: Union-Find with Balanced Tree Structure

By *edge rebalancing*, which balances the workload by keeping each edge inside a processor, BTS also reduces network traffic and memory usage as the number of edges spanning between processors decreases.



Overall Process of BTS

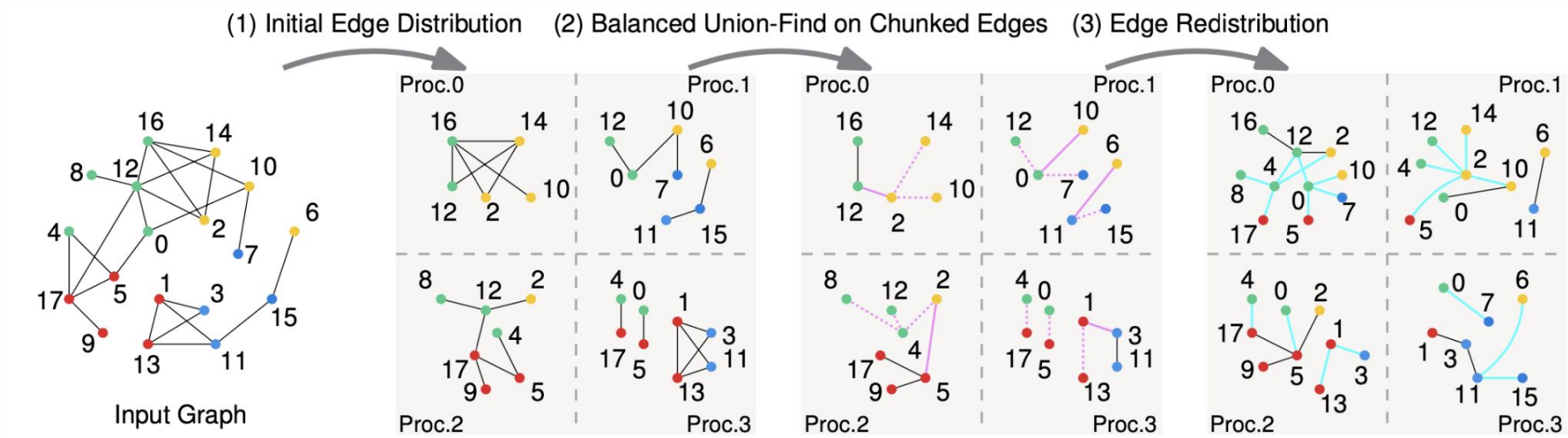


Initial Edge Distribution & Redistribution

Process (1)-(3)
(1) BTS partitions the vertex set into blocks, assigns each block to a processor, and has each processor repeatedly update the parent pointers of given vertices.

(3) Then, each processor redistributes each edge (u,v) to processors ξ(u) and ξ(v) to let them update their parent pointers with the edge.

(2) Before Redistributing, BTS uses *Balanced Union-Find (BUF)*.

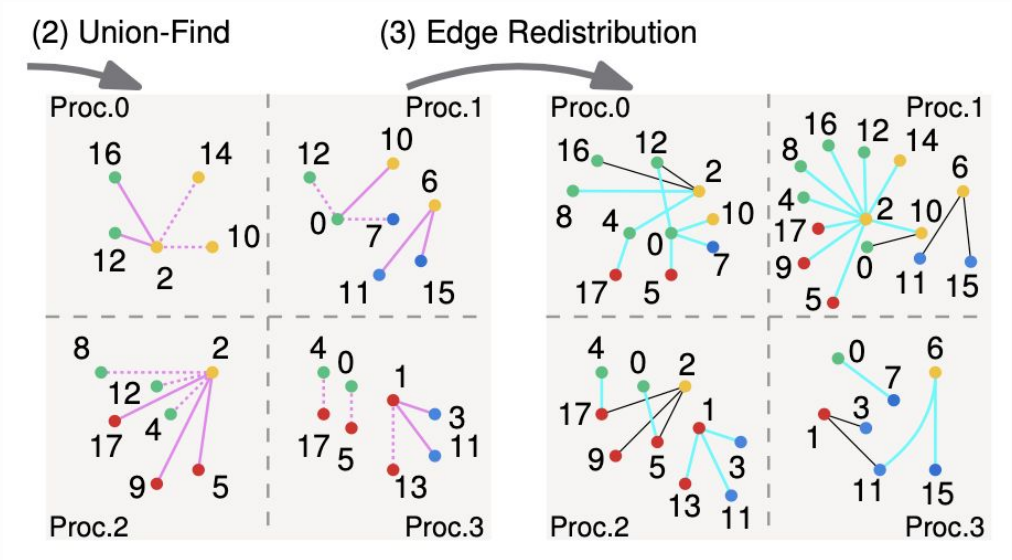


Balanced Union-Find (BUF)

BTS resolves the load balancing problems by introducing Balanced Union-Find (BUF). After Union-Find, BUF adds a rebalancing operation, which modifies parent pointers not to focus on the same vertex.

For each star graph, BUF modifies the parent pointer of each vertex to point to the local root, which is the most preceding vertex among vertices with the same color, and the parent pointers of local root vertices to point to the root vertex.

Balanced Union-Find (BUF)



An example of edge redistribution with plain Union-Find.

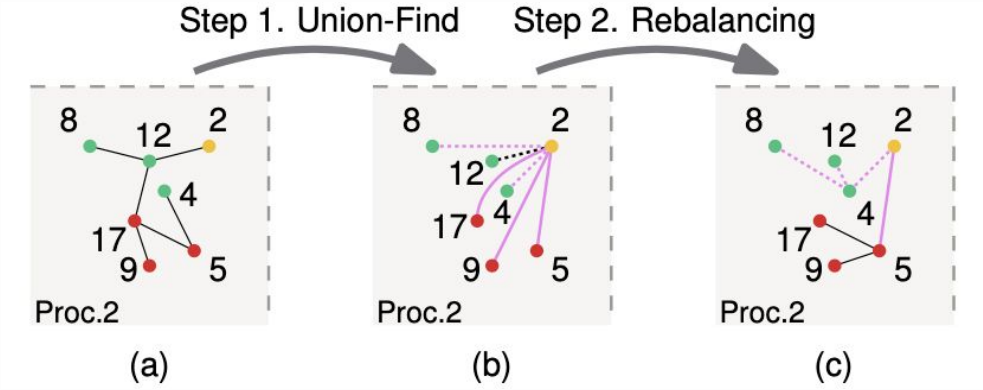
Edges are concentrated on vertex 2, causing a load balancing problem.

☹️

An example of Balanced Union-Find.

BUF prevents edges from being concentrated on a few vertices.

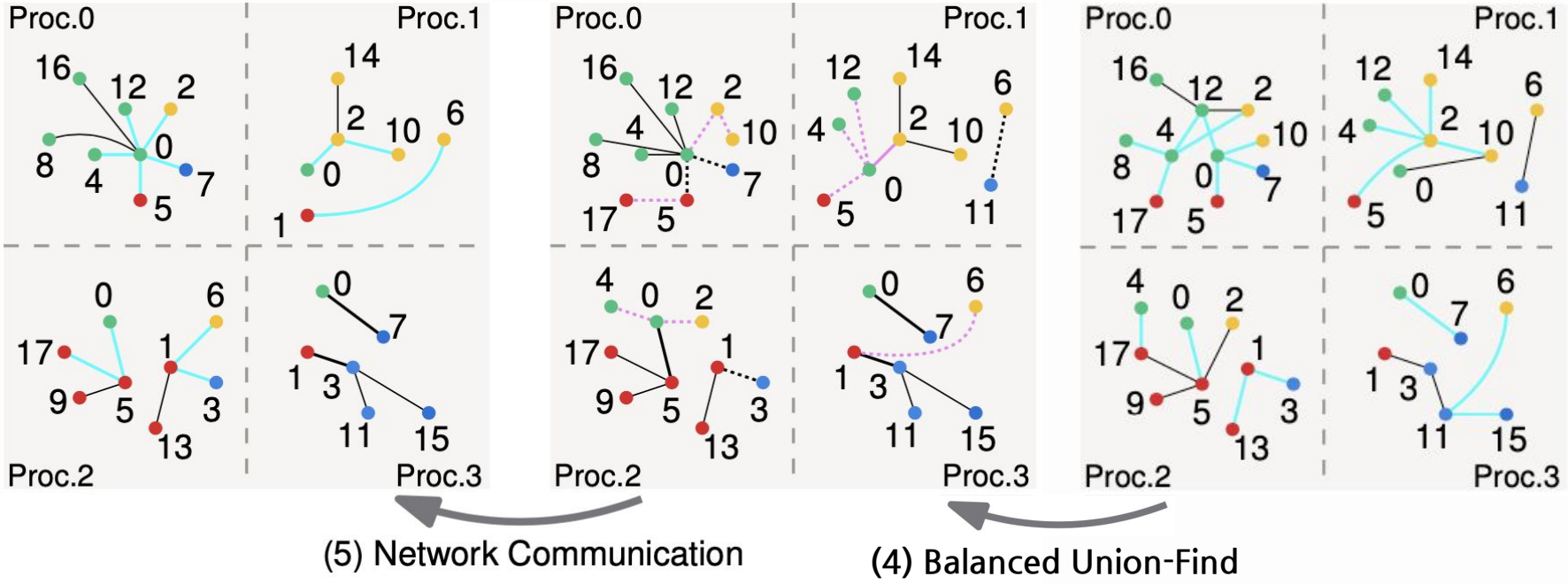
😄



Optimization: Network Communication

1) Excluding unchanged edges from network communication: BTS reduces network traffic by excluding edges not changed by BUF from network communication if the edge's source is not a local root.

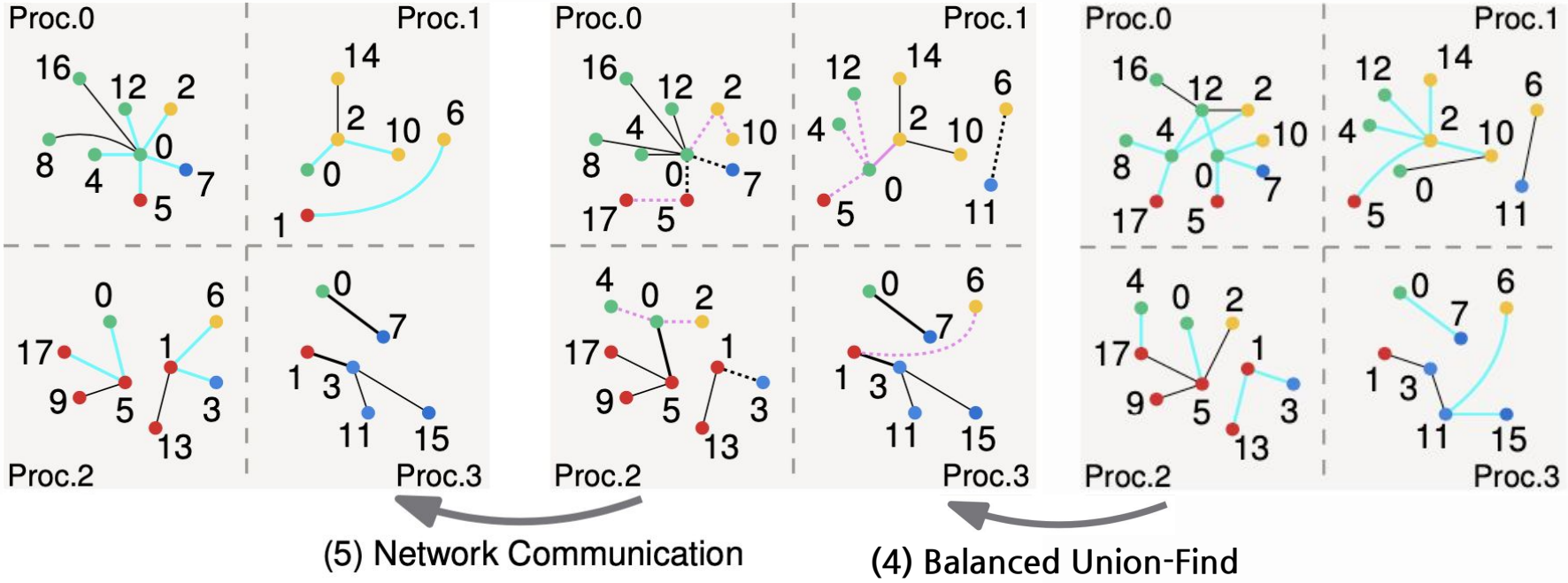
For example, outer edges (5,0) and (7,0) in processor 0 do not change in step (4); thus, they are not sent over the network in step (5).



Optimization: Network Communication

2) Discarding parent pointers of outer vertices: As soon as the network communication ends, BTS discards all the parent pointers of outer vertices to secure the memory space.

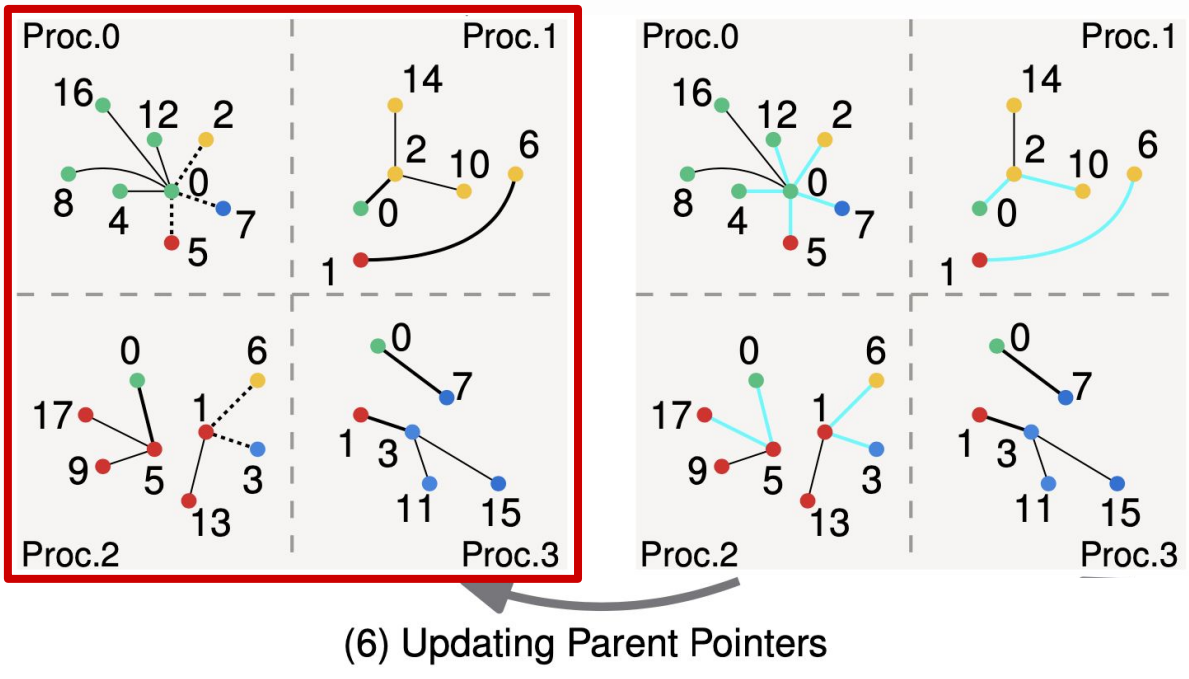
For example, in step (5), edge (6,1) discarded in processor 3 is sent to processors 1 and 2, and edge (5,0) discarded in processor 0 also exists in processor 2.



Iteration Process & Finalization

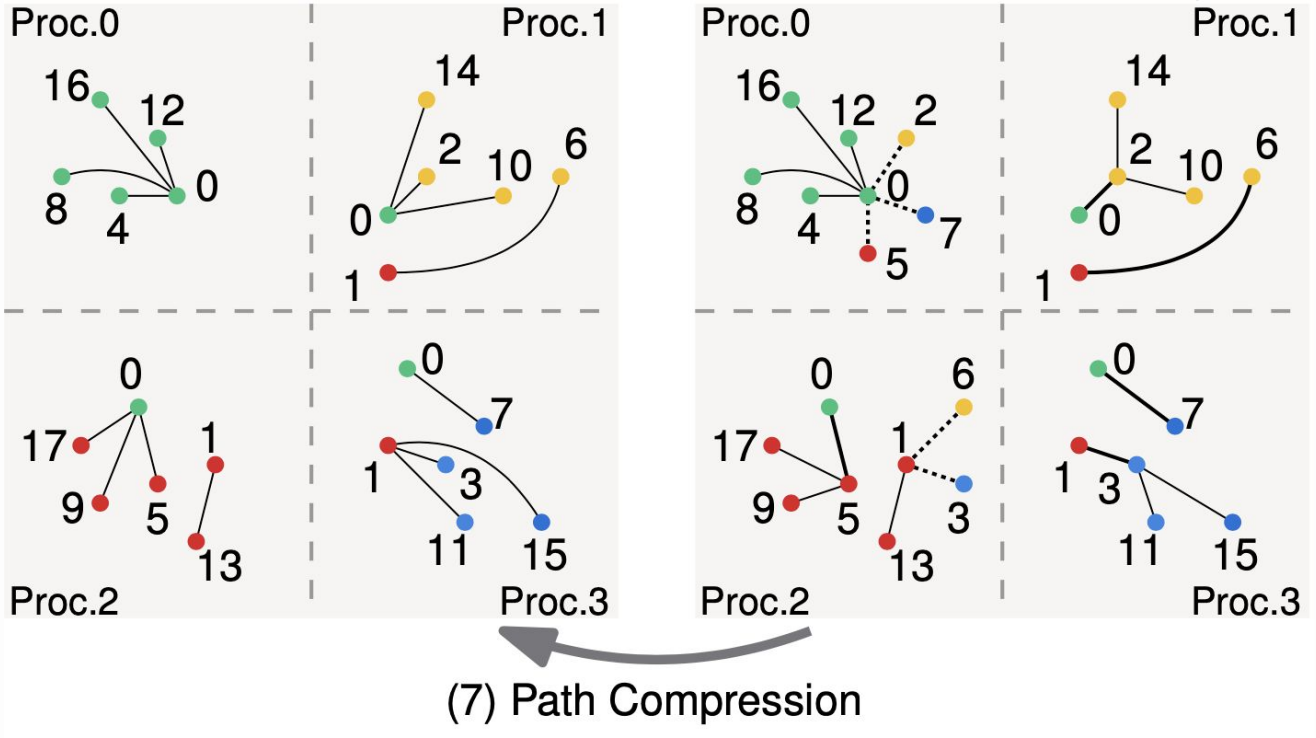
Iteration Process: If updated edges exist, they are sent again over the network. This process repeats until no more edges are updated. When the iteration process ends, all vertices belonging to each processor form star graphs where the centers are local roots, and the star graphs include the root vertices.

Every subgraph forms star graphs!!



Iteration Process & Finalization

Finally, each processor updates the parent pointers to point to the root vertices without network communication.



Experiments

Settings

Datasets

real-world graphs

Dataset	# nodes	# edges
LiveJournal	4,847,571	68,993,773
Twitter	41,652,230	1,468,365,182
Friendster	65,608,366	1,806,067,135
SubDomain	89,247,739	2,043,203,933
GSH-2015	988,490,691	33,877,399,152

synthetic graphs

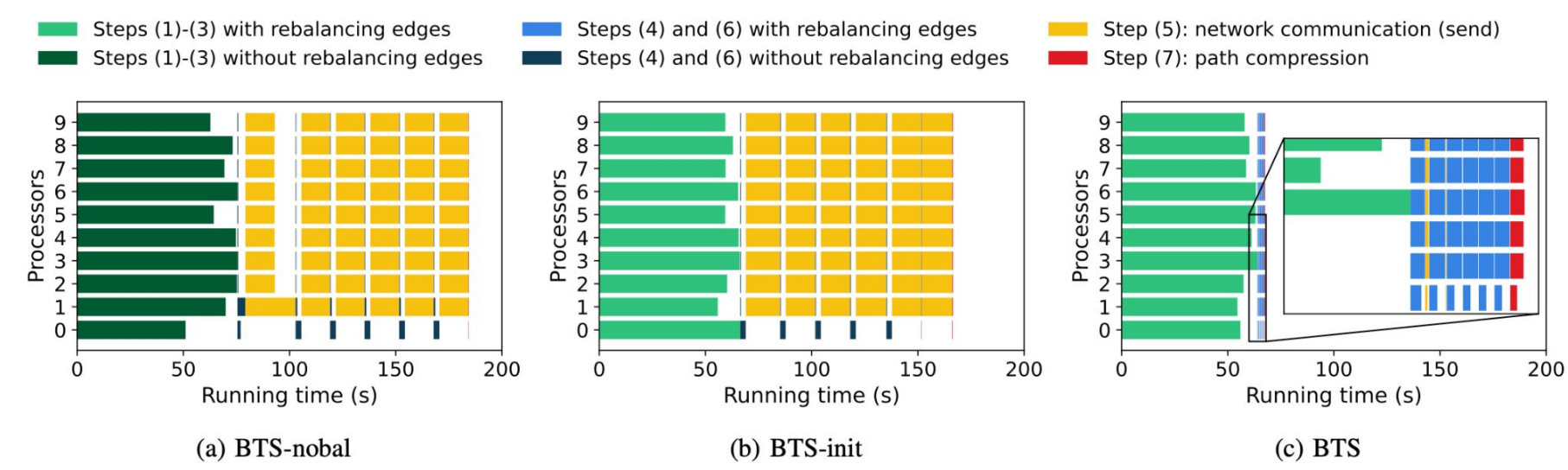
Dataset	# nodes	# edges
RMAT-21	1,114,816	31,457,280
RMAT-23	4,120,785	125,829,120
RMAT-25	15,212,447	503,316,480
RMAT-27	56,102,002	2,013,265,920
RMAT-29	207,010,037	8,053,063,680
RMAT-31	762,829,446	32,212,254,720

Machines

- A cluster server (10 machines)
- CPU: Intel Xeon E3-1220 CPU (4-cores at 3.10GHz)
- RAM: 16GB
- DISK: 2 12TB HDDs

How well does BTS balance the workload?

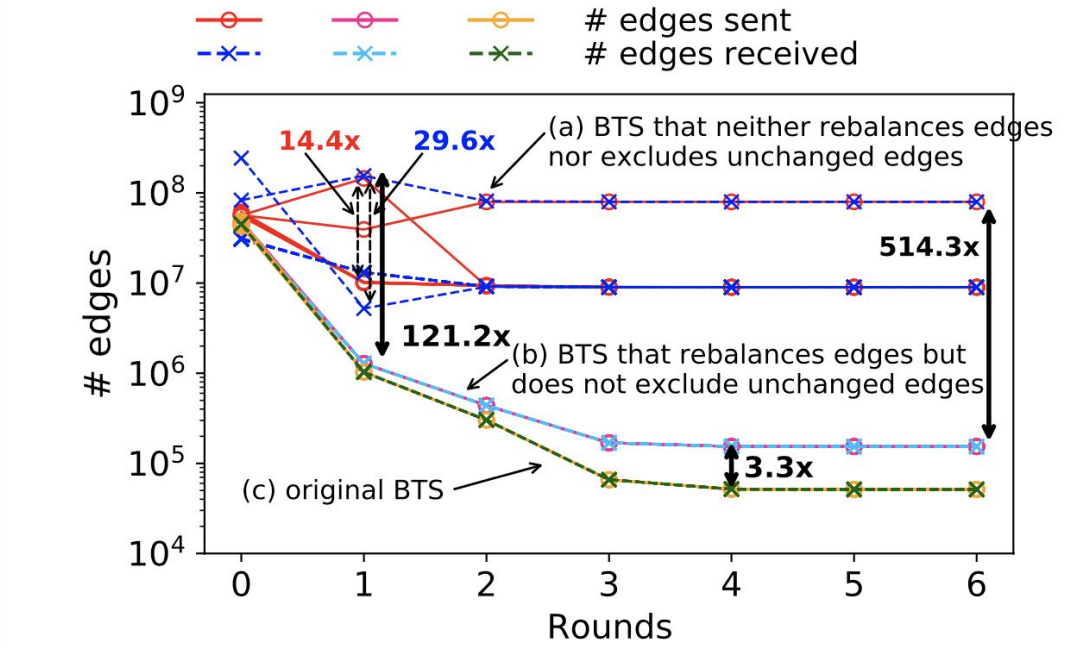
(a) BTS-nobal: BTS without rebalancing edges.
(b) BTS-init: BTS which rebalances edges only at the initial step.
(c) BTS: the original BTS which rebalances edges every round,



Gantt chart showing the running time of 10 processors in three variations of BTS.
Empty parts in bars indicate that the processors are idle or just receiving data from others. The original BTS (c) resolves load balancing problems of BTS-nobal and BTS-init, reducing the running time for network communication by 522x and 553x, respectively.

How much network traffic does BTS reduce?

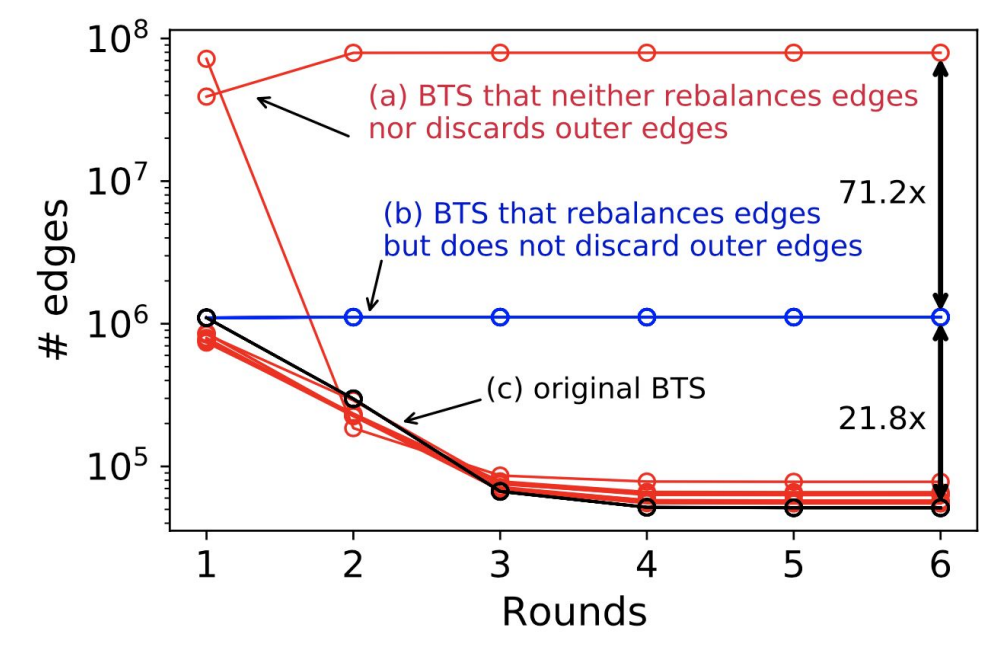
(a) BTS that neither rebalances edges nor excludes unchanged edges .
(b) BTS that rebalances edges but does not exclude unchanged edges.
(c) original BTS



The number of edges 10 processors send and receive for three methods. While (a) suffers from a load balancing problem, (b) greatly reduces the network traffic by up to 514.3× and (c) further decreases the network traffic by up to 3.3×.

How much memory space does BTS require?

(a) BTS that neither rebalances edges nor discards outer edges .
(b) BTS that rebalances edges but does not discard outer edges.
(c) original BTS



The number of outer edges for (a), (b), and (c). The original BTS reduces the number of outer edges by a factor of 1552.

Scalability

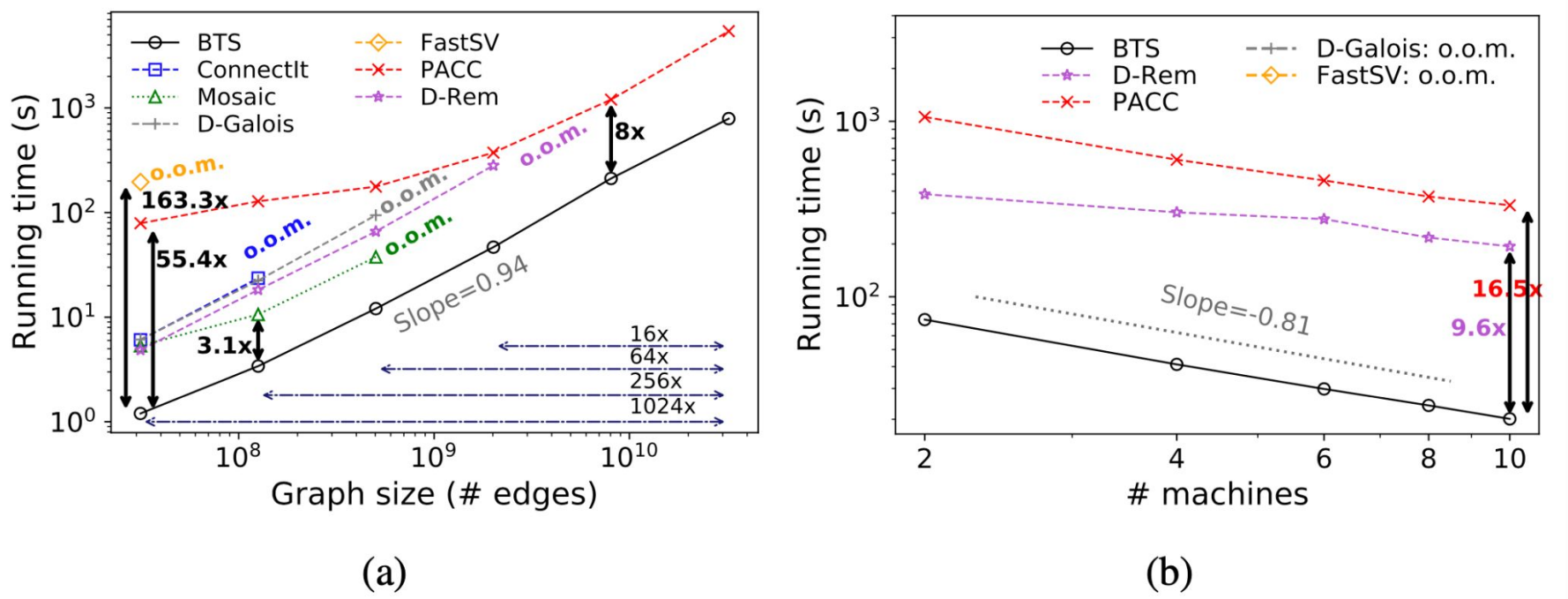
We compare BTS to the other algorithms written in bold in Table I in terms of data and machine scalability.

TABLE I: Table of connected components algorithms

	Union-Find	Graph Traversal	Label Propagation & Shortcutting	Matrix-Vector Multiplica-tion
Parallel, in-memory	Cybenko et al. [23], Anderson & Woll [27], Simsiri et al. [28], **ConnectIt** [29]	Bader & Cong [9], Pearce et al. [10]	Shiloach & Vishkin [13]	LACC [31], FastSV [21], [32]
Parallel, External	Agarwal et al. [30]	Pearce et al. [10]	FlashGraph [14], **Mosaic** [15]	-
Distributed, in-memory	UFM [23], DUF [24], **D-Rem** [25], ALBUF [26], **BTS (our)**	Jain et al. [11]	**D-Galois** [16]	LACC [31], **FastSV** [21], [32]
Distributed, external	-	Asokan [12]	Hash-to-Min [20], Kiveris et al. [17], **PACC** [18], [19]	Pegasus [22]

Scalability

We compare BTS to the other algorithms written in bold in Table I in terms of data and machine scalability.

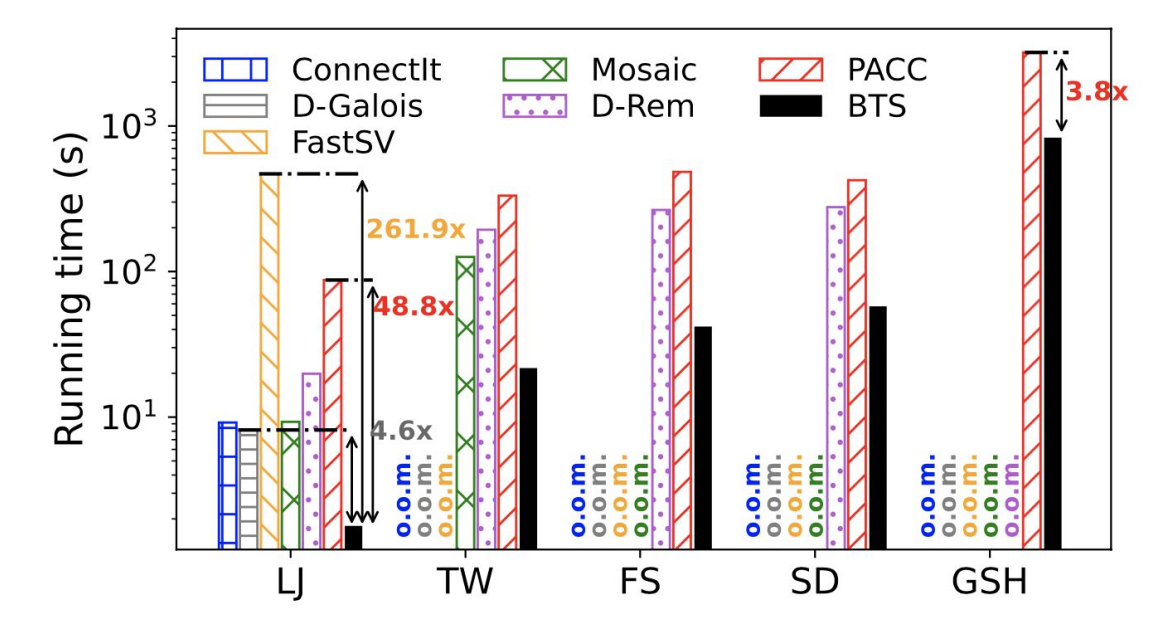


(a) Data scalability of all algorithms on various sized RMAT graphs. BTS is 3.1 to 163.3 times faster and handles 16 to 1024 times larger graphs than other algorithms.

(b) Machine scalability of all distributed algorithms. BTS is the fastest regardless of the number of machines, showing 9.6× and 16.5× faster speeds than D-Rem and PACC, respectively.

On Real-world Graphs

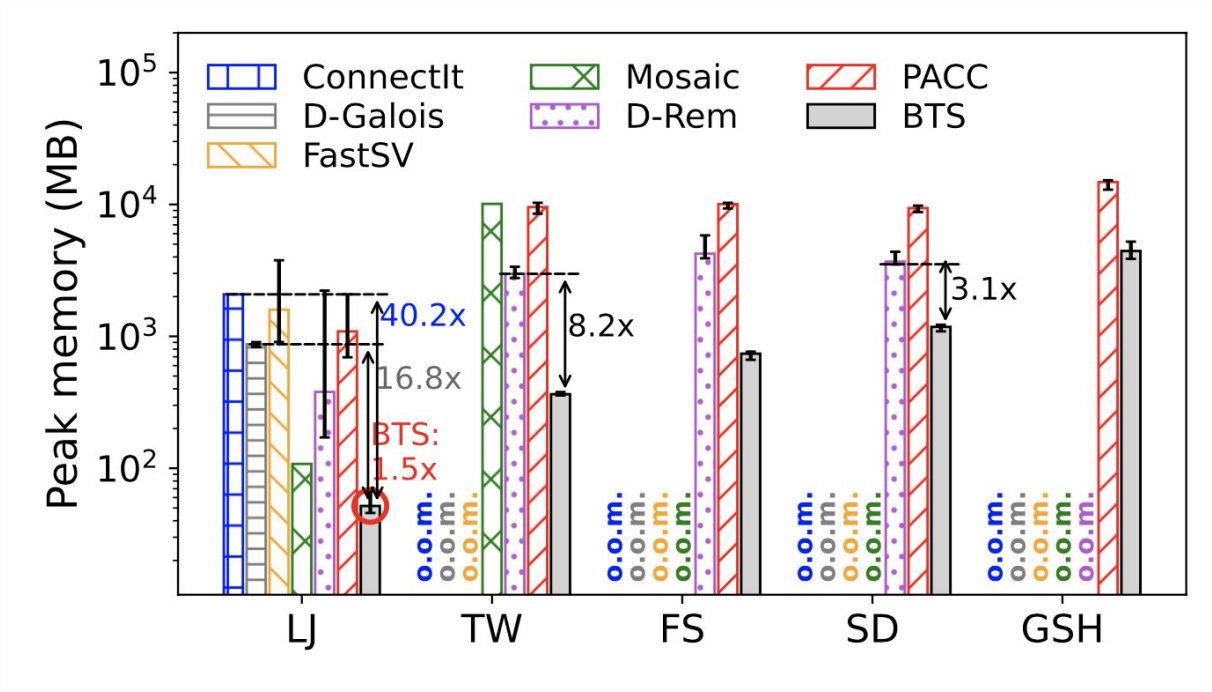
We compare BTS to the other algorithms written in bold in Table I.



The running time of all algorithms. BTS is the fastest showing up to 261.9 times faster speed.

On Real-world Graphs

We compare BTS to the other algorithms written in bold in Table I.



Box: the average peak memory usage of 10 processors.
Error bar: the maximum and minimum values.

BTS consumes the least amount of memory on all graphs.

Conclusion

- Propose **BTS**, a fast and scalable distributed Union-Find algorithm for finding connected components in large graphs.
- **Balanced Union-Find**, which rebalances edges to efficiently resolve load balancing problems, reduces the running time for network communication by **553 times** and shrinking network traffic and memory usage simultaneously by up to **514.3 and 71.2 times**, respectively.
- BTS further decreases network traffic by up to **3.3 times** by excluding unchanged edges and memory usage by up to **21.8 times** by discarding outer edges.

Conclusion

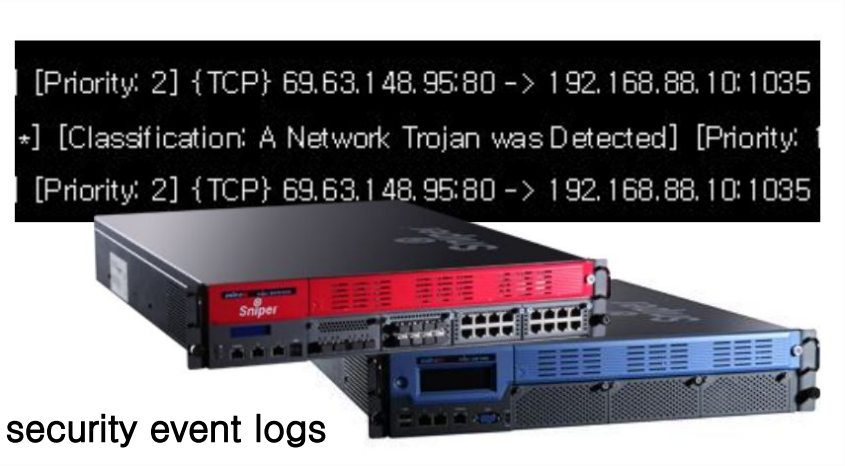
BTS outperforms the state-of-the-art algorithms by processing **16-1024 times** larger graphs with **3.1-261.9 times** faster speeds.

An Anomaly Detection System using Tensor Decomposition

Problem Definition

- Cyber attack is a sensitive issue in the world of Internet security.
- Anomaly detection systems play a crucial role in identifying malicious activities.
- Given large-sized security event logs, how can we detect anomaly detection?

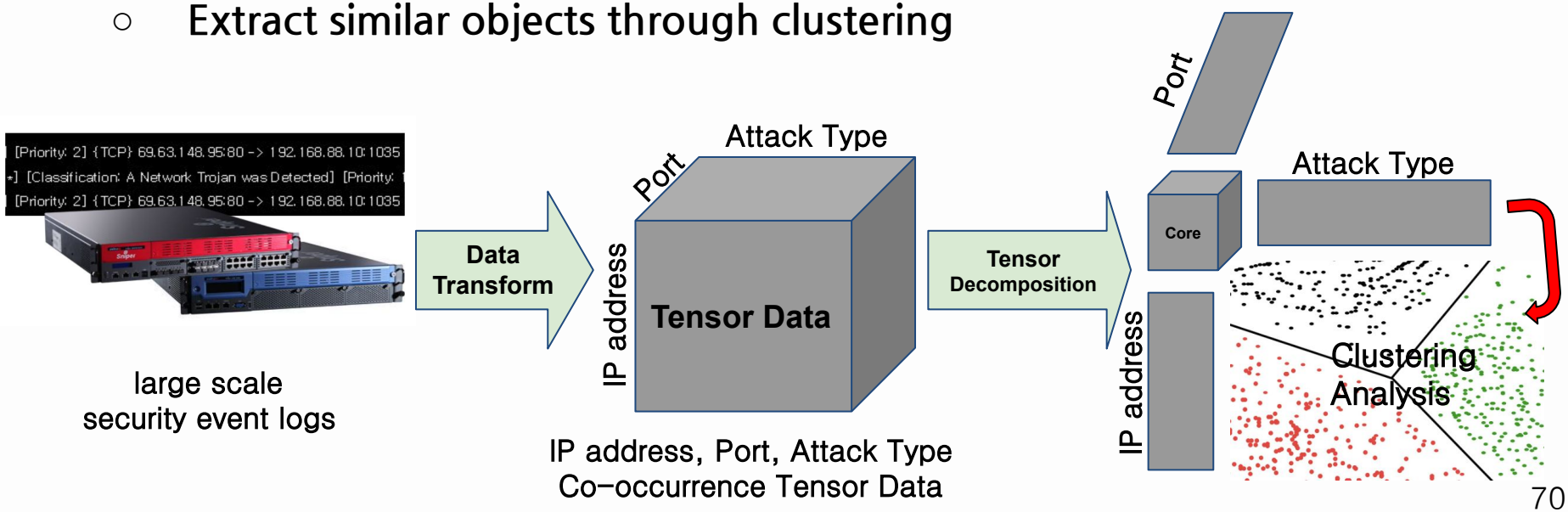
  → Tensor Decomposition!



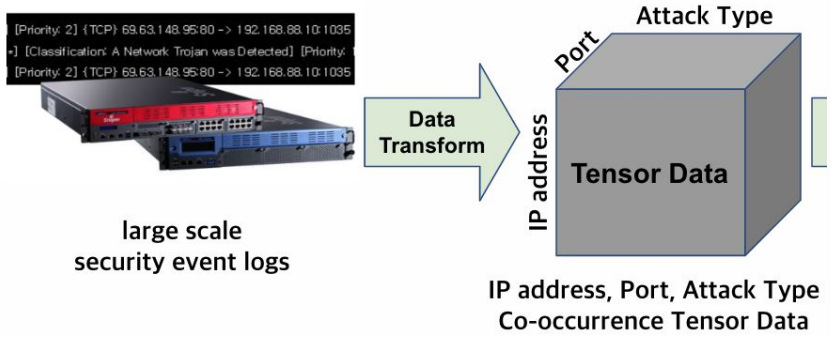
large scale security event logs

Heterogeneous Feature Relationship Analysis

- Given large-sized security event logs, analysis heterogeneous feature relationship
  - Transform relationships between heterogeneous features in secure logs to a tensor
  - feature embedding and relationship analysis by decomposing tensor
  - Extract similar objects through clustering



Data Transform



large scale
security event logs

IP address, Port, Attack Type
Co-occurrence Tensor Data

- KISTI's Security event log → Tensor Data
  - Transform to a tensor consisting 0 or 1
    - 1, if (IP, Port, EventName) has appeared together in the log
    - 0, otherwise

```
20210401032711.65272273558722   0       10.0.14.5
.41     6       443     60402   0       msn site
GFW 2차 -2      0       0       null    0       0
s1vNAAC4GqWEXO5eTCgAOKQG76/I/IbPrbmxWyFAQAfWGCQAAF
ECACMAAAAFAAAWAwMNQwsADT8ADTwAB9gwggfUMIIFvKADAgEC
wNyb3NvZnQgQ29ycG9yYXRpb24xIDAeBgNVBAMTF01pY3Jvc2
CzAJBgNVBAgTAldBMRAwDgYDVQQHEwdSZWRtb25kMR4wHAYDVQ
BAQEFAAOCAQ8AMIIBCgKCAQEA1PCSk7HxrvXSRhAkJDm89tUdr
UZwQlpwSxK+vCrEmUf6t6KGsLiu1cput+n49/sKjQ1Zs7GE/F1
```

Data Transform

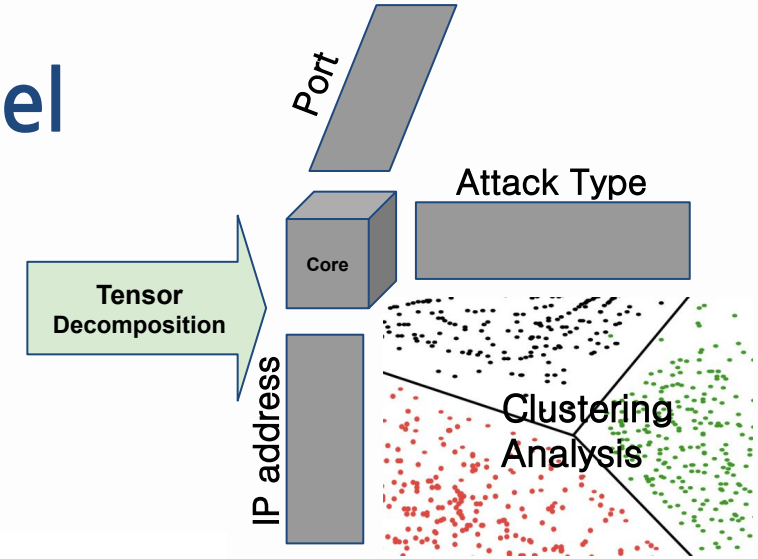
IP address	Port	EventName
203.252.27.235	443	tcp port scan - flag ack
220.81.249.216	56928	tcp port scan - flag ack
203.252.27.235	443	tcp port scan - flag ack
220.81.249.216	58130	tcp port scan - flag ack
203.252.27.235	443	tcp port scan - flag ack
1.245.3.151	61505	tcp port scan - flag ack
203.252.27.235	443	tcp port scan - flag ack
1.245.3.151	61102	tcp port scan - flag ack
203.252.27.235	443	tcp port scan - flag ack
220.81.249.216	58938	tcp port scan - flag ack

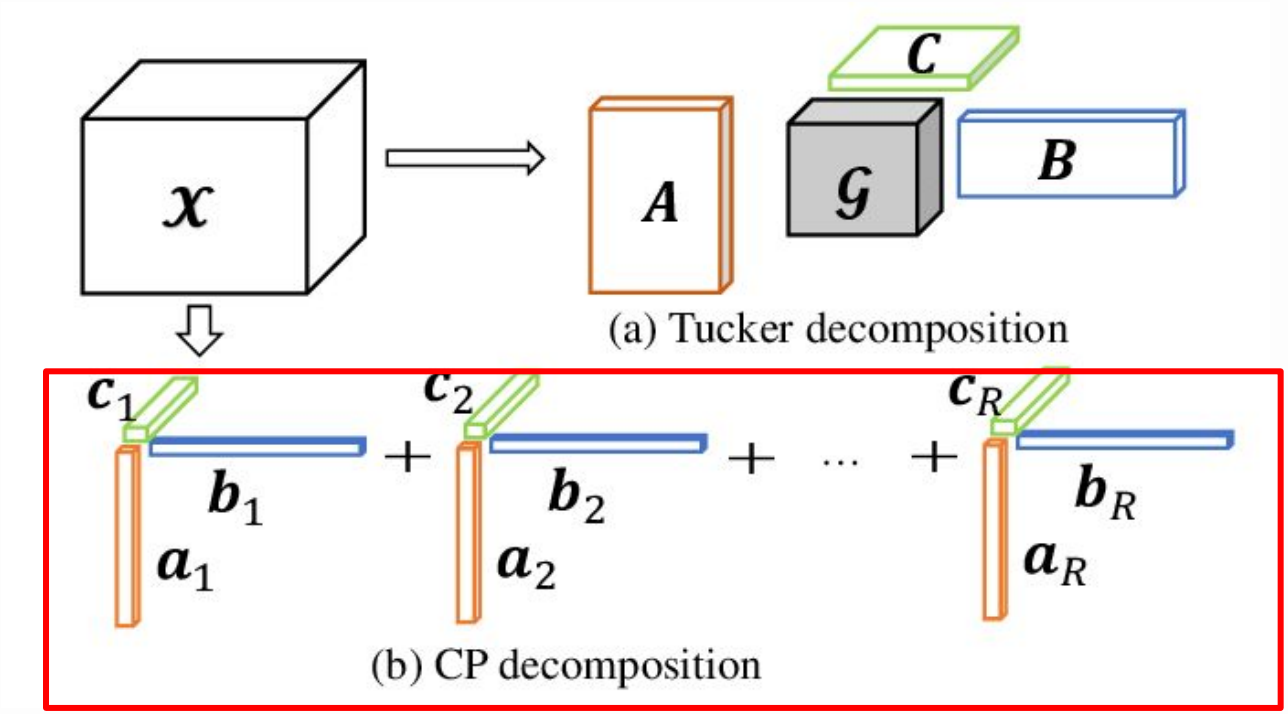
KISTI
security event logs

IP – Port – Event Tensor data

Tensor Decomposition Model

- Use CP Decomposition (PARAFAC)





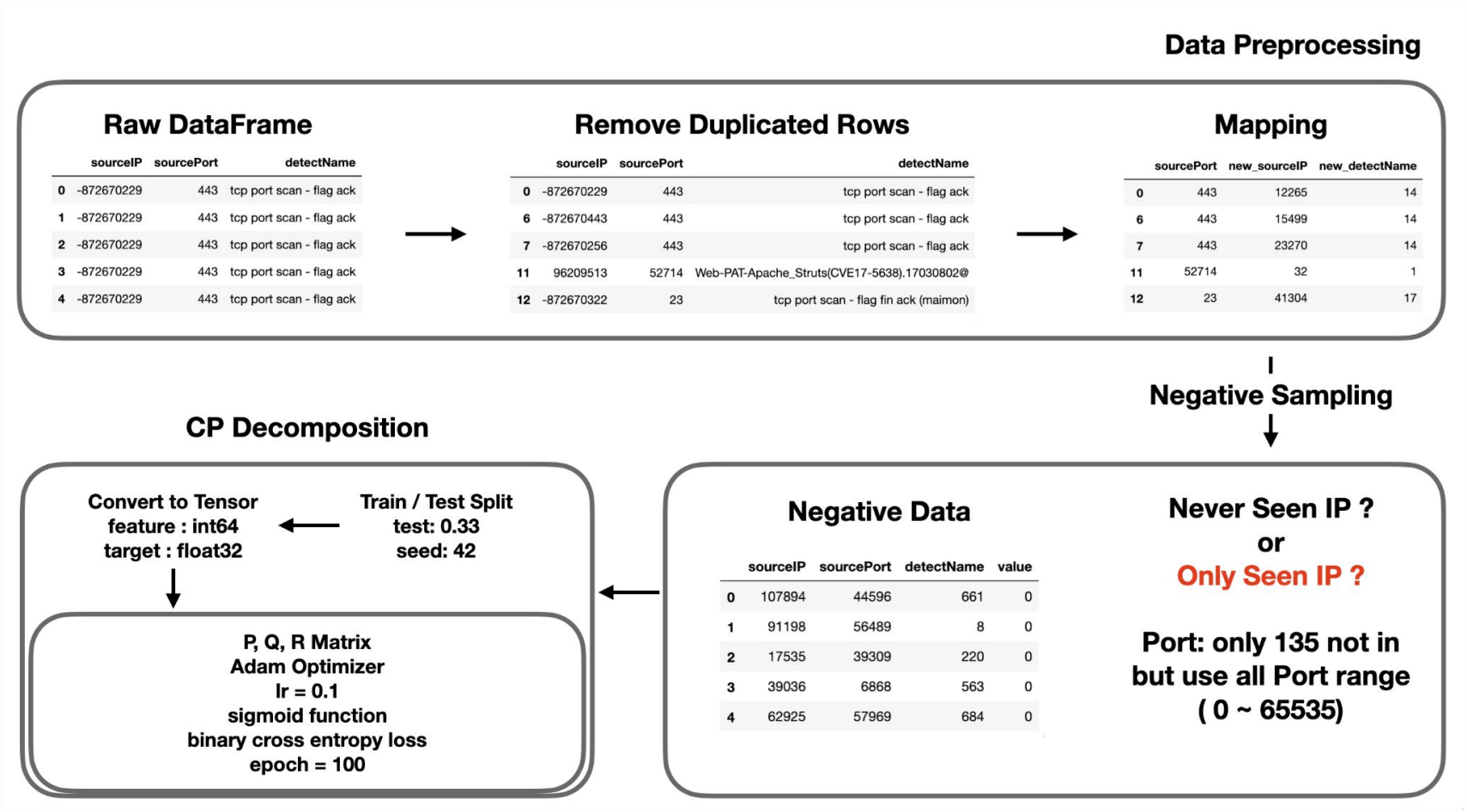
Train the data

- Negative Sampling
  - When we generate boolean tensor where all values are 1, all vectors become the same when the data is trained.
    → Negative Sampling!
  - Negative Sampling: Sampling value 0 for random IP, Port, EventName

- Cost Function
  - Since we use boolean tensor, MSE(Mean Square Error) of CP decomposition cannot be applied.
  - Use Binary Cross Entropy instead of MSE!

Analysis

The overall process of implementing an anomaly detection systems.



	sourceIP	sourcePort	detectName
0	-872670229	443	tcp port scan - flag ack
1	-872670229	443	tcp port scan - flag ack
2	-872670229	443	tcp port scan - flag ack
3	-872670229	443	tcp port scan - flag ack
4	-872670229	443	tcp port scan - flag ack

	sourceIP	sourcePort	detectName
0	-872670229	443	tcp port scan - flag ack
6	-872670443	443	tcp port scan - flag ack
7	-872670256	443	tcp port scan - flag ack
11	96209513	52714	Web-PAT-Apache_Struts(CVE17-5638).17030802@
12	-872670322	23	tcp port scan - flag fin ack (maimon)

	sourcePort	new_sourceIP	new_detectName
0	443	12265	14
6	443	15499	14
7	443	23270	14
11	52714	32	1
12	23	41304	17

	sourceIP	sourcePort	detectName	value
0	107894	44596	661	0
1	91198	56489	8	0
2	17535	39309	220	0
3	39036	6868	563	0
4	62925	57969	684	0

Results

Settings

Dataset: KISTI anomaly dataset (50GB)

Machines:

- CPU: AMD Ryzen 7 3800XT 8-Core Processor
- RAM: 128GB

Training Setup

Model: CP decomposition

Epoch: 100

Loss function: binary cross entropy loss

Optimizer: Adam

Result

```
epoch: 99, cost: 0.031527, acc: 98.354546
```

Thank you