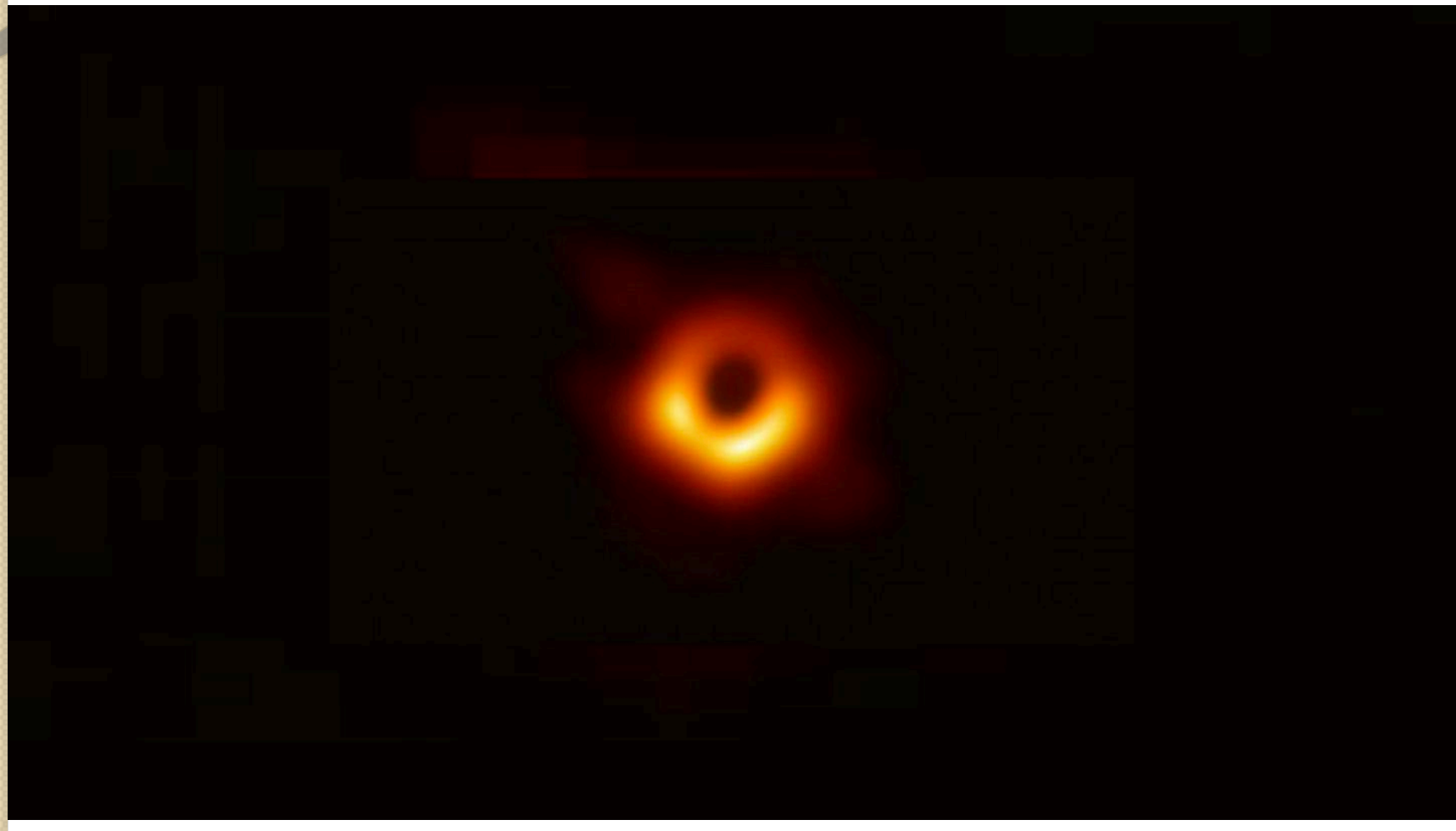
Large-Scale Graph Processing

Longbin Lai

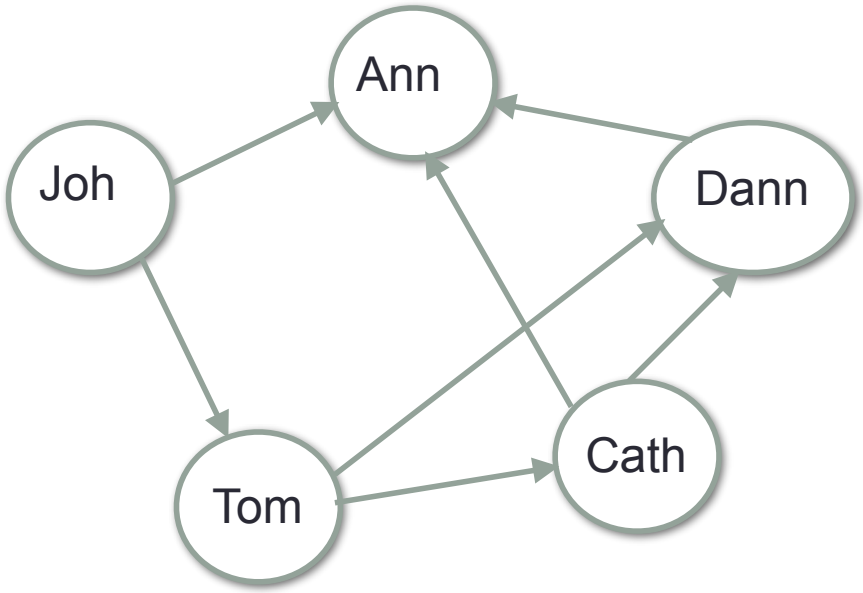
Big Data, Big Vision



BACKGROUND

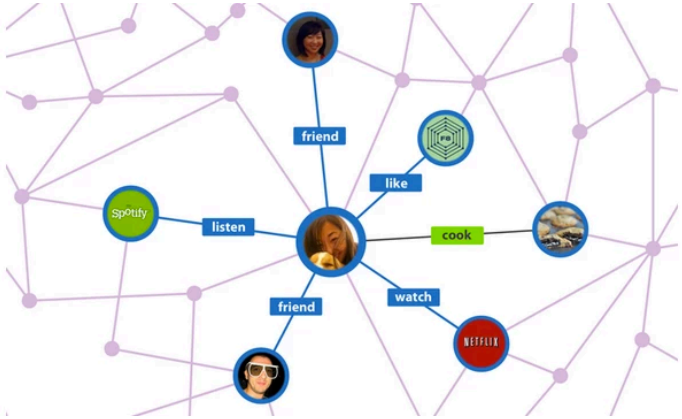
What is a graph?

- G = (V, E), where
  - V represents the set of vertices (entities)
  - E represents the set of edges (relations)
  - Both vertices and edges may contain additional information

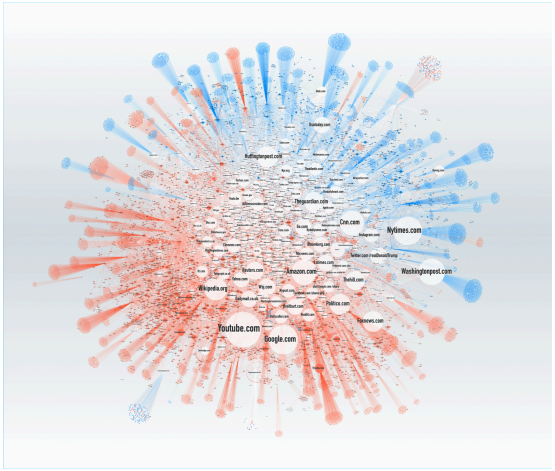


A twitter network

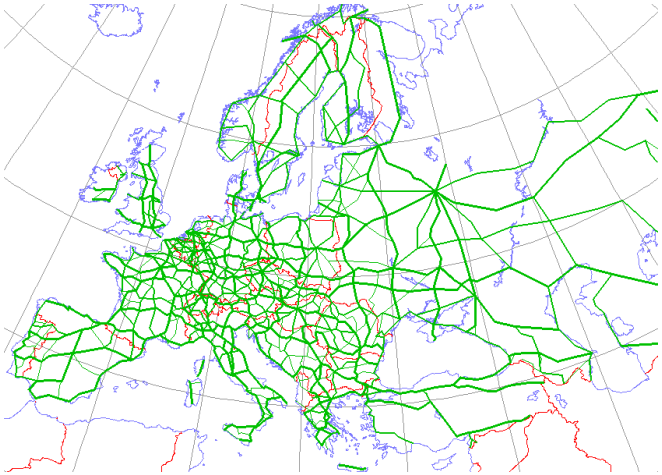
Graph is everywhere



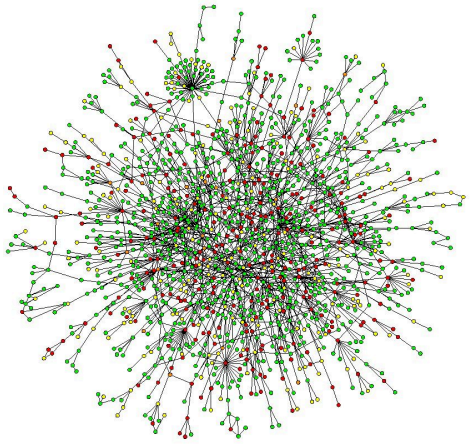
Social Network



Hyperlink Graphs



Road Networks

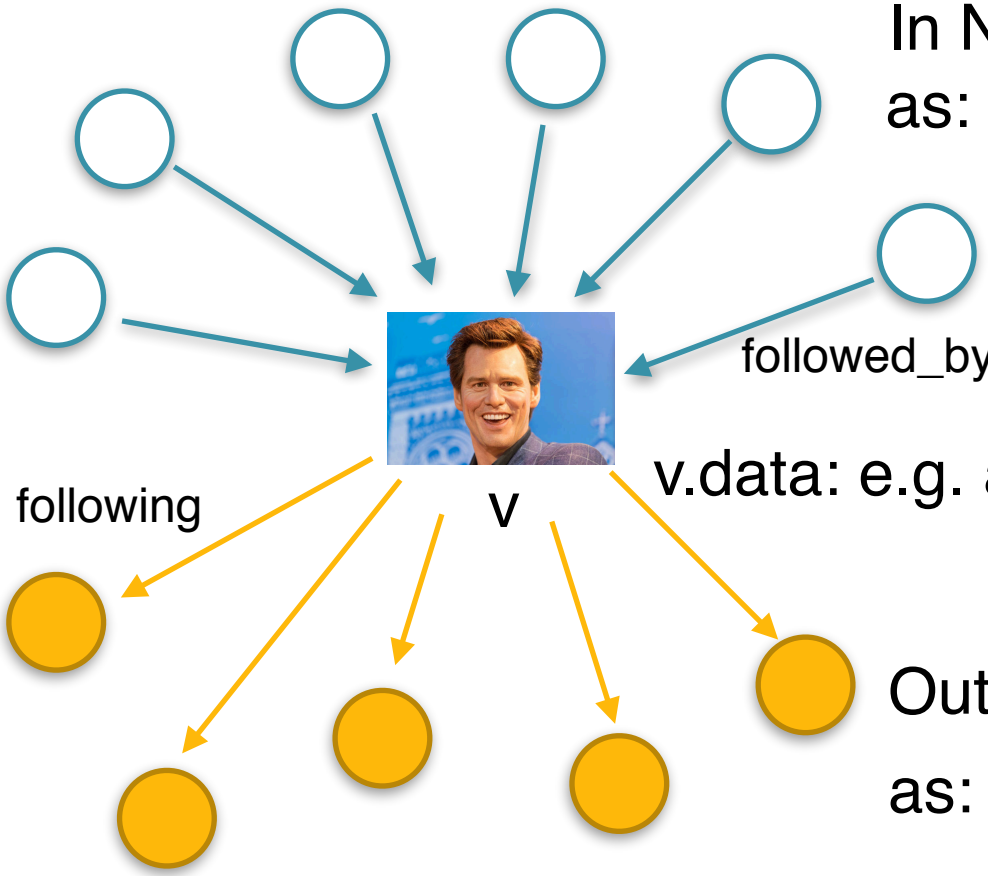


Protein-Protein-Interactions

Applications that graphs involved

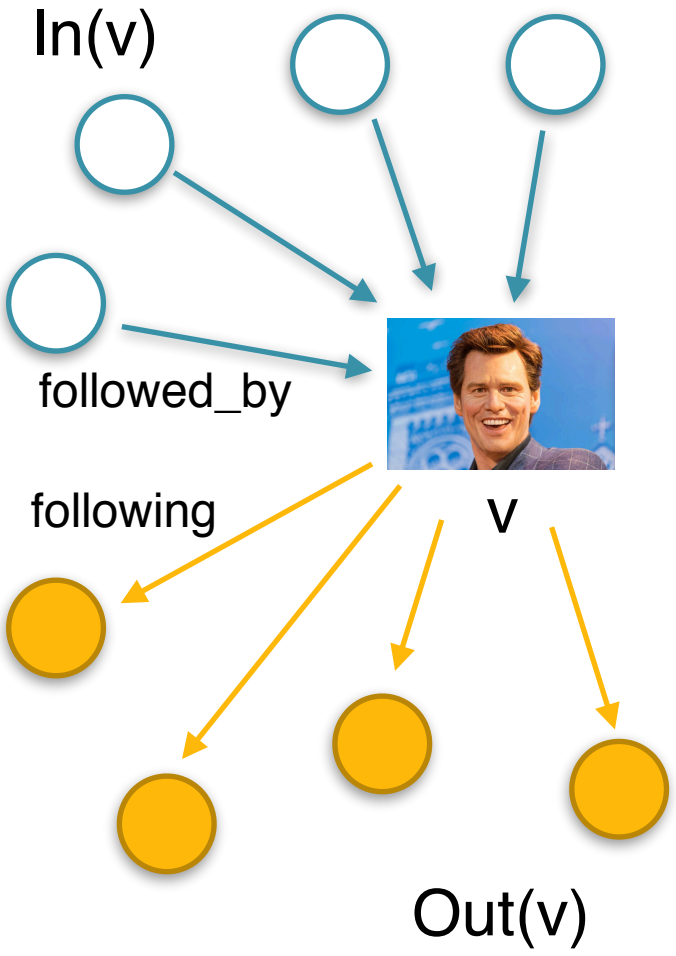
- Social networks
  - Friend recommendation, Event notification
- Hyperlink Graphs
  - PageRanking: How Google determines that certain pages should be put in the front.
- Road Network
  - Navigation: Routing from A to B
  - Neighbour recommendation: restaurant, hotel, ..
- Protein-Protein-Interactions
  - New treatments and drugs.

Graph Notations



Note: Sometimes directions are not important (friendship in Facebook is mutual)

Graph Computing - Basic Operations



A Node-Centric Model:

Gather(v):

```
for each v' in In(v):
    get_data_from(v')
```

Apply(fn, v):

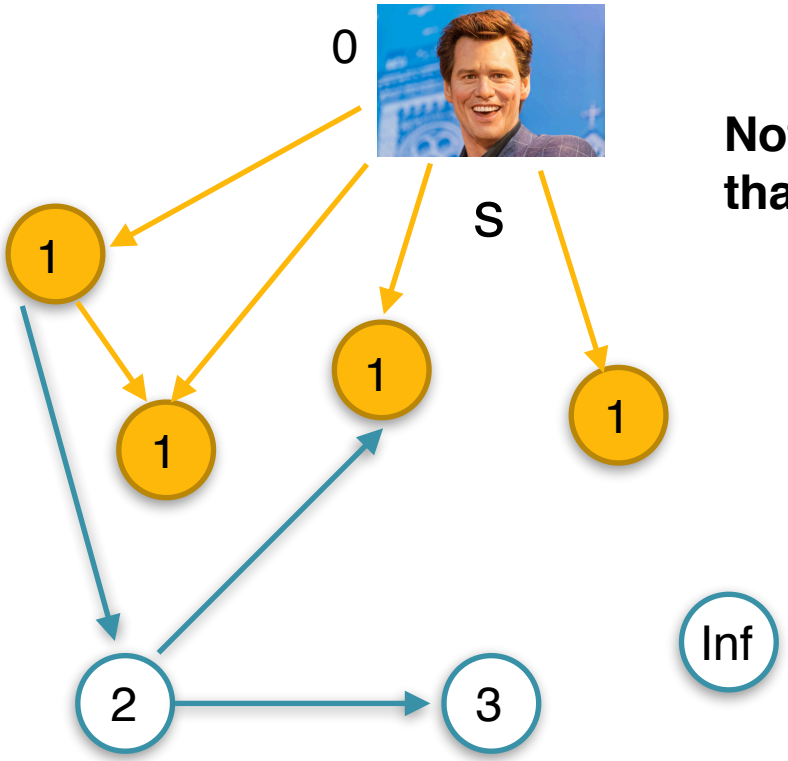
```
    fn(v.data)
```

Scatter(v):

```
for each v' in Out(v):
    send_to(v', d)
```

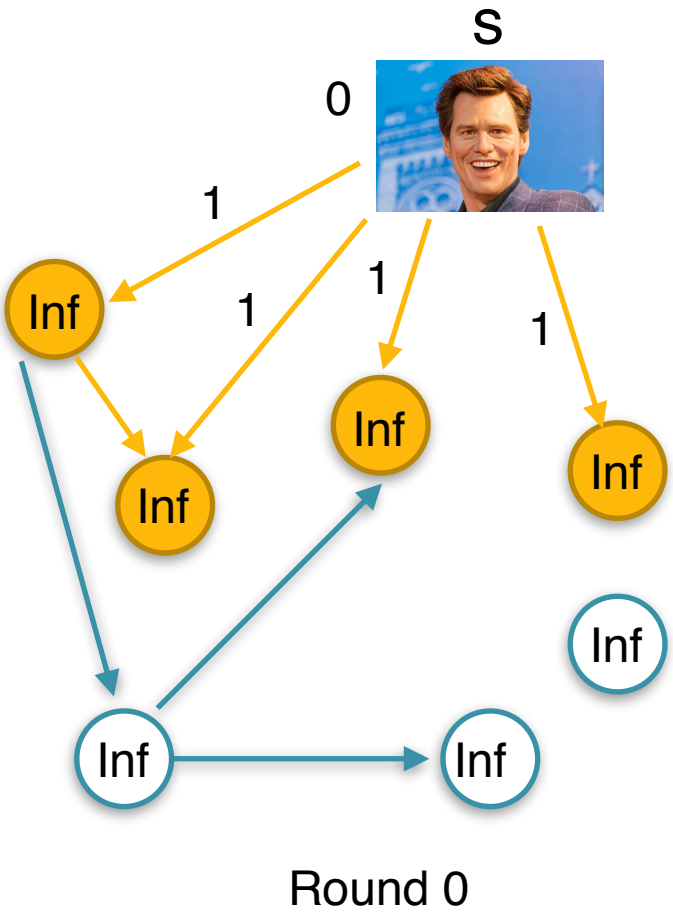
Example: Computing the Single-Sourced Shortest Path (SSSP)

- Df.: Given a node s, computing the shortest path (distance) of all other nodes from s



Note: can have weight (more than one) on each edge

Example: Computing the SSSP



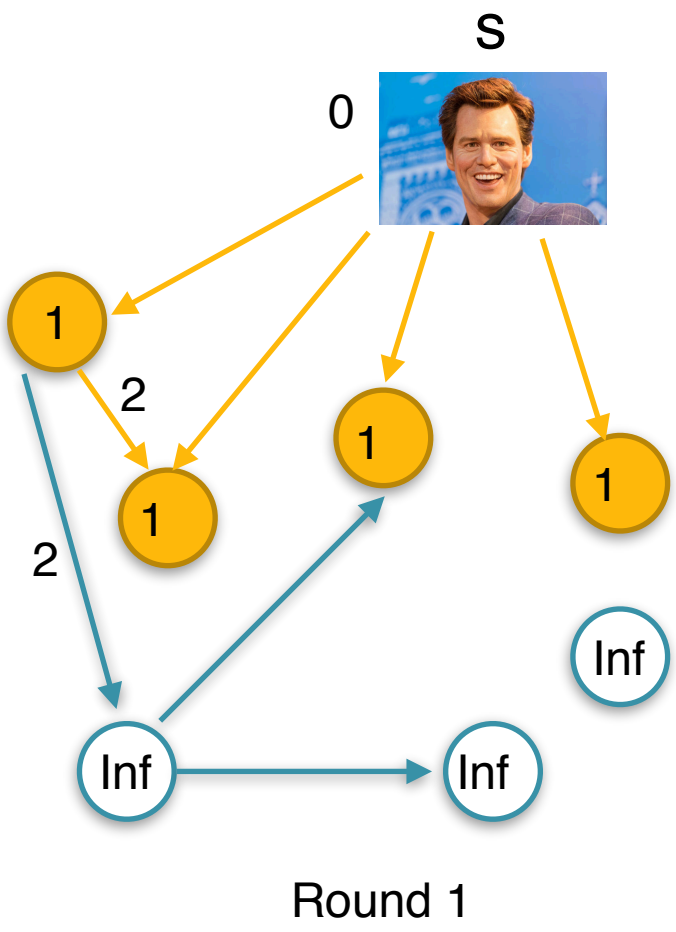
Initialise: for every v

```
if v = s:
    v.data = 0
else:
    v.data = Inf.
```

Scatter(s):

```
for each v' in Out(s):
    send_to(v', 1)
```

Example: Computing the SSSP



Round 1

Gather(v) for every v:

```
Receiving data from In(v)
```

Apply(fn, v) for every v that receive data:

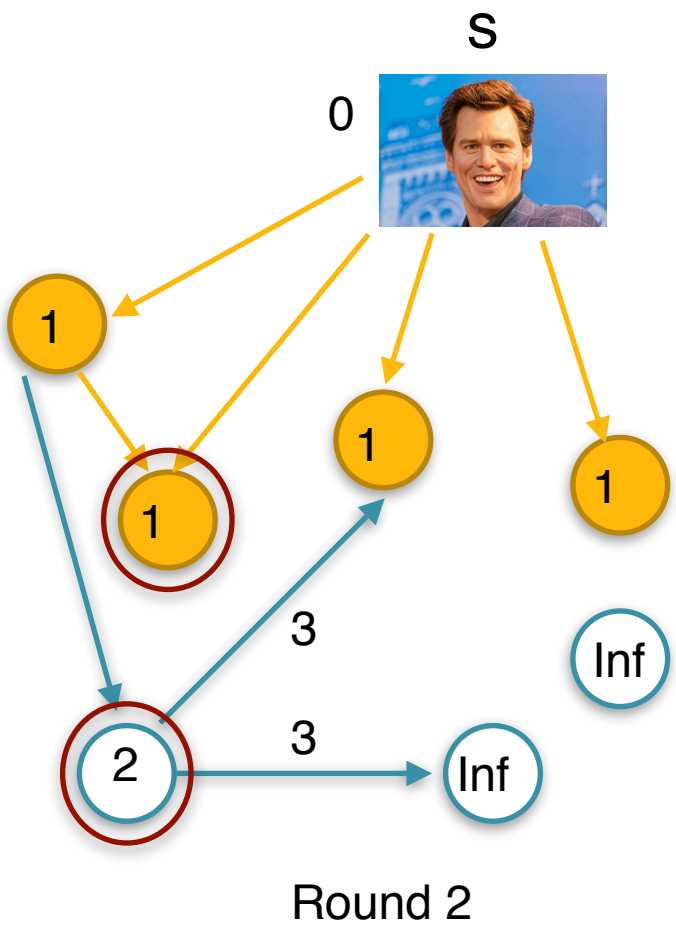
```
for each data received:
    fn: {v.data = min(v.data, data)}
```

Scatter(v) if v.data changed:

```
for each v' in Out(v):
    send_to(v.data + 1, v)
```

Do the above until all v's data remain **unchanged**

Example: Computing the SSSP



Gather(v) for every v:

```
Receiving data from In(v)
```

Apply(fn, v) for every v that receive data:

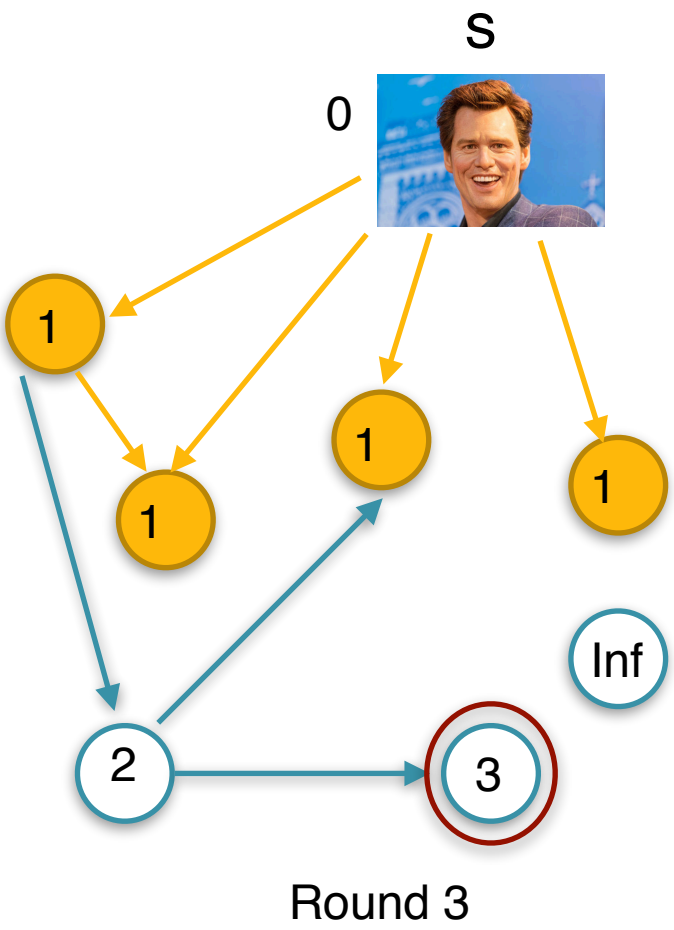
```
for each data received:
    fn: {v.data = min(v.data, data)}
```

Scatter(v) if v.data changed:

```
for each v' in Out(v):
    send_to(v.data + 1, v)
```

Do the above until all v's data remain **unchanged**

Example: Computing the SSSP



Gather(v) for every v:

```
Receiving data from In(v)
```

Apply(fn, v) for every v that receive data:

```
for each data received:
    fn: {v.data = min(v.data, data)}
```

Scatter(v) if v.data changed:

```
for each v' in Out(v):
    send_to(v.data + 1, v)
```

Do the above until all v's data remain **unchanged**

Features of Graph Computing

- Every node only does simple computation
- Exchange data among (in/out) neighbours
- Run in **multiple** iterations (rounds) until converge
- Every iteration there may be **a small portion** of nodes updating their value

MOTIVATION

Graph's getting BIG

- Facebook
  - ~**2.20 billion** monthly active Facebook users for Q1 2018
- Google
  - ~**45 billion** web pages are indiced in April 2018, a figure that is piling up every month.
  - How large is there? **Astronomical**

Motivation

- Many practical computing problems concern large graphs (web graph, social networks, transportation network).
  - Example :
    - (Single Sourced) Shortest Path
    - Clustering
    - Page Rank
    - Connected Components

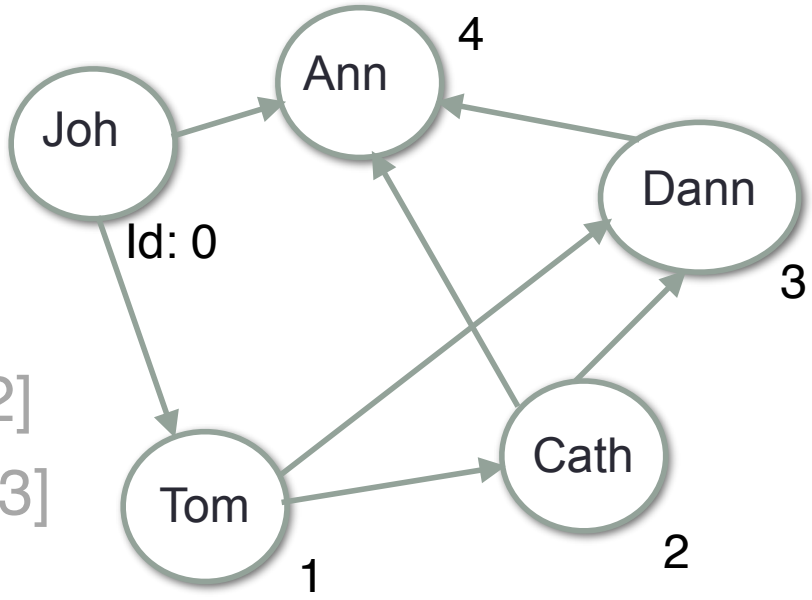
Motivation

- Alternatives :
  - Create distributed infrastructure for ***every*** new algorithm
  - Map Reduce
    - Inter-stage communication overhead
  - Single computer graph library
    - does not scale
- We need to build a **scalable** distributed solution

PARTITION THE GRAPH DATA

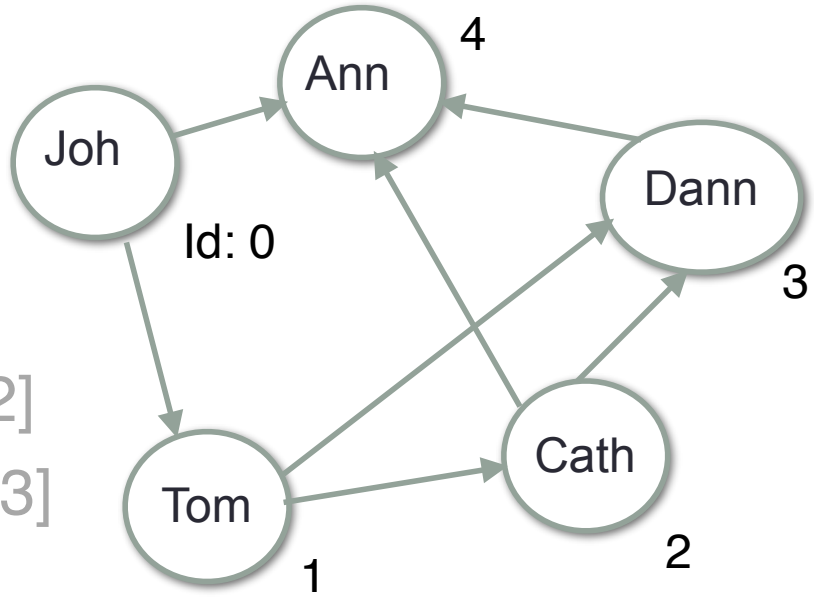
How Graph is stored

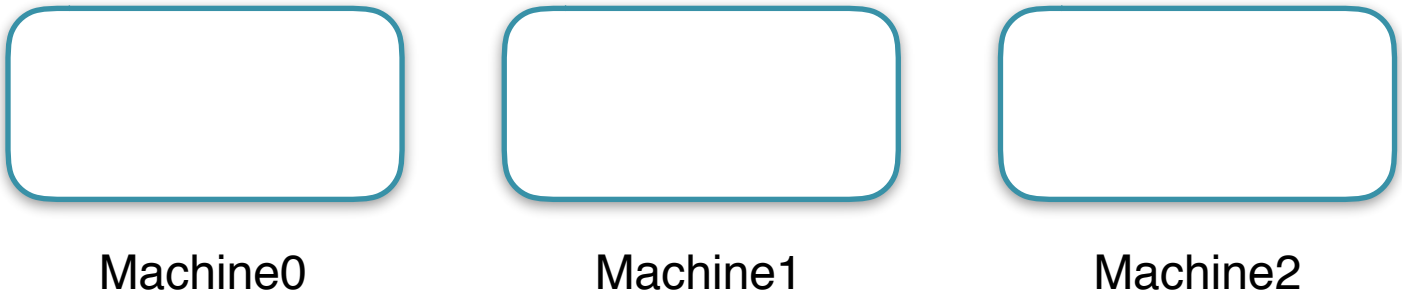
- Adjacency List
  - 0, {John}: [1, 4], []
  - 1, {Tom}: [2, 3], [0]
  - 2, {Cathy}: [3, 4], [1]
  - 3, {Danny}: [4], [1, 2]
  - 4, {Anna}: [], [0, 2, 3]



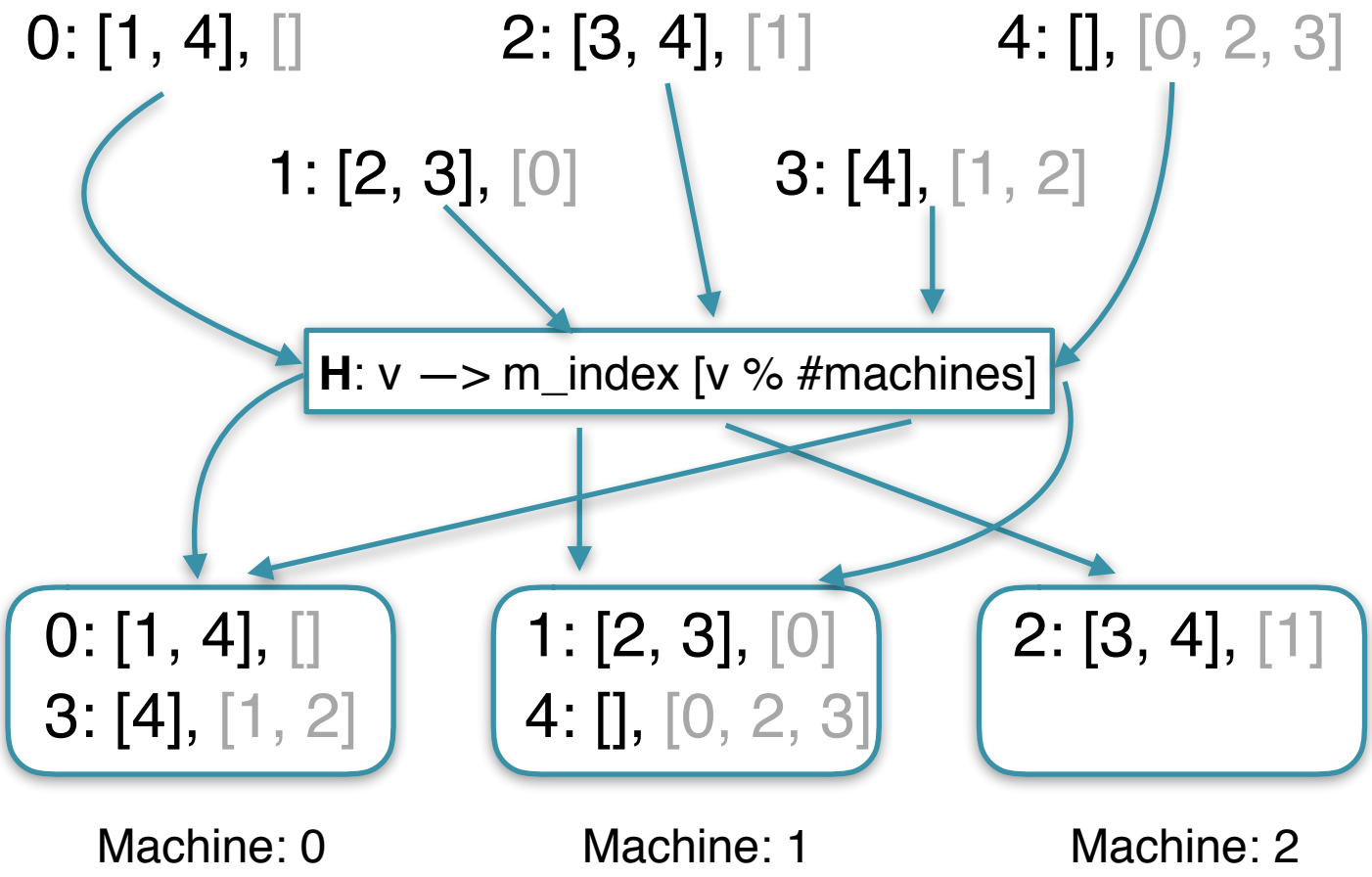
How Graph is partitioned

- Adjacency List
  - 0, {John}: [1, 4], []
  - 1, {Tom}: [2, 3], [0]
  - 2, {Cathy}: [3, 4], [1]
  - 3, {Danny}: [4], [1, 2]
  - 4, {Anna}: [], [0, 2, 3]



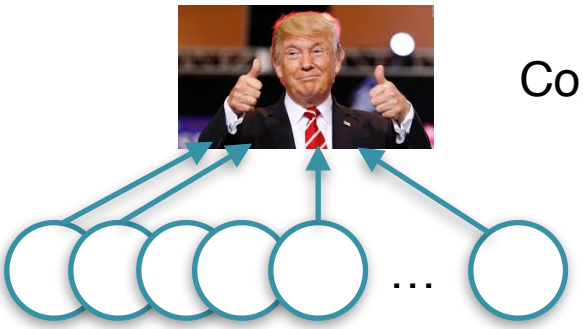


Hash Partition



What about the vertex's data

- The data only goes with the vertex that is specified in that partition.
- In the adjacency list, only id is maintained:
  - Data may be too large to maintained
  - Consistency issue: only data change, need to change everywhere



Copy everyone's data here: No WISE

11.5m followers

Principles of Graph Partition

- Rule I
  - Each partition should contain more or less same amount of data (load balance)
- Rule II
  - Cross Edge: The edge (u, v) where u and v belong to two different partitions
  - Data need to be exchanged via the network
  - Keep the number of cross edges as small as possible to reduce communication

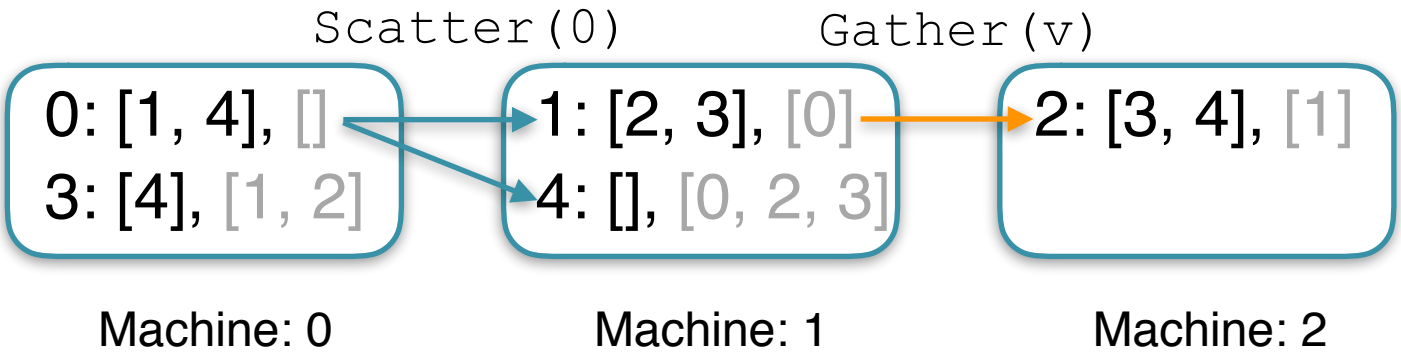
A perfect partition often requires very complicated computations

Graph Computation after partitioning

```
Gather(v):
for each v' in In(v):
    get_data_from(v')
```

```
Apply(v):
    do_something(v.data)

Scatter(v):
for each v' in Out(v):
    send_to(d, v')
```



Data not present in current partition must be exchanged via the network

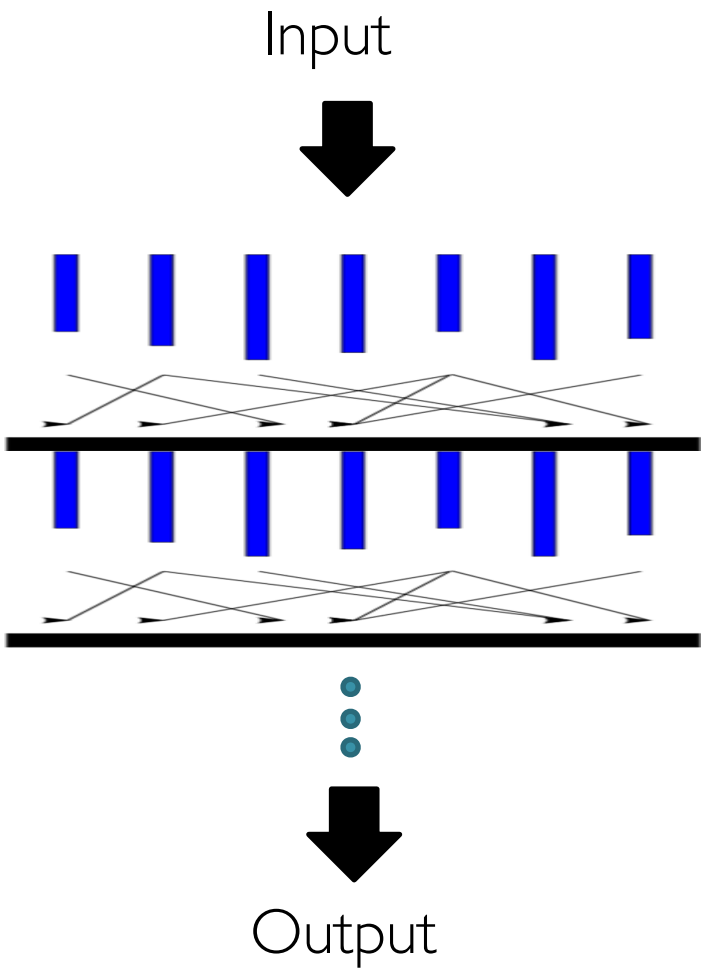
PREGEL: GOOGLE'S DISTRIBUTED GRAPH COMPUTATION

Pregel

- Scalable and Fault-tolerant platform
- API with flexibility to express arbitrary algorithm
- Inspired by Valiant's Bulk Synchronous Parallel (BSP) model[4]
- Node-centric computation (Think like a node):
  - An adapt of gather, apply and scatter

COMPUTATION MODEL

Computation Model (1/3)



BSP Computation Model

Supersteps
(a sequence of iterations)

Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Computation Model (2/3)

- Local computation: Each vertex v
  - **`Gather(v)`**: Receives messages sent in the previous superstep
  - **`Apply(fn, v)`**: Executes the same user-defined function and modifies its value
  - **`Scatter(v)`**: Sends messages to other vertices (to be received in the next superstep)

  - **Votes to halt if it has no further work to do**
    - Remember no every vertex needs to do something in every round (SSSP e.g.)
    - When all vertices are voted to halt, the program terminates

Computation Model (3/3)

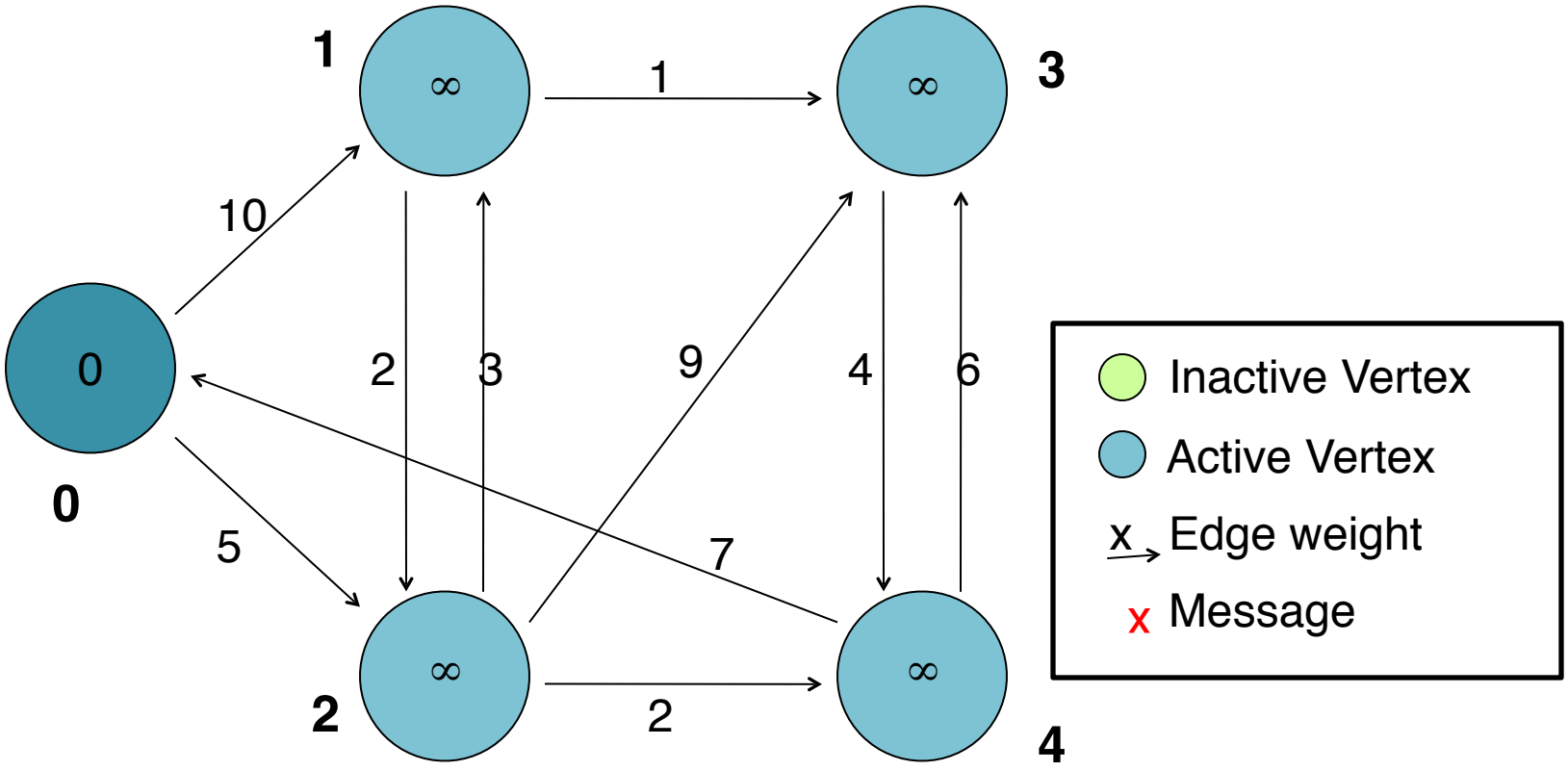
State machine for a vertex

- Termination condition
  - All vertices are simultaneously inactive
  - There are no messages in transit

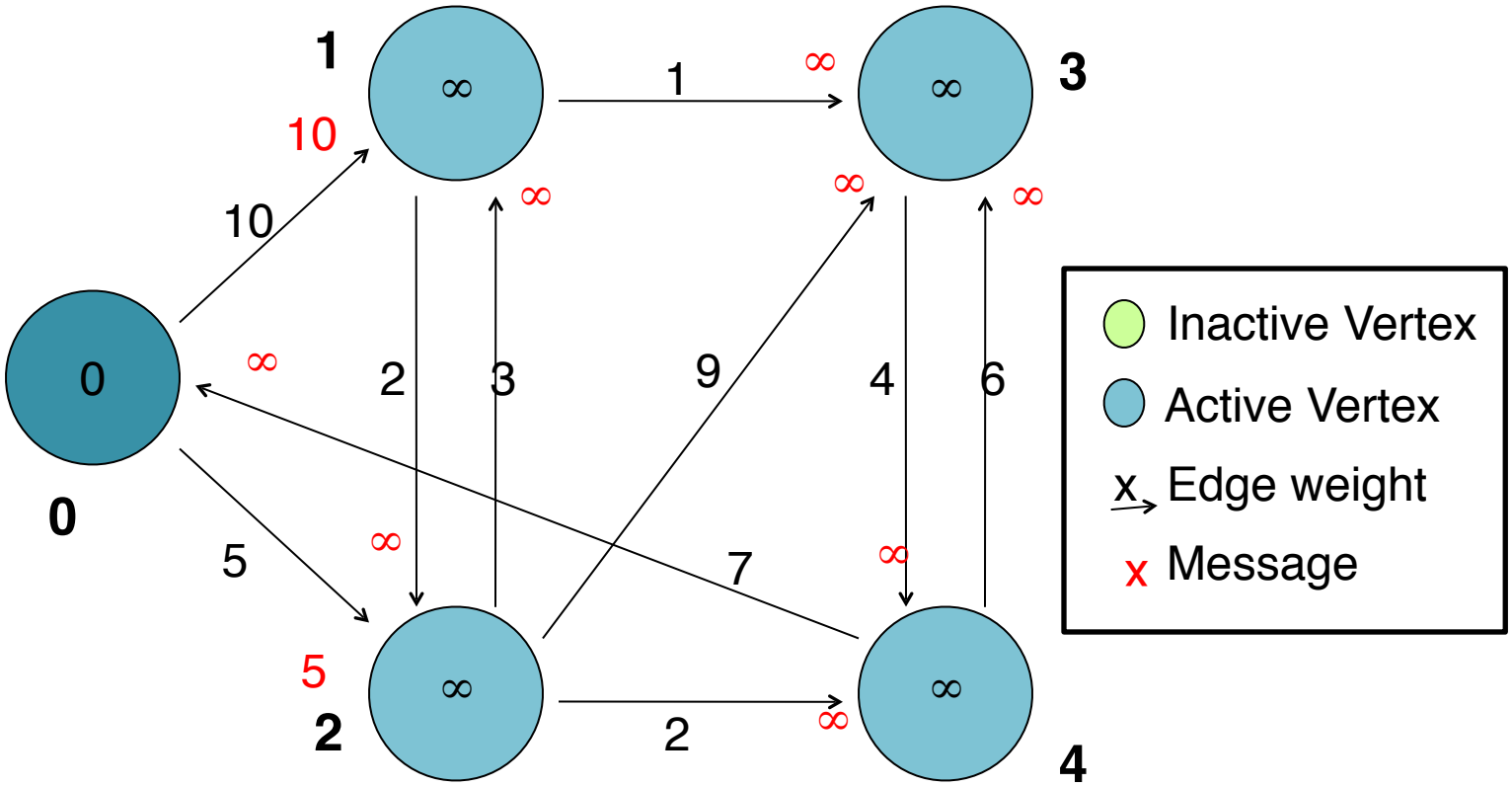
Example

- Single Source Shortest Path
  - Find shortest path from a source node to all target nodes
  - Example taken from talk by Taewhi Lee , 2010

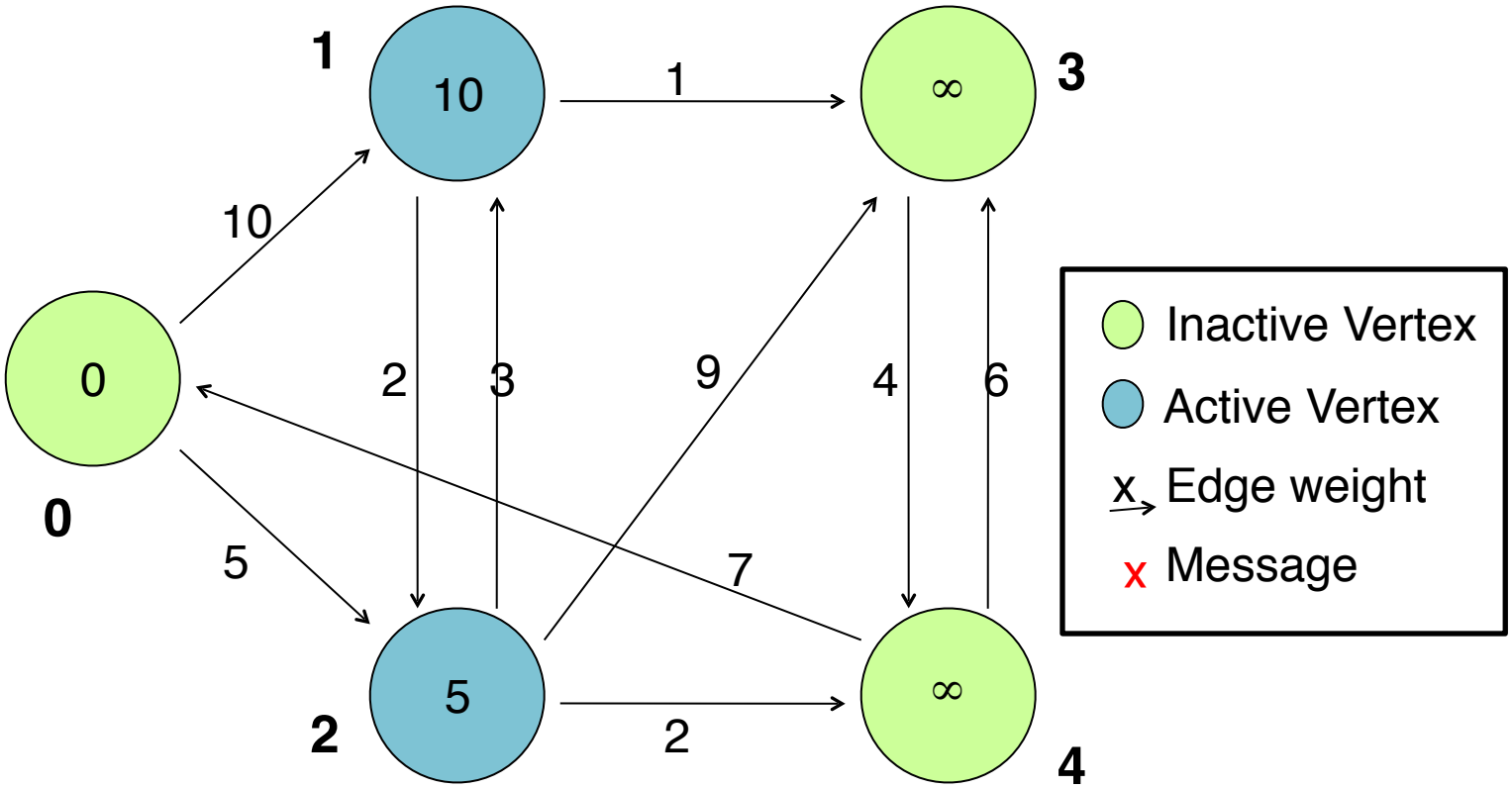
Example: SSSP – Parallel BFS in Pregel



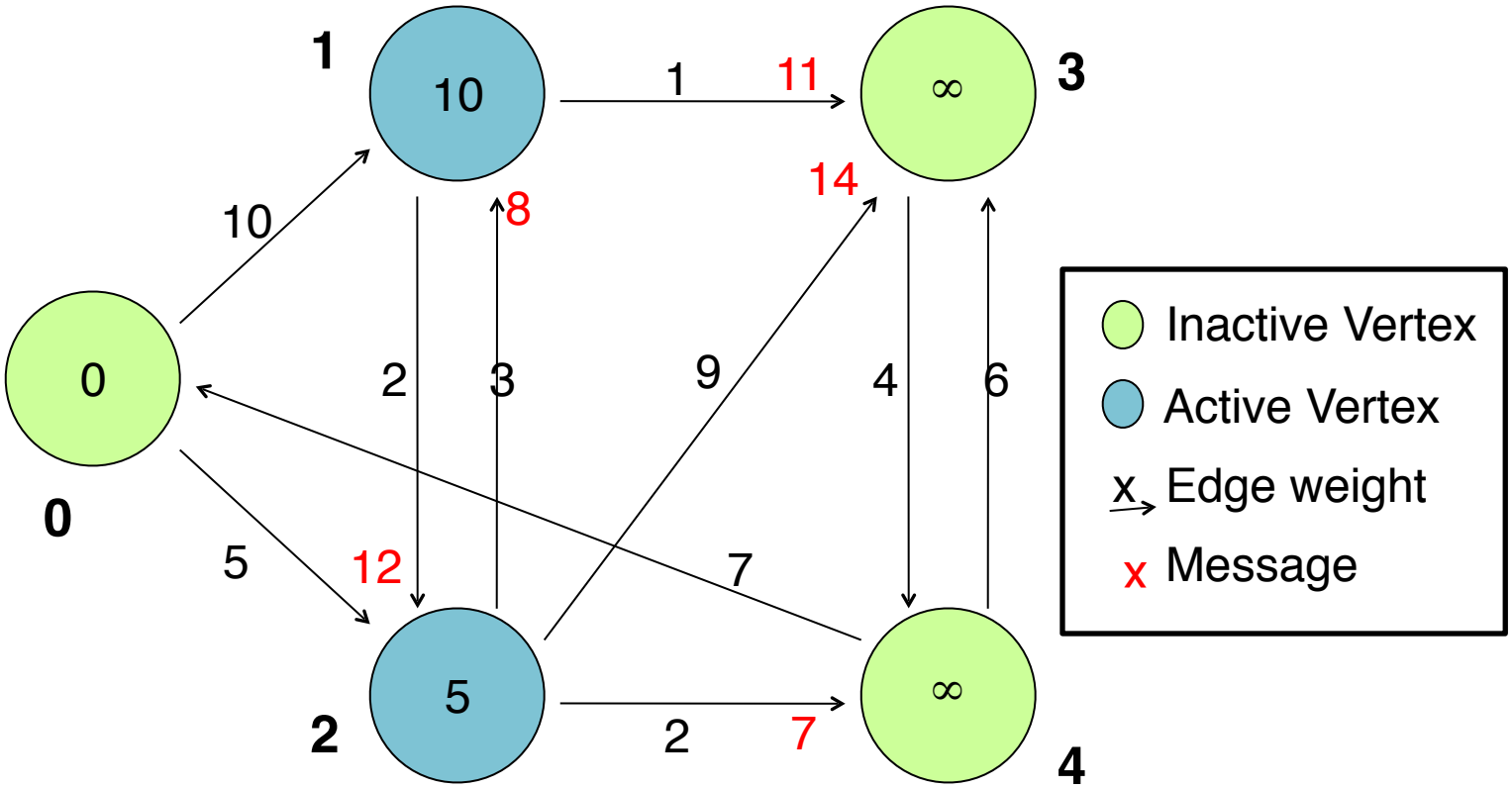
Example: SSSP – Parallel BFS in Pregel



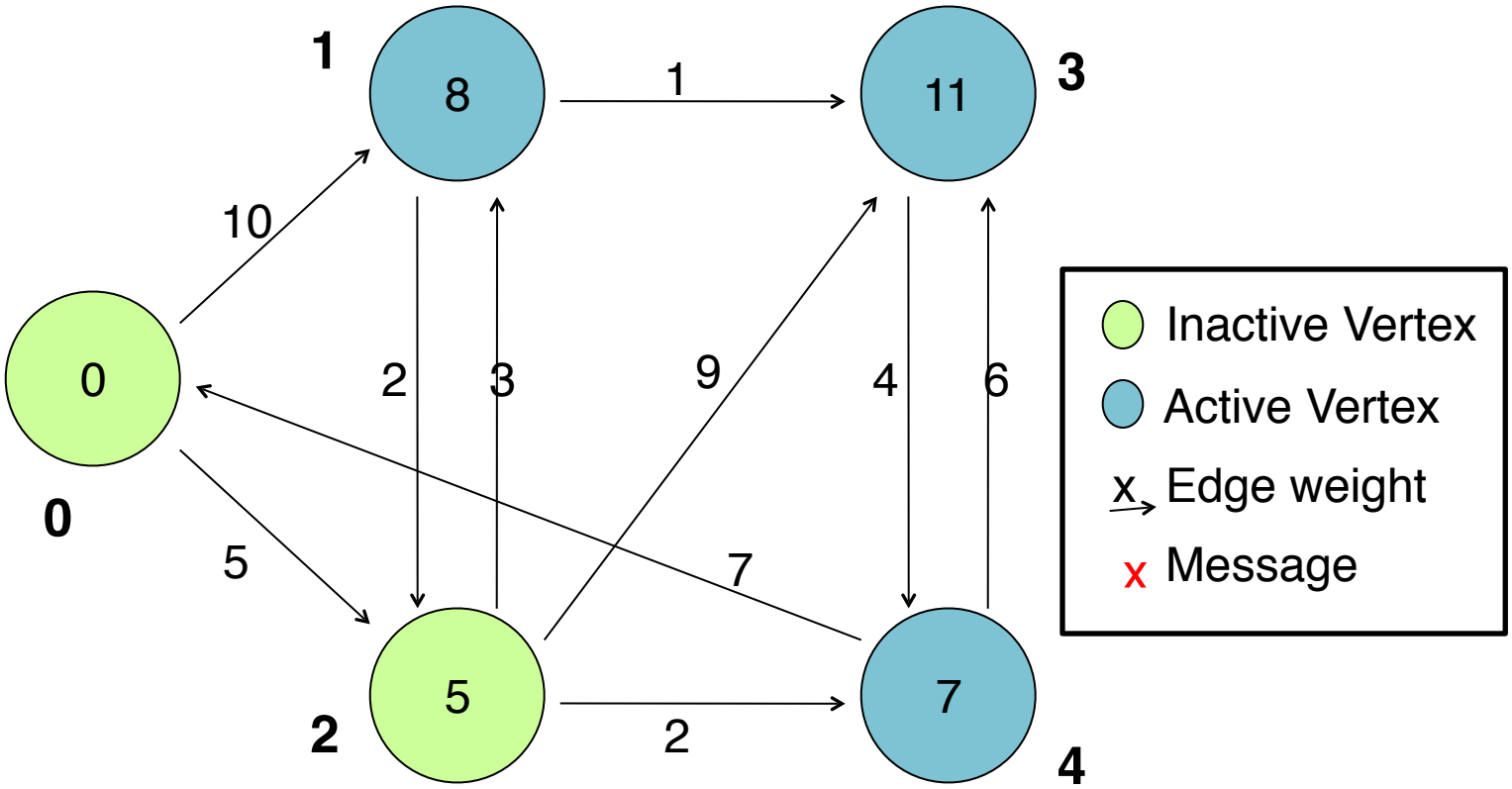
Example: SSSP – Parallel BFS in Pregel



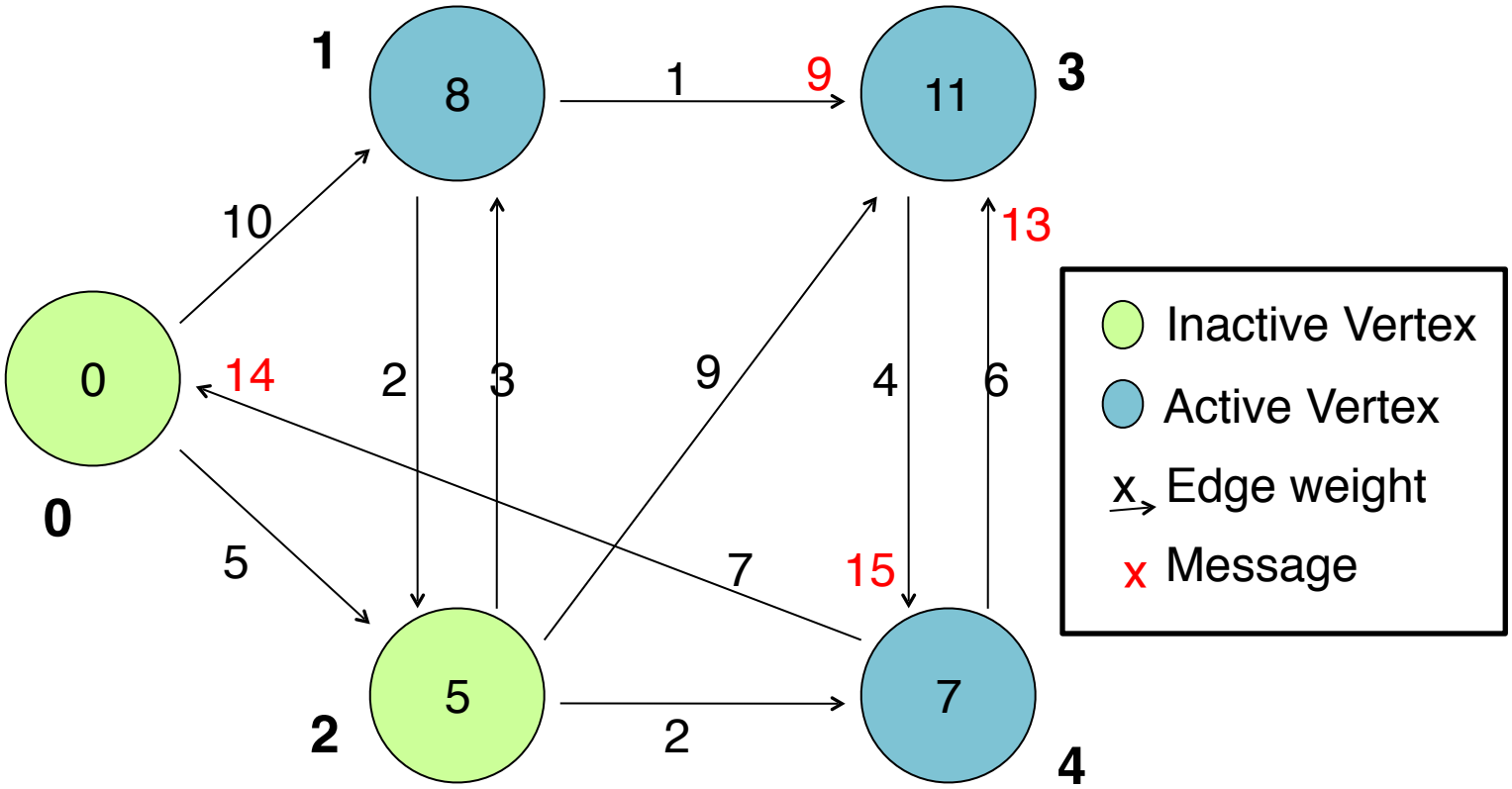
Example: SSSP – Parallel BFS in Pregel



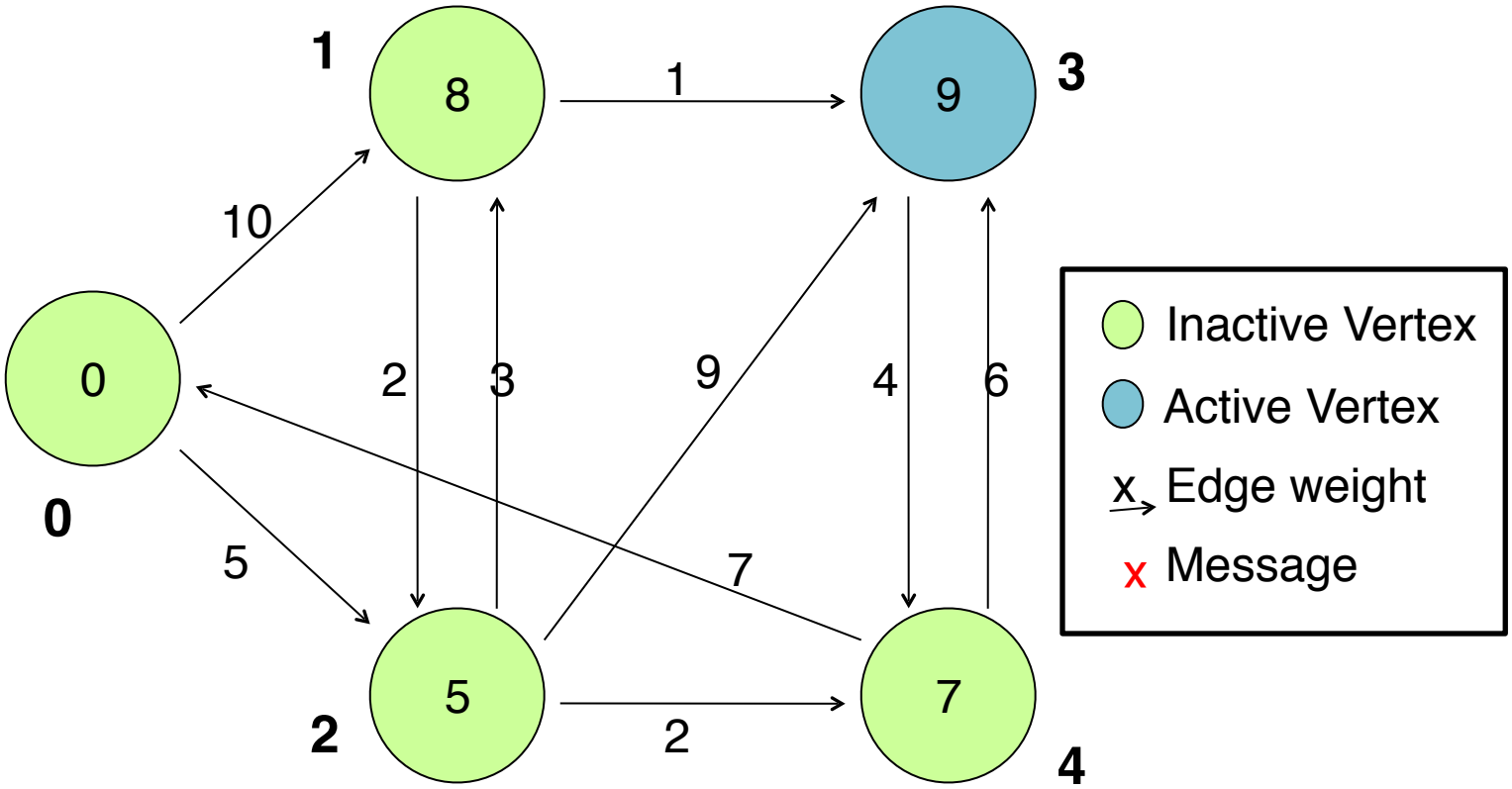
Example: SSSP – Parallel BFS in Pregel



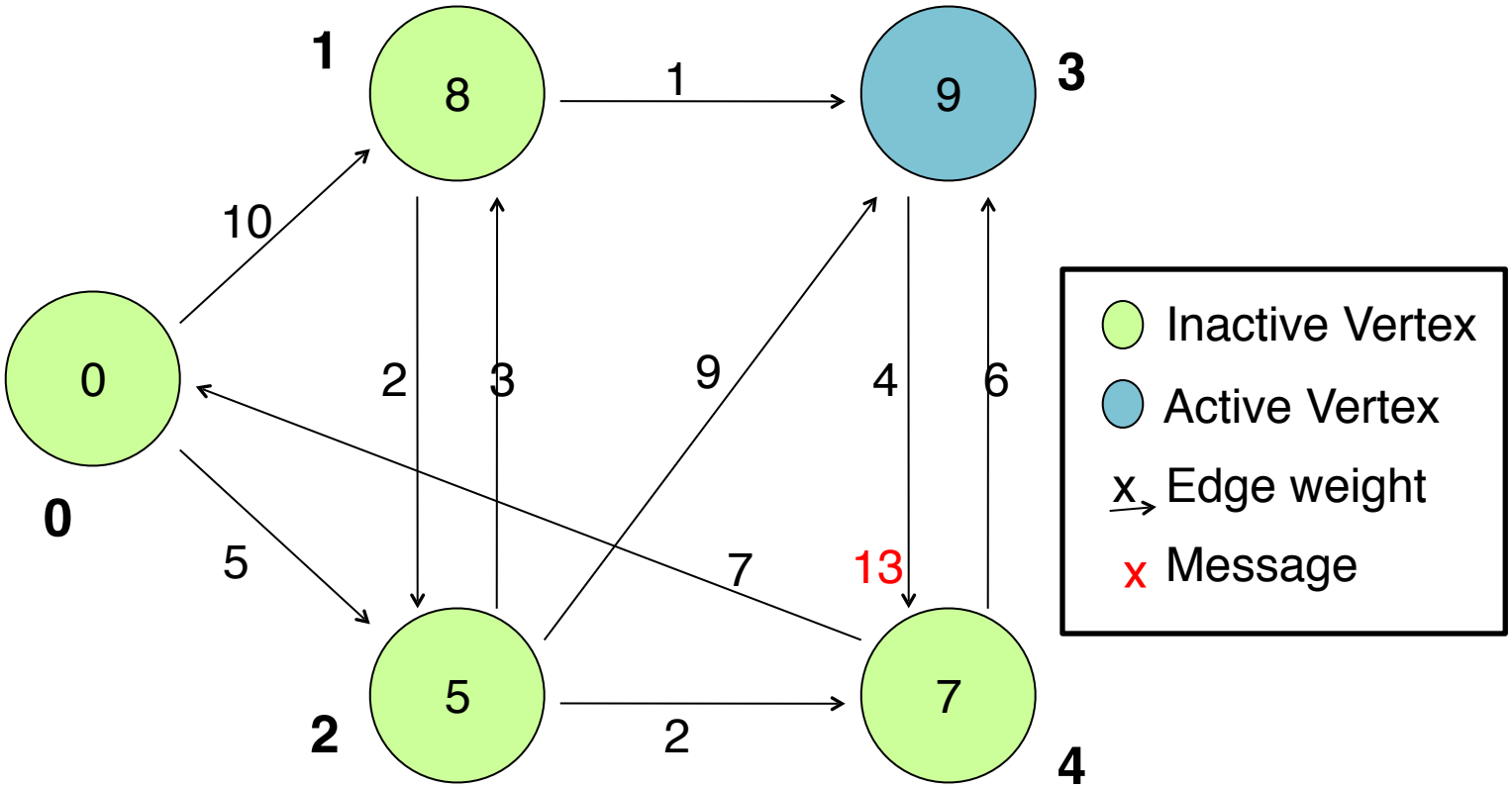
Example: SSSP – Parallel BFS in Pregel



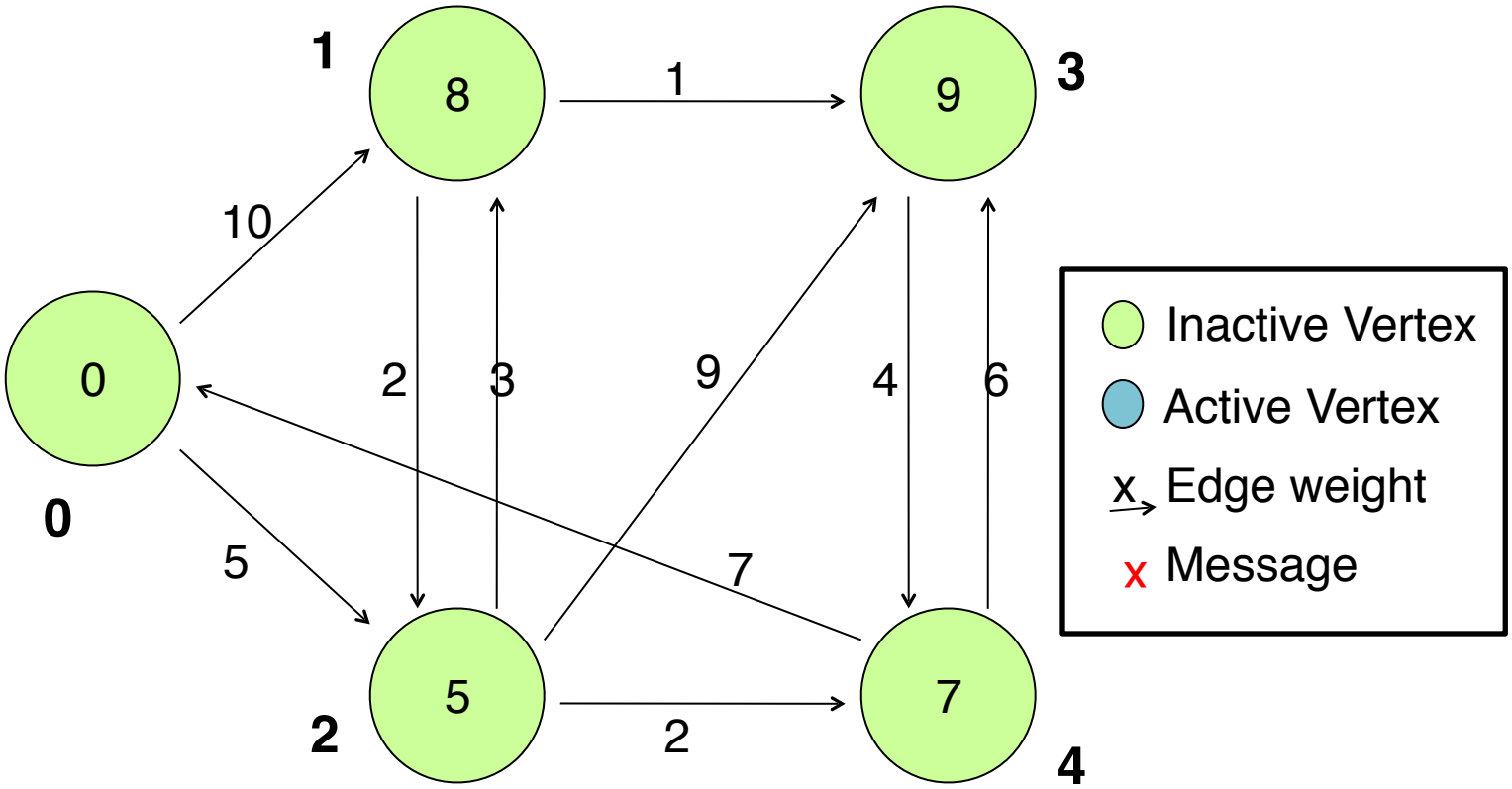
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel

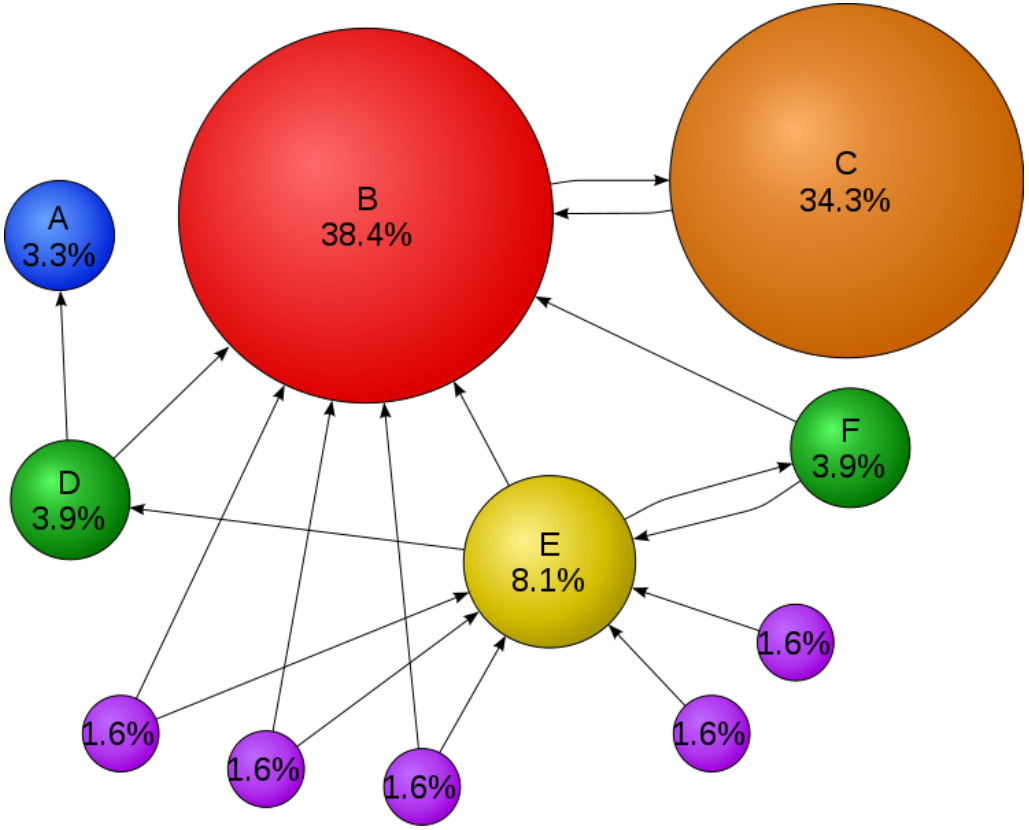


Example: SSSP – Parallel BFS in Pregel



Example: PageRank

- Used to determine the importance of a document based on the number of references to it and the importance of the source documents themselves



Courtesy: Wikipedia

Example: PageRank

- Used to determine the importance of a document based on the number of references to it and the importance of the source documents themselves

A = A given page

T_1 …. T_n = Pages that point to page A (citations)

d = Damping factor between 0 and 1 (usually kept as 0.85)

C(T) = number of links going out of T

PR(A) = the PageRank of page A

$$PR(A) = (1-d) + d \bullet (\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + ... + \frac{PR(T_n)}{C(T_n)})$$

PageRank in Pregel

- **Superstep 0 (Initialize):** Value of each vertex is **1/NumVertices()**

```
void Compute(MessageIterator* msgs) {
   // The system does this in the background
   Gather: {self.In(), msgs};

   Apply:
   double sum = 0;
   for each msg in msgs:
      sum += msgs.data;

   self.pr = 0.15 + 0.85 * sum;

   Scatter:
   for each nbr in self.Out():
       send_to(self.pr / |self.out()|, nbr)

   VoteToHalt()
}
```

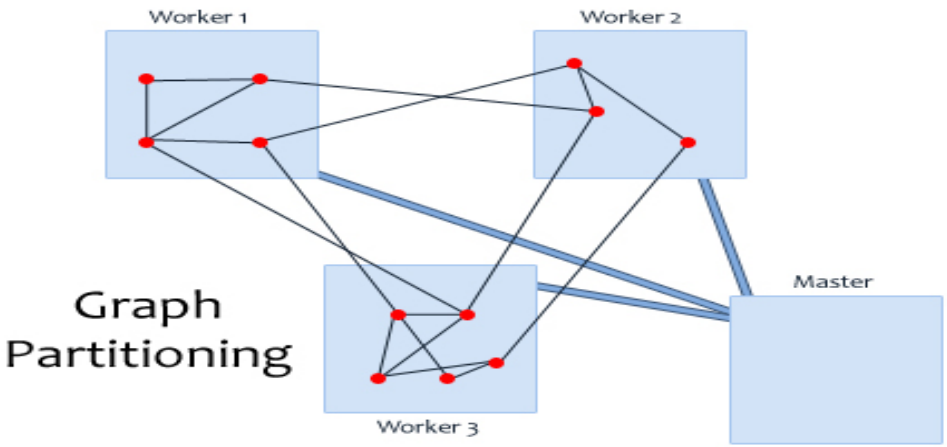
SYSTEM ARCHITECTURE

System Architecture

- Pregel system uses the master/worker model
  - Master
    - Coordinates worker
    - Recovers faults of workers
  - Worker
    - Processes its task
    - Communicates with the other workers
- Persistent data is in distributed storage system
- Temporary data is stored on local disk

Pregel Execution (1/3)

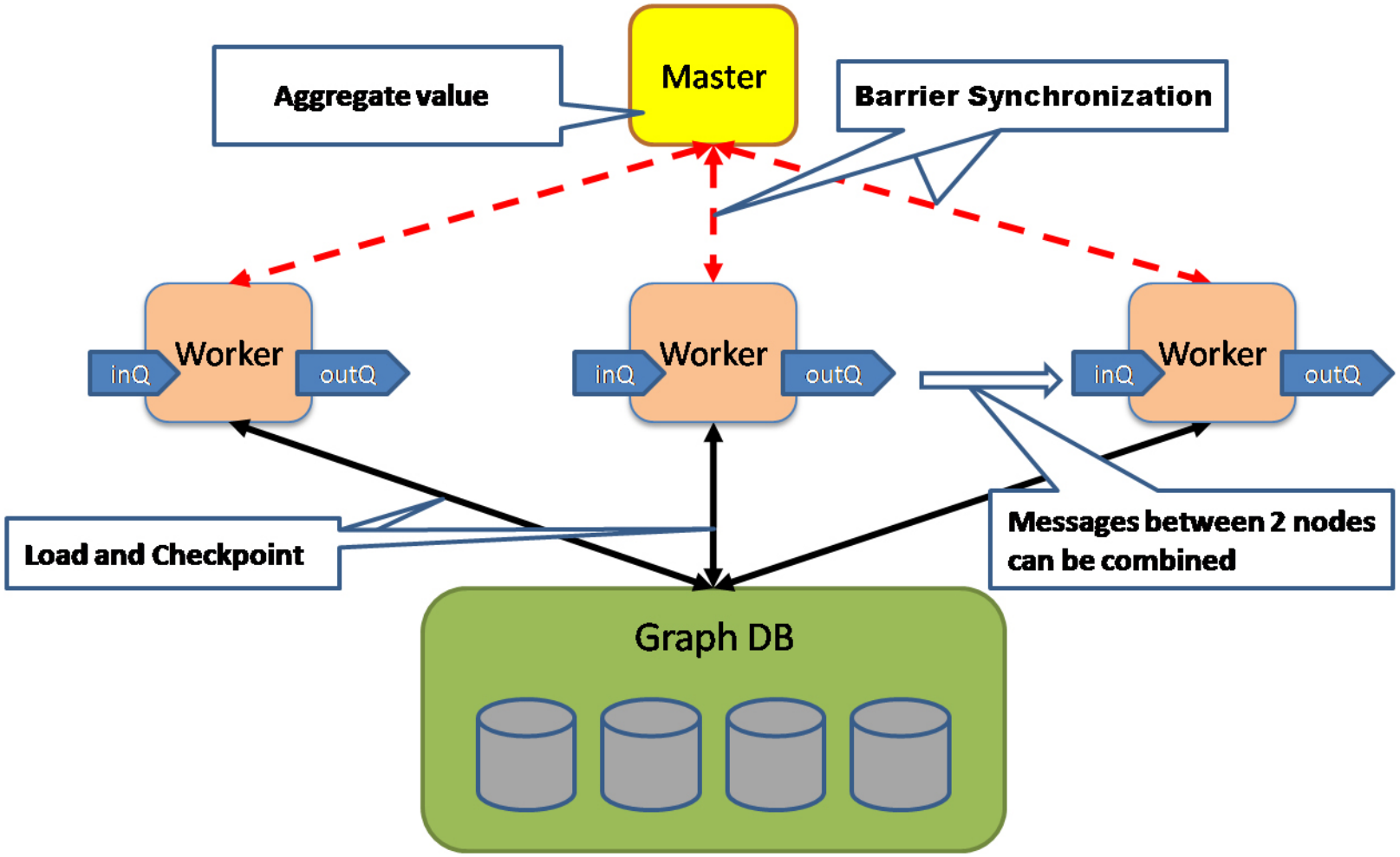
1. Master partitions the graph and assigns one or more partitions to each worker
2. Many copies of the program begin executing on a cluster of machines
3. Master also assigns a partition of the input to each worker
   - Each worker loads the vertices and marks them as active



Pregel Execution (2/3)

4. The master instructs each worker to perform a superstep
   - 4.1. Each worker loops through its active vertices & computes for each vertex
   - 4.2. Messages are sent asynchronously, but are delivered before the end of the superstep
   - 4.3. This step is repeated as long as any vertices are active, or any messages are in transit
5. After the computation halts, the master may instruct each worker to save its portion of the graph

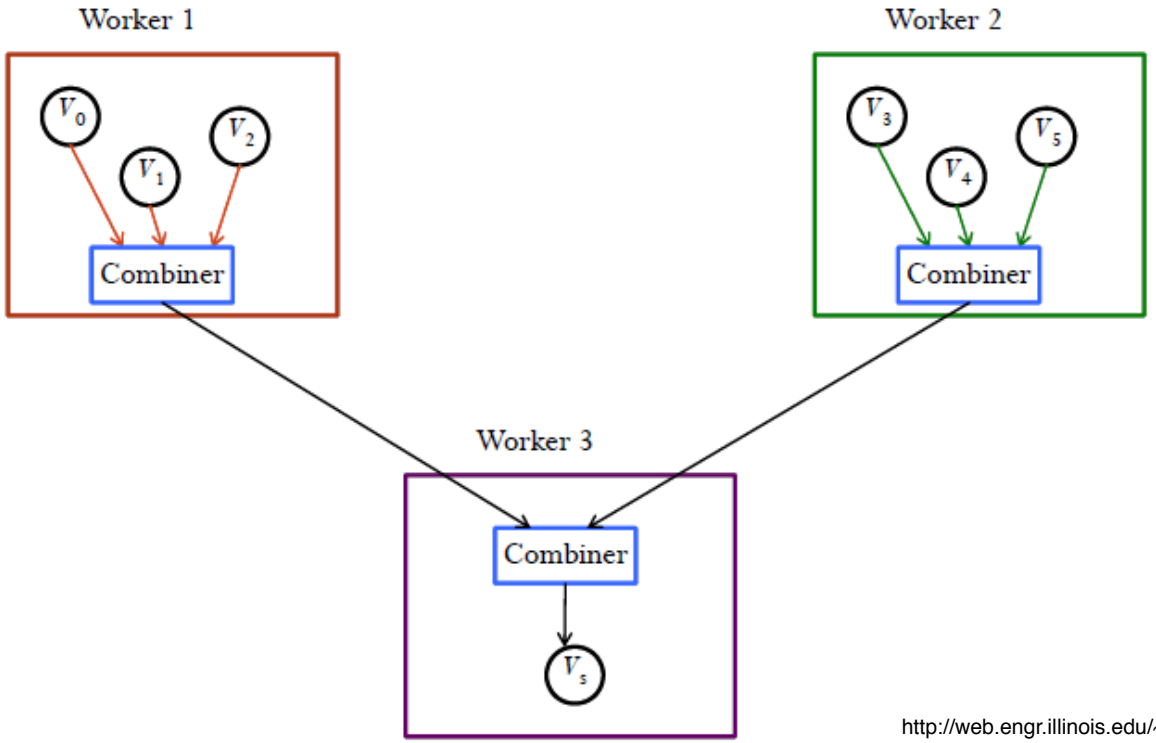
Pregel Execution (3/3)



http://java.dzone.com/news/google-pregel-graph-processing

Combiner

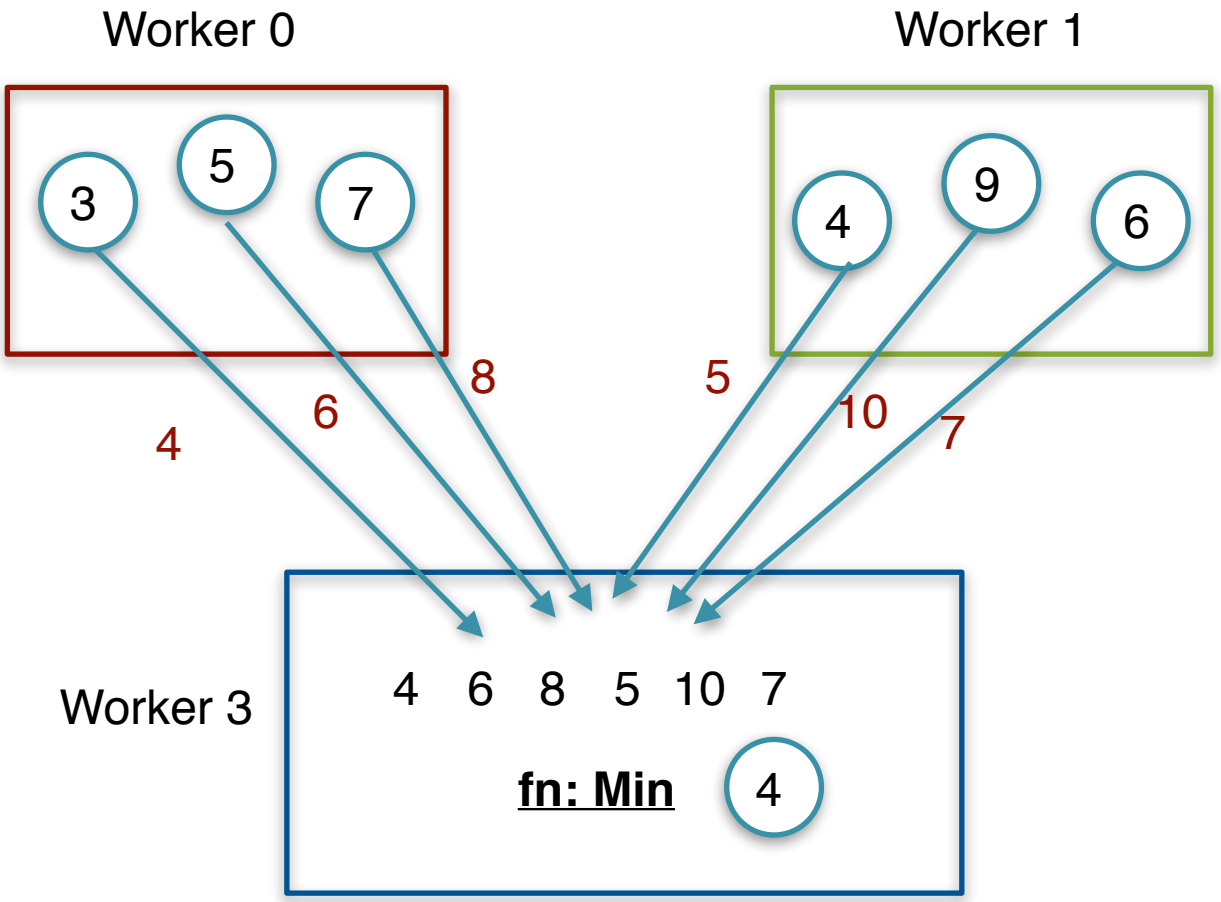
- Worker can combine messages reported by its vertices and send out one single message
- Reduce message traffic and disk space



http://web.engr.illinois.edu/~pzhao4/

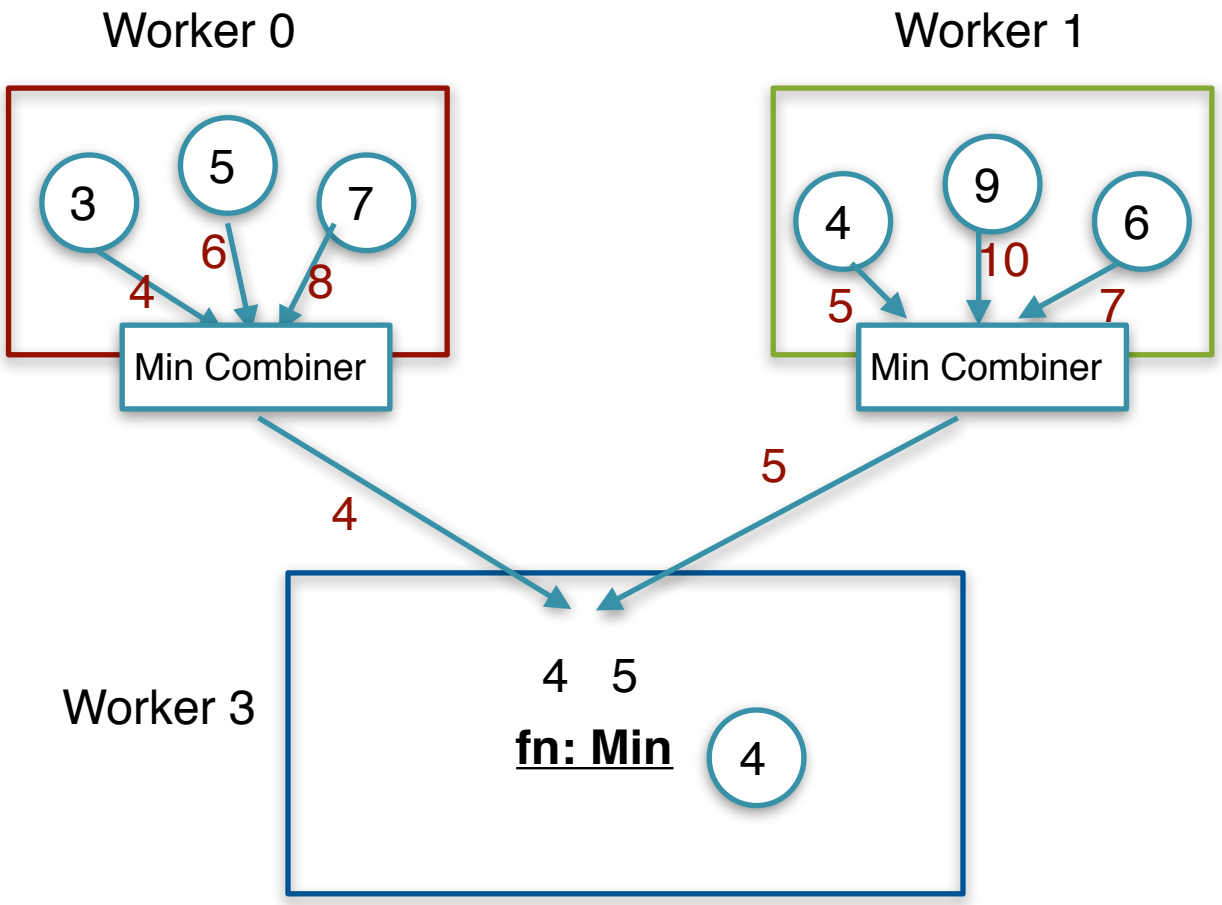
Combiner in SSSP

- **No Combiner**



Combiner in SSSP

- **Min Combiner:**
  - min is associative: min(a, b, c) = min(min(a, b), c)



Fault Tolerance (1/2)

- Checkpointing
  - The master periodically instructs the workers to save the state of their partitions to persistent storage
    - e.g., Vertex values, edge values, incoming messages
- Failure detection
  - Using regular “ping” messages

Fault Tolerance (2/2)

- Recovery
  - The master reassigns graph partitions to the currently available workers
  - All workers reload their partition state from most recent available checkpoint
-

Applications

- PageRank
- Community Detection
- Graph properties computation
  - cluster coefficient
  - core
  - centrality
- Path and Reachability
- etc.

Conclusion

- “Think like a vertex” computation model
- Master – single point of failure ?
- Combiner, Aggregator, topology mutation enables more algorithms to be transformed into Pregel

THANK YOU

ANY QUESTIONS?

References

[1] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry, *Challenges in Parallel Graph Processing*. Parallel Processing Letters 17, 2007, 5-20.

[2] Kameshwar Munagala and Abhiram Ranade, *I/O-complexity of graph algorithms.* in Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms, 1999, 687-694.

[3] Grzegorz Malewicz , Matthew H. Austern , Aart J.C Bik , James C. Dehnert , Ilan Horn , Naty Leiser , Grzegorz Czajkowski, Pregel: a system for large-scale graph processing, Proceedings of the 2010 international conference on Management of data, 2010

[4] Leslie G. Valiant, *A Bridging Model for Parallel Computation.* Comm. ACM 33(8), 1990, 103-111.