Distributed Graph Analysis

COMP9312_23T2

UNSW SYDNEY

Distributed Processing is Non-Trivial

How to assign tasks to different workers in an efficient way?

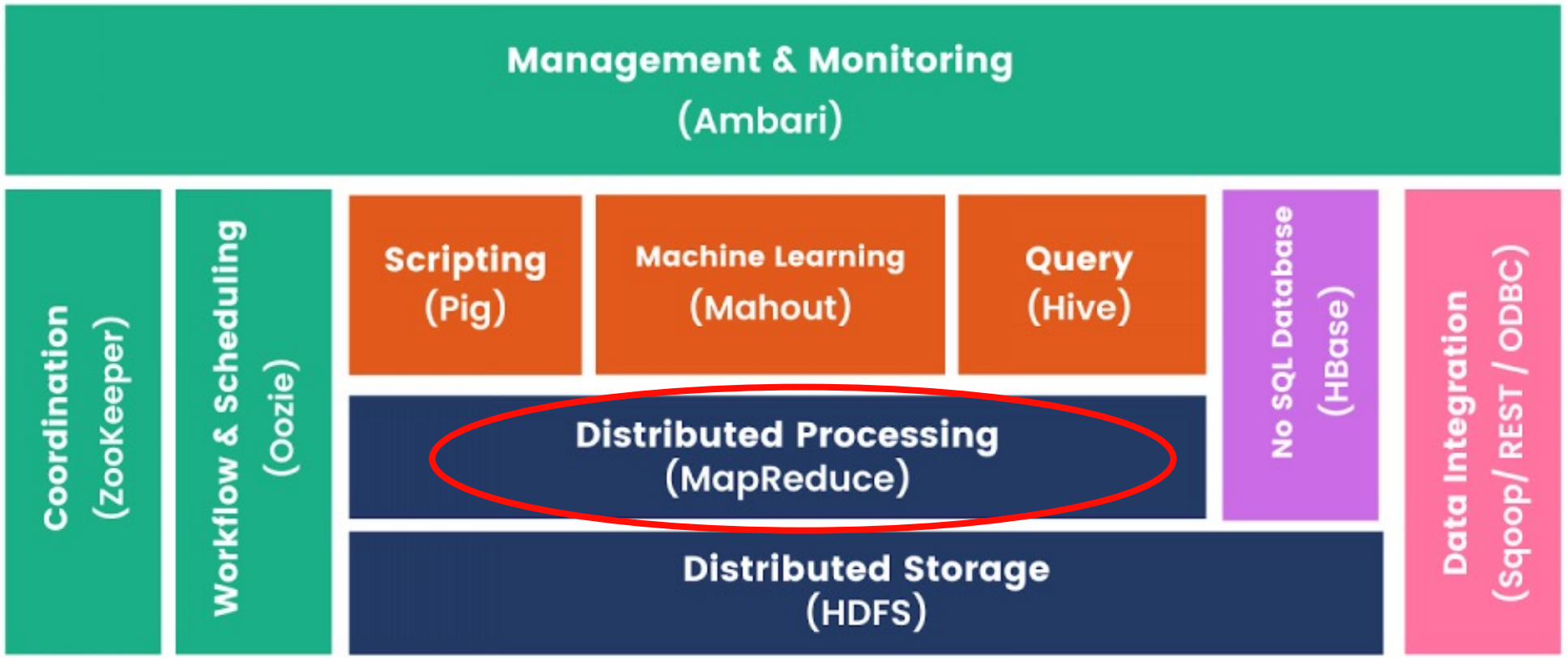
What happens if tasks fail?

How do workers exchange results?

How to synchronize distributed tasks allocated to different workers?

Big Data Tools

Apache Hadoop Ecosystem



We focus on the distributed computing model and algorithms.
Explore more in COMP9313~

MapReduce for General Big Data Processing

Origin from Google

[OSDI'04] MapReduce: Simplified Data Processing on Large Clusters

Programming model for parallel data processing

For large-scale data processing

- Exploits large set of commodity computers
- Executes process in a distributed manner
- Offers high availability

Hadoop MapReduce is an implementation of MapReduce

- MapReduce is a computing paradigm (Google)
- Hadoop MapReduce is an open-source software

Typical Big Data Problem

Iterate over a large number of records

Extract something of interest from each Map

Shuffle and sort intermediate results

Aggregate intermediate results Reduce

Generate final output

Key idea: provide a functional abstraction for these two operations

Idea of MapReduce

Inspired by the map and reduce functions in functional programming

We can view map as a transformation over a dataset

- This transformation is specified by the function *f*
- Each functional application happens in isolation
- The application of *f* to each element of a dataset can be parallelized in a straightforward manner

We can view reduce as an aggregation operation

- The aggregation is defined by the function *g*
- Data locality: elements in the list must be “brought together”
- If we can group elements of the list, also the reduce phase can proceed in parallel

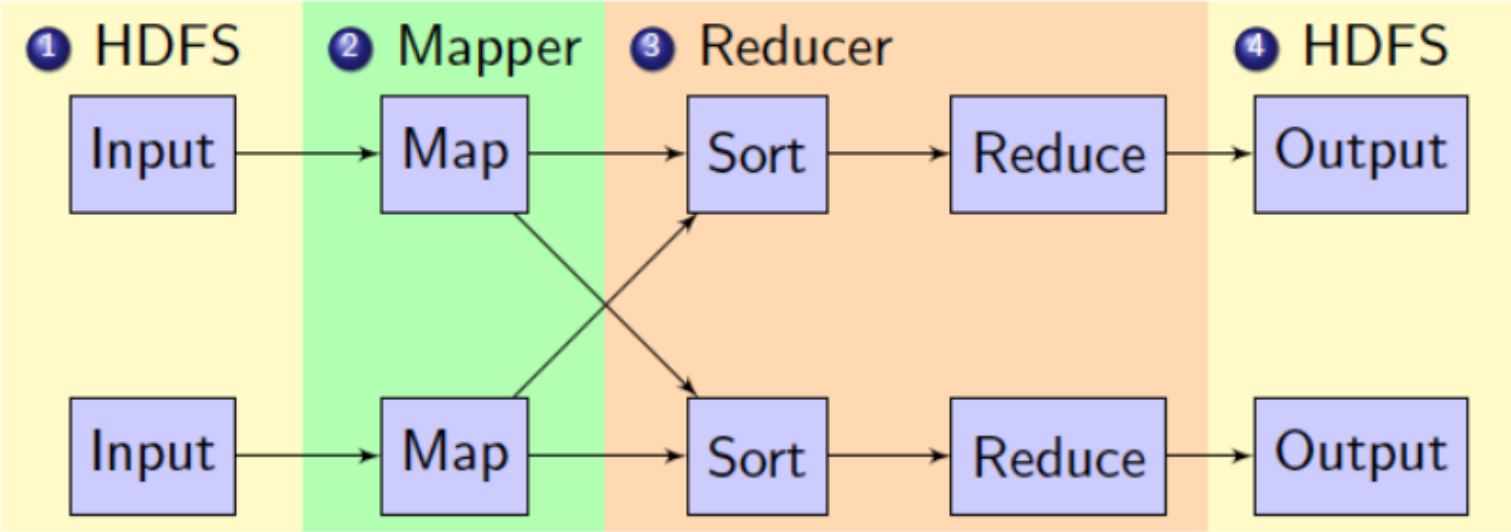
The framework coordinates the map and reduce phases:

- Grouping intermediate results happens in parallel

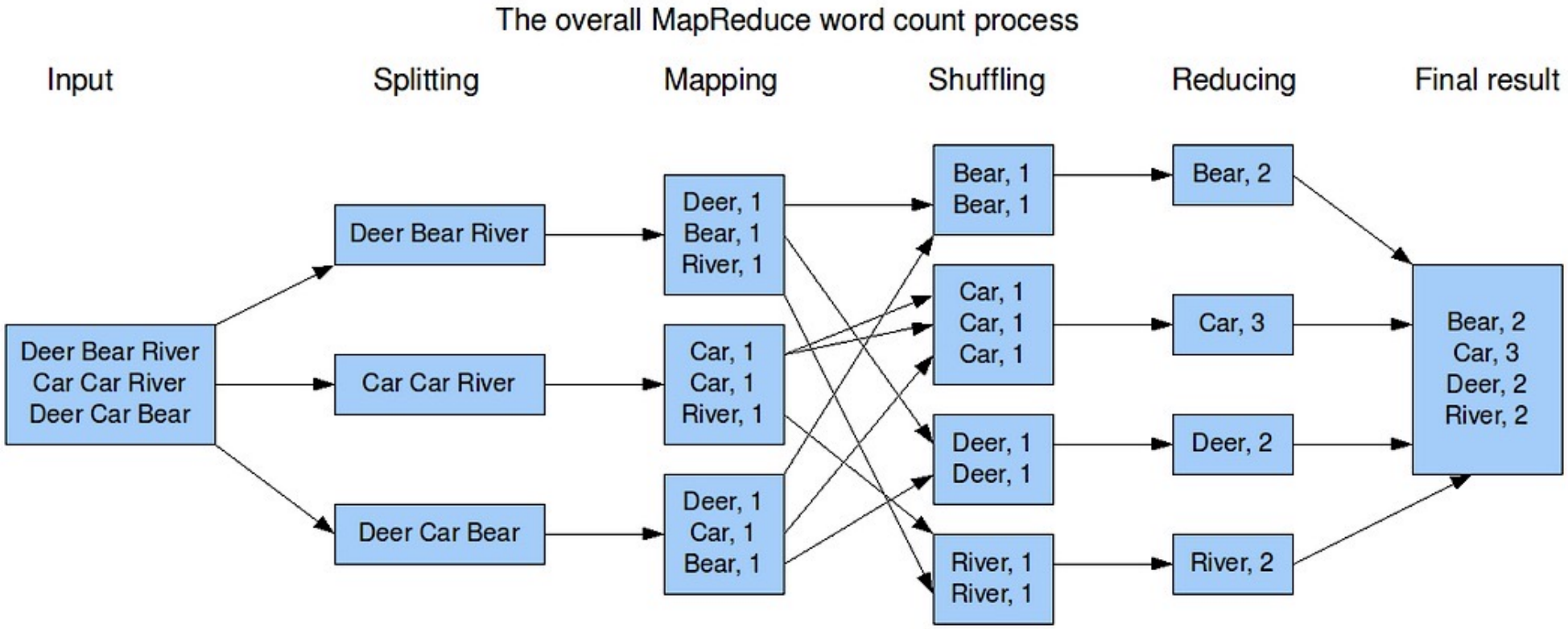
MapReduce Data Flow in Hadoop

1. Mappers read from HDFS
2. Map output is partitioned by key and sent to Reducers
3. Reducers sort input by key
4. Reduce output is written to HDFS

Intermediate results are stored on local FS of Map and Reduce workers



MapReduce Example - WordCount



BFS (SSSP for Unweighted Graphs)

```
1  void Graph::breadth_first_traversal( Vertex first ) {
2      bool<Vertex> visited(|V|, false);
3      visited[first] = true;
4      queue<Vertex> q;
5      q.push( first );
6
7      while ( !q.empty() ) {
8          Vertex v = q.front();
9          q.pop();
10         print the vertex v;
11         for ( Vertex w : v→adjacent_vertices() ) {
12             if ( ! visited[w] ) {
13                 visited[w] = true;
14                 q.push( w );
15             }
16         }
17     }
18 }
19
```

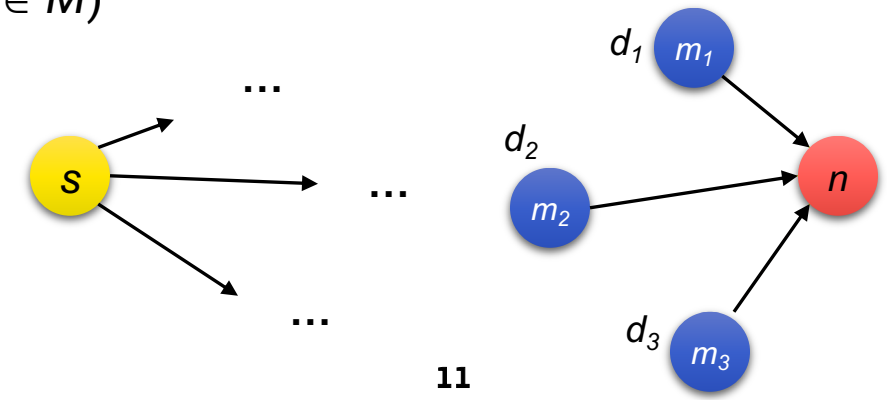
Intuition for parallel BFS

We consider unweighted graphs (BFS) below for simplicity.

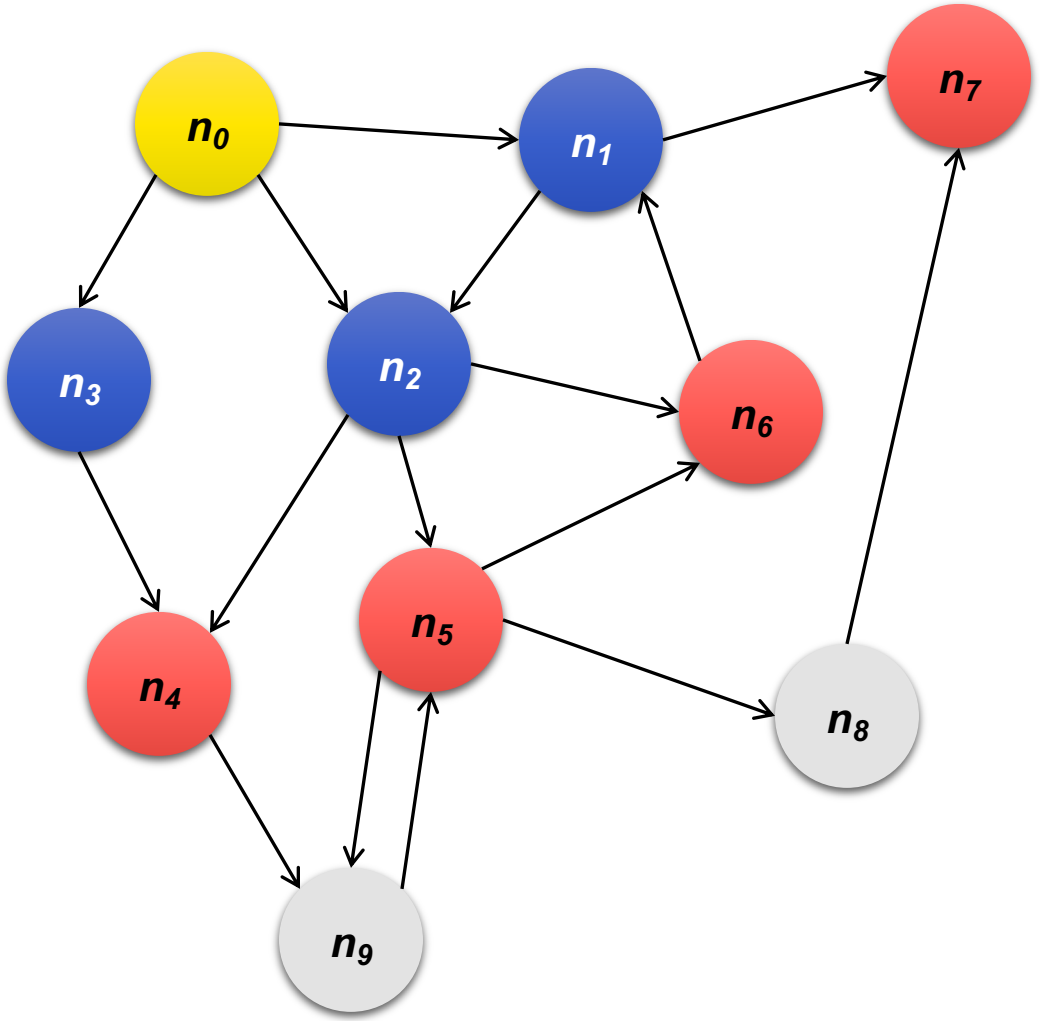
Solution to the problem can be defined inductively

Here's the intuition:

- DISTANCETO(s) = 0
- For all neighbors p of s, DISTANCETO(p) = 1
- For every node u which is the neighbor of some other set of nodes M, DISTANCETO(u) = 1 + min(DISTANCETO(m), $m \in M$)



Visualizing Parallel BFS



From Intuition to Algorithm

Data representation:

- Key: node *n*
- Value: *d* (distance from start), adjacency list (list of nodes reachable from *n*)
- Initialization: for all nodes except for start node, $d = \infty$

Mapper:

- $\forall m \in$ adjacency list: emit $(m, d + 1)$

Sort/Shuffle

- Groups distances by reachable nodes

Reducer:

- Selects minimum distance path for each reachable node
- Additional bookkeeping needed to keep track of actual path

BFS in MapReduce

```
class Mapper
    method Map(nid n, node N)
    d ← N.Distance
    Emit(nid n,N.AdjacencyList)              //Pass along graph structure
    for all nodeid m ∈ N.AdjacencyList do
        Emit(nid m, d+1)                     //Emit distances to reachable nodes
```

```
class Reducer
    method Reduce(nid m, [d1, d2, . . .])
    d_min←∞
    M ← ∅
    for all d ∈ counts [d1, d2, . . .] do
        if IsNode(d) then
            M.AdjacencyList ← d              //Recover graph structure
        else if d < d_min then               //Look for shorter distance
            d_min ← d
    M.Distance ← d_min                       //Update shortest distance
    Emit(nid m, node M)
```

Multiple Iterations Needed

The input of Mapper is the output of Reducer in the previous iteration.

Multiple iterations are needed to explore entire graph.

Preserving graph structure:

- Problem: Where did the adjacency list go?
- Solution: mapper emits (*n*, adjacency list) as well

MapReduce for Graphs?

Graph Computing Paradigm:

```
For each vertex:
      do something;
      notify
neighbors;
```

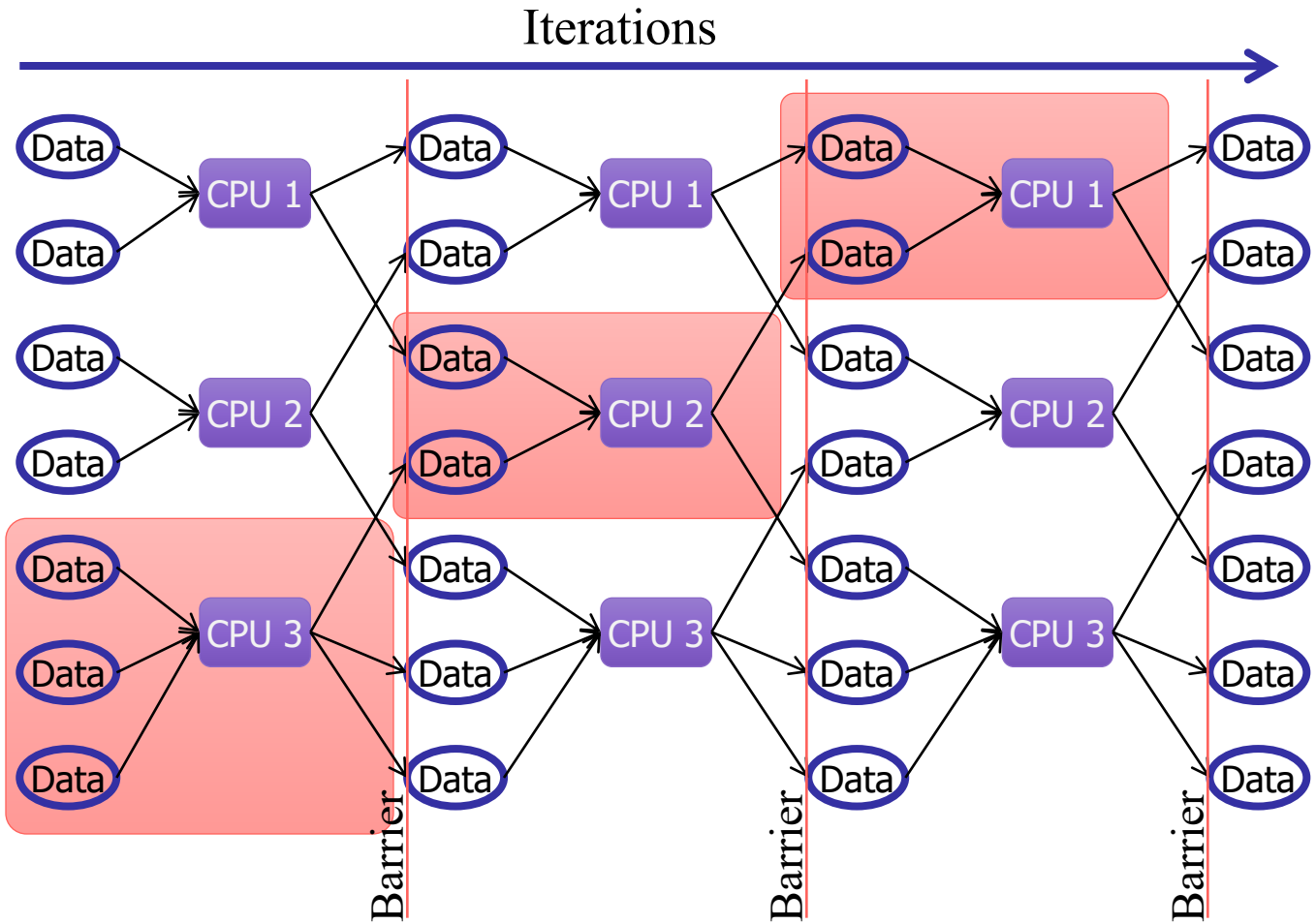
Graph algorithms are expressed in multiple MR iterations.

Data must be reloaded and reprocessed at each iteration.

Need an extra MR Job for each iteration to detect termination condition.

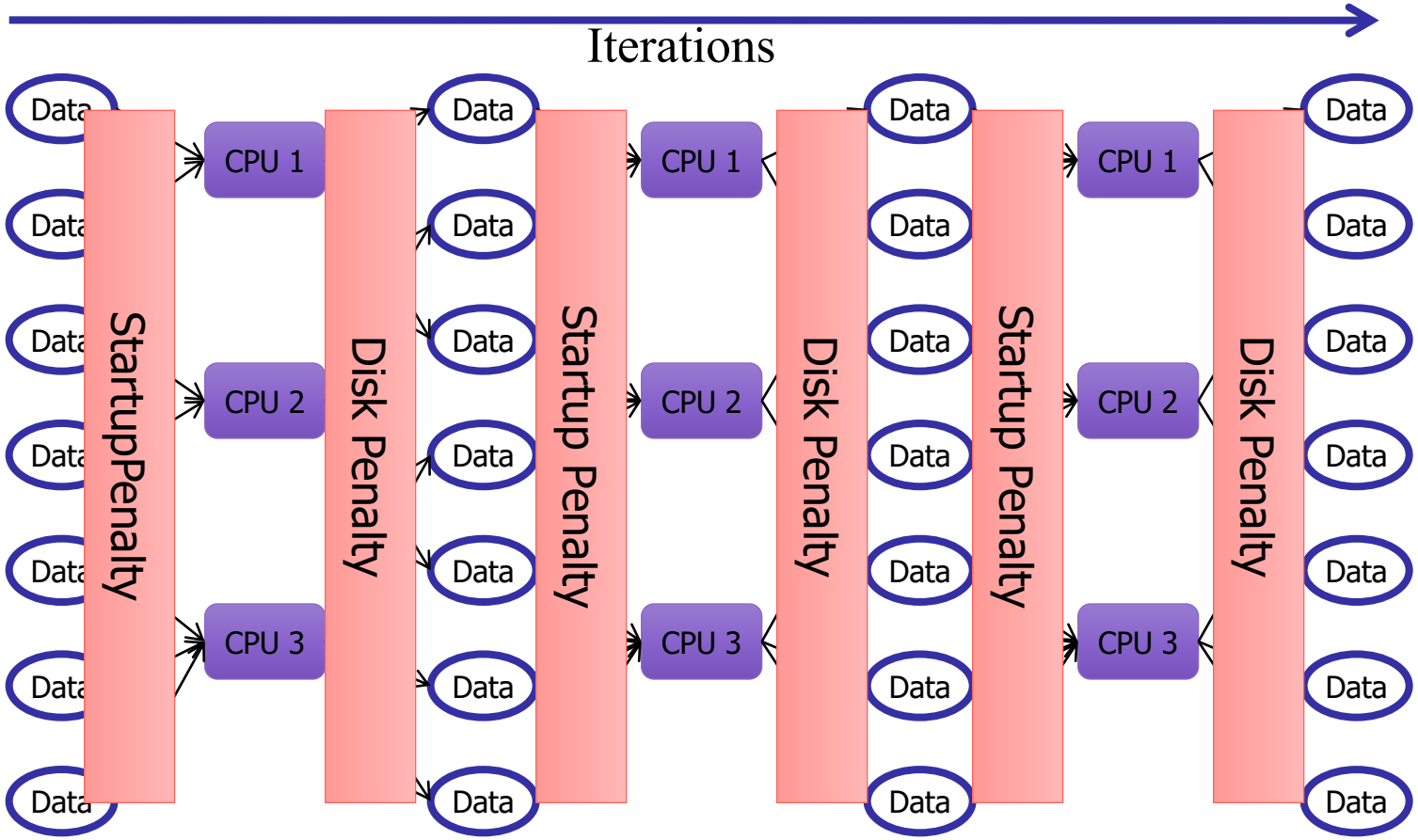
Workload unbalance by various vertex degree

Iterative MapReduce



- Only a subset of data needs computation:

Iterative MapReduce



- System is not optimized for iteration:

Pregel for Distributed Graph Processing

Developed by Google

Computing in Bulk Synchronous Parallel (BSP) model

Computes in vertex-centric fashion

Scalable and fault tolerant

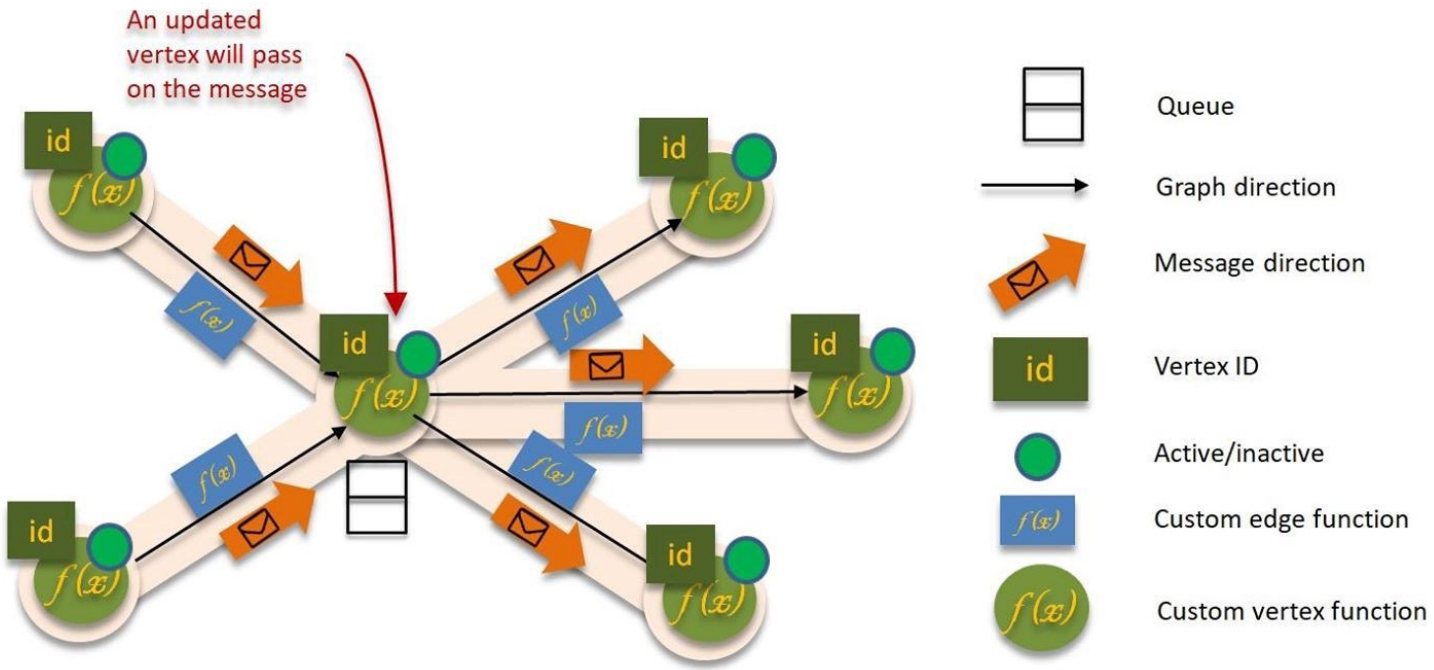
Reference: https://research.google/pubs/pub37252/

Many implementations:

Apache Giraph (Java), Pregel ++ (C/C++), …

Pregel for Distributed Graph Processing

Think Like a Vertex Model



Bulk Synchronous Parallel (BSP)

- Computations are consist of a sequence of iterations, called superstep.

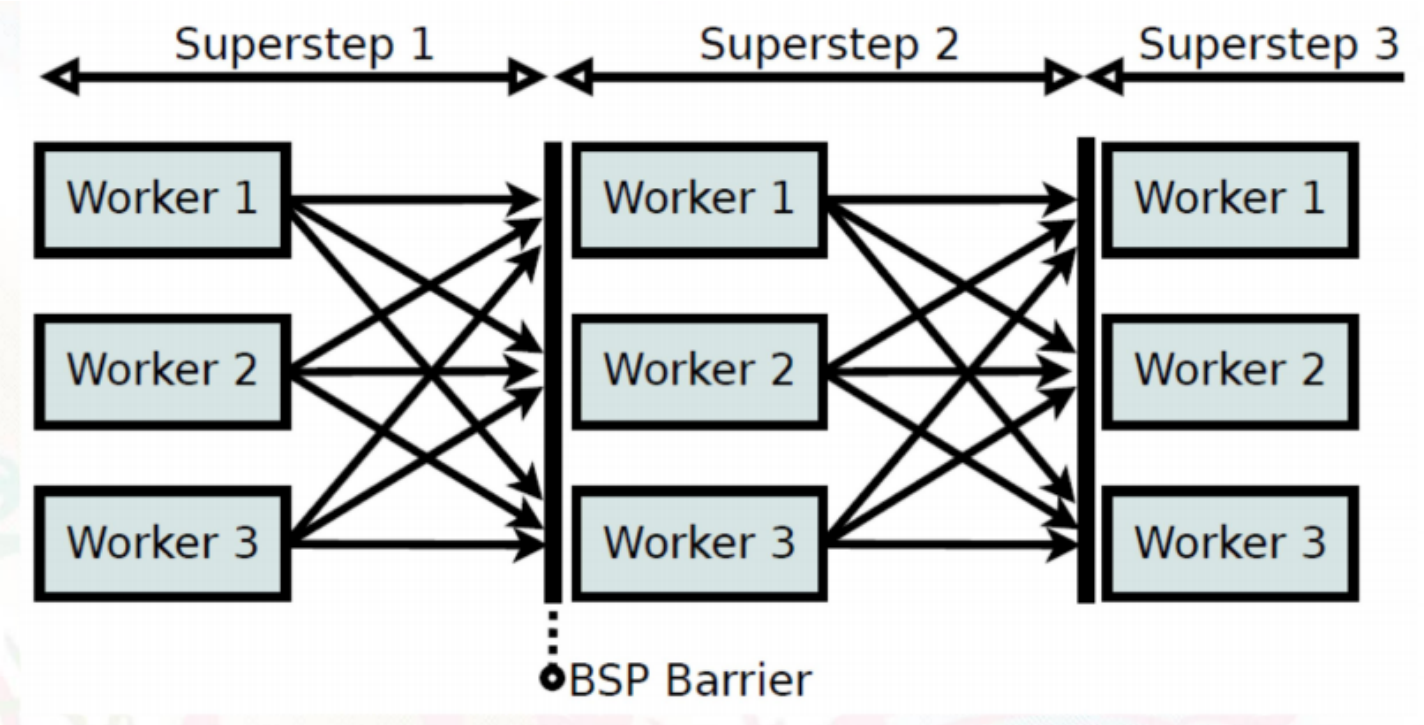
- During superstep, framework calls user-defined computation function on every vertex.

- Computation function specifies behavior at a single vertex V in a superstep.

- Supersteps end with barrier synchronization.

- All communications are from superstep S to superstep S+1.

Bulk Synchronous Parallel (BSP)



Terminates when all vertices are inactive or no messages to be delivered

Vertex in Pregel

- Can mutate local value and value on outgoing edges.
- Can send arbitrary number of messages to any other vertices.
- Receive messages from previous superstep.
- Can mutate local graph topology.
- All active vertices participate in the computation in a superstep.

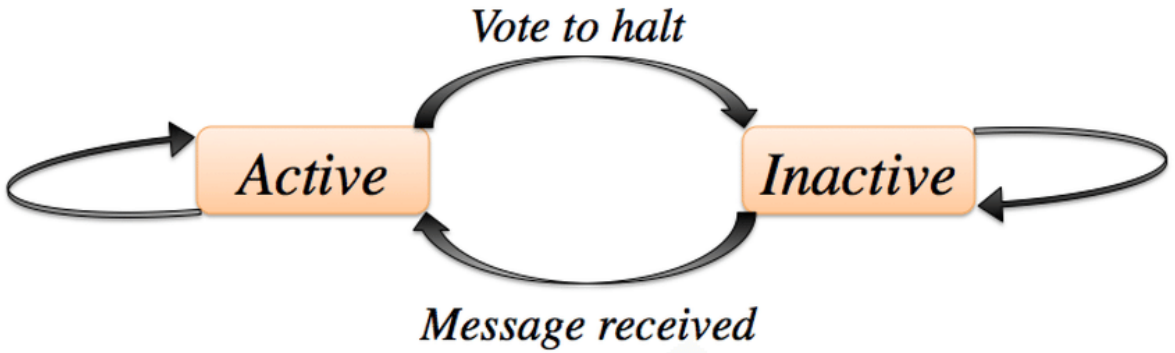
Messages

- Consists of a message value and destination vertex.
- Typically sent along outgoing edges.
- Can be sent to any vertex whose identifier is known.
- Are only available to receiver at the beginning of superstep.
- Guaranteed to be delivered.
- Guaranteed not to be duplicated.
- Can be out of order.

Vertex in Pregel

Initially, every vertices are active.

- A vertex can deactivate itself by vote to halt.
- Deactivated vertices don't participate in computation.
- Vertices are reactivated upon receiving message.



The C++ API of Pregel

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

Override this!

in msgs

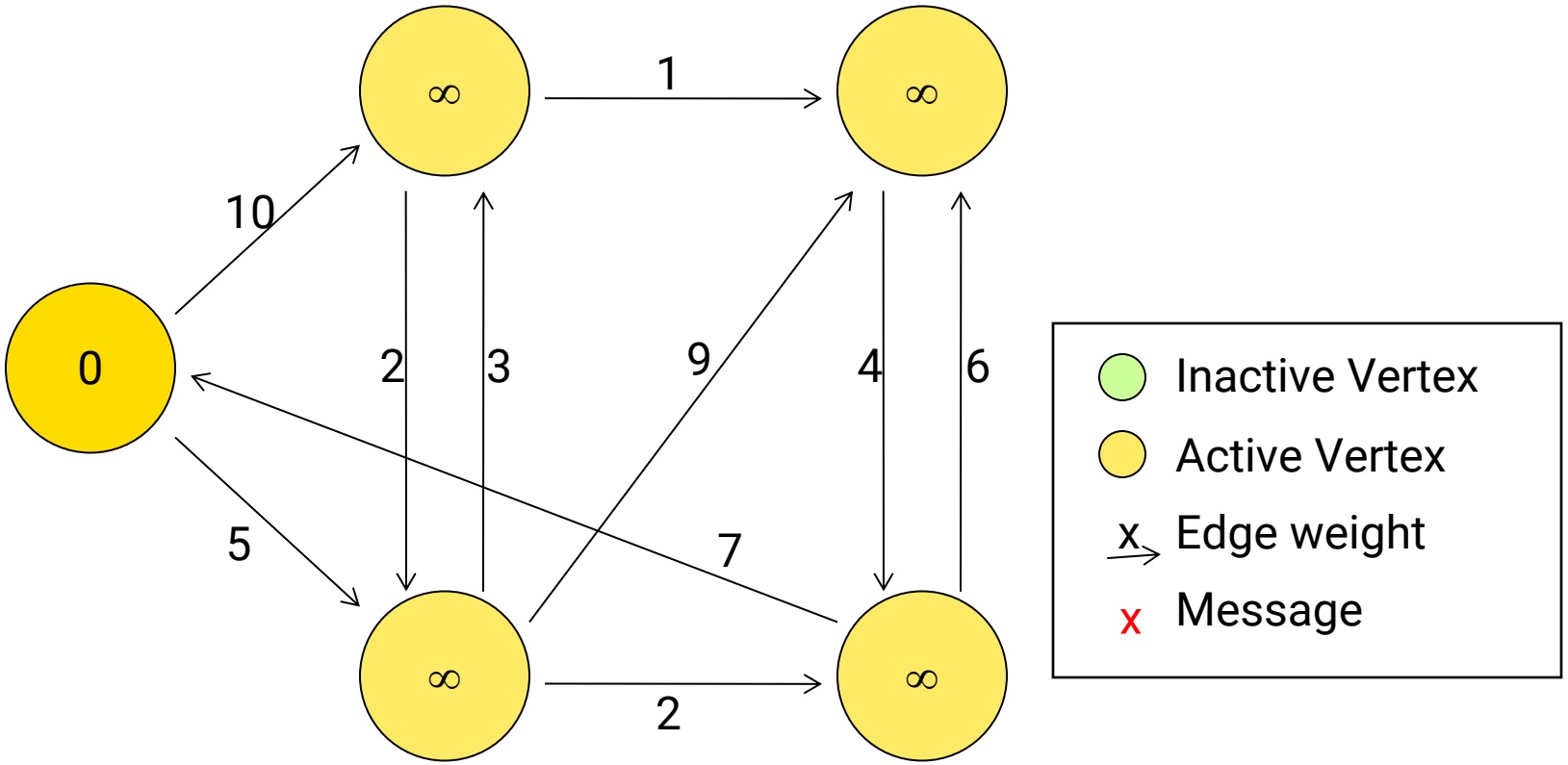
Modify vertex value

out msg

Implementation for SSSP

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

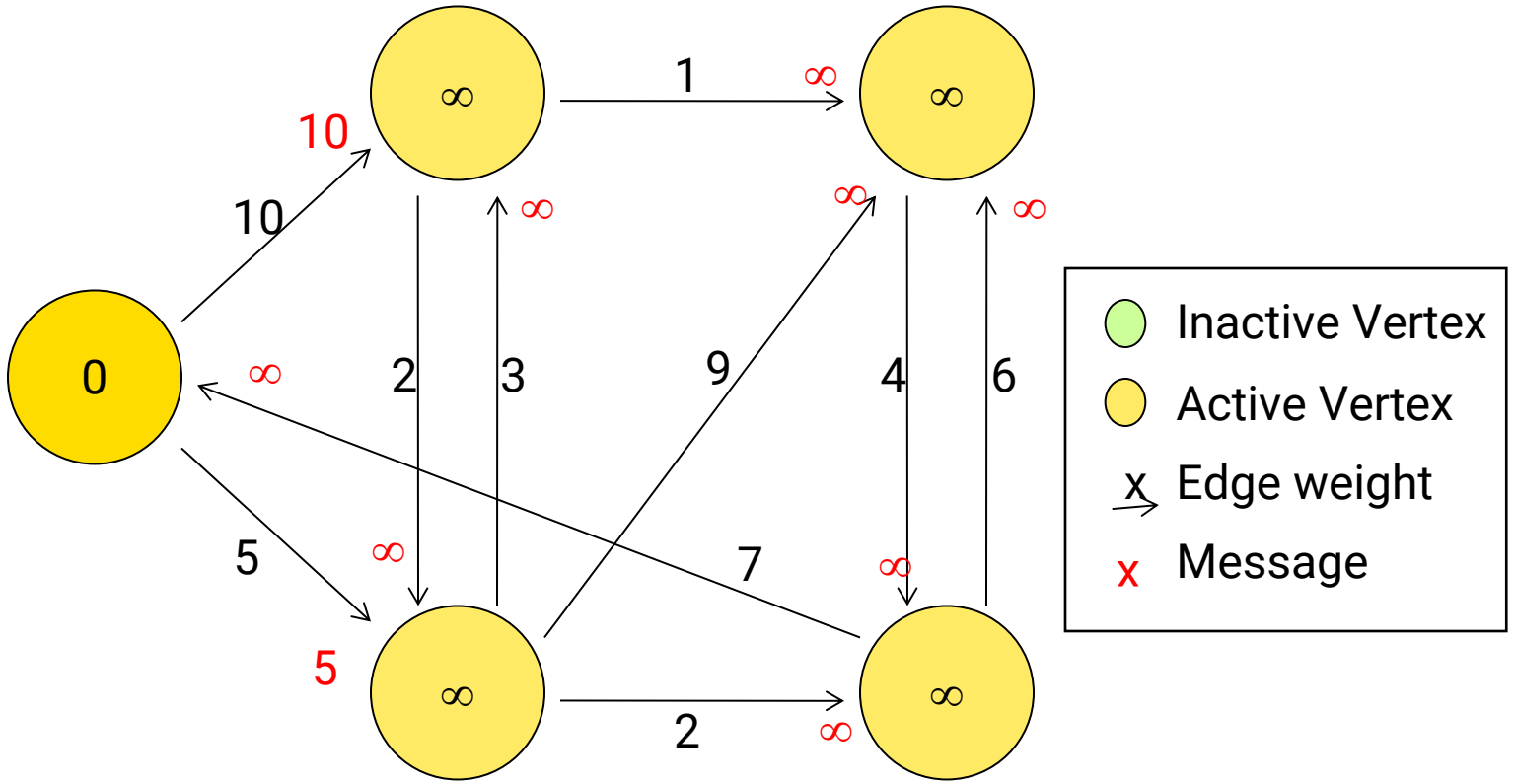
Single-Source Shortest Path (SSSP) in Pregel



0 is the source vertex

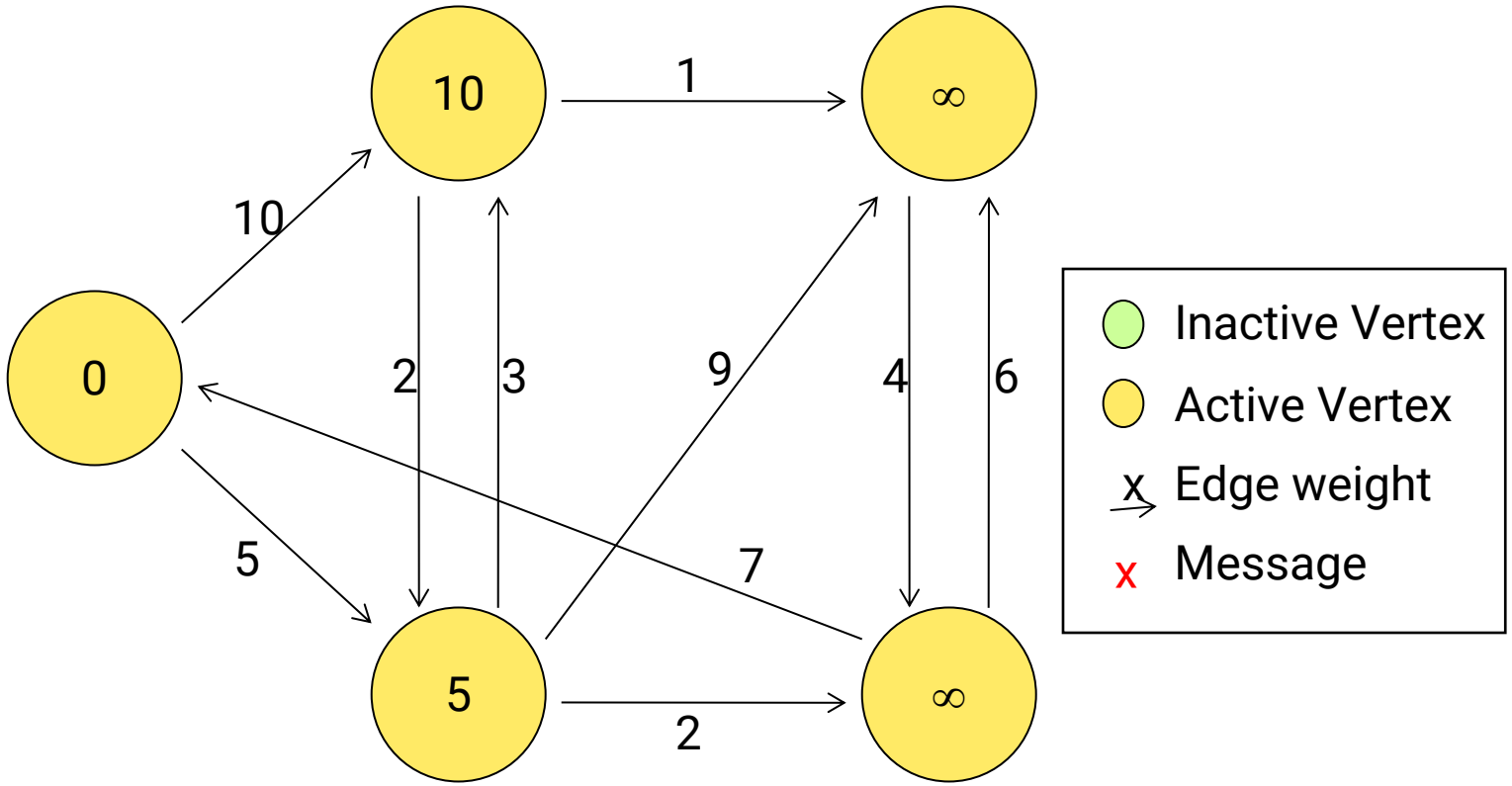
Superstep 1

SSSP in Pregel



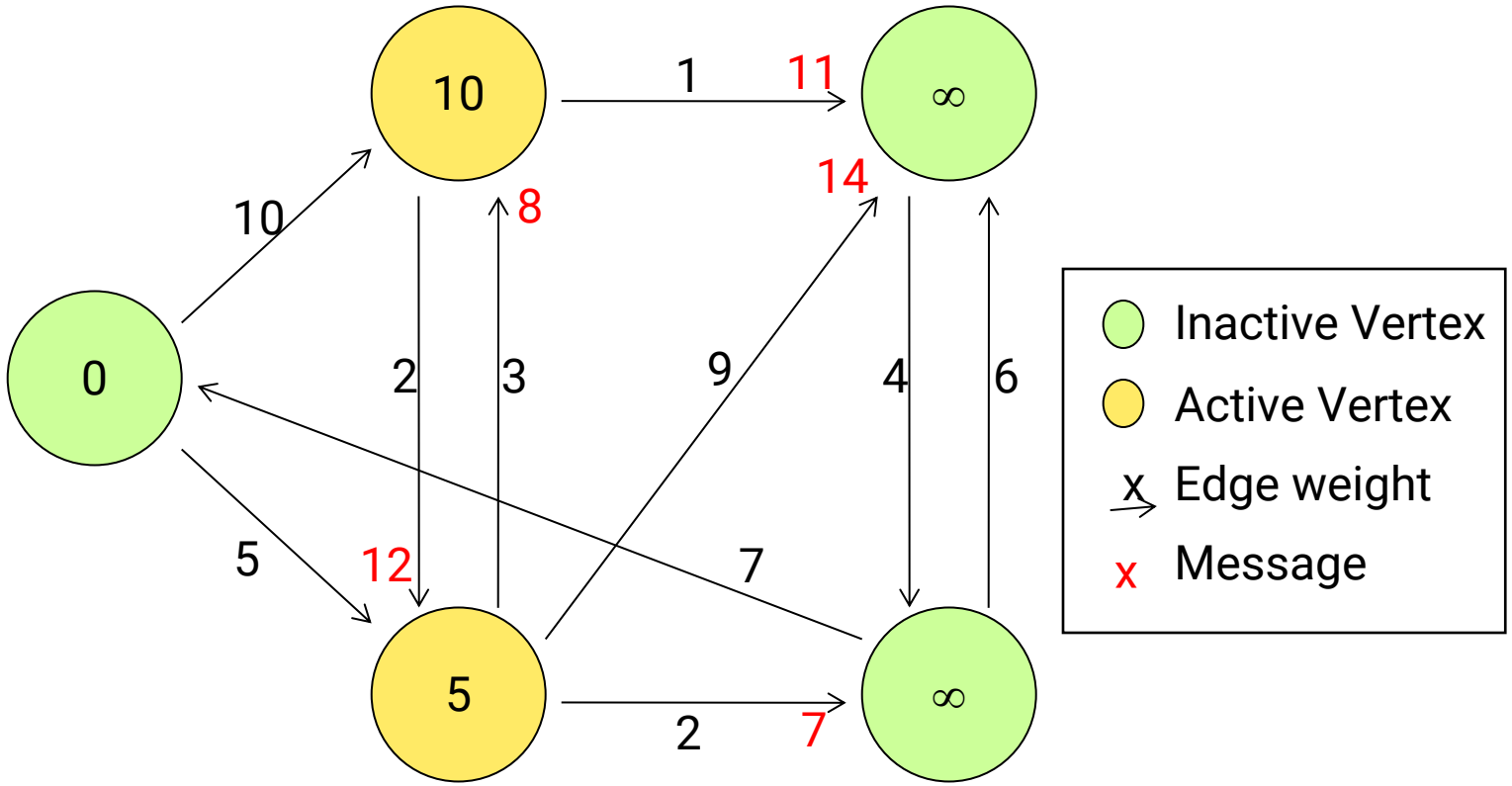
Superstep 1

SSSP in Pregel



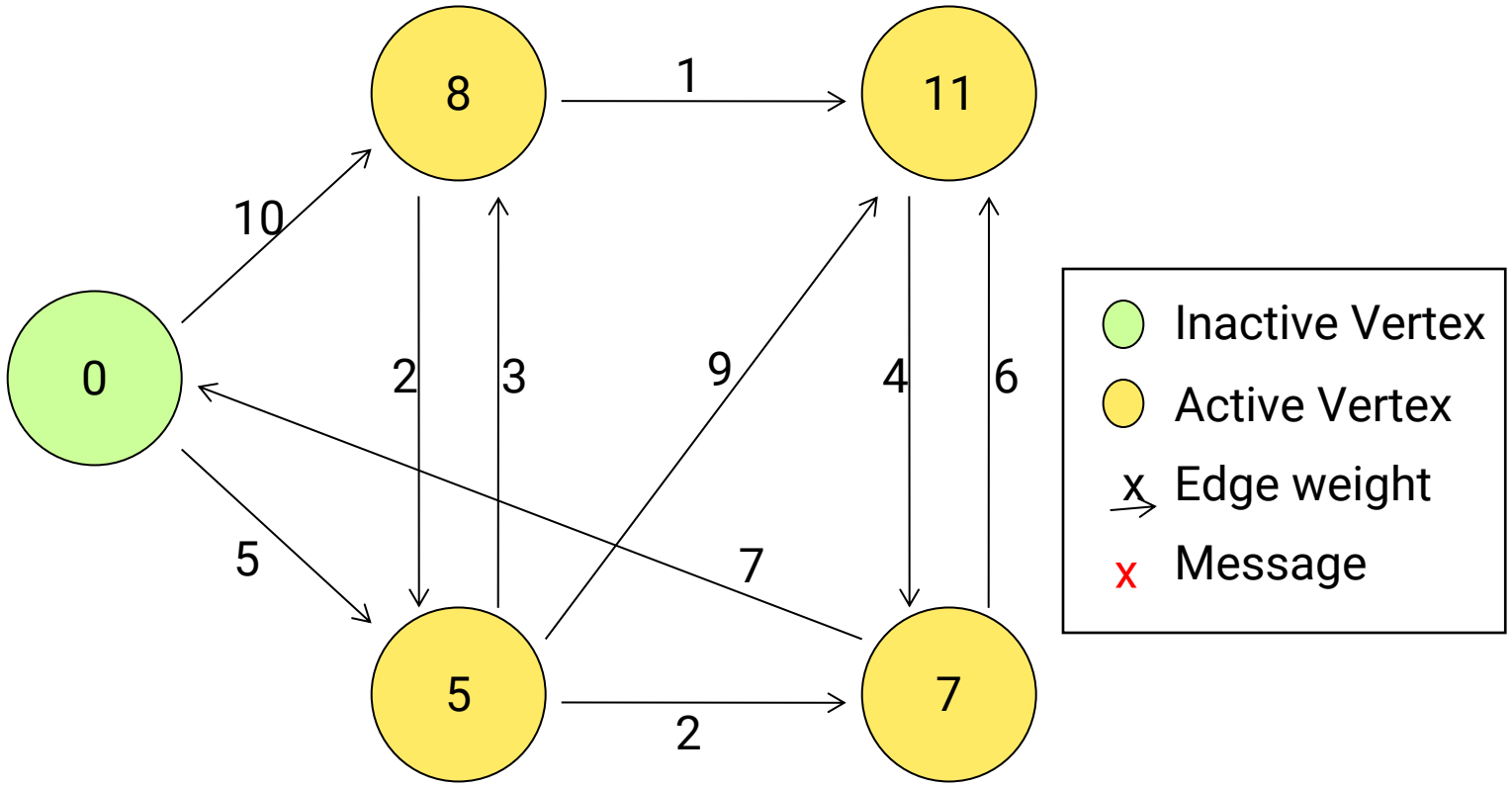
Superstep 2

SSSP in Pregel



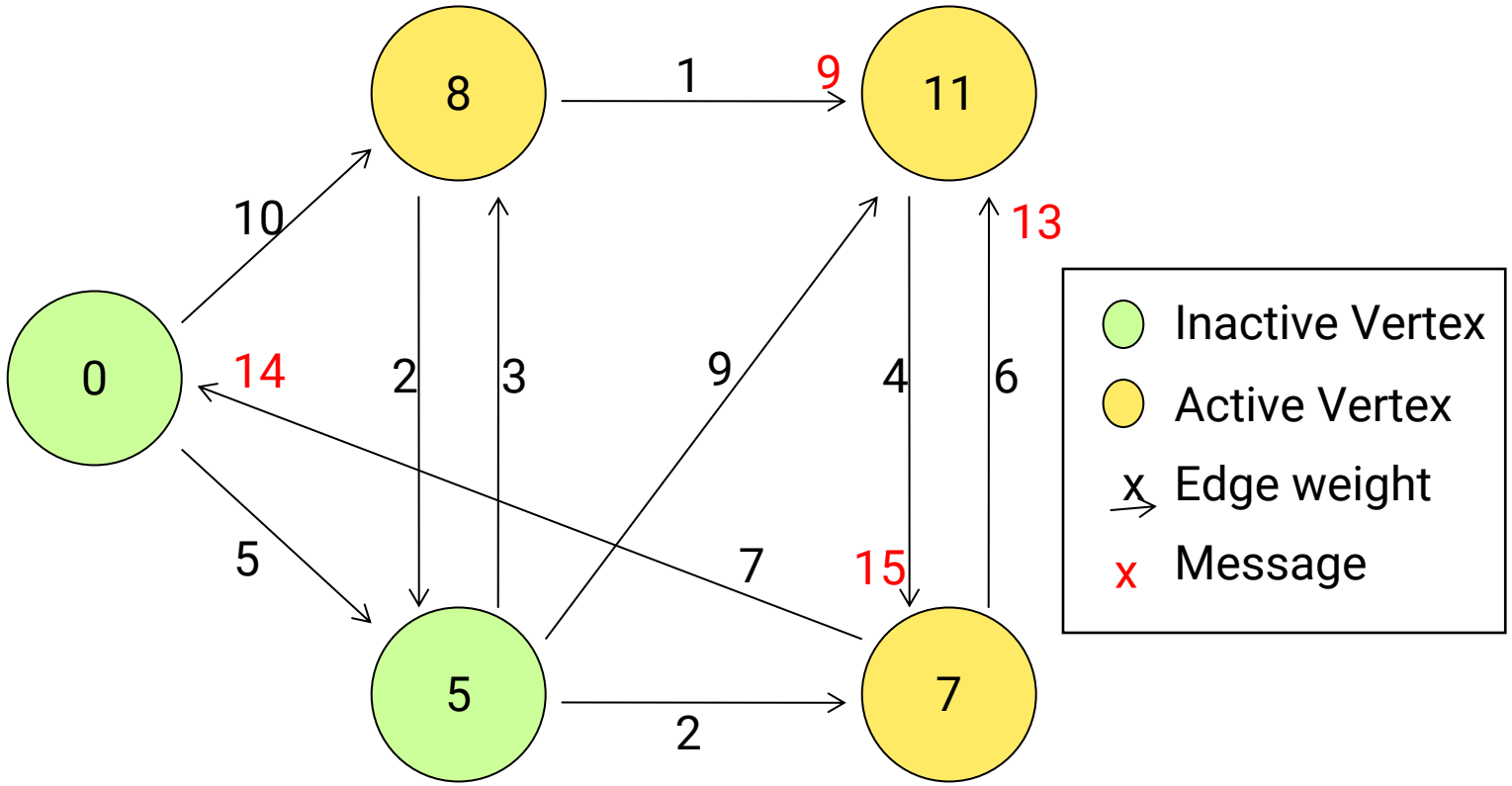
Superstep 2

SSSP in Pregel



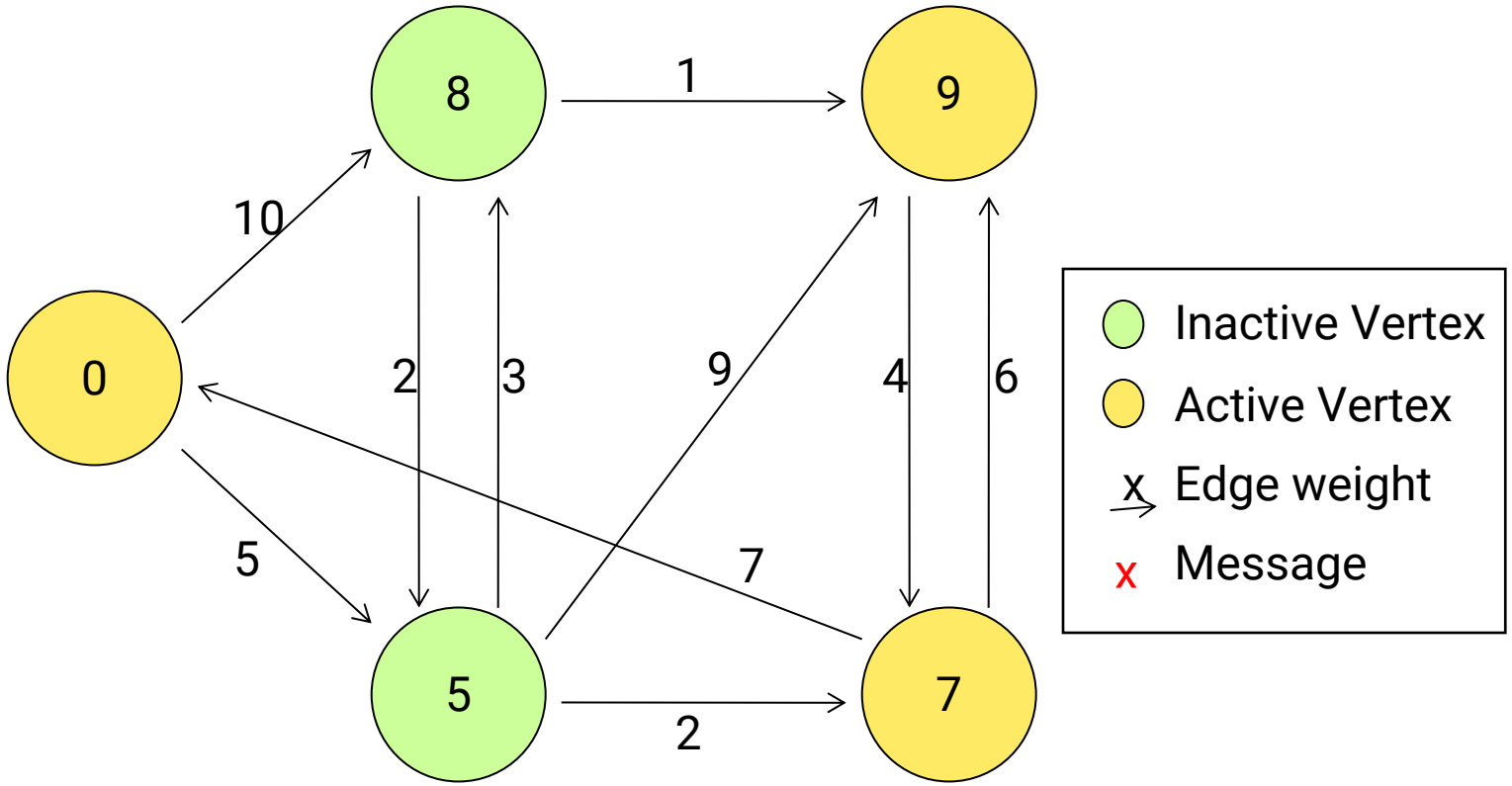
Superstep 3

SSSP in Pregel



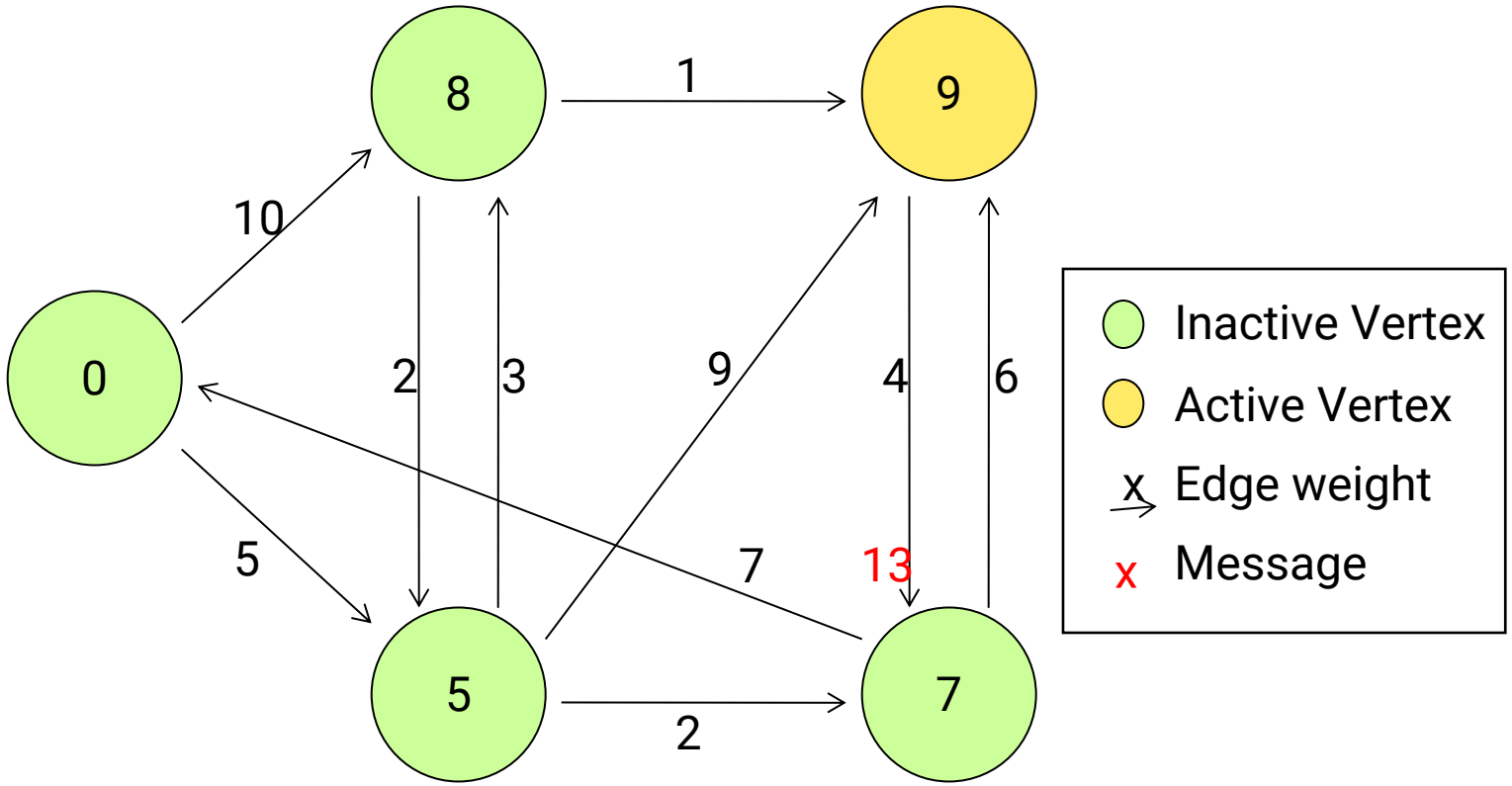
Superstep 3

SSSP in Pregel



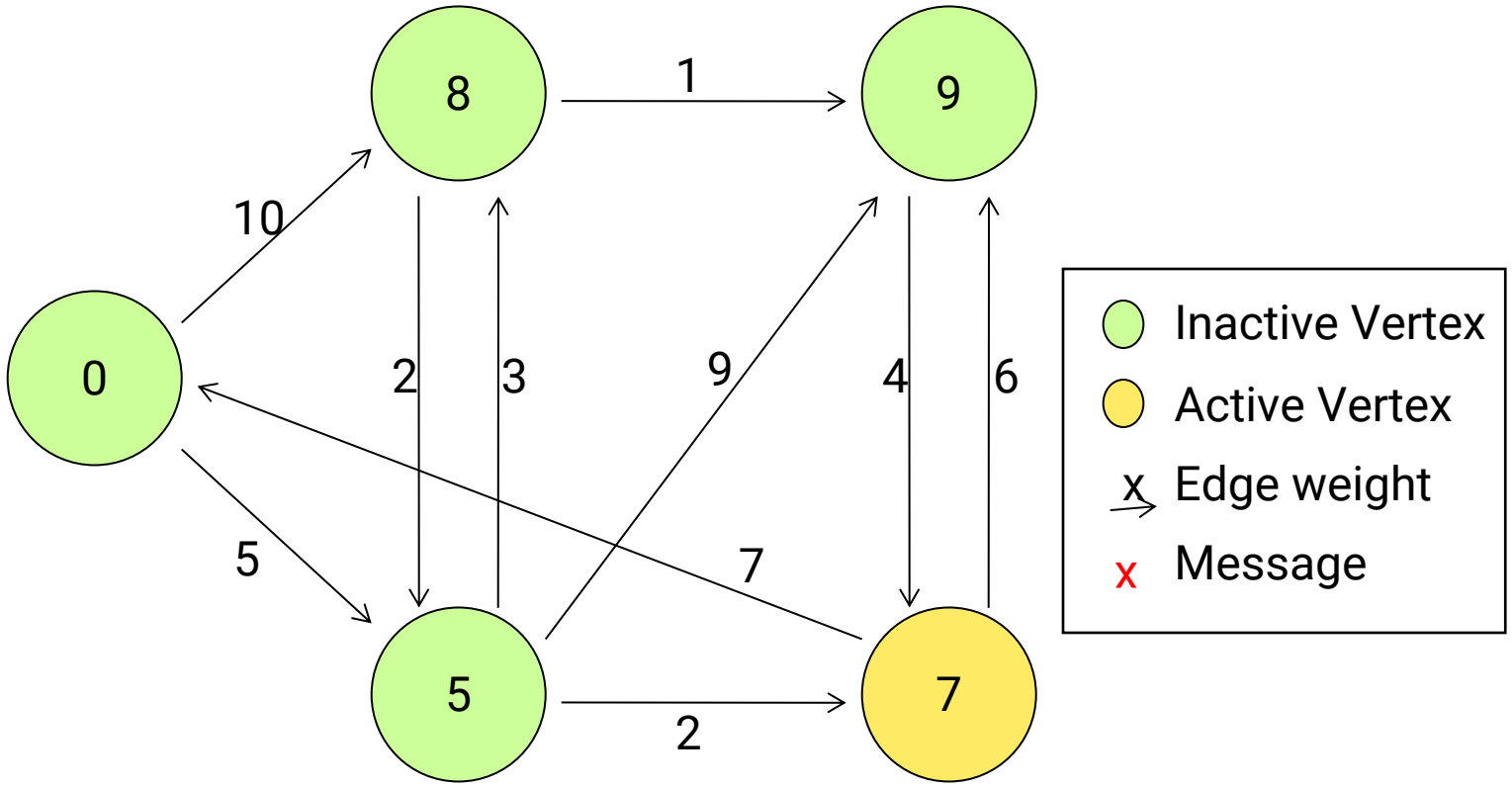
Superstep 4

SSSP in Pregel



Superstep 4

SSSP in Pregel

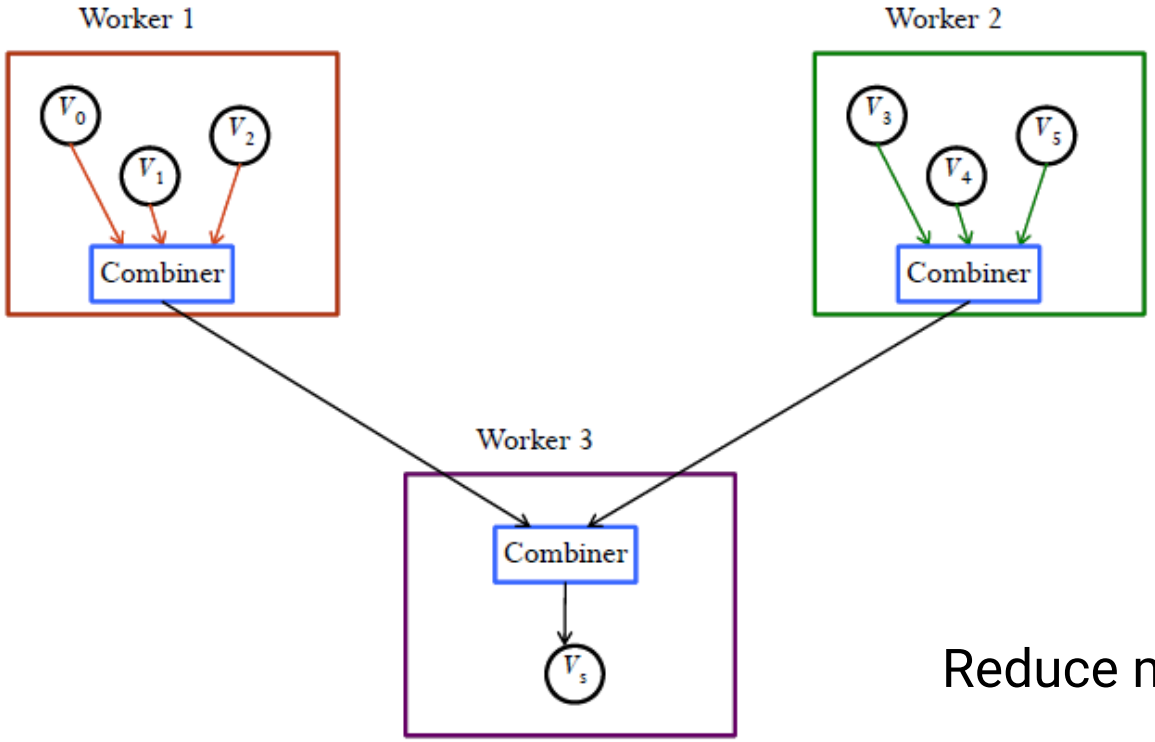


Superstep 5

Combiner in Pregel

- Sending messages incurs overhead.
- System calls Combine() for several messages intended for a vertex into a single message containing the combined message.
- No guarantees which messages will be combined or the order of combination.
- Should be enabled for commutative and associative messages.
- Not enabled by default.

Combiner in Pregel (Cont.)



Reduce message traffic and disk space

Combiner in SSSP

```
class MinIntCombiner : public Combiner<int> {
        virtual void Combine(MessageIterator* msgs) {
                int mindist = INF;
                for (; !msgs->Done(); msgs->Next())
                        mindist = min(mindist, msgs->Value());
                Output("combined_source", mindist);
        }
};
```

Others in Pregel

- Aggregator
  - Used for global communication, global data and monitoring
- Topology Mutations
- Fault Tolerance
  - Checking point, failure detection, recovery . . .

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)

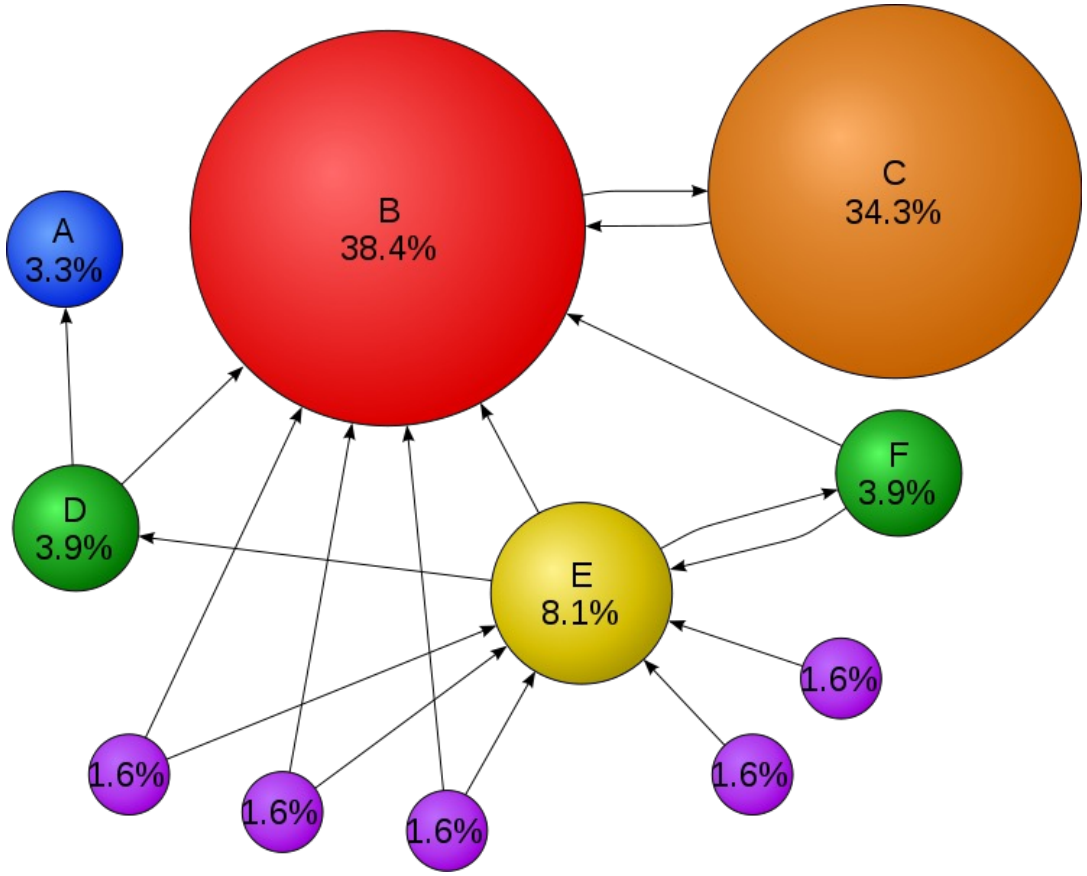
d = Damping factor between 0 and 1 (usually kept as 0.85)

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

PR(A) = the PageRank of page A (initialized as 1/N for each page)

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right)$$

Pagerank



Pagerank in Pregel

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

```
virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
                double sum = 0;
                for (; !msgs->done(); msgs->Next())
                        sum += msgs->Value();
                *MutableValue() = 0.15/NumVertices() + 0.85 * sum;
        }

        if (supersteps() < 30) {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue() / n);            }
  else {
                VoteToHalt();
        }
}
```

Combiner for Pagerank?

```
virtual void Combine(MessageIterator* msgs) {




}
```

Combiner for Pagerank?

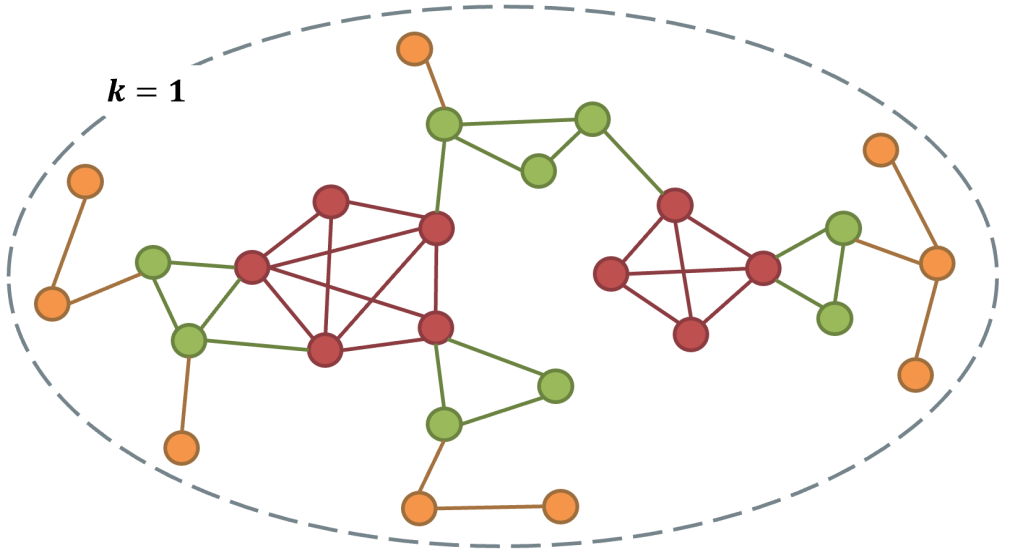
```
virtual void Combine(MessageIterator* msgs) {

            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                  sum += msgs->value();
            Output("combined_source", mindist);

}
```

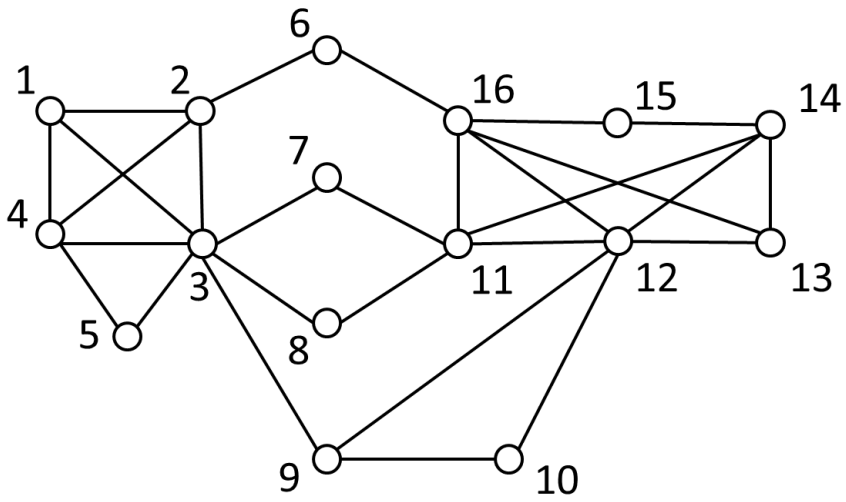
Core Decomposition

```
For each unvisited vertex u with the
lowest degree in G
        assign core(u) as degree(u);
        mark u as visited;
        decrease the degree of its
        unvisited neighbors with higher
        degree than u by 1;
```



The peeling algorithm for core decomposition is hard to parallelize.

Core Decomposition: Review



Question:
Compute the core number of each vertex

Solution:
Iteratively remove the vertex with the smallest degree

index	d	b	D	p
1	3	0	5	7
2	4	1	6	10
3	7	7	7	16
4	4	10	8	11
5	2	13	10	1
6	2	15	15	2
7	2	16	1	3
8	2		9	4
9	3		13	8
10	2		2	5
11	5		4	13
12	6		14	15
13	3		11	9
14	4		16	12
15	2		12	6
16	5		3	14

```
1:  function K-CORES(Graph G)
2:      initialize(d, b, D, p, G)
3:      for all i ← 1 to n do
4:          v ← D[i]
5:          for all u ∈ N_G(v) do
6:              if d[u] > d[v] then
7:                  du ← d[u], pu ← p[u]
8:                  pw ← b[du], w ← D[pw]
9:                  if u ≠ w then
10:                     D[pu] ← w, D[pw] ← u
11:                     p[u] ← pw, p[w] ← pu
12:                 end if
13:                 b[du]++, d[u]−−
14:             end if
15:         end for
16:     end for
17:     return d
18: end function
```

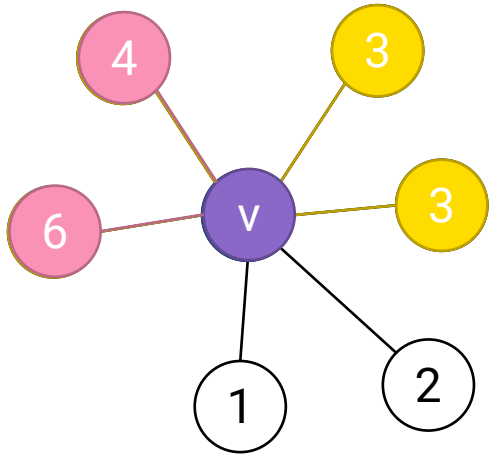
Core Decomposition: Review

Locality Theorem:

Given a vertex and its core number k:

There exists at least k neighbors with core number k;

There does not exist k+1 neighbors with core number k+1.



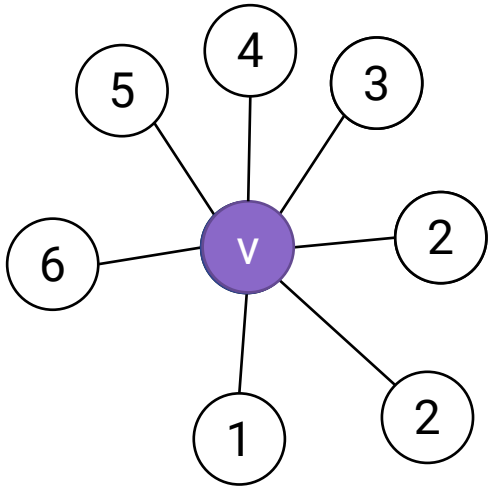
Core(V) = 3 — 4 neighbors with core number at least 3 √

Core(V) = 4 — Only 2 neighbors with core number at least 4 ×

Montresor, Alberto, Francesco De Pellegrini, and Daniele Miorandi. "**Distributed k-core decomposition.**" IEEE Transactions on parallel and distribuated systems 24.2 (2013): 288-300.

Quiz

What is the core number of v?



Core Decomposition Pregel

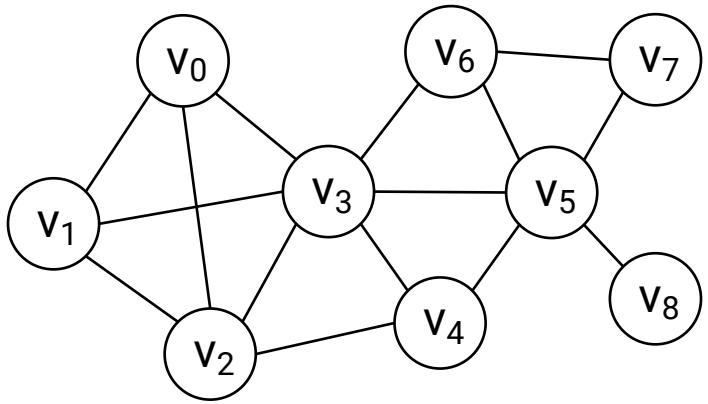
```
virtual void Compute(MessageIterator* msgs) {

      oldValue = GetValue()

      *MutableValue() = compute core number from the neighbor's core
      numbers in msgs based on the locality theorem


      if (GetValue() != oldValue){
            SendMessageToAllNeighbors(GetValue());
      }
      VotetoHalt();
}
```

Core Decomposition: Local-view (Converging)



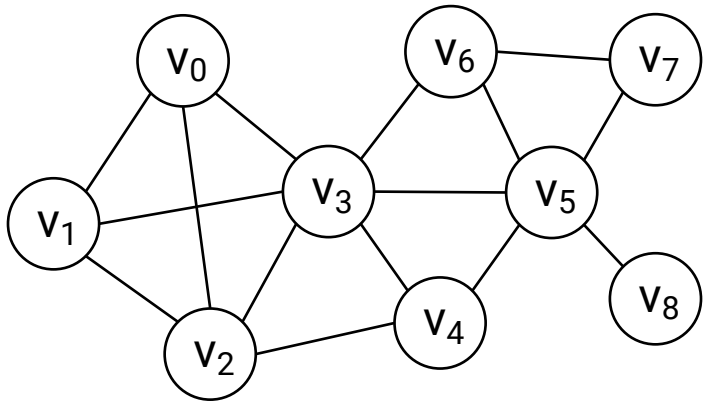
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 1: all vertices are activated

ID	0	1	2	3	4	5	6	7	8
Core	3	3	4	6	3	5	3	2	1

Core Decomposition: Local-view (Converging)



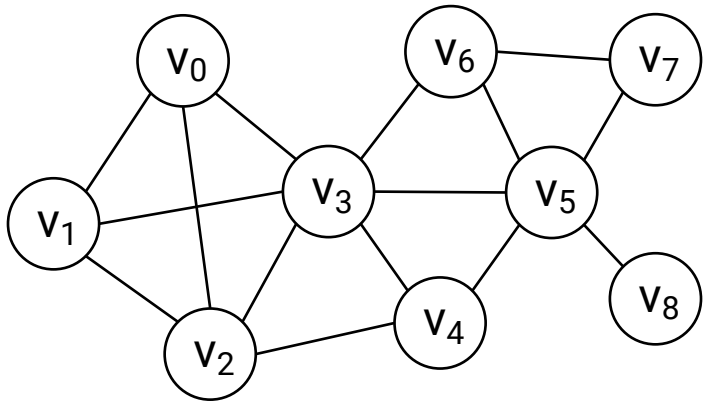
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 1

ID	0	1	2	3	4	5	6	7	8
Core	3	3	4	6	3	5	3	2	1

Core Decomposition: Local-view (Converging)



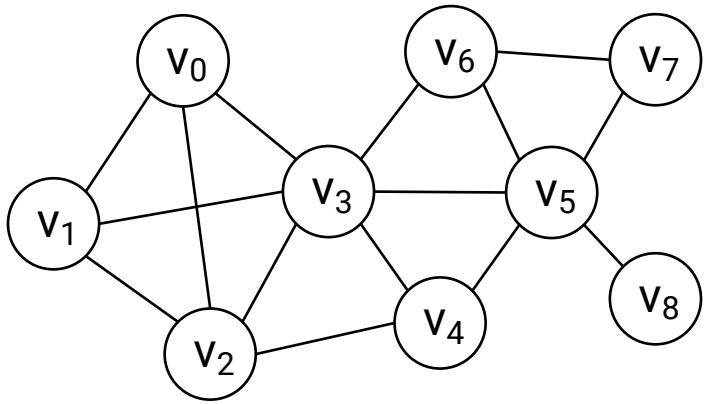
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 1 finishes. All vertices are activated.

ID	0	1	2	3	4	5	6	7	8
Core	3	3	3	3	3	3	2	2	1

Core Decomposition: Local-view (Converging)



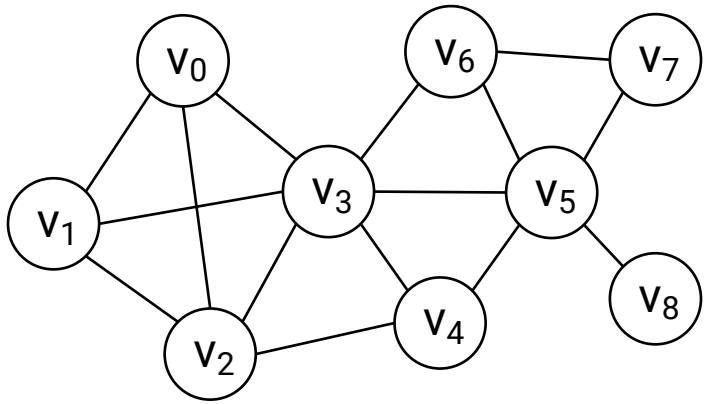
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 2

ID	0	1	2	3	4	5	6	7	8
Core	3	3	3	3	3	3	2	2	1

Core Decomposition: Local-view (Converging)



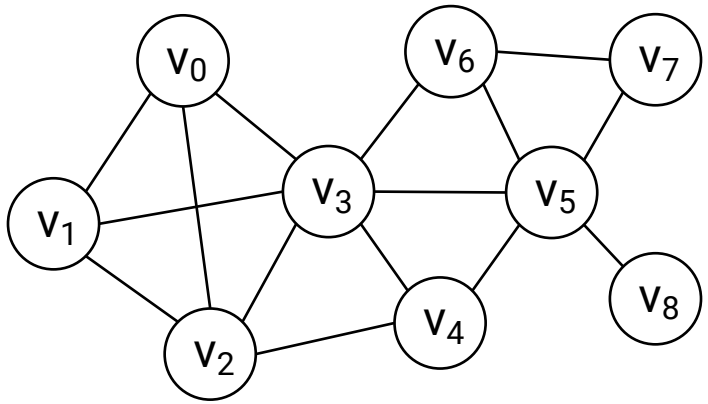
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 2 finishes. v3, v4, v6, v7, and v8 are activated.

ID	0	1	2	3	4	5	6	7	8
Core	3	3	3	3	3	2	2	2	1

Core Decomposition: Local-view (Converging)



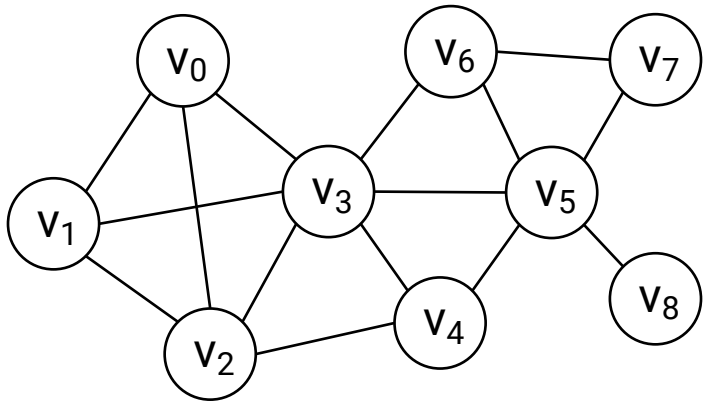
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 3.

ID	0	1	2	3	4	5	6	7	8
Core	3	3	3	3	3	2	2	2	1

Core Decomposition: Local-view (Converging)



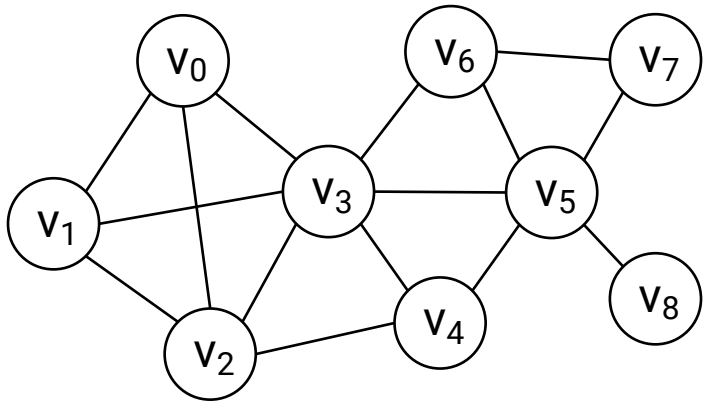
Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Iteration 3 finishes.

ID	0	1	2	3	4	5	6	7	8
Core	3	3	3	3	2	2	2	2	1

Core Decomposition: Local-view (Converging)



Given a vertex and its core number k:
There exists at least k neighbors with core number k;
There does not exist k+1 neighbors with core number k+1.

Initialize the core number by degree

Terminate after iteration 4: no vertex updates

ID	0	1	2	3	4	5	6	7	8
Core	3	3	3	3	2	2	2	2	1

Optimization?

Optional

```
virtual void Compute(MessageIterator* msgs) {

      oldValue = GetValue()

      *MutableValue() = compute core number from the neighbor's core
      numbers in msgs based on the locality theorem


      if (GetValue() != oldValue){
            SendMessageToAllNeighbors(GetValue());
      }
      VotetoHalt();
}
```

Connected Component Detection in Pregel

A basic distributed algorithm to compute connected components:

```
virtual void Compute(MessageIterator* msgs) {
        int minID = GetValue();
        for (; !msgs->done(); msgs->Next())
                minID = min(minID,msg->Value());
        }
        if (minID < GetValue()){
                *MutableValue() = minID;
                 SendMessageToAllNeighbors(minID);
        }
        VoteToHalt();

}
```

Learning Outcome

- Understand the framework of Pregel
- Understand the combiner optimization
- Understand the distributed core decomposition algorithm