

COMP9313: Big Data Management



Lecturer: Xin Cao

Course web site: <http://www.cse.unsw.edu.au/~cs9313/>

Chapter 4: MapReduce IV

Graph Data Processing in MapReduce

What's a Graph?

- ❖ $G = (V, E)$, where
 - V represents the set of vertices (nodes)
 - E represents the set of edges (links)
 - Both vertices and edges may contain additional information
- ❖ Different types of graphs:
 - Directed vs. undirected edges
 - Presence or absence of cycles
- ❖ Graphs are everywhere:
 - Hyperlink structure of the Web
 - Physical structure of computers on the Internet
 - Interstate highway system
 - Social networks

Graph Analytics

- ❖ General Graph
 - Count the number of nodes whose degree is equal to 5
 - Find the diameter of the graphs
- ❖ Web Graph
 - Rank each webpage in the web graph or each user in the twitter graph using PageRank, or other centrality measure
- ❖ Transportation Network
 - Return the shortest or cheapest flight/road from one city to another
- ❖ Social Network
 - Detect a group of users who have similar interests
- ❖ Financial Network
 - Find the path connecting two suspicious transactions;
- ❖

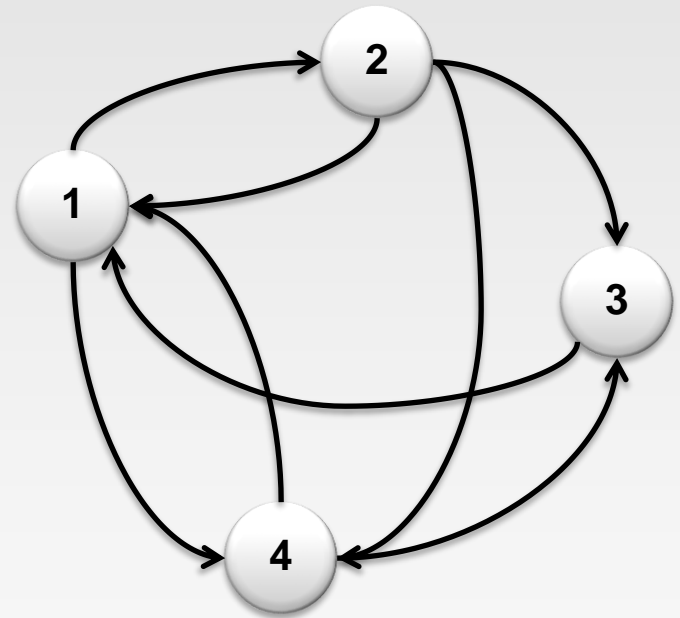
Graphs and MapReduce

- ❖ Graph algorithms typically involve:
 - Performing computations at each node: based on node features, edge features, and local link structure
 - Propagating computations: “traversing” the graph
- ❖ Key questions:
 - How do you represent graph data in MapReduce?
 - How do you traverse a graph in MapReduce?

Representing Graphs

- ❖ Adjacency Matrices: Represent a graph as an $n \times n$ square matrix M
 - $n = |V|$
 - $M_{ij} = 1$ means a link from node i to j

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



Adjacency Matrices: Critique

❖ Advantages:

- Amenable to mathematical manipulation
- Iteration over rows and columns corresponds to computations on outlinks and inlinks

❖ Disadvantages:

- Lots of zeros for sparse matrices
- Lots of wasted space

Representing Graphs

- ❖ Adjacency Lists: Take adjacency matrices... and throw away all the zeros

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

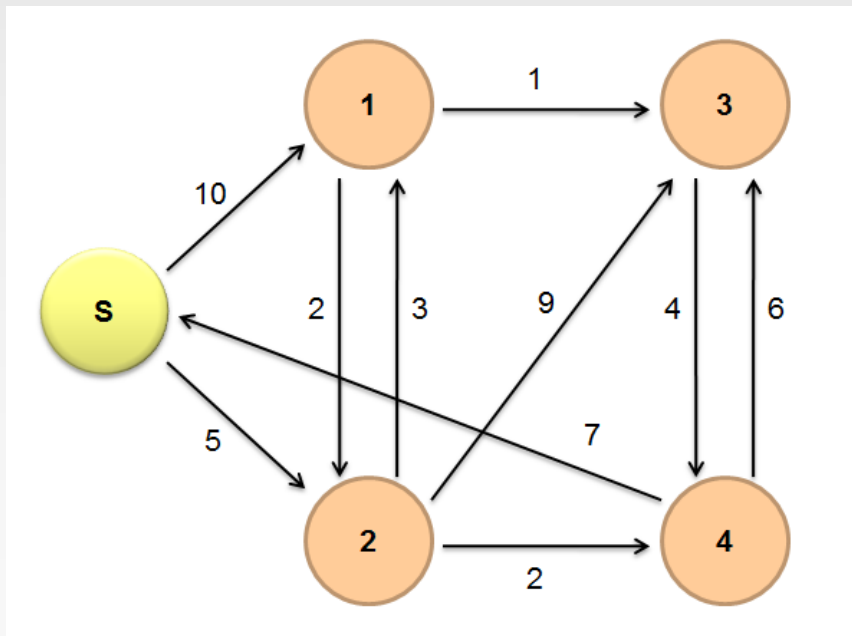
Adjacency Lists: Critique

- ❖ Advantages:
 - Much more compact representation
 - Easy to compute over outlinks
- ❖ Disadvantages:
 - Much more difficult to compute over inlinks

Single-Source Shortest Path

Single-Source Shortest Path (SSSP)

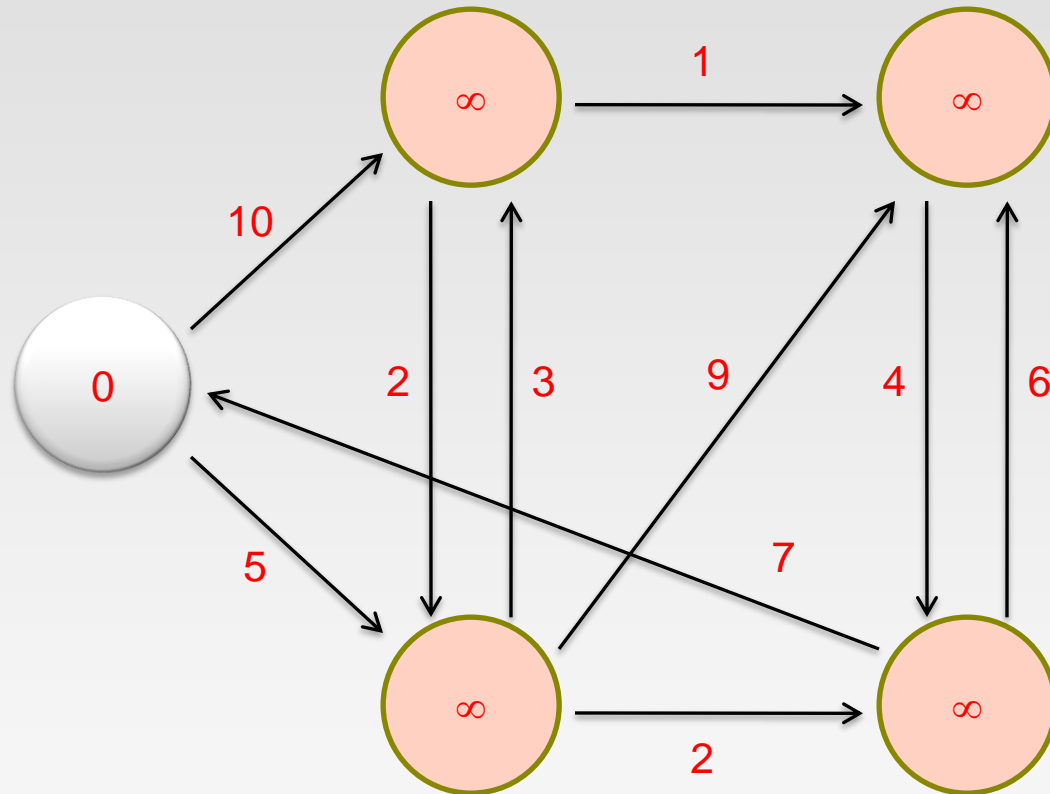
- ❖ **Problem:** find shortest path from a source node to one or more target nodes
 - Shortest might also mean lowest weight or cost
- ❖ Dijkstra's Algorithm:
 - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



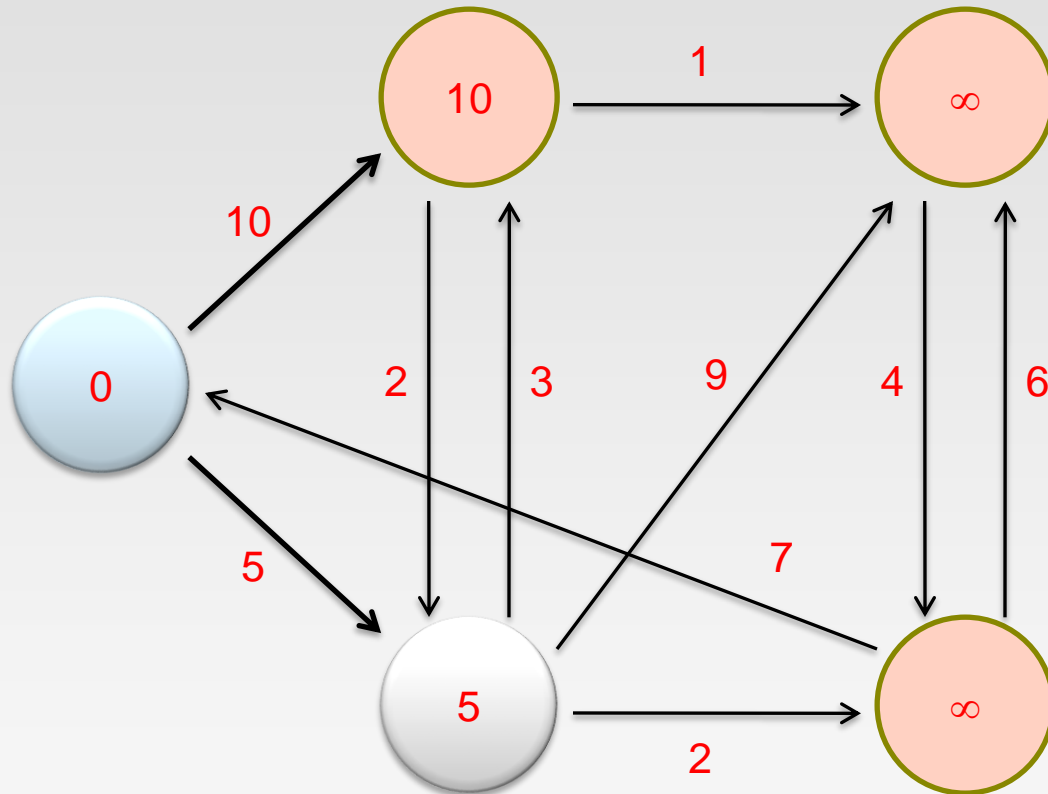
Dijkstra's Algorithm

```
1: DIJKSTRA( $G, w, s$ )
2:    $d[s] \leftarrow 0$ 
3:   for all vertex  $v \in V$  do
4:      $d[v] \leftarrow \infty$ 
5:    $Q \leftarrow \{V\}$ 
6:   while  $Q \neq \emptyset$  do
7:      $u \leftarrow \text{EXTRACTMIN}(Q)$ 
8:     for all vertex  $v \in u.\text{ADJACENCYLIST}$  do
9:       if  $d[v] > d[u] + w(u, v)$  then
10:         $d[v] \leftarrow d[u] + w(u, v)$ 
```

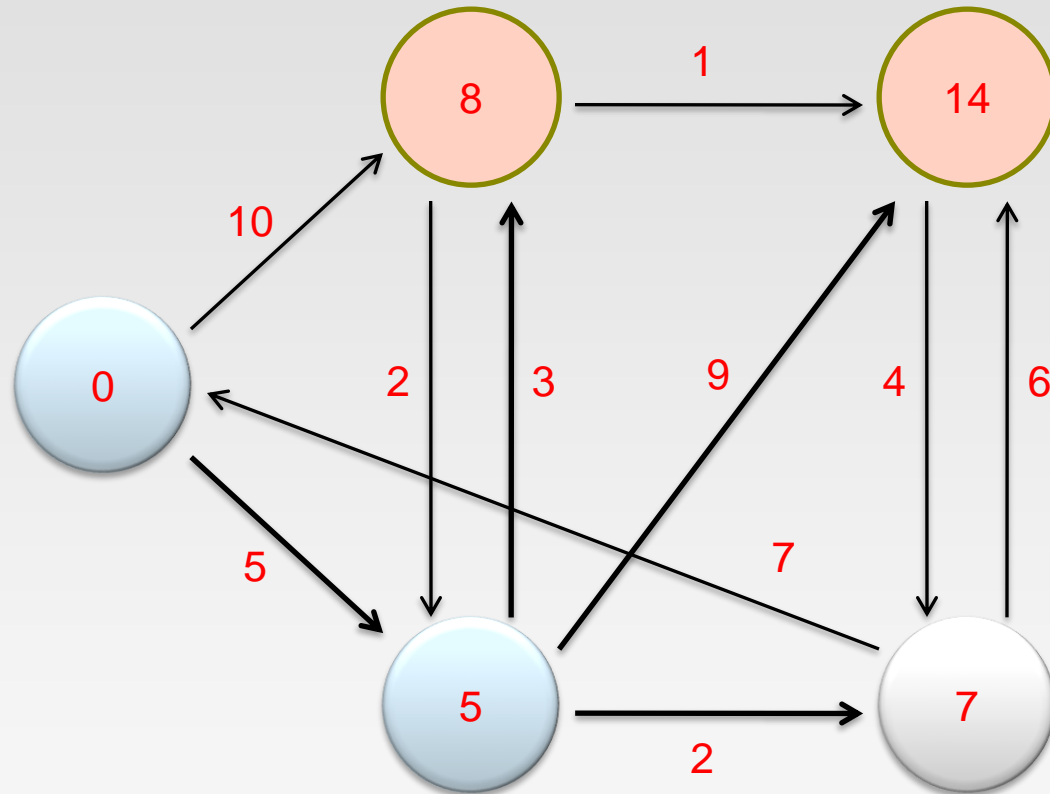
Dijkstra's Algorithm Example



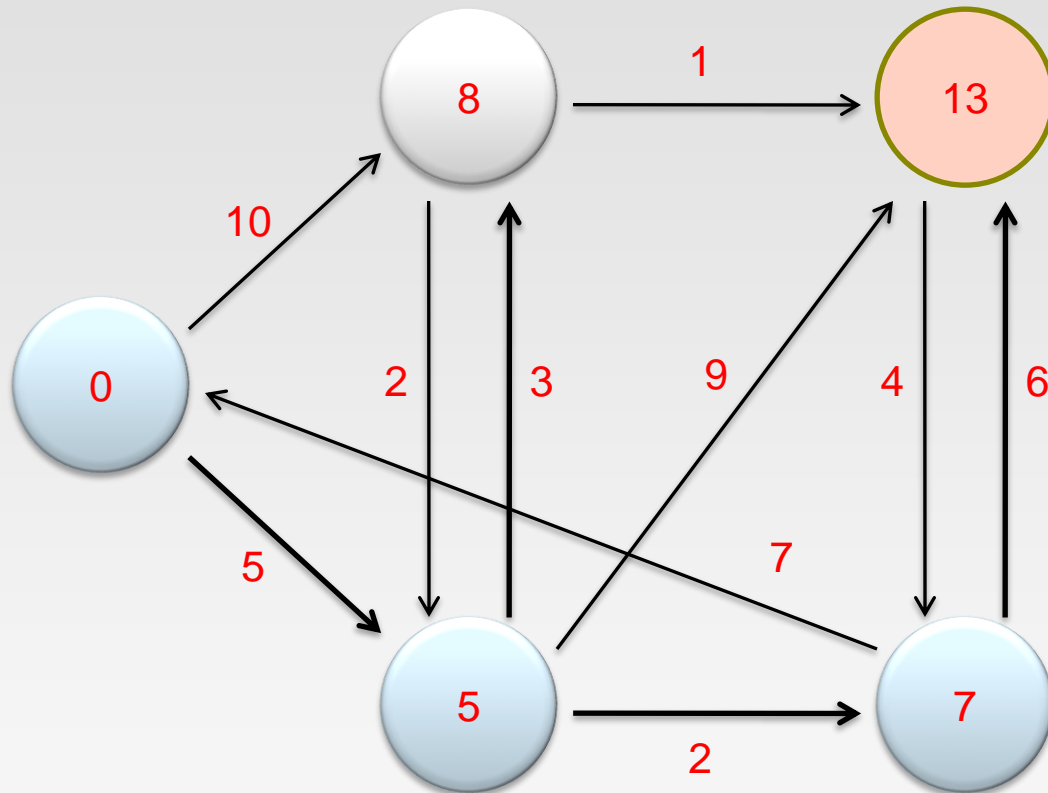
Dijkstra's Algorithm Example



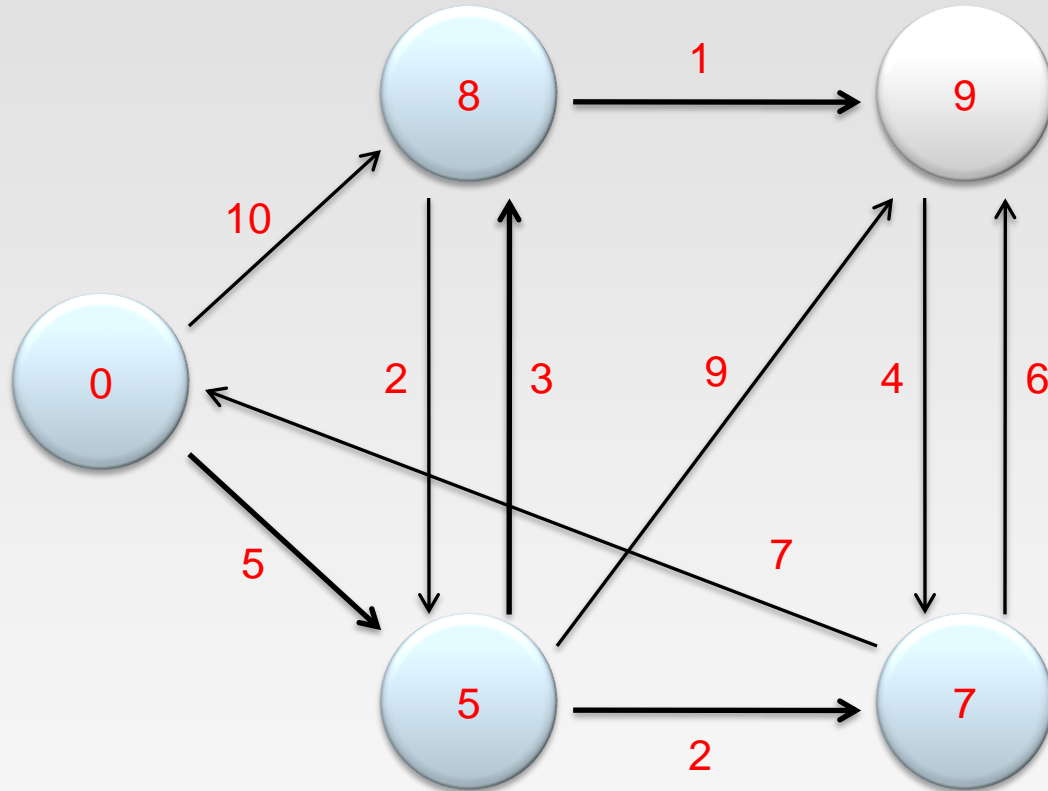
Dijkstra's Algorithm Example



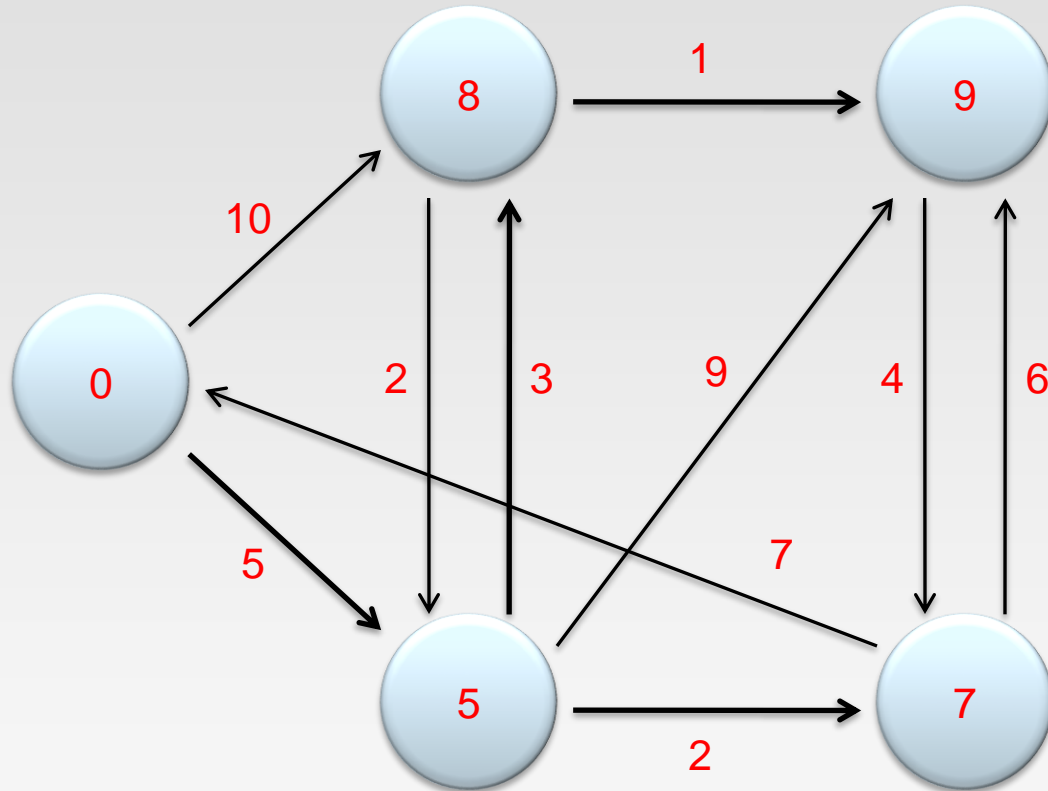
Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



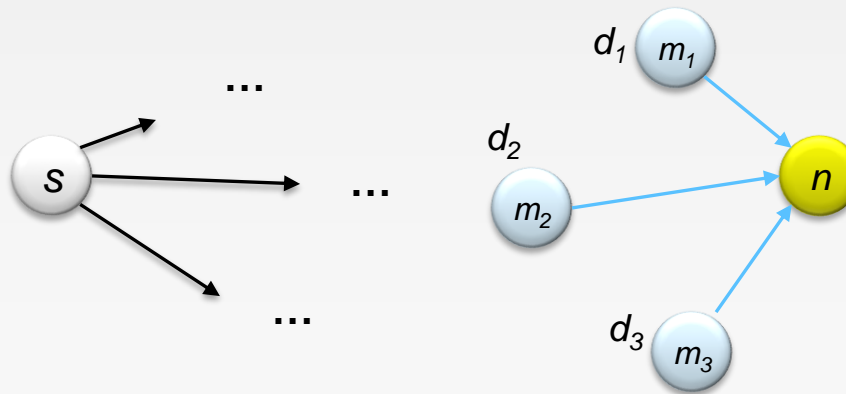
Finish!

Single Source Shortest Path

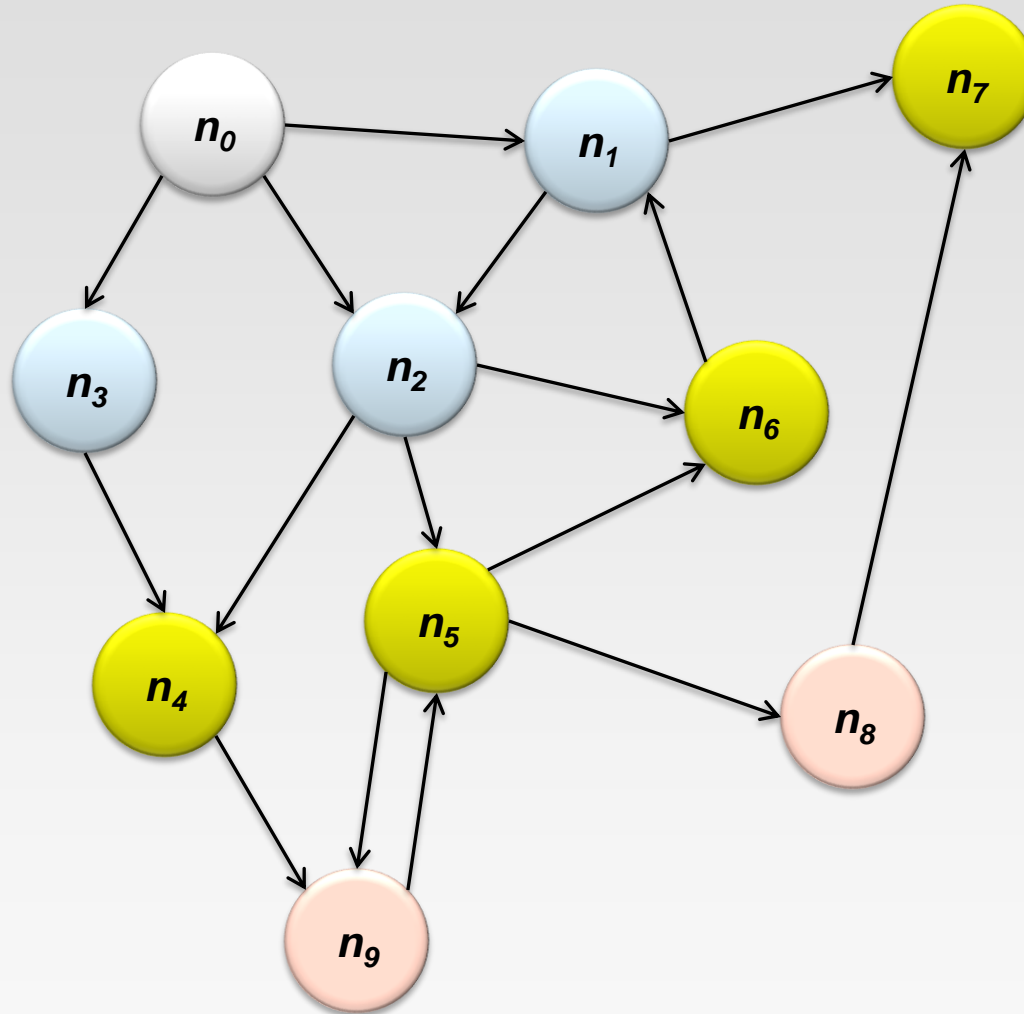
- ❖ **Problem:** find shortest path from a source node to one or more target nodes
 - Shortest might also mean lowest weight or cost
- ❖ Single processor machine: Dijkstra's Algorithm
- ❖ MapReduce: parallel Breadth-First Search (BFS)

Finding the Shortest Path

- ❖ Consider simple case of equal edge weights
- ❖ Solution to the problem can be defined inductively
- ❖ Here's the intuition:
 - Define: b is reachable from a if b is on adjacency list of a
 - $\text{DISTANCETo}(s) = 0$
 - For all nodes p reachable from s ,
 $\text{DISTANCETo}(p) = 1$
 - For all nodes n reachable from some other set of nodes M ,
 $\text{DISTANCETo}(n) = 1 + \min(\text{DISTANCETo}(m), m \in M)$



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

Multiple Iterations Needed

- ❖ Each MapReduce iteration advances the “known frontier” by one hop
 - Subsequent iterations include more and more reachable nodes as frontier expands
 - 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

BFS Pseudo-Code

- ❖ Equal Edge Weights (how to deal with weighted edges?)
- ❖ Only distances, no paths stored (how to obtain paths?)

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

Stopping Criterion

- ❖ How many iterations are needed in parallel BFS (equal edge weight case)?
- ❖ Convince yourself: when a node is first “discovered”, we’ve found the shortest path
- ❖ Now answer the question...
 - The diameter of the graph, or the greatest distance between any pair of nodes
 - Six degrees of separation?
 - ▶ If this is indeed true, then parallel breadth-first search on the global social network would take at most six MapReduce iterations.

Implementation in MapReduce

- ❖ The actual checking of the termination condition must occur outside of MapReduce.
- ❖ The driver (main) checks to see if a termination condition has been met, and if not, repeats.
- ❖ Hadoop provides a lightweight API called “counters”.
 - It can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires.
 - Counters can be designed to count the number of nodes that have distances of ∞ at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

Chained MapReduce Job (Java)

❖ In the main function, you can configure like:

```
String input = IN;
String output = OUT + System.nanoTime();
boolean isdone = false;
while (isdone == false) {
    Job job = Job.getInstance(conf, "traverse job");
    //configure your jobs here such as mapper and reducer classes

    FileInputFormat.addInputPath(job, new Path(input));
    FileOutputFormat.setOutputPath(job, new Path(output));

    job.waitForCompletion(true);    //start the job

    Counters counters = job.getCounters();
    Counter counter = counters.findCounter(MY_COUNTERS.REACHED);

    if(counter.getValue() == 0){    //use the counter to check the termination
        isdone = true;
    }
    input = output;                //make the current output as the next input
    output = OUT + System.nanoTime();
}
```

<https://github.com/himank/Graph-Algorithm-MapReduce/blob/master/src/DijkstraAlgo.java>

Chained MapReduce Job (MRJob)

- ❖ To define multiple steps, override steps() to return a list of MRSteps:

```
class MRMostUsedWord(MRJob):

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]
```

MapReduce Counters

- ❖ Instrument Job's metrics
 - Gather statistics
 - ▶ Quality control – confirm what was expected.
 - E.g., count invalid records
 - ▶ Application-level statistics.
 - Problem diagnostics
 - Try to use counters for gathering statistics instead of log files
- ❖ Framework provides a set of built-in metrics
 - For example, bytes processed for input and output
- ❖ User can create new counters
 - Number of records consumed
 - Number of errors or warnings

Built-in Counters

- ❖ Hadoop maintains some built-in counters for every job.
- ❖ Several groups for built-in counters
 - File System Counters – number of bytes read and written
 - Job Counters – documents number of map and reduce tasks launched, number of failed tasks
 - Map-Reduce Task Counters– mapper, reducer, combiner input and output records counts, time and memory statistics

User-Defined Counters

- ❖ You can create your own counters
 - Counters are defined by a Java enum
 - ▶ serves to group related counters
 - ▶ E.g.,

```
enum Temperature {  
    MISSING,  
    MALFORMED  
}
```
- ❖ Increment counters in Reducer and/or Mapper classes
 - Counters are global: Framework accurately sums up counts across all maps and reduces to produce a grand total at the end of the job

Implement User-Defined Counters

- ❖ Retrieve Counter from Context object
 - Framework injects Context object into map and reduce methods
- ❖ Increment Counter's value
 - Can increment by 1 or more

```
parser.parse(value);
if (parser.isValidTemperature()) {
    int airTemperature = parser.getAirTemperature();
    context.write(new Text(parser.getYear()),
        new IntWritable(airTemperature));
} else if (parser.isMalformedTemperature()) {
    System.err.println("Ignoring possibly corrupt input: " + value);
    context.getCounter(Temperature.MALFORMED).increment(1);
} else if (parser.isMissingTemperature()) {
    context.getCounter(Temperature.MISSING).increment(1);
}
```

Implement User-Defined Counters

- ❖ Get Counters from a finished job in Java
 - `Counter counters = job.getCounters();`
- ❖ Get the counter according to name
 - `Counter c1 = counters.findCounter(Temperature.MISSING)`
- ❖ Enumerate all counters after job is completed

```
for (CounterGroup group : counters) {  
    System.out.println("* Counter Group: " + group.getDisplayName() + " (" +  
        group.getName() + ")");  
    System.out.println(" number of counters in this group: " + group.size());  
    for (Counter counter : group) {  
        System.out.println(" - " + counter.getDisplayName() + ": " +  
            counter.getName() + ": "+counter.getValue());  
    }  
}
```

Counters in MRJob

- ❖ A counter has a group, a name, and an integer value. Hadoop itself tracks a few counters automatically. mrjob prints your job's counters to the command line when your job finishes, and they are available to the runner object if you invoke it programmatically.
- ❖ To increment a counter from anywhere in your job, use the `increment_counter()` method:

```
class MRCountingJob(MRJob):  
  
    def steps(self):  
        # 3 steps so we can check behavior of counters for multiple steps  
        return [MRStep(self.mapper),  
                MRStep(self.mapper),  
                MRStep(self.mapper)]  
  
    def mapper(self, _, value):  
        self.increment_counter('group', 'counter_name', 1)  
        yield _, value
```

- ❖ At the end of your job, you'll get the counter's total value.
- ❖ You can also read the counters by using “`runner.counters()`”

<https://mrjob.readthedocs.io/en/latest/guides/runners.html>

How to Find the Shortest Path?

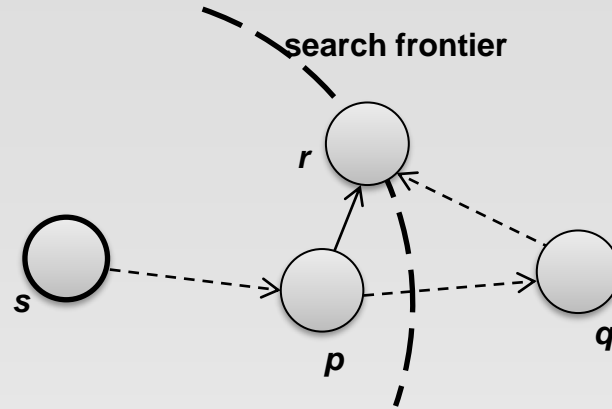
- ❖ The parallel breadth-first search algorithm only finds the shortest distances.
- ❖ Store “back-pointers” at each node, as with Dijkstra's algorithm
 - Not efficient to recover the path from the back-pointers
- ❖ A simpler approach is to emit paths along with distances in the mapper, so that each node will have its shortest path easily accessible at all times
 - The additional space requirement is acceptable

BFS Pseudo-Code (Weighted Edges)

- ❖ The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well
 - Positive weights!
- ❖ In line 6 of the mapper code, instead of emitting $d + 1$ as the value, we must now emit $d + w$, where w is the edge distance
- ❖ **The termination behaviour is very different!**
 - How many iterations are needed in parallel BFS (positive edge weight case)?
 - Convince yourself: when a node is first “discovered”, we’ve found the shortest path

Not true!

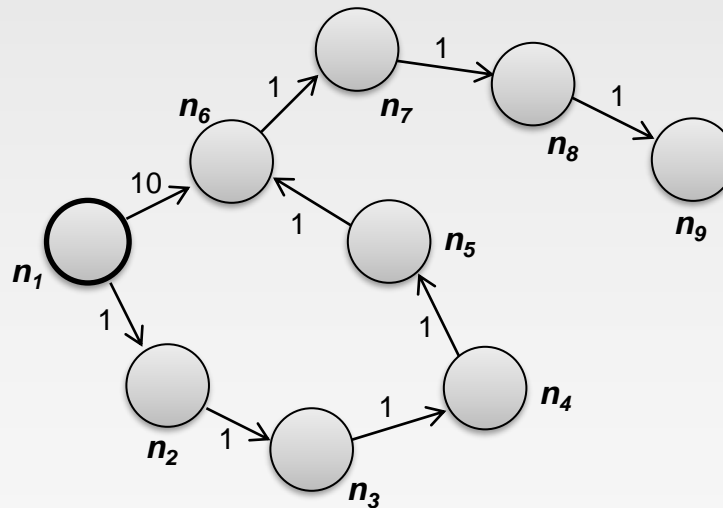
Additional Complexities



- ❖ Assume that p is the current processed node
 - In the current iteration, we just “discovered” node r for the very first time.
 - We've already discovered the shortest distance to node p , and that the shortest distance to r **so far** goes through p
 - **Is $s \rightarrow p \rightarrow r$ the shortest path from s to r ?**
- ❖ The shortest path from source s to node r may go outside the current search frontier
 - It is possible that $p \rightarrow q \rightarrow r$ is shorter than $p \rightarrow r$!
 - We will not find the shortest distance to r until the search frontier expands to cover q .

How Many Iterations Are Needed?

- ❖ In the worst case, we might need as many iterations as there are nodes in the graph minus one
 - A sample graph that elicits worst-case behaviour for parallel breadth-first search.
 - Eight iterations are required to discover shortest distances to all nodes from n_1 .



Example (only distances)

❖ Input file:

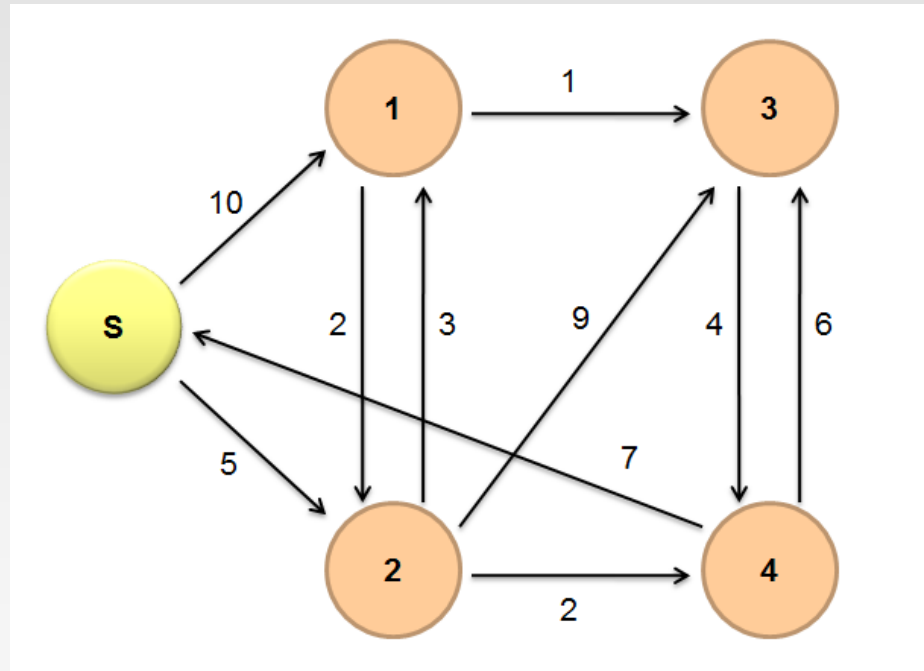
s --> 0 | n1: 10, n2: 5

n1 --> ∞ | n2: 2, n3:1

n2 --> ∞ | n1: 3, n3:9, n4:2

n3 --> ∞ | n4:4

n4 --> ∞ | s:7, n3:6



Iteration 1

❖ Map:

Read s --> 0 | n1: 10, n2: 5

Emit: (n1, 10), (n2, 5), and the adjacency list (s, n1: 10, n2: 5)

The other lists will also be read and emit, but they do not contribute, and thus ignored

❖ Reduce:

Receives: (n1, 10), (n2, 5), (s, <0, (n1: 10, n2: 5)>)

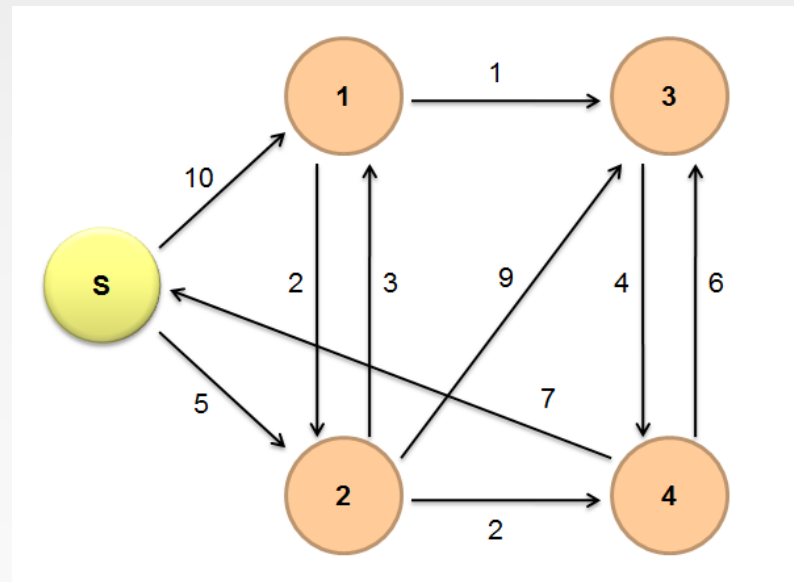
The adjacency list of each node will also be received, ignored in example

Emit:

s --> 0 | n1: 10, n2: 5

n1 --> 10 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2



Iteration 2

❖ Map:

Read: n1 --> 10 | n2: 2, n3:1

Emit: (n2, 12), (n3, 11), (n1, <10, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Ignore the processing of the other lists

❖ Reduce:

Receives: (n1, (8, <10, (n2: 2, n3:1)>)), (n2, (12, <5, n1: 3, n3:9, n4:2>)), (n3, (11, 14)), (n4, 7)

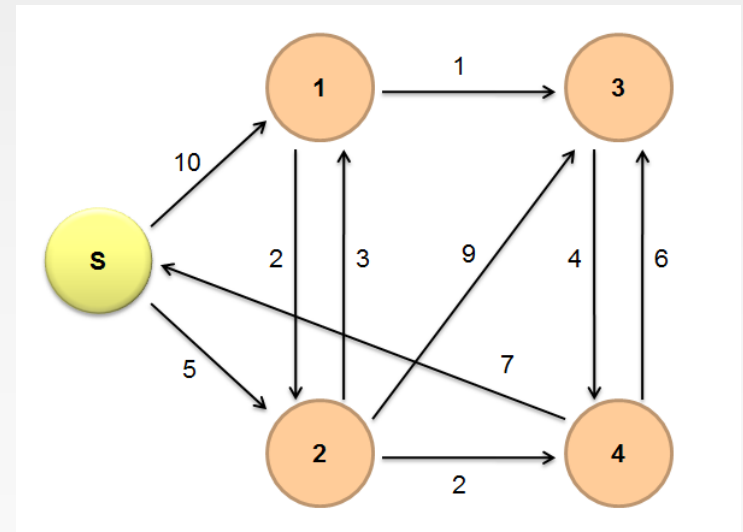
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 11 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 3

❖ Map:

Read: n1 --> 8 | n2: 2, n3:1

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 11 | n4:4

Emit: (n4, 15), (n3, <11, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

❖ Reduce:

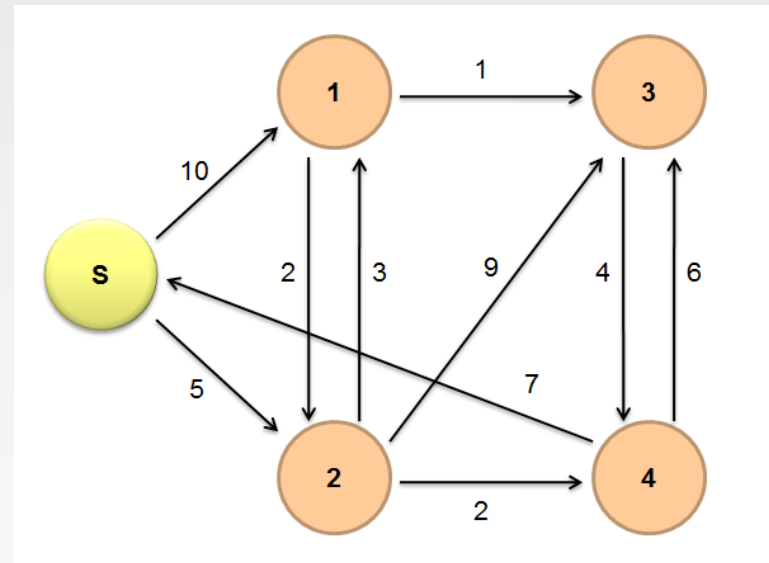
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 4

❖ Map:

Read: n1 --> 8 | n2: 2, n3:1 (**Again!**)

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 9 | n4:4

Emit: (n4, 13), (n3, <9, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6 (**Again!**)

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

❖ Reduce:

Emit:

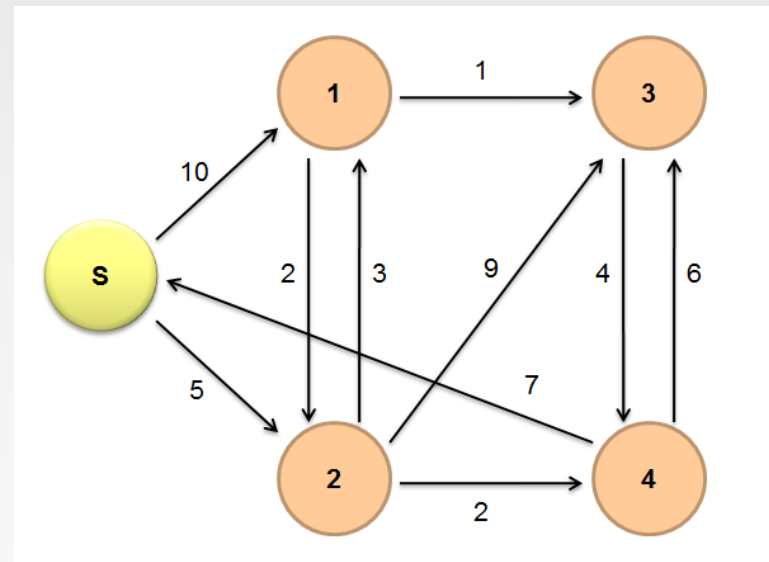
n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6

In order to avoid duplicated computations, you can use a status value to indicate whether the distance of the node has been modified in the previous iteration.



No updates. Terminate.

Comparison to Dijkstra

- ❖ Dijkstra's algorithm is more efficient
 - At any step it only pursues edges from the minimum-cost path inside the frontier
- ❖ MapReduce explores all paths in parallel
 - Lots of “waste”
 - Useful work is only done at the “frontier”
- ❖ Why can't we do better using MapReduce?

Graphs and MapReduce

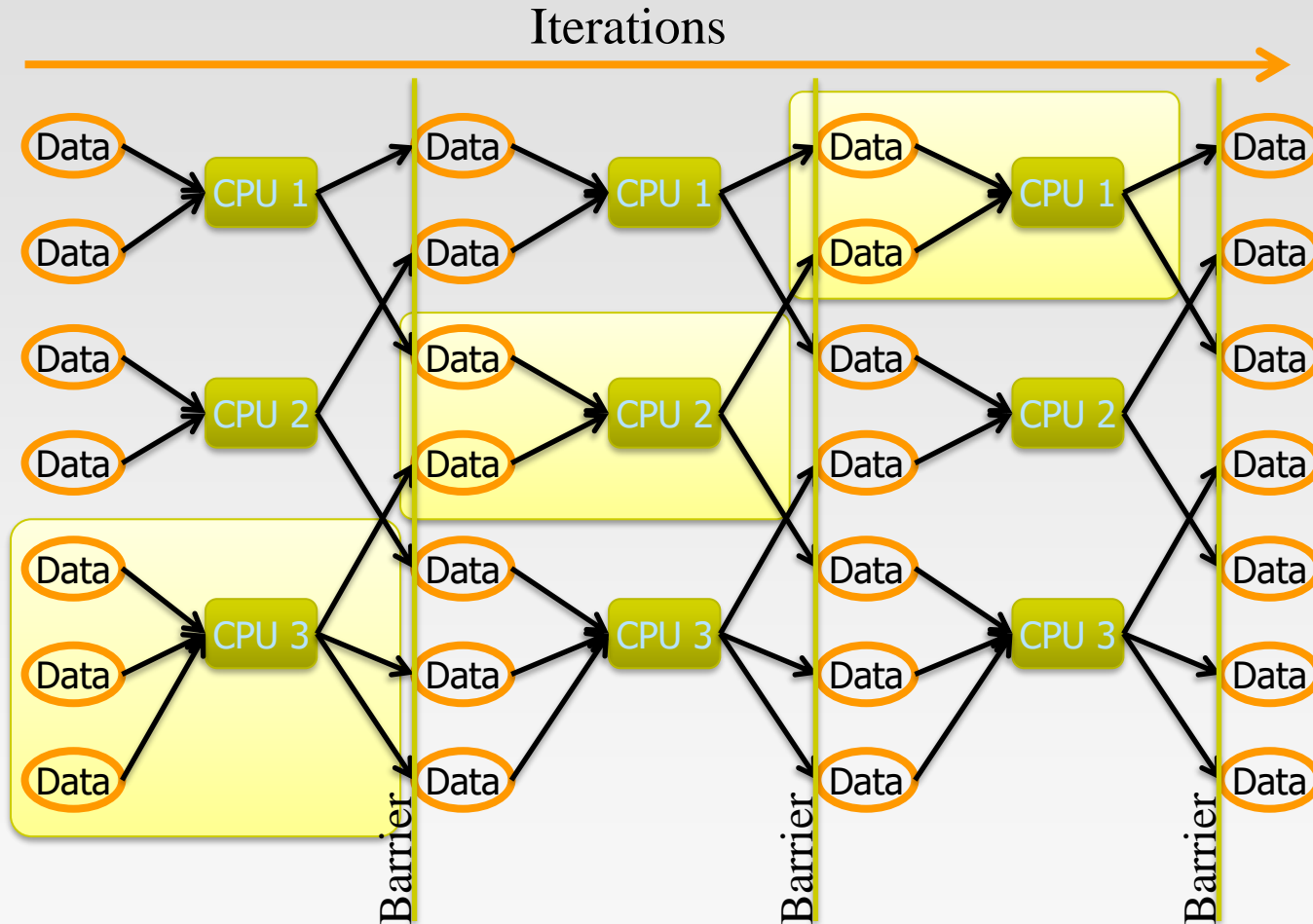
- ❖ Graph algorithms typically involve:
 - Performing computations at each node: based on node features, edge features, and local link structure
 - Propagating computations: “traversing” the graph
- ❖ Generic recipe:
 - Represent graphs as adjacency lists
 - Perform local computations in mapper
 - Pass along partial results via outlinks, keyed by destination node
 - Perform aggregation in reducer on inlinks to a node
 - Iterate until convergence: controlled by external “driver”
 - Don’t forget to pass the graph structure between iterations

Issues with MapReduce on Graph Processing

- ❖ MapReduce Does not support iterative graph computations:
 - External driver. Huge I/O incurs
 - No mechanism to support global data structures that can be accessed and updated by all mappers and reducers
 - ▶ Passing information is only possible within the local graph structure – through adjacency list
 - ▶ Dijkstra's algorithm on a single machine: a global priority queue that guides the expansion of nodes
 - ▶ Dijkstra's algorithm in Hadoop, no such queue available. Do some “wasted” computation instead
- ❖ MapReduce algorithms are often impractical on large, dense graphs.
 - The amount of intermediate data generated is on the order of the number of edges.
 - For dense graphs, MapReduce running time would be dominated by copying intermediate data across the network.

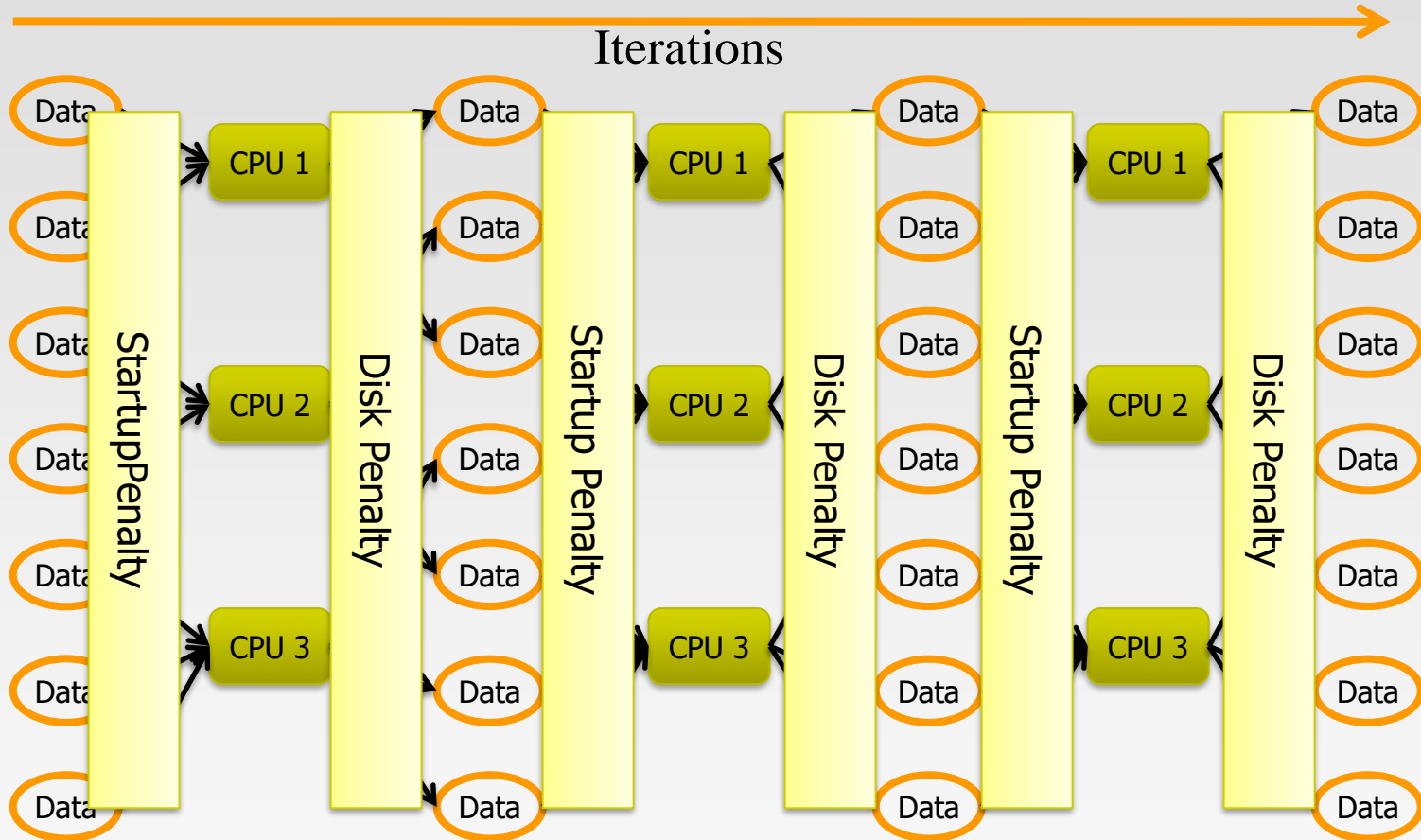
Iterative MapReduce

- ❖ Only a subset of data needs computation:



Iterative MapReduce

- ❖ System is not optimized for iteration:



Better Partitioning

- ❖ Default: hash partitioning
 - Randomly assign nodes to partitions
- ❖ Observation: many graphs exhibit local structure
 - E.g., communities in social networks
 - Better partitioning creates more opportunities for local aggregation
- ❖ Unfortunately, partitioning is **hard**!
 - Sometimes, chick-and-egg...
 - But cheap heuristics sometimes available
 - For webgraphs: range partition on domain-sorted URLs

References

- ❖ Chapter 5, Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.

End of Chapter 3.2

Practices

Practice: Design MapReduce Algorithms

- ❖ Counting total enrollments of two specified courses
- ❖ Input Files: A list of students with their enrolled courses
 - Jamie: COMP9313, COMP9318
 - Tom: COMP9331, COMP9313
 -
- ❖ Mapper selects records and outputs initial counts
 - Input: Key – student, value – a list of courses
 - Output: (COMP9313, 1), (COMP9318, 1), ...
- ❖ Reducer accumulates counts
 - Input: (COMP9313, [1, 1, ...]), (COMP9318, [1, 1, ...])
 - Output: (COMP9313, 16), (COMP9318, 35)

Practice: Design MapReduce Algorithms

- ❖ Remove duplicate records

- ❖ Input: a list of records

2013-11-01 aa
2013-11-02 bb
2013-11-03 cc
2013-11-01 aa
2013-11-03 dd

- ❖ Mapper

- Input (record_id, record)
- Output (record, "")
 - ▶ E.g., (2013-11-01 aa, ""), (2013-11-02 bb, ""), ...

- ❖ Reducer

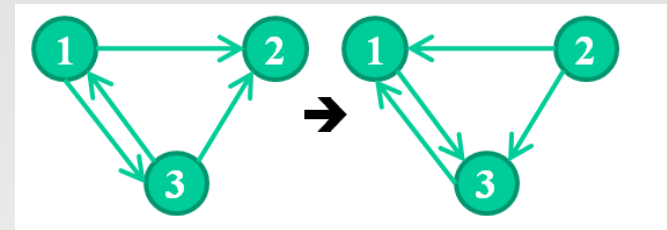
- Input (record, ["", "", "", ...])
 - ▶ E.g., (2013-11-01 aa, ["", ""]), (2013-11-02 bb, [""]), ...
- Output (record, "")

Practice: Design MapReduce Algorithms

- ❖ Reverse graph edge directions & output in node order

- ❖ Input: adjacency list of graph (3 nodes and 4 edges)

(3, [1, 2]) (1, [3])
(1, [2, 3]) → (2, [1, 3])
 (3, [1])



- ❖ Note, the node_ids in the output values are also sorted. But Hadoop only sorts on keys!
- ❖ Solutions: Secondary sort

Practice: Design MapReduce Algorithms

❖ Map

- Input: (3, [1, 2]), (1, [2, 3]).
- Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse direction)
- Output: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1]).
 - ▶ Copy node_ids from value to key.

❖ Partition on Key.field1, and Sort on whole Key (both fields)

- Input: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1])
- Output: (<1, 3>, [3]), (<2, 1>, [1]), (<2, 3>, [3]), (<3, 1>, [1])

❖ Grouping comparator

- Merge according to part of the key
- Output: (<1, 3>, [3]), (<2, 1>, [1, 3]), (<3, 1>, [1])
this will be the reducer's input

❖ Reducer

- Merge according to part of the key
- Output: (1, [3]), (2, [1, 3]), (3, [1])

Practice: Design MapReduce Algorithms

- ❖ Calculate the common friends for each pair of users in Facebook. Assume the friends are stored in format of Person->[List of Friends], e.g.: A -> [B C D], B -> [A C D E], C -> [A B D E], D -> [A B C E], E -> [B C D]. Note that the “friendship” is bi-directional, which means that if A is in B’s list, B would be in A’s list as well. Your result should be like:

- (A B) -> (C D)
- (A C) -> (B D)
- (A D) -> (B C)
- (B C) -> (A D E)
- (B D) -> (A C E)
- (B E) -> (C D)
- (C D) -> (A B E)
- (C E) -> (B D)
- (D E) -> (B C)

Practice: Design MapReduce Algorithms

❖ Mapper:

- Input(user u , List of Friends $[f_1, f_2, \dots,])$
- map(): for each friend f_i , emit ($\langle u, f_i \rangle$, List of Friends $[f_1, f_2, \dots,])$
 - ▶ Need to generate the pair $\langle u, f_i \rangle$ according to an order! Thus $\langle u, f_i \rangle$ and $\langle f_i, u \rangle$ will be the same key

❖ Reducer:

- Input(user pair, list of friends lists[])
- Get the intersection from all friends lists

❖ Example: <http://scaryscientist.blogspot.com/2015/04/common-friends-using-mapreduce.html>

Practice: Design MapReduce Algorithms

- ❖ Assume that in an online shopping system, a huge log file stores the information of each transaction. Each line of the log is in format of “userID\t product\t price\t time”. Your task is to use MapReduce to find out the top-5 expensive products purchased by each user in 2016
- ❖ Mapper:
 - Input(transaction_id, transaction)
 - initialize an associate array H(UserID, priority queue Q of log record based on price)
 - map(): get local top-5 for each user
 - cleanup(): emit the entries in H
- ❖ Reducer:
 - Input(userID, list of queues[])
 - get top-5 products from the list of queues

Practice: Design MapReduce Algorithms

- ❖ Given a large text dataset, find the top-k frequent terms (considering that you can utilize multiple reducers, and the efficiency of your method is evaluated).
- ❖ Two rounds: first round compute term frequency in multiple reducers, and each reducer only stores local top-k. Second round get the local top-k, and compute the final top-k using a single reducer.