COMP9313: Big Data Management

Lecturer: Xin Cao

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

Chapter 8.1: 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;
- … …

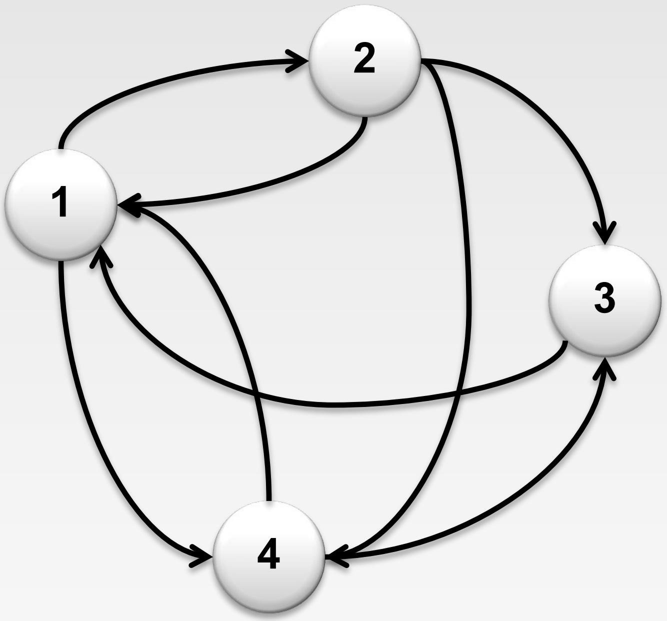
Graphs and MapReduce

- 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 x 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

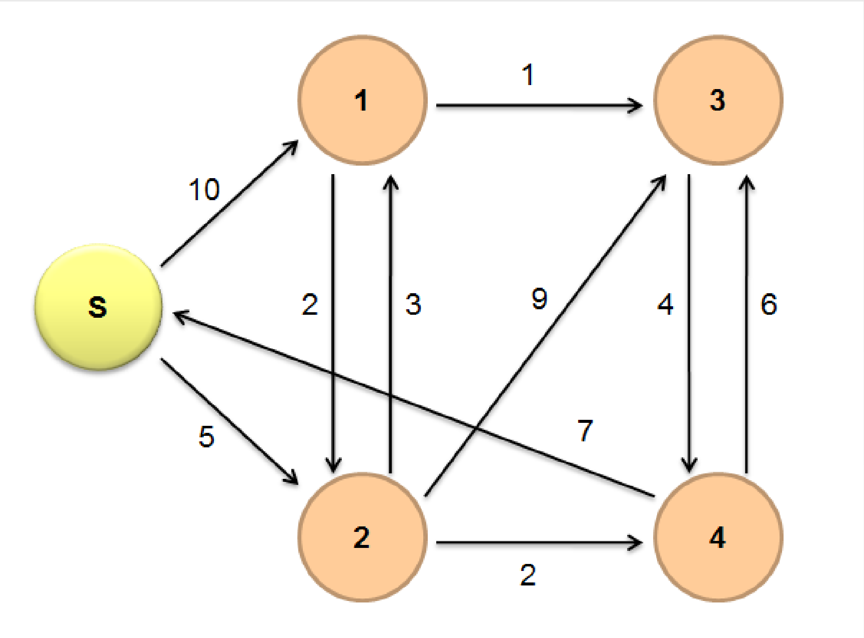
Adjacency Lists: Critique

- 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)

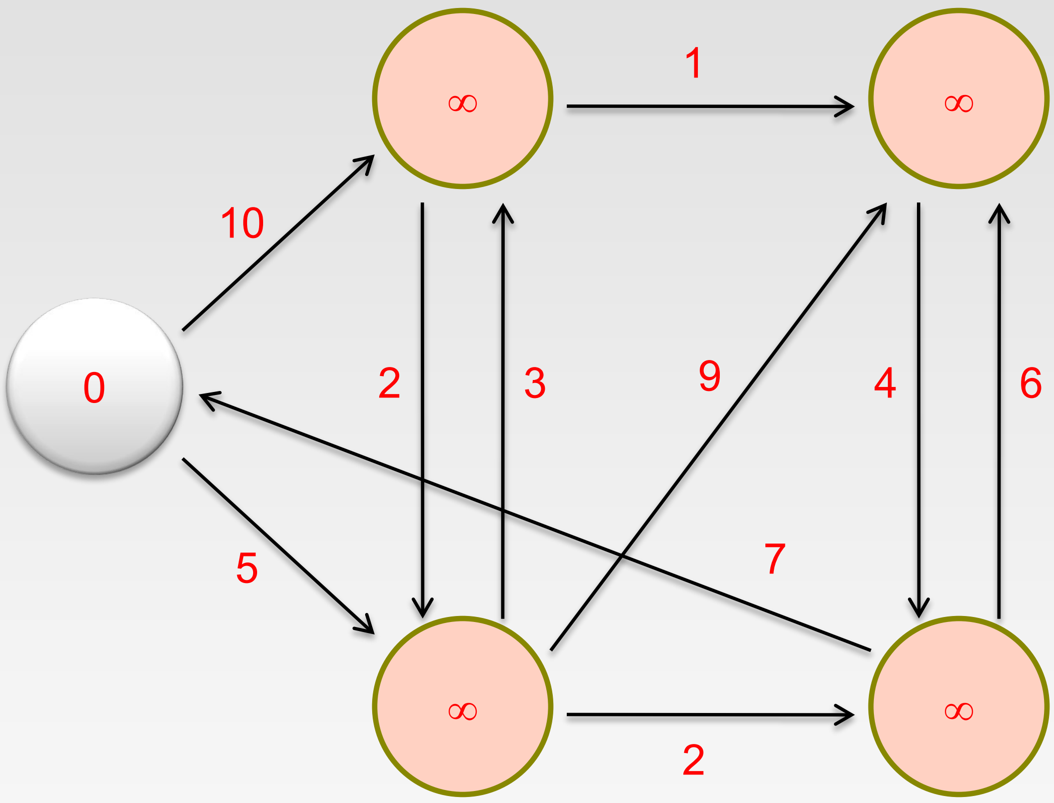
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Dijkstra's Algorithm:
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



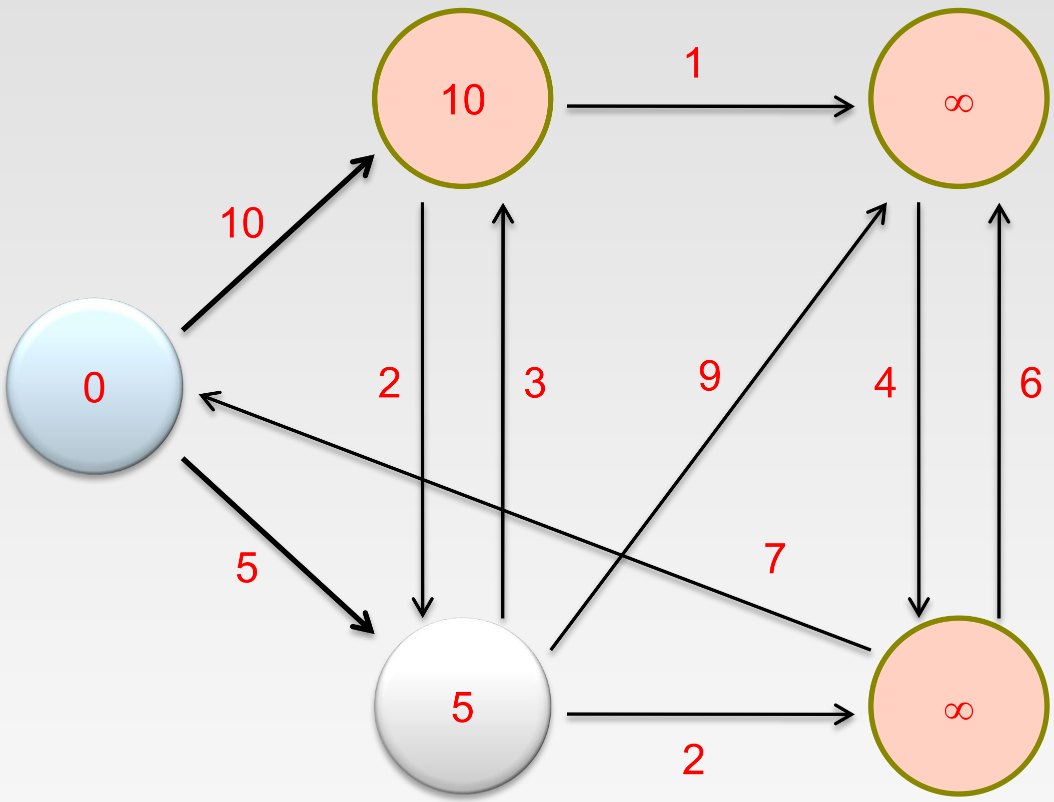
Dijkstra's Algorithm

$\textsc{Dijkstra}(G, w, s)$
    $d[s] \leftarrow 0$
    for all vertex $v \in V$ **do**
        $d[v] \leftarrow \infty$
    $Q \leftarrow \{V\}$
    while $Q \neq \emptyset$ **do**
        $u \leftarrow \textsc{ExtractMin}(Q)$
        for all vertex $v \in u.\textsc{AdjacencyList}$ **do**
            if $d[v] > d[u] + w(u, v)$ **then**
                $d[v] \leftarrow d[u] + w(u, v)$

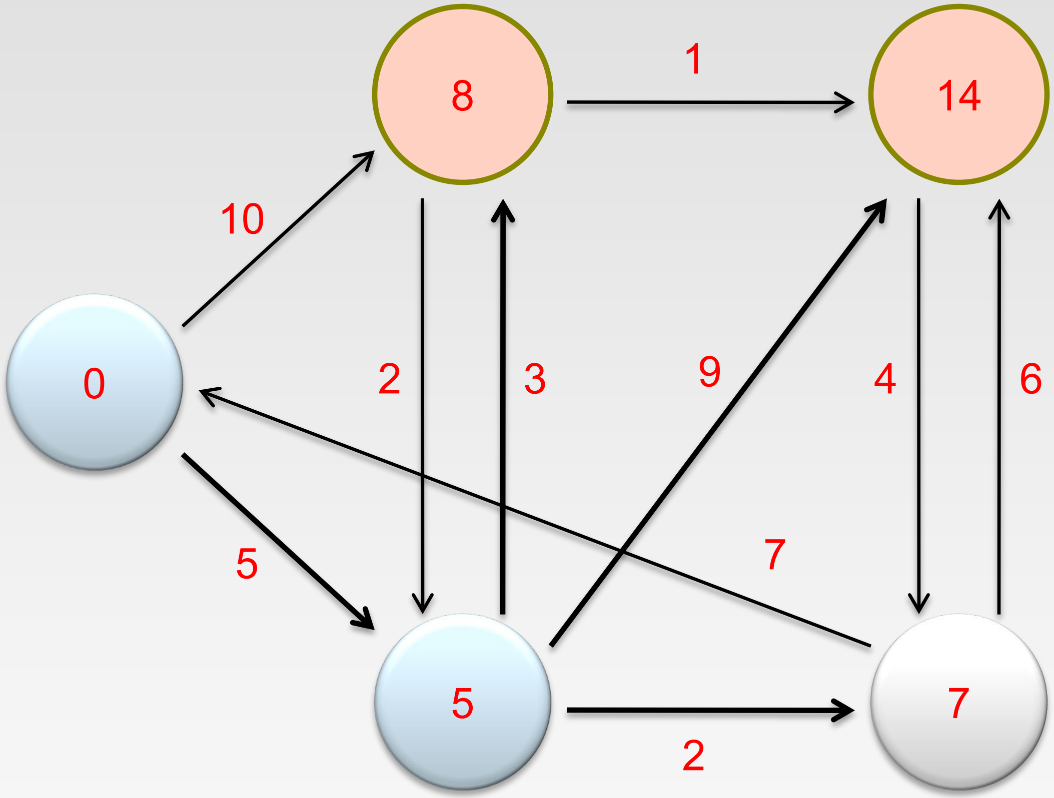
Dijkstra’s Algorithm Example



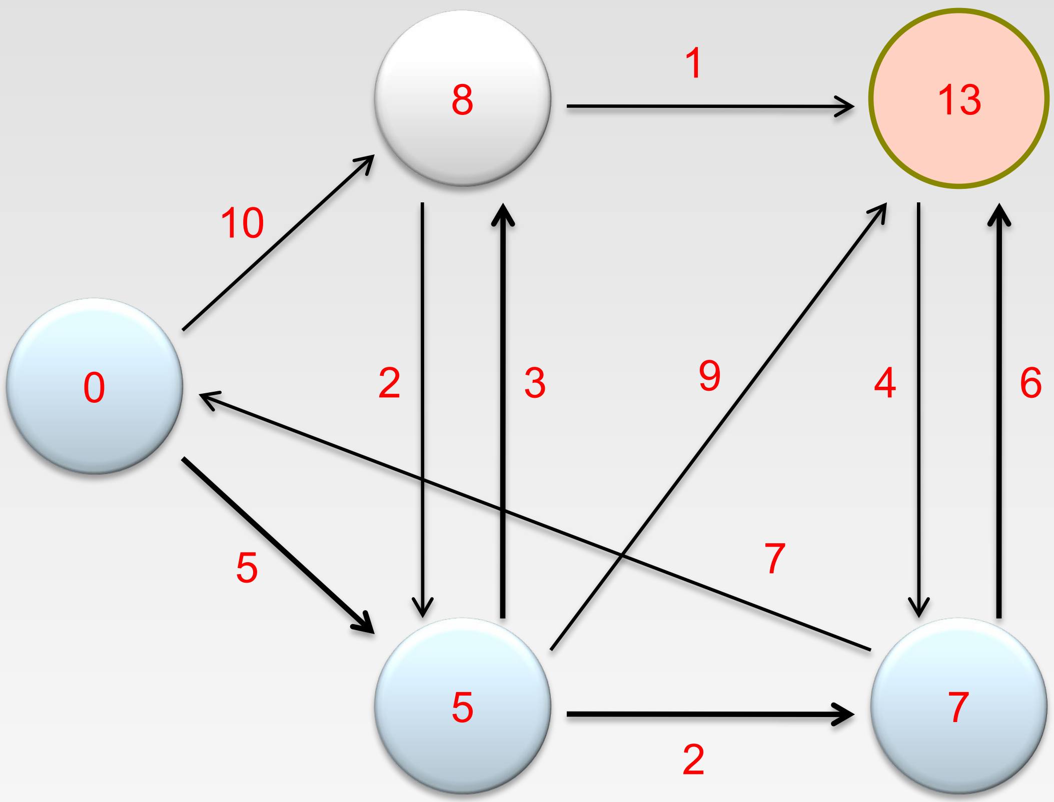
Dijkstra's Algorithm Example



Dijkstra’s Algorithm Example

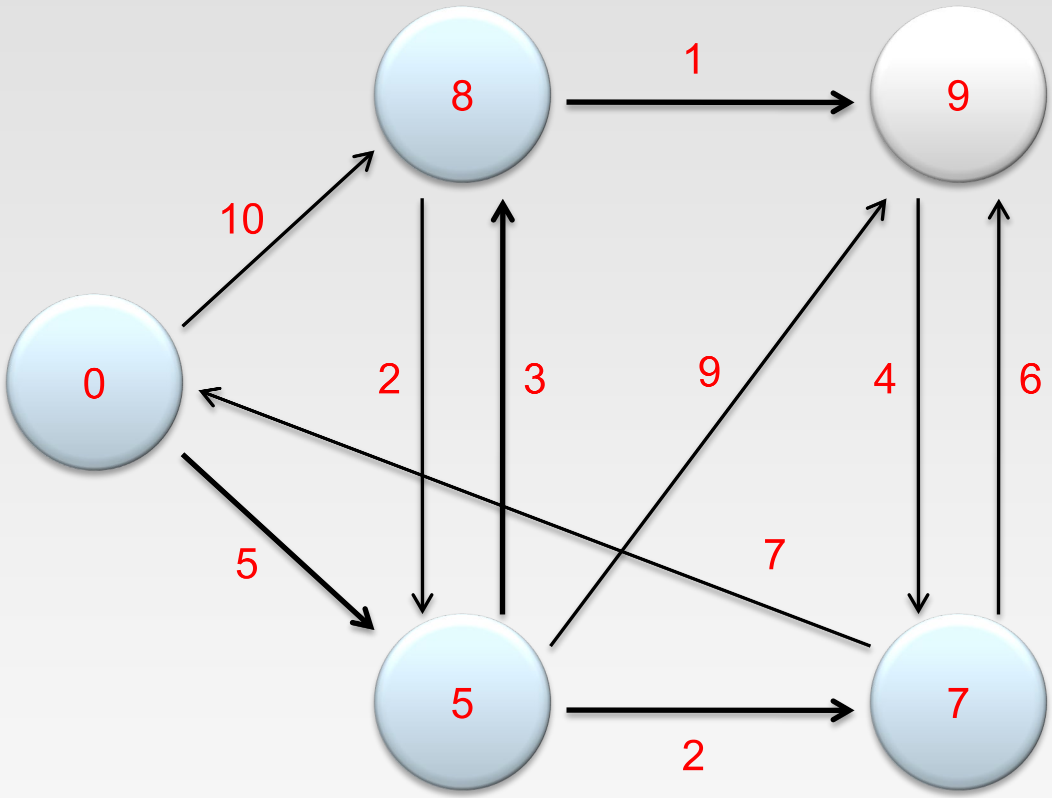


Dijkstra's Algorithm Example

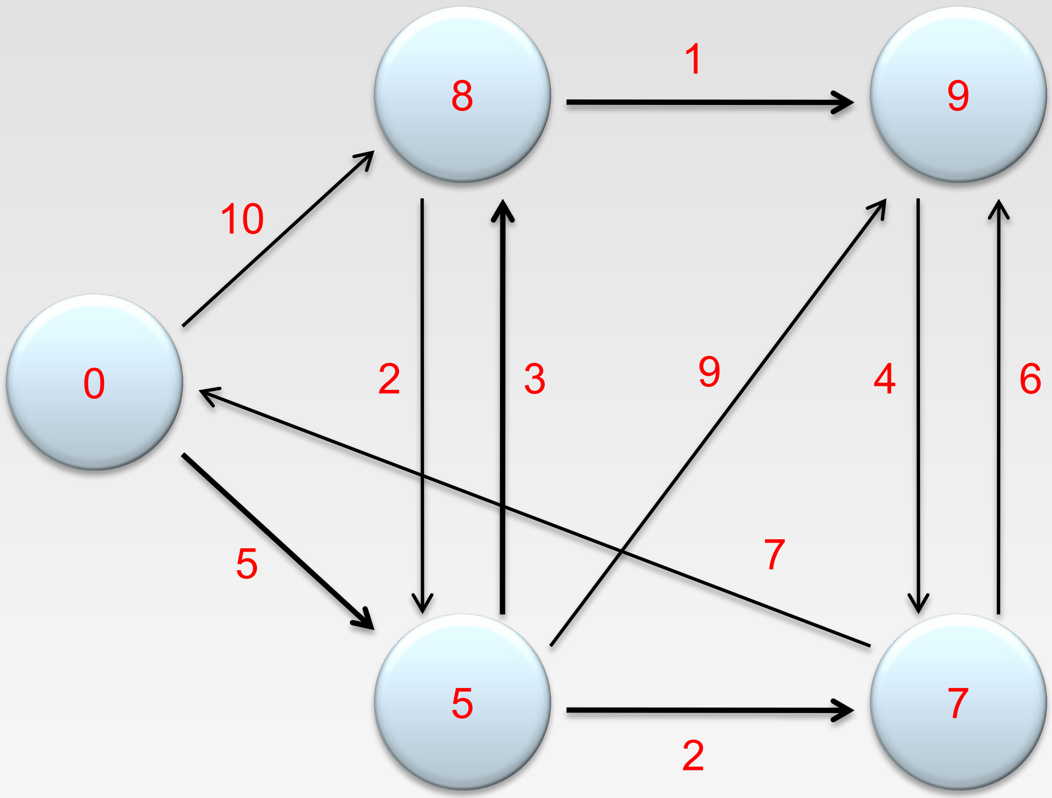


Example from CLR

Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



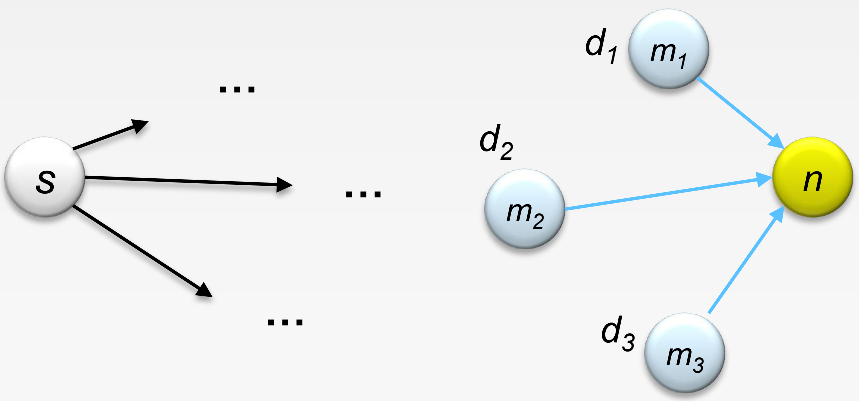
Finish!

Single Source Shortest Path

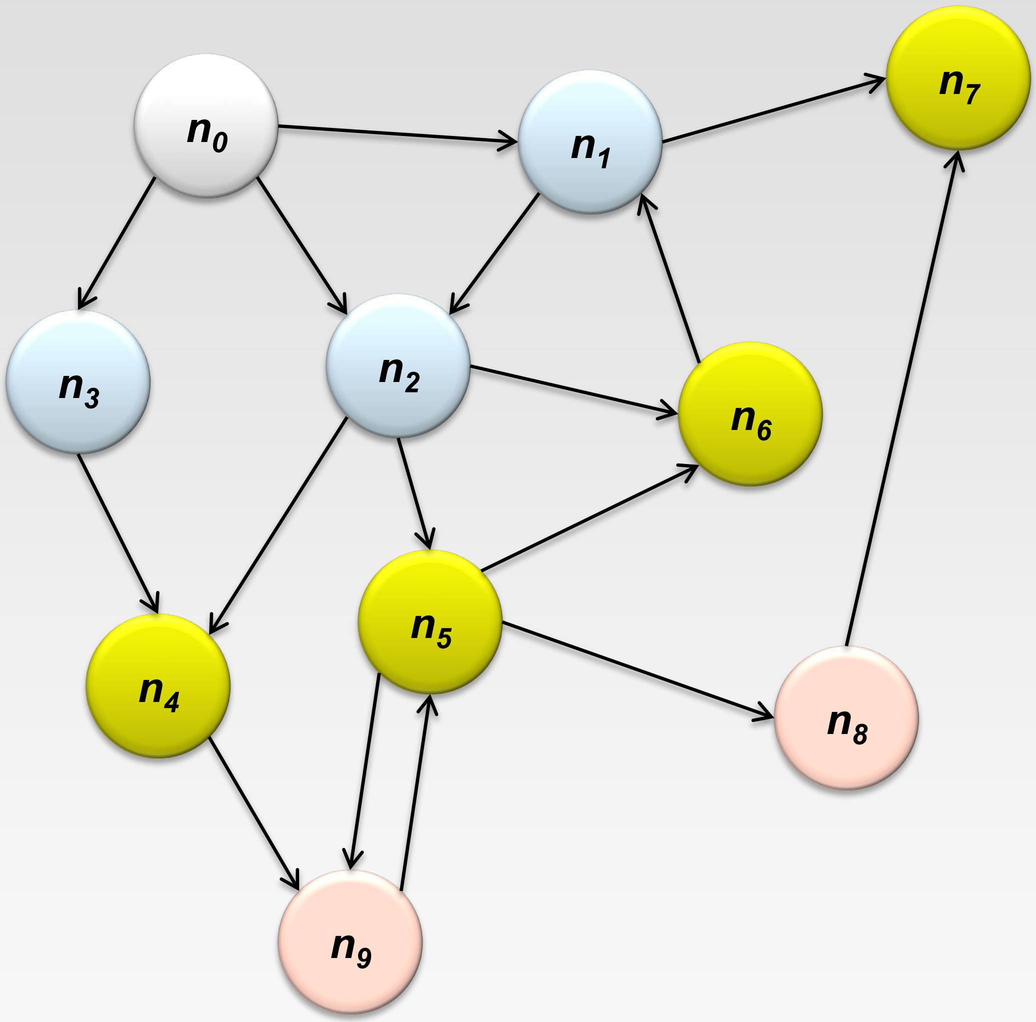
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra’s Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

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*
  - DISTANCETO(s) = 0
  - For all nodes p reachable from s,
    DISTANCETO(p) = 1
  - For all nodes n reachable from some other set of nodes M,
    DISTANCETO(n) = 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

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, . . .])
    d_min ← ∞
    M ← Ø
    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)
```

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/DijikstraAlgo.java

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)

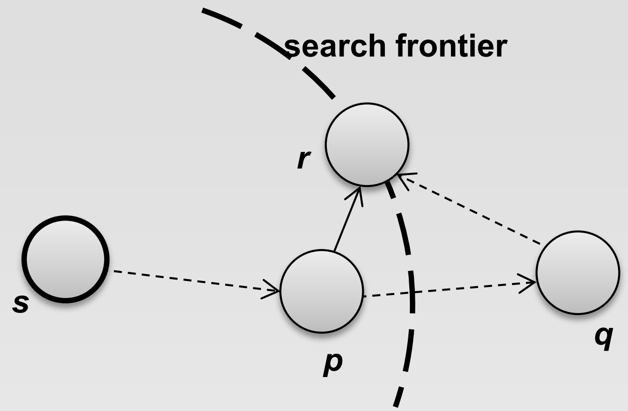
- The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well
  - Positive weights!

- 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!

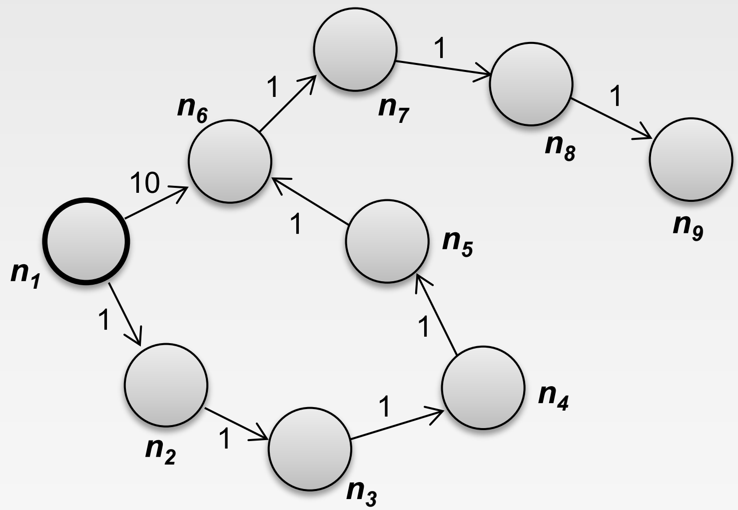
Additional Complexities



- 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*->*p*->*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*->*q*->*r* is shorter than *p*->*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:

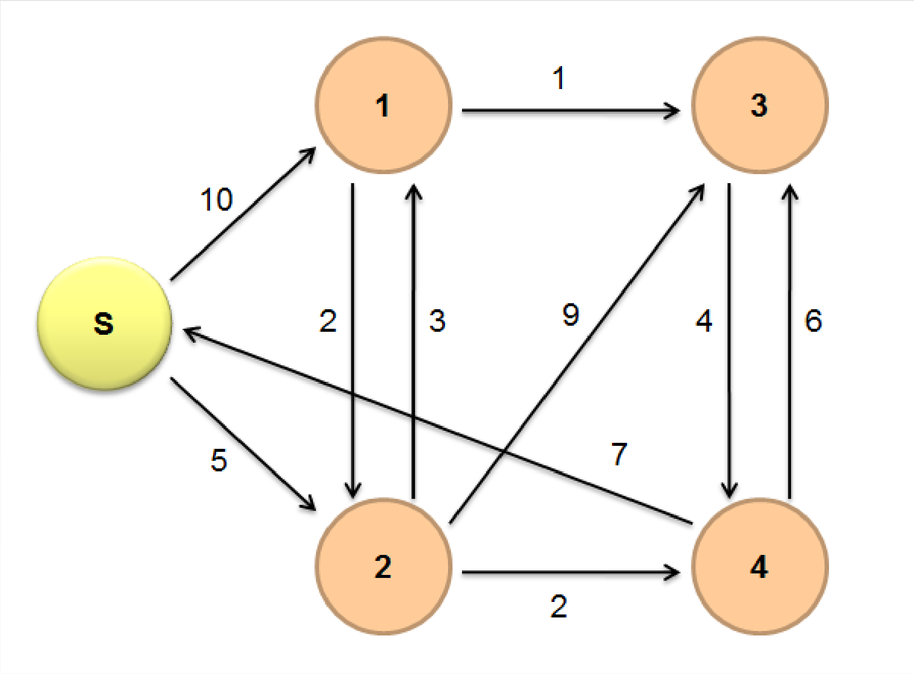
s --> 0 | n1: 10, n2: 5

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)>)

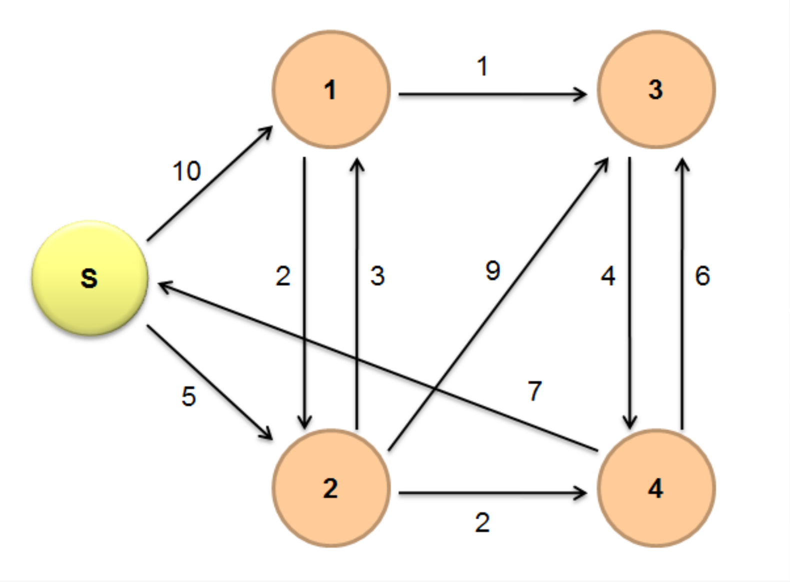
The adjacency list of each node will also be received, ignored in example

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)

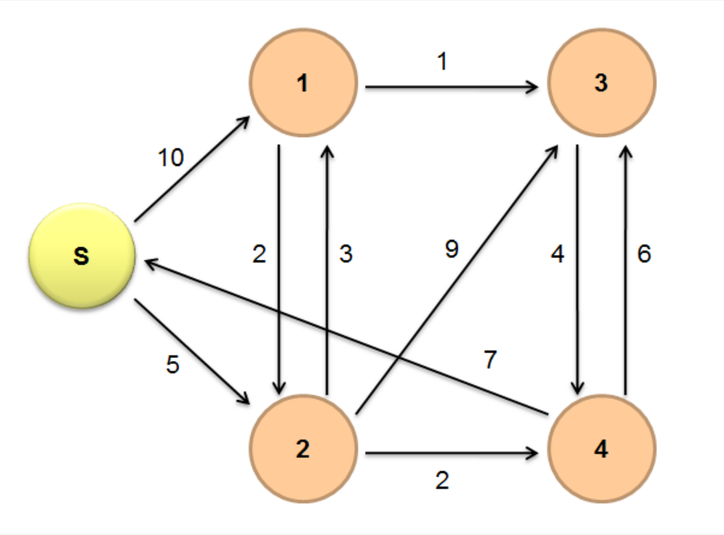
Emit:

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:

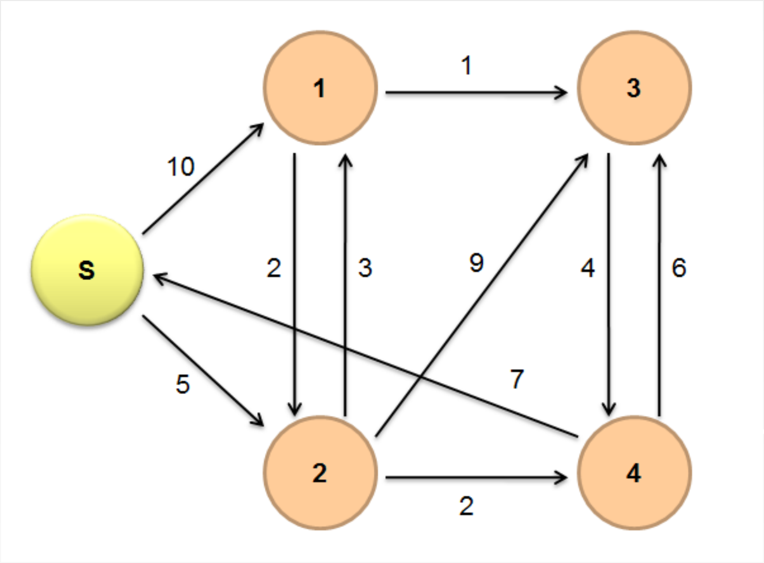
Emit:

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:

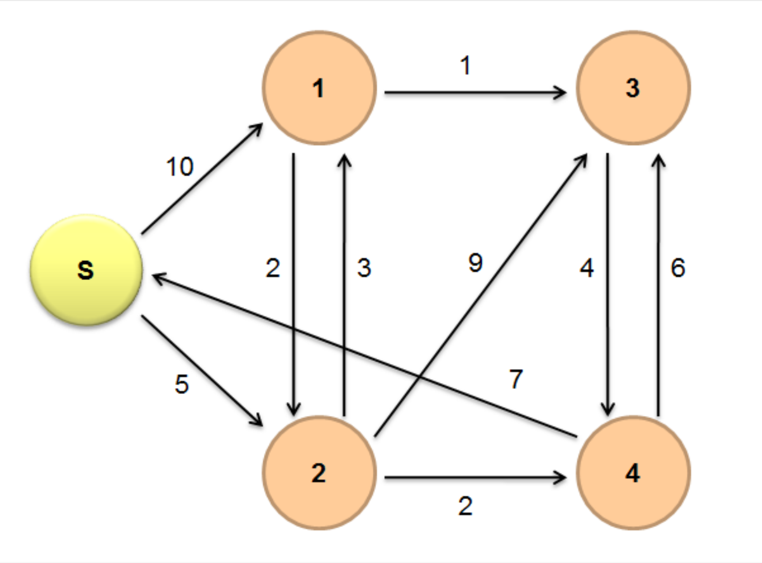
n1 --> 8 | n2: 2, n3:1

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?

References

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

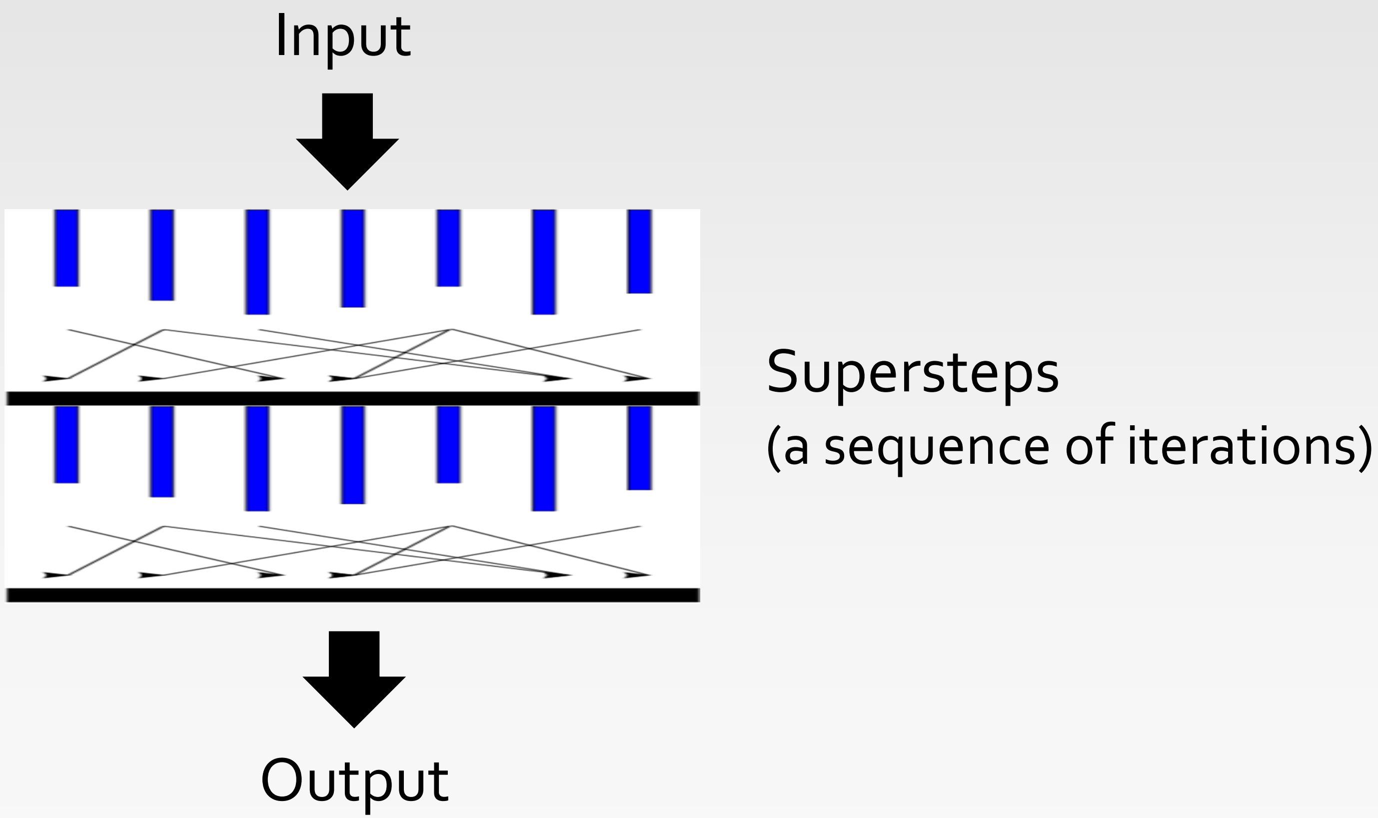
End of Chapter 8.1

Pregel

- **Pregel**: A System for Large-Scale **Graph** Processing (Google) - Malewicz et al. SIGMOD 2010.
- Scalable and Fault-tolerant platform
- API with flexibility to express arbitrary algorithm
- Inspired by Valiant's Bulk Synchronous Parallel model
  - Leslie G. Valiant: A Bridging Model for Parallel Computation. Commun. ACM 33 (8): 103-111 (1990)
- Vertex centric computation (Think like a vertex)

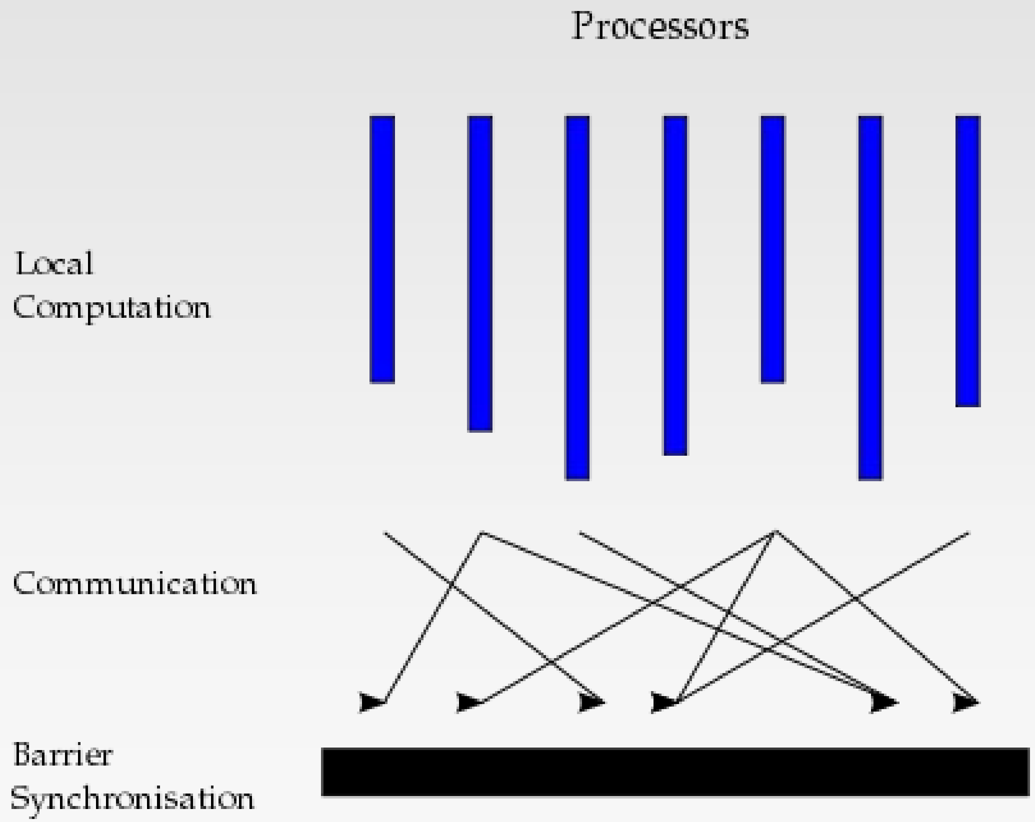
Pregel Computation Model

- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture



Pregel Computation Model (Cont')

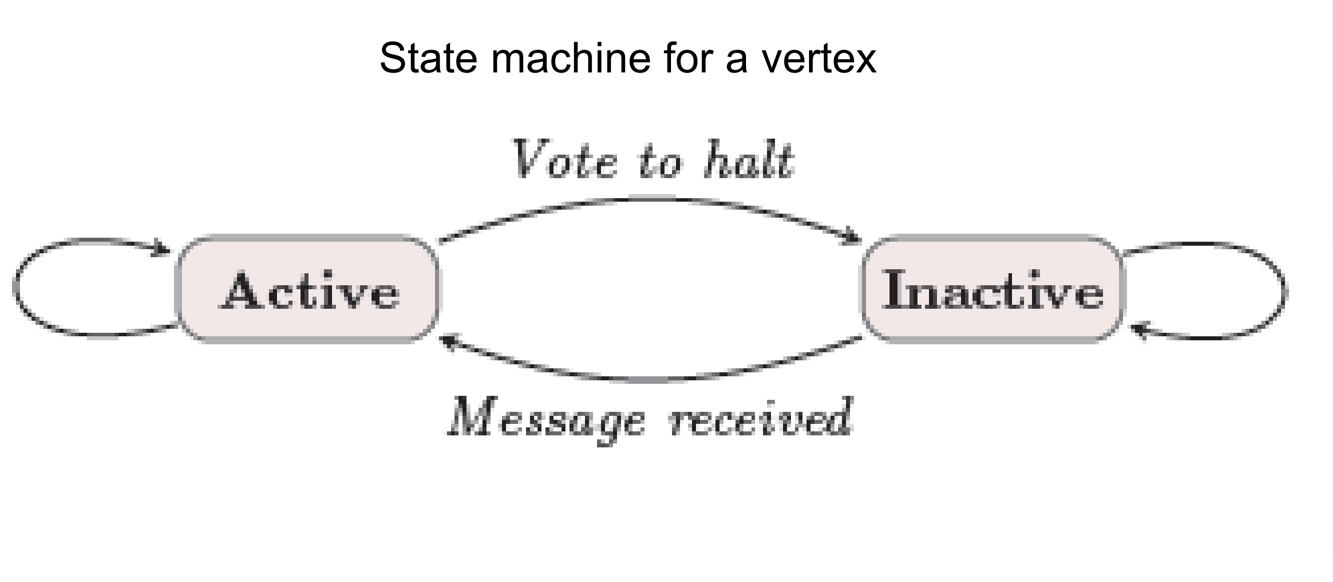
- Concurrent computation and Communication need not be ordered in time
- Communication through message passing



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

Pregel Computation Model (Cont')

- Superstep: the vertices compute in parallel
  - Each vertex



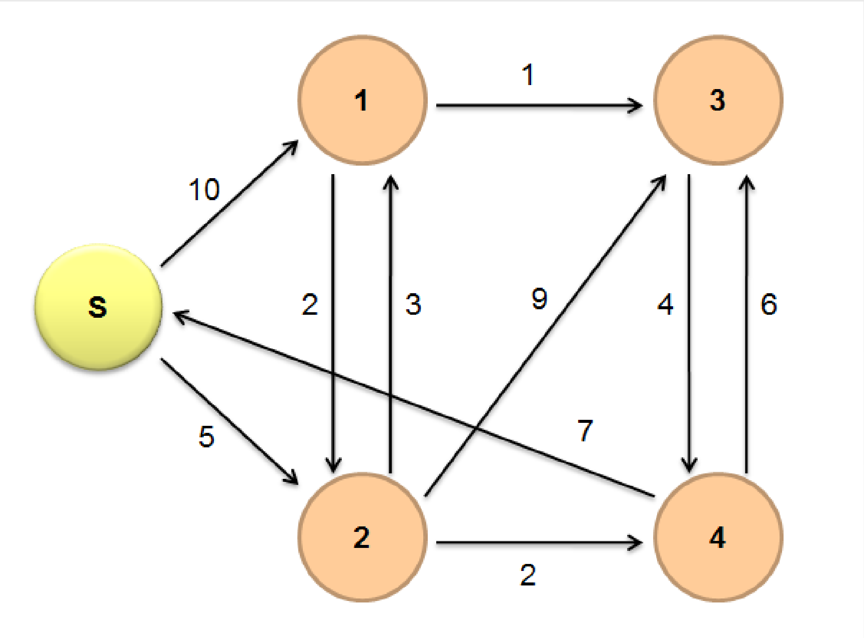
  - Termination condition
    - All vertices are simultaneously inactive
    - A vertex can choose to deactivate itself
    - Is "woken up" if new messages received

Superstep

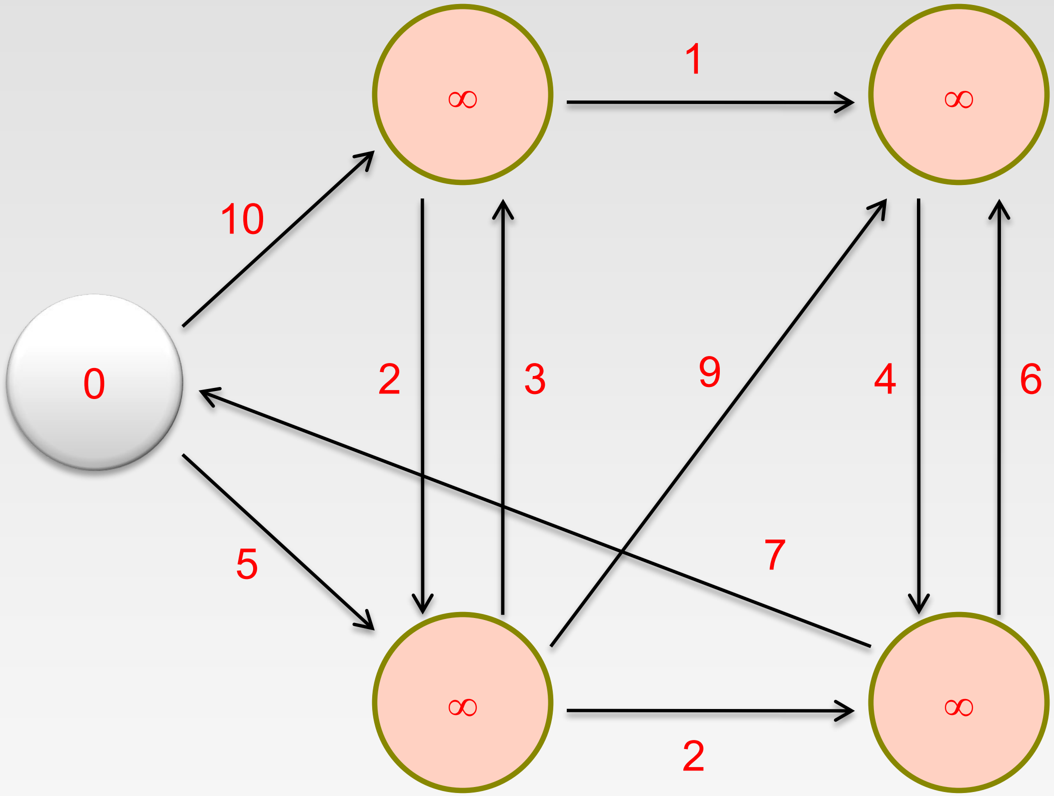
- During a superstep, the following can happen in the framework:
  - It receives and reads messages that are sent to v from the previous superstep s-1.
  - It applies a user-defined function f to each vertices in parallel, so f essentially specifies the behaviour of a single vertex v at a single superstep s.
  - It can mutate the state of v.
  - It can send messages to other vertices (typically along outgoing edges) that the vertices will receive in the next superstep s+1.
- All communications are between supersteps s and s+1

Single-Source Shortest Path (SSSP)

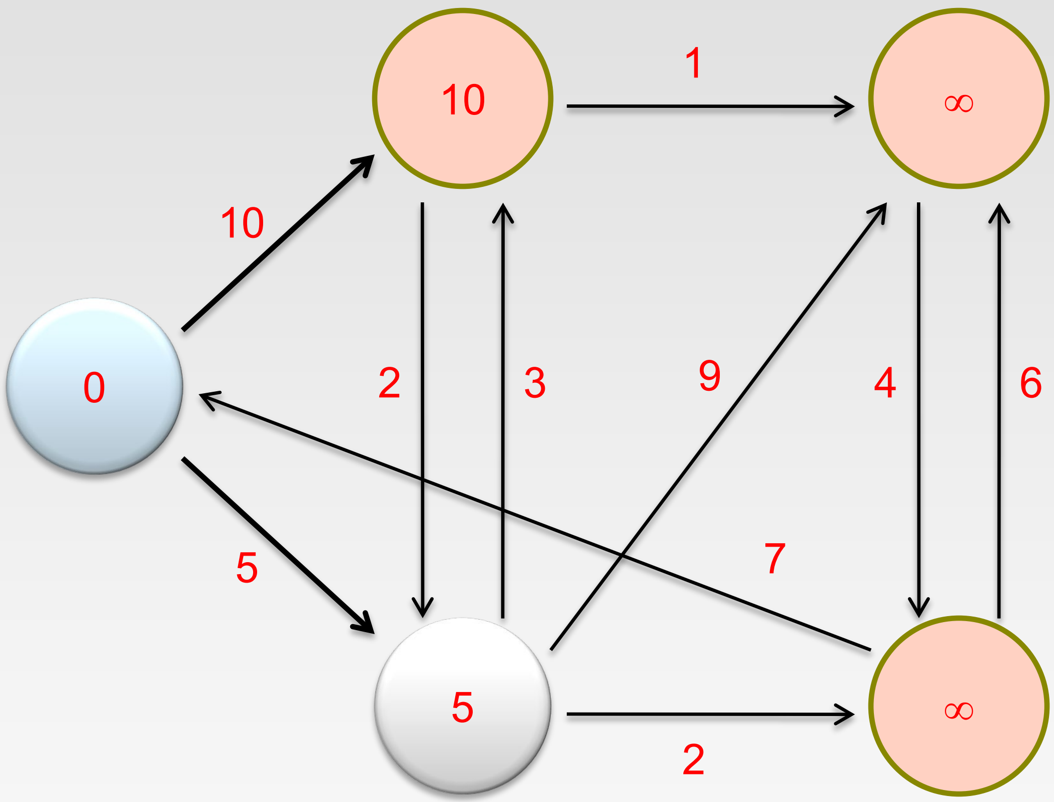
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Dijkstra's Algorithm:
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



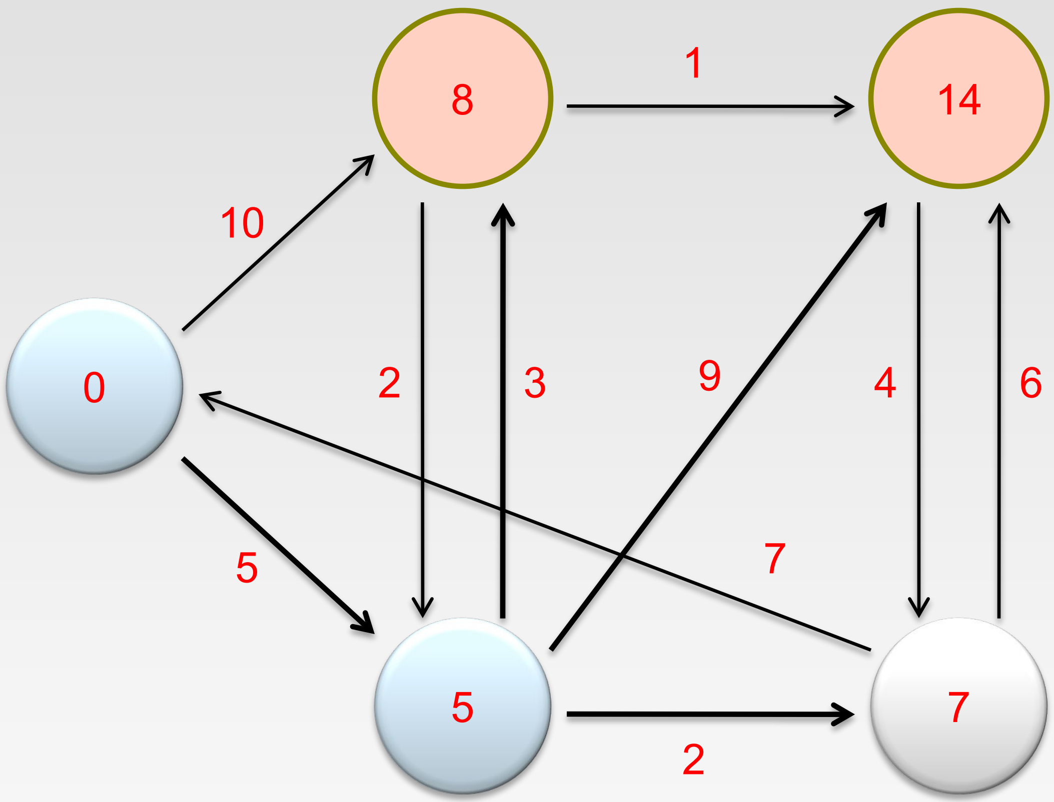
Dijkstra's Algorithm Example



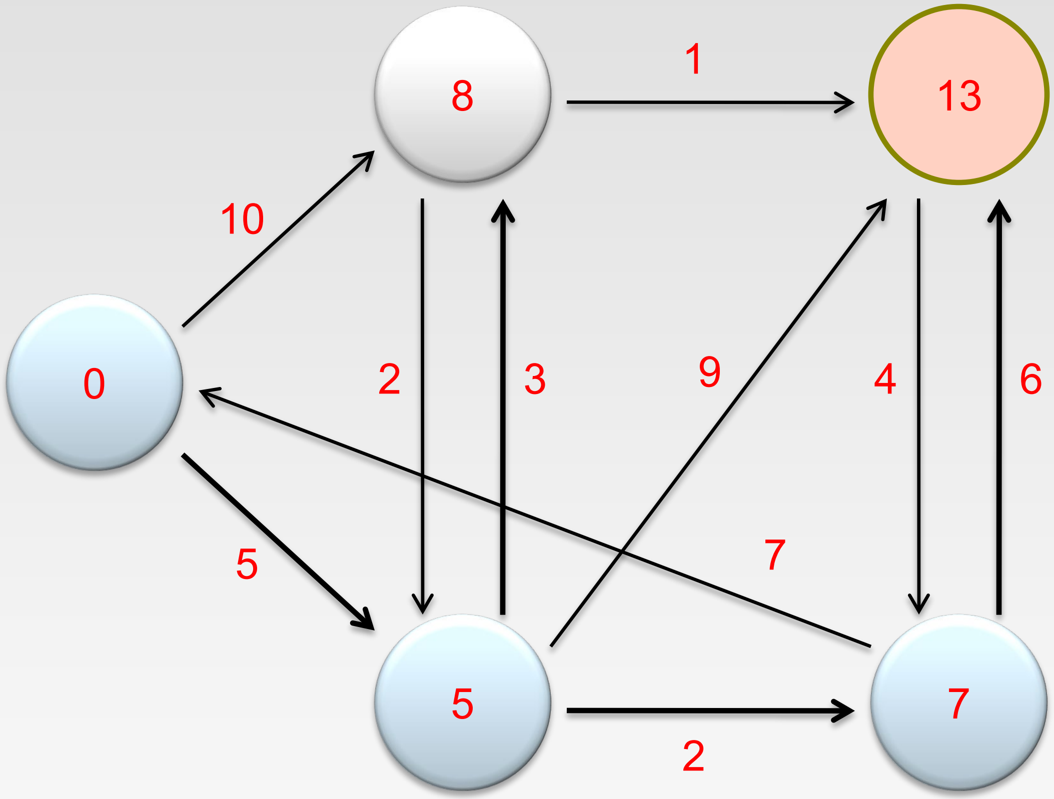
Dijkstra's Algorithm Example



Dijkstra's Algorithm Example

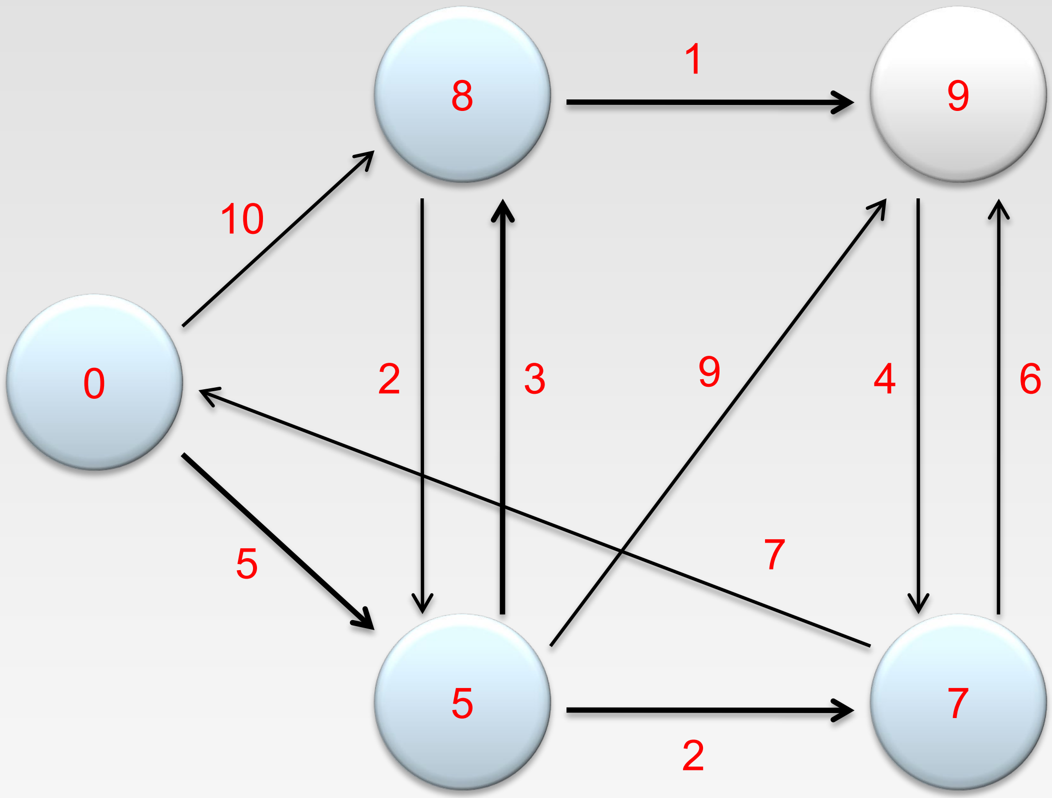


Dijkstra's Algorithm Example

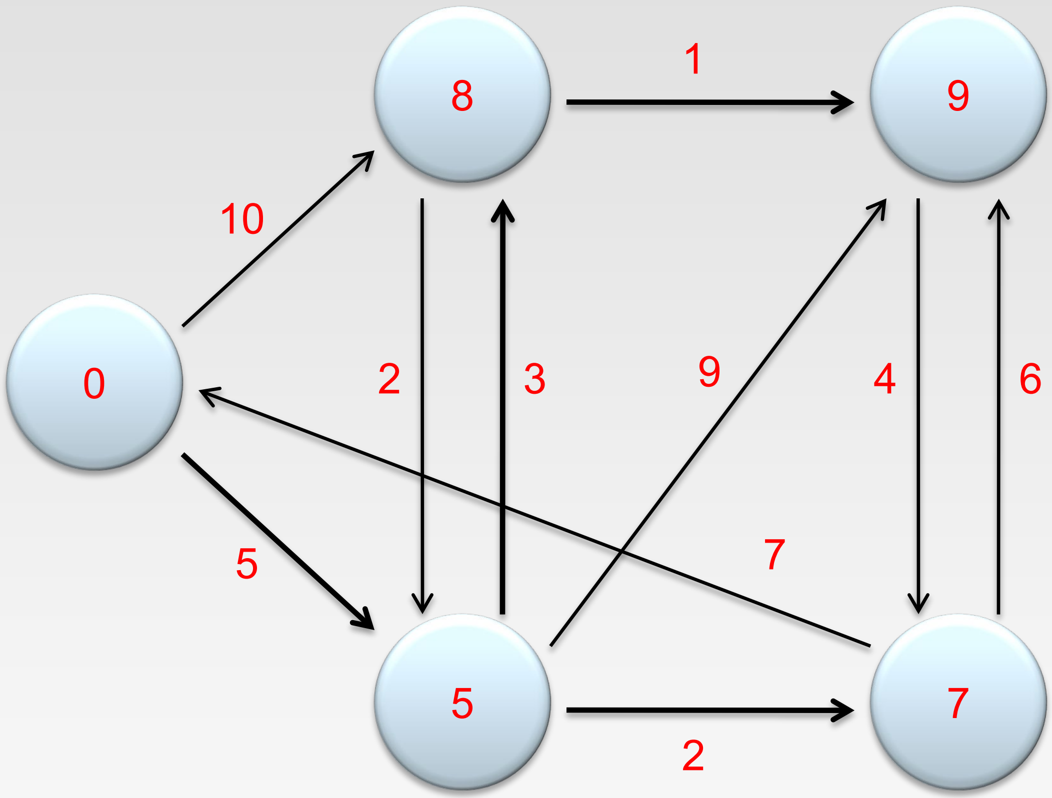


Example from CLR

Dijkstra's Algorithm Example

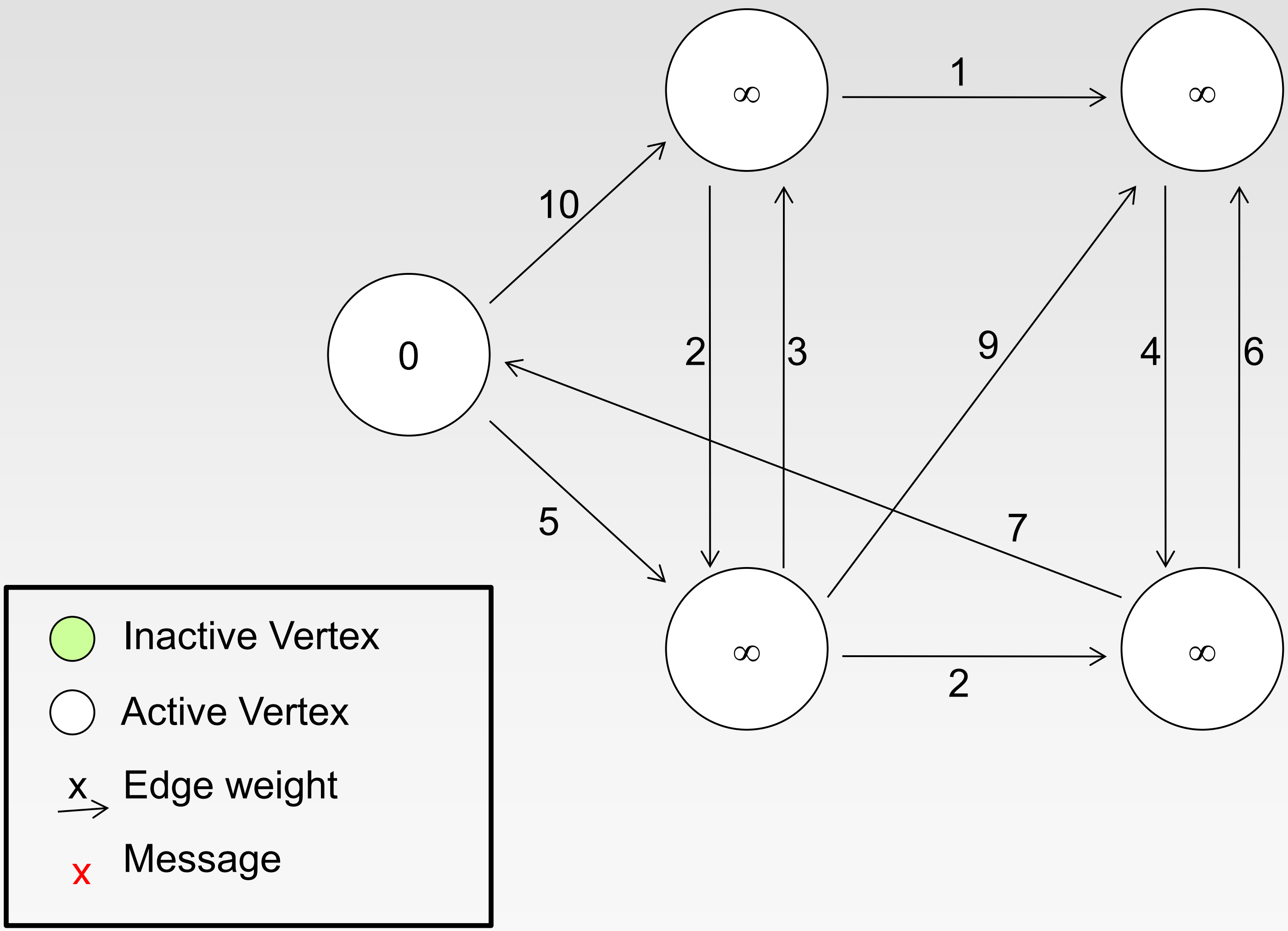


Dijkstra’s Algorithm Example

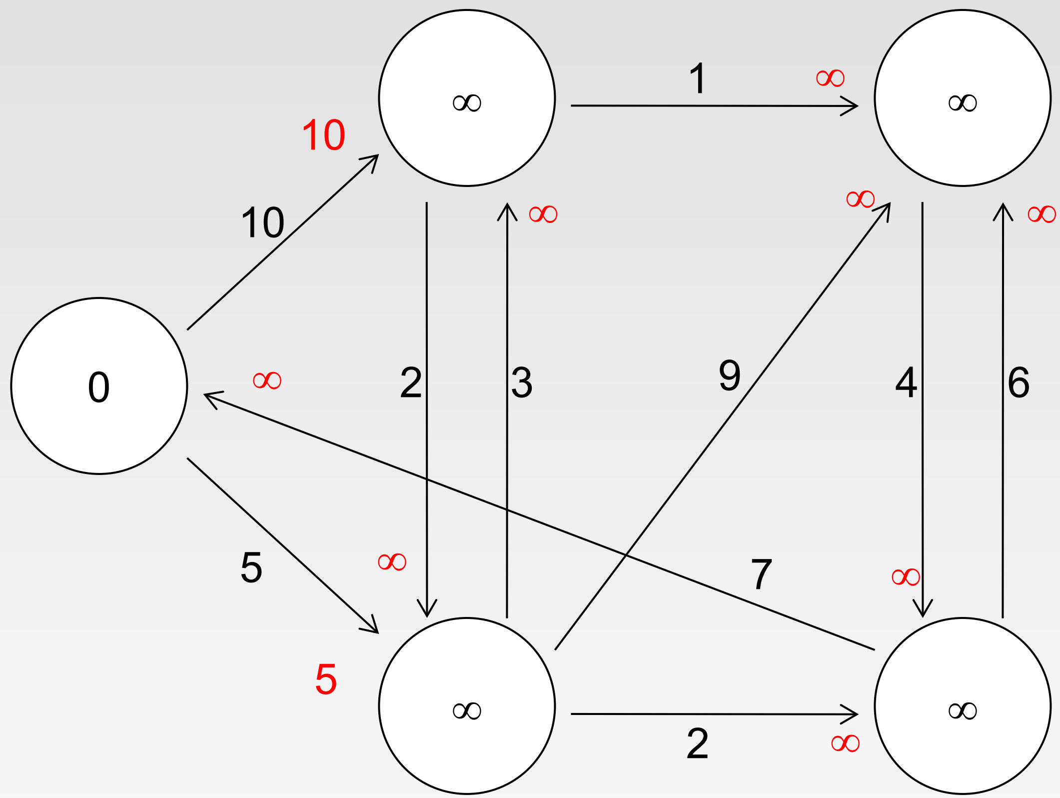


Finish!

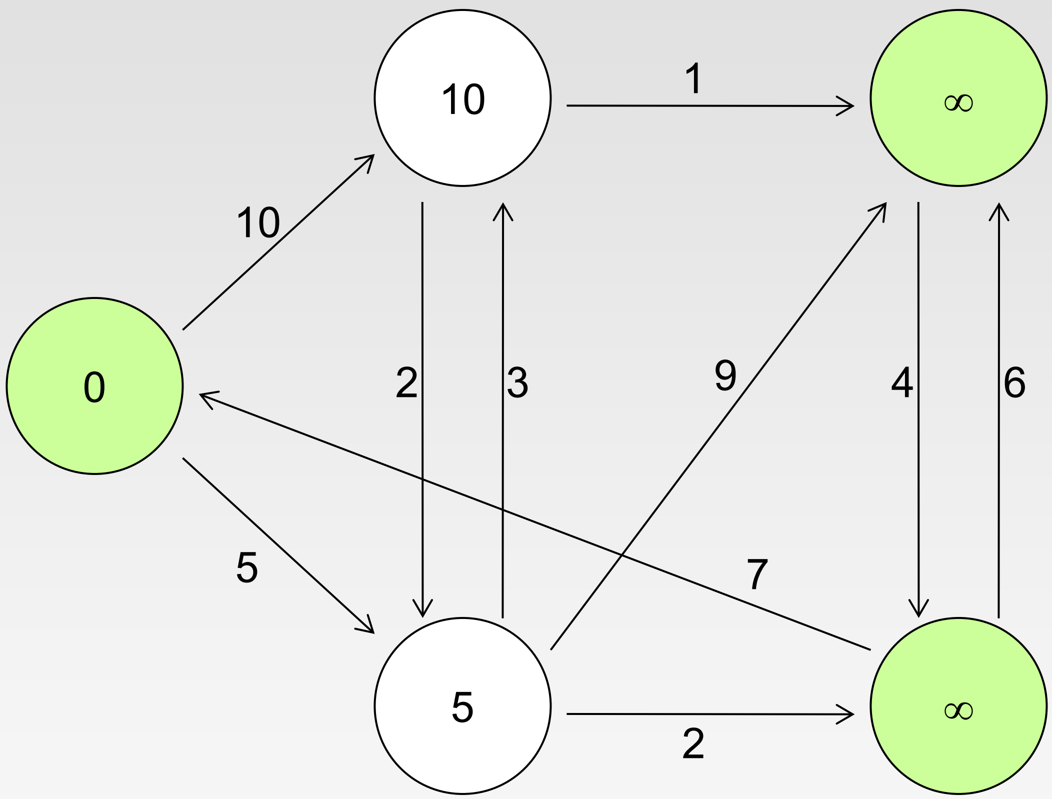
Example: SSSP – Parallel BFS in Pregel



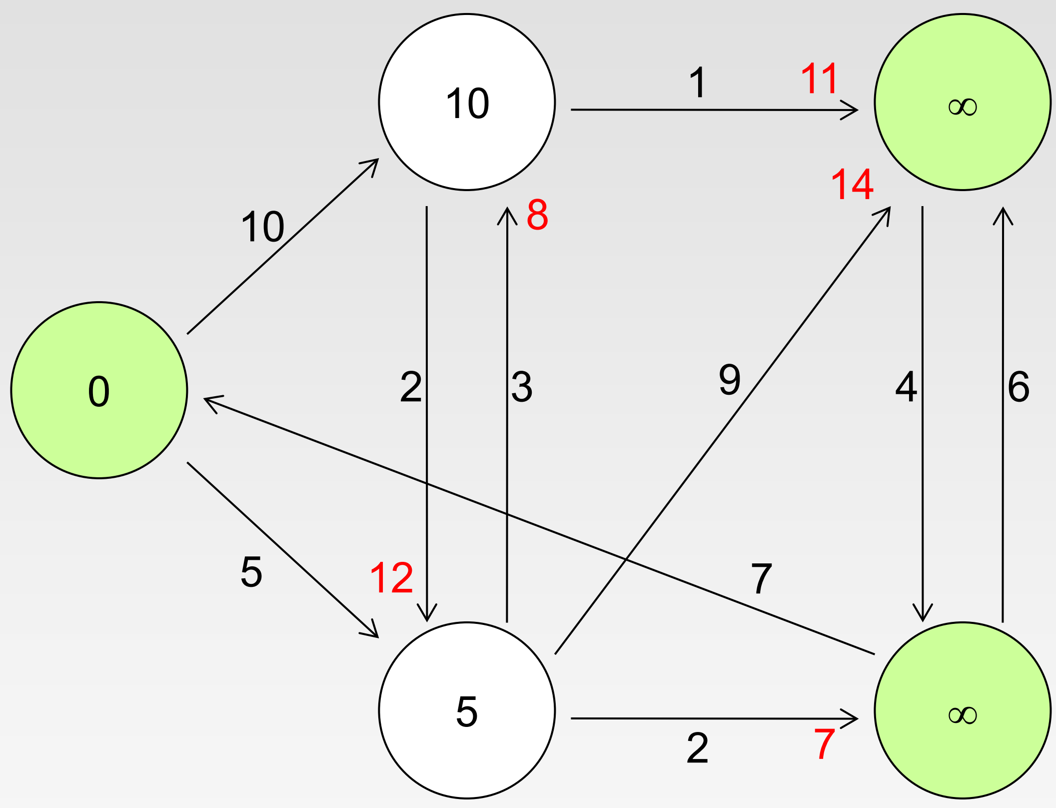
Example: SSSP – Parallel BFS in Pregel



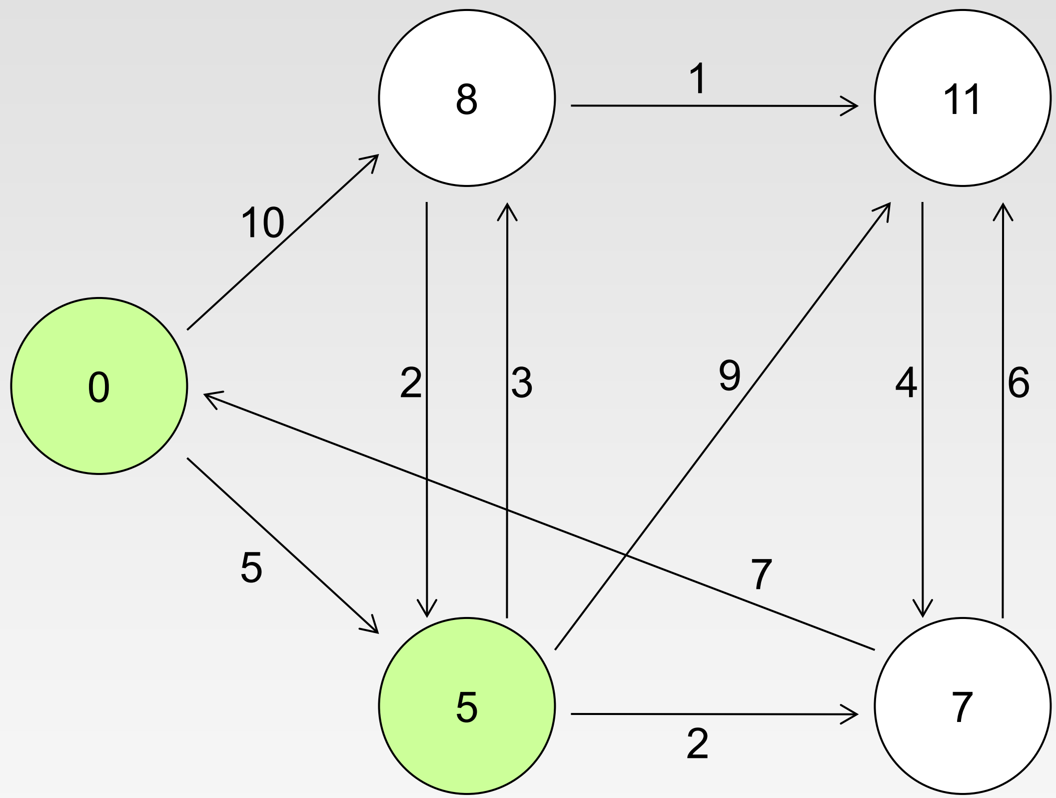
Example: SSSP – Parallel BFS in Pregel



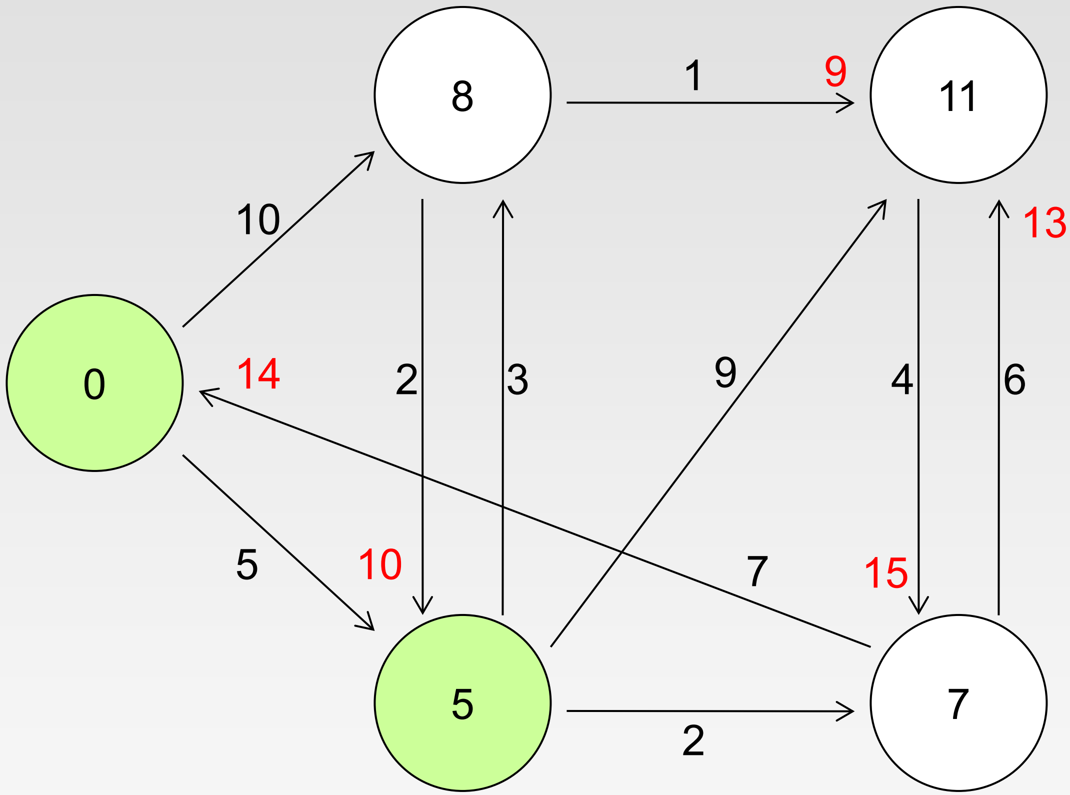
Example: SSSP – Parallel BFS in Pregel



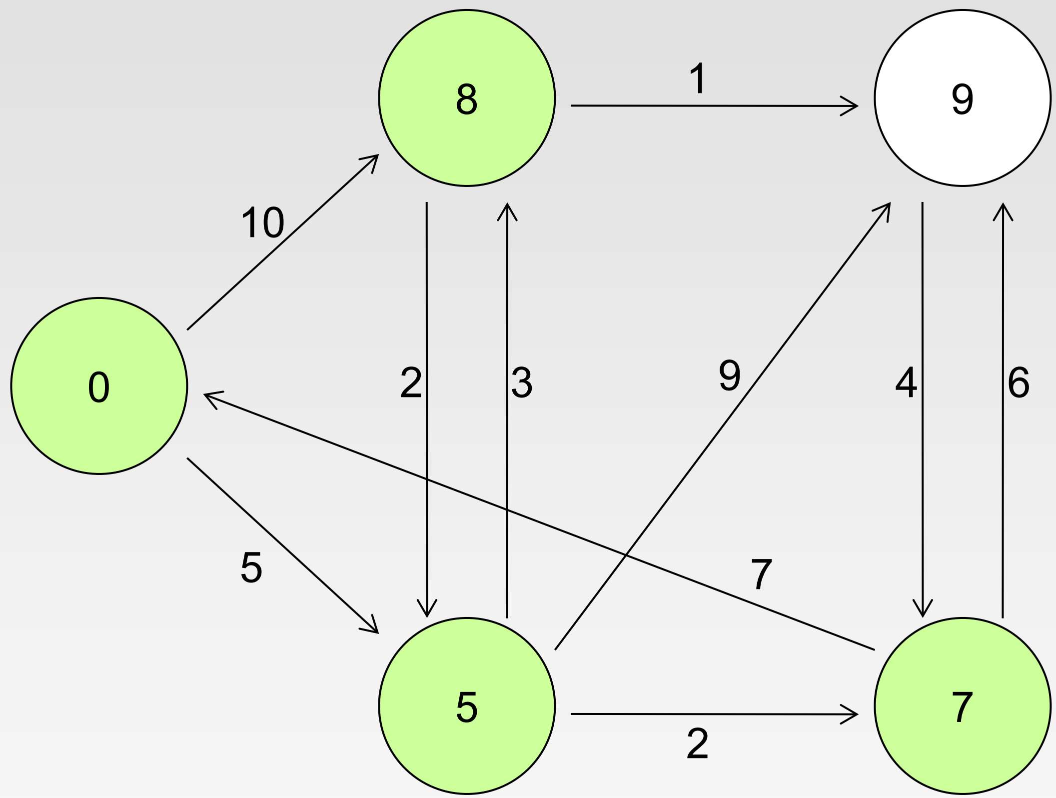
Example: SSSP – Parallel BFS in Pregel



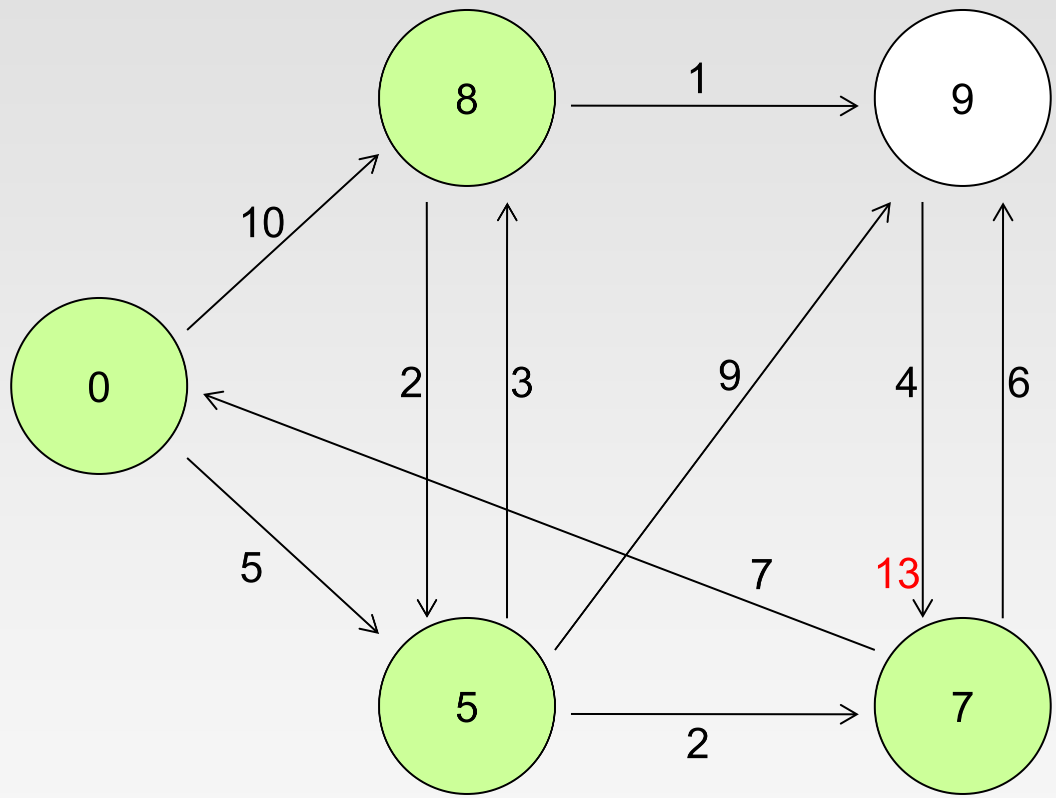
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel