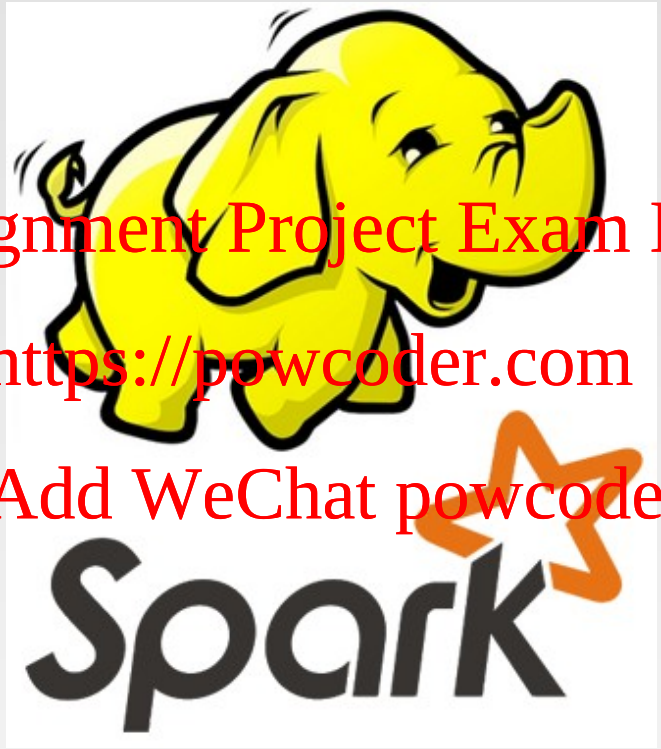
COMP9313: Big Data Management



Lecturer: Xin Cao

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

Chapter 5: Graph Data Processing in MapReduce

What's a Graph?

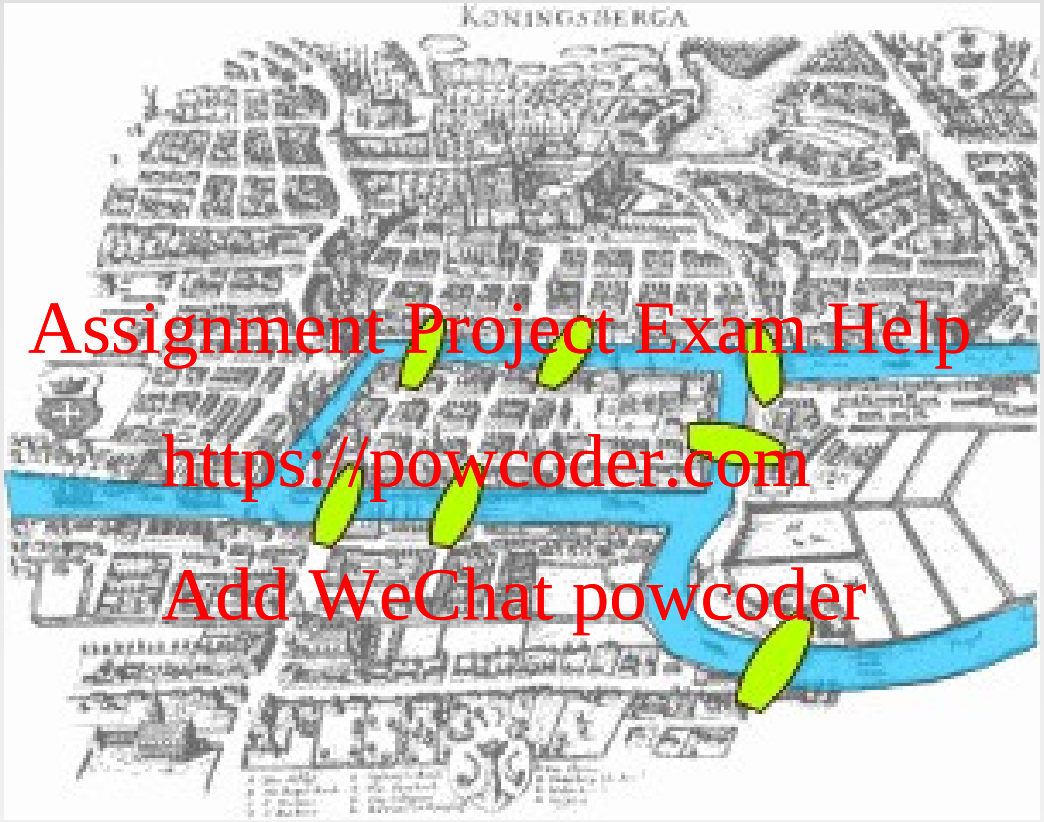
- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles
- Graphs are everywhere:
  - Hyperlink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

Graph Data: Social Networks

Facebook social graph

4-degrees of separation [Backstrom-Boldi-Rosa-Ugander-Vigna, 2011]

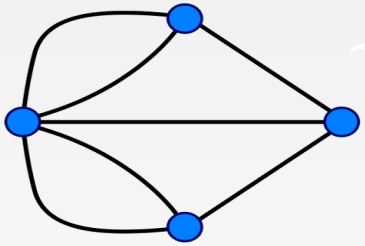
Graph Data: Technological Networks



Seven Bridges of Königsberg

[Euler, 1735]

Return to the starting point by traveling each link of the graph once and only once.



Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank

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 web page 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;
- … …

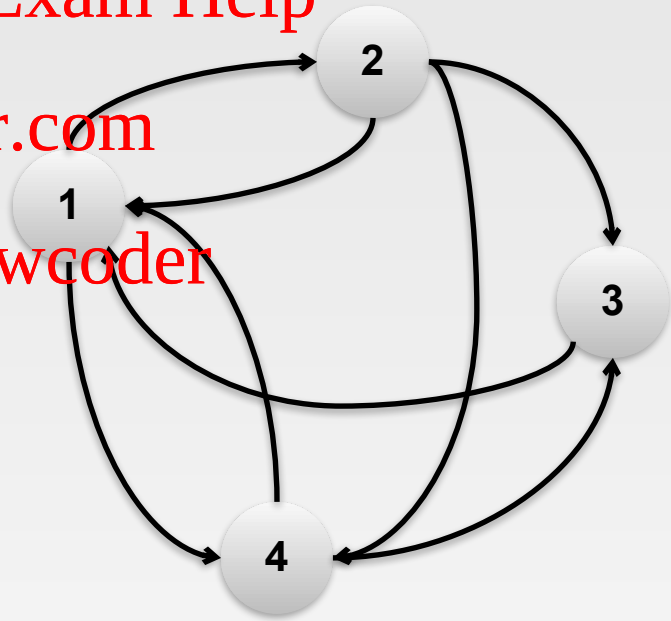
Graphs and MapReduce

- 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

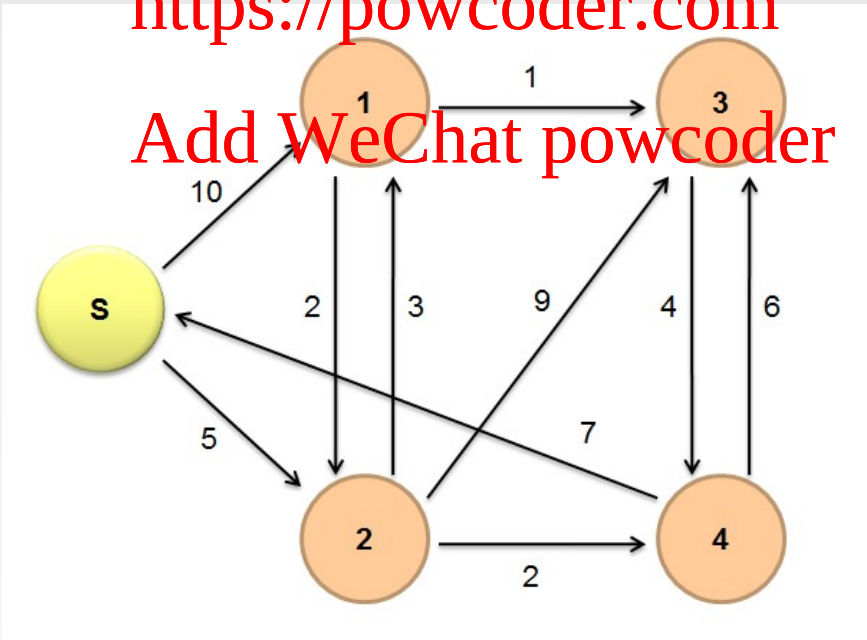
Adjacency Lists: Critique

- 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)

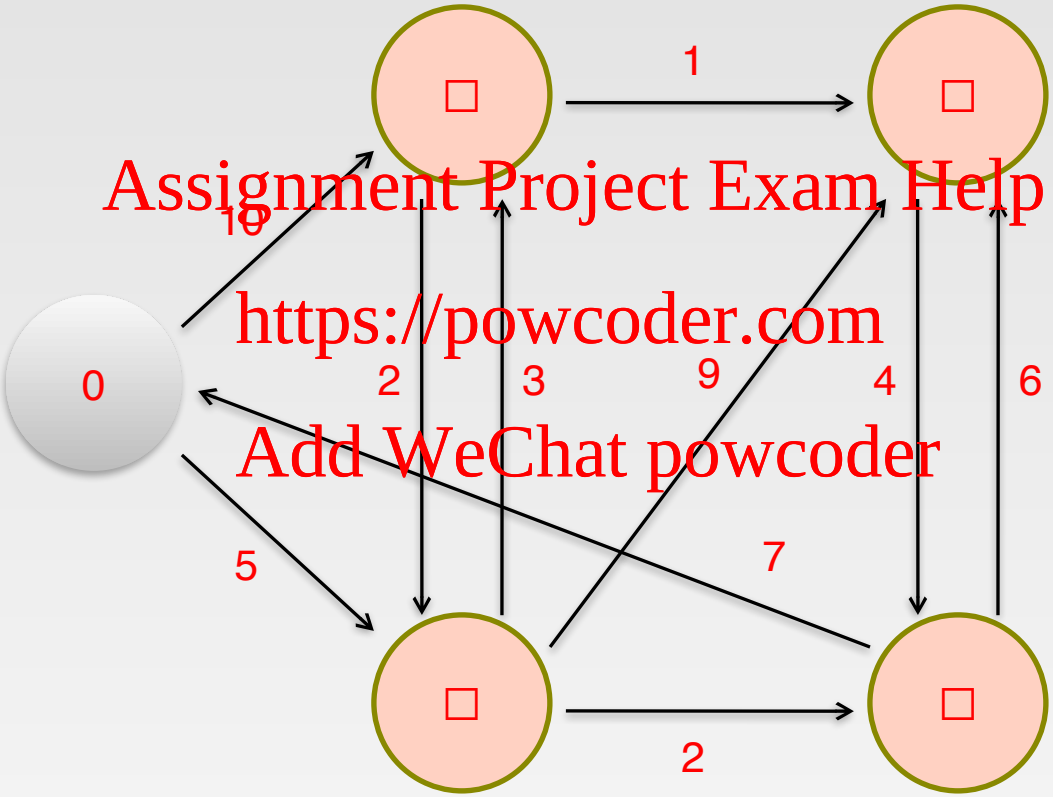
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Dijkstra's Algorithm:
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



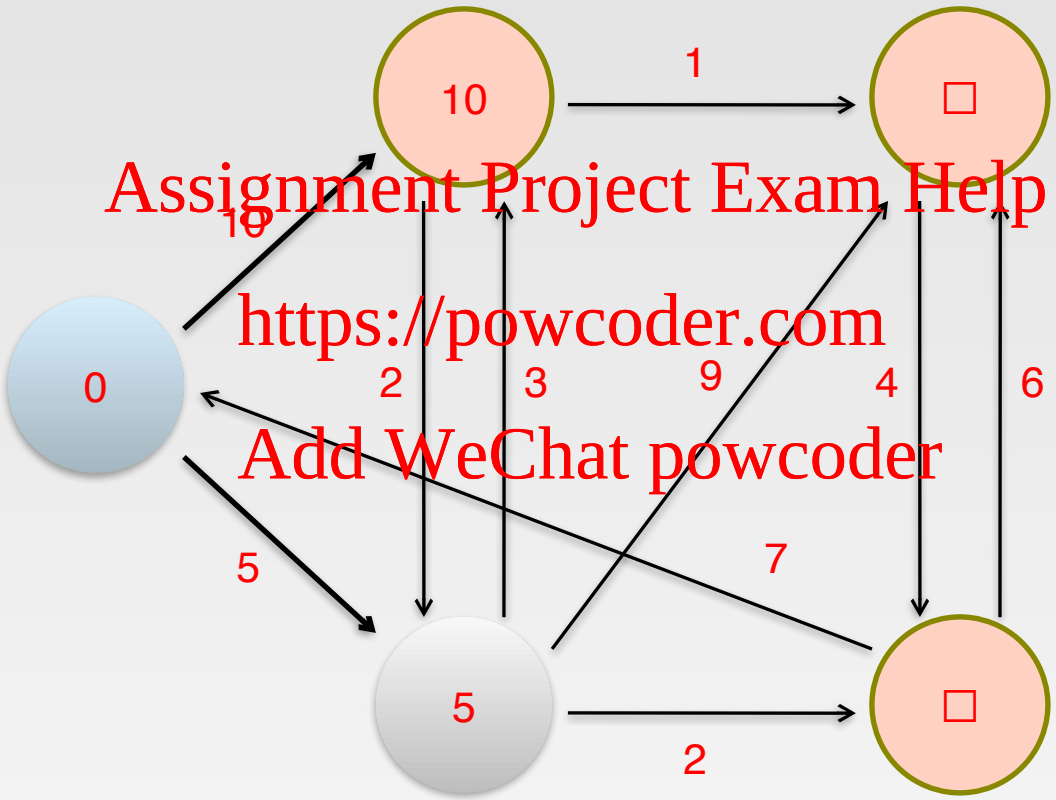
Dijkstra's Algorithm

```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0
3:      for all vertex v ∈ V do
4:          d[v] ← ∞
5:      Q ← {V}
6:      while Q ≠ ∅ do
7:          u ← EXTRACTMIN(Q)
8:          for all vertex v ∈ u.ADJACENCYLIST do
9:              if d[v] > d[u] + w(u, v) then
10:                 d[v] ← d[u] + w(u, v)
```

Dijkstra's Algorithm Example

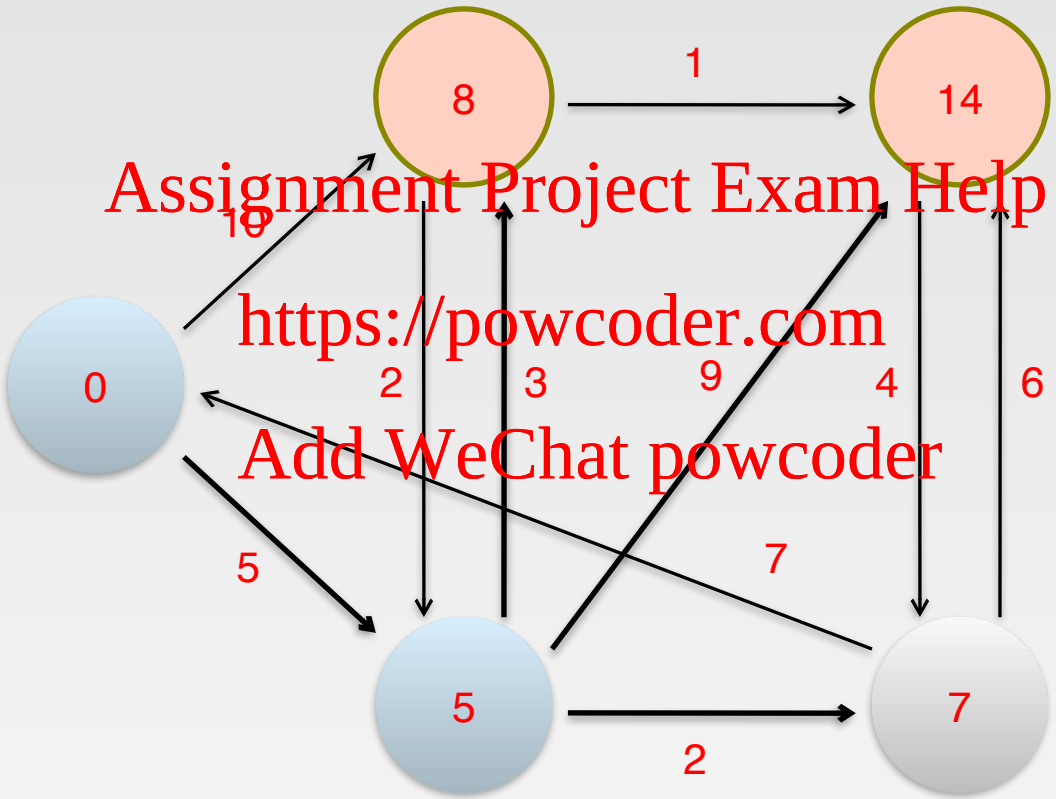


Dijkstra's Algorithm Example

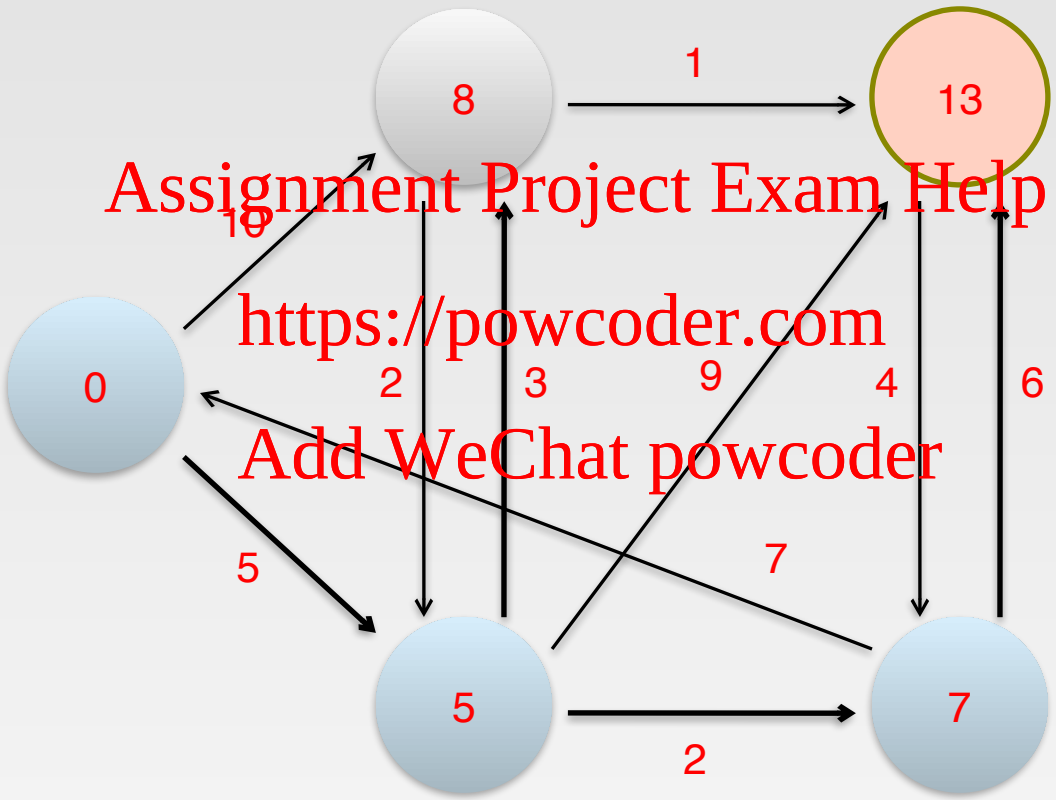


Example from CLR

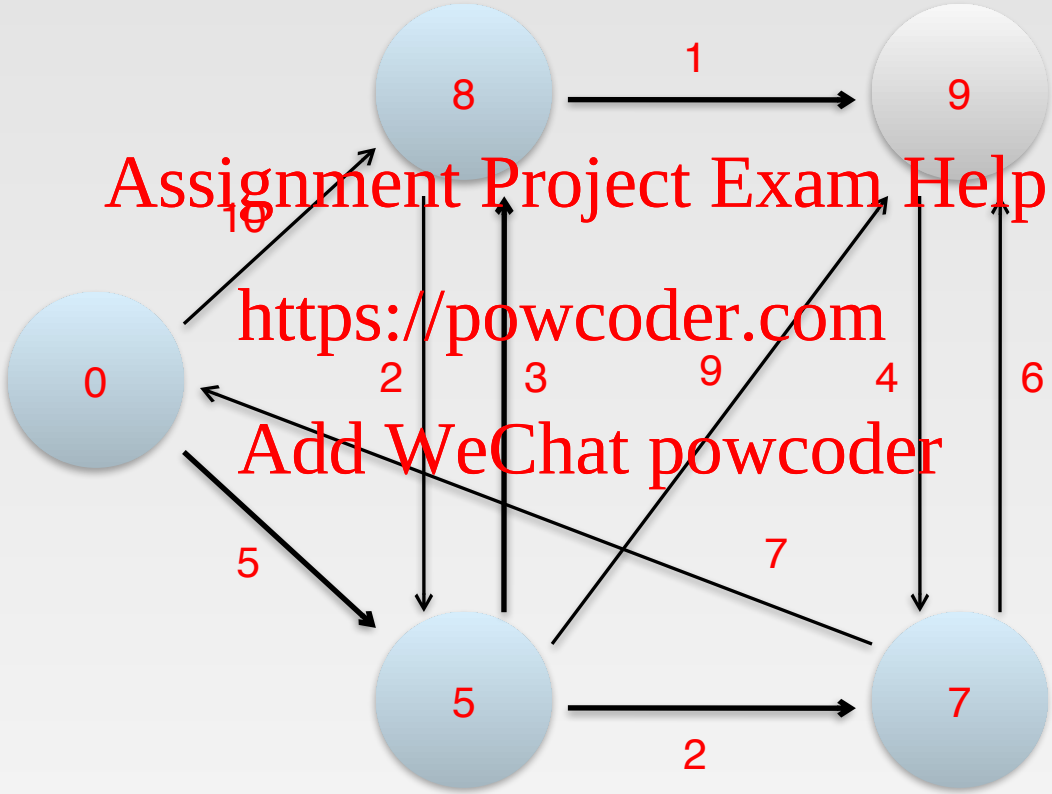
Dijkstra's Algorithm Example



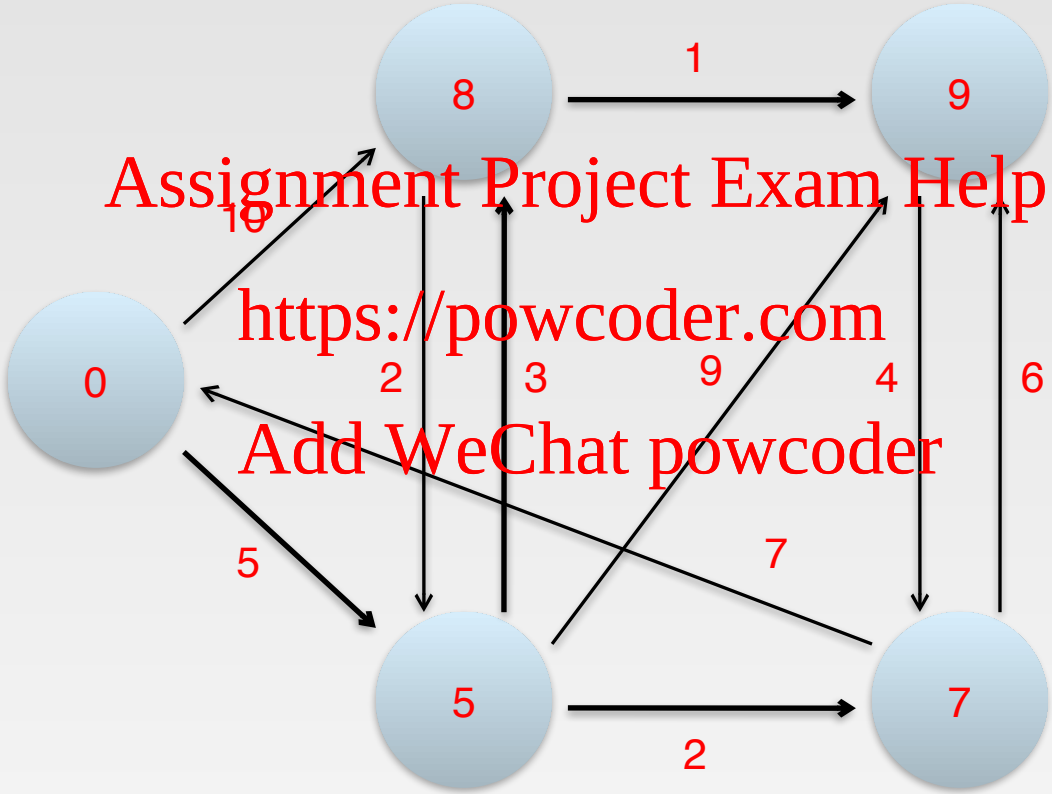
Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



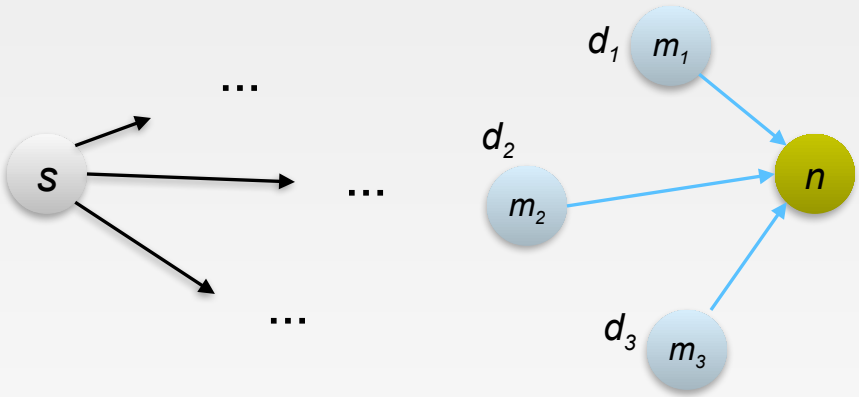
Finish!

Single Source Shortest Path

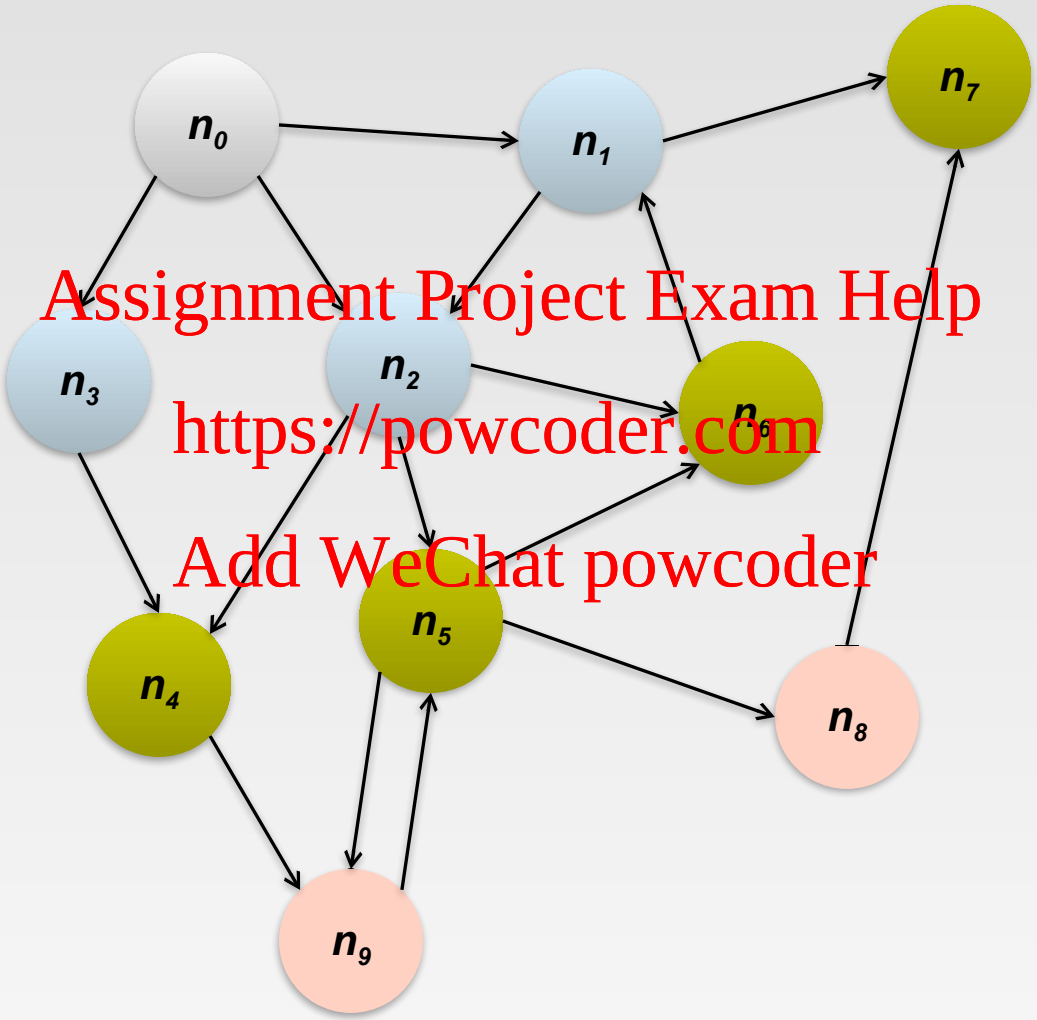
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

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*
  - 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)                          //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 ← 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.

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

Mapper

```
public static class TheMapper extends Mapper<LongWritable, Text, LongWritable, Text> {
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        Text word = new Text();
        String line = value.toString();//looks like 1 0 2:3:
        String[] sp = line.split(" ");//splits on space
        int distanceadd = Integer.parseInt(sp[1]) + 1;
        String[] PointsTo = sp[2].split(":");
        for(int i=0; i<PointsTo.length; i++){
            word.set("VALUE "+distanceadd);//tells me to look at distance value
            context.write(new LongWritable(Integer.parseInt(PointsTo[i])), word);
            word.clear(); }
        //pass in current node's distance (if it is the lowest distance)
        word.set("VALUE "+sp[1]);
        context.write( new LongWritable( Integer.parseInt( sp[0] ) ), word );
        word.clear();
        word.set("NODES "+sp[2]);//tells me to append on the final tally
        context.write( new LongWritable( Integer.parseInt( sp[0] ) ), word );
        word.clear();
    }
}
```

Reducer

```
public static class TheReducer extends Reducer<LongWritable, Text, LongWritable, Text> {
    public void reduce(LongWritable key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        String nodes = "UNMODED";
        Text word = new Text();
        int lowest = INFINITY;//start at infinity

        for (Text val : values) {
            //looks like NODES/VALUES 1 0 2:3:, we need to use the first as a key
            String[] sp = val.toString().split(" ");//splits on space
            //look at first value
            if(sp[0].equalsIgnoreCase("NODES")){
                nodes = null;
                nodes = sp[1];
            }else if(sp[0].equalsIgnoreCase("VALUE")){
                int distance = Integer.parseInt(sp[1]);
                lowest = Math.min(distance, lowest);
            }
        }
        word.set(lowest+" "+nodes);
        context.write(key, word);
        word.clear();
    }
}
```


https://github.com/himank/Graph-Algorithm-Map Reduce/blob/master/src/DijikstraAlgo.java

BFS Pseudo-Code (Weighted Edges)

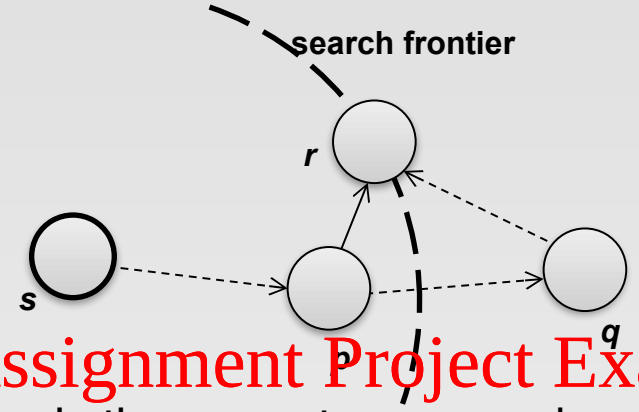
- The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well
  - Positive weights!

- 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!

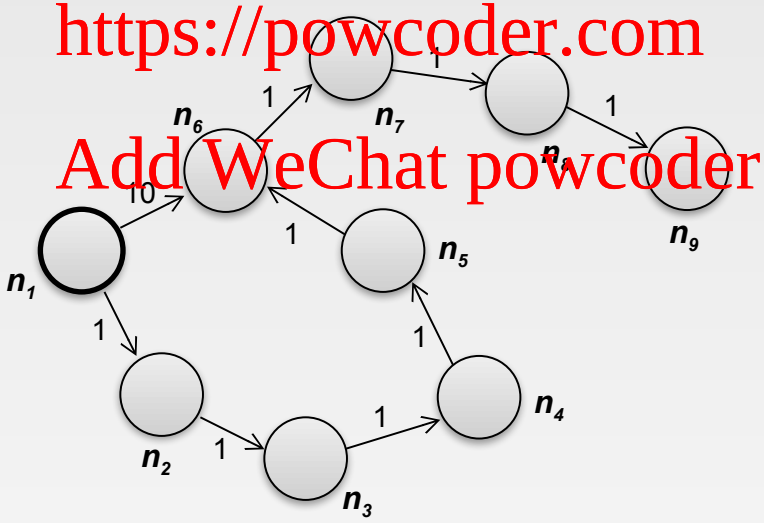
Additional Complexities



- 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:

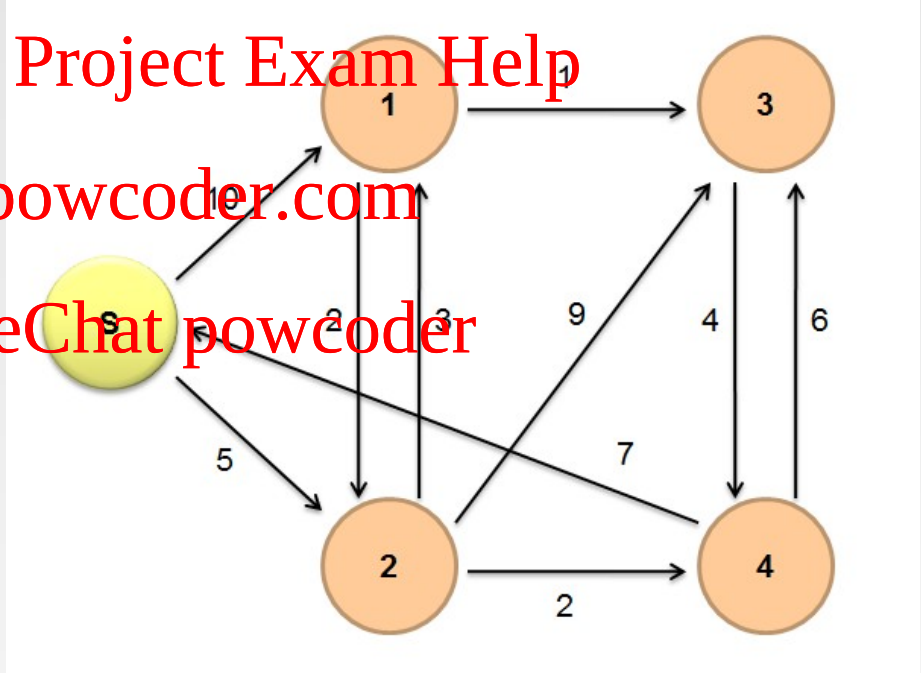
s --> 0 | n1: 10, n2: 5

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)>)

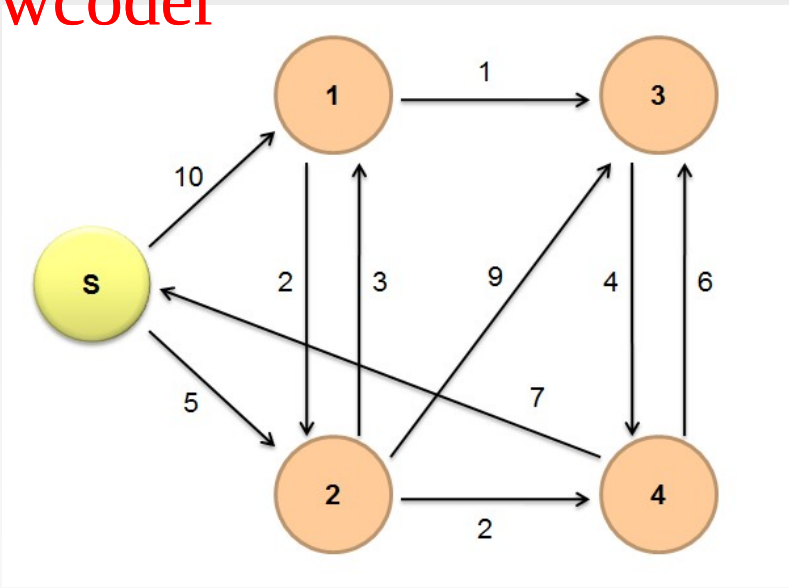
The adjacency list of each node will also be received, ignored in example

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)

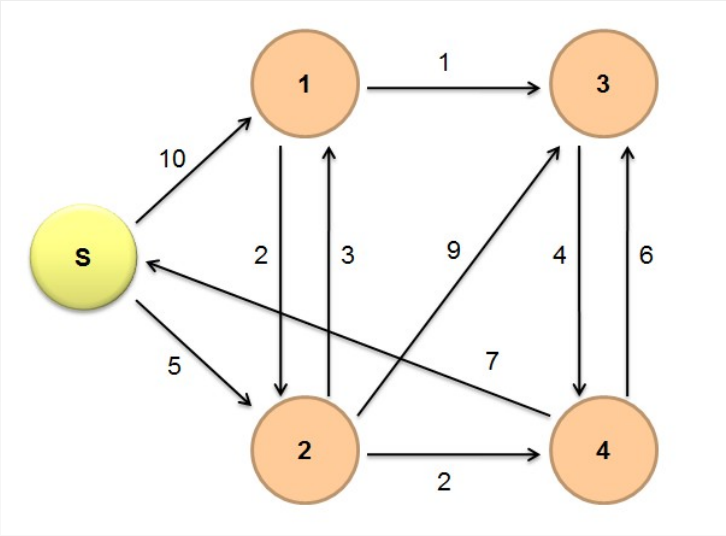
Emit:

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:

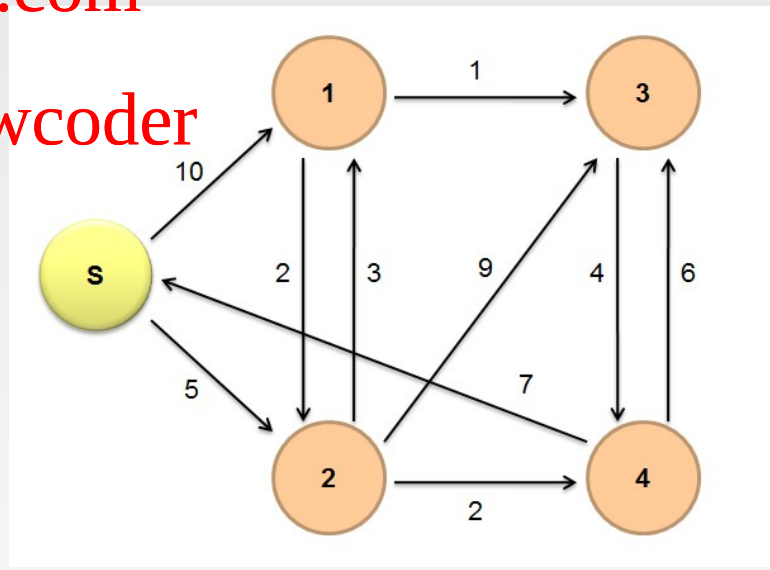
Emit:

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 l n2: 2, n3:1 (**Again!**)

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

Read: n2 --> 5 l 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 l n4:4

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

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

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

- Reduce:

Emit:

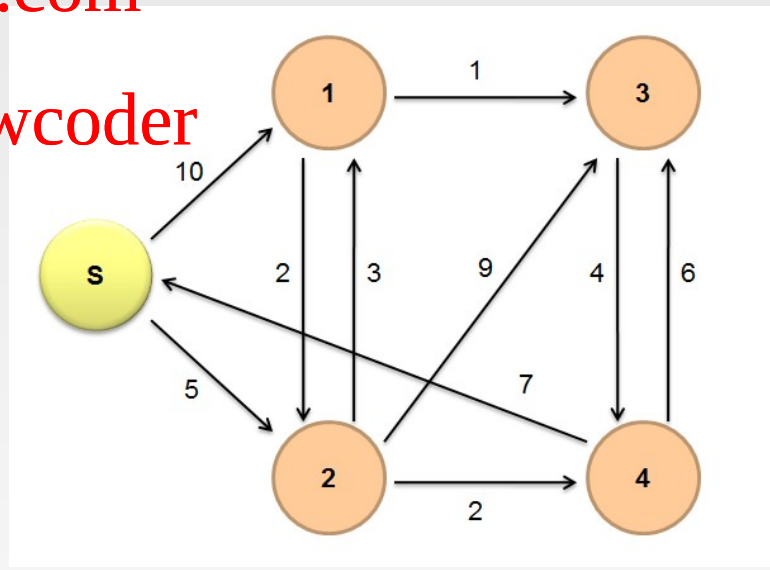
n1 --> 8 l n2: 2, n3:1

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

n3 --> 9 l n4:4

n4 --> 7 l 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

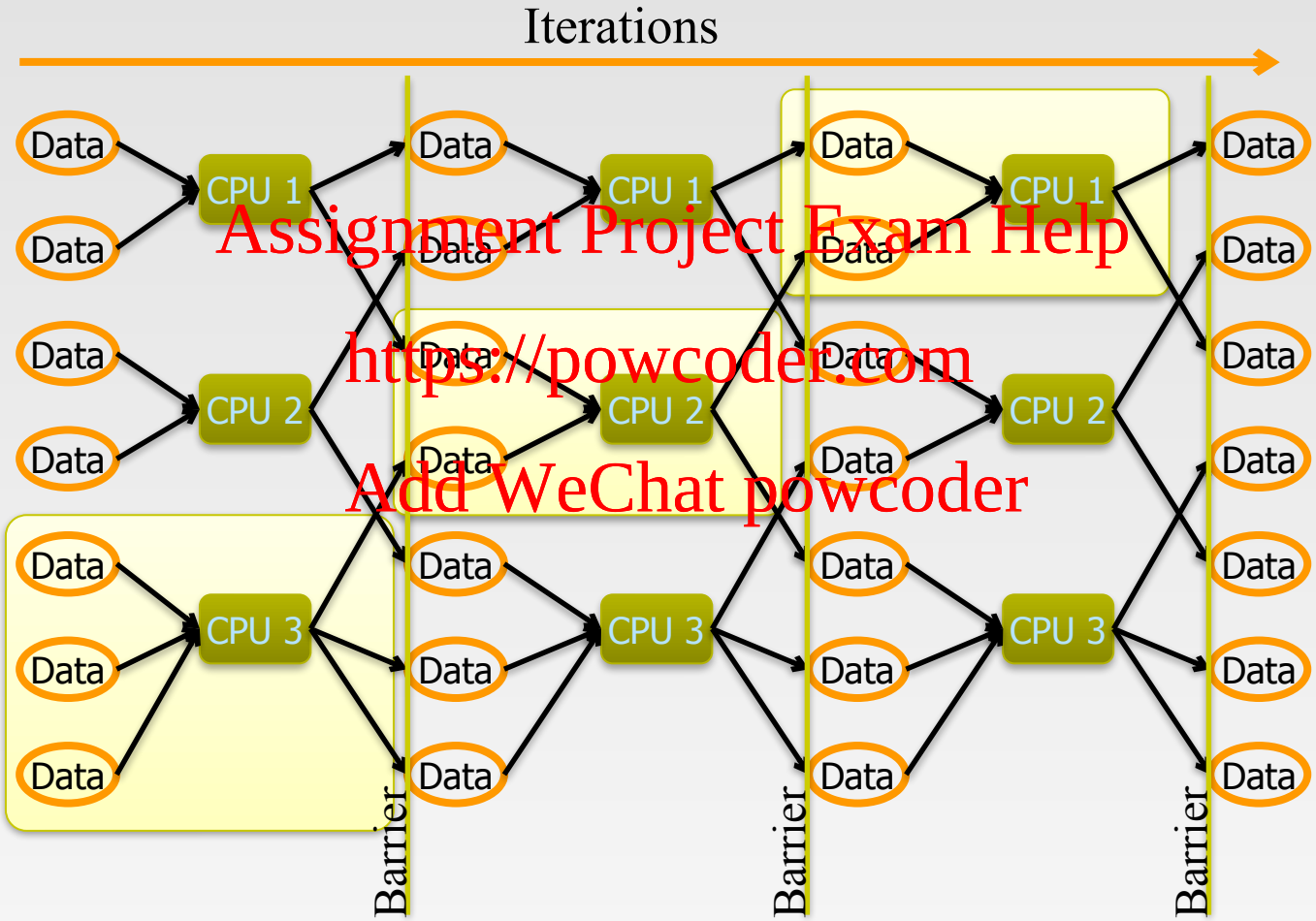
- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass 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.

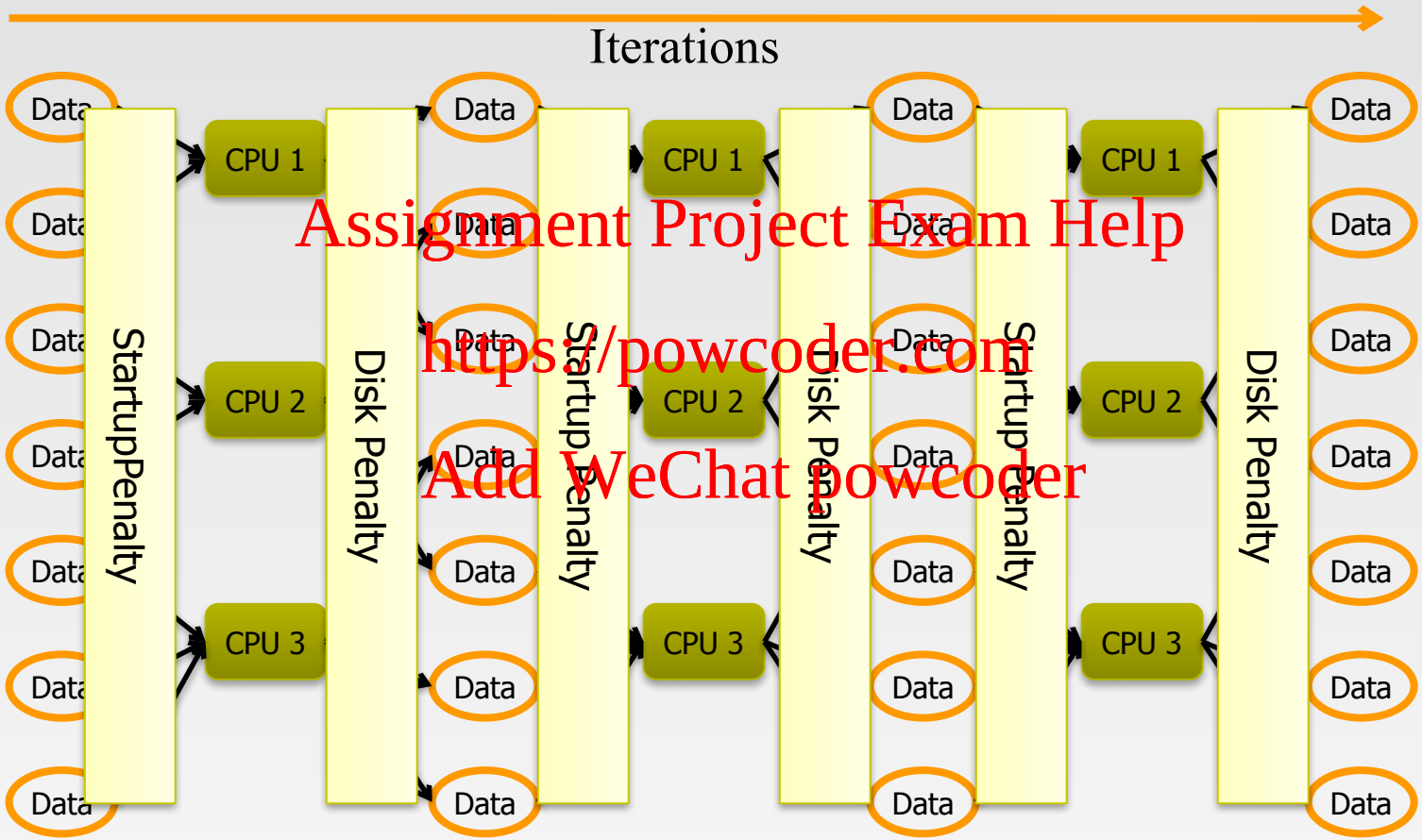
Iterative MapReduce

- Only a subset of data needs computation:



Iterative MapReduce

- System is not optimized for iteration:



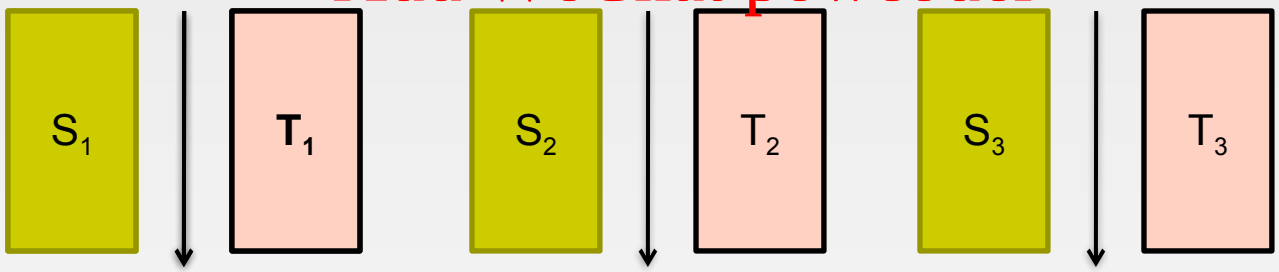
Better Partitioning

- 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

Schimmy Design Pattern

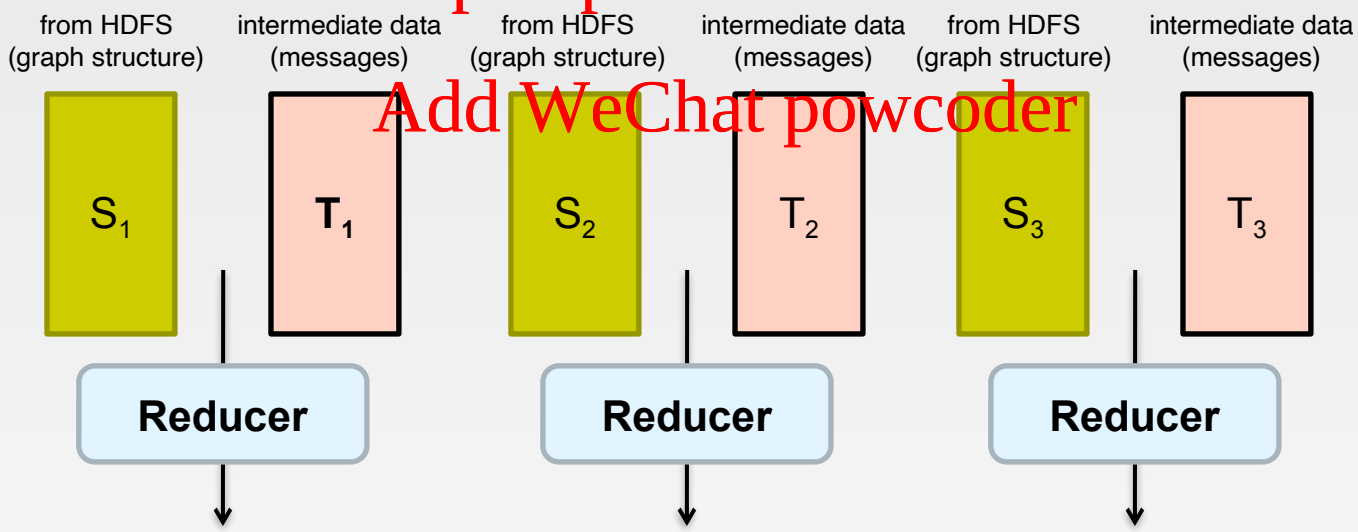
- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy: separate the two dataflows, shuffle only the messages
  - Basic idea: merge join between graph structure and messages

both relations sorted by join key both relations consistently partitioned and sorted by join key



Do the Schimmy!

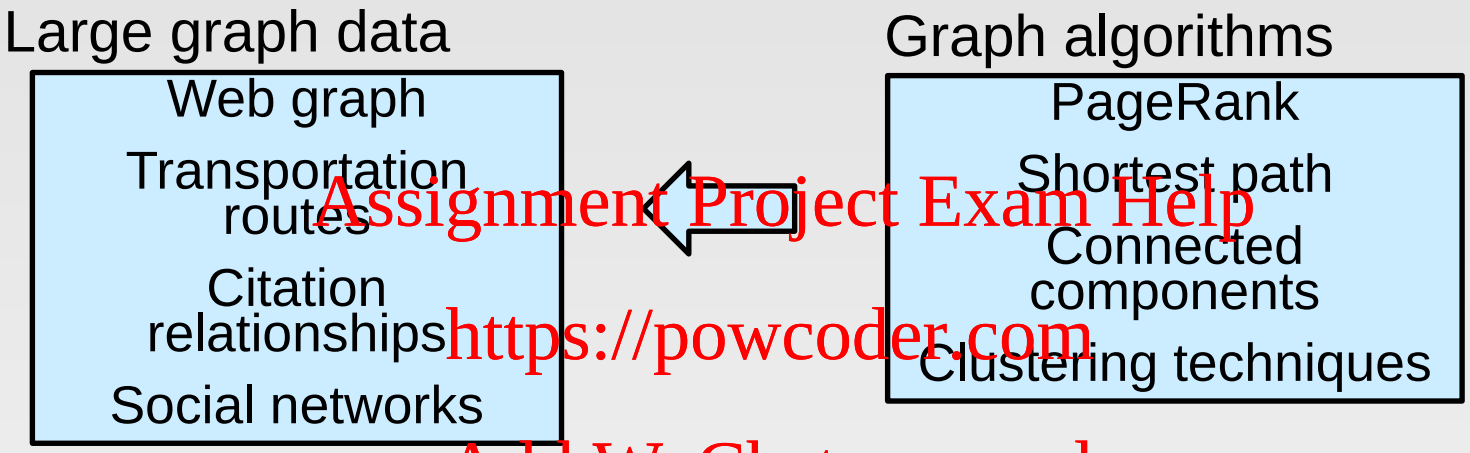
- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  - Reducers read graph structure directly from HDFS



Introduction to Pregel

Motivation of Pregel

- Many practical computing problems concern large graphs



- Single computer graph library does not scale
- MapReduce is ill-suited for graph processing
  - Many iterations are needed for parallel graph processing
  - Materializations of intermediate results at every MapReduce iteration harm performance

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)

Bulk Synchronous Parallel Model (BSP)

analogous to MapReduce rounds

- Processing: a series of supersteps
- Vertex: computation is defined to run on each vertex
- Superstep S: *all vertices compute in parallel; each vertex v may*
  - receive messages sent to v from superstep S – 1;
  - perform some computation: modify its states and the states of its outgoing edges
  - Send messages to other vertices (to be received in the next superstep)

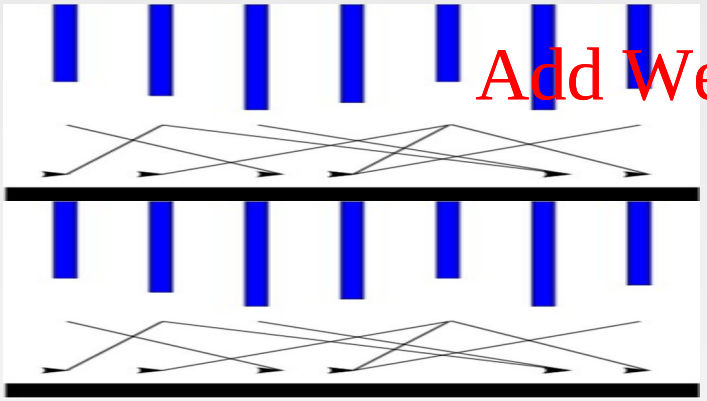
Message passing

Vertex-centric, message passing

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

Input

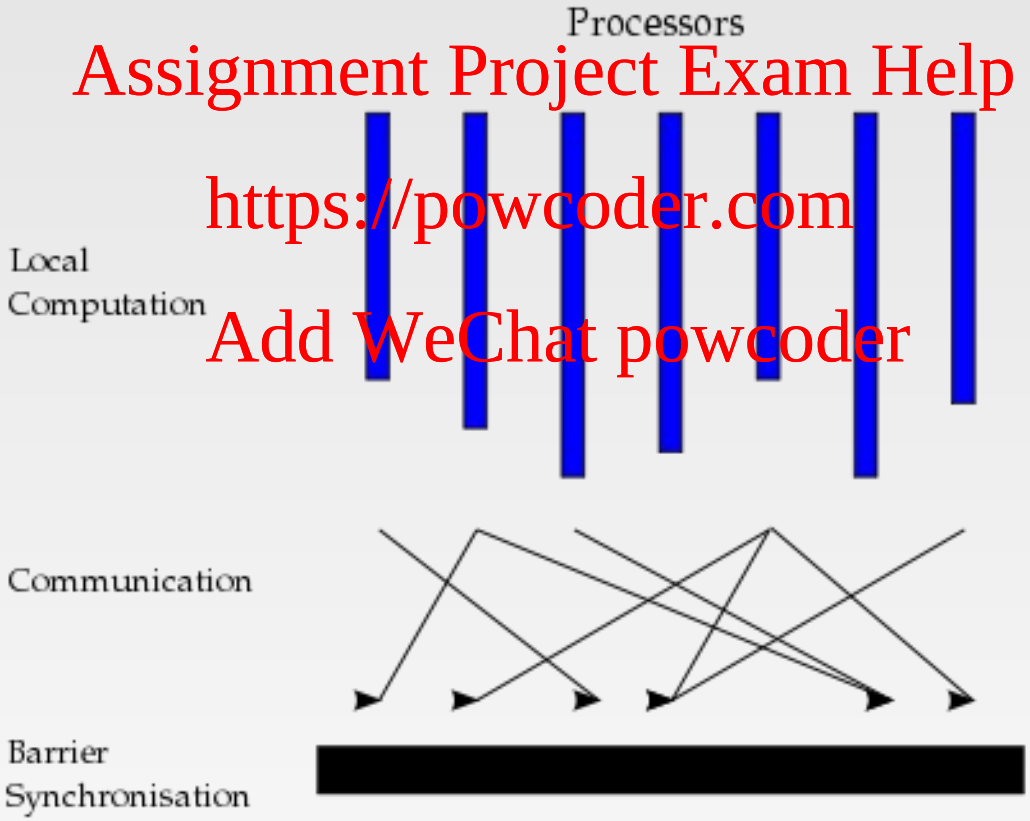


Supersteps
(a sequence of iterations)

Output

Pregel Computation Model (Cont')

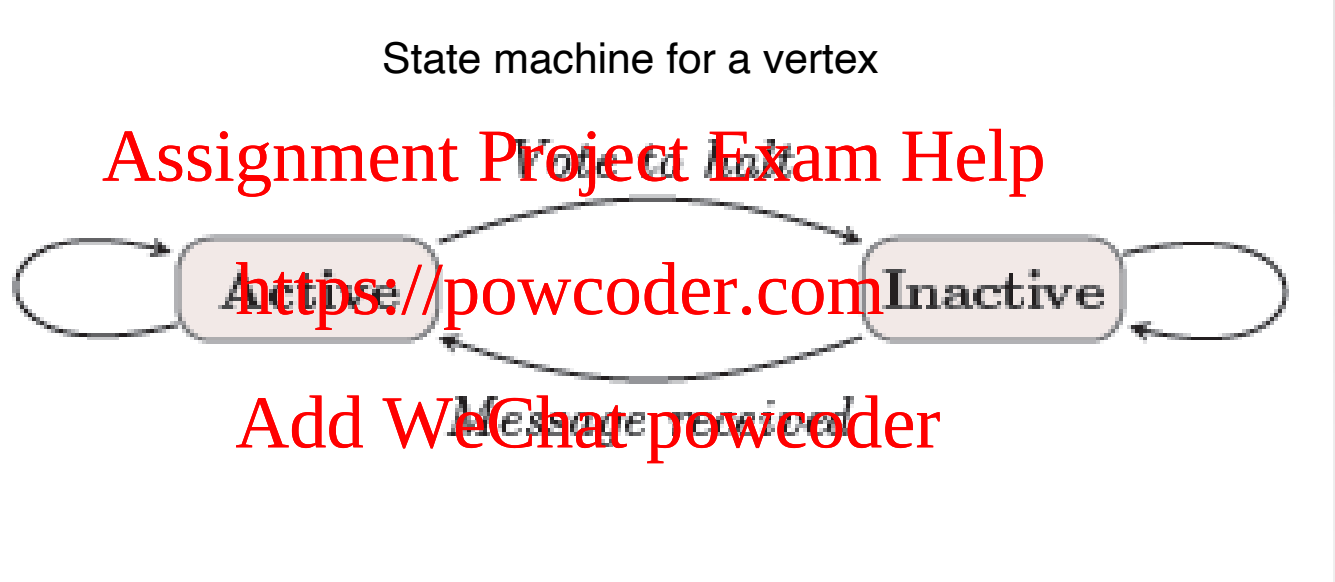
- Concurrent computation and Communication need not be ordered in time
- Communication through message passing



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

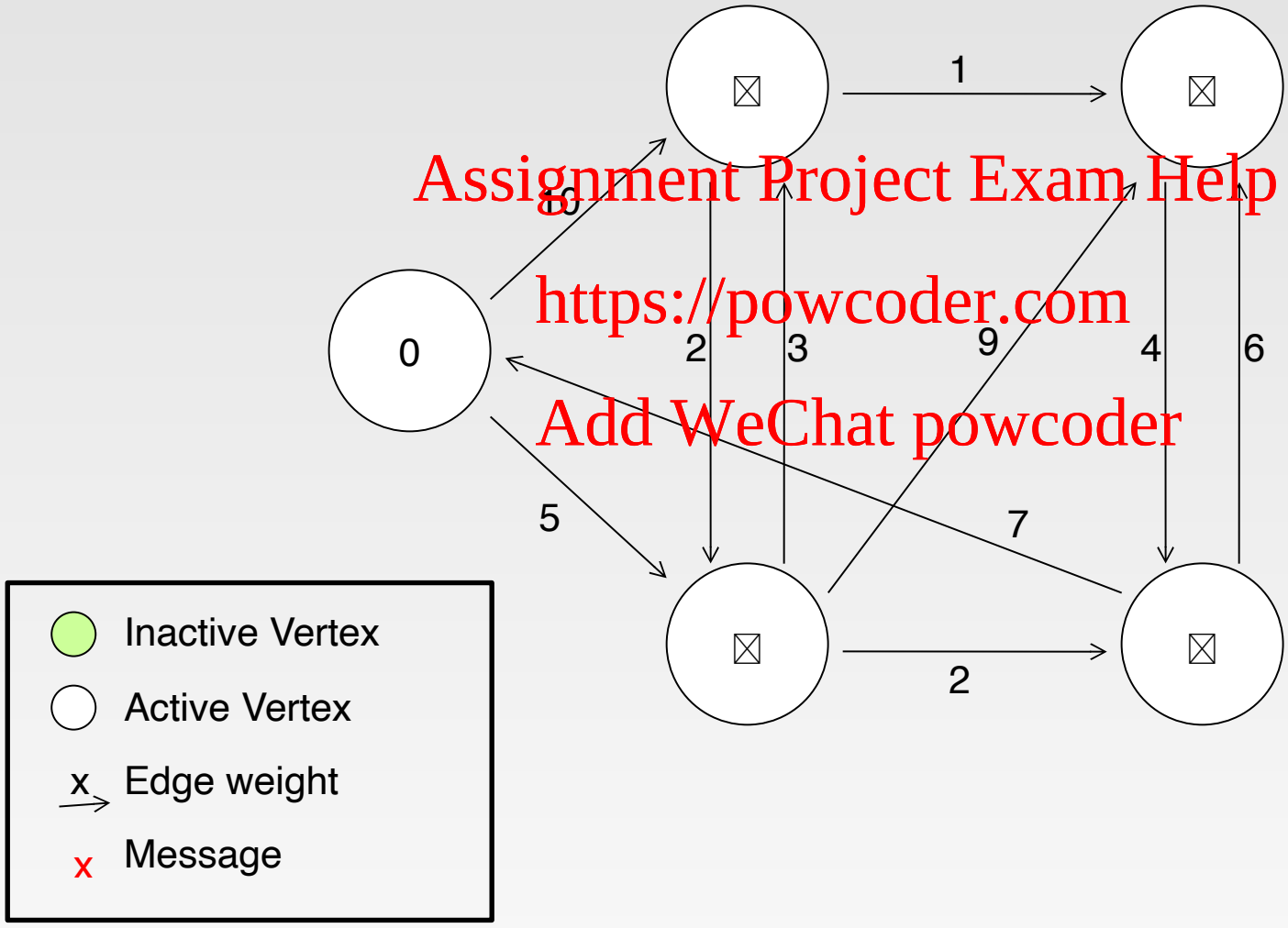
Pregel Computation Model (Cont')

- Superstep: the vertices compute in parallel
  - Each vertex

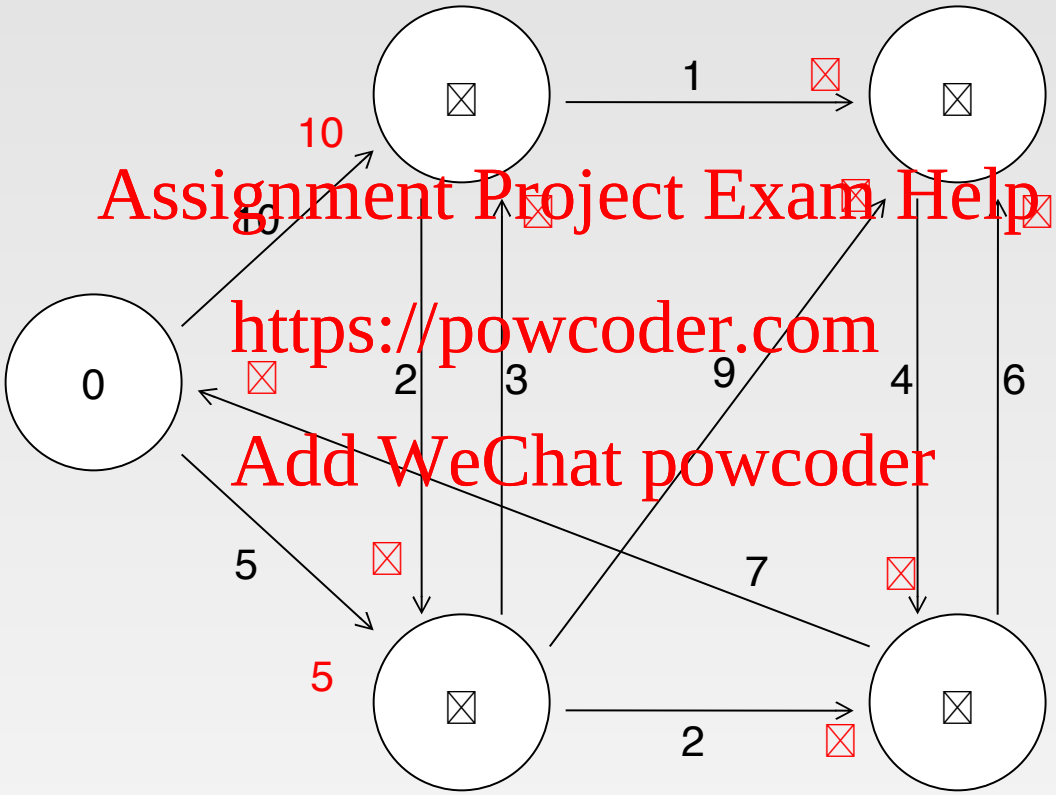


  - Termination condition
    - All vertices are simultaneously inactive
    - A vertex can choose to deactivate itself
    - Is "woken up" if new messages received

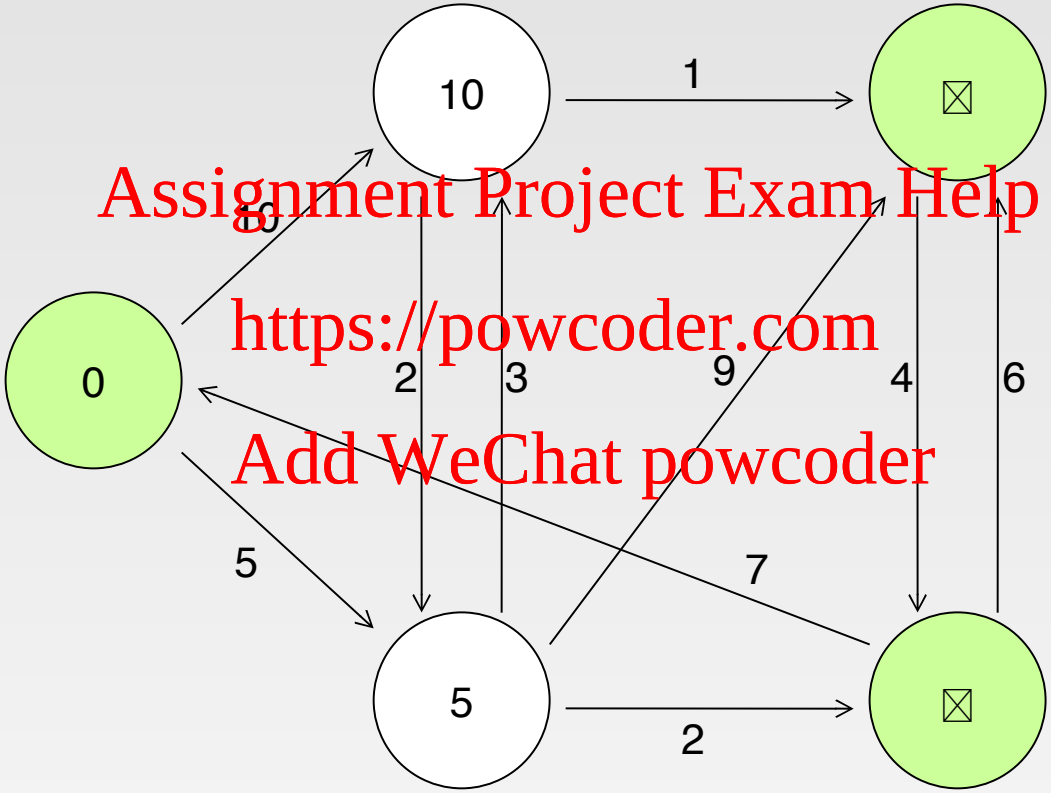
Example: SSSP – Parallel BFS in Pregel



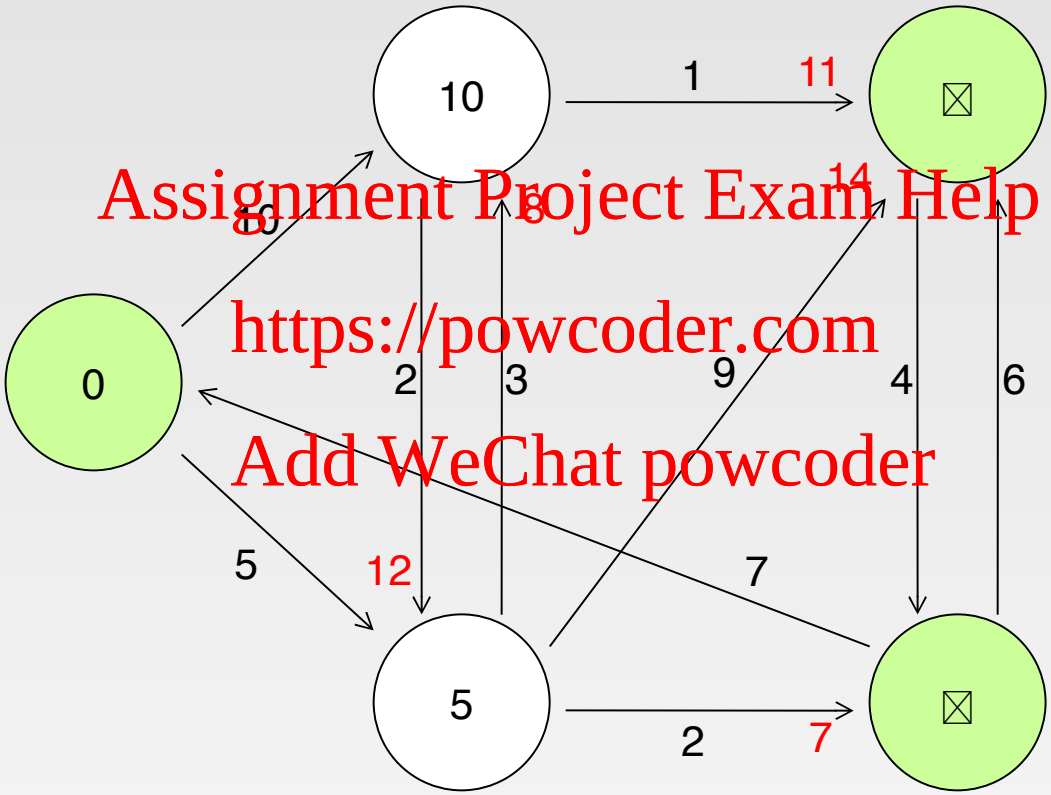
Example: SSSP – Parallel BFS in Pregel



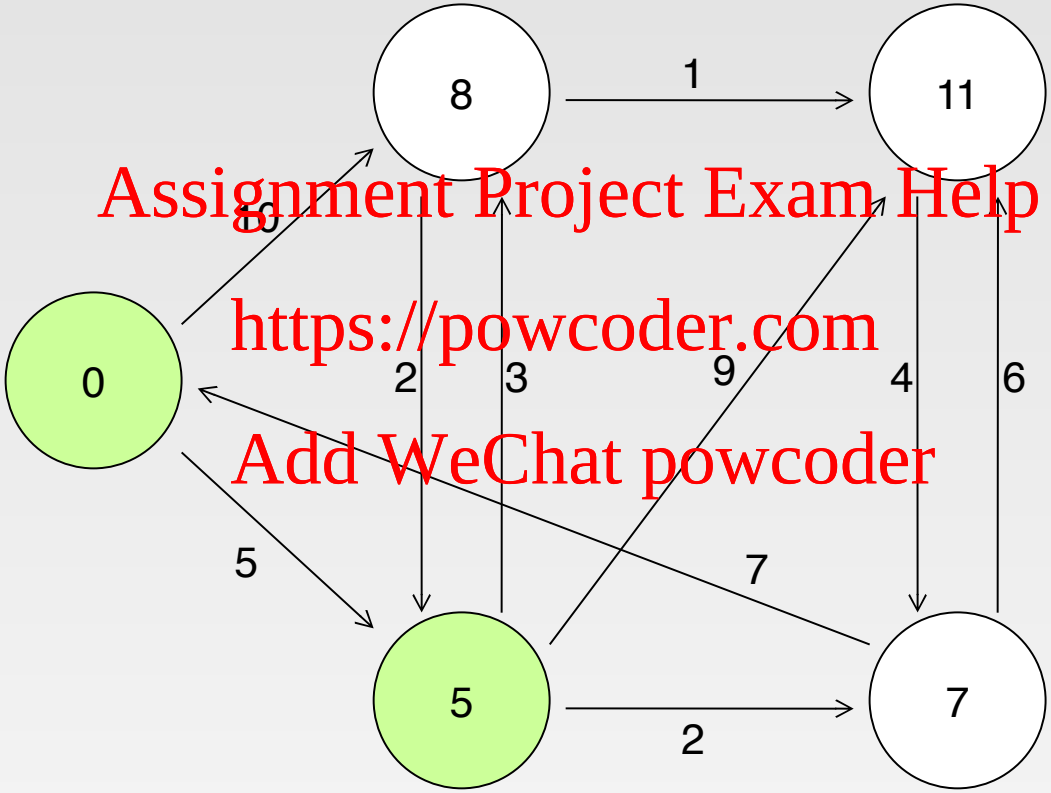
Example: SSSP – Parallel BFS in Pregel



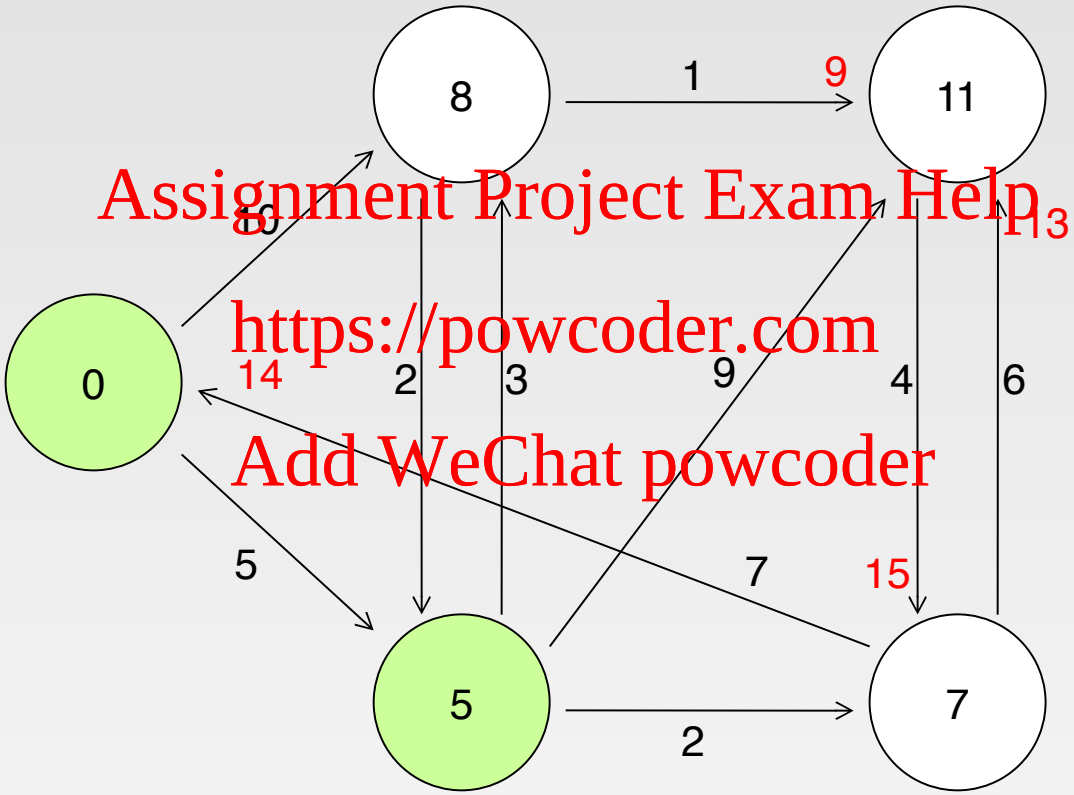
Example: SSSP – Parallel BFS in Pregel



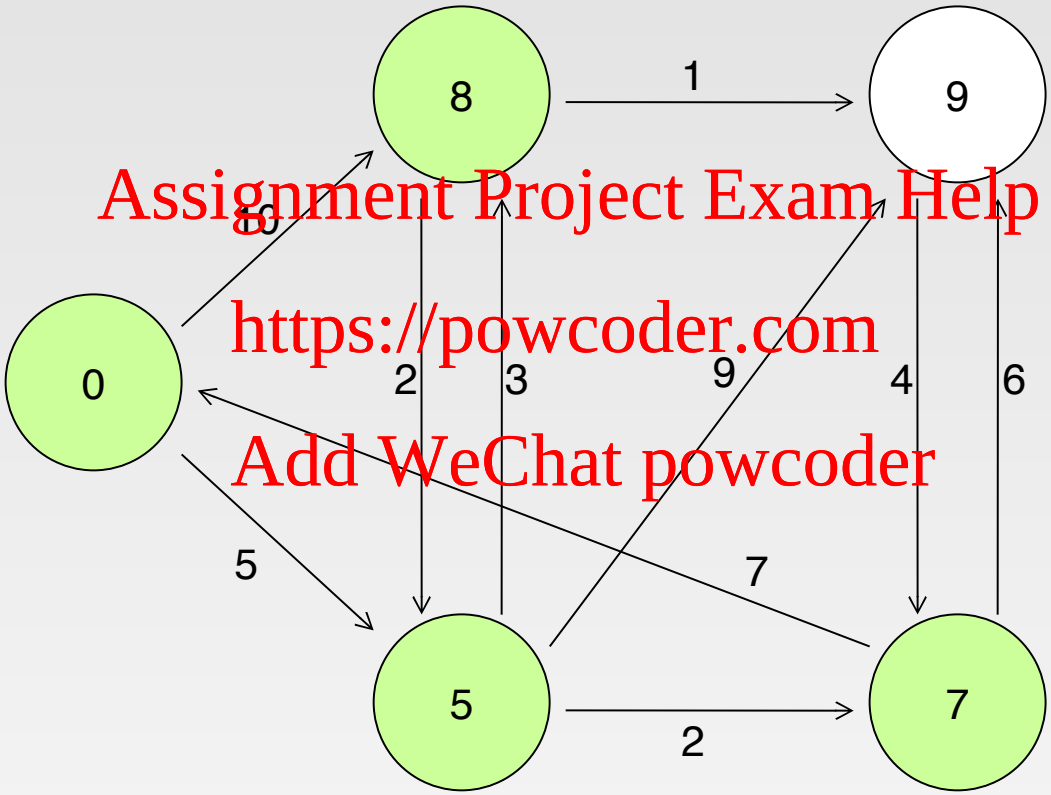
Example: SSSP – Parallel BFS in Pregel



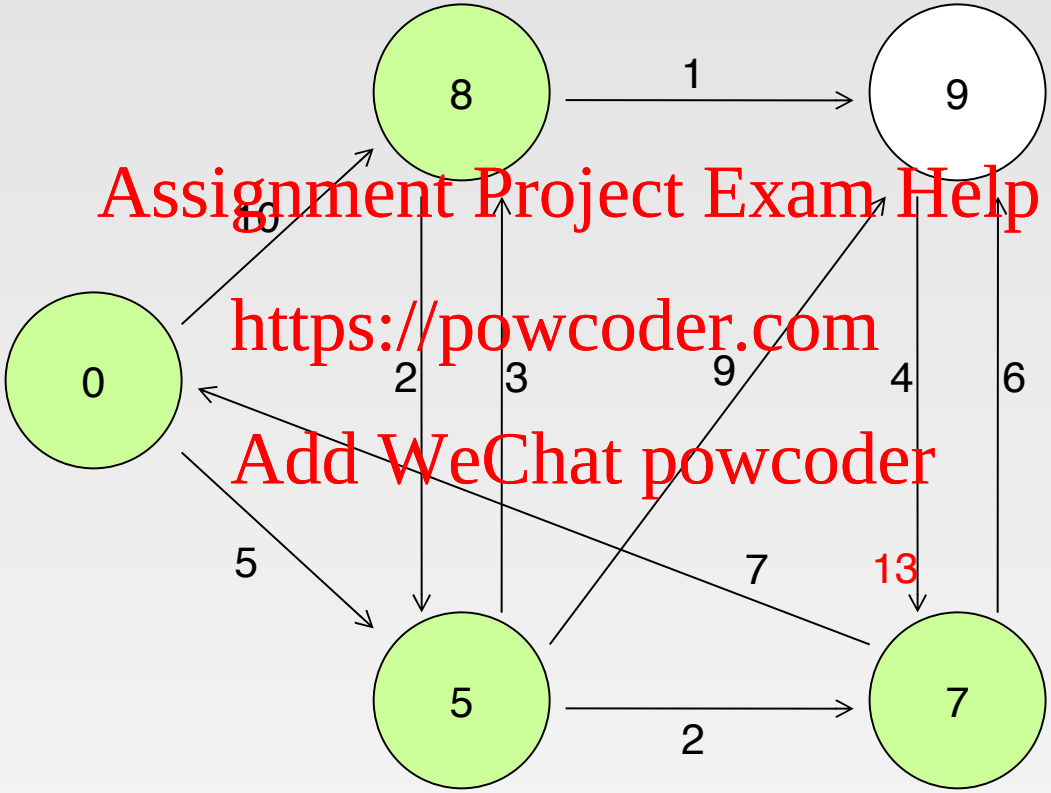
Example: SSSP – Parallel BFS in Pregel



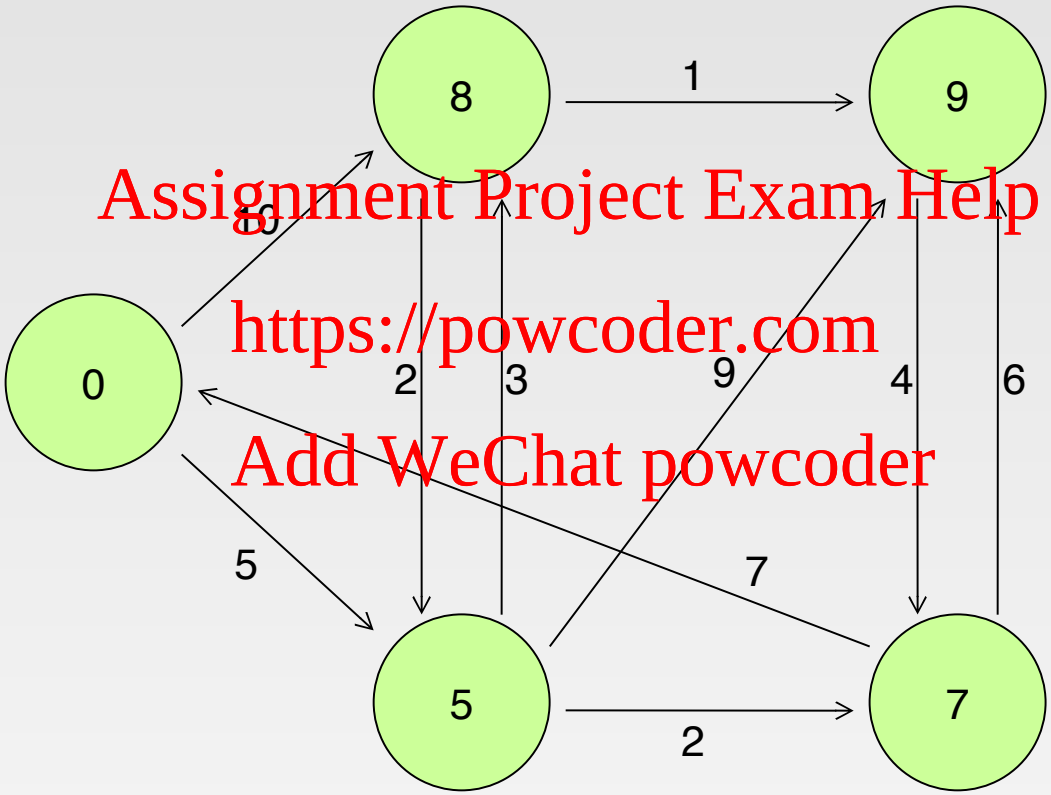
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Differences from MapReduce

- Graph algorithms can be written as a series of chained MapReduce jobs

- Pregel
  - Keeps vertices & edges on the machine that performs computation
  - Uses network transfers only for messages

- MapReduce
  - Passes the entire state of the graph from one stage to the next
  - Needs to coordinate the steps of a chained MapReduce

Writing a Pregel Program (C++)

- Subclassing the predefined **Vertex** class

```
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

Pregel: SSSP (C++)

```
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();
  }
};
```

Refer to the current node as u

aggregation

Messages: distances to u

MutableValue: the current distance

Pass revised distance to its neighbors

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.

More Tools on Big Graph Processing

- Graph databases: Storage and Basic Operators
  - http://en.wikipedia.org/wiki/Graph_database
  - Neo4j (an open source graph database)
  - InfiniteGraph
  - VertexDB
  - … …
- Distributed Graph Processing (mostly in-memory-only)
  - Google's Pregel (vertex centered computation)
  - Giraph (Apache)
  - GraphX (Spark)
  - GraphLab
  - … …

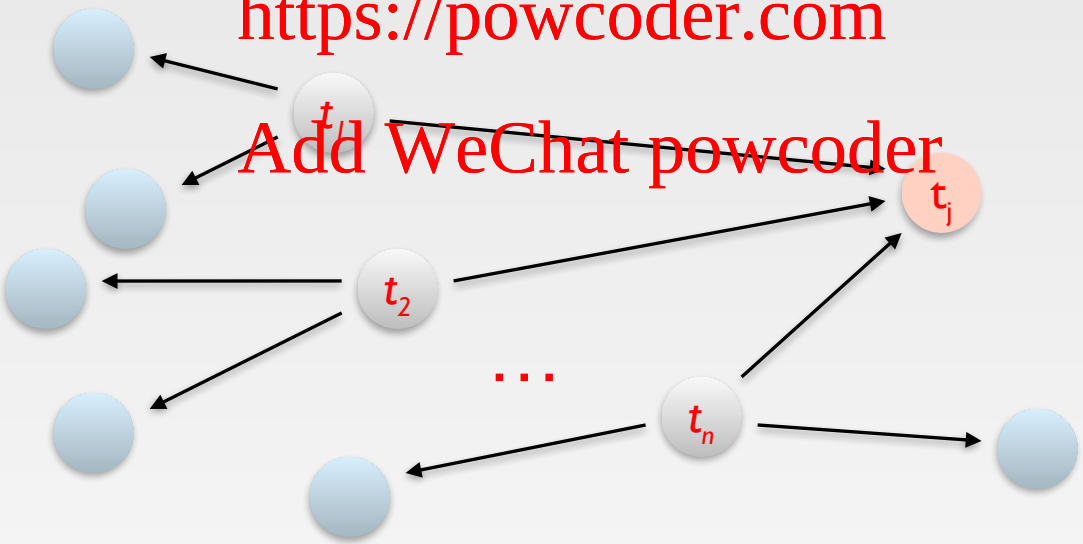
References

- Chapter 5. Data-Intensive Text Processing with MapReduce
- Chapter 5. Mining of Massive Datasets.

End of Chapter 5

PageRank Review

- Given page t_j with in-coming neighbors $t_1 \ldots t_n$, where
  - d_i is the out-degree of t_i
  - β is the teleport probability
  - N is the total number of nodes in the graph



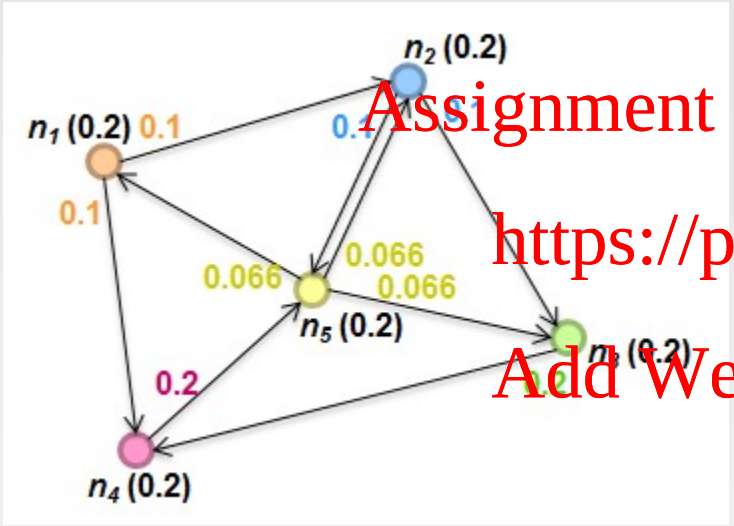
Computing PageRank

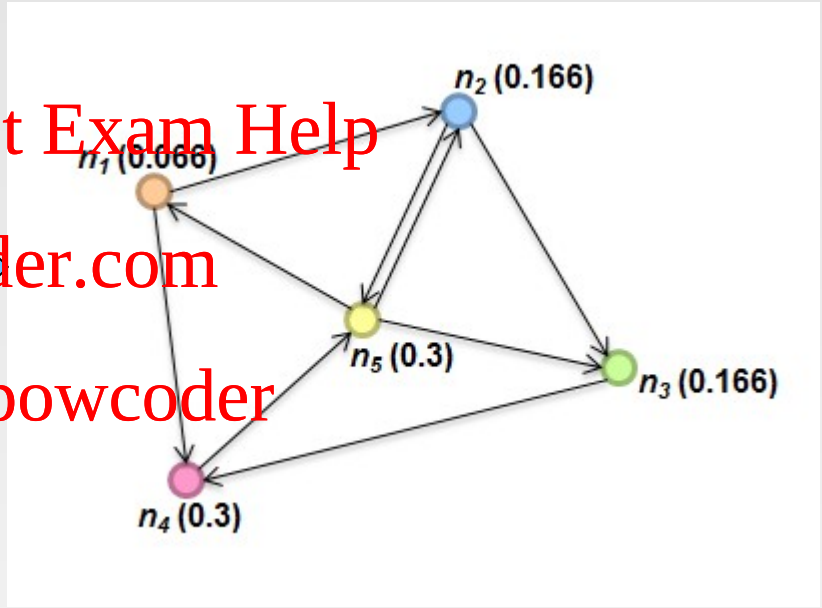
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed r_i values
  - Each page distributes r_i "credit" to all pages it links to
  - Each target page t_j adds up "credit" from multiple in-bound links to compute r_j
  - Iterate until values converge

Simplified PageRank

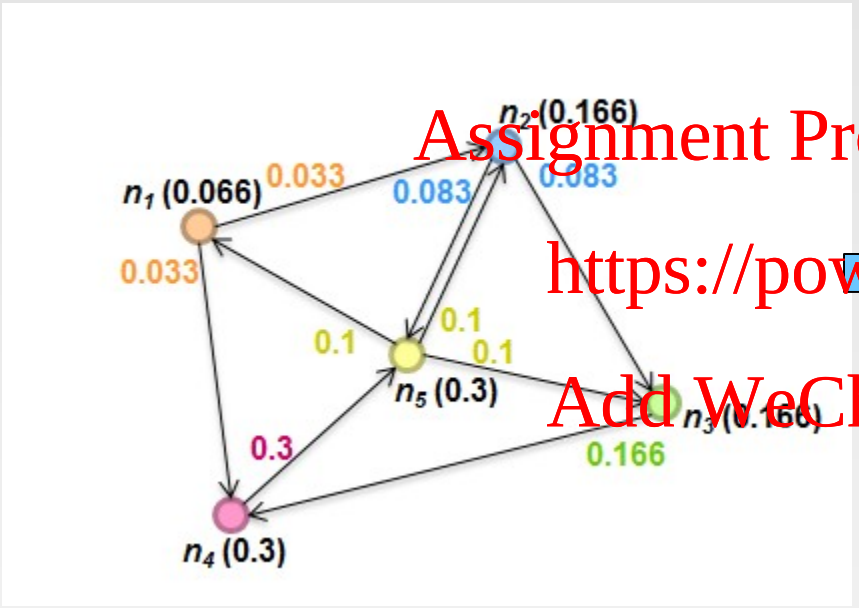
- First, tackle the simple case:
  - No teleport
  - No dangling nodes (dead ends)
- Then, factor in these complexities…
  - How to deal with the teleport probability?
  - How to deal with dangling nodes?

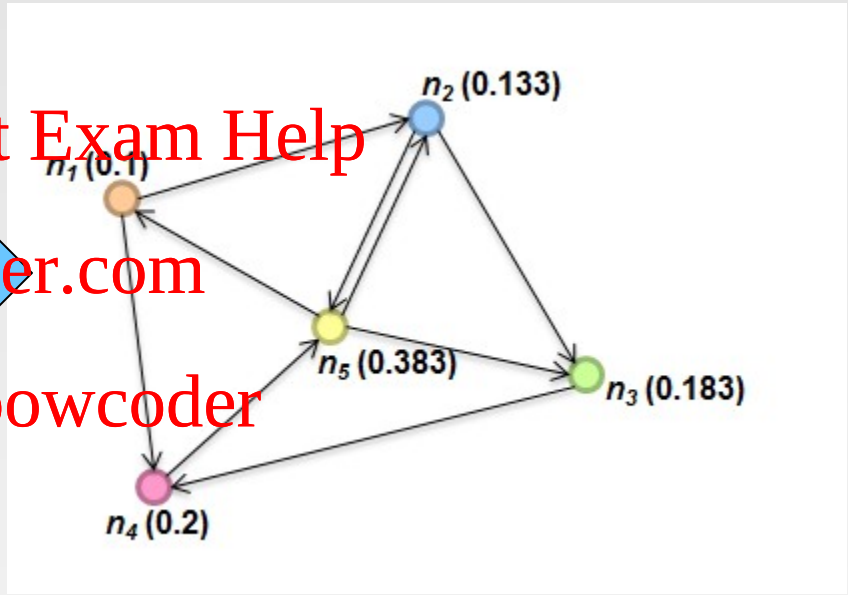
Sample PageRank Iteration (1)



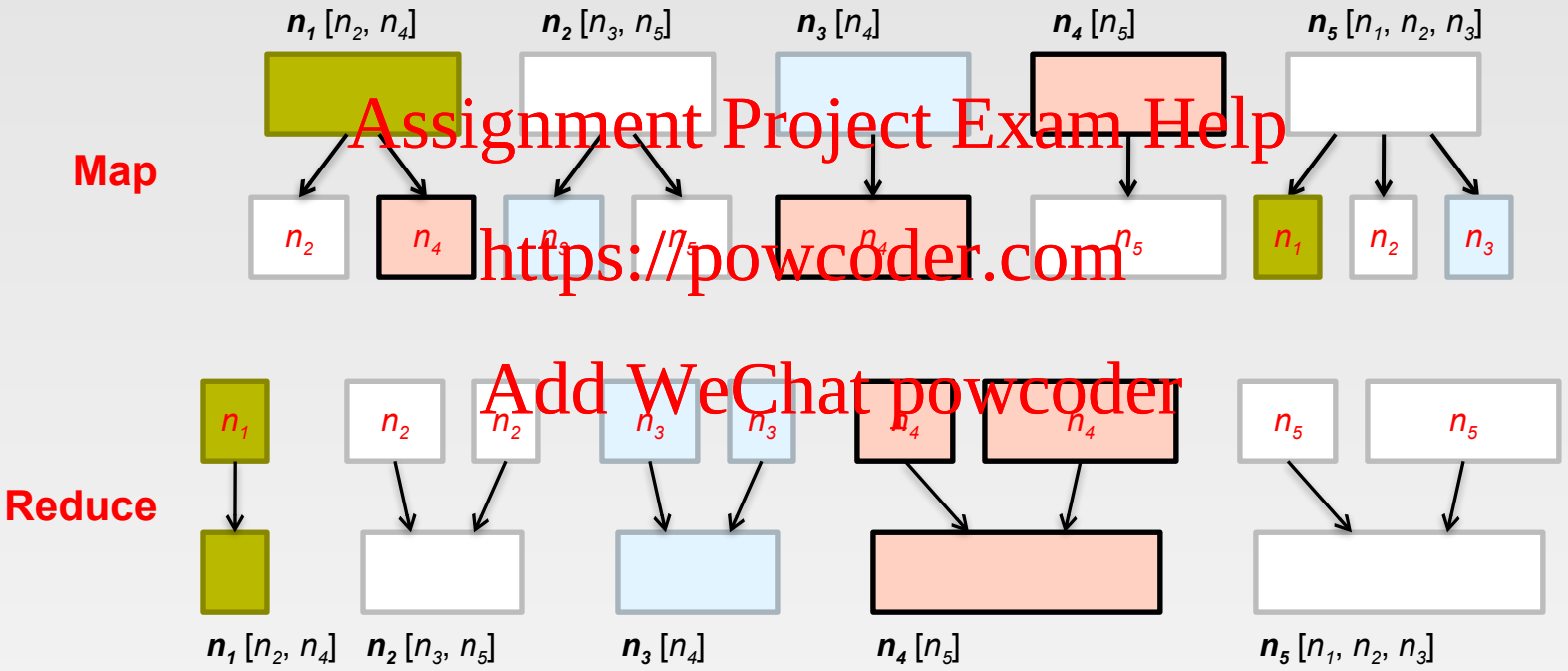


Sample PageRank Iteration (2)





PageRank in MapReduce (One Iteration)



PageRank Pseudo-Code

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sums incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

PageRank vs. BFS

	PageRank	BFS
Map	r_i/d_i	d+w
Reduce	sum	min

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 + 0.85 * sum;
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
}
```