COMP9313: Big Data Management

Lecturer: Xin Cao

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

Chapter ocessing

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.
  - Presence or
- 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

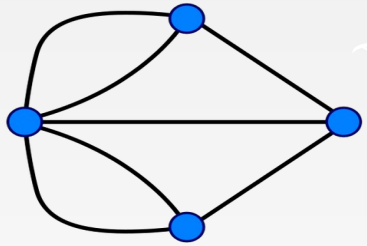
4-degrees of separation [Backstrom-Boldi-Rosa-Ugander-Vigna, 2011]

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 sche
- Identify “special”
  - Breaking up terrorist cells, spre
- 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 webpage in the web graph or each user in the twitter graph using easure
- Transportation
  - Return the shortest or cheapest 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 duce?
  - How do you ?

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		
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 zero
  - Lots of wast

Representing Graphs

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

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

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

Adjacency Lists: Critique

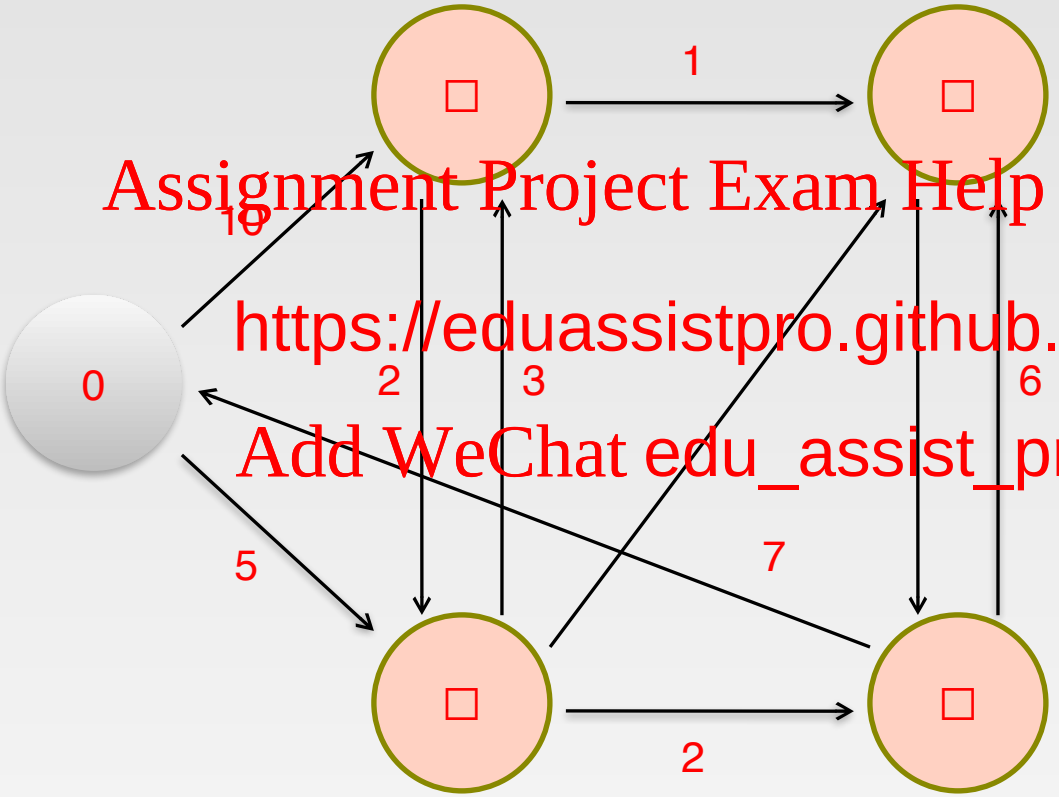
- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

Single-Source Shortest Path (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

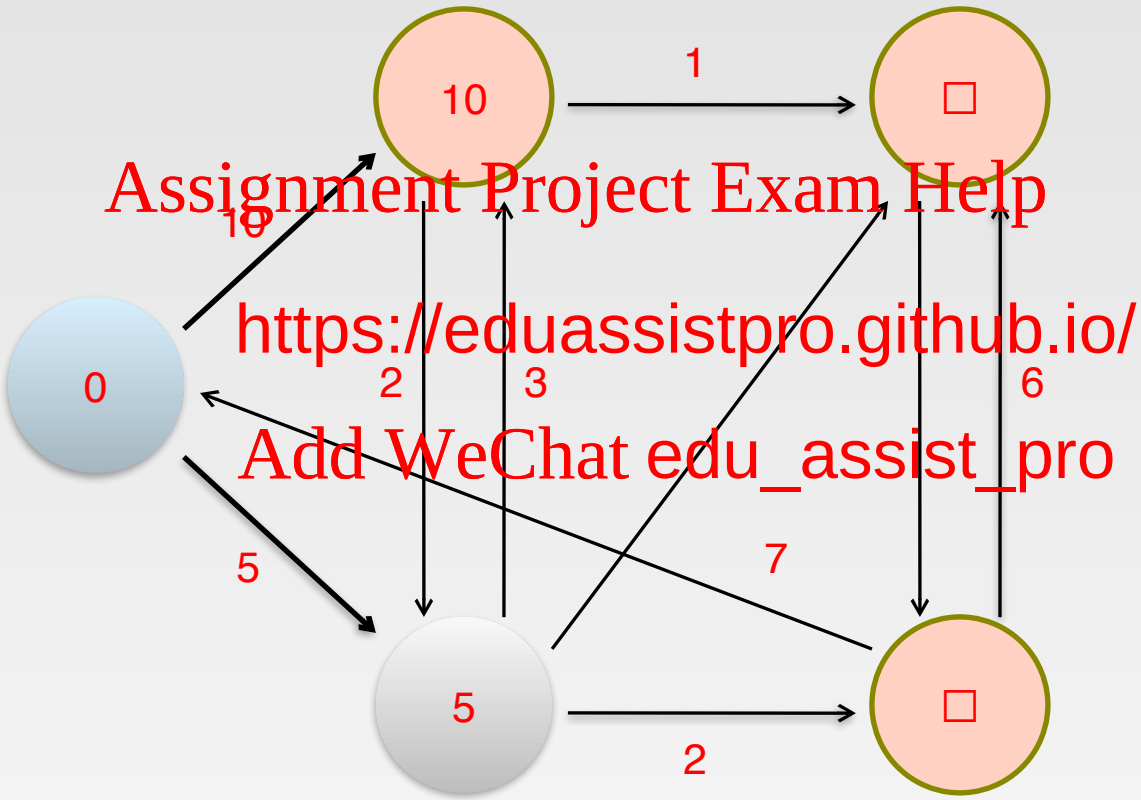
Dijkstra's Algorithm

Dijkstra's Algorithm Example

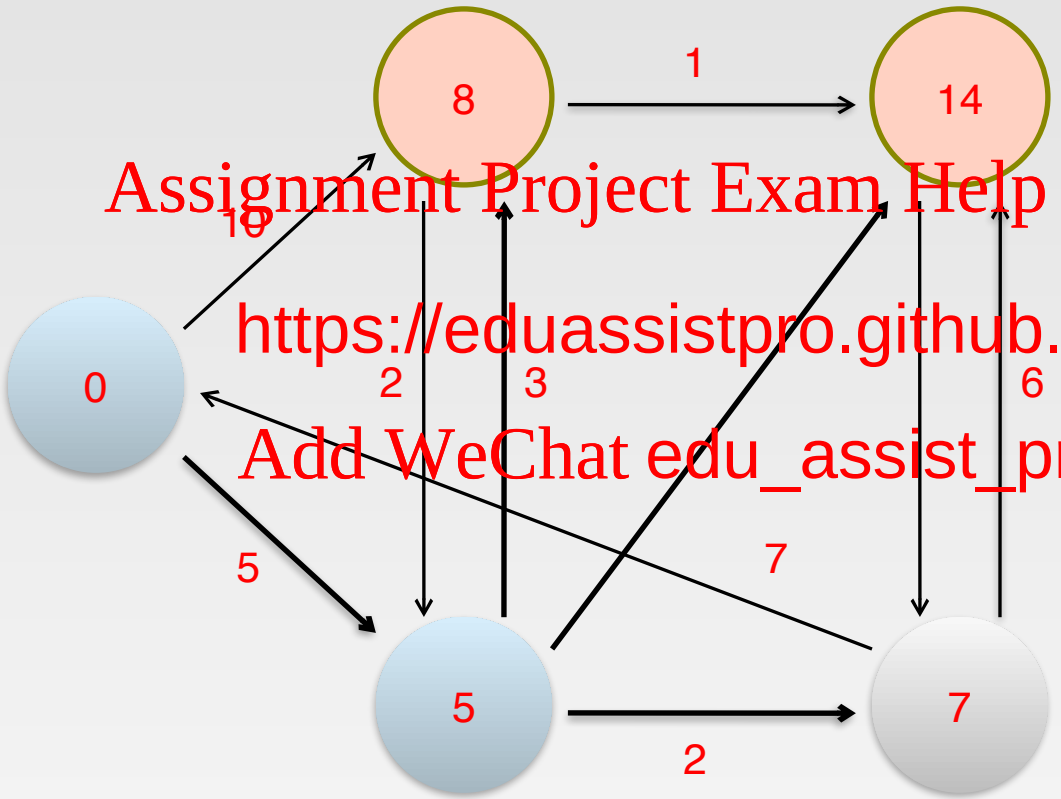


Example from CLR

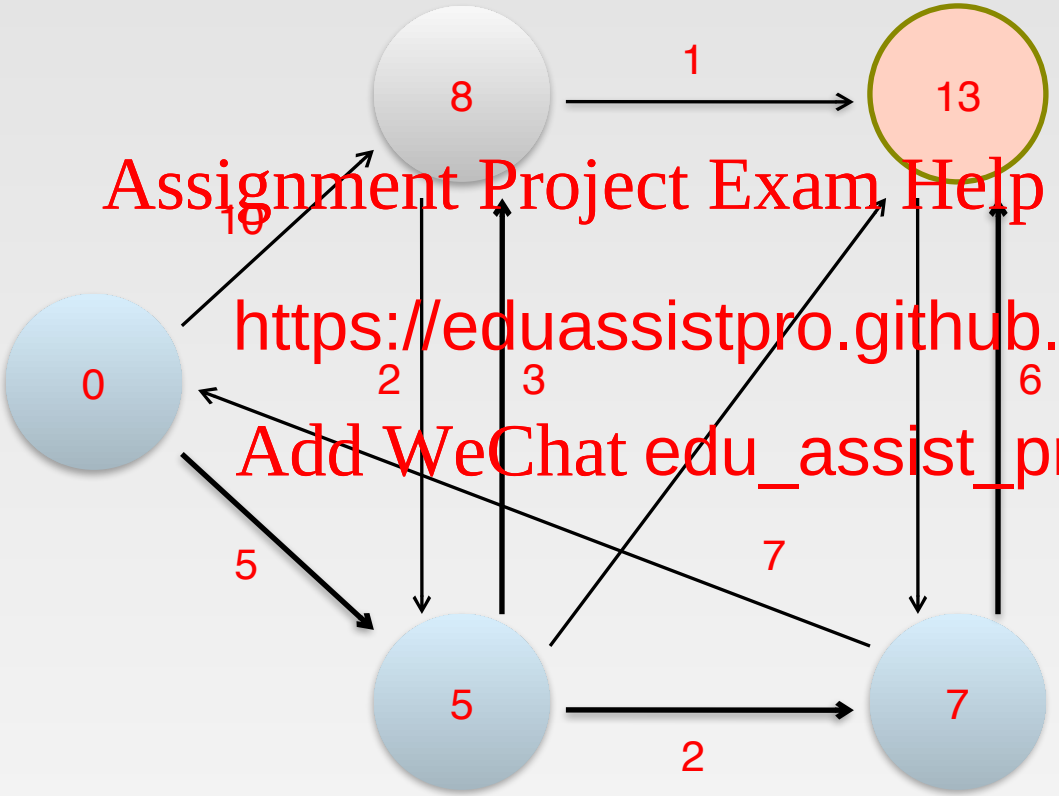
Dijkstra's Algorithm Example



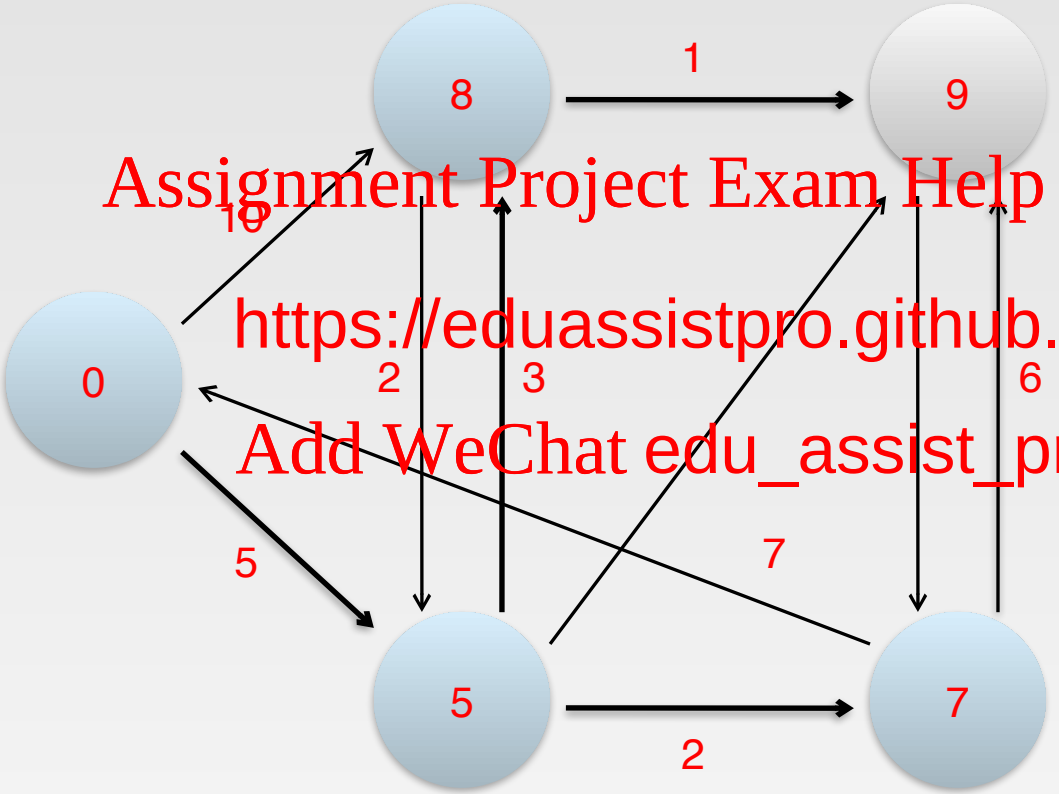
Dijkstra's Algorithm Example



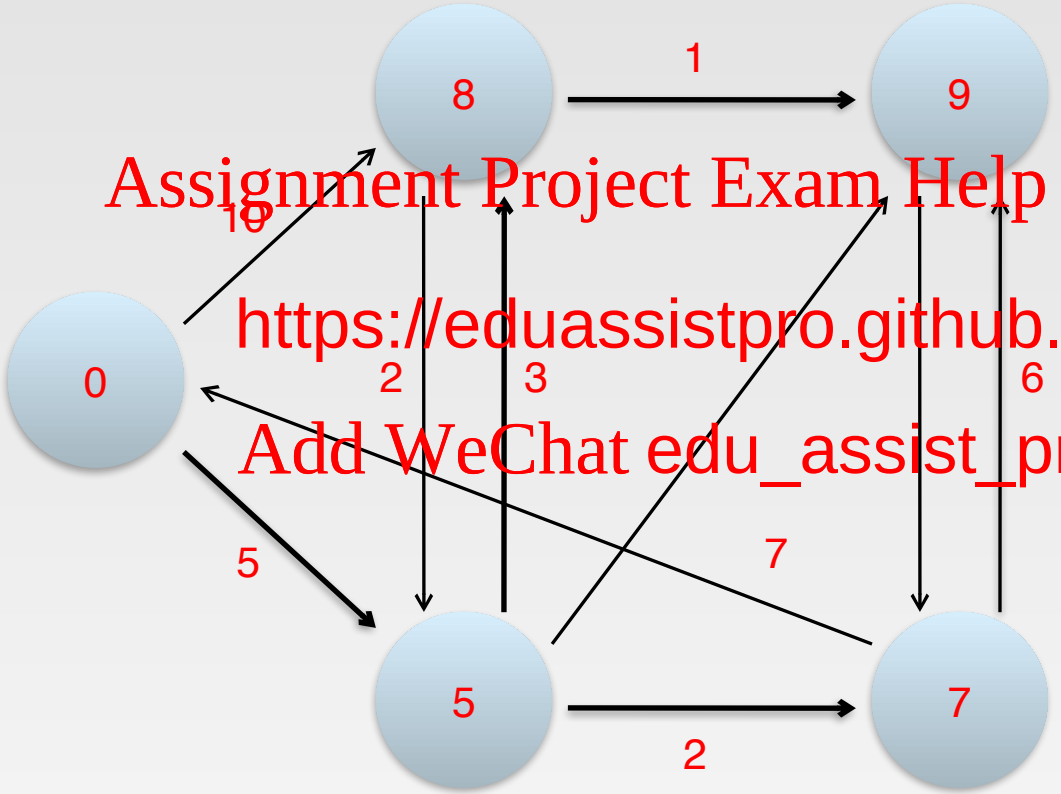
Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



Dijkstra's Algorithm Example



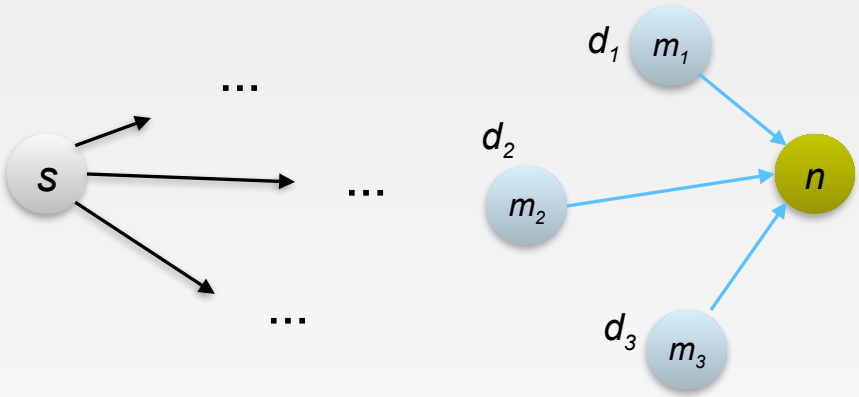
Finish!

Single Source Shortest Path

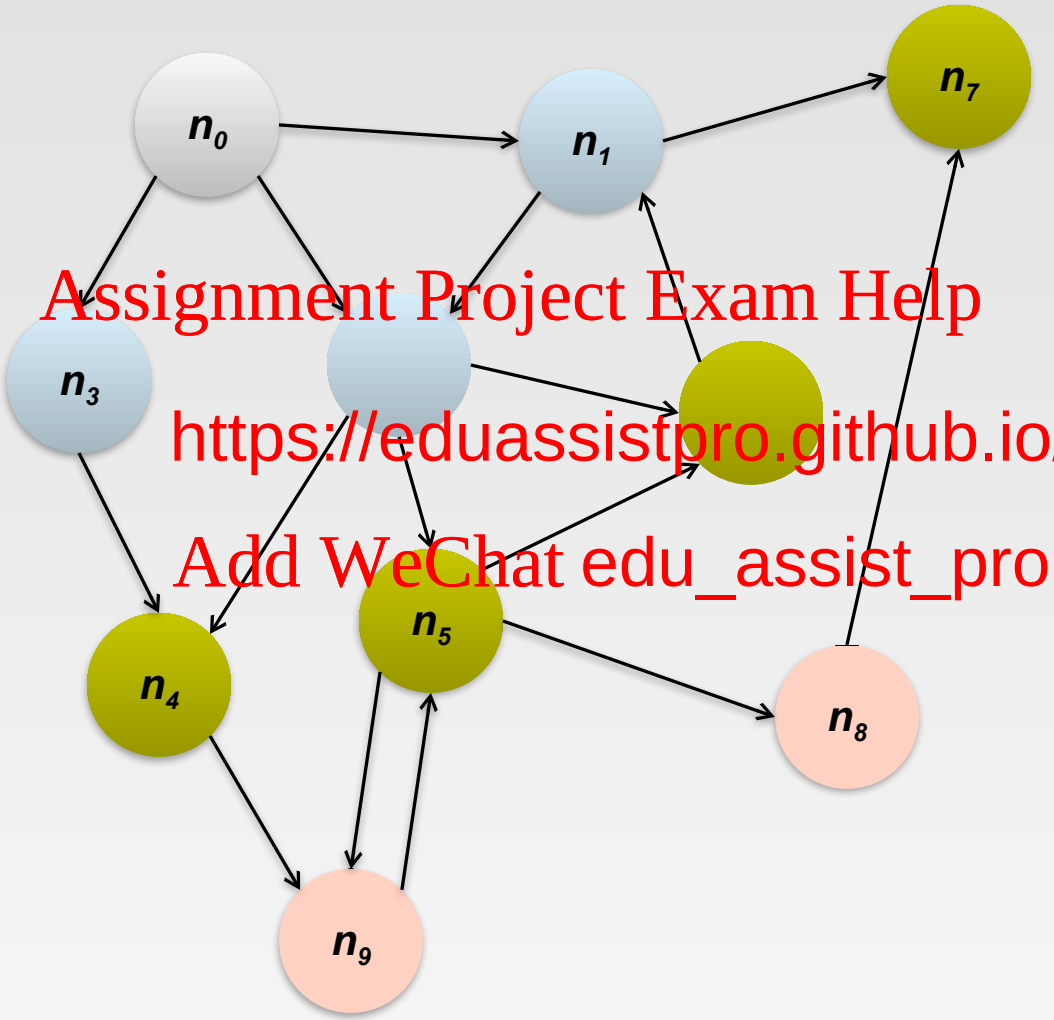
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel Breadth-First Search (BFS)

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*
  - DISTANCETO(*s*) = 0
  - For all node
    DISTANCE
  - For all nodes *n* reachable from of nodes *M*,
    DISTANCETO(*n*) = 1 + min(DISTANCETO(*m*)), *m* ∈ *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:
  - $\forall m \in$ adjac
- Sort/Shuffle
  - Groups distances by reachable
- 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 adjace
  - 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.
        Emit(nid m, d+                     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 node
  - Six degrees
    - If this is indeed true, then parallel breadth-first search on the global social network would take only 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., numbe times a certain condition is nmer desires.
  - Counters can be designed to co r of nodes that have distances of ∞ at the end of the 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 appro istances in the mapper, so that each node will hav ath easily accessible at all times
  - The additional space requireme e

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 "+dista                                    ue
            context.write(new LongW                                    rd)
            word.clear(); }
        //pass in current node's distance (if it is the lowest dis
        word.set("VALUE "+sp[1]);
        context.write( new LongWritable( Integer.parseInt( sp
        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

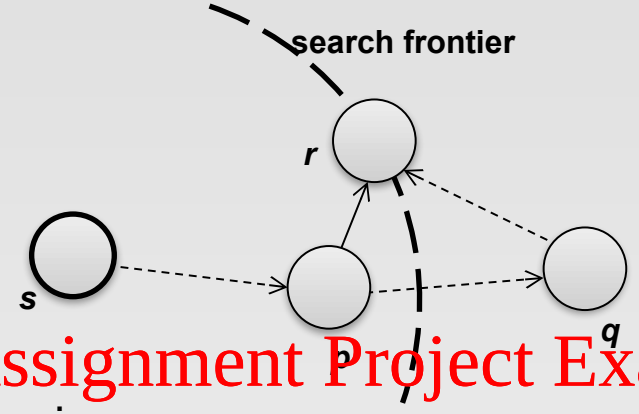
```
public static class TheReducer extends Reducer<LongWritable, Text, LongWritable, Text> {
    public void reduce(LongWritable key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        String nodes = "UNMODED";
        Text word = new Text();
        int lowest = 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.toS
            //look at first value
            if(sp[0].equalsIgnor
                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**
  - 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

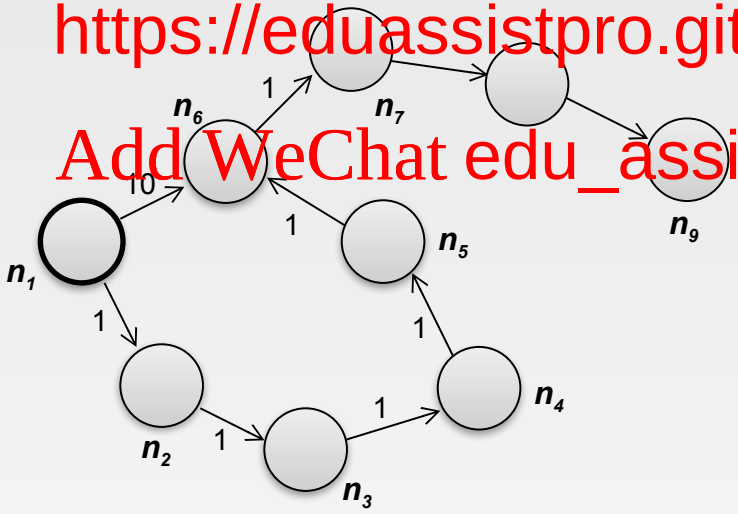
Additional Complexities



- Assume that *p* i
  - In the curre node r for the very first time.
  - We've already discovered the s e to node *p*, and that the shortest distance to *r* s ugh *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),

The adjacency list of each node will als 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 l n2: 2, n3:1

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

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

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

Ignore the processi

- Reduce:

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

Emit:

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

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

n3 --> 11 l n4:4

n4 --> 7 l 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 | n

Emit: (n4, 15) , (n

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

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

- 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:

Emit: (n4, 13) , (n

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

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

- 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
- Why can't we do

Graphs and MapReduce

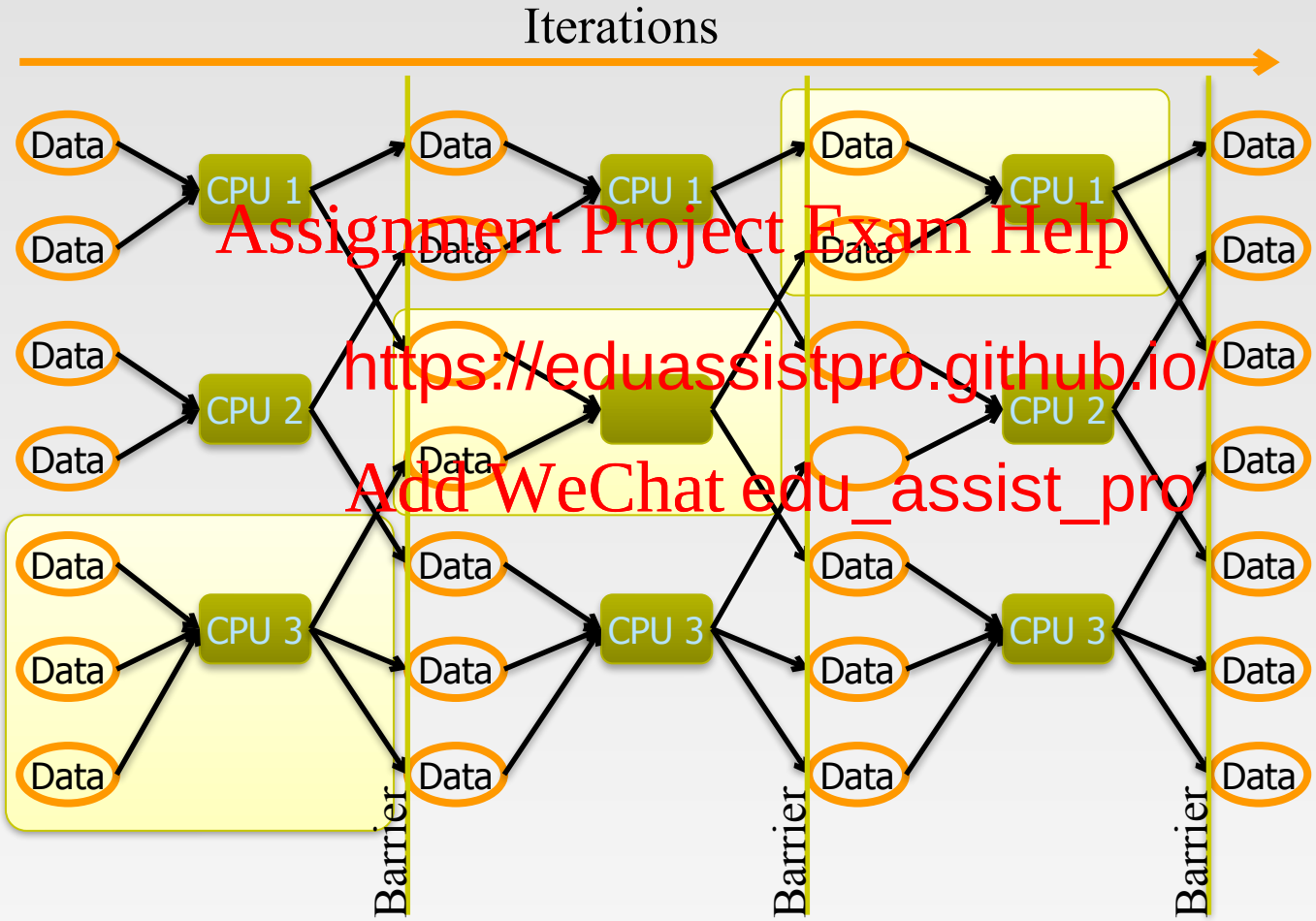
- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Generic recipe:
  - Represent g
  - Perform loc
  - Pass along partial results via ou y destination node
  - Perform aggregation in reducer 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 : a global priority queue that guid
    - Dijkstra's algorithm in Hado eue available. Do some "wasted" computation
- 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.

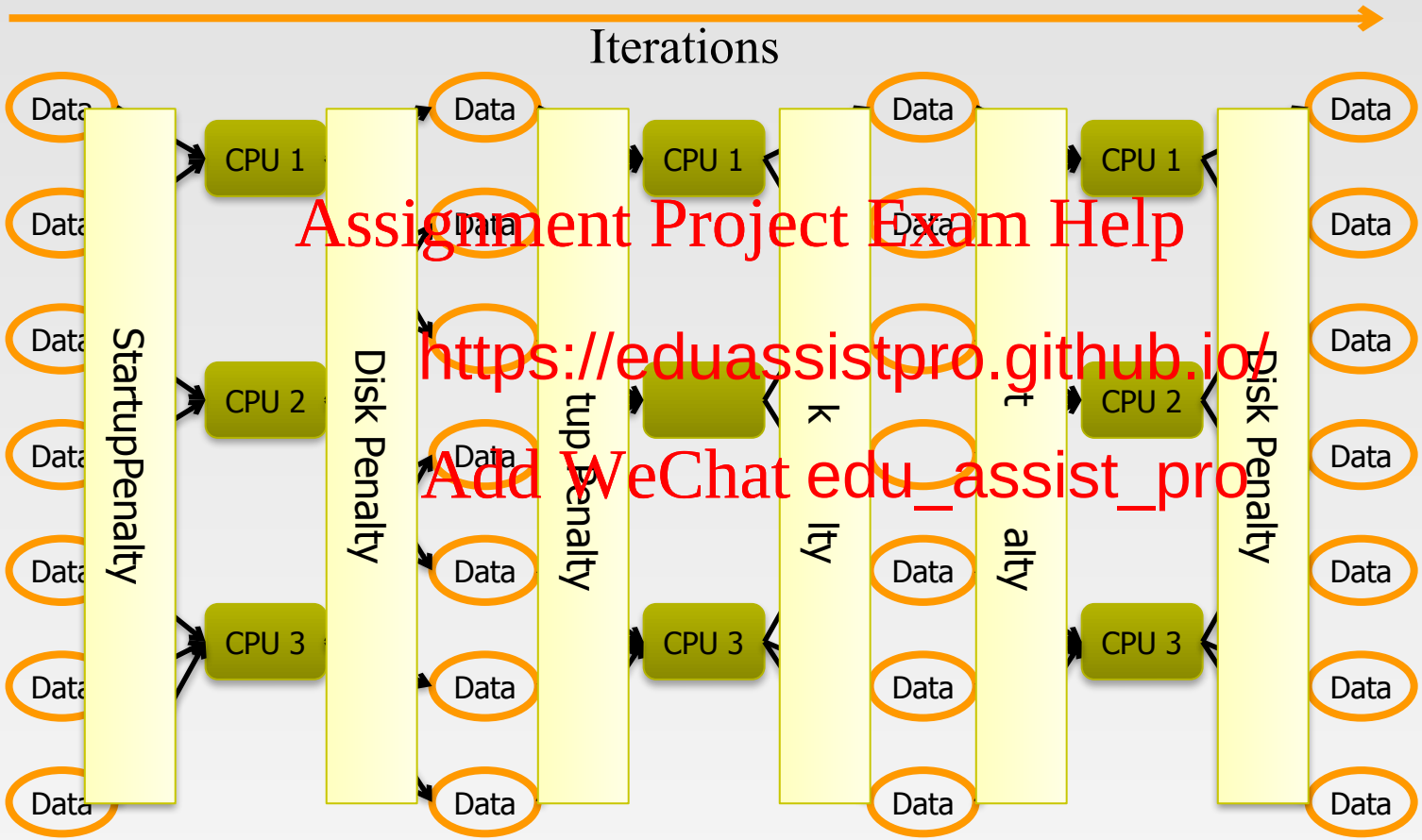
Iterative MapReduce

- Only a subset of data needs computation:



Iterative MapReduce

- System is not optimized for iteration:



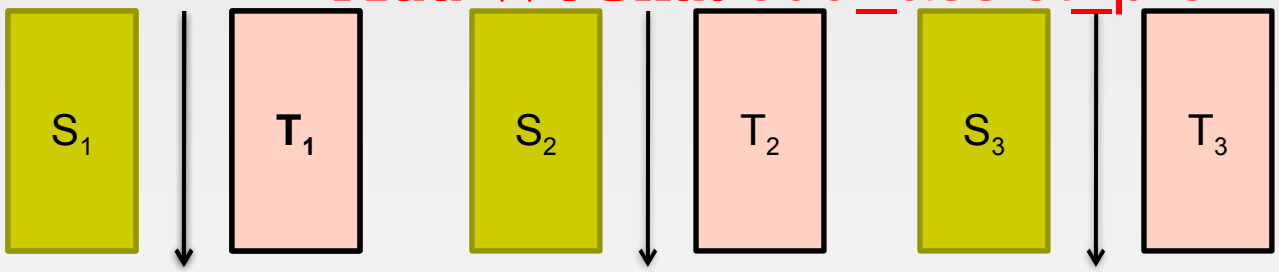
Better Partitioning

- 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, p
  - Sometimes,
  - But cheap heuristics sometimes
  - For webgraphs: range partition ted URLs

Schimmy Design Pattern

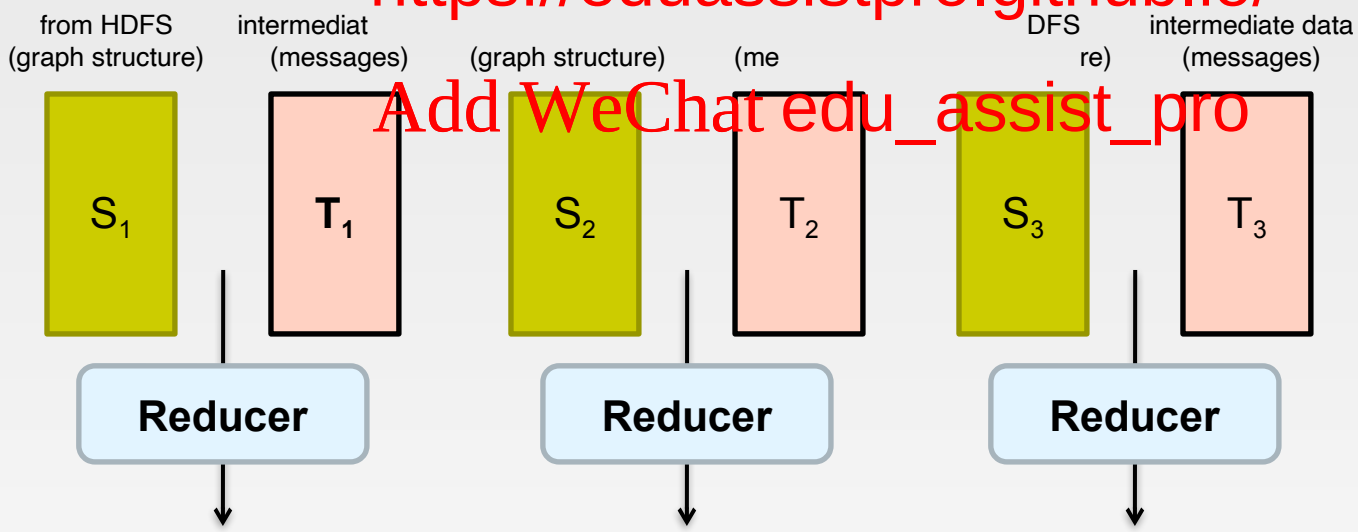
- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy: separate the two dataflows, shuffle only the messages
  - Basic idea: merge join between graph structure and messages

both relations sorted by join key both relations consistently partitioned



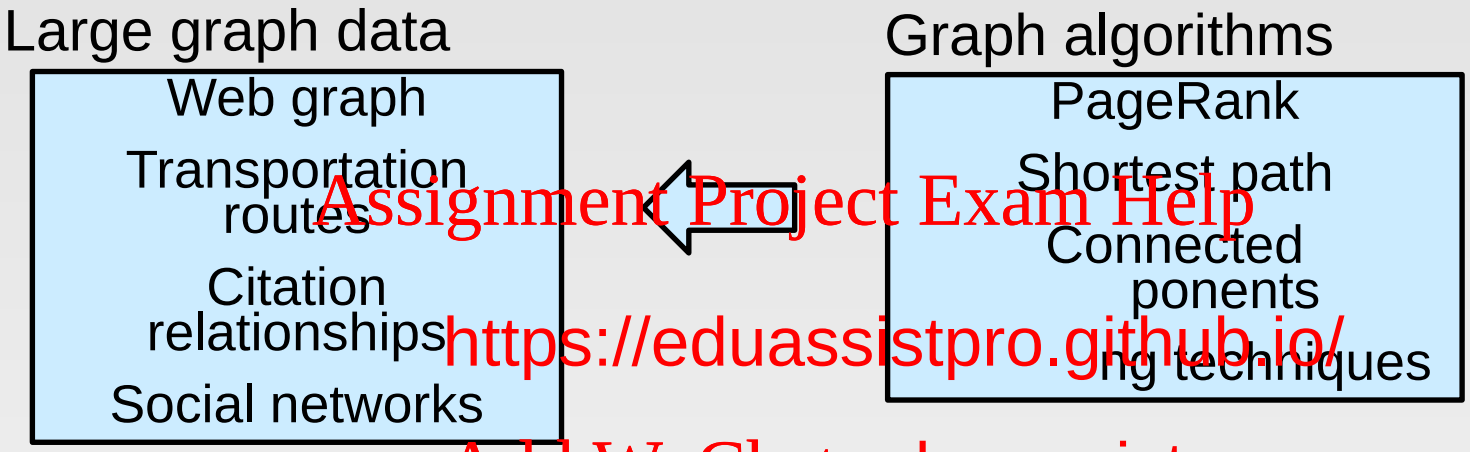
Do the Schimmy!

- 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



Motivation of Pregel

- Many practical computing problems concern large graphs



- Single computer graph library does
- 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 flexibilit

- Inspired by Valiant's Bulk Synchron odel
  - Leslie G. Valiant: A Bridging M 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 ed
  - Send messa eived in the next superstep)

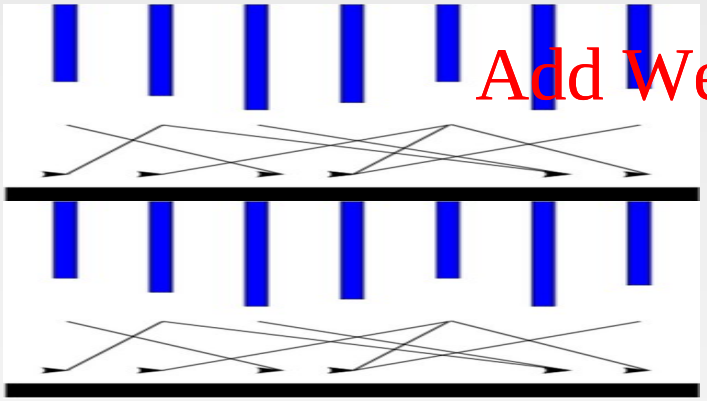
Message

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

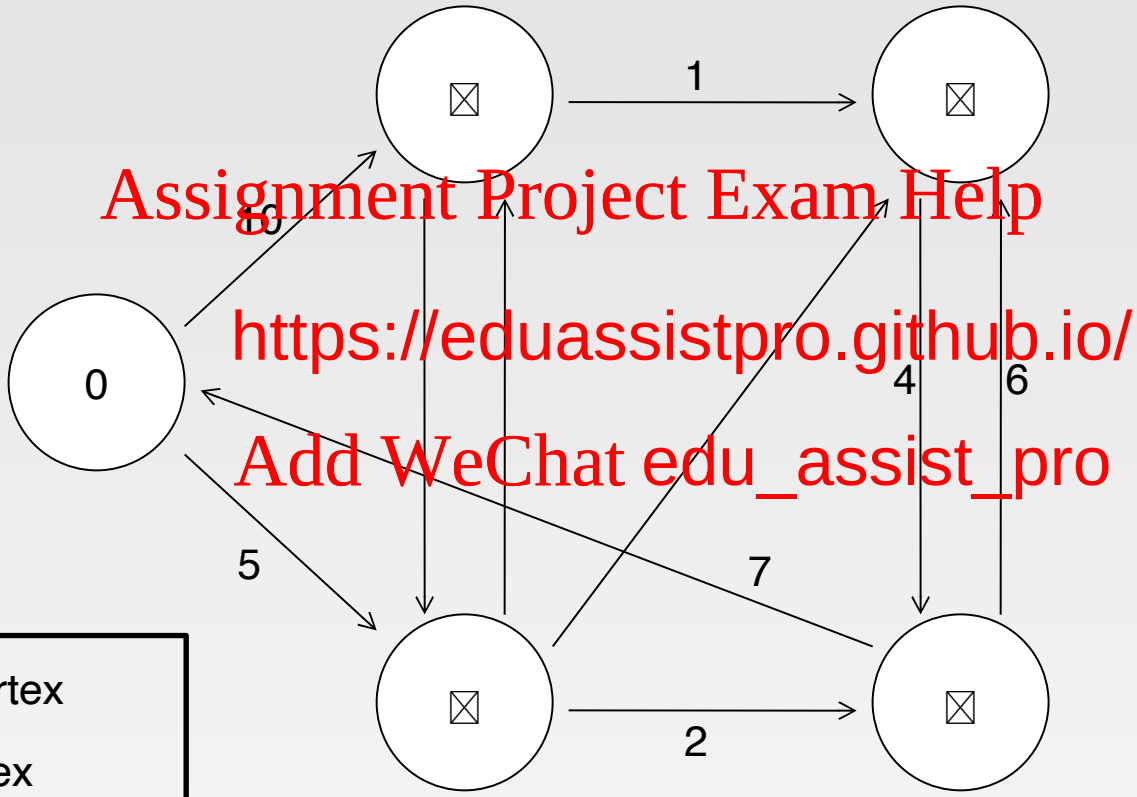
Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Pregel Computation Model (Cont')

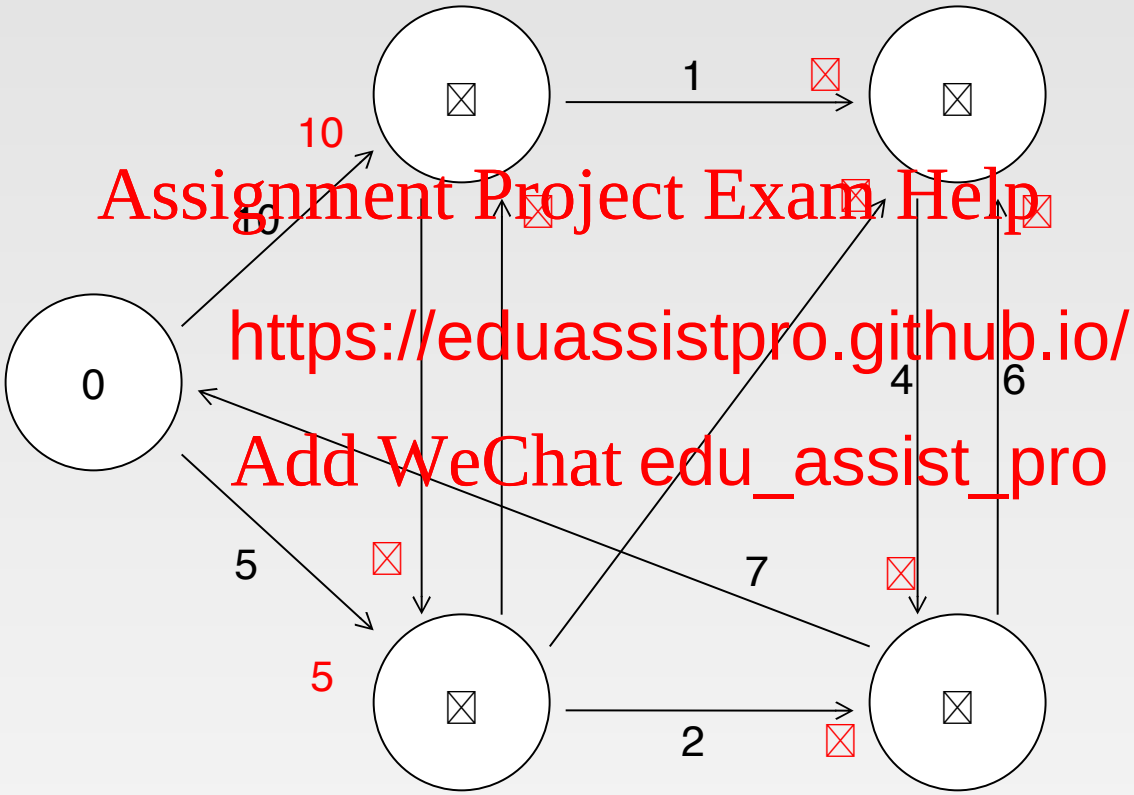
- Superstep: the vertices compute in parallel
  - Each vertex
    - Receives messages sent in the previous superstep
    - Executes the same user-defined function
    - Modifies edges
    - Sends messages to other v ceived in the next superstep)
    - Votes to halt if it has no further work to do
  - Termination condition
    - All vertices are simultaneously inactive
    - A vertex can choose to deactivate itself
    - Is "woken up" if new messages received

State machine for a vertex

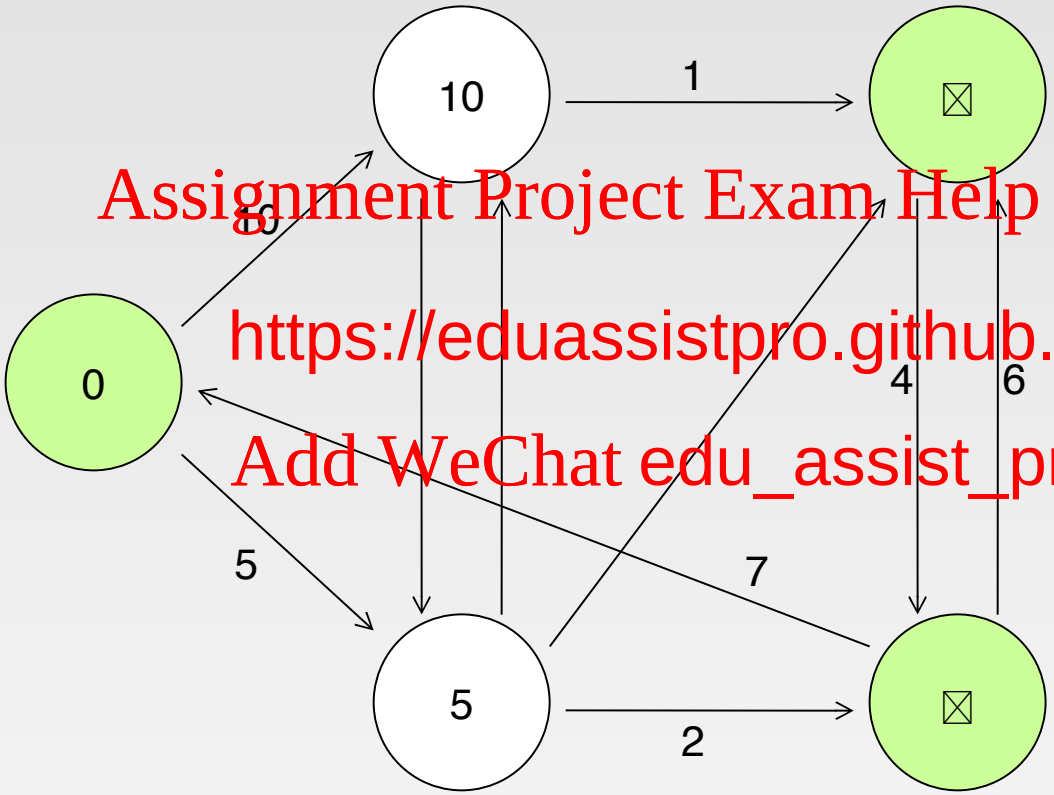
Example: SSSP – Parallel BFS in Pregel



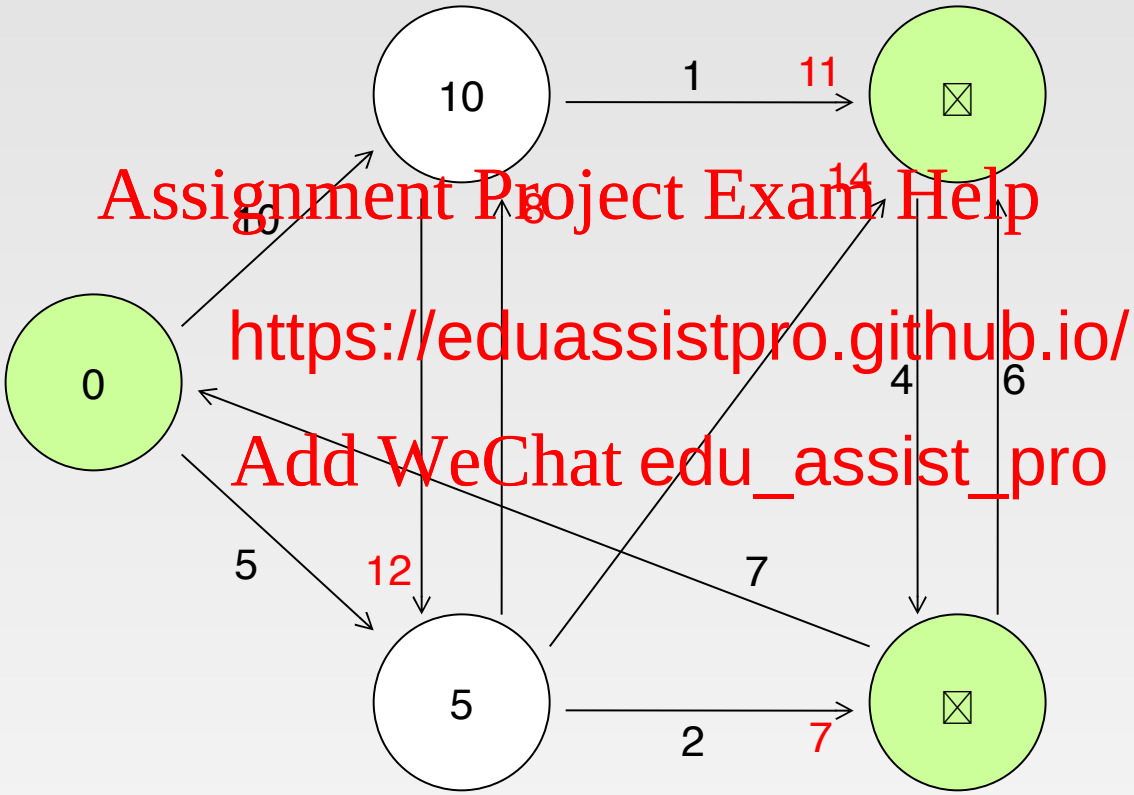
Example: SSSP – Parallel BFS in Pregel



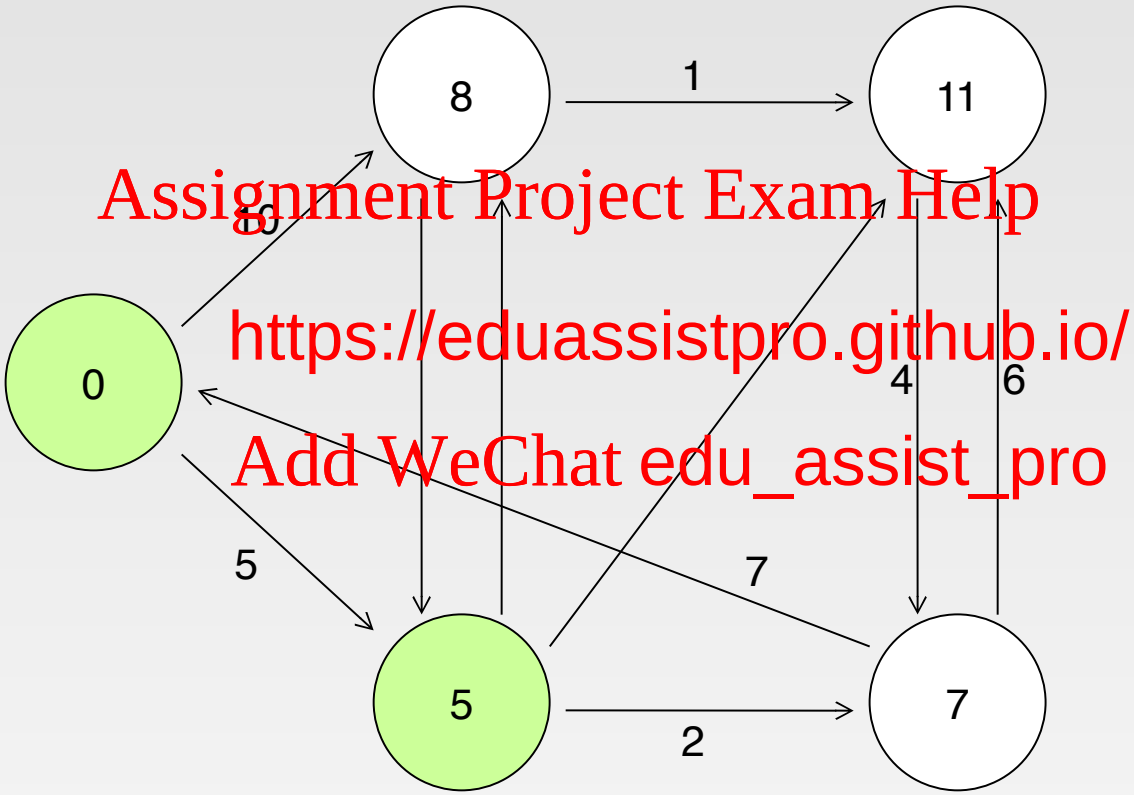
Example: SSSP – Parallel BFS in Pregel



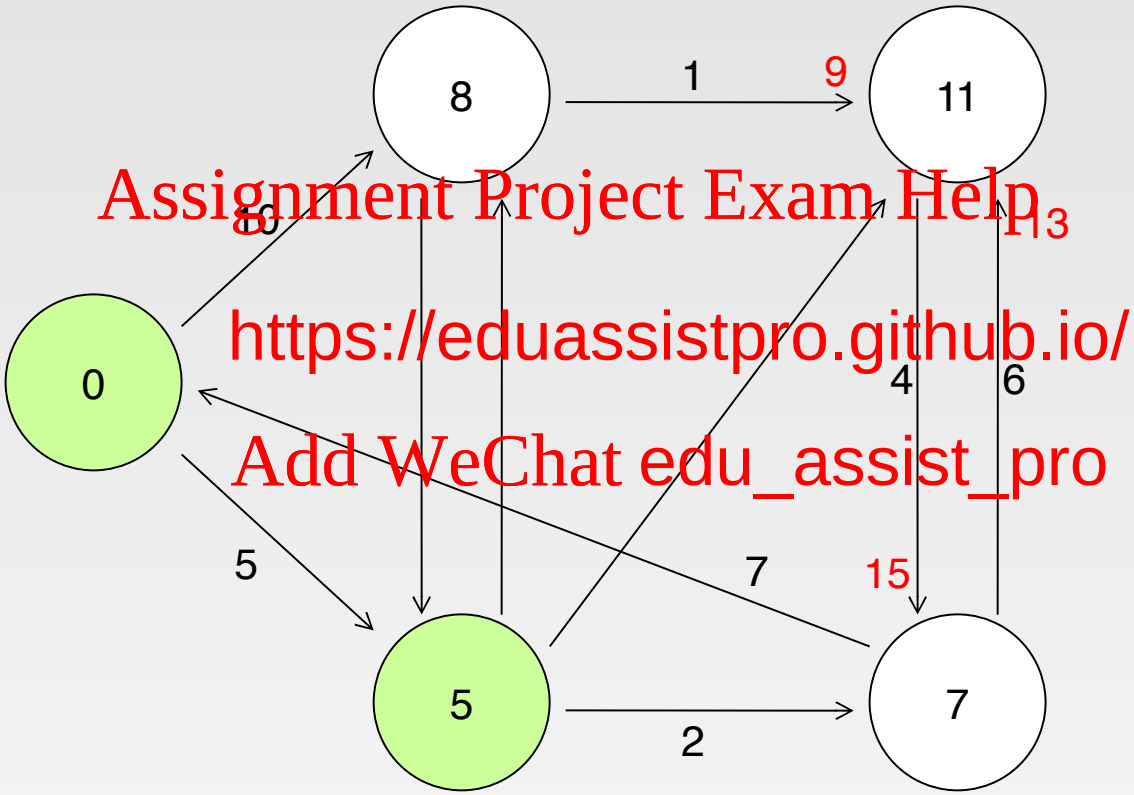
Example: SSSP – Parallel BFS in Pregel



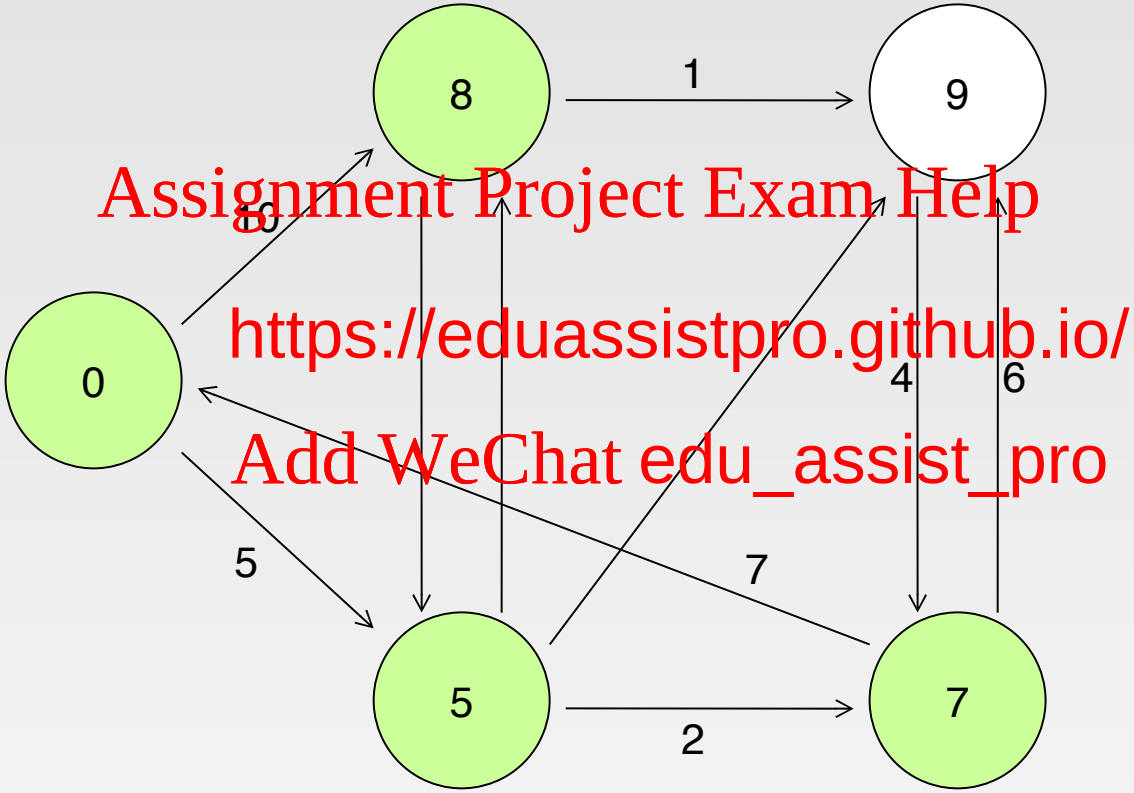
Example: SSSP – Parallel BFS in Pregel



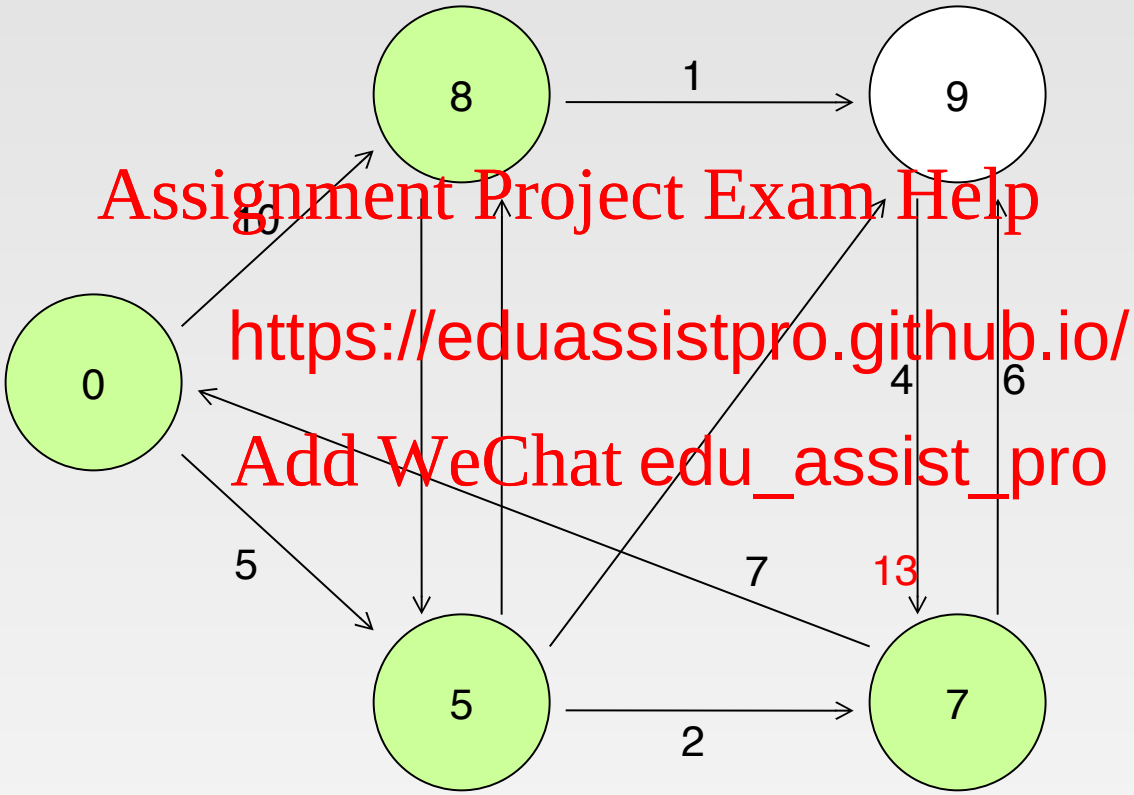
Example: SSSP – Parallel BFS in Pregel



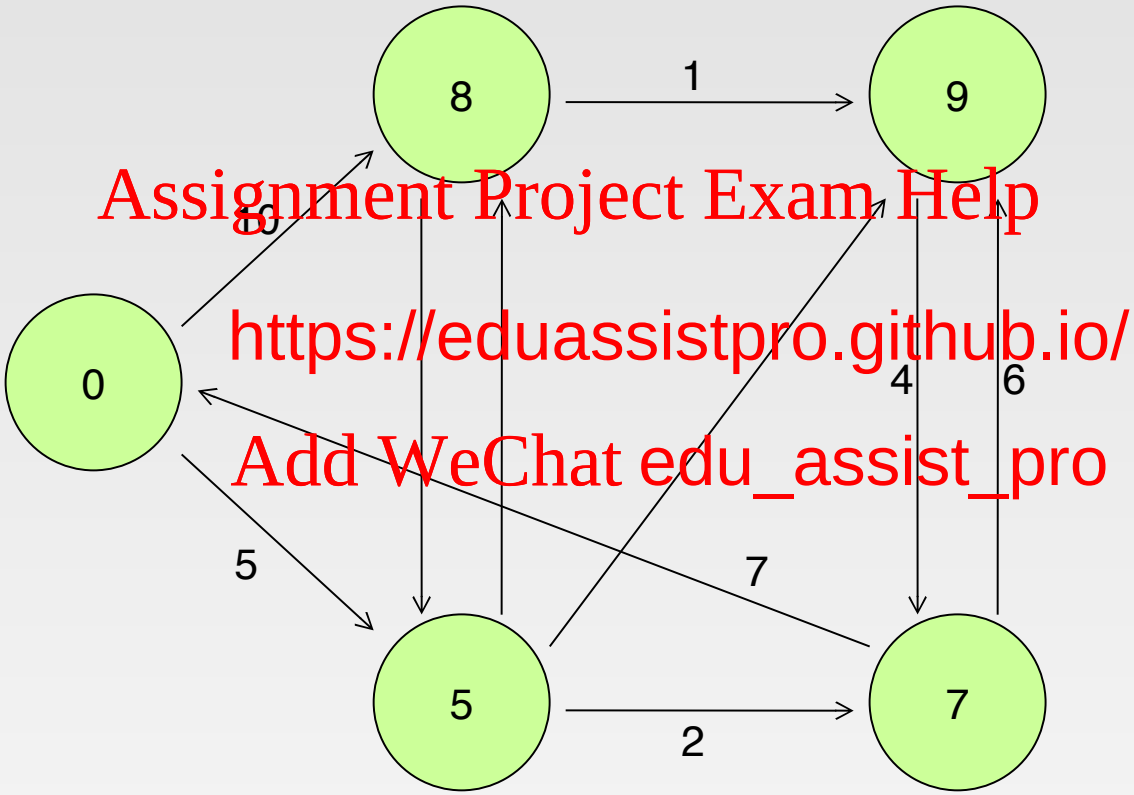
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Differences from MapReduce

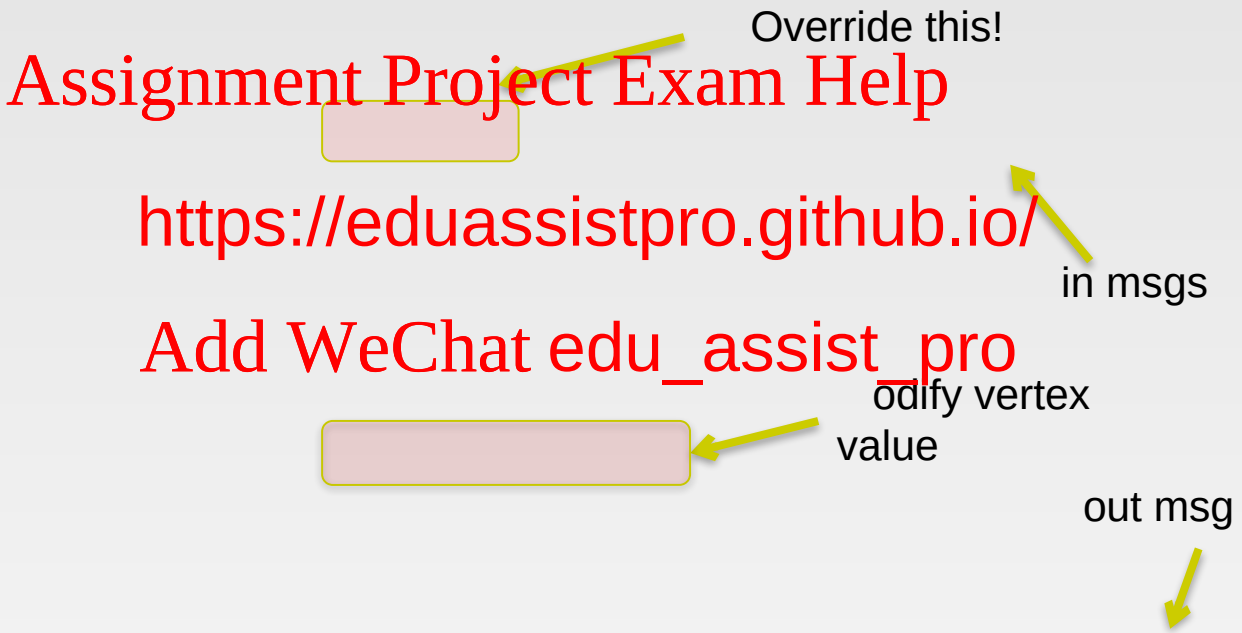
- Graph algorithms can be written as a series of chained MapReduce jobs

- Pregel
  - Keeps vertic t performs computation
  - Uses network transfers only for

- 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



Pregel: SSSP (C++)

```
class ShortestPathVertex : public Vertex<int, int, int>
{
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;

    for (; !ms
e());       mindist
      if (mind
        *MutableValue() = mi
        OutEdgeIterator iter                geIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target
        mindist + iter.GetValue());
      }
    VoteToHalt();
  }
};
```

Refer to the current node as u

aggregation

Messages: distances to u

ble value the current e

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

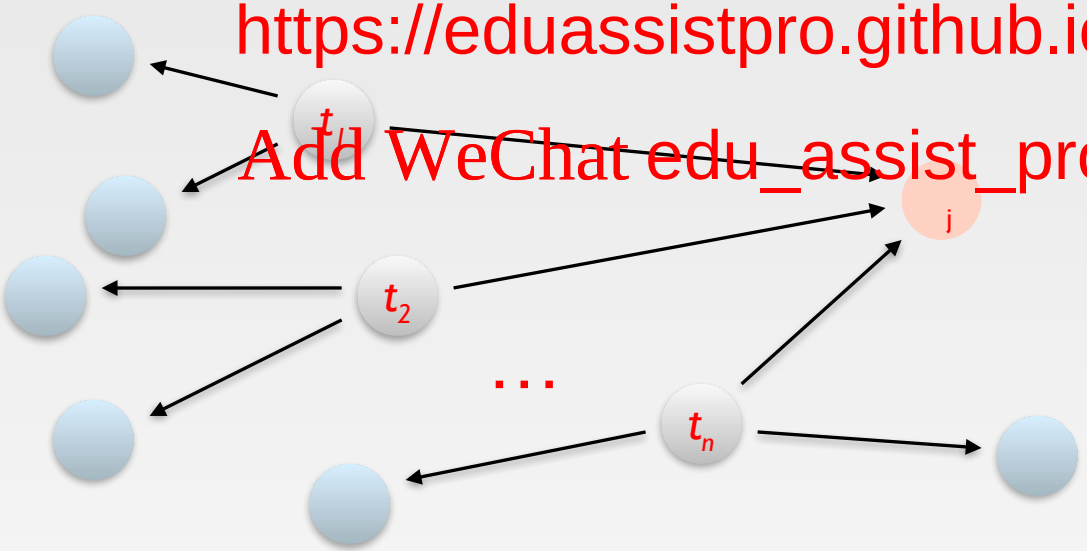
- Graph databases: Storage and Basic Operators
  - http://en.wikipedia.org/wiki/Graph_database
  - Neo4j (an open source graph database)
  - InfiniteGraph
  - VertexDB
  - … …
- Distributed Graph Processing (most nly)
  - Google's Pregel (vertex centere)
  - Giraph (Apache)
  - GraphX (Spark)
  - GraphLab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

References

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

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_j values
  - Each page it links to
  - Each target j ltiple in-bound links to compute r_j
  - Iterate until values converge

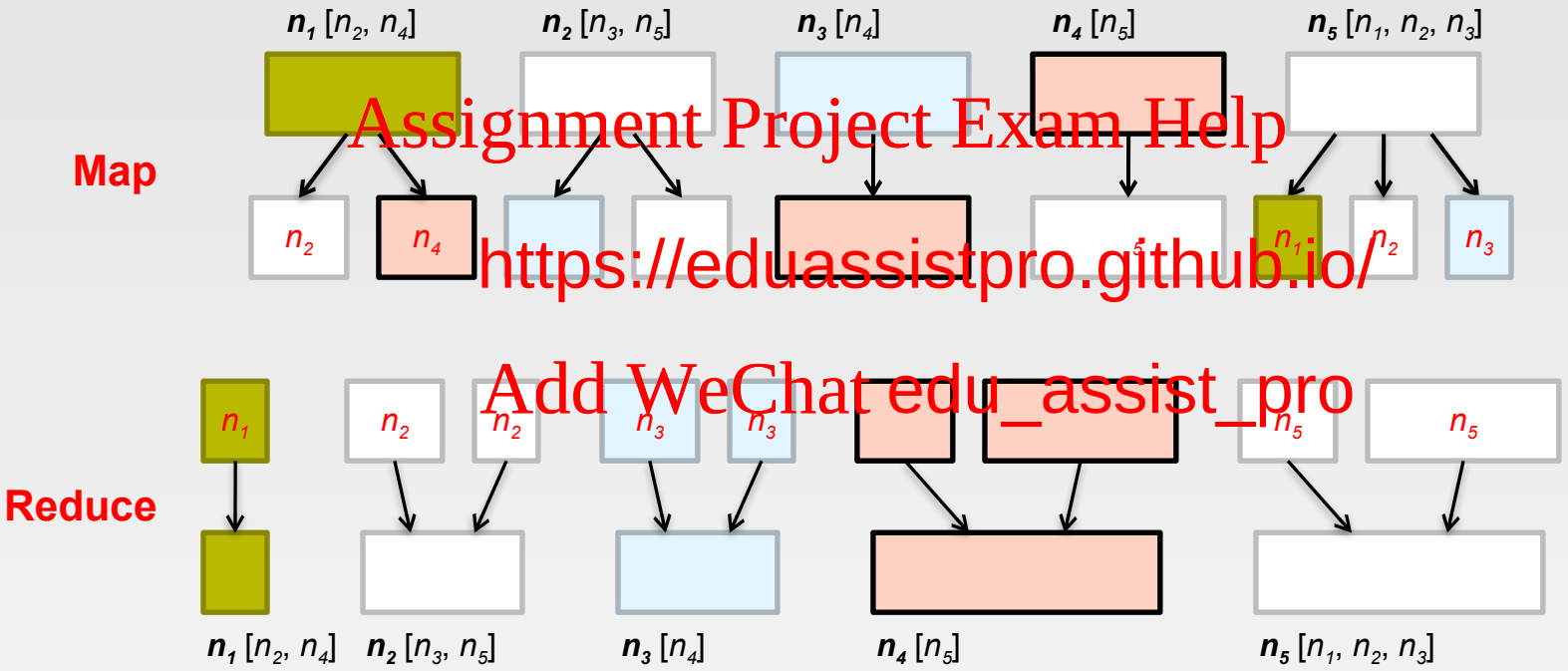
Simplified PageRank

- 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

Sample PageRank Iteration (1)

Sample PageRank Iteration (2)

PageRank in MapReduce (One Iteration)



PageRank Pseudo-Code

PageRank vs. BFS

	PageRank	BFS
Map	$r_j \cdot \frac{1}{d_j}$	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();
            SetValue(0.15 / NumVertices() + 0.85 * sum);
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
}
```