COMP9313: Big Data Management

Lecturer: Xin Cao

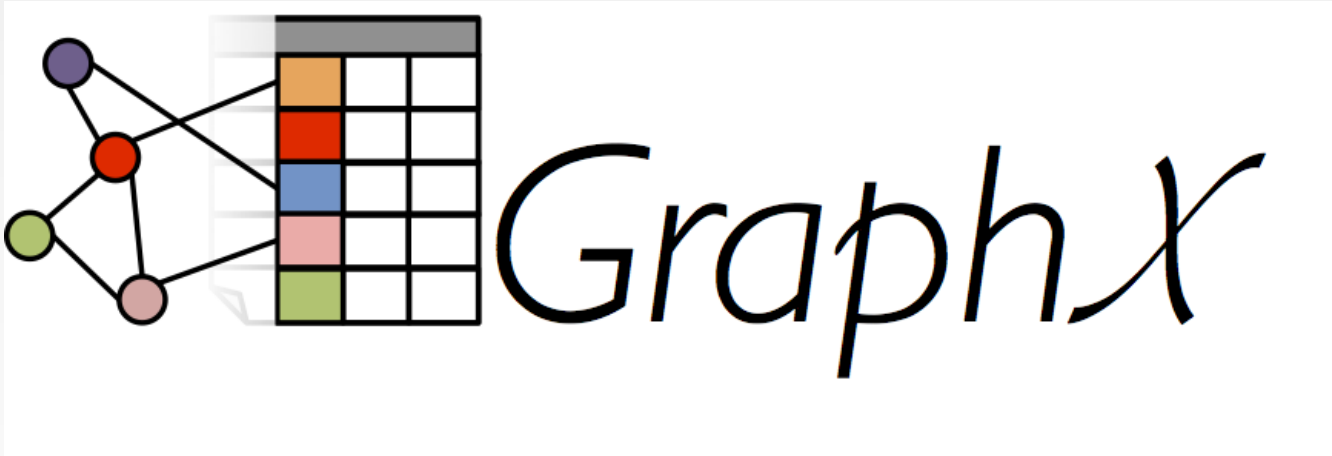
Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 5.2: Spark IV

Part 1: Spark GraphX

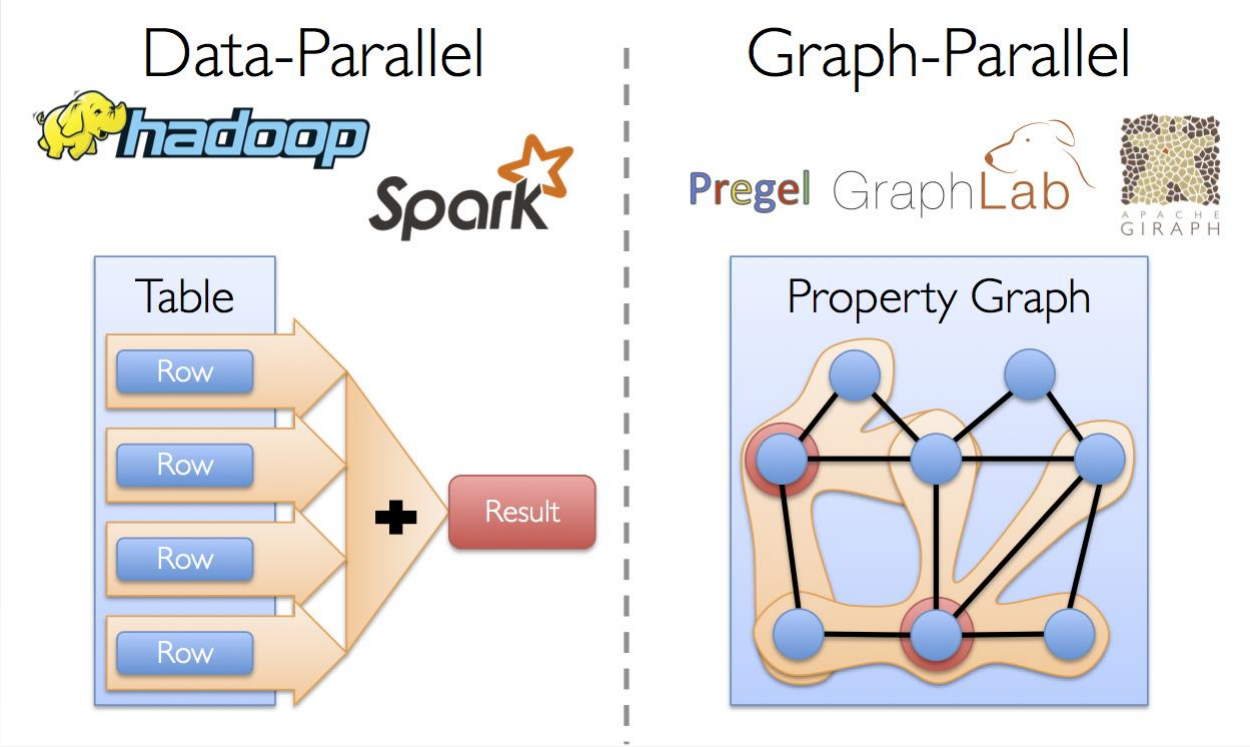
Spark GraphX

- **GraphX** is Apache Spark's API for graphs and graph-parallel computation.
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge
- To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices) as well as an optimized variant of the Pregel API
- GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



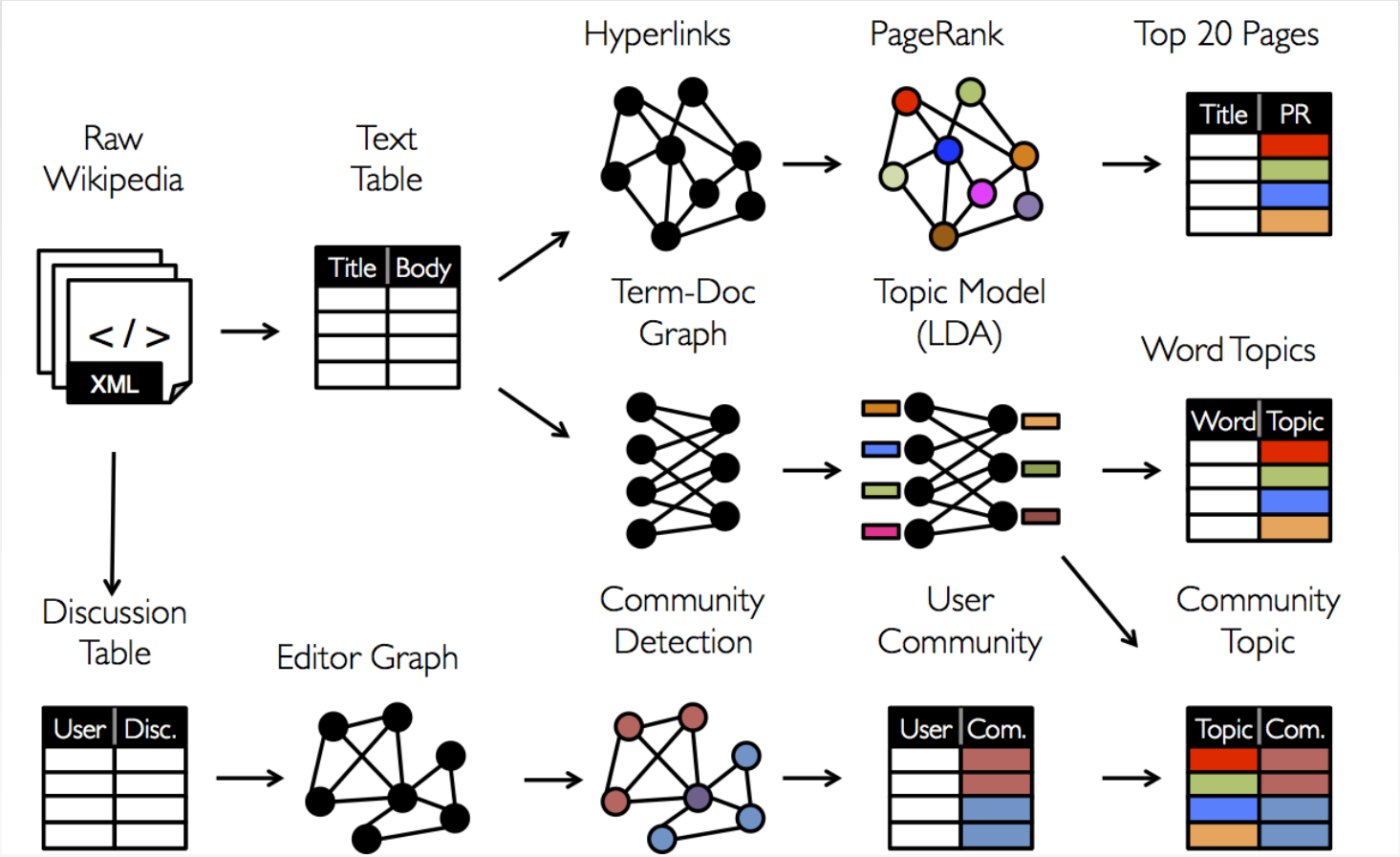
Graph-Parallel Computation

- The growing scale and importance of graph data has driven the development of numerous new graph-parallel systems (e.g., Giraph and GraphLab)
- These systems can efficiently execute sophisticated graph algorithms orders of magnitude faster than more general *data-parallel* systems.
  - Expose specialized APIs to simplify graph programming



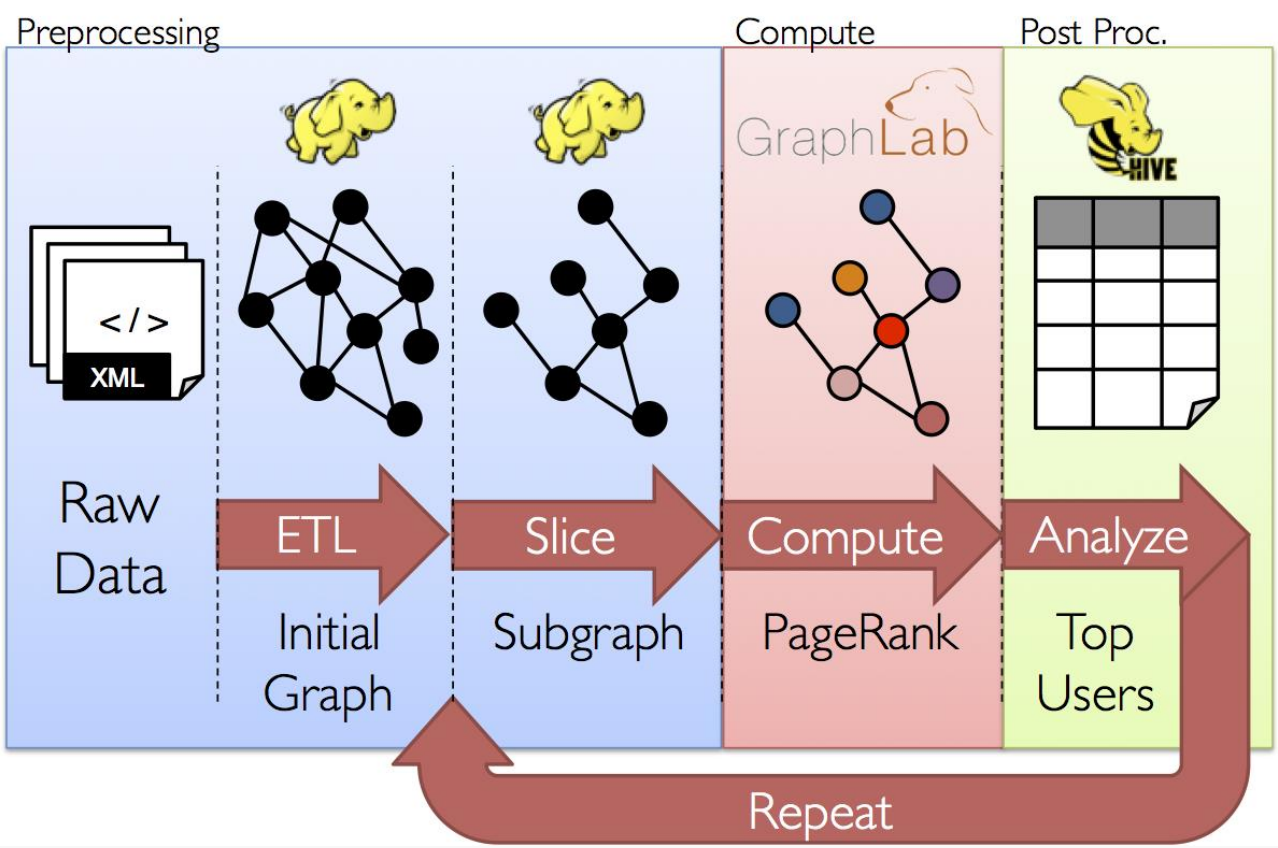
Specialized Systems May Miss the Bigger Picture

- It is often desirable to be able to move between table and graph views of the same physical data and to leverage the properties of each view to easily and efficiently express computation



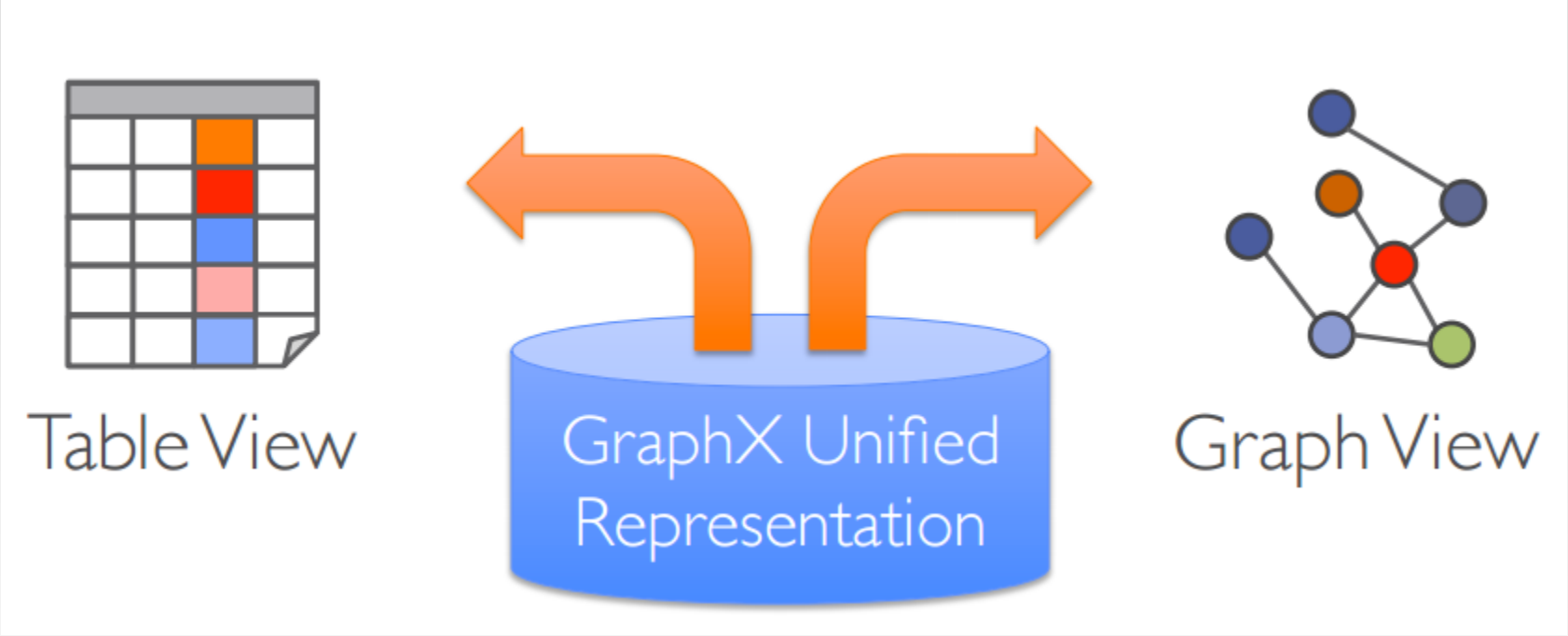
GraphX Motivation

- The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API.
- The GraphX API enables users to view data both as graphs and as collections (i.e., RDDs) without data movement or duplication.



GraphX Motivation

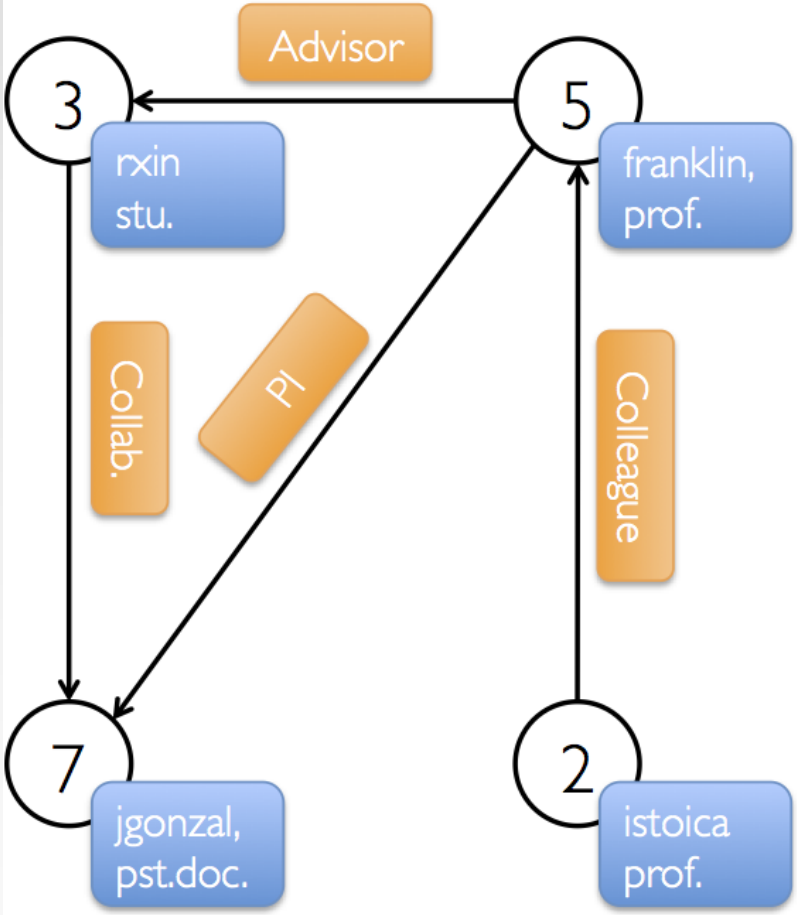
- Tables and Graphs are composable views of the same physical data



- Each view has its own operators that exploit the semantics of the view to achieve efficient execution

View a Graph as a Table

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Table Operators

- Table (RDD) operators are inherited from Spark:

map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
leftOuterJoin	cross	save
rightOuterJoin	zip	...

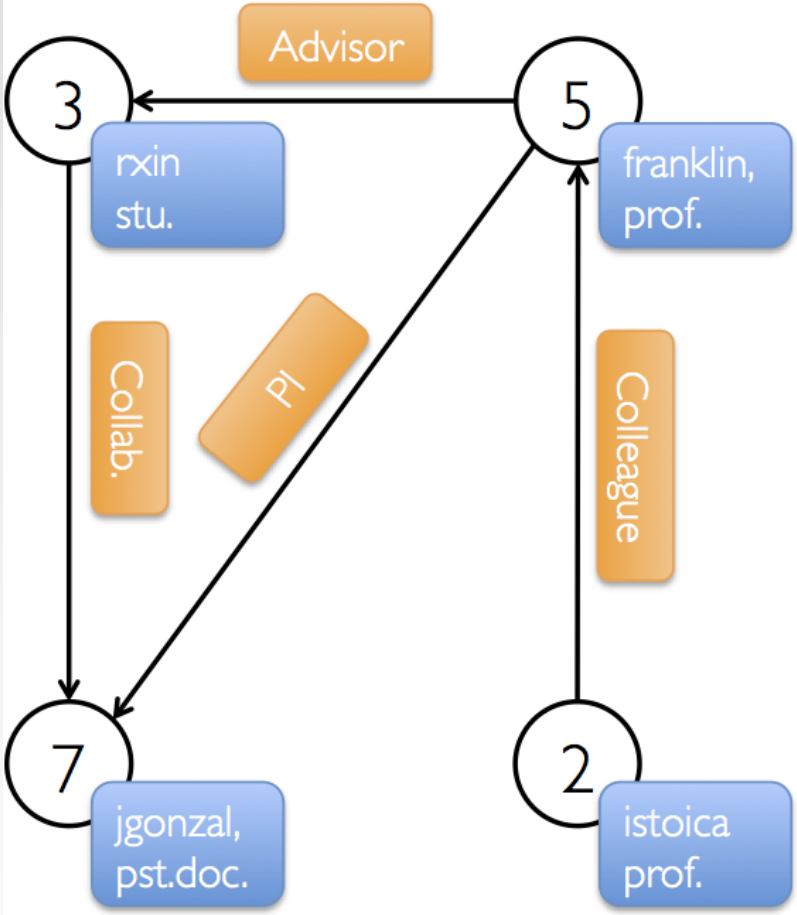
The Property Graph

- The property graph is a directed multigraph with user defined objects attached to each vertex and edge.
- A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex
- The property graph is parameterized over the vertex (VD) and edge (ED) types. These are the types of the objects associated with each vertex and edge respectively.
- Each vertex is keyed by a *unique* 64-bit long identifier (VertexID). Similarly, edges have corresponding source and destination vertex identifiers.
- Logically the property graph corresponds to a pair of typed collections (RDDs) encoding the properties for each vertex and edge.

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Example Property Graph

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

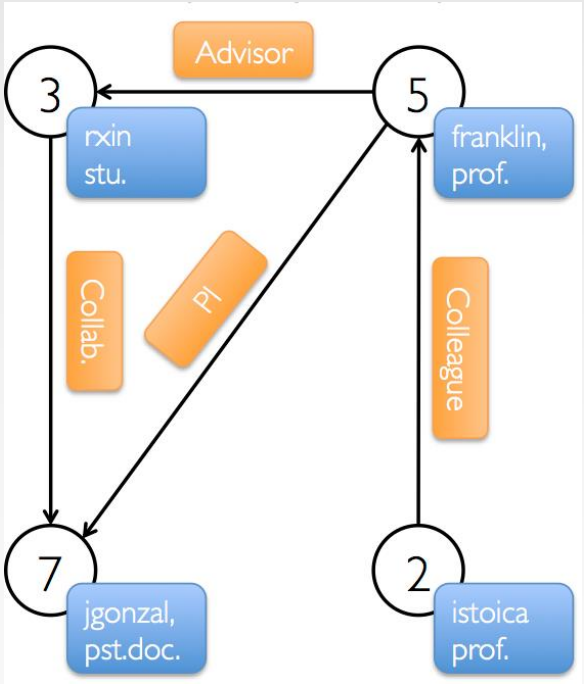
GraphX Example

- Import Spark and GraphX into your project

```
import org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

- We begin by creating the property graph from arrays of vertices and edges

```
val vertexArray = Array(
  (3L, ("rxin", "student")),
  (7L, ("jgonzal", "postdoc")),
  (5L, ("franklin", "prof")),
  (2L, ("istoica", "prof"))
)
val edgeArray = Array(
  Edge(3L, 7L, "collab"),
  Edge(5L, 3L, "advisor"),
  Edge(2L, 5L, "colleague"),
  Edge(5L, 7L, "pi"),
)
```



Construct a Property Graph

- The most general method of constructing a property graph is to use the Graph object

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(vertexArray)
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(edgeArray)
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

  - Edges have a srcId and a dstId corresponding to the source and destination vertex identifiers.
  - In addition, the Edge class has an attr member which stores the edge property

Deconstruct a Property Graph

- In many cases we will want to extract the vertex and edge RDD views of a graph
- The graph class contains members (graph.vertices and graph.edges) to access the vertices and edges of the graph

```
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

  - Note that graph.vertices returns an VertexRDD[(String, String)] which extends RDD[(VertexId, (String, String))] and so we use the scala case expression to deconstruct the tuple.
  - graph.edges returns an EdgeRDD containing Edge[String]objects. We could have also used the case class type constructor as in the following:

    graph.edges.filter { **case Edge**(src, dst, prop) **=>** src > dst }.count

Deconstruct a Property Graph

- Another example: use graph.vertices to display the names of the users who are professors

```
graph.vertices.filter { case (id, (name, pos)) => pos == "prof"
}.collect.foreach { case (id, (name, age)) => println(s"$name is
Professor") }
```

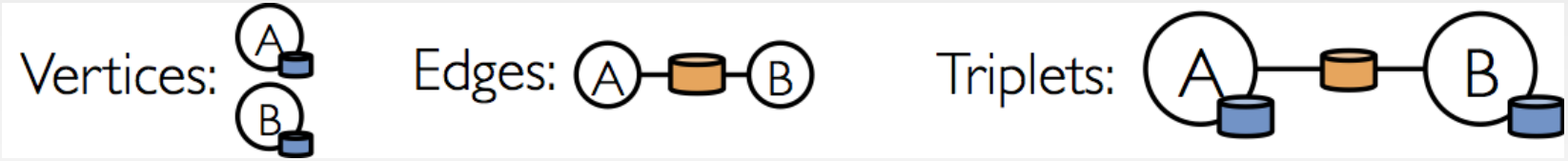
  - We first get the vertices who are professors. It can also be written as: graph.vertices.filter(x => x._2._2==“prof”).collect
  - Next, we print their names, which can also be written as: ……foreach(x=>println(x._2._1+" is Professor"))

Triplet View

- The triplet view logically joins the vertex and edge properties yielding an RDD[EdgeTriplet[VD, ED]] containing instances of the EdgeTriplet class
- This *join* can be expressed in the following SQL expression:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

or graphically as:



EdgeTriplet class

- The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.
- We can use the triplet view of a graph to render a collection of strings describing relationships between users.

```
// Constructed from above
val graph: Graph[(String, String), String]
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

Graph Operators

- Property graphs have a collection of basic operators that take user defined functions and produce new graphs with transformed properties and structure (like RDD operations).

```
class Graph[VD, ED] {
        // Information about the Graph
        val numEdges: Long
        val numVertices: Long
        val inDegrees: VertexRDD[Int]
        val outDegrees: VertexRDD[Int]
        val degrees: VertexRDD[Int]
        // Views of the graph as collections
        val vertices: VertexRDD[VD]
        val edges: EdgeRDD[ED]
        val triplets: RDD[EdgeTriplet[VD, ED]]
        // Transform vertex and edge attributes
        def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
        def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
        def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
        // Modify the graph structure
        def reverse: Graph[VD, ED]
        def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
        // …other operators…
}
```

Property Operators

- Like the RDD map operator, the property graph contains the following:

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

  - Each of these operators yields a new graph with the vertex or edge properties modified by the user defined map function.
  - Note that in each case the graph structure is unaffected

```
val newVertices = graph.vertices.map { case (id, attr) => (id,
mapUdf(id, attr)) }
val newGraph = Graph(newVertices, graph.edges)
```

```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

  - The second one can preserve the structural indices of the original graph and would benefit from the GraphX system optimizations

Structural Operators

- Currently GraphX supports only a simple set of commonly used structural operators

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

  - val graphR = graph.reverse
  - val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "prof")

Neighborhood Aggregation

- A key step in many graph analytics tasks is aggregating information about the neighborhood of each vertex.
- The core aggregation operation in GraphX is aggregateMessages.

```
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

  - A user-defined sendMsg function, to send messages for each edge triplet in the graph
  - A user-defined mergeMsg function, to aggregate those messages at their destination vertex.

Neighborhood Aggregation

- We can use the aggregateMessages operator to compute the average age of the more senior followers of each user

```
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst((1, triplet.srcAttr))
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) =>
    value match { case (count, totalAge) => totalAge / count } )
// Display the results
```

Pregel Operators

```
def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
              … …
     }
```

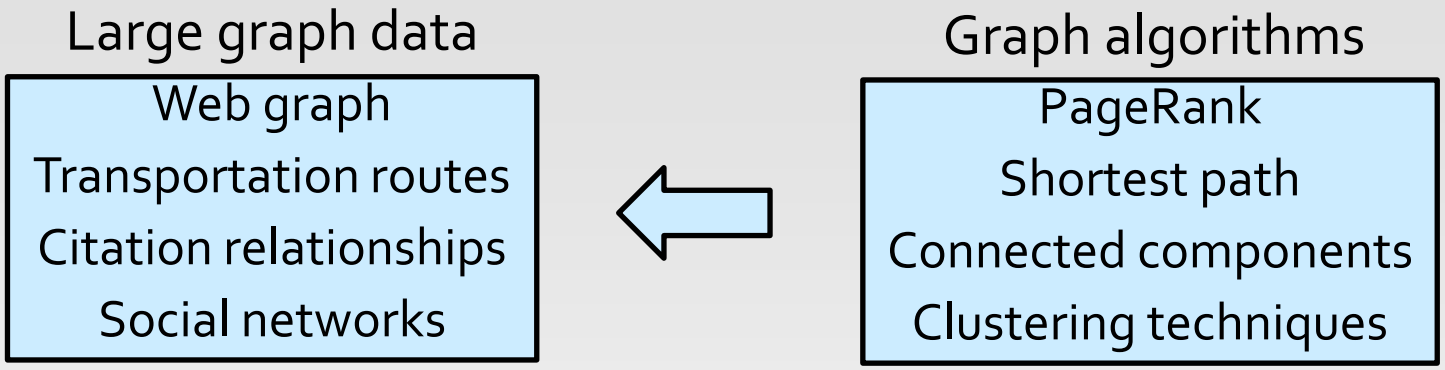
- The first argument list contains configuration parameters including the initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).

- The second argument list contains the user defined functions for receiving messages (the vertex program vprog), computing messages (sendMsg), and combining messages mergeMsg.

Pregel Introduction

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

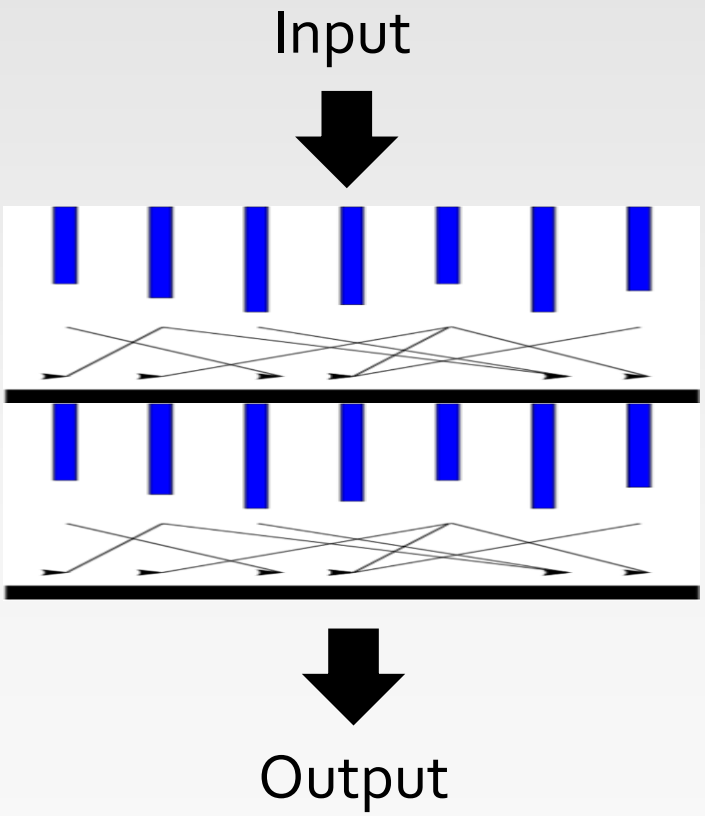
- Processing: a series of supersteps
- Vertex: computation is defined to run on each vertex
- Superstep S: *all vertices compute in parallel; each vertex v may*
  - receive messages sent to v from superstep S – 1;
  - perform some computation: modify its states and the states of its outgoing edges
 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

Message passing

Vertex-centric, message passing

Pregel Computation Model

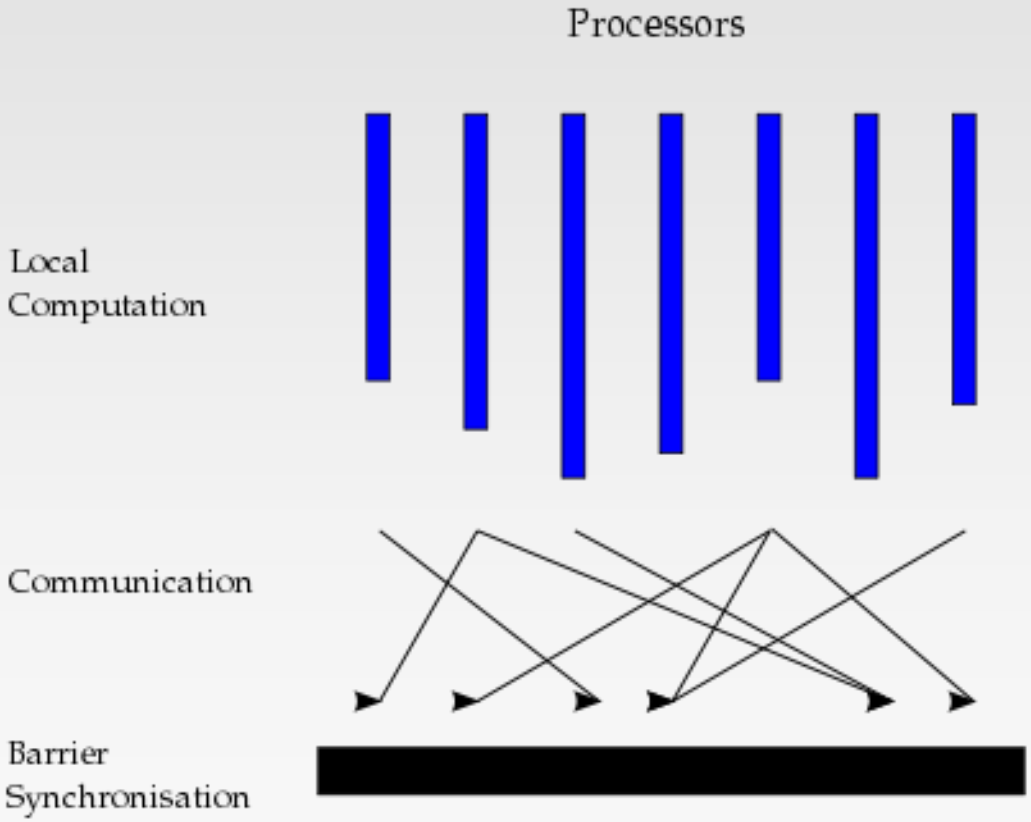
- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture



Supersteps
(a sequence of iterations)

Pregel Computation Model (Cont')

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

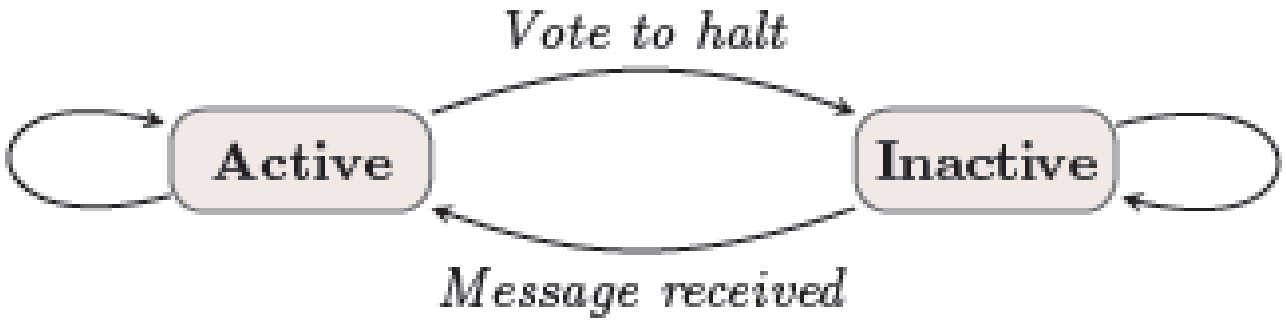


Source: http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Pregel Computation Model (Cont')

- Superstep: the vertices compute in parallel
  - Each vertex

State machine for a vertex

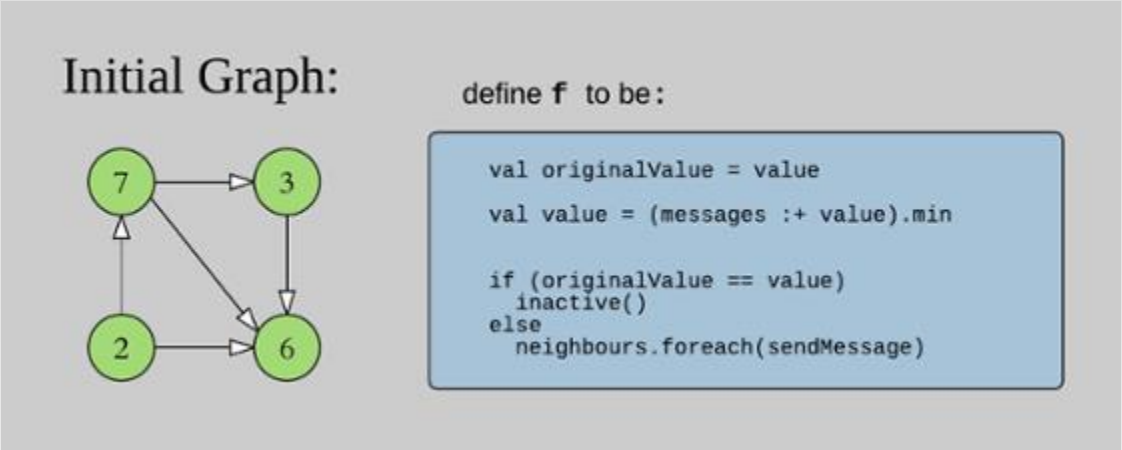


  - Termination condition
    - All vertices are simultaneously inactive
    - A vertex can choose to deactivate itself
    - Is "woken up" if new messages received

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

Example: Find the minimum value in a graph

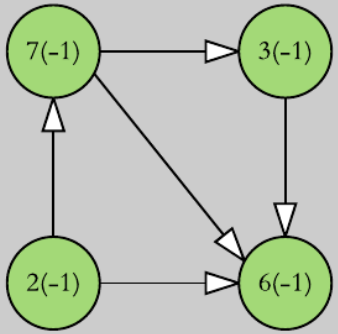


- The pseudo-code definition of f is also given above, it will:
  - Set originalValue to the current value of the vertex.
  - Mutate the value of the vertex to the minimum of all the incoming messages and originalValue.
  - If originalValue and value are the same, then we will render the vertex inactive. Otherwise, send message out to all its outgoing neighbours.

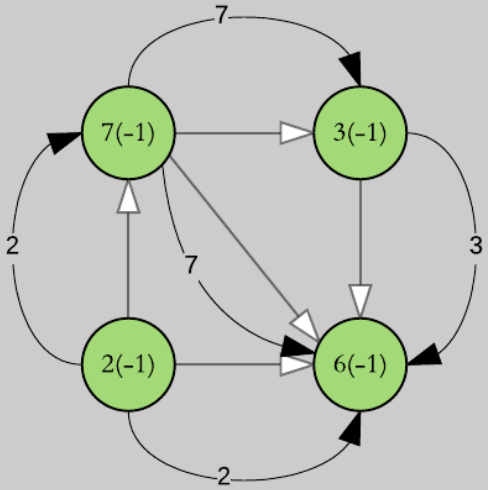
Superstep 0

Superstep 0:

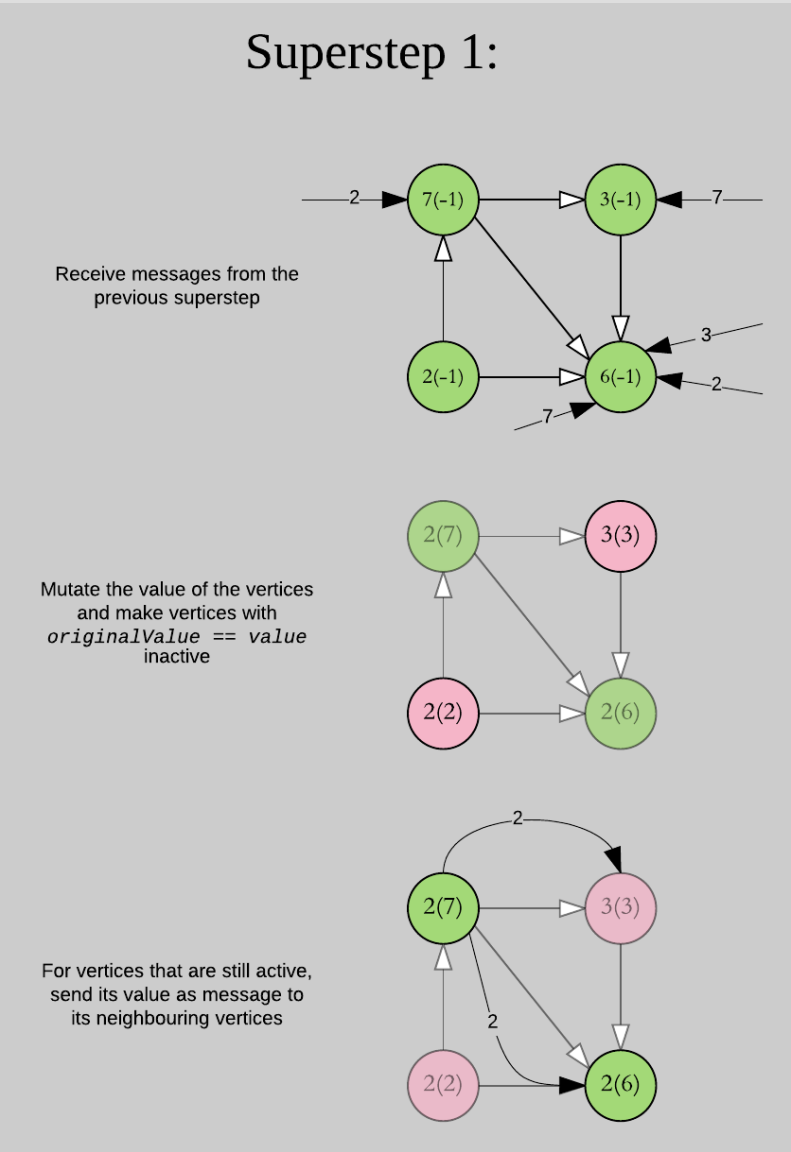
Initialise the graph
with `-1` picked as the
`originalValue`



All active vertices
send its value as message to
its neighbouring vertices



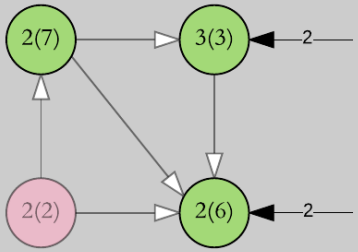
Superstep 1


Superstep 1:
Receive messages from the previous superstep
2
7(-1)
3(-1)
7
3
2(-1)
6(-1)
2
7
Mutate the value of the vertices and make vertices with originalValue == value inactive
2(7)
3(3)
2(2)
2(6)
For vertices that are still active, send its value as message to its neighbouring vertices
2
2(7)
3(3)
2
2(2)
2(6)

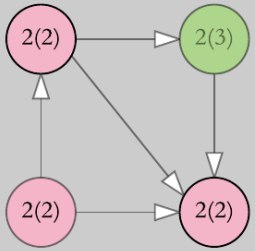
Superstep 2

Superstep 2:

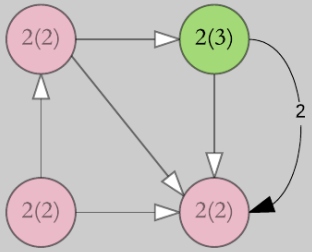
Receive message from the previous superstep



Mutate the value of the vertices and make vertices with *originalValue == value* inactive



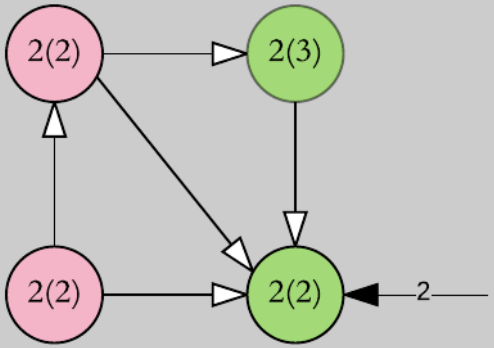
For vertices that are still active, send its value as message to its neighbouring vertices



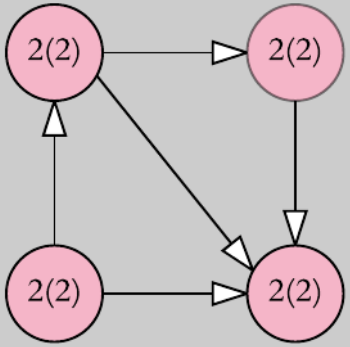
Superstep 3

Superstep 3:

Receive message from the previous superstep

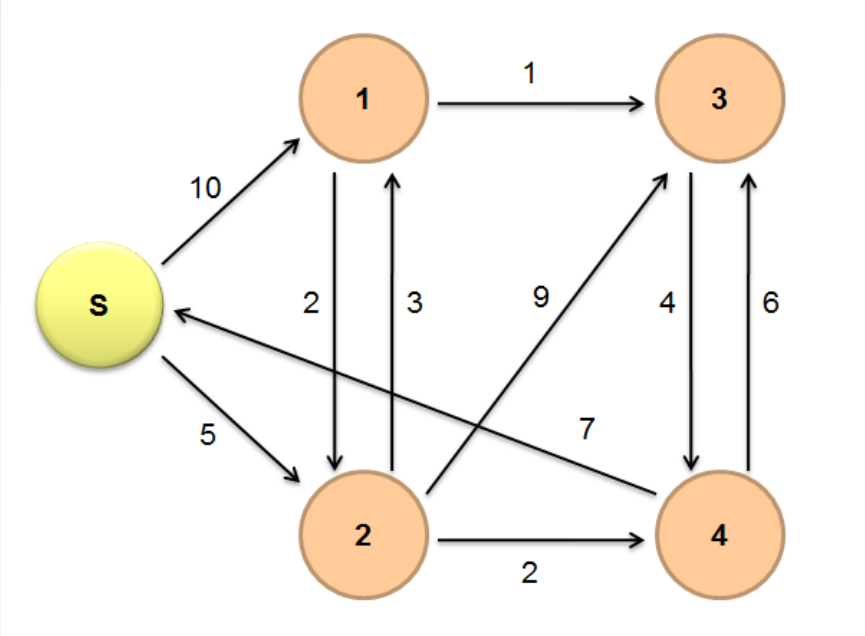


Mutate the value of the vertices and make vertices with *originalValue == value* inactive



Single-Source Shortest Path (SSSP)

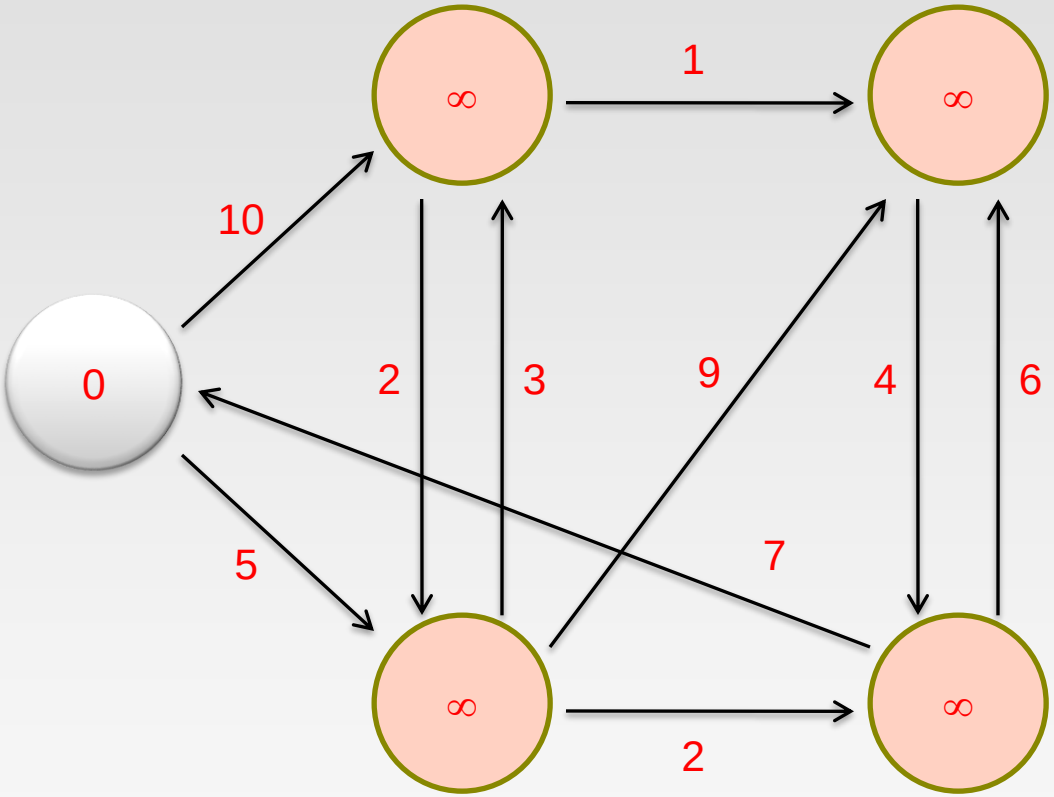
- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Dijkstra's Algorithm:
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



Dijkstra's Algorithm

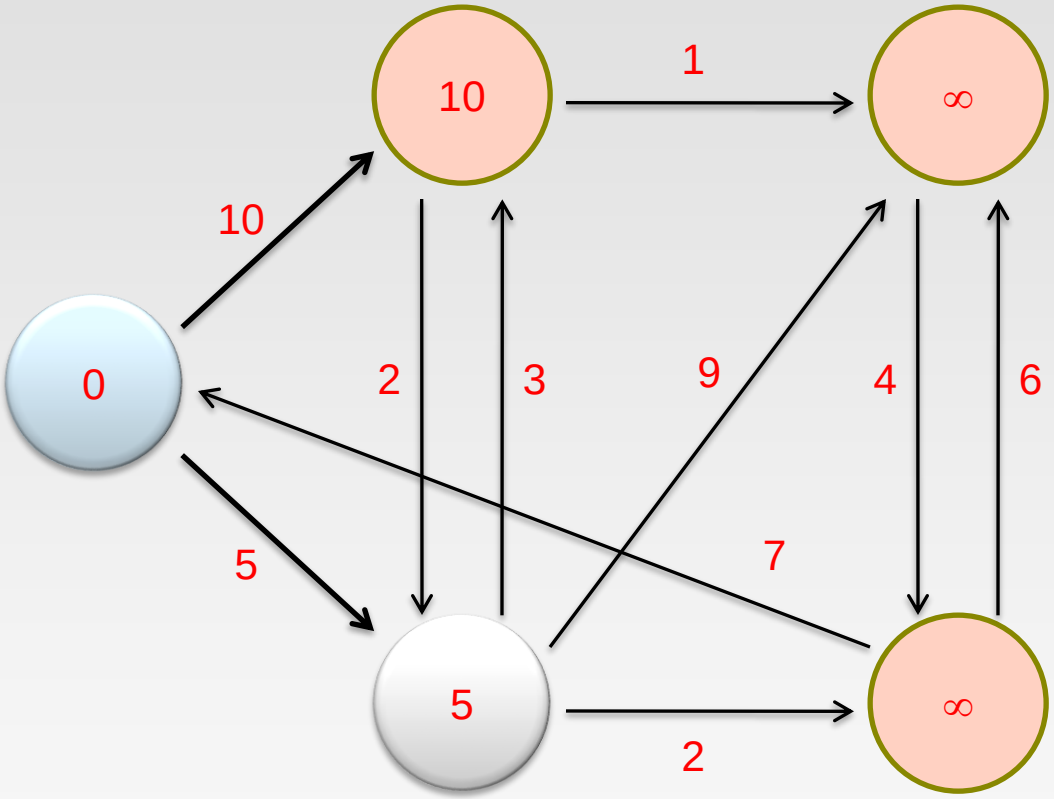
```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0
3:      for all vertex v ∈ V do
4:          d[v] ← ∞
5:      Q ← {V}
6:      while Q ≠ ∅ do
7:          u ← EXTRACTMIN(Q)
8:          for all vertex v ∈ u.ADJACENCYLIST do
9:              if d[v] > d[u] + w(u, v) then
10:                 d[v] ← d[u] + w(u, v)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

Dijkstra's Algorithm Example



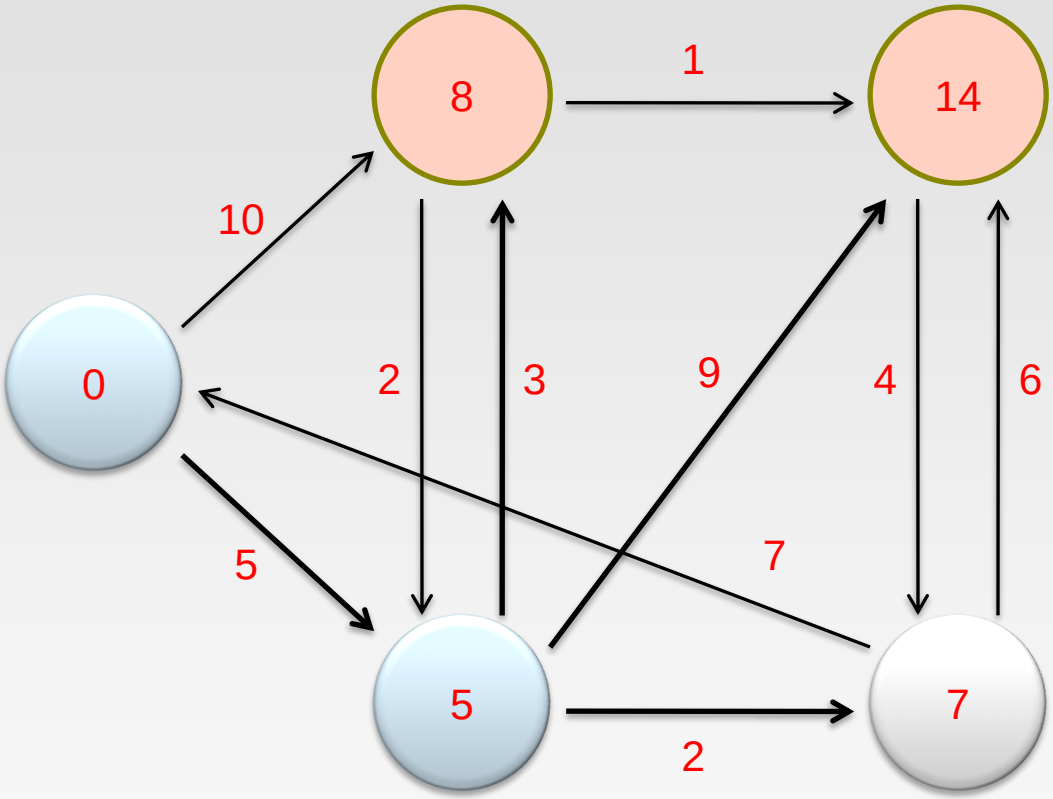
Example from CLR

Dijkstra's Algorithm Example



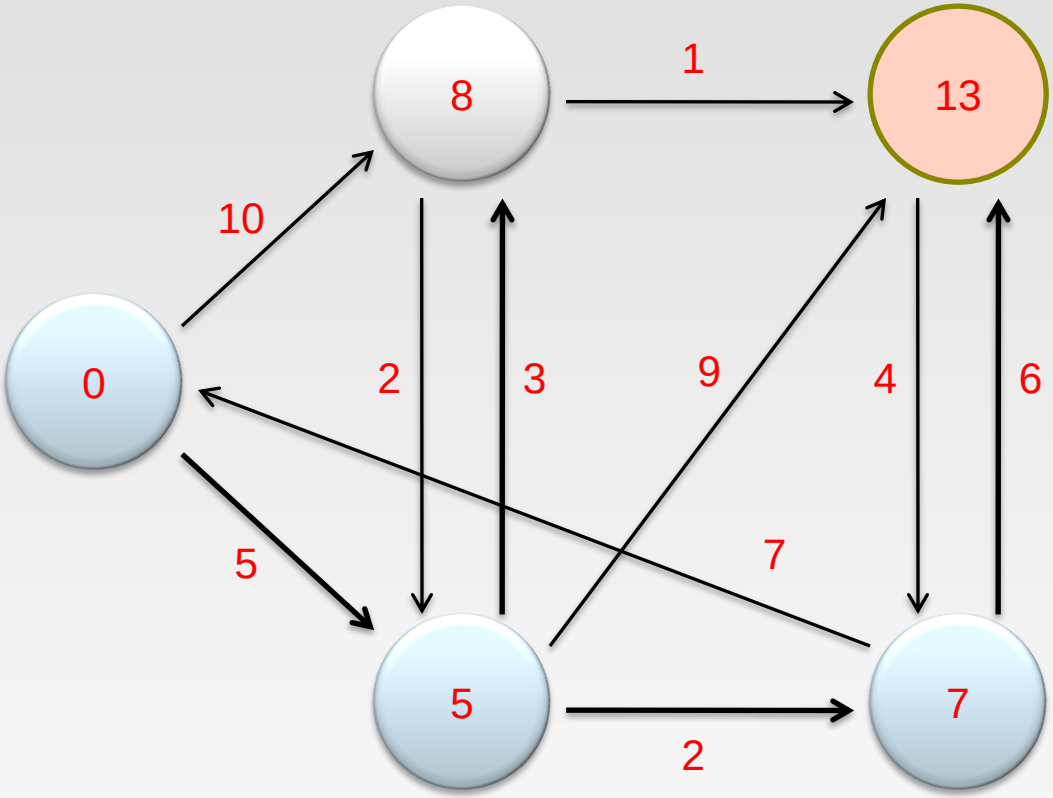
Example from CLR

Dijkstra's Algorithm Example



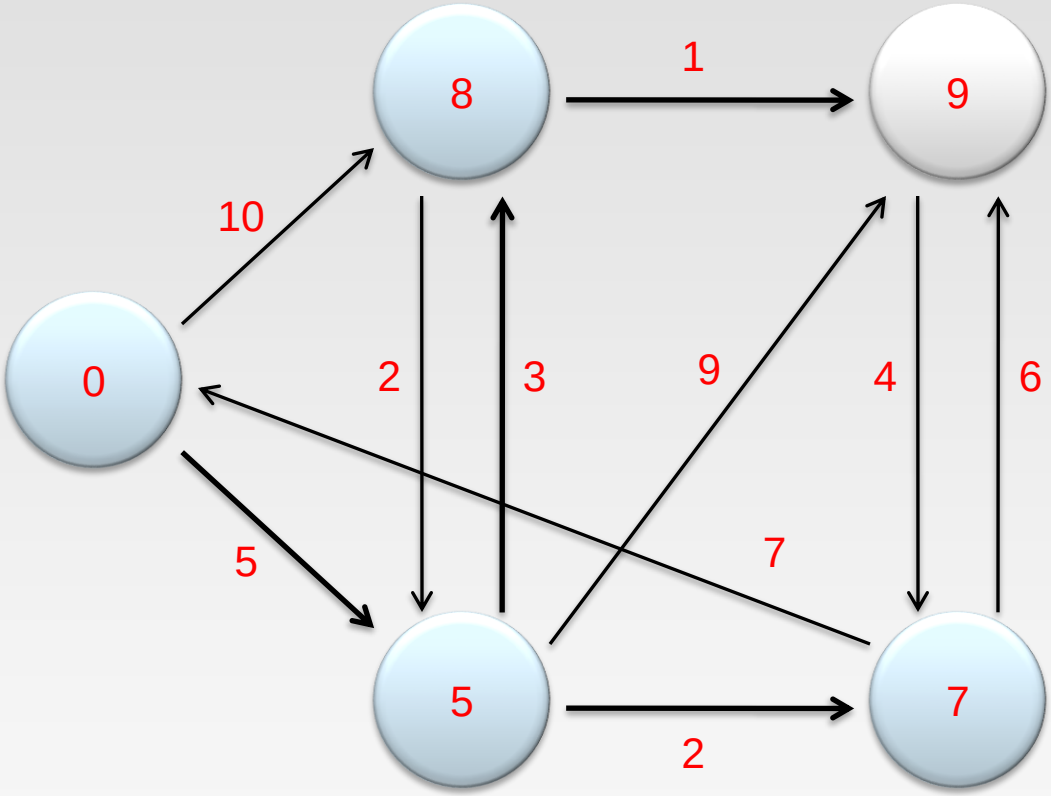
Example from CLR

Dijkstra's Algorithm Example



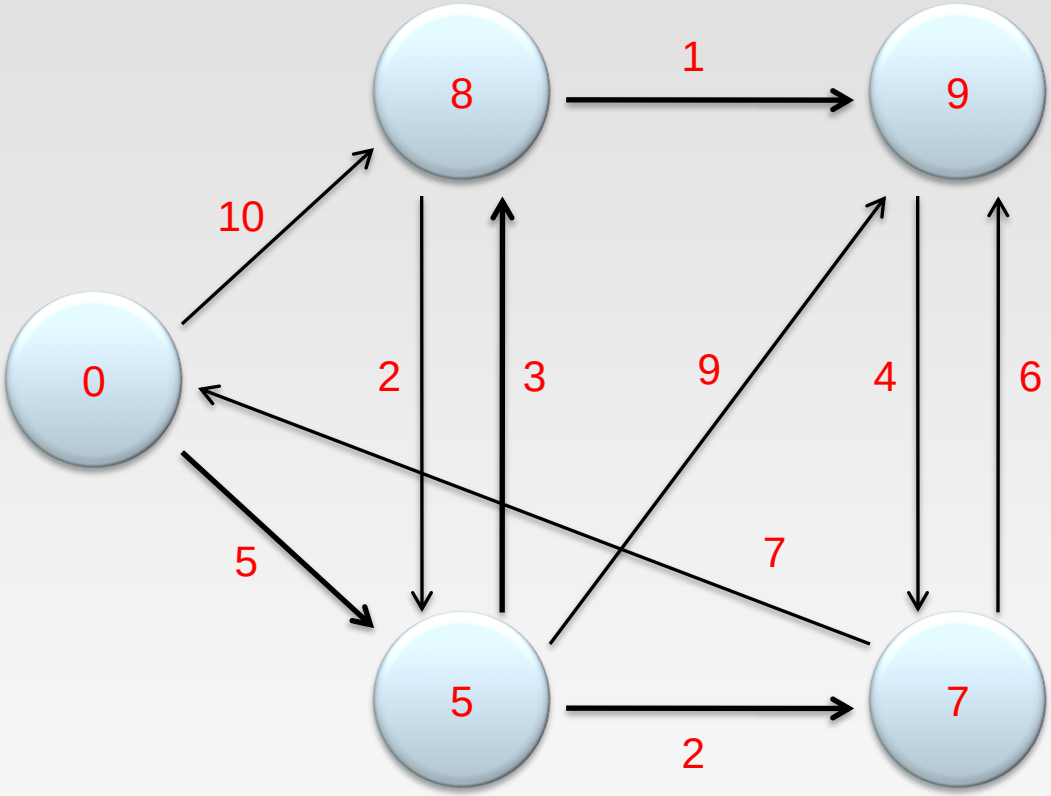
Example from CLR

Dijkstra's Algorithm Example



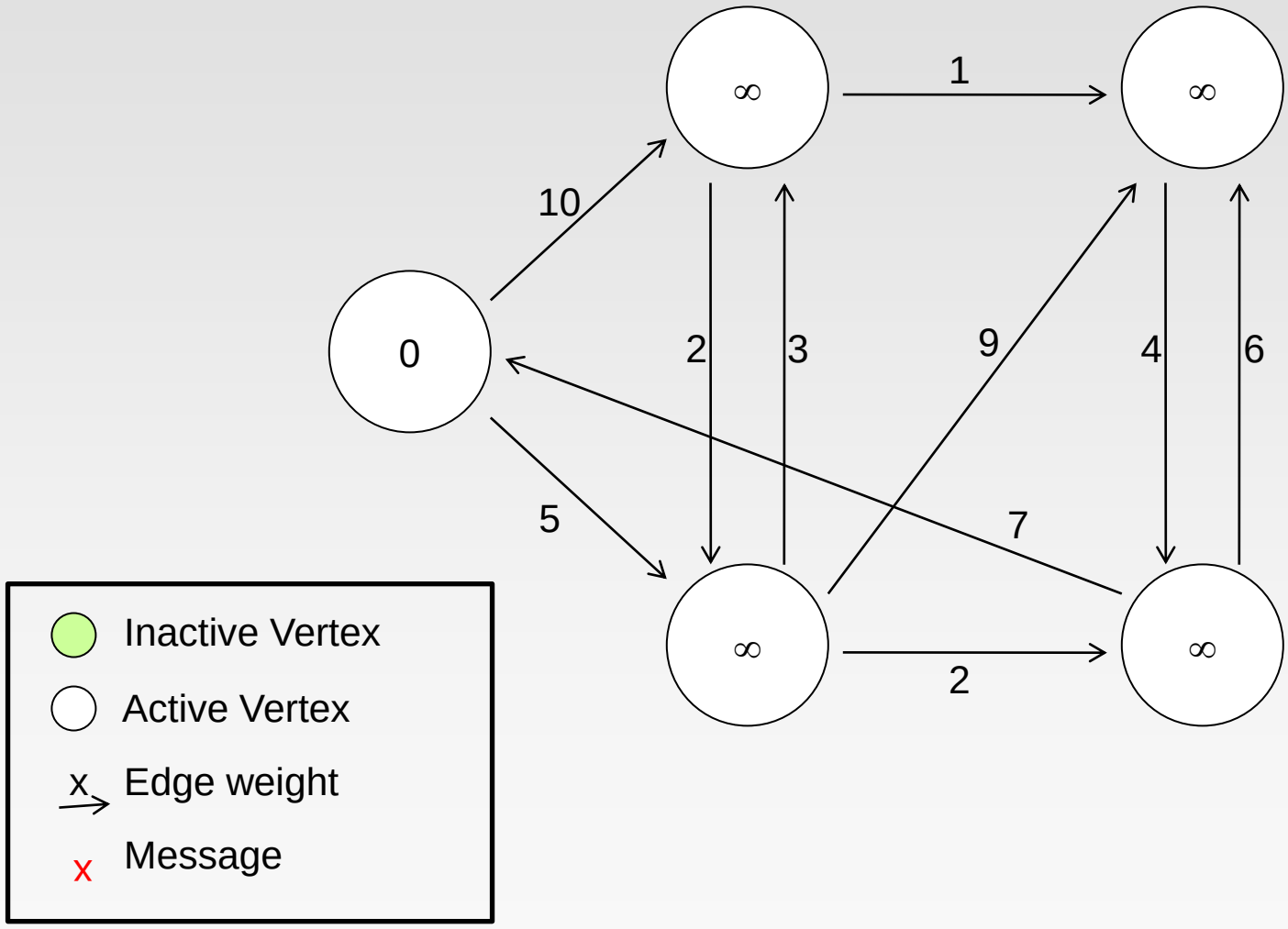
Example from CLR

Dijkstra's Algorithm Example

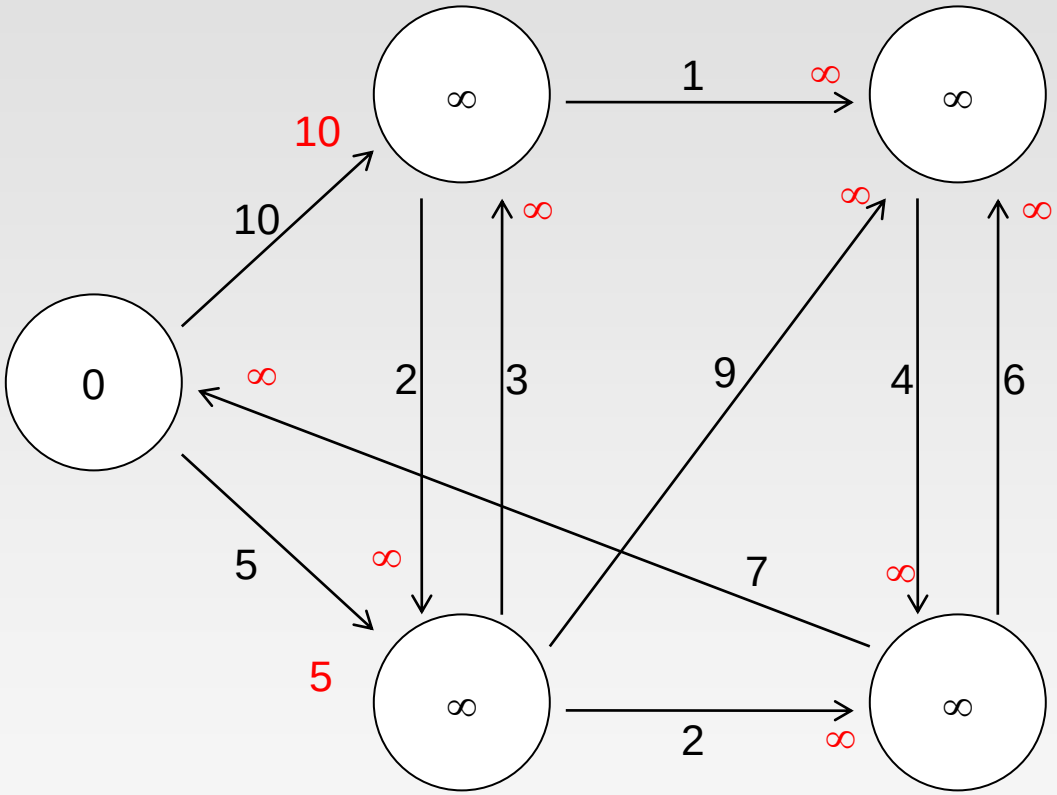


Finish!

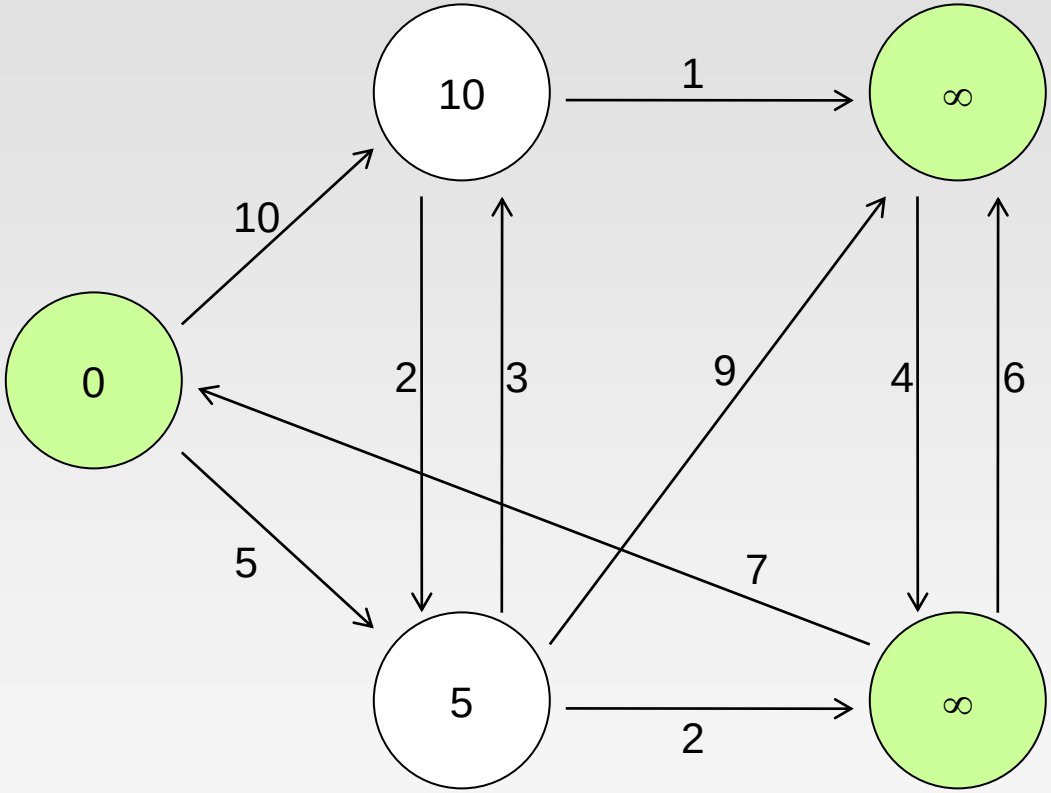
Example: SSSP – Parallel BFS in Pregel



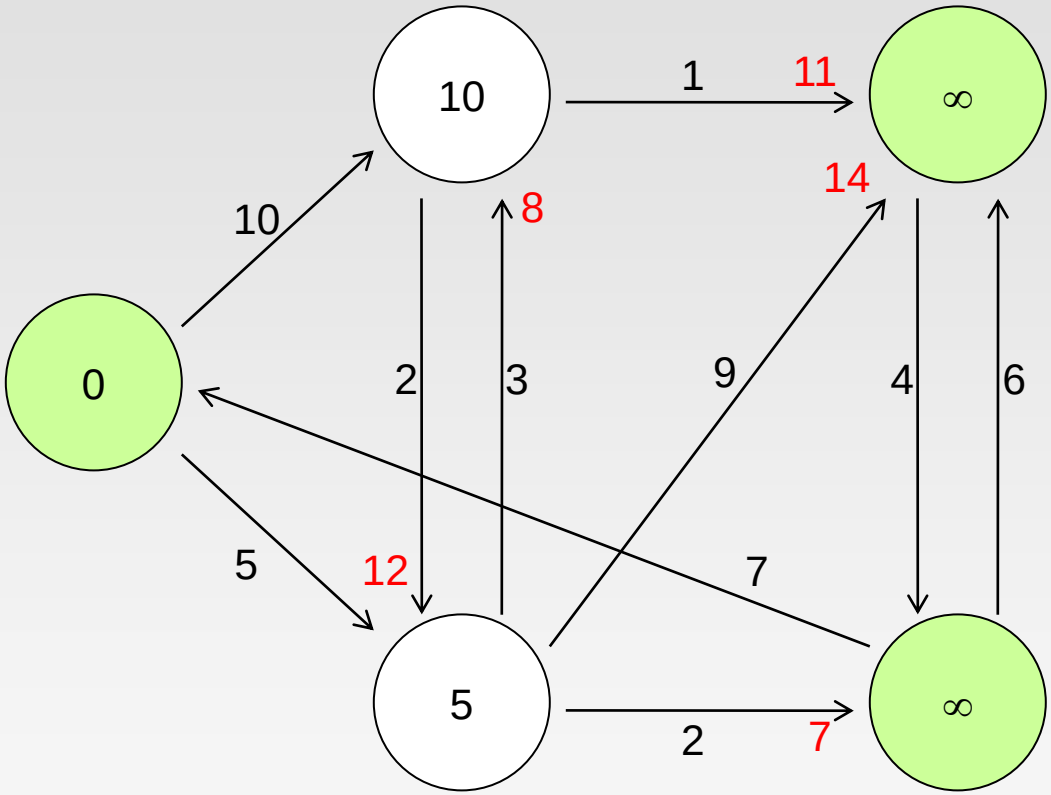
Example: SSSP – Parallel BFS in Pregel



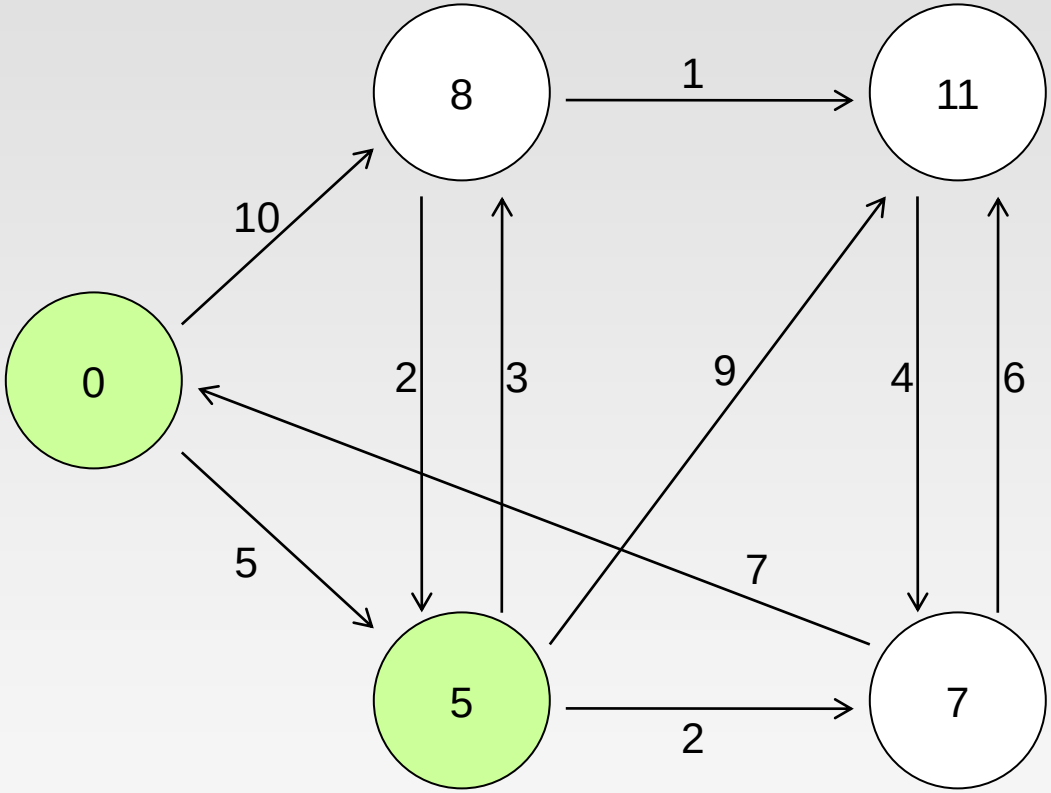
Example: SSSP – Parallel BFS in Pregel



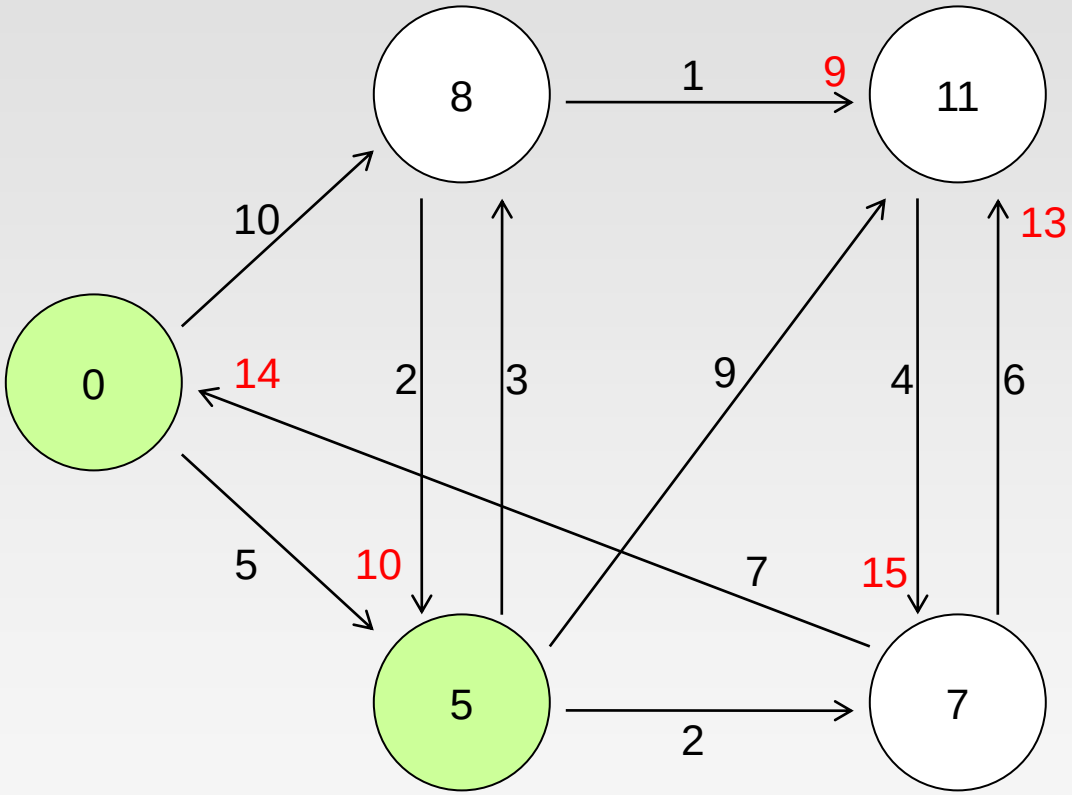
Example: SSSP – Parallel BFS in Pregel



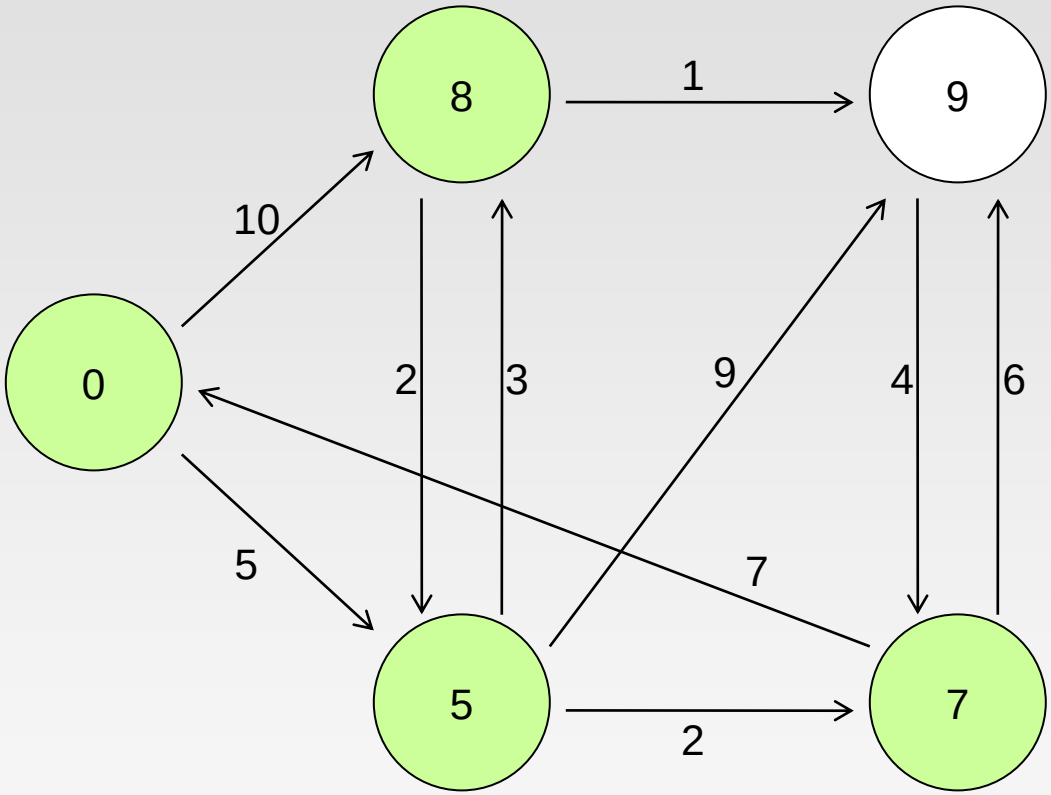
Example: SSSP – Parallel BFS in Pregel



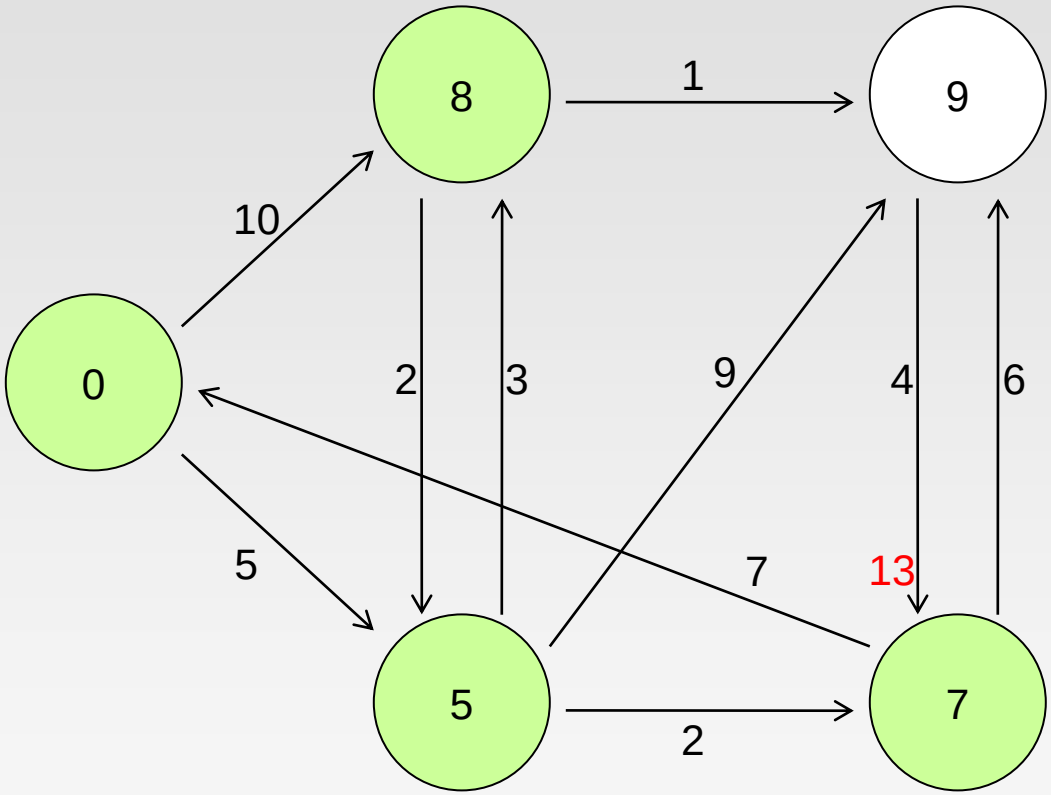
Example: SSSP – Parallel BFS in Pregel



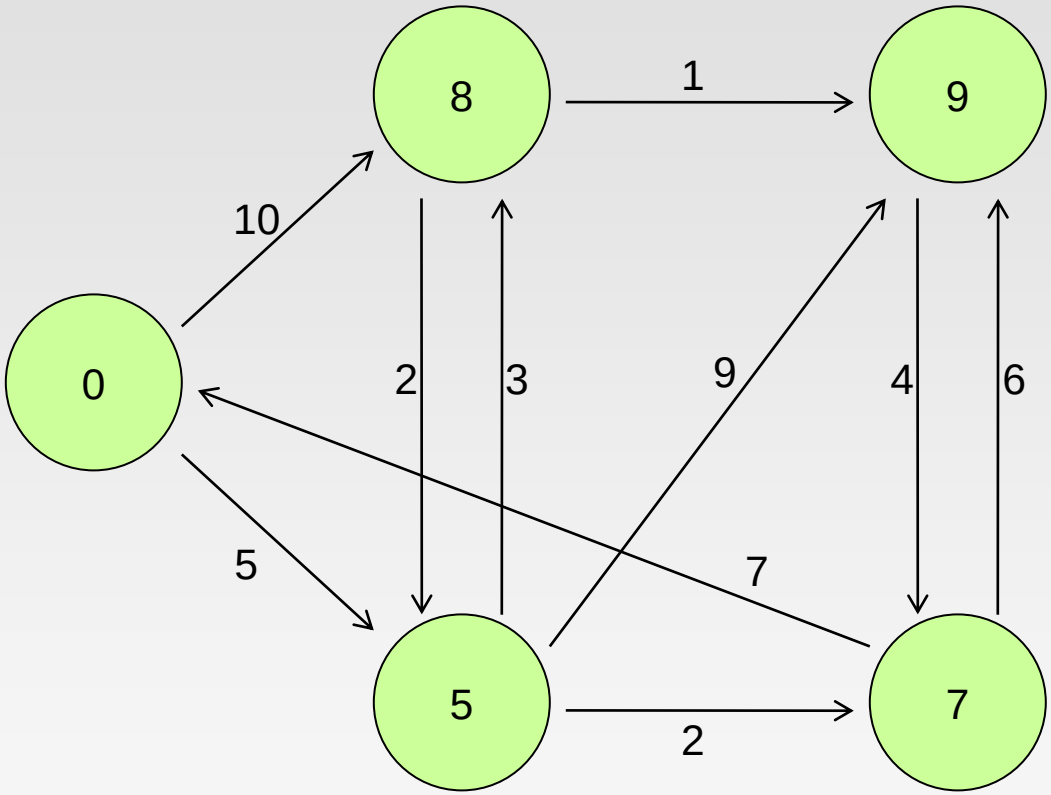
Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Example: SSSP – Parallel BFS in Pregel



Pregel Operator

```
def pregel[A]
      (initialMsg: A,
       maxIter: Int = Int.MaxValue,
       activeDir: EdgeDirection = EdgeDirection.Out)
      (vprog: (VertexId, VD, A) => VD,
       sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
       mergeMsg: (A, A) => A)
    : Graph[VD, ED] = {
              … …
     }
```

- Notice that Pregel takes two argument lists
  - The first argument list contains configuration parameters including the initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).
  - The second argument list contains the user defined functions for receiving messages (the vertex program vprog), computing messages (sendMsg), and combining messages mergeMsg.

Scala Currying

- Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

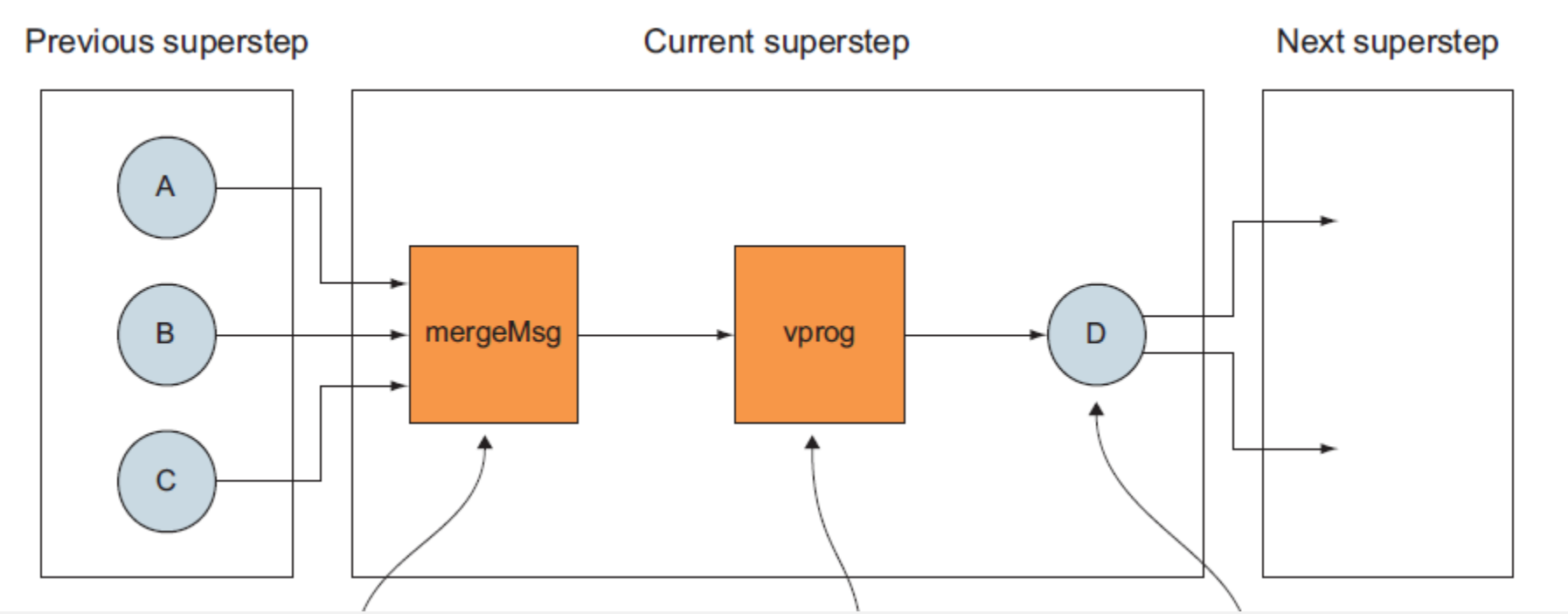
```
def modN(n: Int)(x: Int) = ((x % n) == 0)

val nums = List(1, 2, 3, 4, 5, 6, 7, 8)

nums.filter(modN(2))
```

- Results:
  - nums.filter(modN(2)) = nums.filter(x => modN(2)(x))
  - x is treated as the argument: List(2,4,6,8)

Pregel Operator

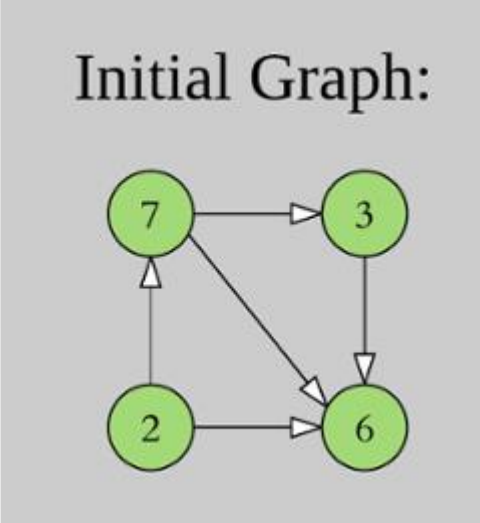


Messages delivered from vertices in the previous superstep are combined to a single message by a custom mergeMsg function

The custom vprog method decides how to update the vertex data based on the message received from mergeMsg

The custom sendMsg function decides which vertices will receive messages in the next superstep

Find the minimum value in a graph




```
val initialMsg = 9999
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) =
{
        if (message == initialMsg)     value
        else     (message min value._1, value._1)
}

def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]): Iterator[(VertexId,
Int)] = {
        val sourceVertex = triplet.srcAttr
        if (sourceVertex._1 == sourceVertex._2)     Iterator.empty
        else     Iterator((triplet.dstId, sourceVertex._1))
}

def mergeMsg(msg1: Int, msg2: Int): Int = msg1 min msg2

val minGraph = graph.pregel(initialMsg)(vprog,  sendMsg,  mergeMsg)
```

Single Source Shortest Path

- vprog:

```
(id, dist, newDist) => math.min(dist, newDist)
```

- sendMsg:

```
triplet => {
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {  Iterator.empty  }
  }
```

- mergeMsg:

```
(a, b) => math.min(a, b)
```

https://github.com/apache/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/lib/ShortestPaths.scala
or
https://spark.apache.org/docs/latest/graphx-programming-guide.html#pregel-api

- Full Pregel function call:

```
val initialGraph = graph.mapVertices((id, _) =>
    if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {  Iterator.empty  }
  },
  (a, b) => math.min(a, b) // Merge Message
)
```

Part 2: Spark GraphFrame

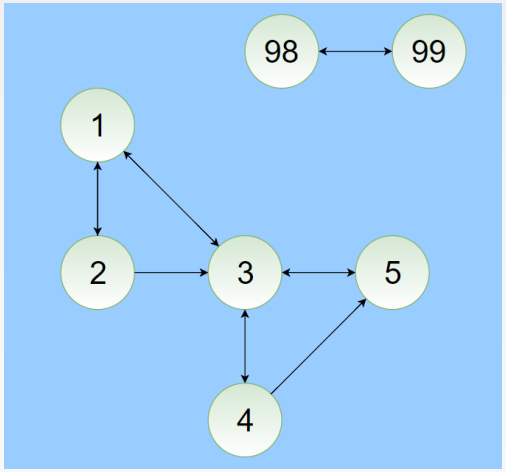
What are GraphFrames

- GraphFrames support general graph processing, similar to Apache Spark's GraphX library. However, GraphFrames are built on top of Spark DataFrames, resulting in some key advantages:
  - Python, Java & Scala APIs: GraphFrames provide uniform APIs for all 3 languages. For the first time, all algorithms in GraphX are available from Python & Java.
  - Powerful queries: GraphFrames allow users to phrase queries in the familiar, powerful APIs of Spark SQL and DataFrames.
  - Saving & loading graphs: GraphFrames fully support DataFrame data sources, allowing writing and reading graphs using many formats like Parquet, JSON, and CSV.

Creating a GraphFrame

```
>>> v = spark.createDataFrame([('1', 'Carter',
'Derrick', 50),
('2', 'May', 'Derrick', 26),
('3', 'Mills', 'Jeff', 80),
('4', 'Hood', 'Robert', 65),
('5', 'Banks', 'Mike', 93),
('98', 'Berg', 'Tim', 28),
('99', 'Page', 'Allan', 16)],
['id', 'name', 'firstname', 'age'])
```

```
e = spark.createDataFrame([('1', '2', 'friend'),
('2', '1', 'friend'),
('3', '1', 'friend'),
('1', '3', 'friend'),
('2', '3', 'follows'),
('3', '4', 'friend'),
('4', '3', 'friend'),
('5', '3', 'friend'),
('3', '5', 'friend'),
('4', '5', 'follows'),
('98', '99', 'friend'),
('99', '98', 'friend')],
['src', 'dst', 'type'])
```




```
>>> g = GraphFrame(v, e)
```

Some Basic Functions

```
>>> g.vertices.show()
+---+------+---------+---+
| id|  name|firstname|age|
+---+------+---------+---+
|  1|Carter|  Derrick| 50|
|  2|   May|  Derrick| 26|
|  3| Mills|     Jeff| 80|
|  4|  Hood|   Robert| 65|
|  5| Banks|     Mike| 93|
| 98|  Berg|      Tim| 28|
| 99|  Page|    Allan| 16|
+---+------+---------+---+
```

```
>>> g.edges.show()
 +---+---+-------+
|  1|  2| friend|
|  2|  1| friend|
|  3|  1| friend|
|  1|  3| friend|
|  2|  3|follows|
|  3|  4| friend|
|  4|  3| friend|
|  5|  3| friend|
|  3|  5| friend|
|  4|  5|follows|
| 98| 99| friend|
| 99| 98| friend|
+---+---+-------+
```

Some Basic Functions

```
>>> inDegreeDF.sort(['inDegree'], ascending=[0]).show() # Sort and show
---+--------+
| id|inDegree|
+---+--------+
|  3|       4|
|  5|       2|
|  1|       2|
|  4|       1|
| 99|       1|
| 98|       1|
|  2|       1|
+---+--------+
```

```
>>> outDegreeDF.sort(['outDegree'], ascending=[0]).show()
+---+---------+
| id|outDegree|
+---+---------+
|  3|        3|
|  4|        2|
|  1|        2|
|  2|        2|
| 99|        1|
|  5|        1|
| 98|        1|
+---+---------+
```

Shortest Path Computation

- shortestPaths(): Computes shortest paths from each vertex to the given set of landmark vertices, where landmarks are specified by vertex ID.

```
>>> results = g.shortestPaths(landmarks=["a", "d"])
>>> results.select("id", "distances").show()
+---+---------------+
| id|      distances|
+---+---------------+
|  b|             []|
|  e|[d -> 1, a -> 2]|
|  a|       [a -> 0]|
|  f|             []|
|  d|[d -> 0, a -> 1]|
|  c|             []|
+---+---------------+
```

Communications

- GraphFrames provides primitives for developing graph algorithms. The two key components are:
  - aggregateMessages: Send messages between vertices, and aggregate messages for each vertex. GraphFrames provides a native aggregateMessages method implemented using DataFrame operations. This may be used analogously to the GraphX API.
  - joins: Join message aggregates with the original graph. GraphFrames rely on DataFrame joins, which provide the full functionality of GraphX joins.

```
from graphframes.lib import AggregateMessages
as AM# For each user, sum the ages of the
adjacent users.
msgToSrc = AM.dst["age"]
msgToDst = AM.src["age"]
>>> agg.show()
>>> agg = g.aggregateMessages(
    sqlsum(AM.msg).alias("summedAges"),
    sendToSrc=msgToSrc,
    sendToDst=msgToDst)
```

```
+---+----------+
| id|summedAges|
+---+----------+
|  f|        62|
|  e|        65|
|  d|        66|
|  c|       108|
|  b|        94|
|  a|        65|
+---+----------+
```

References

- Spark GraphX guide: http://spark.apache.org/docs/latest/graphx-programming-guide.html
- Graph Analytics with Graphx. https://github.com/databricks/spark-training/blob/master/website/graph-analytics-with-graphx.md
- Spark GraphX in Action. Manning Publications
- GraphFrames: https://graphframes.github.io/graphframes/docs/_site/index.html

End of Chapter 5.2