

ST446 Distributed Computing for Big Data

Lecture 5

Graph data processing



Milan Vojnovic

<https://github.com/lse-st446/lectures2021>

Goals of this lecture

- Learn about main principles of graph data processing systems
- Learn about graph queries and graph algorithms
- Learn about Spark GraphX and GraphFrame API

Topics of this lecture

- Graph data processing: introduction
- Iterative and distributed graph computation, GraphFrames API
- Graph algorithms
- More on graph algorithms


Graph data processing: introduction


Graph data examples

- Many data can be naturally represented by a **graph**: nodes connected with edges
 - Web graph
 - Social networks, ex [Facebook](#), [LinkedIn](#)
 - Knowledge graphs, ex [Google](#), [Bing Satori](#), [Yago](#), [Thomson-Reuters](#)
 - Co-authorship graphs, ex [DBLP](#)
 - Road transportation systems
 - Communication network graphs
 - ...

Knowledge bases

- **Knowledge base (KB)**: a technology used to store complex facts about the world
- **Ex - the Google Knowledge Graph**: a knowledge base used by Google and its services to enhance its search engine's results
 - Information gathered from various sources
 - 70 billion facts (2016)
 - Used for deriving information presented to users in an infobox next to the search results
 - Used to answer direct spoken questions in Google Assistant and Google Home voice queries
- **Other examples of knowledge graphs**:
 - Bing Satori, Yago, DBpedia



Edsger W. Dijkstra 

Dutch essayist

Edsger Wybe Dijkstra was a Dutch systems scientist, programmer, software engineer, science essayist, and pioneer in computing science. A theoretical physicist by training, he worked as a programmer at the Mathematisch Centrum from 1952 to 1962. [Wikipedia](#)

Born: 11 May 1930, [Rotterdam, Netherlands](#)

Died: 6 August 2002, [Nuenen, Netherlands](#)

Spouse: [Maria C. Dijkstra Debets](#) (m. 1957)

Children: [Rutger M. Dijkstra](#), [Marcus J. Dijkstra](#), [Femke E. Dijkstra](#)





Quotes [View 4+ more](#)

Computer science is no more about computers than astronomy is about telescopes.





The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.

Program testing can be used to show the presence of bugs, but never to show their absence!

Books [View 2+ more](#)

			
A Discipline of Progr...	Selected Writings on Com...	A Method of Program...	Predicate Calculus and Pro...
1976	1982	1988	1990

People also search for [View 15+ more](#)

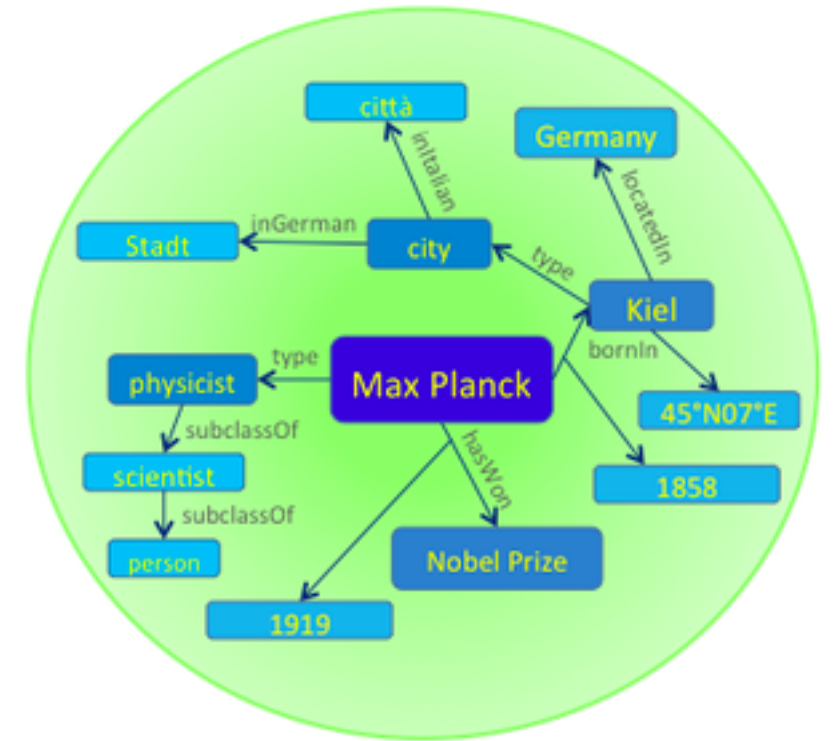
			
Tony Hoare	Niklaus Wirth	Donald Knuth	John McCarthy

[Feedback](#)

YAGO (Yet Another Great Ontology)

- An open source knowledge base developed at the Max Planck Institute for Computer Science in Saarbrücken
- 10+M entities and 120M facts (2019)
- Information extracted from Wikipedia, WordNet, and GeoNames
- Provided in Turtle (Terse RDF Triple Language): a syntax and file format for expressing data in RDF data model
- Provided also in tsv

To probe further: see [here](#) and GitHub [repo](#)



[WordNet](#): a lexical database for English

[Geonames](#): geographical database covering all countries containing over 11 M placenames

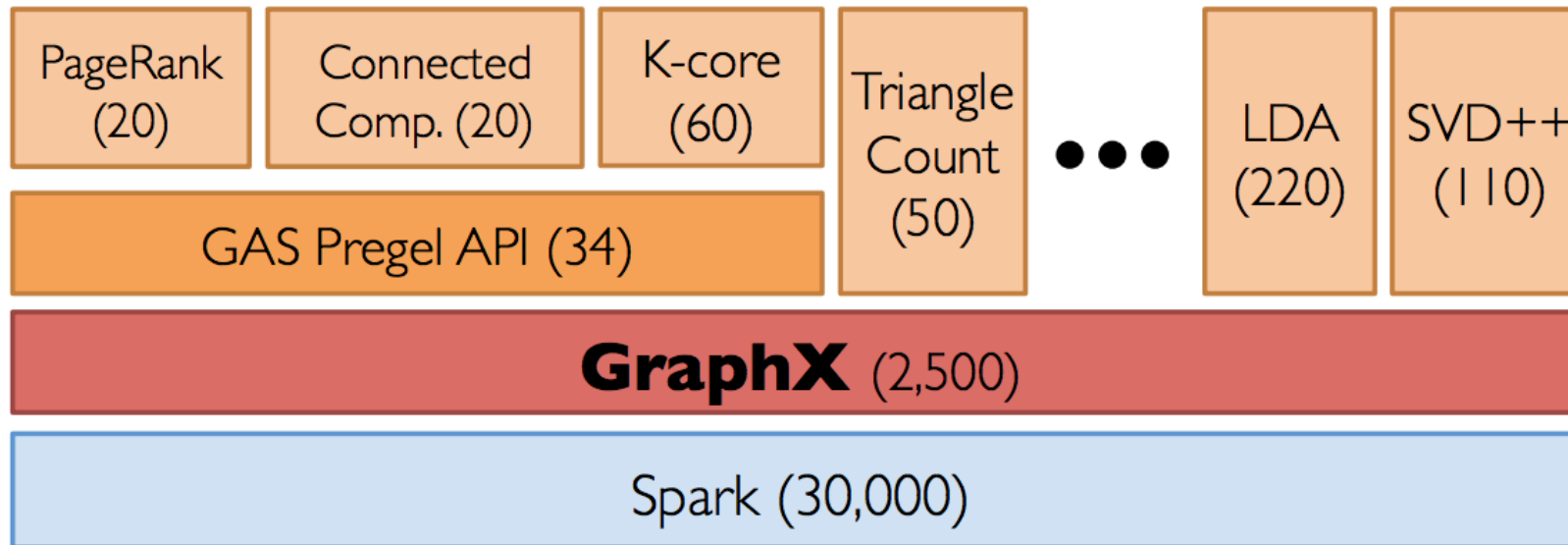
Types of graph data processing

- Queries on graphs:
 - Query languages for *graph pattern matching queries, aggregate statistics*
 - Declarative programming
 - Ex. [Gremlin](#), [SPARQL](#), [openCypher](#), [GraphQL](#), Spark motif queries
- Computation of graph properties:
 - Degree, in-degree, out-degree
 - Connected components
 - Node centrality measures, ex PageRank, Betweenness, ...
 - Triangle count
 - Singular value decomposition of graph adjacency matrix
 - Ex. [Apache Giraph](#), [Apache Spark GraphX](#), [Turi](#) (formerly Dato, GraphLab)

Spark graph data processing

- **GraphX**: Apache Spark's API for graphs and graph-parallel computation
 - Supports Scala and RDD API
 - Architecture described in an OSDI 2014 paper
- **GraphFrames**: a Spark package
 - Based on Spark DataFrames
 - Supports Scala, Java and Python API
 - Some GraphX features not yet implemented (ex. graph partitioning)

GraphX software stack



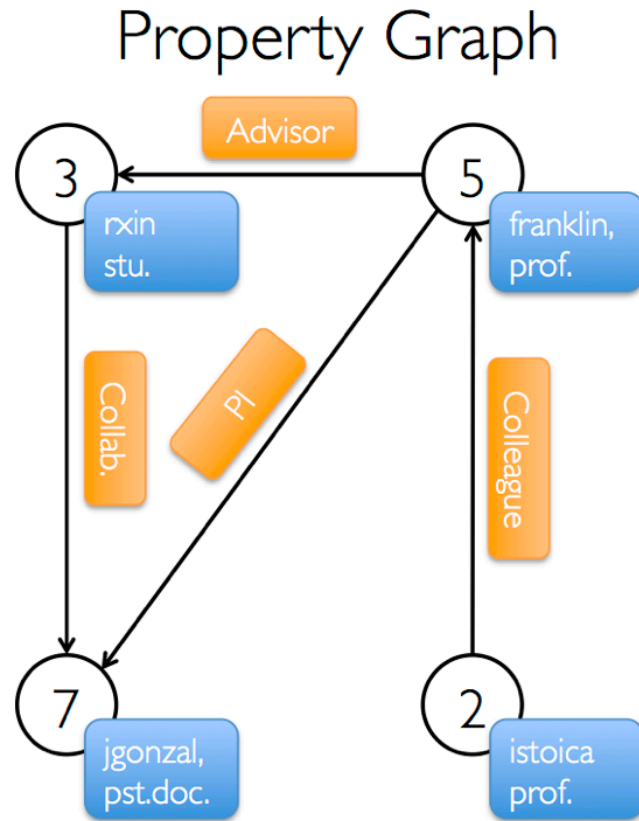
GraphX

- An RDD extension with a graph abstraction
 - A **directed multigraph** with **properties** attached to **nodes** and **edges**
- Exposes a set of **basic operators**
 - Subgraph
 - JoinVertices
 - aggregateMessage
- Implements a variant of Pregel API
- Includes graph algorithms and builders to simplify graph analytics tasks

Property graph data model

- **Property graph**: a directed multigraph with user-defined objects *attached to nodes and edges*
 - Each node is keyed by a unique 64-bit identifier (VertexId)
 - Multigraph: there can be multiple edges between pairs of nodes
 - Suitable for describing multiple types of relationships (ex friend, lives in, born in, ...)
- **Property graph RDD implementation**: **immutable**, **distributed**, and **fault-tolerant**
 - Changes to the values or structure require producing a new graph (reusing substantial part of the original graph)
- **Distributed computation**: graph partitioned over executors
 - Partitioning using *a vertex partitioning algorithm*
 - Each partition can be recreated on a different machine in the event of a failure

Property graph example



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

Property graph implementation

- Property graph defined by *a pair* of typed collections (RDDs) encoding the properties of vertices and edges
- Several different ways to construct a property graph
 - from raw files
 - from RDDs
 - from graph generators (builders)

Iterative and distributed graph computation, GraphFrames API

Pregel computation model

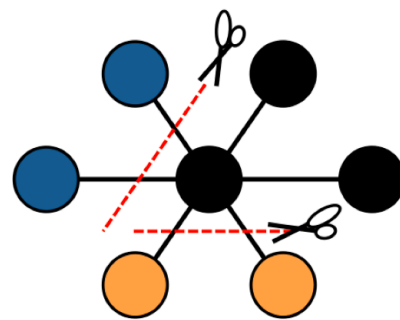
- **Pregel (reminder)**: a computational model for processing large-scale graphs
 - Bulk-synchronous parallel message abstraction constrained to the graph topology
- Some features:
 - Programs expressed as **a sequence of iterations**
 - **Iterative computation**: in each iteration a vertex (a) receives messages from its neighbor vertices sent in the previous iteration, (b) sends messages to its neighbor vertices and (c) modifies its own state and that of its outgoing edges
 - **Vertex-centric computation**: expressive and easy to program "think like a vertex"
 - Computation model flexible enough to express a broad set of algorithms
 - Designed to be efficient, scalable and fault-tolerant for implementation in clusters of computers
 - Synchronicity of iterations makes reasoning about programs easier
 - Distributed computation aspects hidden behind an abstract API

Pregel computation model (cont'd)

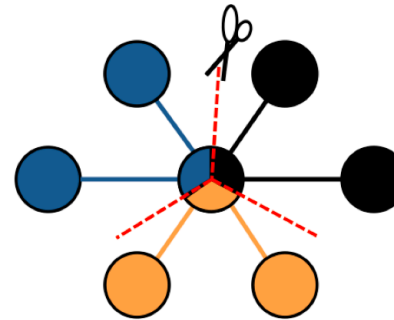
- **Neighborhood aggregation**: aggregation of information over neighborhood of each vertex
 - Ex number of followers each user has
 - Ex average age of followers of users
- Iterative graph algorithms repeatedly aggregate properties of neighborhood vertices
 - Ex PageRank, shortest path to source vertex, the smallest reachable vertex ID
 - Core aggregation operation: `aggregateMessages`
 - A user-defined function `sendMsg` applied to each edge (like Map)
 - A user-defined function `mergeMsg` applied to aggregate messages at their destination vertex (like Reduce)
 - Returns `VertexRDD[Msg]`

Graph partitioning

- **Graph partitioning**: assigning nodes or edges to different compute nodes for graph parallel-computation
- **Vertex partitioning (edge cuts)**: partition the set of vertices in given number of disjoint components
- **Edge partitioning (vertex cuts)**: partition the set of edges in given number of disjoint components



Edge Cut



Vertex Cut

Graph partitioning objectives

- **Bicriteria graph partitioning objective:**
 - Balance the processing load over compute nodes
 - Minimize the communication between compute nodes
- **Balanced graph partitioning:** combinatorial optimization problem formulation
 - Minimize the cut cost subject to balancing constraints
 - NP hard
 - Approximation algorithms exist but are impractical for scalable implementation
- **Examples of graph partitioning heuristics:**
 - **Greedy assignment:** minimizes marginal cut cost subject to balancing constraints
 - **Streaming algorithms:** vertices or edges assigned in one pass in some order

GraphFrames

- GraphFrame defined by two DataFrames
 - **Vertex DataFrame**: contains a special column named `id` which specifies unique IDs for each vertex in the graph
 - **Edge DataFrame**: contains two special columns `src` (source vertex ID of edge) and `dst` (destination vertex ID of edge)
- Both Vertex and Edge DataFrame can have **arbitrary other columns**
 - These columns allow to represent vertex and edge attributes (properties)
- A GraphFrame can be constructed from a single DataFrame containing edge information
 - The vertices are inferred from the sources and destinations of the edges

GraphFrame example

```
# Vertex DataFrame
```

```
v = sqlContext.createDataFrame([("a", "Alice", 34), ("b", "Bob", 36), ("c", "Charlie", 30), ("d", "David", 29), ("e", "Esther", 32), ("f", "Fanny", 36), ("g", "Gabby", 60)], ["id", "name", "age"])
```

```
# Edge DataFrame
```

```
e = sqlContext.createDataFrame([("a", "b", "friend"), ("b", "c", "follow"), ("c", "b", "follow"), ("f", "c", "follow"), ("e", "f", "follow"), ("e", "d", "friend"), ("d", "a", "friend"), ("a", "e", "friend")], ["src", "dst", "relationship"])
```

```
# Create a GraphFrame
```

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

Examples of basic graph queries

```
# Get a DataFrame with columns "id" and "inDegree" (in-degree)  
vertexInDegrees = g.inDegrees
```

```
# Find the youngest user's age in the graph  
# This queries the vertex DataFrame  
g.vertices.groupBy().min("age").show()
```

```
# Count the number of "follows" in the graph  
# This queries the edge DataFrame  
numFollows = g.edges.filter("relationship = 'follow']").count()
```

Graph algorithms

Examples of graph computation tasks

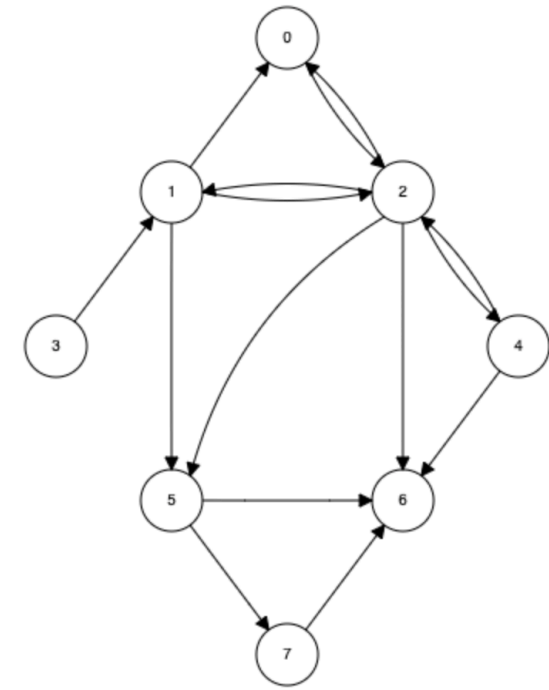
- Breadth-first search
- Motif queries (subgraph matching)
- Connected components
- Strongly connected components
- Label propagation
- PageRank (and personalized PageRank)
- Shortest paths
- Triangle count
- ...

Examples of graph computation tasks

- Breadth-first search
- Motif queries (subgraph matching)
- Connected components
- Strongly connected components
- Label propagation
- PageRank (and personalized PageRank)
- Shortest paths
- Triangle count
- ...

Graph traversal queries

- **Depth-first search (DFS)**: traverse or search a graph data starting from a root node and explore as far as possible along each branch
- **Breadth-first search (BFS)**: traverse or search a graph data starting from a root node and explore all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level



Ex start vertex **2**

DFS:

2->0

2->1

2->1->5

2->1->5->6

2->1->5->7

2->4

BFS:

2->0

2->1

2->4

2->5

2->6

2->5->7

Animations:

DFS: <https://www.cs.usfca.edu/~galles/visualization/DFS.html>

BFS: <https://www.cs.usfca.edu/~galles/visualization/BFS.html>

Breadth-first search example

```
# Search from "Esther" for users of age < 32  
paths = g.bfs("name = 'Esther'", "age < 32")
```

Depth-limited BFS and edge filters:

```
filteredPaths = g.bfs(  
    fromExpr = "name = 'Esther'",  
    toExpr = "age < 32",  
    edgeFilter = "relationship != 'friend'",  
    maxPathLength = 3)
```

Graph motif queries

- Graph motif query: find a subgraph satisfying specified conditions
- Ex: find reciprocal relationships

```
motifs = g.find("(a)-[e]->(b): (b)-[e2]->(a)")
```

- Ex: find friends-of-friends

```
p = g.find("(a)-[e]->(b): (b)-[e2]->(c)").filter("e.relationship =  
'friend'" and "e2.relationship = 'friend'")
```

```
fof = p.select("e.src", "e2.dst")
```

Connected components

- **Connected components problem**: find connected components of an input graph
- **Connected component**: a maximal subgraph such that there exists a path between each pair of nodes in this subgraph
- **Input**: a graph
- **Output**: a graph with vertex attributes containing the smallest vertex id in each connected component

```
result = g.connectedComponents()  
result.select("id", "component").orderBy("component").show()
```

EdgeTriplet object

```
/**
 * An edge triplet represents an edge along with the vertex attributes of its neighboring vertices.
 *
 * @tparam VD the type of the vertex attribute.
 * @tparam ED the type of the edge attribute
 */
class EdgeTriplet[VD, ED] extends Edge[ED] {
  /**
   * The source vertex attribute
   */
  var srcAttr: VD = _ // nullValue[VD]

  /**
   * The destination vertex attribute
   */
  var dstAttr: VD = _ // nullValue[VD]

  /**
   * Set the edge properties of this triplet.
   */

  protected[spark] def set(other: Edge[ED]): EdgeTriplet[VD, ED] =
  {
    srcId = other.srcId
    dstId = other.dstId
    attr = other.attr
    this
  }
}
```

Source: [EdgeTriplet.scala](#)

Connected components algorithm

- **Key idea:** propagation of minimum vertex id using Pregel
 - Output: each vertex assigned minimum vertex id within the connected component in which this vertex resides

```
object ConnectedComponents {  
  
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD, ED],  
                                       maxIterations: Int): Graph[VertexId, ED] = {  
    require(maxIterations > 0, s"Maximum of iterations must be greater than 0," +  
      s" but got ${maxIterations}")  
  
    val ccGraph = graph.mapVertices { case (vid, _) => vid }  
  
    def sendMessage(edge: EdgeTriplet[VertexId, ED]): Iterator[(VertexId, VertexId)] =  
    {  
      if (edge.srcAttr < edge.dstAttr) {  
        Iterator((edge.dstId, edge.srcAttr))  
      } else if (edge.srcAttr > edge.dstAttr) {  
        Iterator((edge.srcId, edge.dstAttr))  
      } else {  
        Iterator.empty  
      }  
    }  
  }  
}
```

Connected components algorithm (cont'd)

```
val initialMessage = Long.MaxValue
val pregelGraph = Pregel(ccGraph, initialMessage,
    maxIterations, EdgeDirection.Either)(
    vprog = (id, attr, msg) => math.min(attr, msg),
    sendMsg = sendMessage,
    mergeMsg = (a, b) => math.min(a, b))
ccGraph.unpersist()
pregelGraph
} // end of connectedComponents
```


Strongly connected components

- **Strongly connected graph**: a directed graph is **strongly connected** if every vertex is reachable from every other vertex, i.e. there is a path in each direction between each pair of vertices
- **Strongly connected components problem**: compute the strongly connected component (SSC) of each vertex and return a graph with each vertex assigned to the SCC containing that vertex
- Similar algorithm as for connected components

```
res = g.stronglyConnectedComponents(maxIter = 10)
```

Source: [StronglyConnectedComponents.scala](#)

Graph algorithms cont'd

Examples of graph computation tasks

- Breadth-first search
- Motif queries (subgraph matching)
- Connected components
- Strongly connected components
- Label propagation
- PageRank (and personalized PageRank)
- Shortest paths
- Triangle count
- ...

Label propagation

- **Label propagation**: an iterative algorithm for graph clustering
 - Introduced by Raghavan, Phys. Rev. E, 76, 2007
- Algorithm key ideas:
 - Initialization: each node assigned a unique cluster id
 - Iterative step: nodes send messages containing information about their cluster ids to their neighbors, nodes update their state to the mode cluster id value of the incoming messages
- The algorithm is not guaranteed to converge, and it may converge to a trivial solution
 - Ex. All nodes mapping to the same cluster id

```
result = g.labelPropagation(maxIter=5)
```

PageRank

- PageRank: a network node centrality measure
 - Introduced by Page, Brin, Rajeev and Winograd, [The PageRank Citation Ranking: Bringing Order to the Web](#), unpublished report, Stanford University, 1998
- Input: (directed) graph $G = (V, E)$
- PageRank definition: vector r with entries $r(u)$, for $u \in V$, defined as the solution of

$$r(u) = (1 - \alpha) \sum_{v \in V: (v, u) \in E} \frac{1}{|N(v)|} r(v) + \alpha \frac{1}{|V|}$$

where $N(v)$ is the set of nodes node v has links to $N(v) = \{u \in V: (v, u) \in E\}$ and $0 < \alpha \leq 1$ is a parameter (teleportation parameter; reset probability parameter)

- Different interpretations:
 - Principal eigenvector (associated with the largest eigenvalue, of value 1) of a graph adjacency matrix
 - Stationary distribution of a Markov chain defined by a random walk on the graph

PageRank: user-defined number of iterations

```
# Run PageRank for a fixed number of iterations  
res = g.pageRank(resetProbability=0.15, maxIter=10)
```

PageRank: prescribed error tolerance

- Absolute error tolerance – run until for every vertex the absolute value of the difference of PageRank values of this vertex at two successive iterations is smaller than or equal to the user-defined tolerance parameter ϵ

```
# Run PageRank until convergence to tolerance "tol"  
result = g.pageRank(resetProbability=0.15, tol=0.01)
```

```
# Display resulting pageranks and final edge weights  
result.vertices.select("id", "pagerank").show()  
result.edges.select("src", "dst", "weight").show()
```

Personalized PageRank

- Resets to a specified source vertex: personalized in the sense of biasing random walks to return to the specified source vertex at the “teleportation” events

```
# Run PageRank personalized for vertex "a"  
res = g.pageRank(resetProbability=0.15, maxIter=10, sourceId="a")
```


Personalized PageRank (cont'd)

```
var oldPR = Array.fill(n)( 1.0 )
val PR = (0 until n).map(i => if sourceIds.contains(i) alpha else 0.0)
for( iter <- 0 until maxIter ) {
  swap(oldPR, PR)
  for( i <- 0 until n ) {
    PR[i] = (1 - alpha) * inNbrs[i].map(j => oldPR[j] / outDeg[j]).sum
    if (sourceIds.contains(i)) PR[i] += alpha
  }
}
```

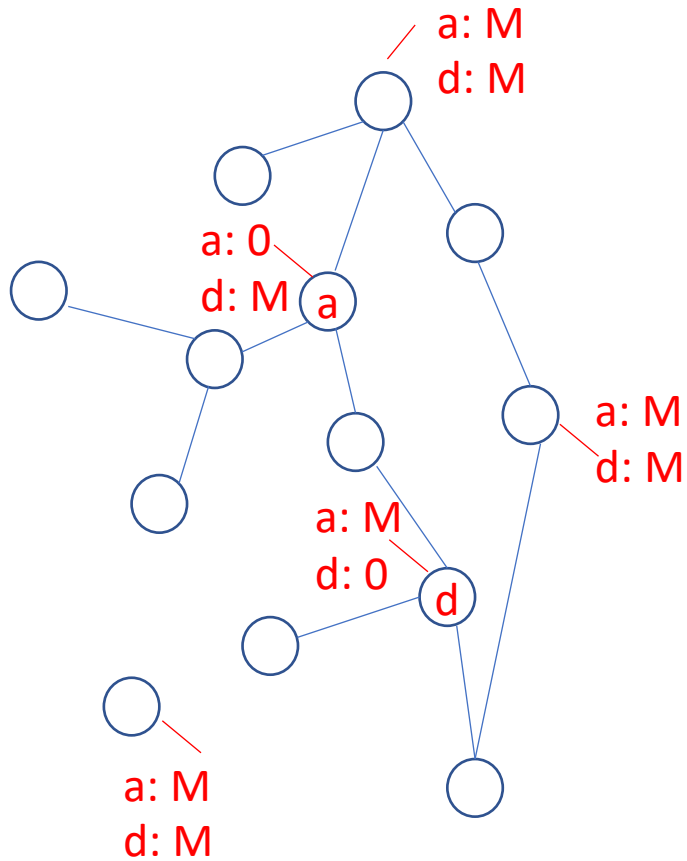
Shortest paths

- **Computation problem**: compute shortest paths to the given set of (landmark) vertices
- **Output**: a graph where each vertex attribute is a map containing the shortest-path distance to each reachable landmark vertex

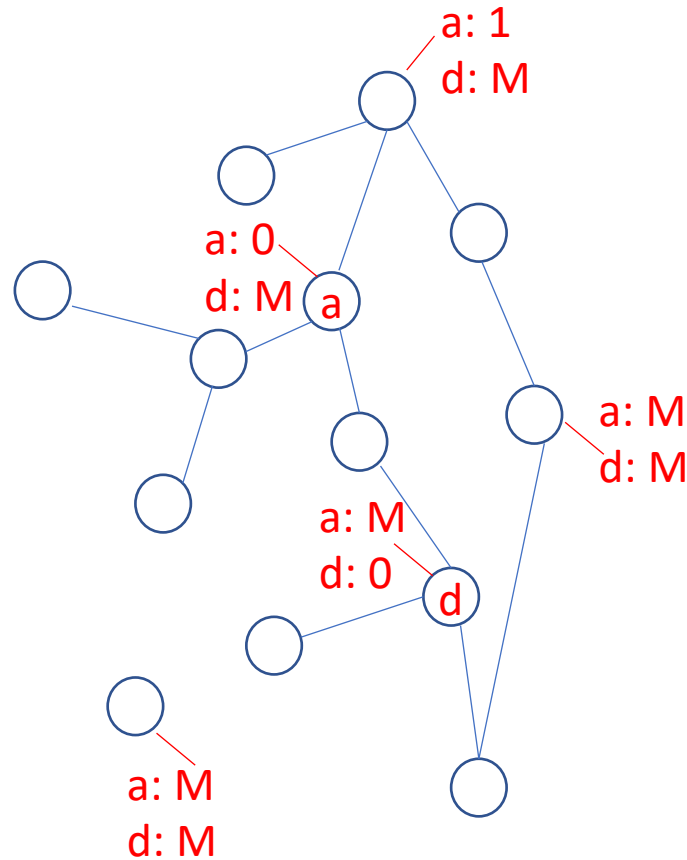
```
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends() # Get example graph

results = g.shortestPaths(landmarks=["a", "d"])
results.select("id", "distances").show()
```

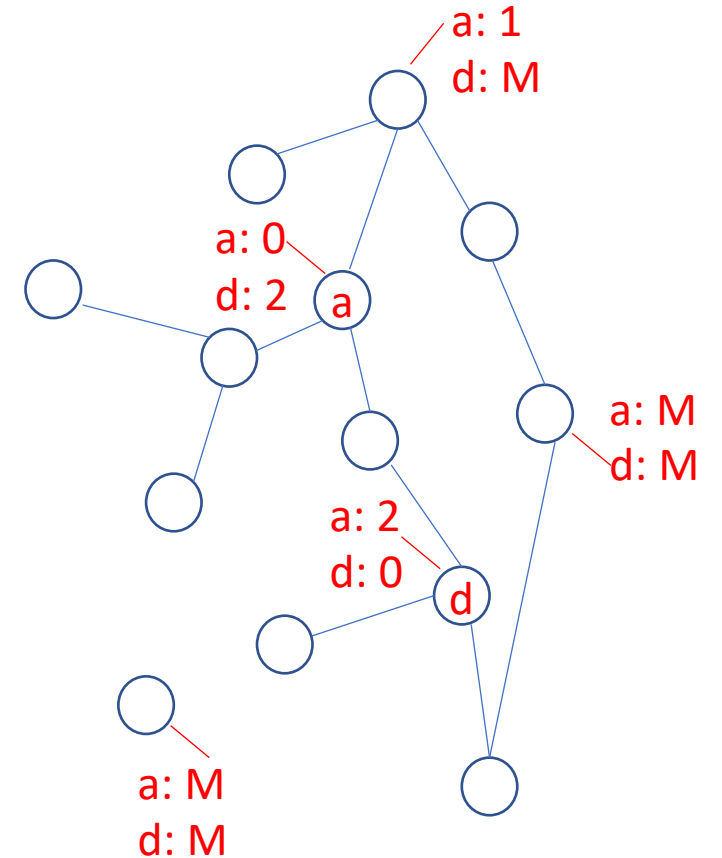
Shortest paths (cont'd)



initial state



super step 1

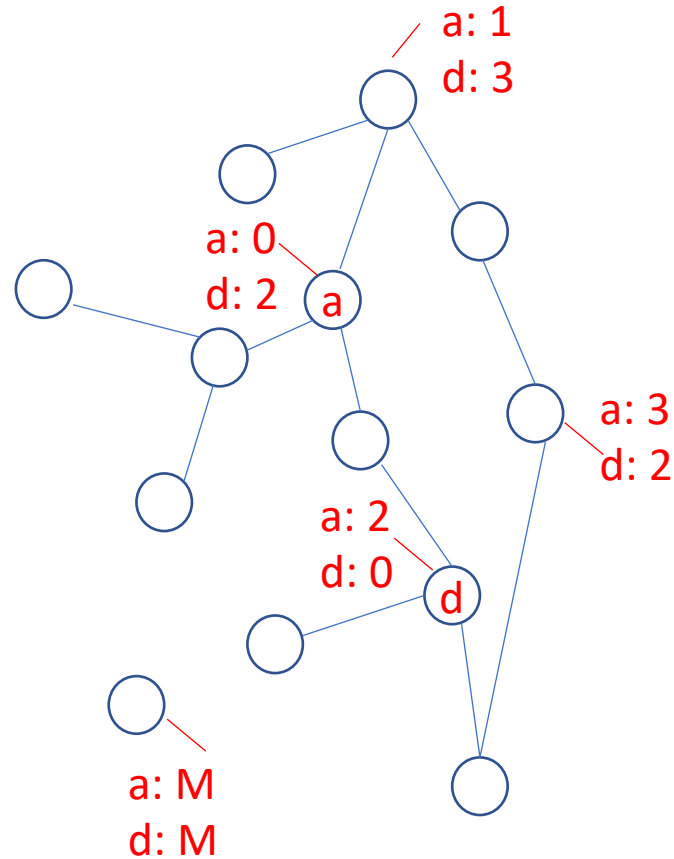


super step 2

M := "infinite" value (in practice, some maximum value, ex. maximum value of an integer type)

Note: we show (a: val) and (d: val) maps only for some nodes; every node has these maps

Shortest paths (cont'd)



final state

M := "infinite" value (in practice, some maximum value, ex. maximum value of an integer type)
Note: we show (a: val) and (d: val) maps only for some nodes; every node has these maps

Shortest path algorithm

```
object ShortestPaths extends Serializable {  
  /** Stores a map from the vertex id of a landmark to the distance to that landmark.  
  */  
  type SPMAP = Map[VertexId, Int]  
  
  private def makeMap(x: (VertexId, Int)*) = Map(x: _*)  
  
  private def incrementMap(spmap: SPMAP): SPMAP = spmap.map { case (v, d) => v -> (d +  
1) }  
  
  private def addMaps(spmap1: SPMAP, spmap2: SPMAP): SPMAP = {  
    (spmap1.keySet ++ spmap2.keySet).map {  
      k => k -> math.min(spmap1.getOrElse(k, Int.MaxValue), spmap2.getOrElse(k, Int.Max  
Value))  
    }(collection.breakOut)  
  }
```

Shortest path algorithm (cont'd)

```
def run[VD, ED: ClassTag](graph: Graph[VD, ED], landmarks: Seq[VertexId]): Graph[SPMap, ED] = {  
  val spGraph = graph.mapVertices { (vid, attr) =>  
    if (landmarks.contains(vid)) makeMap(vid -> 0) else makeMap()  
  }  
  
  val initialMessage = makeMap()  
  
  def vertexProgram(id: VertexId, attr: SPMap, msg: SPMap): SPMap = {  
    addMaps(attr, msg)  
  }  
  
  def sendMessage(edge: EdgeTriplet[SPMap, _]): Iterator[(VertexId, SPMap)] = {  
    val newAttr = incrementMap(edge.dstAttr)  
    if (edge.srcAttr != addMaps(newAttr, edge.srcAttr)) Iterator((edge.srcId, newAttr))  
    else Iterator.empty  
  }  
  
  Pregel(spGraph, initialMessage)(vertexProgram, sendMessage, addMaps)  
}
```

Triangle count

- **Computation problem:** compute the number of triangles passing through each vertex
 - Triangle: a set of edges (a, b) , (b, c) , (c, a)
 - Often used in analysis of social networks
 - A measure of the degree to which nodes in a graph *cluster* together

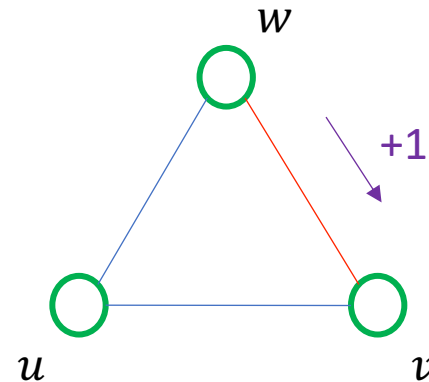
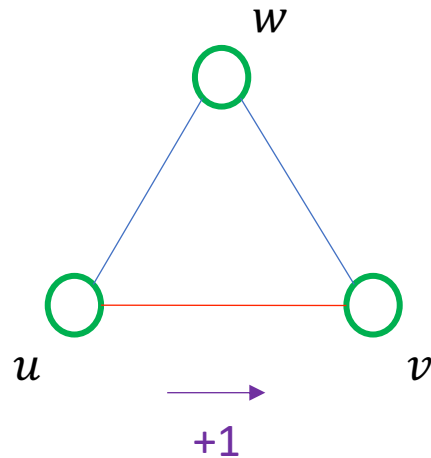
```
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends() # Get example graph

results = g.triangleCount()
results.select("id", "count").show()
```

- To probe further: local and global clustering coefficients see [here](#)

Triangle count algorithm

- Algorithm steps:
 - Compute the set of neighbors for each vertex
 - For each edge (u, v) compute the intersection of the neighborhood sets of vertices u and v and send the count (cardinality of the intersection set) to both u and v
 - Compute the sum at each vertex and divide by 2 since each triangle is counted twice



References

- Apache Spark: [GraphX programming guide](#)
- Databricks: [GraphX and GraphFrames](#)
- GraphFrames:
 - Github: <https://github.com/graphframes/graphframes>
 - User Guide <http://graphframes.github.io/user-guide.html>
 - User Guide Python <https://docs.databricks.com/spark/latest/graph-analysis/graphframes/user-guide-python.html>
 - Python API <https://graphframes.github.io/api/python/index.html>
 - Python code examples <https://github.com/mapr-demos/spark-graphframes>

References (cont'd)

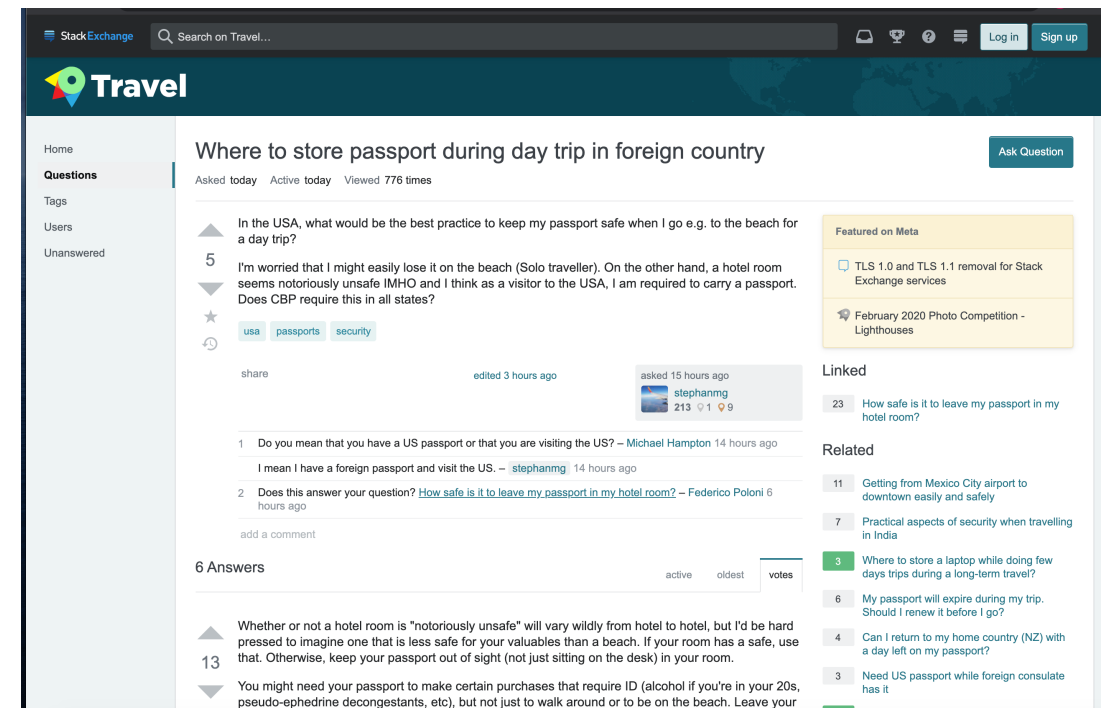
- Gonzales et al, [GraphX: Graph Processing in a Distributed Dataflow Framework](#), OSDI 2014
- Gonzales et al, [PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs](#), OSDI 2012
- Malewicz et al, [Pregel: A System for Large-Scale Graph Processing](#), SIGMOD 2010
- Ching et al, [One Trillion Edges: Graph Processing at Facebook-Scale](#), VLDB 2015
- A new computation paradigm: Fan et al, [Parallelizing Sequential Graph Computations](#), SIGMOD 2017
- Kabiljo et al, [A comparison of state-of-the-art graph processing systems](#), post, Facebook core data, data infrastructure, 2016

References (cont'd)

- Cypher
 - [OpenCypher](#): open query language for property graph databases
 - Neo4j developer guides, [Cypher Basics](#)
 - Neo4j developer guides, [Cypher Query Language](#)
- [Gremlin](#): an open source graph traversal language
- [GraphQL](#): an open source data query and manipulation language for APIs
- [Sparql](#) query language

Seminar class 5: Graph data analysis using Spark

- Analysis of StackExchange data using Spark GraphFrames API
 - Loading data
 - Basic queries
 - PageRank
 - Motif queries
 - Connected components



<https://github.com/lse-st446/lectures2021/blob/master/Week05/class/README.md>