SparklingGraph Documentation

Release 0.0.6

Roman Bartusiak

May 13, 2017

Contents

For bigger insight please refer to a API documentation in ScalaDocs.

CHAPTER 1

How To

Release

Publish process is based on sbt-sonatype plugin

Export credentials for sonatype repository:

```
export SONATYPE_USERNAME=???
export SONATYPE_PASSWORD=???
```

To publish signed artifacts to sonatype repository use

```
sbt 'release cross'
```

After that close staging repository and release to central using

```
sbt sonatypeRelease
```

CHAPTER 2

Loading graph data

Library support loading graphs from multiple file formats. Nevertheless, we will be implementing more of them in next releases.

Graph loading API

Main graph loading object is a LoadGraph

It takes implementations of a GraphLoader and lets you easily configure loading process. Parameters (Parameter) for configuration are set using `using(parameter: Parameter)` method. Parameters are specific for each GraphLoader

Loading from CSV

To load graph from CSV file you must use CSV implementation of GraphLoader trait:

```
import ml.sparkling.graph.api.loaders.GraphLoading.LoadGraph
import ml.sparkling.graph.loaders.csv.GraphFromCsv.CSV
import org.apache.spark.SparkContext

implicit val ctx:SparkContext=???
// initialize your SparkContext as implicit value so it will be passed automatically␣
↪to graph loading API

val filePath="your_graph_path.csv"

val graph=LoadGraph.from(CSV(filePath)).load()
```

That is simplest way of loading standard CSV file:

```
"vertex1","vertex2"
"<numerical_id_of_vertex_1>","<numerical_id_of_vertex_2>"
```

In order to change file format you can use parameters like:

```
import ml.sparkling.graph.loaders.csv.GraphFromCsv.LoaderParameters.{Delimiter,
↪Quotation}
import ml.sparkling.graph.api.loaders.GraphLoading.LoadGraph
import ml.sparkling.graph.loaders.csv.GraphFromCsv.CSV
import org.apache.spark.SparkContext

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value so it will be passed automatically␣
↪to graph loading API

val filePath="your_graph_path.csv"
val graph=LoadGraph.from(CSV(filePath)).using(Delimiter(";")).using(Quotation("'")).
↪load()
```

Presented snipet will load graph from file with format:

```
'vertex1';'vertex2'
'<numerical_id_of_vertex_1>';'<numerical_id_of_vertex_2>'
```

Loading graphs with vertex identifiers that are not numerical

Because in some cases vertices identifiers can be not numerical (username as string). You can load this kind of graph specifying that Indexing is required:

```
import ml.sparkling.graph.api.loaders.GraphLoading.LoadGraph
import ml.sparkling.graph.loaders.csv.GraphFromCsv.CSV
import ml.sparkling.graph.loaders.csv.GraphFromCsv.LoaderParameters.Indexing
import org.apache.spark.SparkContext

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value so it will be passed automatically␣
↪to graph loading API

val filePath="your_graph_path.csv"

val graph=LoadGraph.from(CSV(filePath)).using(Indexing).load()
```

That approach gives you ability to load graphs from CSV files with any structure and vertex identifiers of any type. For example:

```
"vertex1","vertex2"
"centralized","computation"
"is","lame"
```

Full list of CSV loading parameters is available in here

Loading from GraphML

To load graph from GraphML XML file you must use GraphML implementation of GraphLoader trait:

```
import ml.sparkling.graph.api.loaders.GraphLoading.LoadGraph
import ml.sparkling.graph.loaders.graphml.GraphFromGraphML.GraphML
import org.apache.spark.SparkContext

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value so it will be passed automatically
↪to graph loading API

val filePath="your_graph_path.xml"

val graph=LoadGraph.from(GraphML(filePath)).load()
```

That is simplest way of loading standard GraphML XML file (vertices are automatically indexed, and receive `VertexId` identifier):

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
    <key id="v_name" for="node" attr.name="name" attr.type="string"/>
    <key id="v_type" for="node" attr.name="type" attr.type="string"/>
    <graph id="G" edgedefault="undirected">
        <node id="n0">
            <data key="v_name">name0</data>
            <data key="v_type">type0</data>
        </node>
        <node id="n1">
            <data key="v_name">name1</data>
        </node>
        <node id="n2">
            <data key="v_name">name2</data>
        </node>
        <node id="n3">
            <data key="v_name">name3</data>
        </node>
        <edge id="e1" source="n0" target="n1"/>
        <edge id="e2" source="n1" target="n2"/>
    </graph>
</graphml>
```

All attributes associated with vertices will be puted into GraphProperties type which expands to `Map[String,Any]`. By default each edge and vertex has `id` attribute.

```
import ml.sparkling.graph.api.loaders.GraphLoading.LoadGraph
import ml.sparkling.graph.loaders.graphml.GraphFromGraphML.{GraphProperties, GraphML}
import org.apache.spark.SparkContext

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value so it will be passed automatically
↪to graph loading API

val filePath="your_graph_path.xml"

val graph: Graph[GraphProperties, GraphProperties] =LoadGraph.from(GraphML(filePath)).
↪load()
val verticesIdsFromFile: Array[String] = graph.vertices.map(_._2("id").
↪asInstanceOf[String]).collect()
```

CHAPTER 3

Graph generators

Using library you can easily generate networks using commonly used models.

Ring

Generator creates simple ring network with given number of node.

```
import ml.sparkling.graph.generators.ring.{RingGenerator, RingGeneratorConfiguration}
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value

val graph =RingGenerator.generate(RingGeneratorConfiguration(numberOfNodes=5))

// do operations on graph
```

Network can be also created in undirected version:

```
import ml.sparkling.graph.generators.ring.{RingGenerator, RingGeneratorConfiguration}
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value

val graph =RingGenerator.generate(RingGeneratorConfiguration(numberOfNodes=5,
↪undirected = true))

// do operations on graph
```

Watts And Strogatz

Model accepts three parameters:

- n - number of nodes
- k - mean degree
- β - probability of rewiring

Generation is done in two steps:

1. Ring network with n nodes is created, each of nodes is connected to $\frac{k}{2}$ nodes on left and right
2. Each edge is rewired with probability β, where new destination node is seleted randomly from all possible not exsisting connections

For further informations please refer to *[Watts]*

```
import ml.sparkling.graph.generators.wattsandstrogatz.{WattsAndStrogatzGenerator,␣
↪WattsAndStrogatzGeneratorConfiguration}
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value

val graph =WattsAndStrogatzGenerator.
↪generate(WattsAndStrogatzGeneratorConfiguration(numberOfNodes = 10,meanDegree = 2,
↪rewiringProbability = 0.5))

// do operations on graph
```

References:

CHAPTER 4

Community detection

Using library you can easily use state-of-the-art methods for community detection.

SCAN (PSCAN)

Implementation is based on *[Zhao]*. PSCAN bject implements the whole logic of algorithm. Method *computeConnectedComponents(<graph>,<epsilon>)*, takes two parameters:

- *graph* - on with algorithm will be executed
- ϵ - used for graph pruning based on similarity measure of edges.

Mentioned similarity is computed as follows:

$sim(v,u) = \frac{|N(v) \cap N(u)|}{\sqrt{|N(v)||N(u)|}}$

where $N(v)$ is neighbours set of vertex v . Edeges with similarity lower than ϵ ($sim(v,u) < \epsilon$) are removed from graph before main part of community detection.

Main part is based on label propagation and is implemented using apropriate data structures and PREGEL operator

```
import ml.sparkling.graph.operators.OperatorsDSL._
import ml.sparkling.graph.operators.algorithms.pscan.PSCAN.ComponentID
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val components: Graph[ComponentID, Int] = PSCAN.computeConnectedComponents(graph)
// Graph where each vertex is associated with its component identifier
```

You can also use more readable method using DSL

```
import ml.sparkling.graph.operators.OperatorsDSL._
import ml.sparkling.graph.operators.algorithms.pscan.PSCAN.ComponentID
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val components: Graph[ComponentID, Int] = graph.PSCAN(epsilon=0.5)
// Graph where each vertex is associated with its component identifier
```

References:

CHAPTER 5

Graph coarsening

In order to limit computation, you can decrease graph size using coarsening operator. New graph will be smaller because neighborhood vertices will be coarsed into single vertices. Edges are created using edges from input graph, filtering self loops.

Label propagation based graph coarsening

One of implementation is based on label propagation. Implementation propagates vertex identifier to neighbours. Neighbours groups them and sorts by number of occurences. If number of occurences is same, minimal one is selected (in order to gurante deterministic execution). Otherwise, vertex identifier with bigest number of occurencies (or minimal one in case of same occurencies number) is selected .

```
import ml.sparkling.graph.operators.OperatorsDSL._
import ml.sparkling.graph.api.operators.algorithms.coarsening.CoarseningAlgorithm.
↪Component
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val coarsedGraph: Graph[Component, _] = graph.LPCoarse()
// Graph where each vertex has new ID and is associated vertex IDs from input graph␣
↪that where coarsed and forms together new vertex
```

You can also coarse graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import ml.sparkling.graph.api.operators.algorithms.coarsening.CoarseningAlgorithm.
↪Component
import org.apache.spark.SparkContext
```

```
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val coarsedGraph: Graph[Component, _] = graph.LPCoarse(treatAsUndirected=true)
// Graph coarsed treating input graph as undirected
```

CHAPTER 6

Graph measures

Using SparklingGraph you can utilize multiple well-known measures for graphs.

Graph measures API

Graph measures

Each graph mesure extends GraphMeasure trait, defining what kind of value will be returned for whole graph.

Vertex measures

Each vertex mesure extends VertexMeasure trait, defining what kind of value will be returned for each vertex. For main part of measures that will be a single number (like Double) but for some of them a tupple (or other data type) can be returned (like (Double,Double)). Each measure defines also implicit methods for graph, thanks to what your code will be more readable, and you will develop your experiments faster.

Measures accepts VertexMeasureConfiguration in order to configure computation process. You can set following parameters:

- BucketSizeProvider - used in more complex computations in order to divide data into buckets
- `treatAsUndirected:Boolean` - graph will be treated as undirected during computations

Edges measures

Each edge mesure extends EdgeMeasure trait, defining what kind of value will be returned for each edge, and what kind of data is expected for each vertex. Each measure defines also implicit methods for graph, thanks to what your code will be more readable, and you will develop your experiments faster.

Measures accepts parameters:

- `treatAsUndirected:Boolean` - graph will be treated as undirected during computations

Beside defining methods for computing measure for whole graph, method (computeValues) for single edge is also present.

Measures

Curretnly you can use following measures:

- Vertex measures:

Closeness centrality

Closeness centrality measure is defined as inverted sum of distances (`d(y,x)`) from given node to all other nodes. Distance is defined as length of shortest path.

$C(x) = \frac{1}{\sum_{y \neq x} d(y,x)}$

Measure can be understood as how far away from other nodes given node is located. For further informations please refer to *[Sabidussi]*.

Because of computational complexity of shortest paths computation, measure computation can be time consuming. Library uses pregel operator in order to do computations.

For memory consumption optimization, informations about distances are held in memory efficient implementations of collections available in fastutil library.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.closenessCentrality()
// Graph where each vertex is associated with its closeness centrality
```

In order to limit memory consumption during computation closeness is computed for each vertex separately. In near future there will be functionality that will let you to decide for how many nodes at once computation should be done.

You can also compute closeness centrality for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.
↪closenessCentrality(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its closeness centrality computed␣
↪for undirected graph
```

References:

Eigenvector centrality

Eigenvector centrality measure give us information about how given node is important in network. It is based on degree centrality. In here we have more sophisticated version, where connections are not equal.

$E(x) = \frac{1}{\lambda} \sum_{j=1}^{n} A_{ij} x_j$

Eigenvector centrality is more general approach than PageRank. For further informations please refer to *[Newman]*.

Library uses pregel operator in order to do computations.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.eigenvectorCentrality()
// Graph where each vertex is associated with its eigenvector centrality
```

You can also compute eigenvector centrality for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.
↪eigenvectorCentrality(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its eigenvector centrality computed␣
↪for undirected graph
```

Eigenvector centrality is implemented using iterative approach and Pregel operator. Because of that you can provide your own computation stop predicate:

```
import org.apache.spark.graphx.GraphLoader
import org.apache.spark.sql.SparkSession
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import ml.sparkling.graph.operators.measures.vertex.eigenvector.
↪EigenvectorCentrality
import ml.sparkling.graph.operators.OperatorsDSL._

// initialize your SparkContext as implicit value
```

```
val graph =???
val eic = EigenvectorCentrality.computeEigenvector(graph,
↪VertexMeasureConfiguration(),(iteration,oldValue,newValue)=>iteration<999).
↪vertices
```

As you can see, you can also use average values of Eigenvector centrality in consecutive iterations.

References:

HITS

After measure computation, each vertex of graph will have assigned two scores `(hub,authority)`. Where `hub` score is proportional to sum of `authority` score of its neighbours, and `authority` score is proportional to sum of `hub` score of its neighbours.

For further informations please refer to *[Kleinberg]*.

Here you can see how to use measure:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[(Double,Double), _] = graph.hits()
// Graph where each vertex is associated with its hits score (represented as a␣
↪tuple (auth,hub):(Double,Double))
```

You can also compute HITS for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.
↪hits(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its hits score computed for␣
↪undirected graph
```

References:

Degree centrality

Degree of a node is number of connections that its has. When we have directed network, we can distinguish indegree (input edges) and outdegree (output edges). We can treate degree as a centrality measure. Nodes with high degree can be assumed as important. Ofcourse it depends on the sitution, and interpretations can differ.

For further informations please refer to *[lecture]*.

Method returns a tuple `(outdegree,indegree):(Int,Int)`. If computations will be done using `treatAsUndirected`, both values will be equal.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[(Int,Int), _] = graph.degreeCentrality()
// Graph where each vertex is associated with its degree centrality in form of␣
↪tuple (outdegree,indegree):(Int,Int)
```

You can also compute closeness centrality for graph treated it as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.
↪degreeCentrality(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its degree centrality computed for␣
↪undirected graph in form of tuple (degree,degree):(Int,Int)
```

References:

Neighborhood Connectivity

Neighborhood connectivity is a measure based on degree centrality. Connectivity of a vertex is its degree. Neighborhood connectivity is average connectivity of neighbours of given vertex.

$NC(x) = \frac{\sum_{k \in N(x)} |N(k)|}{|N(x)|}$

Where $N(x)$ is set of neighbours of vertex x

For further informations please refer to *[Maslov]*.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)
```

```
val centralityGraph: Graph[Double, _] = graph.neighborhoodConnectivity()
// Graph where each vertex is associated with its neighborhood connectivity
```

You can also compute neighborhood connectivity for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.
↪neighborhoodConnectivity(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its neighborhood connectivity␣
↪computed for undirected graph
```

References:

Vertex Embeddedness

Is an average embededdness of neighbours of given vertex.

$VE(x) = \frac{1}{|N(x)|} \sum_{v \in N(x)} \frac{|N(x) \cap N(v)|}{|N(x) \cup N(v)|}$

Where $N(x)$ is set of neighbours of vertex x

For further informations please refer to *[Dong]*.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.vertexEmbeddedness()
// Graph where each vertex is associated with its vertex embeddedness
```

You can also compute vertex embeddedness for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)
```

```
val centralityGraph: Graph[Double, _] = graph.
↪vertexEmbeddedness(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its vertex embeddedness computed␣
↪for undirected graph
```

References:

Local Clustering Coefficient

Local Clustering Coefficient for vertex tells us howe close its neighbors are. It's number of existing connections in neighborhood divided by number of all possible connections.

$LC(x) = \sum_{v \in N(x)} \frac{|N(x) \cap N(v)|}{|N(x)| * (|N(x)| - 1)}$

Where $N(x)$ is set of neighbours of vertex x

For further informations please refer to *[Watts]*.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.localClustering()
// Graph where each vertex is associated with its local clustering coefficient
```

You can also compute local clustering coefficient for graph treating it as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.api.operators.measures.VertexMeasureConfiguration
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val centralityGraph: Graph[Double, _] = graph.
↪localClustering(VertexMeasureConfiguration(treatAsUndirected=true))
// Graph where each vertex is associated with its local clustering coefficient␣
↪computed for undirected graph
```

References:

- Graph measures:

Freeman's network centrality

Freeman's centrality tells us how heterogenous is degree centrality ammong vertices of network. For start network, we will get a value 1.

$FC(g) = \frac{\sum_{x \in g} N_{max} - |N(x)|}{(|g| - 1) * (|g| - 2)}$

Where g is given graph, $N(x)$ returns set of neighbours of vertex x, $|g|$ is number of vertices in graph g and N_{max} is maximal degree that can be observed in network.

For further informations please refer to *[Freeman]*.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val freemanCentrality: Double= graph.freemanCentrality()
// Freeman centrality value for graph
```

References:

Modularity

Modularity measures strength of division of a network into communities (modules,clusters). Measures takes values from range $< -1, 1 >$. Value close to 1 indicates strong community structure. When $Q = 0$ then the community division is not better than random.

$Q = \sum_{i=1}^{k} (e_{ii} - a_i^2)$

Where k is number of communities, e_{ii} is number of edges that has both ends in community i and a_i is number of edges with one end in community i

For further informations please refer to *[lecture]* and *[Newman]*.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val modularity: Double= graph.modularity()
// Modularity value for graph
```

References:

- Edges measures:

Adamic/Adar

Adamic/Adar measures is defined as inverted sum of degrees of common neighbours for given two vertices.

$A(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{log(|N(u)|)}$

Where $N(x)$ is set of neighbours of vertex x

For further informations please refer to *[Adamic]*.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val adamicAdarGraph: Graph[_, Double] = graph.
↪adamicAdar(VertexMeasureConfiguration((g:Graph[_,_])=>10l))
// Graph where each edge is associated with its Adamic/Adar measure
```

You can also compute closeness centrality for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val adamicAdarGraph: Graph[_, Double] = graph.adamicAdar(treatAsUndirected=true)
// Graph where each edge is associated with its Adamic/Adar measure  where edges␣
↪are treated as undirected
```

References:

Common Neighbours

Common Neighbours measure is defined as number of common neighbours of two given vertices.

$CN(x, y) = |N(x) \cap N(y)|$

Where $N(x)$ is set of neighbours of vertex x

For further informations please refer to *[Newman]*.

For memory consumption optimization, informations about neighbours are held in memory efficient implementations of collections available in fastutil library.

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val commonNeighbours: Graph[_, Int] = graph.commonNeighbours()
// Graph where each edge is associated with number of common neighbours of␣
↪vertices on edge
```

You can also compute common neighbours for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val commonNeighbours: Graph[_, Int] = graph.
 ↪commonNeighbours(treatAsUndirected=true)
// raph where each edge is associated with number of common neighbours of␣
 ↪vertices on edge where edges are treated as undirected
```

References:

CHAPTER 7

Partitioning methods

Library provides multiple methods for graph partitioning. By default GraphX provides only random methods, in SparklingGraph you can find approaches that are using structural properties of graphs in order to minimize computation times and storage overheads.

All methods can be found in partitioning package

Propagation bases

In that approach, label propagation is used in order to determine vertex cluster id. In iterative way, alghoritm propagates vertices ids. In each step, vertex selects minimal id from all recived. Steps are repeated until number of components in graph is less than or equal number of requested partitions. If number of unique clusters ids is not equal to the number of requested partitions, alghoritm selects closer solution.

```
import ml.sparkling.graph.operators.partitioning.PropagationBasedPartitioning
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph = ???
// load your graph (for example using Graph loading API)
val numberOfRequiredPartitions=24
val partitionedGraph =  PropagationBasedPartitioning.partitionGraphBy(graph,
↪numberOfRequiredPartitions)
```

Naive PSCAN

Aglhorimt use PSCAN alghoritm to determine comunities in graph and then use them as partitions. Without configuration, method use default PSCAN configuration, but that can be changed if it is needed.

```
import ml.sparkling.graph.operators.partitioning.CommunityBasedPartitioning
import ml.sparkling.graph.operators.algorithms.community.pscan.PSCAN
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph = ???
// load your graph (for example using Graph loading API)
val communityDetectionMethod=PSCAN
val partitionedGraph =  CommunityBasedPartitioning.partitionGraphBy(graph,
↪communityDetectionMethod)
```

In order to change parameters you can use

```
import ml.sparkling.graph.operators.partitioning.CommunityBasedPartitioning
import ml.sparkling.graph.operators.algorithms.community.pscan.PSCAN
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph = ???
// load your graph (for example using Graph loading API)
val partitionedGraph =  CommunityBasedPartitioning.partitionGraphBy(graph,PSCAN.
↪computeConnectedComponents(_,epsilon = 0))
```

Dynamic PSCAN

That is solution that use PSCAN alghoritm in conduction with epsilon parameter search. Aglhoritm looks for possible epsilon values and use binary search to find one that terurns clustering that hase size closest to requested number of partitions. Found clustering is used as partitioning.

```
import ml.sparkling.graph.operators.partitioning.PSCANBasedPartitioning
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph = ???
// load your graph (for example using Graph loading API)
val numberOfRequiredPartitions=24
val partitionedGraph =  PSCANBasedPartitioning.partitionGraphBy(graph,
↪numberOfRequiredPartitions)
```

CHAPTER 8

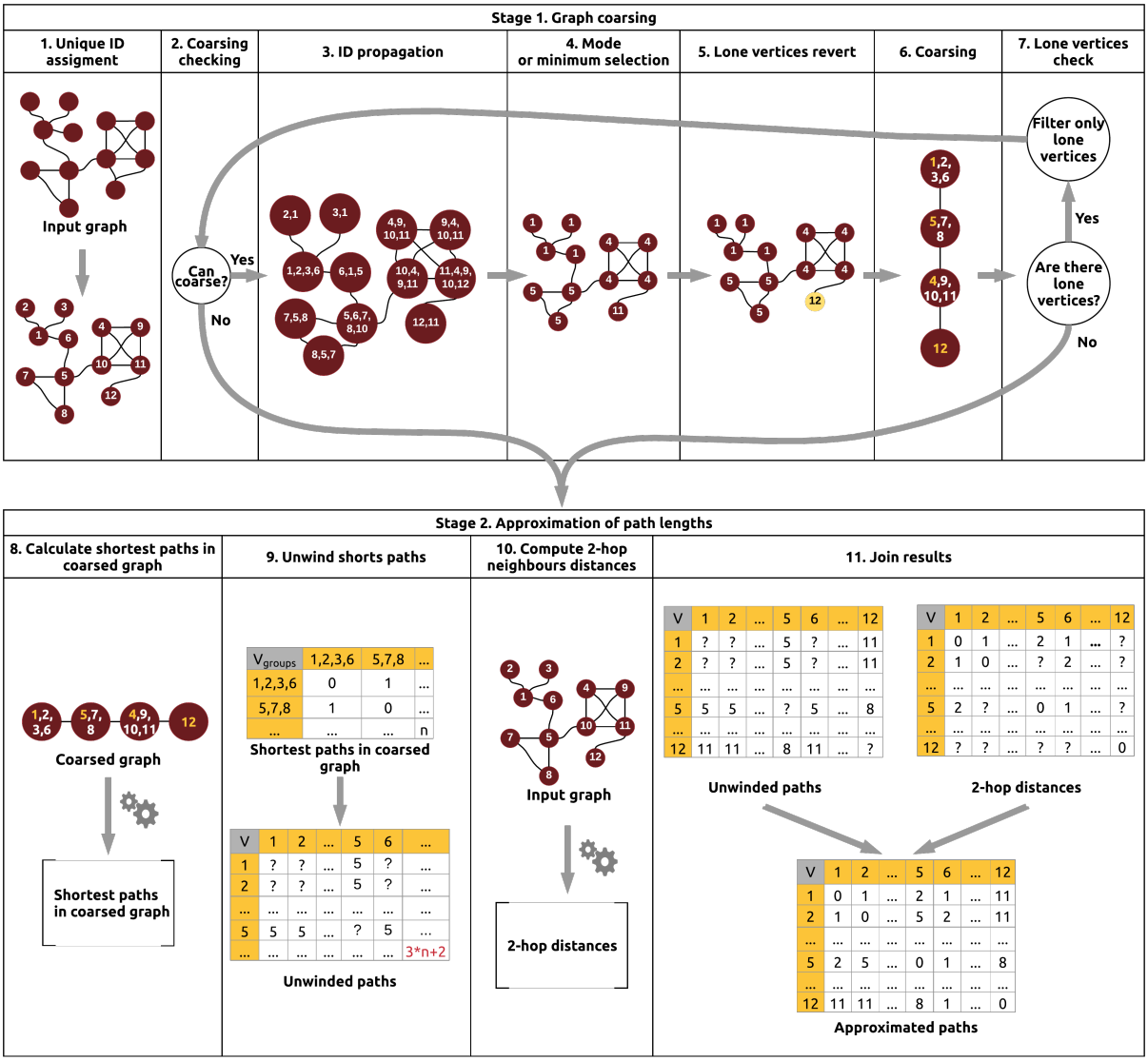
Shortest paths approximation

In order to limit computation time of shortest paths for large graphs, library gives ability to approximate them. Approximation can be divided into four main phases:

1. Graph coarsening
2. Paths calculation in coarsed graph
3. 2-hop neighborhood distances calculation
4. Paths approximation

Approximation gives worst-case result of 3*p+2 where p is real path. Result is not awesome in terms of beeing exact, but it keeps rankings of vertices and can be used for measures approximation (Closeness) or in tasks where order of vertices is important, not exact distance.

Alghotim block scheme



Examples

Alghoritm API lets to compute paths :

- For single vertex:

```
import ml.sparkling.graph.operators.algorithms.aproximation.
↪ApproximatedShortestPathsAlgorithm
import org.apache.spark.SparkContext
import org.apache.spark.graphx.{Graph, VertexRDD}

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
```

```
val graph = ???
// load your graph (for example using Graph loading API)
val sourceVertexId=1
val graphWithPaths=ApproximatedShortestPathsAlgorithm.
↪computeSingleShortestPathsLengths(graph,sourceVertexId)
val paths : VertexRDD[Iterable[(VertexId, JDouble)]]  =  graphWithPaths.vertices
```

- For whole graph:

```
import ml.sparkling.graph.operators.algorithms.aproximation.
↪ApproximatedShortestPathsAlgorithm
import org.apache.spark.SparkContext
import org.apache.spark.graphx.{Graph, VertexRDD}

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph = ???
// load your graph (for example using Graph loading API)
val graphWithPaths =  ApproximatedShortestPathsAlgorithm.
↪computeShortestPaths(graph)
val paths : VertexRDD[Iterable[(VertexId, JDouble)]]  =  graphWithPaths.vertices
```

- using iterative approach:

```
import ml.sparkling.graph.operators.algorithms.aproximation.
↪ApproximatedShortestPathsAlgorithm
import org.apache.spark.SparkContext
import org.apache.spark.graphx.{Graph, VertexRDD}

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph = ???
// load your graph (for example using Graph loading API)
val bucketSize=10
val graphWithPaths =  ApproximatedShortestPathsAlgorithm.
↪computeShortestPathsLengthsIterative(graph, (g:Graph[_,_])=>bucketSize)
val paths : VertexRDD[Iterable[(VertexId, JDouble)]]  =  graphWithPaths.vertices
```

CHAPTER 9

Link Prediction

Using library you can easily use state-of-the-art methods for link prediction.

Measure based link prediction

Basic appraoch that is using simmilarity computed between two vertices. ‘MeasureBasedLnkPredictor‘<http://sparkling-graph.github.io/sparkling-graph/latest/api/#ml.sparkling.graph.api.operators.algorithms.link.MeasureBasedLnkPredictor>_ is trait for that approach

Basic measure based link prediction

Most basic implementation of measure based link prediction. All possible vertices combinations are computed for given graph. In next step, similarity measure is computed for each combination. Combinations that exsits or creates loops (self connections) are filtered out. Combinations that have similarity lower than given treshold are also filtered out. Implementation can be found in BasicLinkPredictor

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.operators.measures.edge.CommonNeighbours
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val predictedEdges: RDD[(VertexId,VertexId)] = graph.
↪predictLinks(edgeMeasure=CommonNeighbours,threshold=10)
// RDD with predicted edges using Common Neighbours measure, and 10 as a minimal␣
↪number of common neighbours
```

You can also predict links for graph treated as undirected one:

```
import ml.sparkling.graph.operators.OperatorsDSL._
import org.apache.spark.SparkContext
import ml.sparkling.graph.operators.measures.edge.AdamicAdar
import org.apache.spark.graphx.Graph

implicit ctx:SparkContext=???
// initialize your SparkContext as implicit value
val graph =???
// load your graph (for example using Graph loading API)

val predictedEdges: RDD[(VertexId,VertexId)] = graph.
↪predictLinks(edgeMeasure=AdamicAdar,threshold=2,treatAsUndirected=true)
// RDD with predicted edges using Adamic/Adar measure, and 2 as a minimal value of␣
↪measure, graph is treated as undirected
```

CHAPTER 10

Project TO-DO

Please check code issue tracker and docs issue tracker

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Bibliography

[Watts] Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world'networks. nature, 393(6684), 440-442. Nature

[Zhao] Zhao, W., Martha, V., & Xu, X. (2013, March). PSCAN: a parallel Structural clustering algorithm for big networks in MapReduce. In Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on (pp. 862-869). IEEE. PDF

[Sabidussi] Sabidussi, G. (1966). The centrality index of a graph.Psychometrika, 31(4):581–603, Springer

[Newman] Newman, M. E. (2008). The mathematics of networks. The new palgrave encyclopedia of economics, 2(2008):1–12., PDF

[Kleinberg] Kleinberg, J. M. (1999). Hubs, authorities, and communities. ACM Computing Surveys (CSUR), 31(4es):5., PDF

[lecture] Dr. Cecilia Mascolo, Social and Technological Network'Analysis PDF

[Maslov] Maslov S, Sneppen K . Specificity and stability in topology of protein networks. Science 2002;296:910-913. HTML

[Dong] Dong, Y., Yang, Y., Tang, J., Yang, Y., & Chawla, N. V. (2014, August). Inferring user demographics and social strategies in mobile social networks. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 15-24). ACM. PDF

[Watts] 4. (a) Watts and Steven Strogatz, "Collective dynamics of 'small-world' networks", Nature, vol. 393, pp 440-442, 1998 HTML

[Freeman] Freeman, L. C. (1978). Centrality in social networks conceptual clarification. Social networks, 1(3), 215-239., PDF

[lecture] Carl Kingsford (2009). Modularity, PDF

[Newman] Newman, M. E., & Girvan, M. (2004). Finding and evaluating community structure in networks. Physical review E, 69(2), 026113. PDF

[Adamic] Adamic, L. A. and Adar, E. (2003). Predicting missing links via local information.SocialNetworks, 25(3):211–230 Springer

[Newman] Newman, M. E. J. (2001). Clustering and preferential attachment in growing networks. pages1–13, PDF