GraphFrames User Guide

Creating GraphFrames

Users can create GraphFrames from vertex and edge DataFrames.

- *Vertex DataFrame*: A vertex DataFrame should contain a special column named "id" which specifies unique IDs for each vertex in the graph.
- *Edge DataFrame*: An edge DataFrame should contain two special columns: "src" (source vertex ID of edge) and "dst" (destination vertex ID of edge).

Both DataFrames can have arbitrary other columns. Those columns can represent vertex and edge attributes.

A GraphFrame can also be constructed from a single DataFrame containing edge information. The vertices will be inferred from the sources and destinations of the edges.

Scala **Python**

The following example demonstrates how to create a GraphFrame from vertex and edge DataFrames.

```
# 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)
```

The GraphFrame constructed above is available in the GraphFrames package:

```
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends()
```

Basic graph and DataFrame queries

GraphFrames provide several simple graph queries, such as node degree.

Also, since GraphFrames represent graphs as pairs of vertex and edge DataFrames, it is easy to make powerful queries directly on the vertex and edge DataFrames. Those DataFrames are made available as `vertices` and `edges` fields in the GraphFrame.

Scala | **Python**

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

# Display the vertex and edge DataFrames
g.vertices.show()
# +--+-------+---+
# |id|   name|age|
# +--+-------+---+
# | a|  Alice| 34|
# | b|    Bob| 36|
# | c|Charlie| 30|
# | d|  David| 29|
# | e| Esther| 32|
# | f|  Fanny| 36|
# | g|  Gabby| 60|
# +--+-------+---+

g.edges.show()
# +---+---+------------+
# |src|dst|relationship|
# +---+---+------------+
# |  a|  b|      friend|
# |  b|  c|      follow|
# |  c|  b|      follow|
# |  f|  c|      follow|
# |  e|  f|      follow|
# |  e|  d|      friend|
# |  d|  a|      friend|
# |  a|  e|      friend|
# +---+---+------------+

# 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()
```

Motif finding

Motif finding refers to searching for structural patterns in a graph.

GraphFrame motif finding uses a simple Domain-Specific Language (DSL) for expressing structural queries. For example, `graph.find("(a)-[e]->(b); (b)-[e2]->(a)")` will search for pairs of vertices `a,b` connected by edges in both directions. It will return a `DataFrame` of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. In this case, the returned columns will be “a, b, e, e2.”

DSL for expressing structural patterns:

- The basic unit of a pattern is an edge. For example, `"(a)-[e]->(b)"` expresses an edge `e` from vertex `a` to vertex `b`. Note that vertices are denoted by parentheses `(a)`, while edges are denoted by square brackets `[e]`.
- A pattern is expressed as a union of edges. Edge patterns can be joined with semicolons. Motif `"(a)-[e]->(b); (b)-[e2]->(c)"` specifies two edges from `a` to `b` to `c`.
- Within a pattern, names can be assigned to vertices and edges. For example, `"(a)-[e]->(b)"` has three named elements: vertices `a,b` and edge `e`. These names serve two purposes:
  - The names can identify common elements among edges. For example, `"(a)-[e]->(b); (b)-[e2]->(c)"` specifies that the same vertex `b` is the destination of edge `e` and source of edge `e2`.
  - The names are used as column names in the result `DataFrame`. If a motif contains named vertex `a`, then the result `DataFrame` will contain a column "a" which is a `StructType` with sub-fields equivalent to the schema (columns) of `GraphFrame.vertices`. Similarly, an edge `e` in a motif will produce a column "e" in the result `DataFrame` with sub-fields equivalent to the schema (columns) of `GraphFrame.edges`.
  - Be aware that names do *not* identify *distinct* elements: two elements with different names may refer to the same graph element. For example, in the motif `"(a)-[e]->(b); (b)-[e2]->(c)"`, the names `a` and `c` could refer to the same vertex. To restrict named elements to be distinct vertices or edges, use post-hoc filters such as `resultDataframe.filter("a.id != c.id")`.
- It is acceptable to omit names for vertices or edges in motifs when not needed. E.g., `"(a)-[]->(b)"` expresses an edge between vertices `a,b` but does not assign a name to the edge. There will be no column for the anonymous edge in the result `DataFrame`. Similarly, `"(a)-[e]->()"` indicates an out-edge of vertex `a` but does not name the destination vertex. These are called *anonymous* vertices and edges.
- An edge can be negated to indicate that the edge should *not* be present in the graph. E.g., `"(a)-[]->(b); !(b)-[]->(a)"` finds edges from `a` to `b` for which there is *no* edge from `b` to `a`.

Restrictions:

- Motifs are not allowed to contain edges without any named elements: `"()-[]->()"` and `"!()-[]->()"` are prohibited terms.
- Motifs are not allowed to contain named edges within negated terms (since these named edges would never appear within results). E.g., `"!(a)-[ab]->(b)"` is invalid, but `"!(a)-[]->(b)"` is valid.

More complex queries, such as queries which operate on vertex or edge attributes, can be expressed by applying filters to the result `DataFrame`.

This can return duplicate rows. E.g., a query `"(u)-[]->()"` will return a result for each matching edge, even if those edges share the same vertex `u`.

Scala **Python**

For API details, refer to the API docs.

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

# Search for pairs of vertices with edges in both directions between them.
motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")
motifs.show()

# More complex queries can be expressed by applying filters.
motifs.filter("b.age > 30").show()
```

Many motif queries are stateless and simple to express, as in the examples above. The next examples demonstrate more complex queries which carry state along a path in the motif. These queries can be expressed by combining GraphFrame motif finding with filters on the result, where the filters use sequence operations to construct a series of `DataFrame` `Column`s.

For example, suppose one wishes to identify a chain of 4 vertices with some property defined by a sequence of functions. That is, among chains of 4 vertices `a->b->c->d`, identify the subset of chains matching this complex filter:

- Initialize state on path.
- Update state based on vertex `a`.
- Update state based on vertex `b`.
- Etc. for `c` and `d`.
- If final state matches some condition, then the chain is accepted by the filter.

The below code snippets demonstrate this process, where we identify chains of 4 vertices such that at least 2 of the 3 edges are "friend" relationships. In this example, the state is the current count of "friend" edges; in general, it could be any `DataFrame Column`.

Scala Python

```
from pyspark.sql.functions import col, lit, when
from pyspark.sql.types import IntegerType
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends()  # Get example graph

chain4 = g.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[cd]->(d)")

# Query on sequence, with state (cnt)
#  (a) Define method for updating state given the next element of the motif.
sumFriends =\
  lambda cnt,relationship: when(relationship == "friend", cnt+1).otherwise(cnt)
#  (b) Use sequence operation to apply method to sequence of elements in motif.
#      In this case, the elements are the 3 edges.
condition =\
  reduce(lambda cnt,e: sumFriends(cnt, col(e).relationship), ["ab", "bc", "cd"], lit(0))
#  (c) Apply filter to DataFrame.
chainWith2Friends2 = chain4.where(condition >= 2)
chainWith2Friends2.show()
```

The above example demonstrated a stateful motif for a fixed-length chain. Currently, in order to search for variable-length motifs, users need to run one query for each possible length. However, the above query patterns allow users to re-use the same code for each length, with the only change being to update the sequence of motif elements ("ab", "bc", "cd" above).

Subgraphs

In GraphX, the `subgraph()` method takes an edge triplet (edge, src vertex, and dst vertex, plus attributes) and allows the user to select a subgraph based on triplet and vertex filters.

GraphFrames provide an even more powerful way to select subgraphs based on a combination of motif finding and DataFrame filters. We provide three helper methods for subgraph selection. `filterVertices(condition)`, `filterEdges(condition)`, and `dropIsolatedVertices()`.

Simple subgraph: vertex and edge filters: The following example shows how to select a subgraph based upon vertex and edge filters.

Scala Python

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

# Select subgraph of users older than 30, and relationships of type "friend".
# Drop isolated vertices (users) which are not contained in any edges (relationships).
g1 = g.filterVertices("age > 30").filterEdges("relationship = 'friend'").dropIsolatedVertices()
```

Complex subgraph: triplet filters: The following example shows how to select a subgraph based upon triplet filters which operate on an edge and its src and dst vertices. This example could be extended to go beyond triplets by using more complex motifs.

Scala Python

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

# Select subgraph based on edges "e" of type "follow"
# pointing from a younger user "a" to an older user "b".
paths = g.find("(a)-[e]->(b)")\
  .filter("e.relationship = 'follow'")\
  .filter("a.age < b.age")
# "paths" contains vertex info. Extract the edges.
e2 = paths.select("e.src", "e.dst", "e.relationship")
# In Spark 1.5+, the user may simplify this call:
#  val e2 = paths.select("e.*")

# Construct the subgraph
g2 = GraphFrame(g.vertices, e2)
```

Graph algorithms

GraphFrames provides the same suite of standard graph algorithms as GraphX, plus some new ones. We provide brief descriptions and code snippets below. See the API docs for more details.

Some of the algorithms are currently wrappers around GraphX implementations, so they may not be more scalable than GraphX. More algorithms will be migrated to native GraphFrames implementations in the future.

Breadth-first search (BFS)

Breadth-first search (BFS) finds the shortest path(s) from one vertex (or a set of vertices) to another vertex (or a set of vertices). The beginning and end vertices are specified as Spark DataFrame expressions.

See Wikipedia on BFS for more background.

Scala **Python**

The following code snippets uses BFS to find path between vertex with name "Esther" to a vertex with age < 32.
For API details, refer to the API docs.

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

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

# Specify edge filters or max path lengths.
g.bfs("name = 'Esther'", "age < 32",\
  edgeFilter="relationship != 'friend'", maxPathLength=3)
```

Connected components

Computes the connected component membership of each vertex and returns a graph with each vertex assigned a component ID.

See Wikipedia for background.

NOTE: With GraphFrames 0.3.0 and later releases, the default Connected Components algorithm requires setting a Spark checkpoint directory. Users can revert to the old algorithm using `connectedComponents.setAlgorithm("graphx")`.

Scala **Python**

For API details, refer to the API docs.

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

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

Strongly connected components

Compute the strongly connected component (SCC) of each vertex and return a graph with each vertex assigned to the SCC containing that vertex.

See Wikipedia for background.

Scala **Python**

For API details, refer to the API docs.

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

result = g.stronglyConnectedComponents(maxIter=10)
result.select("id", "component").orderBy("component").show()
```

Label Propagation Algorithm (LPA)

Run static Label Propagation Algorithm for detecting communities in networks.

Each node in the network is initially assigned to its own community. At every superstep, nodes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages.

LPA is a standard community detection algorithm for graphs. It is very inexpensive computationally, although (1) convergence is not guaranteed and (2) one can end up with trivial solutions (all nodes are identified into a single community).

See Wikipedia for background.

Scala **Python**

For API details, refer to the API docs.

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

result = g.labelPropagation(maxIter=5)
result.select("id", "label").show()
```

PageRank

There are two implementations of PageRank.

- The first one uses the `org.apache.spark.graphx.graph` interface with `aggregateMessages` and runs PageRank for a fixed number of iterations. This can be executed by setting `maxIter`.
- The second implementation uses the `org.apache.spark.graphx.Pregel` interface and runs PageRank until convergence and this can be run by setting `tol`.

Both implementations support non-personalized and personalized PageRank, where setting a `sourceId` personalizes the results for that vertex.

See Wikipedia for background.

Scala | **Python**

For API details, refer to the API docs.

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

# Run PageRank until convergence to tolerance "tol".
results = g.pageRank(resetProbability=0.15, tol=0.01)
# Display resulting pageranks and final edge weights
# Note that the displayed pagerank may be truncated, e.g., missing the E notation.
# In Spark 1.5+, you can use show(truncate=False) to avoid truncation.
results.vertices.select("id", "pagerank").show()
results.edges.select("src", "dst", "weight").show()

# Run PageRank for a fixed number of iterations.
results2 = g.pageRank(resetProbability=0.15, maxIter=10)

# Run PageRank personalized for vertex "a"
results3 = g.pageRank(resetProbability=0.15, maxIter=10, sourceId="a")

# Run PageRank personalized for vertex ["a", "b", "c", "d"] in parallel
results4 = g.parallelPersonalizedPageRank(resetProbability=0.15, sourceIds=["a", "b", "c", "d"], ma
xIter=10)
```

Shortest paths

Computes shortest paths from each vertex to the given set of landmark vertices, where landmarks are specified by vertex ID. Note that this takes edge direction into account.

See Wikipedia for background.

Scala | **Python**

For API details, refer to the API docs.

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

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

Triangle count

Computes the number of triangles passing through each vertex.

Scala | **Python**

For API details, refer to the API docs.

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

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

Saving and loading GraphFrames

Since GraphFrames are built around DataFrames, they automatically support saving and loading to and from the same set of datasources. Refer to the Spark SQL User Guide on datasources for more details.

The below example shows how to save and then load a graph.

Scala Python

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

# Save vertices and edges as Parquet to some location.
g.vertices.write.parquet("hdfs://myLocation/vertices")
g.edges.write.parquet("hdfs://myLocation/edges")

# Load the vertices and edges back.
sameV = sqlContext.read.parquet("hdfs://myLocation/vertices")
sameE = sqlContext.read.parquet("hdfs://myLocation/edges")

# Create an identical GraphFrame.
sameG = GraphFrame(sameV, sameE)
```

Message passing via AggregateMessages

Like GraphX, 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.

The below code snippets show how to use `aggregateMessages` to compute the sum of the ages of adjacent users.

Scala Python

For API details, refer to the API docs.

```
from pyspark.sql.functions import sum as sqlsum
from graphframes.lib import AggregateMessages as AM
from graphframes.examples import Graphs
g = Graphs(sqlContext).friends()  # Get example graph

# For each user, sum the ages of the adjacent users.
msgToSrc = AM.dst["age"]
msgToDst = AM.src["age"]
agg = g.aggregateMessages(
    sqlsum(AM.msg).alias("summedAges"),
    sendToSrc=msgToSrc,
```