Big data: architectures and data analytics

Graph Analytics in Spark

Part 2

Graph Algorithms with GraphFrames

Algorithms over graphs

- A graph is just a logical representation of data
- Graph theory provides many algorithms for analyzing data in this format
  - Breadth first search
  - Shortest paths
  - Connected components
  - Strongly connected component
  - Label propagation
  - PageRank
  - ...
- Custom algorithms can be built
- Development continues as new algorithms are added to GraphFrames

Algorithms over graphs

- A graph is just a logical representation of data
- Graph theory provides many algorithms for analyzing data in this format
  - Breadth first search
  - Shortest paths
  - Connected components
  - Strongly connected component
  - Label propagation
  - PageRank
  - ...
- Custom algorithms can be built
- Development continues as new algorithms are added to GraphFrames

Here presented

Checkpoint directory

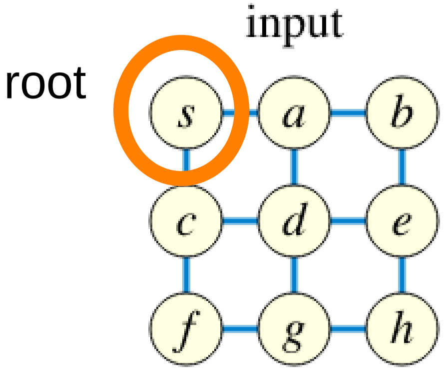
- To run some expensive algorithms, set a checkpoint directory that will store the state of the job at every iteration
- This allow you to continue where you left off if the job crashes
- Create such a folder to set the checkpoint directory with:
  sc.setCheckpointDir('graphframes_ckpts_dir'**)**
- graphframes_ckpts_dir is your new checkpoint folder
- **sc** is your **spark.sparkContext**

Breadth first search

- Breadth-first search (BFS) is an algorithm for traversing or searching graph data structures
- It finds the **shortest path** from a vertex to other vertices
- Used in many other algorithms: length of shortest paths, connected components,...

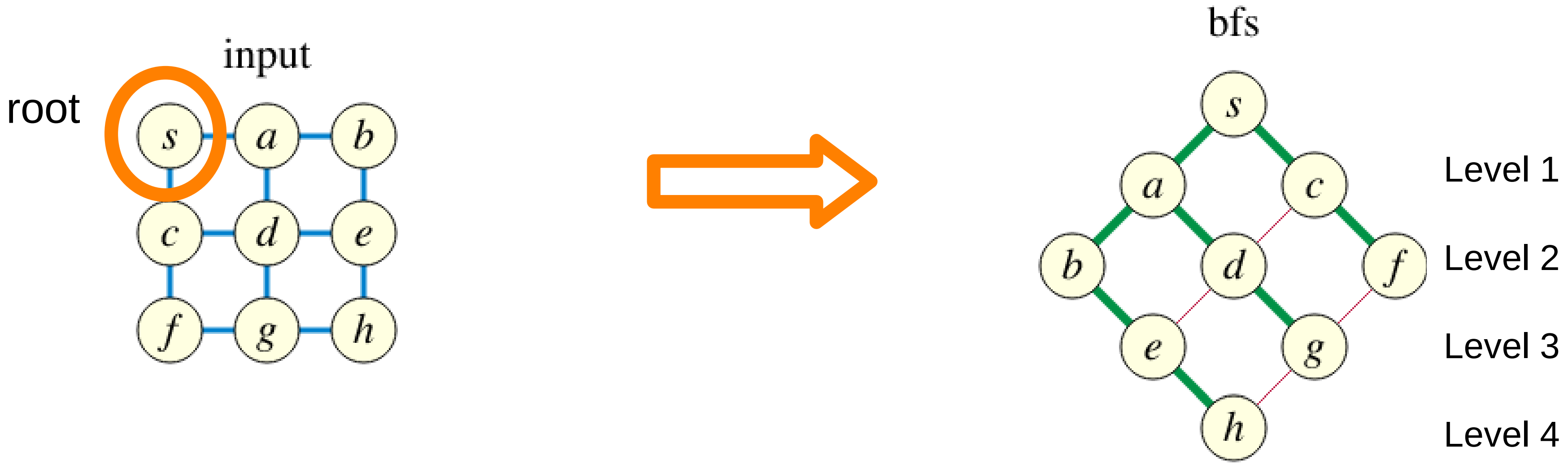
Breadth first search

- It starts at an arbitrary node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level



Breadth first search

- It starts at an arbitrary node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level



Breadth first search

- 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)
- **bfs()** method returns a DataFrame of valid shortest paths from vertices matching fromExpr to vertices matching toExpr
- Shortest means globally shortest path. If there are many vertices matching fromExpr and toExpr, only the couple with shortest length is showed
- If multiple paths are valid and **have the same length,** the DataFrame will return one Row for each path

Breadth first search

- Parameters:
  - fromExpr: Spark SQL expression specifying valid starting vertices for the BFS. E.g., to start from a specific vertex, "id = [start vertex id]"
  - toExpr: Spark SQL expression specifying valid target vertices for the BFS
  - maxPathLength: Limit on the length of paths (default = 10)
  - edgeFilter: Spark SQL expression specifying edges which may be used in the search

Breadth first search: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

1) Find the shortest path from Esther to Charlie

2) Find the shortest path from Esther to users of age more than 34, without using edges of type “follow”

Breadth first search: example

```
# 1)
#Search shortest path from "Esther" to "Charlie"
paths = g.bfs(fromExpr="id = 'e'", toExpr="id = 'c'")
paths.show()

# 2)
# Find the shortest path from Esther to users of age more than 34,
    without using edges of type “follow”
# Specify edge filters or max path lengths.
paths2=g.bfs("name = 'Esther'", "age > 34",\
  edgeFilter="relationship != 'follow'")
paths2.show()
```

Breadth first search: example

1) Find the shortest path from Esther to Charlie

from	e0	v1	e1	to
[e, Esther, 32]	[e, f, follow]	[f, Fanny, 36]	[f, c, follow]	[c, Charlie, 30]

2) Find the shortest path from Esther to users of age more than 34, without using edges of type "follow"

from	e0	v1	e1	v2	e2	to
[e, Esther, 32]	[e, d, friend]	[d, David, 29]	[d, a, friend]	[a, Alice, 34]	[a, b, friend]	[b, Bob, 36]

Shortest path

- **shortestPaths()** method of a GraphFrame computes length of shortest paths from each vertex to a given set of landmark vertices
- Landmarks are specified by vertex ID.
- It uses the breadth-first search
- The returned DataFrame contains all graph vertex IDs and an additional column
  - a map containing for each reachable landmark vertex (key), the shortest-path distance (value)

Shortest paths: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Find the shortest paths going to Alice and David

Shortest paths: example

```
#list of landmark nodes
landmarks=["a", "d"]
results = g.shortestPaths(landmarks=landmarks)

results.show()
```

Shortest paths: example

Nodes DataFrame

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36
g	Gabby	60

Edge DataFrame

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

Find the shortest paths to Alice and David

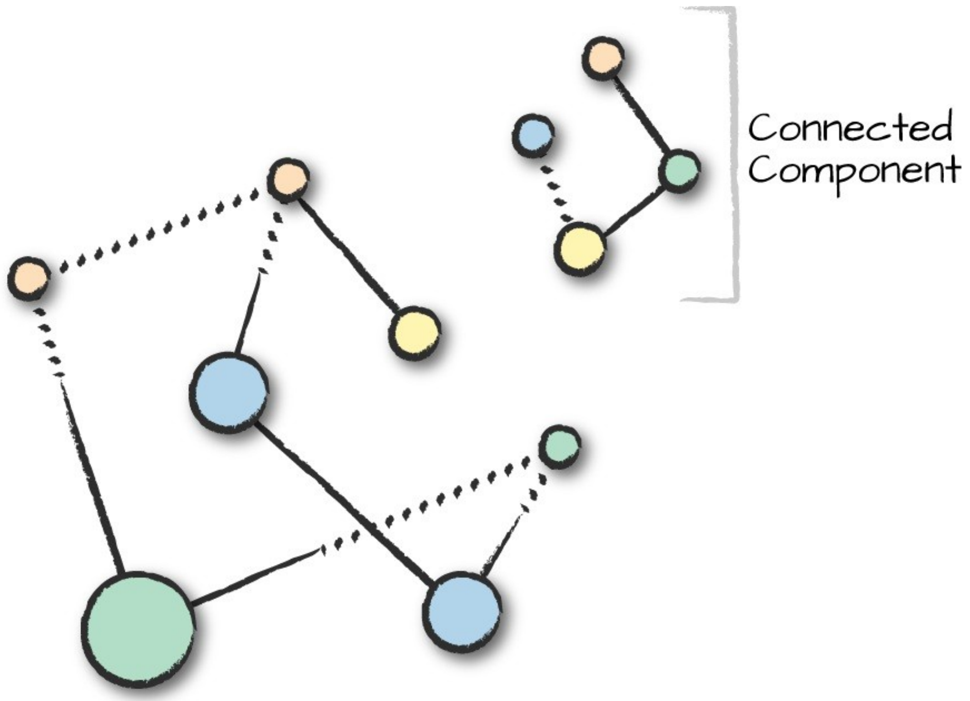
id	name	age	distances
g	Gabby	60	[]
b	Bob	36	[]
e	Esther	32	[d -> 1, a -> 2]
a	Alice	34	[a -> 0, d -> 2]
f	Fanny	36	[]
d	David	29	[d -> 0, a -> 1]
c	Charlie	30	[]

Connected components

- A connected component of a graph is a **subgraph**
- Any two vertices are connected to each other by one or more edges
- The set of vertices is not connected to any additional vertices in the original graph
- Direction of edges is not considered
- Connected components detection can be interesting for clustering, but also to make your computations more efficient

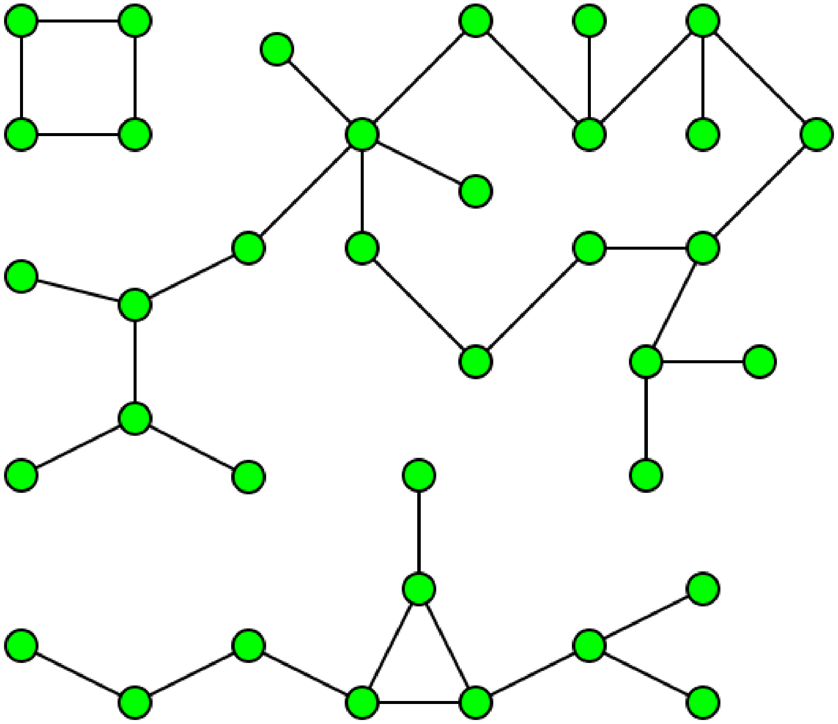
Connected components

- Two connected components



Connected components

- Three connected components



Connected components

- The algorithm to compute the connected components exploit the **breadth-first search**
- For non-distributed graphs, we can compute the components of a graph in linear time (numbers of the vertices and edges)
- A search that begins at some particular vertex v will find the entire component containing v (and no more) before returning.
- To find all the components of a graph, loop through its vertices, starting a new breadth first whenever the loop reaches a vertex not included in a found component

Connected components

- **connectedComponents()** method of a GraphFrame
- It is an expensive algorithm – expect delays
- The default Connected Components algorithm requires setting a Spark **checkpoint** directory
- Parameters:
  - checkpointInterval – checkpoint interval in terms of number of iterations (default: 2)
  - broadcastThreshold – broadcast threshold in propagating component assignments (default: 1000000)
- Returns:
  - DataFrame with new vertices column “component”

Connected components: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Find the connected components of the graph

Connected components: example

```
#set checkpoint folder
sc.setCheckpointDir("tmp_ckpts")

#run the algorithm
connComp=g.connectedComponents()

#show results. Order by component in order to have nodes of the
    same component in adjacent rows
connComp.orderBy("component").show()

nComp=connComp.select("component").distinct().count()
print("Number of connected components: ", nComp)
```

Connected components: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Find the connected components of the graph

```
+---+-------+---+------------+
| id|   name|age|   component|
+---+-------+---+------------+
|  g|  Gabby| 60|146028888064|
|  c|Charlie| 30|412316860416|
|  a|  Alice| 34|412316860416|
|  b|    Bob| 36|412316860416|
|  e| Esther| 32|412316860416|
|  d|  David| 29|412316860416|
|  f|  Fanny| 36|412316860416|
+---+-------+---+------------+
```

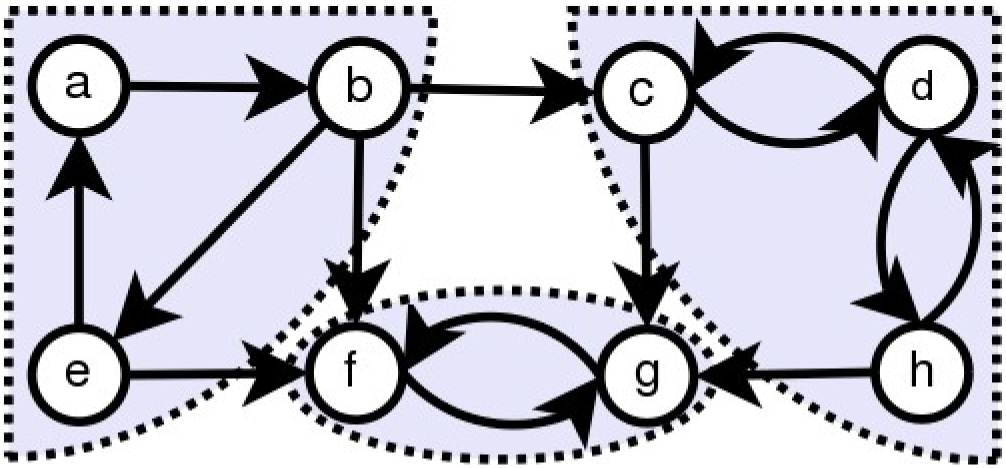
```
Number of connected components:  2
```

Strongly Connected components

- A graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph
- For undirected graph, connected and strongly connected components are the same

Strongly Connected components

- A graph with 3 strongly connected components



Strongly Connected components

- **stronglyConnectedComponents()** method of a GraphFrame
- Requires setting a Spark **checkpoint** directory
- Better to run on a cluster with **yarn** scheduler even with small graphs
- Parameters:
  - maxIter – the number of iterations to run
- Returns:
  - DataFrame with new vertices column “component”

Strongly connected components: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Find the strongly connected components of the graph

Strongly connected components: example

```
#set checkpoint folder
sc.setCheckpointDir("tmp_ckpts")

#run the algorithm
strongConnComp = g.stronglyConnectedComponents(maxIter=10)

#show results. Order by component in order to have nodes of the
    same components in adjacent rows
strongConnComp.orderBy("component").show()

nComp=strongConnComp.select("component").distinct().count()
print("Number of strongly connected components: ", nComp)
```

Strongly connected components: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

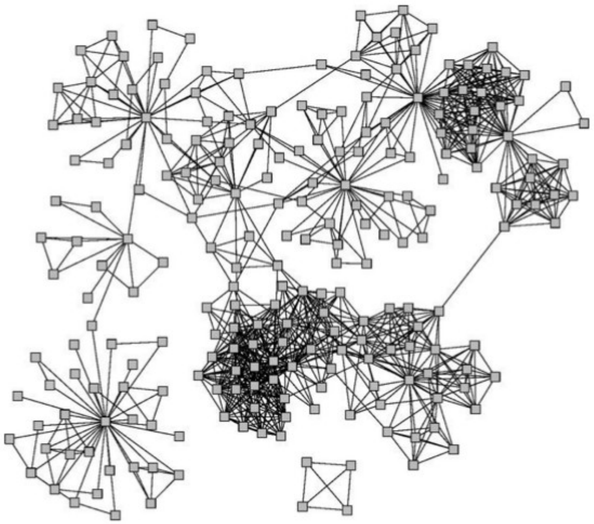
```
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
|  f|  c|      follow|
|  e|  f|      follow|
|  e|  d|      friend|
|  d|  a|      friend|
|  a|  e|      friend|
+---+---+------------+
```

Find the connected components of the graph

```
+---+-------+---+-------------+
| id|   name|age|    component|
+---+-------+---+-------------+
|  g|  Gabby| 60| 146028888064|
|  f|  Fanny| 36| 412316860416|
|  a|  Alice| 34| 670014898176|
|  e| Esther| 32| 670014898176|
|  d|  David| 29| 670014898176|
|  b|    Bob| 36|1047972020224|
|  c|Charlie| 30|1047972020224|
+---+-------+---+-------------+
```

```
Number of strongly connected components:  4
```

Communities in graphs



What makes a community (cohesive subgroup):

- Mutuality of ties. Everyone in the group has ties (edges) to one another
- Compactness. Closeness or reachability of group members in small number of steps, not necessarily adjacency
- Density of edges. High frequency of ties within the group
- Separation. Higher frequency of ties among group members compared to non-members

Communities in graphs

Airline flights

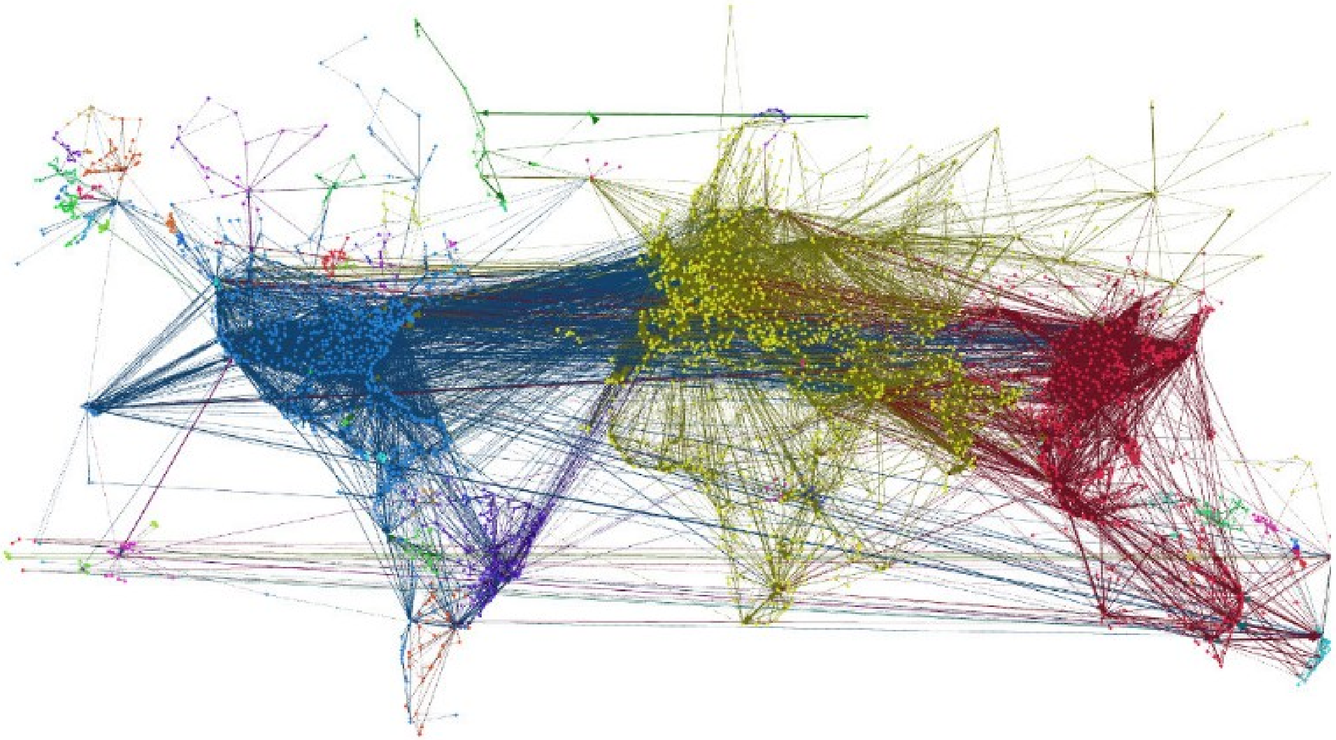


image from Lab41 blog

Label propagation

- Label Propagation an algorithm for **detecting communities in graphs**
- Like clustering but exploiting connectivity
- It is not expensive computationally, but (1) convergence is not guaranteed and (2) one can end up with trivial solutions
- Each node in the network is initially assigned to its own community.
- At every step, nodes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages.

Label propagation

- **labelPropagation()** method of a GraphFrame
- Parameters:
  - maxIter – the number of iterations to run
- Returns:
  - DataFrame with new vertices column “label”

Label propagation: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Detect communities with label propagation algorithm

Label propagation: example

```
result = g.labelPropagation(maxIter=20)

result.select("id", "label").orderBy("label").show()
```

Label propagation: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Detect communities with label propagation algorithm

```
+---+------------+
| id|       label|
+---+------------+
|  g| 146028888064|
|  e|1047972020224|
|  a|1047972020224|
|  c|1047972020224|
|  f|1382979469312|
|  b|1382979469312|
|  d|1382979469312|
+---+------------+
```

Label propagation: example

Nodes DataFrame

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36
g	Gabby	60

Edge DataFrame

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

Detect communities with label propagation algorithm

id	label
g	146028888064
e	1047972020224
a	1047972020224
c	1047972020224
f	1382979469312
b	1382979469312
d	1382979469312

[g] is a community (and a connected components)

Label propagation: example

Nodes DataFrame

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36
g	Gabby	60

Edge DataFrame

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

Detect communities with label propagation algorithm

id	label
g	146028888064
e	1047972020224
a	1047972020224
c	1047972020224
f	1382979469312
b	1382979469312
d	1382979469312

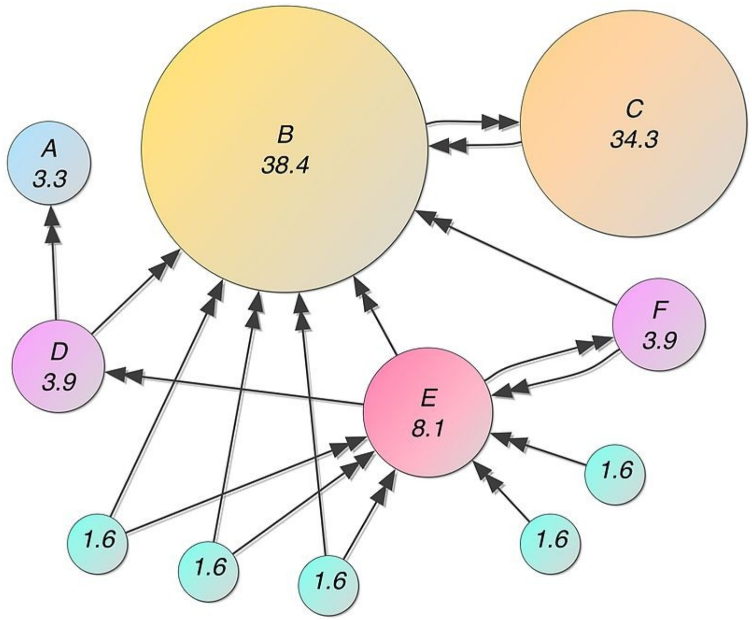
Take care! The algorithm finds two communities [e,a,c] and [f,b,d], but different runs may have different results

PageRank

- PageRank is the original algorithm used by Google Search to rank vertices in a graph by order of importance
- For Google search, nodes are the pages in the World Wide Web, edges are hyperlinks on the pages
- It is part of the broader algorithms to find **centrality** of nodes
- It assigns a numerical weighting to each node

PageRank

- It outputs a likelihood that a person randomly clicking on links will arrive at any particular page
- For a page PageRank, it is important not only how many pages link to it, but also their quality (i.e., their PageRank)



PageRank

- **pageRank()** method of a GraphFrame
- It returns a **GraphFrame** with new vertices column “pagerank” (not normalized) and new edges column “weight”
- Can be run for a fixed number of iterations, by setting **maxIter**
- Can be run until convergence by setting **tol**
- Can be personalized (computing rank with respect to a certain node), by defining **sourceId**

PageRank: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Detect PageRank centrality of nodes

Detect PageRank centrality of nodes personalized with respect to Bob

PageRank: example

```
# Run PageRank until convergence to tolerance "tol".
results = g.pageRank(tol=0.03)

# Display resulting pageranks
results.vertices.show(truncate=False)

# Run PersonalizedPageRank until convergence to tolerance "tol".
resultsPers = g.pageRank(tol=0.03,sourceId="b")

# Display resulting pageranks
resultsPers.vertices.show(truncate=False)
```

PageRank: example

Detect PageRank centrality of nodes

id	name	age	pagerank
g	Gabby	60	0.20481473364387 33
b	Bob	36	2.615150188473630 3
e	Esther	32	0.397295780046129 6
a	Alice	34	0.452896579770014 8
f	Fanny	36	0.36030111878170495
d	David	29	0.36030111878170495
c	Charlie	30	2.60924048050294 2

Detect PageRank centrality of nodes personalized with respect to Bob

id	name	age	pagerank
g	Gabby	60	0.0
b	Bob	36	0.542517589576432
e	Esther	32	0.0
a	Alice	34	0.0
f	Fanny	36	0.0
d	David	29	0.0
c	Charlie	30	0.457482410423568

Custom graph algorithms

- GraphFrames provides primitives for developing yourself other graph algorithms
- It is based on message passing approach
- The two key components are:
  - **aggregateMessages**: Send messages between vertices, and aggregate messages for each vertex.
    https://graphframes.github.io/graphframes/docs/_site/api/python/graphframes.lib.html
  - **joins**: Join message aggregates with the original graph (DataFrame joins)

Custom graph algorithm: example

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

For each user, compute the sum of the ages of adjacent users

Custom graph algorithm: example

```
from pyspark.sql.functions import sum as sqlsum
from graphframes.lib import AggregateMessages

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

Custom graph algorithm: example

Nodes DataFrame

id	name	age
a	Alice	34
b	Bob	36
c	Charlie	30
d	David	29
e	Esther	32
f	Fanny	36
g	Gabby	60

Edge DataFrame

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

For each user, compute the sum of the ages of adjacent users

id	sum(MSG)
f	62
e	99
d	66
c	108
b	94
a	97

Visualization of a graph

Visualize Big Data

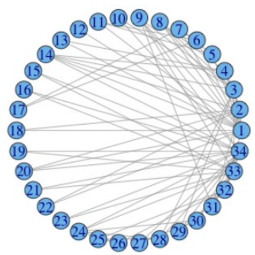
- In general, it is not a good idea to plot Big Data
- After big data processing, often you can plot results → they are no more big data
- You can plot python objects with many libraries, e.g., **matplotlib**
- You can transform spark DataFrame into pandas dataframe: dfPandas = df.**toPandas()**

Visualize graphs

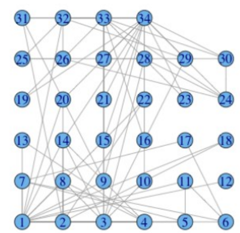
- Why visualize a graph?
  - Get an idea of structure. E.g., A good visualization can show if there are some clusters in a graph
  - To show results of processing. E.g., after performing label propagatin, color nodes by labels
  - Some graph visualization algorithms are also dimension reduction tools (e.g., PCA)

Visualize graphs: layouts

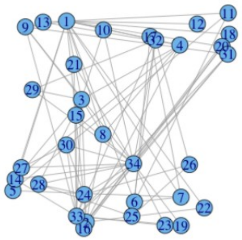
- Layout is a way to map a coordinate to each vertex (usually, on 2D plane)



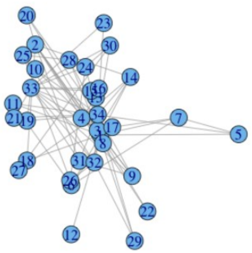
Circular layout



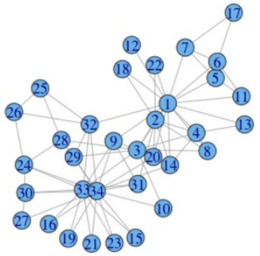
Grid layout



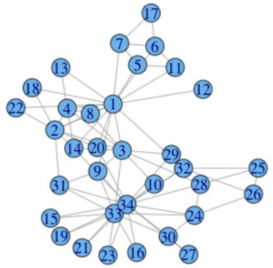
Random layout



Spring based



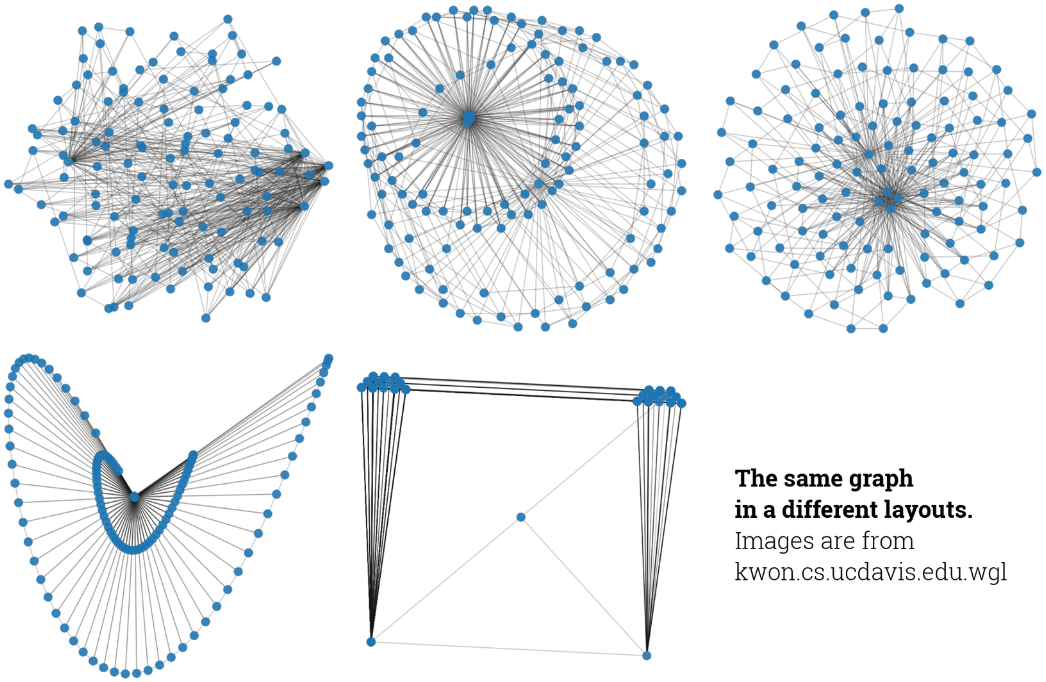
Kamada-Kawai



Frutcherman-Reingold

Visualize graphs: layouts

- Layout is a way to map a coordinate to each vertex (usually, on 2D plane)
- For the same graph, there are different layouts



The same graph in a different layouts.
Images are from kwon.cs.ucdavis.edu.wgl

Visualize graphs

- Problems in visualizing large graphs (>10k vertices and/or edges) :
  - Readability: visualization of a large graph often looks messy because there are too many objects in one plot.
  - Speed: graph visualization algorithms mostly have > quadratic algorithmic complexity (number of edges or vertices). Too long to find good/optimal parameters.

Visualize graphs

Readability issue

Visualize graphs

- There is no native GraphFrame library that visualizes data
- Use python libraries for graph plot
  - **NetworkX**, **graphviz**, **matplotlib**, **pydot**,...
- Use external tools for graph plot
  - **Gephi**, **LargeViz**,...

https://towardsdatascience.com/large-graph-visualization-tools-and-approaches-2b8758a1cd59

Visualize graphs: networkx

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Create a NetworkX graph from the GraphFrame graph and visualize it

Visualize graphs: networkx

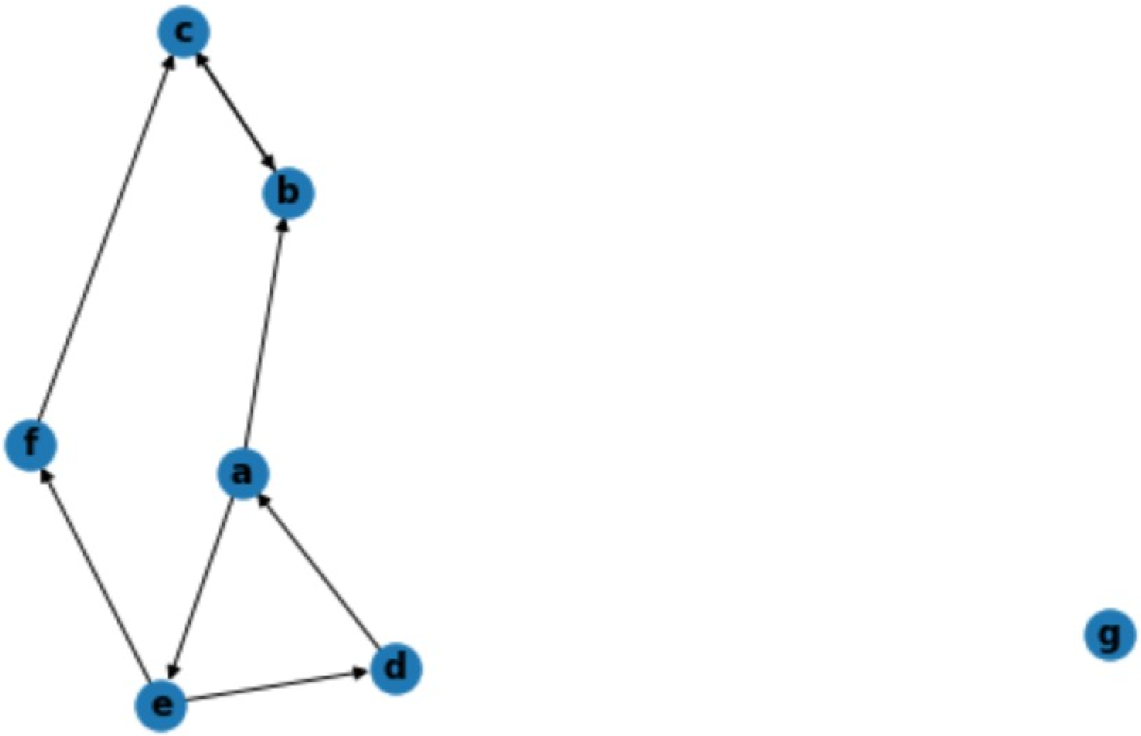
```
import networkx as nx

def xGraph(edge_list,node_list):
    Gplot=nx.DiGraph()
    edges=edge_list.collect()
    nodes=node_list.collect()
    for row in edges:
        Gplot.add_edge(row['src'],row['dst'])
    for row in nodes:
        Gplot.add_node(row['id'])
    return Gplot

Gplot=xGraph(g.edges,g.vertices)
nx.draw(Gplot, with_labels=True, font_weight='bold')
```

Visualize graphs: networkx

Create a NetworkX graph from the GraphFrame graph and visualize it



Visualize graphs: graphviz

Nodes DataFrame

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
|  g|  Gabby| 60|
+---+-------+---+
```

Edge DataFrame

```
+---+---+------------+
|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|
+---+---+------------+
```

Create a graphviz graph from the GraphFrame graph and visualize it

Visualize graphs: graphviz

```
from graphviz import Digraph

def vizGraph(edge_list,node_list):
    Gplot=Digraph()
    edges=edge_list.collect()
    nodes=node_list.collect()
    for row in edges:
        Gplot.edge(row['src'],row['dst'],label=row['relationship'])
    for row in nodes:
        Gplot.node(row['id'],label=row['name'])
    return Gplot

Gplot=vizGraph(g.edges,g.vertices)
Gplot
```

Visualize graphs: graphviz

Create a graphviz graph from the GraphFrame graph and visualize it