Graph Analytics in Spark

Graph Algorithms with GraphFrames

Algorithms over graphs

- GraphFrame provides the parallel implementation of a set of state of the art algorithms for graph analytics
  - Breadth first search
  - Shortest paths
  - Connected components
  - Strongly connected component
  - Label propagation
  - PageRank
  - ...
- Custom algorithms can be designed and implemented

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 SparkContext object
    - Retrieve it from a SparkSession by using spark.sparkContext

Breadth first search

- Breadth-first search (BFS) is an algorithm for traversing/searching graph data structures
  - It finds the shortest path(s) from one vertex (or a set of vertexes) to another vertex (or a set of vertexes.
  - It is used in many other algorithms
    - Length of shortest paths
    - Connected components
    - …

Breadth first search

- The **bfs(fromExpr, toExpr, edgeFilter=None maxPathLength=10)** method of the GraphFrame class returns the **shortest path(s)** from the vertexes matching expression **fromExpr** expression to vertexes matching expression **toExpr**
  - If there are many vertexes matching **fromExpr** and **toExpr**, only the couple(s) with the shortest length is returned

Breadth first search

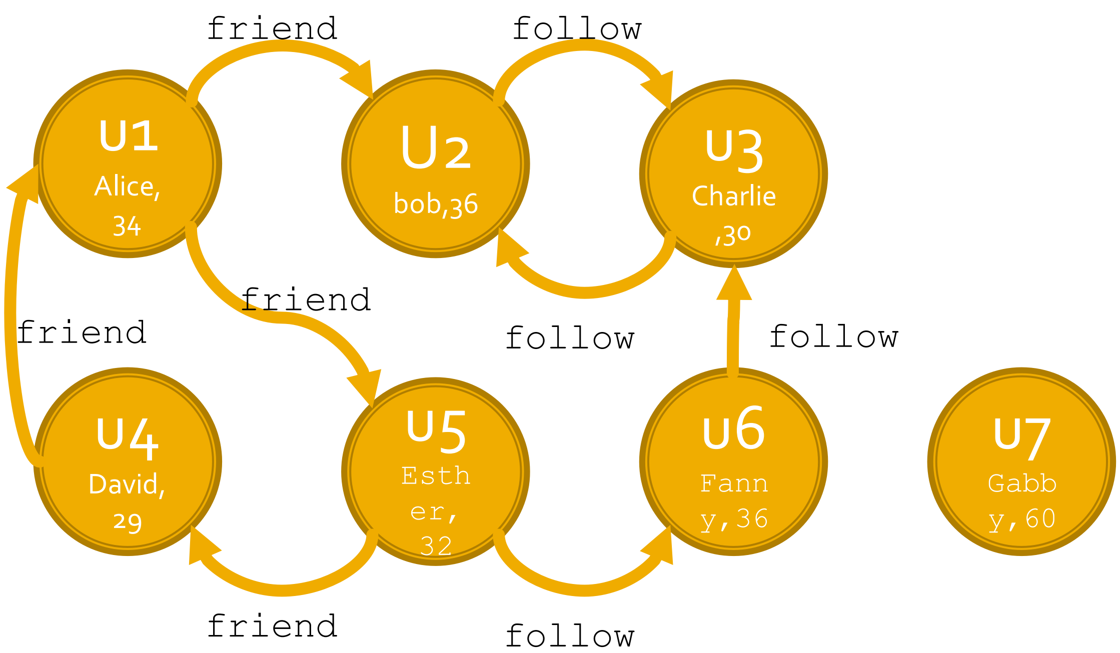
- **fromExpr**: Spark SQL expression specifying valid starting vertexes for the execution of the BFS algorithm
  - E.g., to start from a specific vertex
    - "id = [start vertex id]"
- **toExpr**: Spark SQL expression specifying valid target vertexes for the BFS algorithm
- **maxPathLength**: Limit on the length of paths (default = 10)
- **edgeFilter**: Spark SQL expression specifying edges that may be used in the search (default None)

Breadth first search

- bfs() returns a DataFrame containing the selected shortest path(s)
  - If multiple paths are valid and their length is equal to the shortest length, the returned DataFrame will contain one Row for each path
  - The number of columns of the returned DataFrame is equal to
    - (length of the shortest path*2)+1

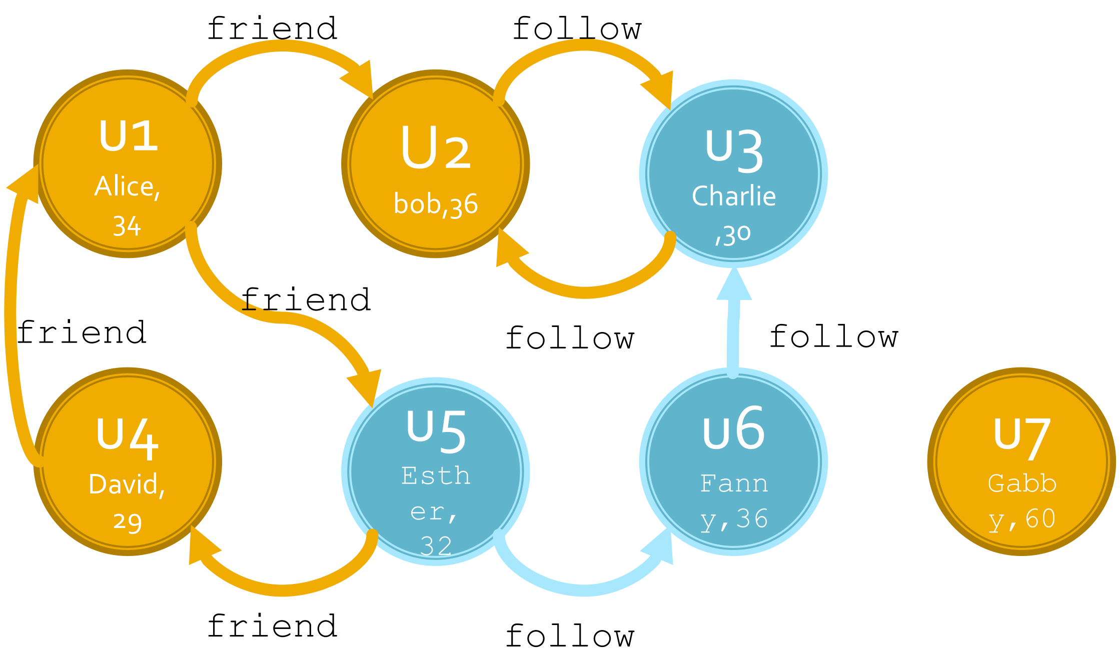
Breadth first search: Example 1

- Find the shortest path from Esther to Charlie
- Store the result in a DataFrame



Breadth first search: Example 1

- Find the shortest path from Esther to Charlie
- Store the result in a DataFrame



Breadth first search: Example 1

- Find the shortest path from Esther to Charlie
- Store the result in a DataFrame

Content of the returned DataFrame

from	e0	v1	e1	to
[u5, Esther, 32]	[u5, u6, follow]	[u6, Fanny, 36]	[u6, u3, follow]	[u3, Charlie, 30]

Breadth first search: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                          ["id", "name", "age"])
```

Breadth first search: Example 1

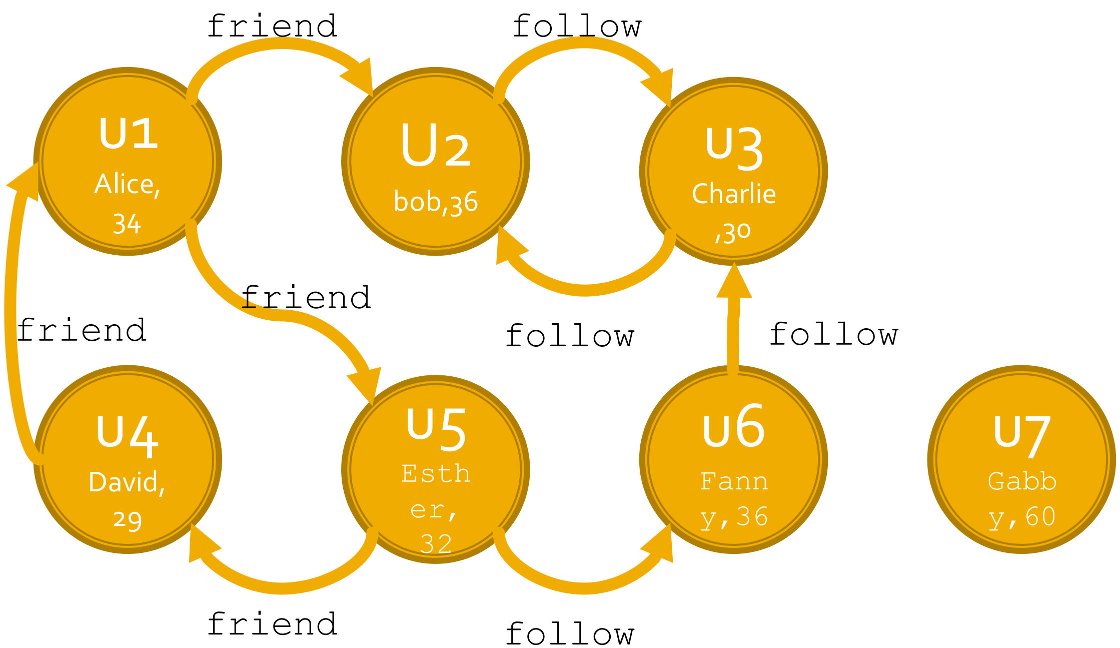
```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Breadth first search: Example 1

```
# Search from vertex with name = "Esther" to vertex with name = "Charlie"

shortestPaths = g.bfs("name = 'Esther' ", "name = 'Charlie' ")
```

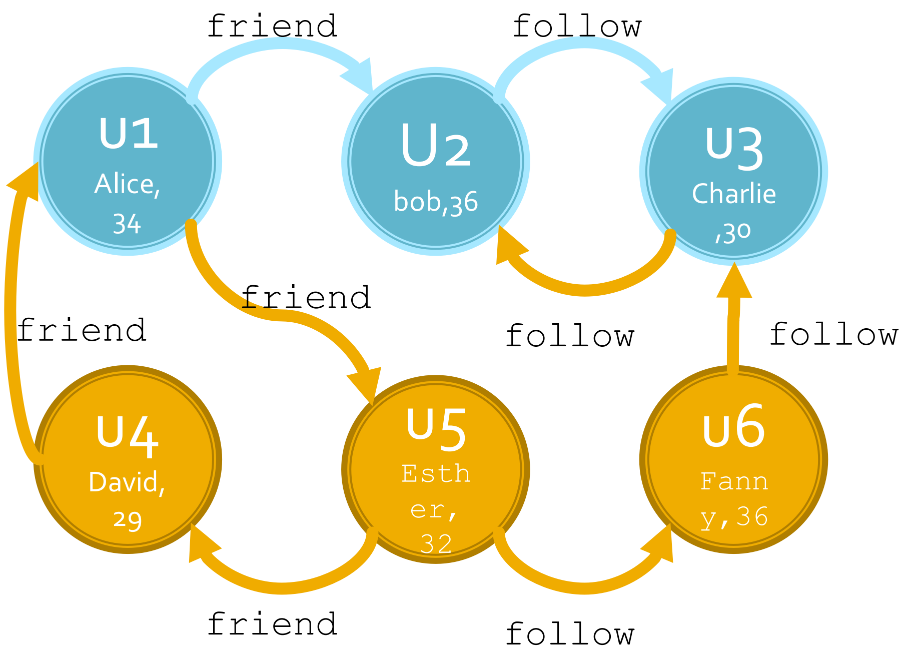
Breadth first search: Example 2

- Find the shortest path from Alice to a user who is 30 years old
- Store the result in a DataFrame



Breadth first search: Example 2

- Find the shortest path from Alice to a user who is 30 years old
- Store the result in a DataFrame



Breadth first search: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

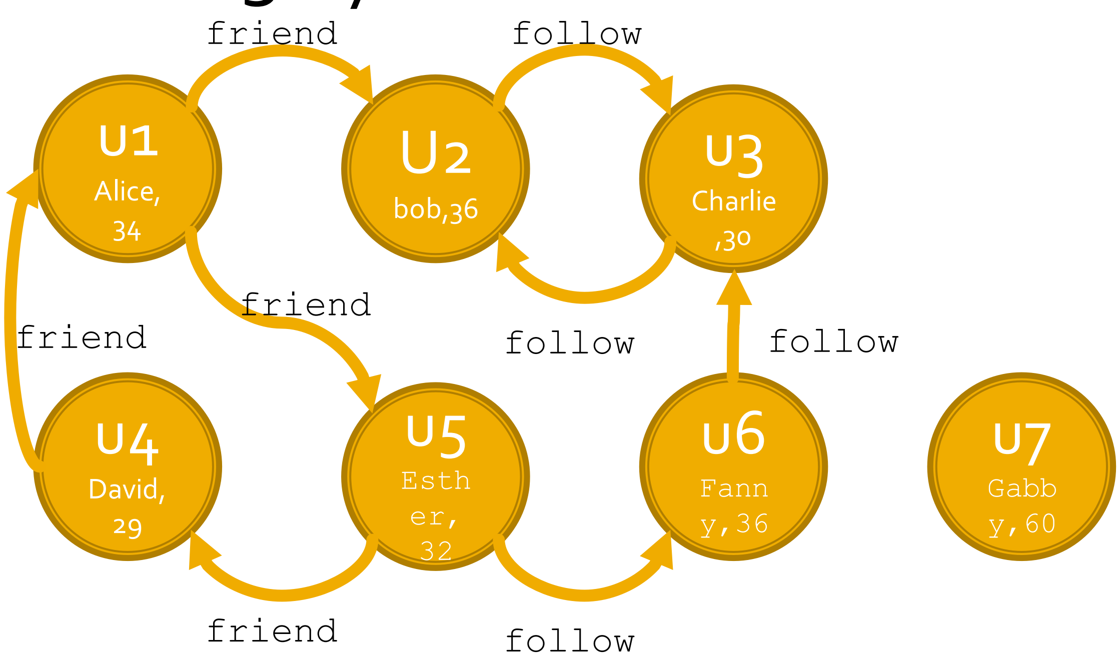
Breadth first search: Example 2

Find the shortest path from Alice to a user who is 30 years old

```
shortestPaths = g.bfs("name = 'Alice' ", "age= 30")
```

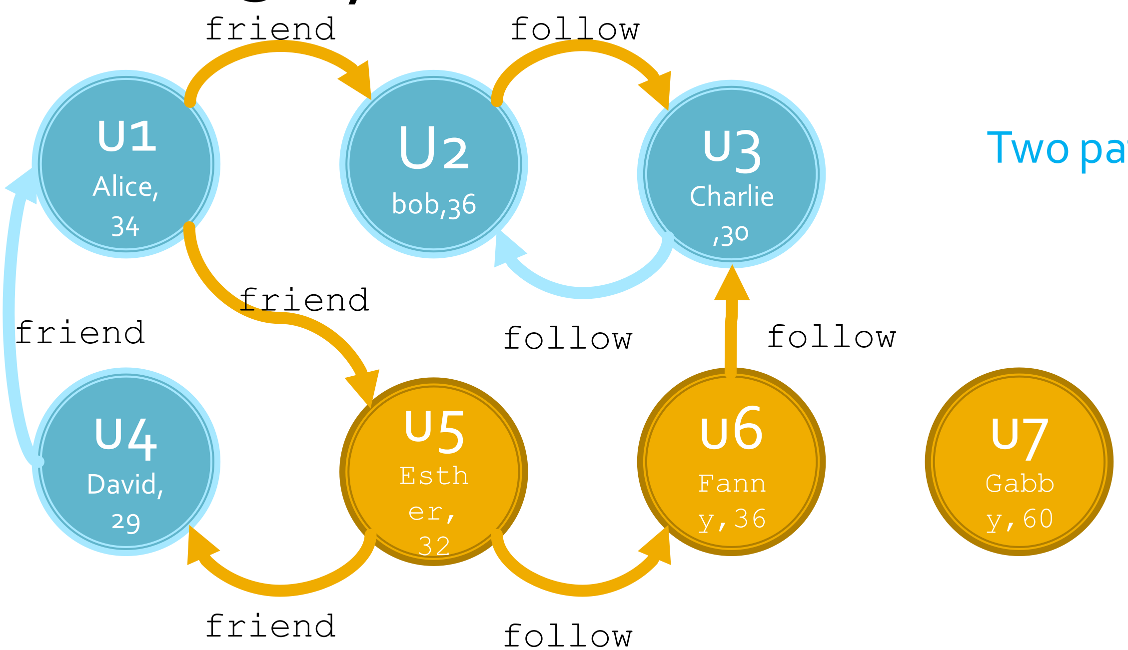
Breadth first search: Example 3

- Find the shortest path from any user who is less than 31 years old to any user who is more than 30 years old



Breadth first search: Example 3

- Find the shortest path from any user who is less than 31 years old to any user who is more than 30 years old



Two paths are selected in this case

Breadth first search: Example 3

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 3

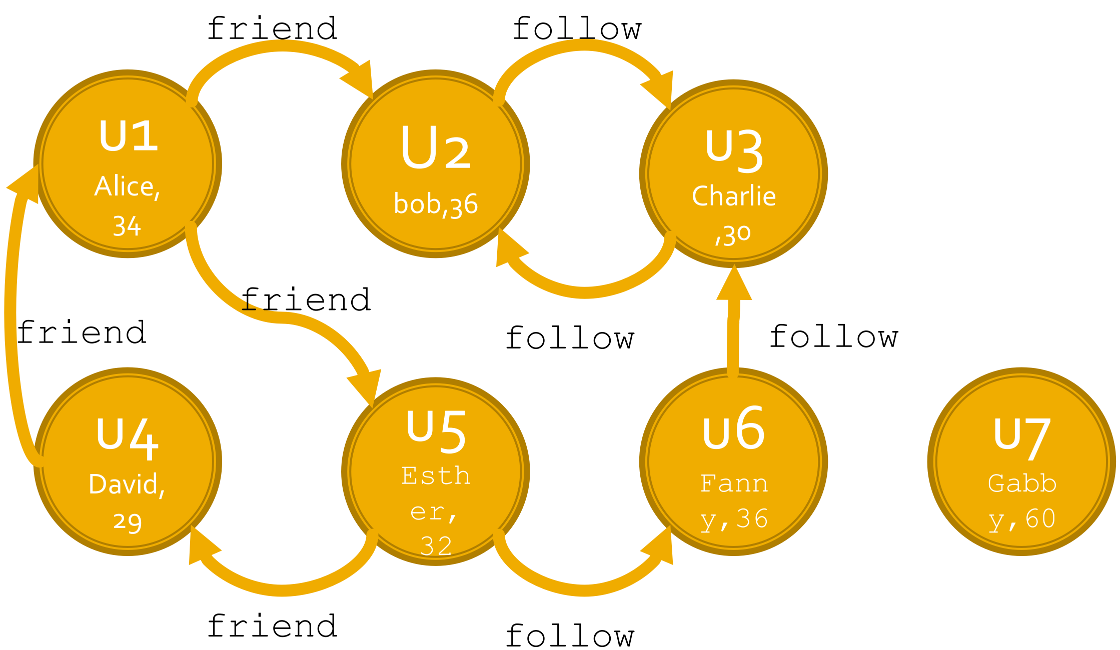
```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Breadth first search: Example 3

```
# Find the shortest path from any user who is less than 31 years old
#  to any user who is more than 30 years old

shortestPaths = g.bfs("age<31", "age>30")
```

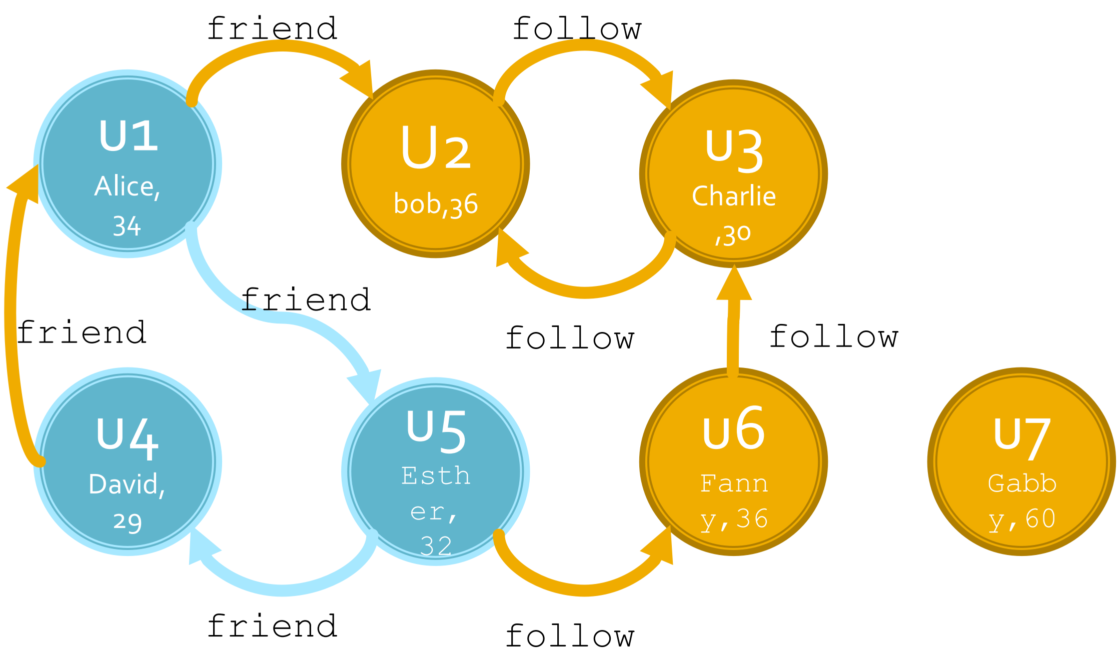
Breadth first search: Example 4

- Find the shortest path from Alice to any user who is less than 31 years old without using “follow” edges



Breadth first search: Example 4

- Find the shortest path from Alice to any user who is less than 31 years old without using "follow" edges



Breadth first search: Example 4

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Breadth first search: Example 4

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Breadth first search: Example 4

```
# Find the shortest path from Alice to any user who is less
# than 31 years old without using "follow" edges

shortestPaths = g.bfs("name = 'Alice' ", "age<31", "relationship<> 'follow'  ")
```

Shortest path

- The shortest path method selects the length of the shortest path(s) from each vertex to a given set of landmark vertexes
  - It uses the BFS algorithm for computing the shortest paths

Shortest path

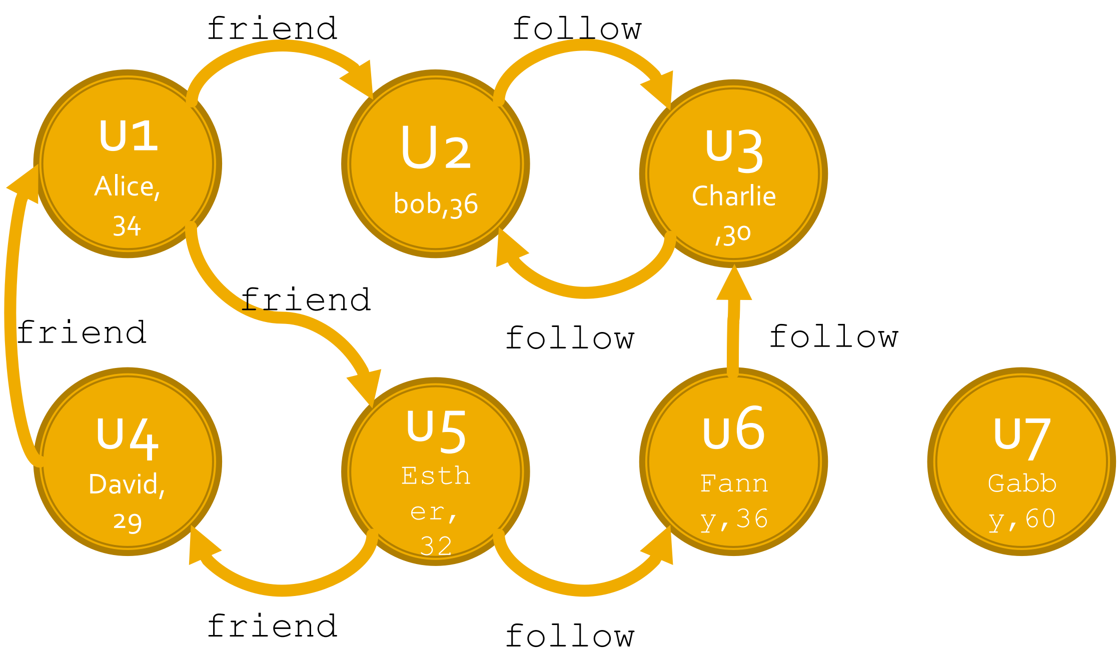
- The **shortestPaths(landmarks)** method of the GraphFrame class returns the **length of the shortest path(s)** from each vertex to a given set of **landmarks** vertexes
  - For each vertex, one shortest path for each landmark vertex is computed and its length is returned
  - **landmarks**: list of IDs of landmark vertexes
    - E.g., [‘u1’, ‘u4’]

Shortest path

- **shortestPaths()** returns a DataFrame that
  - Contains one record/Row for each distinct vertex of the input graph
    - Also for the non-connected ones
  - Is characterized by the following columns
    - One column for each attribute of the vertexes
    - distances (type map)
      - For each landmark **lm** it contains one pair (ID **lm**: length shortest path from the vertex of the current record to **lm**)

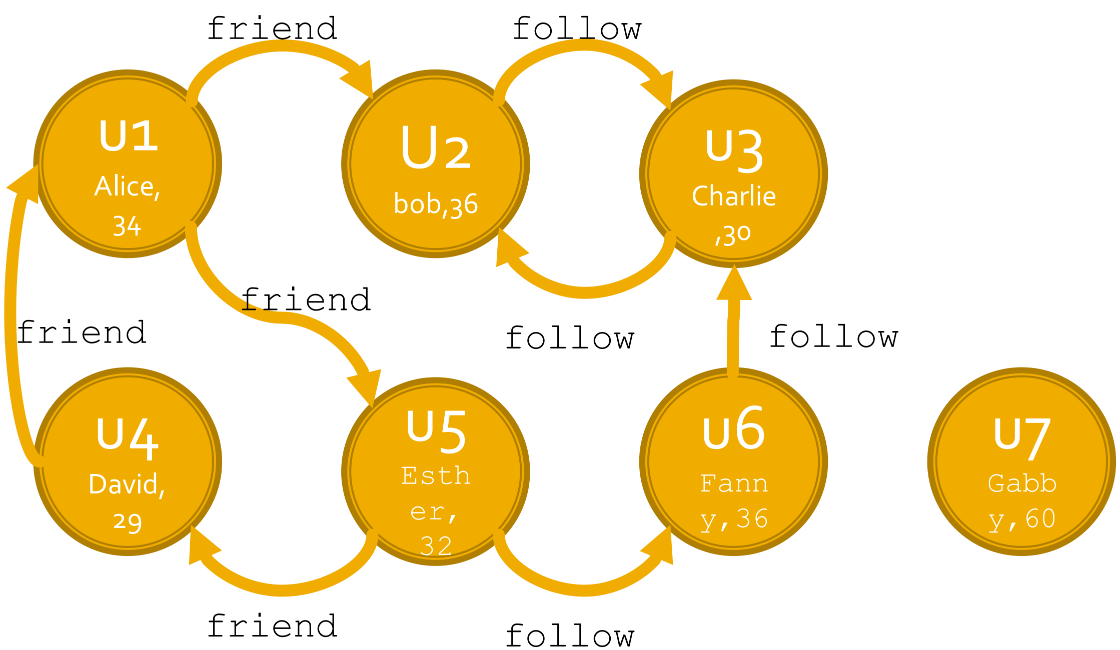
Shortest path: Example 1

- Find for each user the length of the shortest path to user u1 (i.e., Alice)



Shortest path: Example 1

- Find for each user the length of the shortest path to user u1 (i.e., Alice)



Vertex	Distance to u1
U1	0
U2	-
U3	-
U4	1
U5	2
u6	-
U7	-

Shortest path: Example 1

- Find for each user the length of the shortest path to user u1 (i.e., Alice)

Content of the returned DataFrame

id	name	age	distances
u1	Alice	34	[u1 -> 0]
u2	Bob	36	[]
u3	Charlie	30	[]
u4	David	29	[u1 -> 1]
u5	Esther	32	[u1 -> 2]
u6	Fanny	36	[]
u7	Gabby	60	[]

Shortest path: Example 1

- Find for each user the length of the shortest path to user u1 (i.e., Alice)

Content of the returned DataFrame

```
+---+--------+-----+------------+
|id |  name  | age | distances  |
+---+--------+-----+------------+
|u1 | Alice  |34   | [u1 -> 0]  |
|u2 |  Bob   |36   |     []     |
|u3 |Charlie |30   |     []     |
|u4 | David  |29   | [u1 -> 1]  |
|u5 |Esther  |32   | [u1 -> 2]  |
|u6 | Fanny  |36   |     []     |
|u7 | Gabby  |60   |     []     |
+---+--------+-----+------------+
```

Data type: map
- It stores a set of pairs (Key: Value)

Shortest path: Example 1

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Shortest path: Example 1

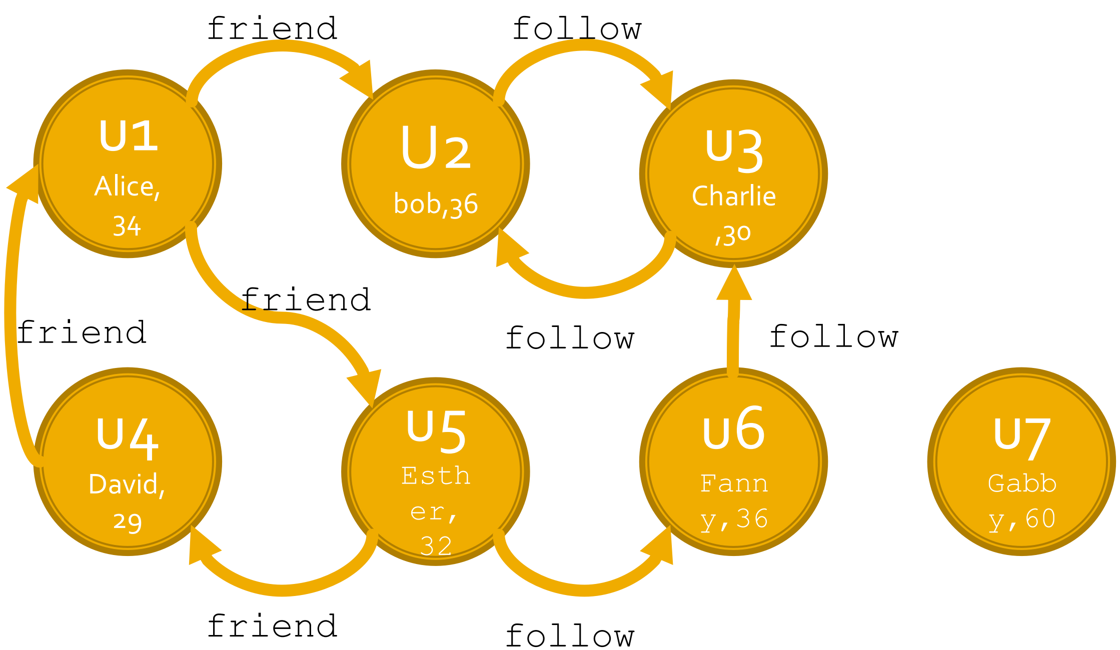
```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Shortest path: Example 1

```
# Find for each user the length of the shortest path to user u1

shortestPaths = g.shortestPaths(["u1"])
```

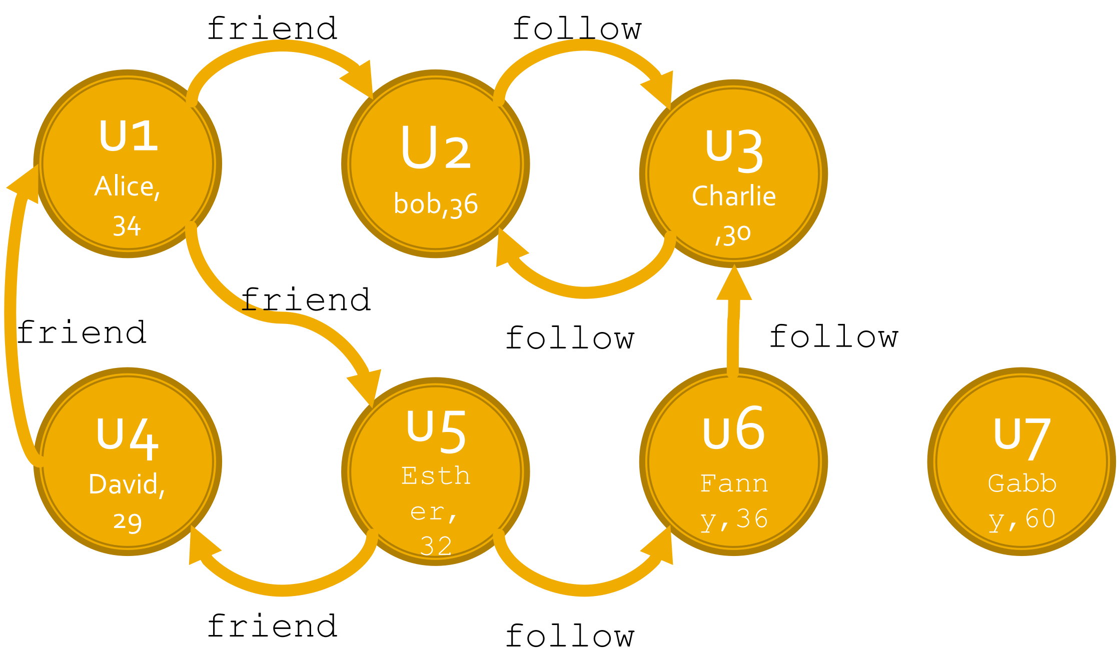
Shortest path: Example 2

- Find for each user the length of the shortest path to users u1 (Alice) and u4 (David)



Shortest path: Example 2

- Find for each user the length of the shortest path to users u1 (Alice) and u4 (David)



Vertex	Distance to u1	Distance to u4
U1	0	2
U2	-	-
U3	-	-
U4	1	0
U5	2	1
u6	-	-
U7	-	-

Shortest path: Example 2

- Find for each user the length of the shortest path to users u1 (Alice) and u4 (David)

  Content of the returned DataFrame

```
+----+--------+-----+-----------------------+
|id  |  name  |age  |distances              |
+----+--------+-----+-----------------------+
|u1  | Alice  |34   |[u1 -> 0, u4 -> 2]     |
|u2  |  Bob   |36   |      []               |
|u3  |Charlie |30   |      []               |
|u4  | David  |29   |[u1 -> 1, u4 -> 0]     |
|u5  |Esther  |32   |[u1 -> 2, u4 -> 1]     |
|u6  | Fanny  |36   |      []               |
|u7  | Gabby  |60   |      []               |
+----+--------+-----+-----------------------+
```

Shortest path: Example 2

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Shortest path: Example 2

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Shortest path: Example 2

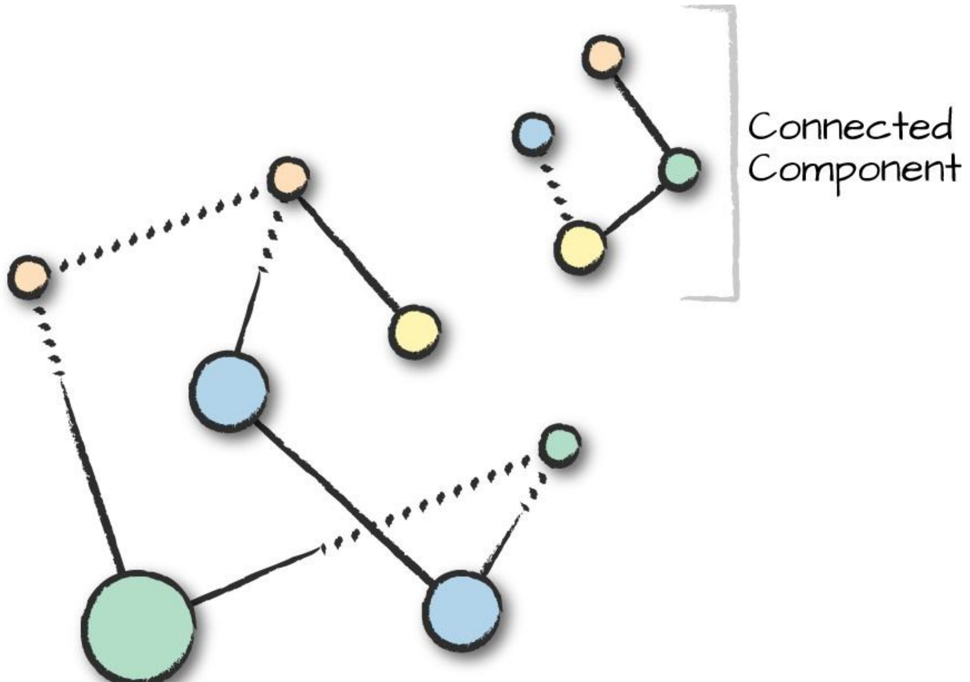
```
# Find for each user the length of the shortest paths to users u1 and u4

shortestPaths = g.shortestPaths(["u1", "u4"])
```

Connected components

- A connected component of a graph is a subgraph **sg** such that
  - Any two vertexes in **sg** are connected to each other by at least one path
  - The set of vertexes in **sg** is not connected to any additional vertexes in the original graph
  - Direction of edges is not considered

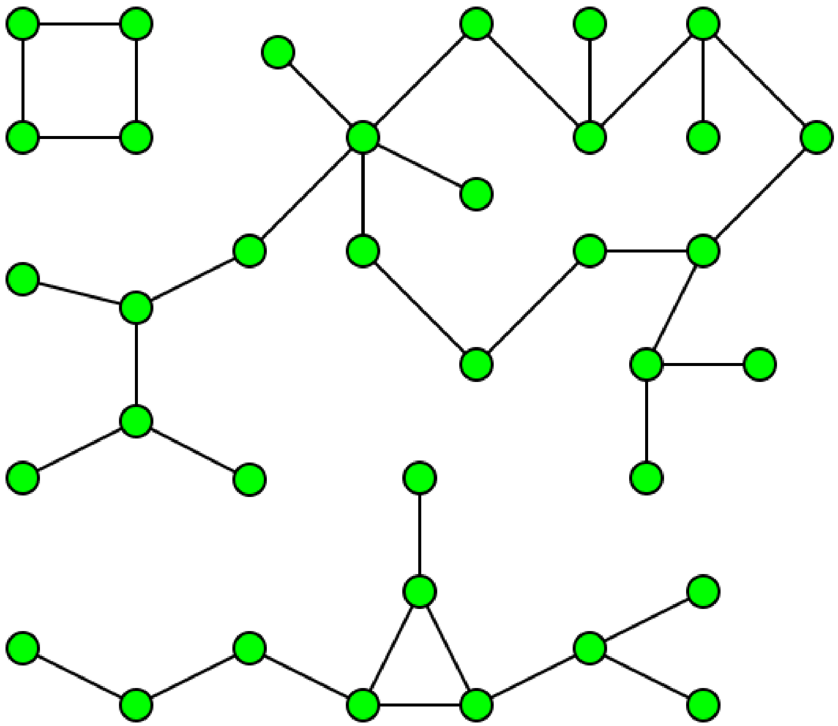
Connected components

- Two connected components



Connected components

- Three connected components



Connected components

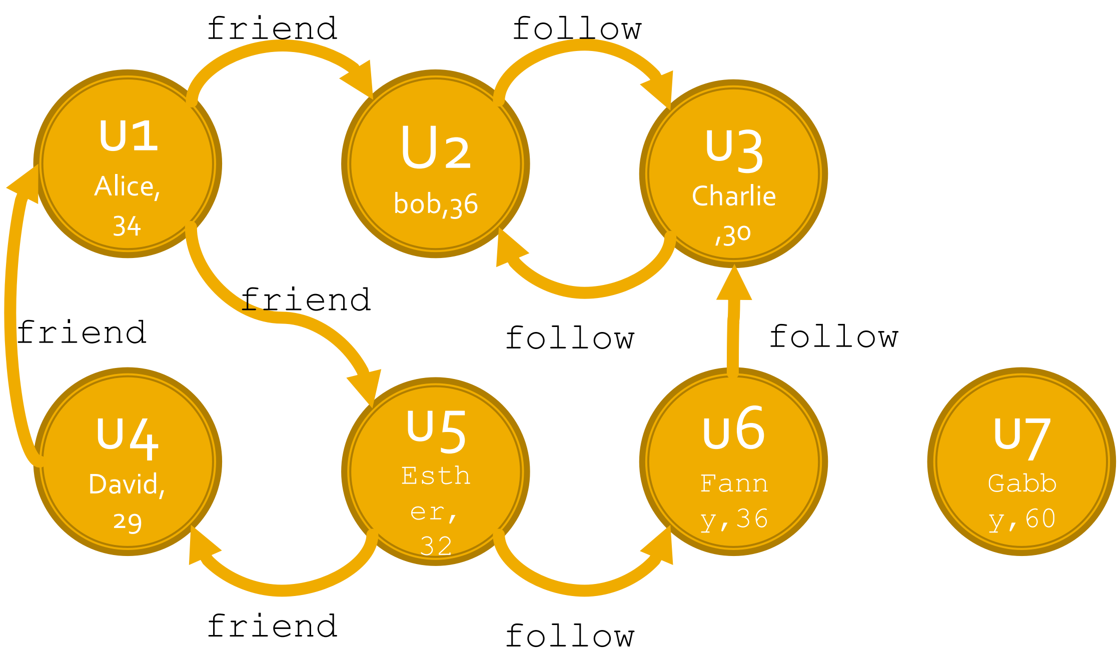
- The **connectedComponents()** method of the GraphFrame class returns the connected components of the input graph
  - It is an expensive algorithm
  - It requires setting a Spark checkpoint directory

Connected components

- connectedComponents() returns a DataFrame that
  - Contains one record/Row for each distinct vertex of the input graph
  - Is characterized by the following columns
    - One column for each attribute of the vertexes
    - component (type long)
      - It is the identifier of the connected component to which the current vertex has been assigned

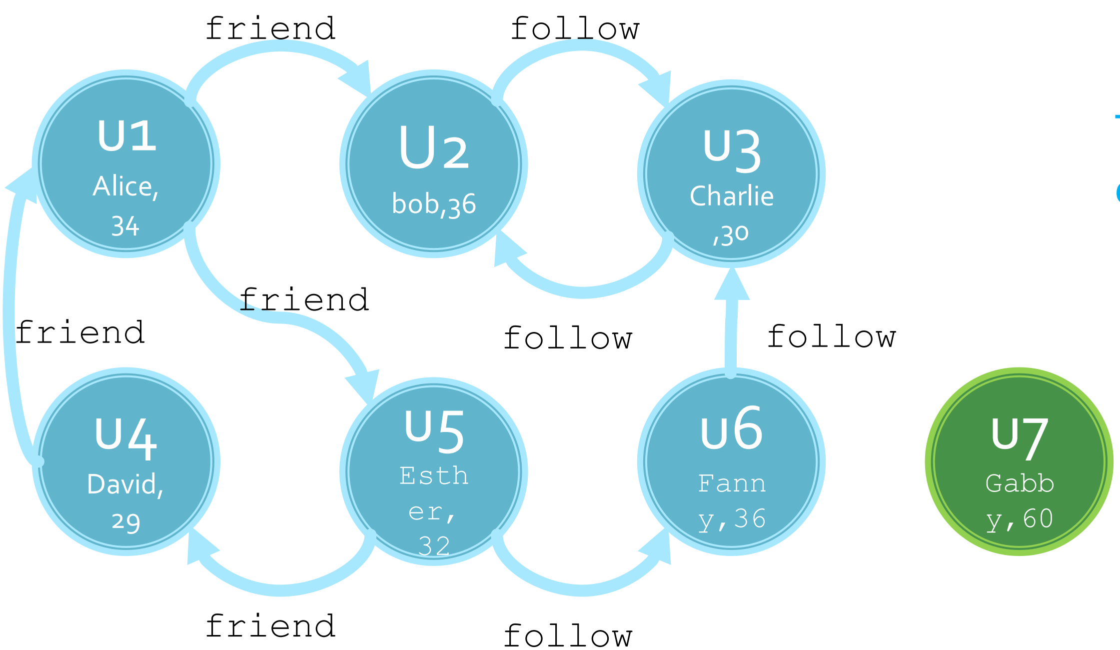
Connected components: Example

- Print on the stdout the number of connected components of the following graph



Connected components: Example

- Print on the stdout the number of connected components of the following graph



The are two connected components on this graph

Connected components: Example

- Print on the stdout the number of connected components of the following graph

  Content of the DataFrame used to store the two identified connected components

id	name	age	component
u6	Fanny	36	146028888064
u1	Alice	34	146028888064
u3	Charlie	30	146028888064
u5	Esther	32	146028888064
u2	Bob	36	146028888064
u4	David	29	146028888064
u7	Gabby	60	1546188226560

Connected components: Example

- Print on the stdout the number of connected components of the following graph

Content of the DataFrame used to store the two identified connected components

```
+---+---------+----+-------------------+
|id |  name   |age |    component      |
+---+---------+----+-------------------+
|u6 | Fanny   |36  |146028888064 |
|u1 | Alice   |34  |146028888064 |
|u3 |Charlie  |30  |146028888064 |
|u5 |Esther   |32  |146028888064 |
|u2 |  Bob    |36  |146028888064 |
|u4 | David   |29  |146028888064 |
|u7 | Gabby   |60  |1546188226560 |
+---+---------+----+-------------------+
```

Vertexes of the first component

Vertexes of the second component

Connected components: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                          ["id", "name", "age"])
```

Connected components: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Connected components: Example

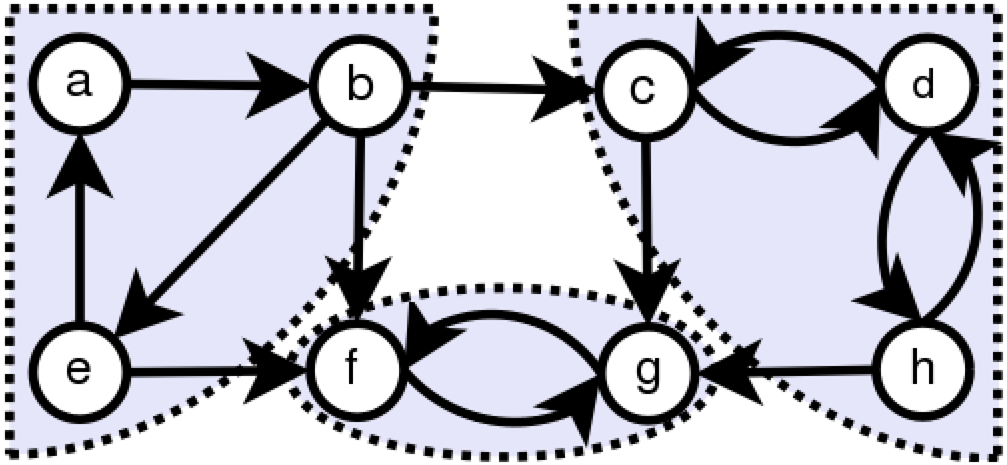
```
# Set checkpoint folder
sc.setCheckpointDir("tmp_ckpts")

# Run the algorithm
connComp=g.connectedComponents()

# Count the number of components
nComp=connComp.select("component").distinct().count()

print("Number of connected components: ", nComp)
```

Strongly Connected components

- A directed subgraph **sg** is called strongly connected if every vertex in **sg** is reachable from every other vertex in **sg**
  - For undirected graph, connected and strongly connected components are the same

Strongly Connected components

- A graph with 3 strongly connected subgraphs/components



Strongly Connected components

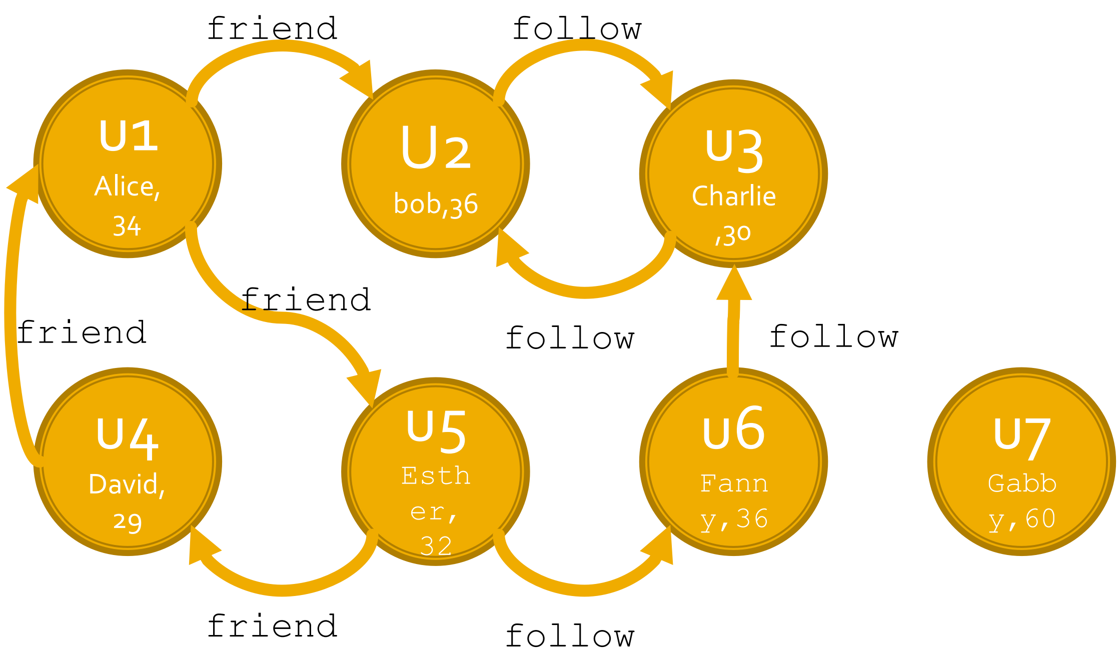
- The **stronglyConnectedComponents()** method of the GraphFrame class returns the strongly connected components of the input graph
  - It is an expensive algorithm
    - Better to run on a cluster with yarn scheduler even with small graphs
  - It requires setting a Spark checkpoint directory

Strongly Connected components

- **stronglyConnectedComponents()** returns a DataFrame that
  - Contains one record/Row for each distinct vertex of the input graph
  - Is characterized by the following columns
    - One column for each attribute of the vertexes
    - component (type long)
      - It is the identifier of the strongly connected component to which the current vertex has been assigned

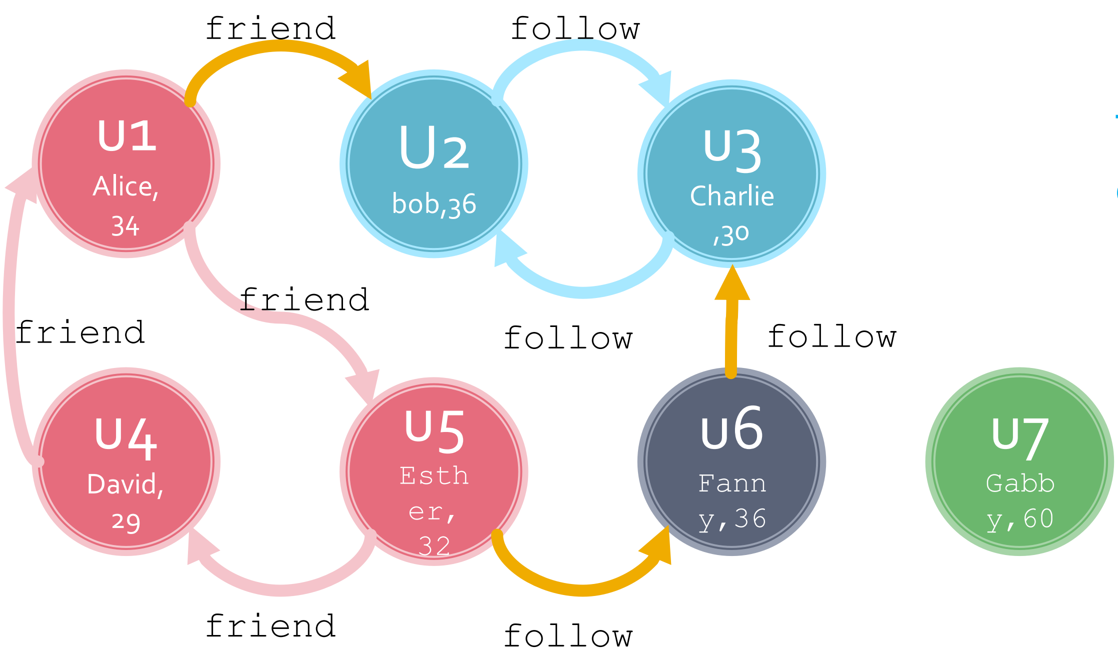
Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph



Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph



The are four connected components on this graph

Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph

  Content of the DataFrame used to store the identified strongly connected components

id	name	age	component
u3	Charlie	30	146028888064
u2	Bob	36	146028888064
u1	Alice	34	498216206336
u5	Esther	32	498216206336
u4	David	29	498216206336
u6	Fanny	36	1090921693184
u7	Gabby	60	1546188226560

Strongly Connected components: Example

- Print on the stdout the number of strongly connected components of the input graph

Content of the DataFrame used to store the identified strongly connected components

```
+---+--------+----+------------------+
|id |  name  |age |    component     |
+---+--------+----+------------------+
|u3 |Charlie |30  |146028888064 |
|u2 |  Bob   |36  |146028888064 |
|u1 | Alice  |34  |498216206336 |
|u5 |Esther  |32  |498216206336 |
|u4 | David  |29  |498216206336 |
|u6 | Fanny  |36  |1090921693184|
|u7 | Gabby  |60  |1546188226560|
+---+--------+----+------------------+
```

Vertexes of the first SCC

Vertexes of the second SCC

Vertexes of the third component

Vertexes of the fourth component

Strongly Connected components: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Strongly Connected components: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Strongly Connected components: Example

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

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

# Count the number of strongly connected components
nComp=strongConnComp.select("component").distinct().count()

print("Number of strongly connected components: ", nComp)
```

Label propagation

- Label Propagation is an algorithm for detecting communities in graphs
  - Like clustering but exploiting connectivity
  - **Convergence is not guaranteed**
  - One can end up with trivial solutions

Label propagation

- The Label Propagation algorithm
  - Each vertex in the network is initially assigned to its own community
  - At every step, vertexes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages

Label propagation

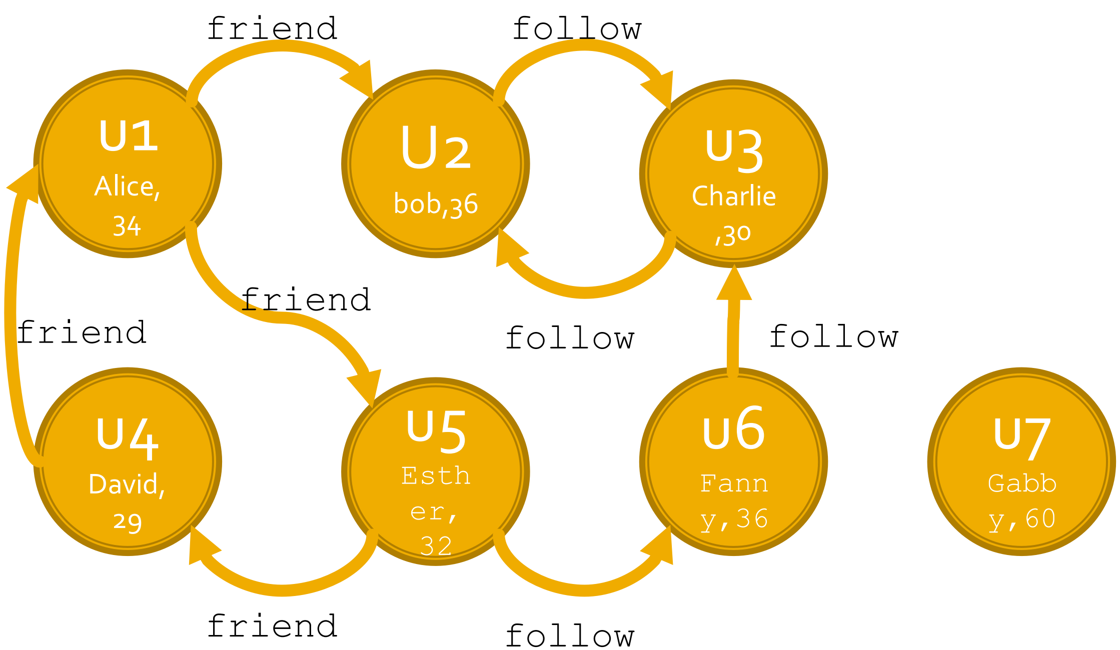
- The **labelPropagation(maxIter)** method of the GraphFrame class runs and returns the result of the label propagation algorithm
  - Parameter maxIter:
    - The number of iterations to run

Label propagation

- labelPropagation() returns a DataFrame that
  - Contains one record/Row for each distinct vertex of the input graph
  - Is characterized by the following columns
    - One column for each attribute of the vertexes
    - label (type long)
      - It is the identifier of the community to which the current vertex has been assigned

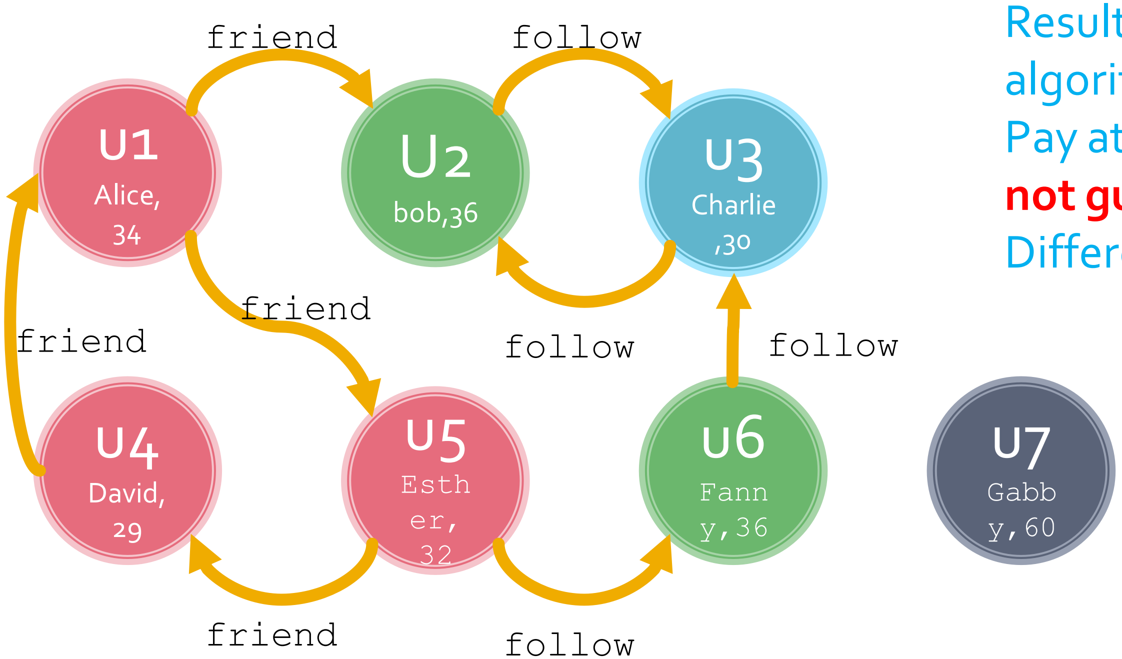
Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm



Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm



Result returned by one run of the algorithm.
Pay attention that **convergence is not guarantee**.
Different results for different runs.

Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm

Content of the DataFrame used to store the identified communities

```
+---+---------+----+-------------------+
|id |  name   |age |   label           |
+---+---------+----+-------------------+
|u3 |Charlie  |30|   146028888064 |
|u4 | David   |29  | 498216206336 |
|u1 | Alice   |34  | 498216206336 |
|u5 | Esther  |32  | 498216206337  |
|u7 | Gabby   |60  |1546188226560|
|u2 |   Bob   |36  |1606317768704 |
|u6 | Fanny   |36  |1606317768704 |
+---+---------+----+-------------------+
```

Label propagation: Example

- Split in groups the vertexes of the graph by using the label propagation algorithm

Content of the DataFrame used to store the identified communities

id	name	age	label
u3	Charlie	30	146028888064
u4	David	29	498216206336
u1	Alice	34	498216206336
u5	Esther	32	498216206337
u7	Gabby	60	1546188226560
u2	Bob	36	1606317768704
u6	Fanny	36	1606317768704

Vertexes of the first community

Vertexes of the second community

Vertexes of the third community

Vertexes of the fourth community

Label propagation: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Label propagation: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Label propagation: Example

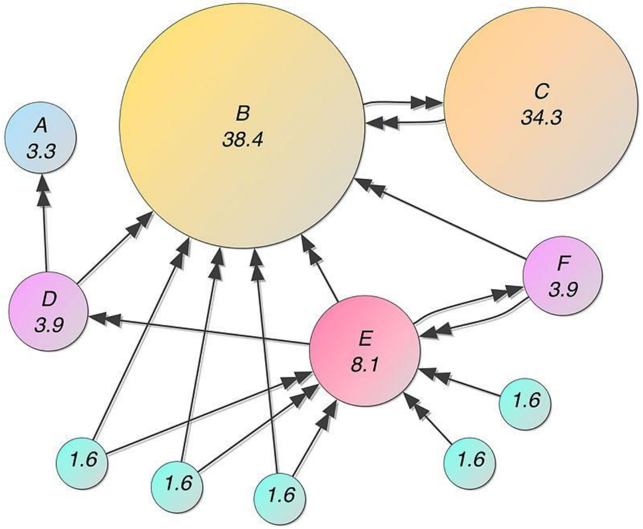
```
# Run the label propagation algorithm
labelComm = g.labelPropagation(10)
```

PageRank

- PageRank is the original famous algorithm used by the Google Search engine to rank vertexes (web pages) in a graph by order of importance
  - For the Google search engine
    - Vertexes are web pages in the World Wide Web,
    - Edges are hyperlinks among web pages
  - It assigns a numerical weighting (importance) to each node

PageRank

- It computes a likelihood that a person randomly clicking on links will arrive at any particular web page
- For a high PageRank, it is important to
  - Have many in-links
  - Be liked by relevant pages (pages characterized by a high PageRank)



PageRank

- Basic idea
  - Each link's vote is proportional to the importance of its source page **p**
  - If page **p** with importance **PageRank(p)** has **n** out-links, each out-link gets **PageRank(p)/n** votes
  - Page **p**'s importance is the sum of the votes on its in-links

PageRank: Simple recursive formulation

1. # Initialize each page's rank to 1.0
   For each p in pages set PageRank(p) to 1.0
2. Iterate for max iterations
   a. Page p sends a contribution PageRank(p)/numOutLinks(p) to its neighbors (the pages it links)
   b. Update each page's rank PageRank(p) to sum(received contributions)
   c. Go to step 2

PageRank with Random jumps

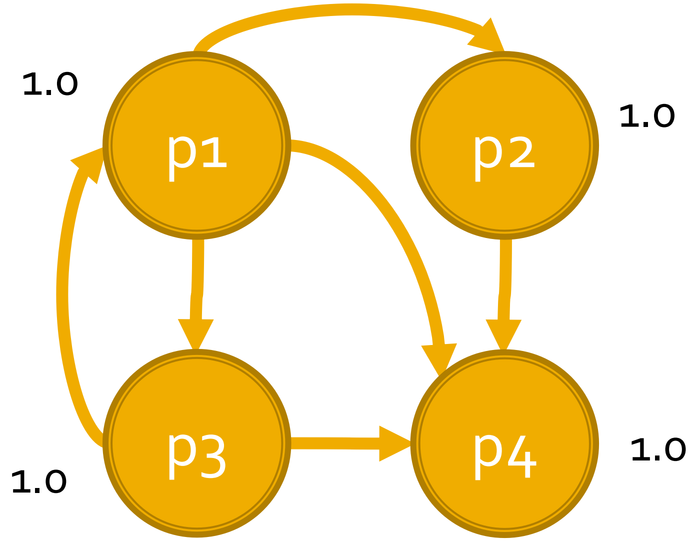
- The PageRank algorithm simulates the random walk of a user on the web
- At each step of the random walk, the random surfer has two options:
  - With probability 1-α, follow a link at random among the ones in the current page
  - With probability α, jump to a random page

PageRank with Random jumps

1. # Initialize each page's rank to 1.0
   For each p in pages set PageRank(p) to 1.0
2. Iterate for max iterations
   a. Page p sends a contribution PageRank(p)/numOutLinks(p) to its neighbors (the pages it links)
   b. Update each page's rank PageRank(p) to $\alpha + (1 - \alpha) * \text{sum(received contributions)}$
   c. Go to step 2

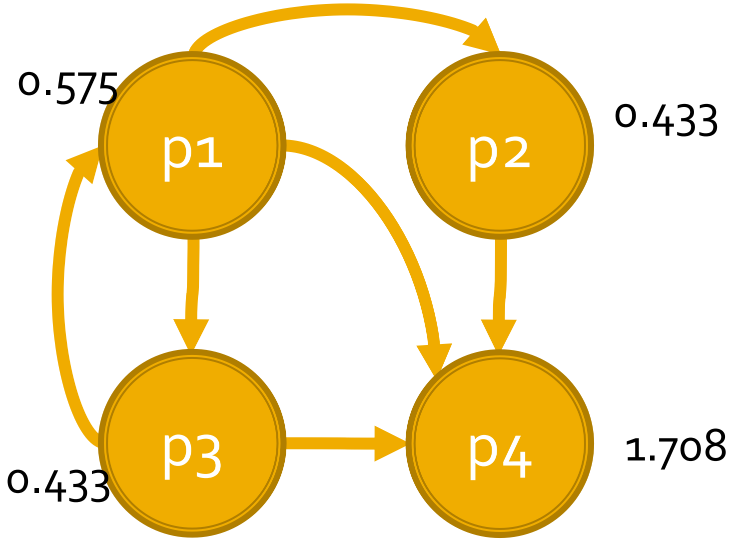
PageRank: Example

- $\alpha = 0.15$
- Initialization: PageRange(p) = 1.0 $\forall$p



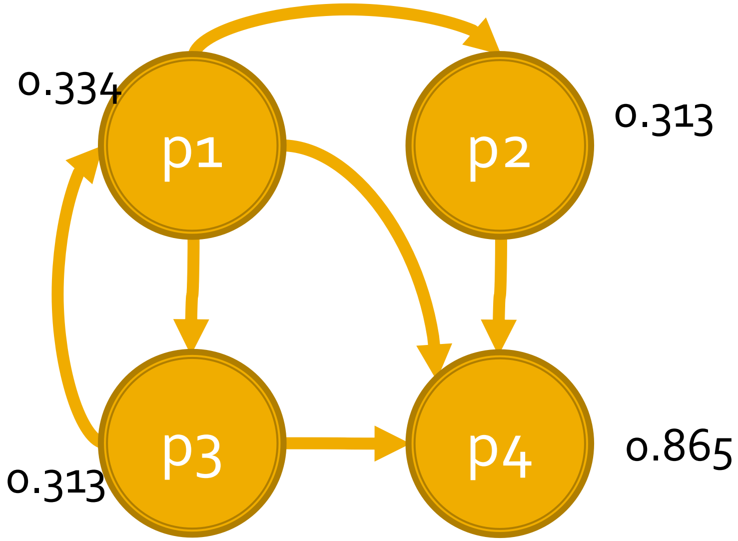
PageRank: Example

- Iteration #1



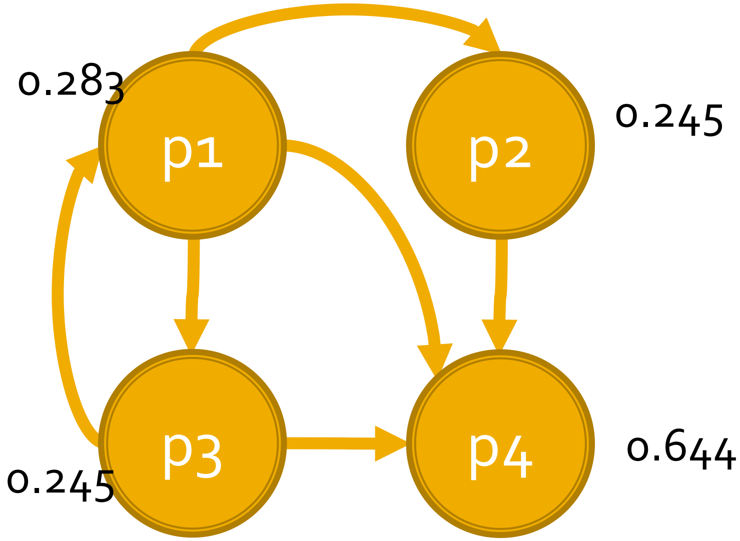
PageRank: Example

- Iteration #2



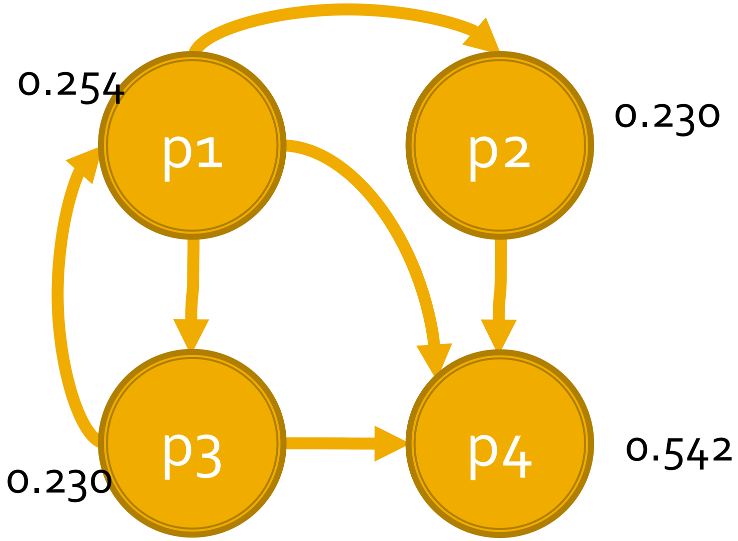
PageRank: Example

- Iteration #3



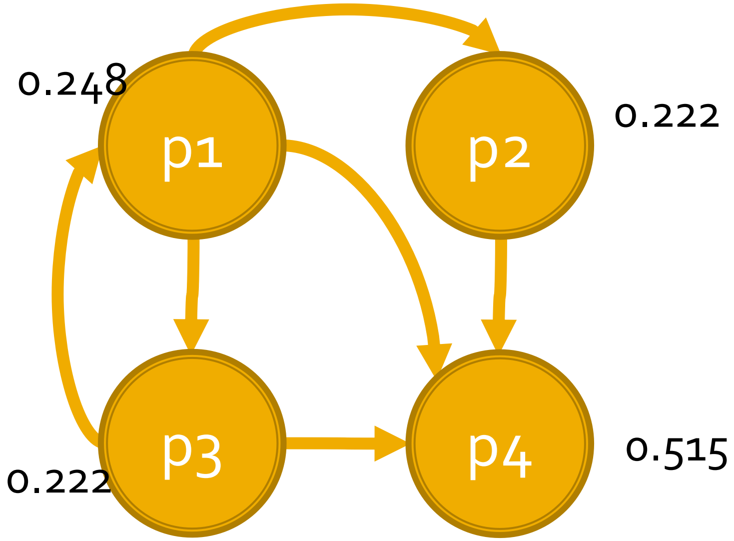
PageRank: Example

- Iteration #4



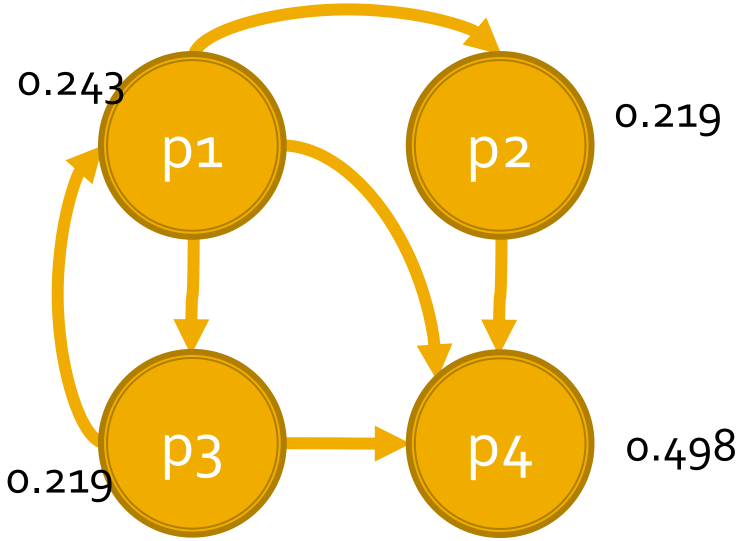
PageRank: Example

- Iteration #5



PageRank: Example

- Iteration #50



PageRank

- The **pageRank()** method of the GraphFrame class runs the PageRank algorithm on the input graph

PageRank

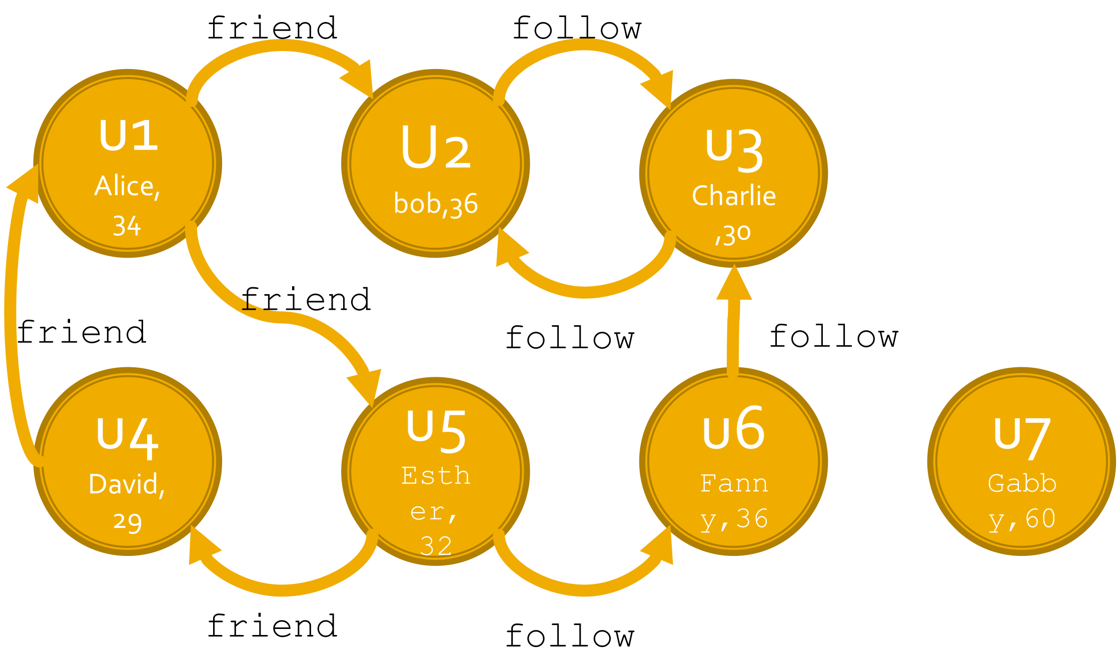
- Parameters
  - resetProbability
    - Probability of resetting to a random vertex (probability α associated with random jumps)
  - maxIter
    - If set, the algorithm is run for a fixed number of iterations
    - This may not be set if the tol parameter is set
  - Tol
    - If set, the algorithm is run until the given tolerance
    - This may not be set if the numIter parameter is set
  - sourceId (optional)
    - The source vertex for a personalized PageRank

PageRank

- **pageRank()** returns a new GraphFrame that
  - Contains the same vertexes and edges of the input graph
  - All the vertexes of the new graph are characterized by one new attribute, called "pagerank", that stores the PageRank of the vertexes
  - The edges of the new graph are characterized by one new attribute, called "weight", that stores the weight (PageRank contribution) propagated through that edge

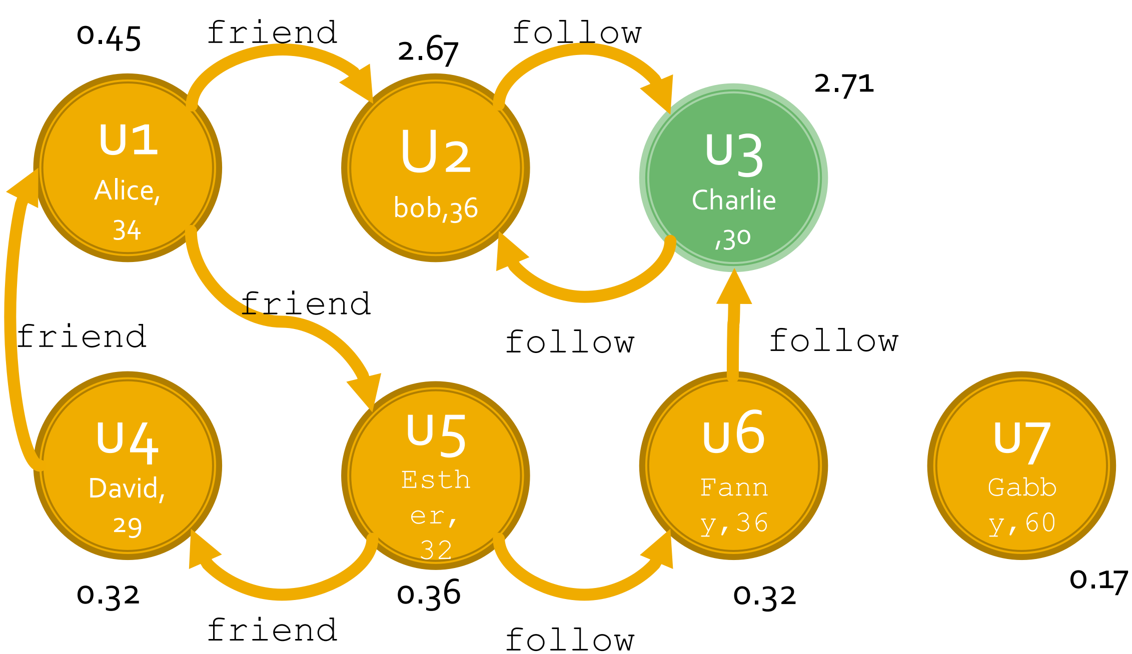
PageRank: Example

- Apply the PageRank algorithm on the following graph and select the user associated with the highest PageRank value



PageRank: Example

- Apply the PageRank algorithm on the following graph and select the user associated with the highest PageRank value



PageRank: Example

```
from graphframes import GraphFrame

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

PageRank: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                           ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

PageRank: Example

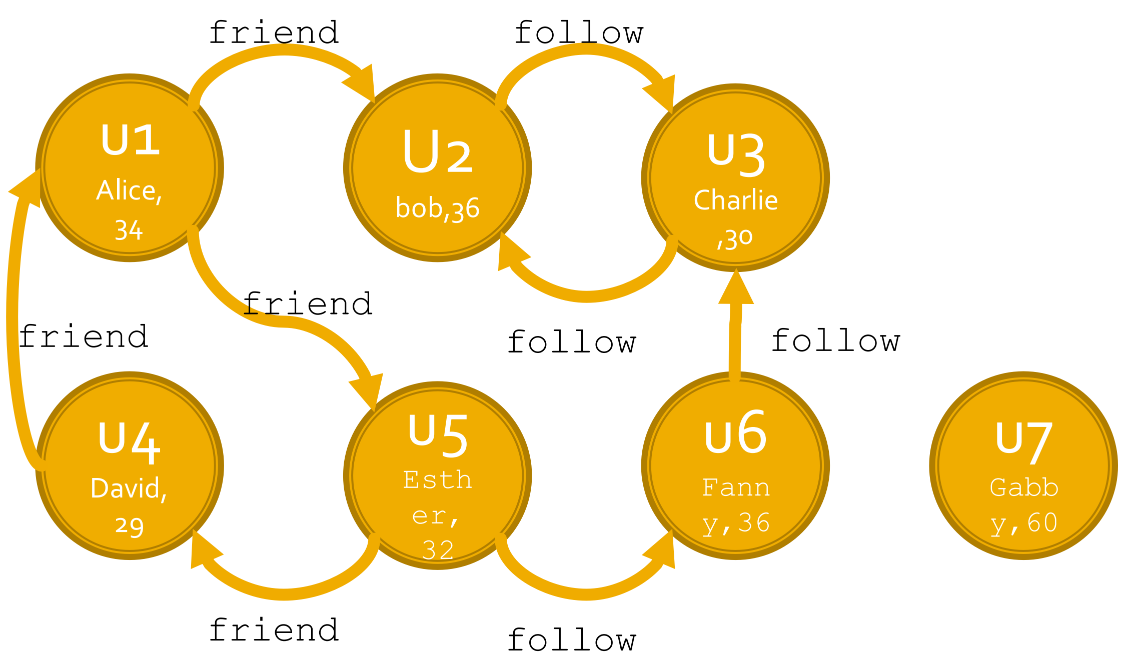
```
# Run the PageRank algorithm
pageRanks = g.pageRank(maxIter=30)

# Select the maximum value of PageRank
maxPageRank = pageRanks.vertices.agg({"pagerank":"max"})\
                    .first()["max(pagerank)"]

# Select the user with the maximum PageRank
pageRanks.vertices.filter(pageRanks.vertices.pagerank==maxPageRank)\
                       .show()
```

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 vertexes, and aggregate messages for each vertex
    - Joins
      - Join message aggregates with the original graph

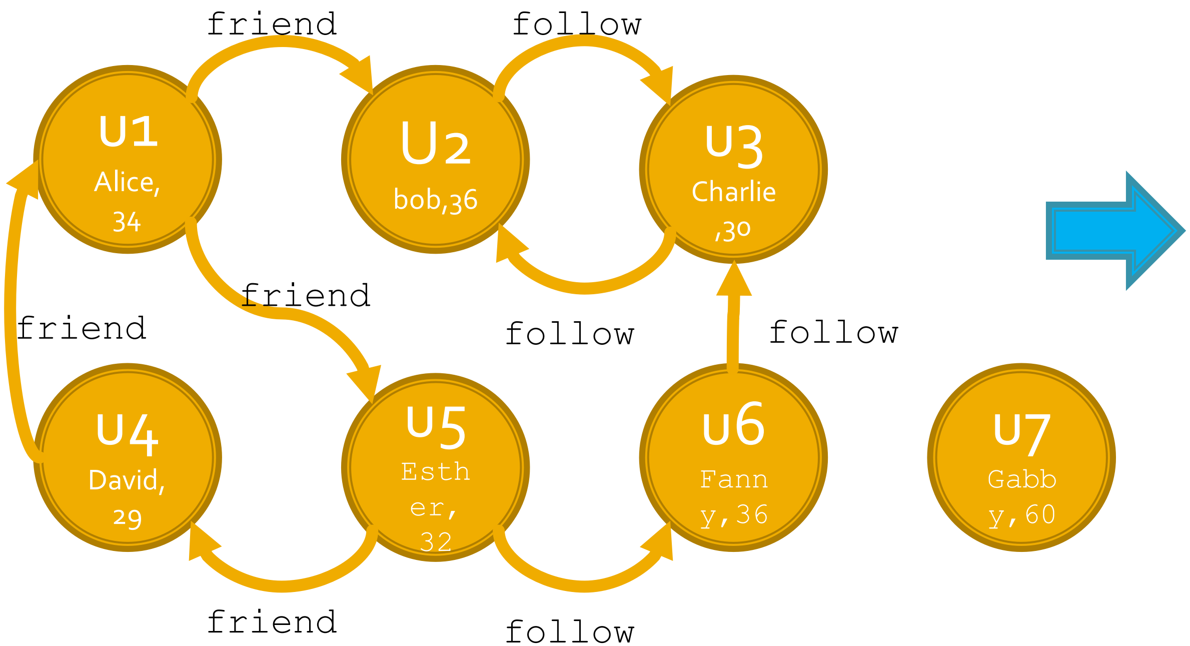
Custom graph algorithm: Example

- For each user, compute the sum of the ages of adjacent users (count many times the same adjacent user if there are many links)



Custom graph algorithm: Example

- For each user, compute the sum of the ages of adjacent users (count many times the same adjacent user if there are many links)



Vertex	SumAges
U1	97
U2	94
U3	108
U4	66
U5	99
u6	62

Custom graph algorithm: Example

```
from graphframes import GraphFrame
from pyspark.sql.functions import sum
from graphframes.lib import AggregateMessages

# Vertex DataFrame
v = spark.createDataFrame([ ("u1", "Alice", 34),\
                            ("u2", "Bob", 36),\
                            ("u3", "Charlie", 30),\
                            ("u4", "David", 29),\
                            ("u5", "Esther", 32),\
                            ("u6", "Fanny", 36),\
                            ("u7", "Gabby", 60)],\
                           ["id", "name", "age"])
```

Custom graph algorithm: Example

```
# Edge DataFrame
e = spark.createDataFrame([ ("u1", "u2", "friend"),\
                            ("u2", "u3", "follow"),\
                            ("u3", "u2", "follow"),\
                            ("u6", "u3", "follow"),\
                            ("u5", "u6", "follow"),\
                            ("u5", "u4", "friend"),\
                            ("u4", "u1", "friend"),\
                            ("u1", "u5", "friend")],\
                          ["src", "dst", "relationship"])

# Create the graph
g = GraphFrame(v, e)
```

Custom graph algorithm: Example

```
# For each user, sum the ages of the adjacent users

# Send the age of each destination of an edge to its source
msgToSrc = AggregateMessages.dst["age"]

# Send the age of each source of an edge to its destination
msgToDst = AggregateMessages.src["age"]

# Aggregate messages
aggAge = g.aggregateMessages(sum(AggregateMessages.msg),\
                sendToSrc=msgToSrc,\
                sendToDst=msgToDst)
#Show result
aggAge.show()
```