Graphs and Big Data

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

https://tinyurl.com/cis545-lecture-02-21-22

Networks (Graphs) are Everywhe

- Transportation
- Economics
- Society / Friendships and Interest groups
- Information sources
- Biology
- Computing
- …

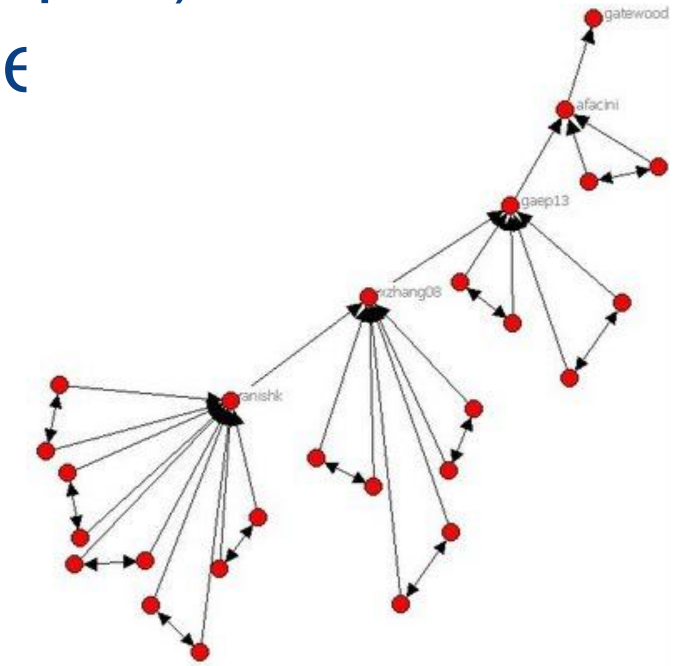


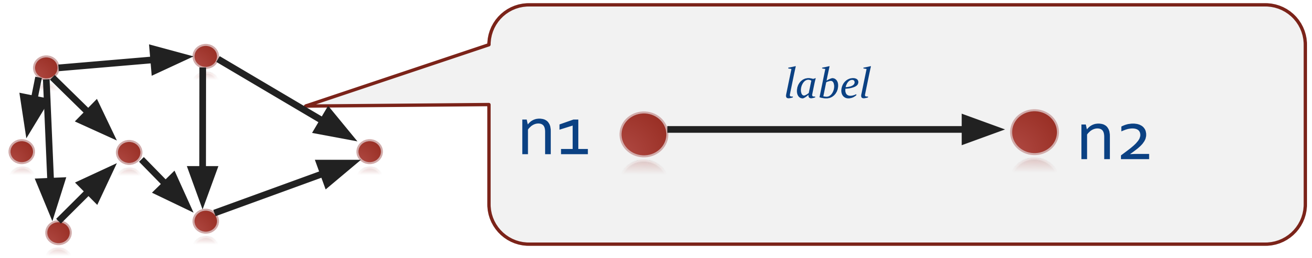
Figure by Bruce Hoppe, Creative Commons Licensed

May be implicit (we compute links) or explicit (we can observe links)

For our running example: we'll look at the LinkedIn connection network

https://tinyurl.com/cis545-lecture-02-21-22

Refresher: Graph Theory Basics



Graph G = (V,E)

V is a set of *vertices* or *nodes*, possibly with *properties*

E is a set of **tuples** of vertices, called edges, and may have lables or other data

V(node, label, prop1) e.g., (n1, "bob", 20)
E(source, label, target) e.g. (n1, 'friend_of', n2)

https://tinyurl.com/cis545-lecture-02-21-22

Some Terminology

u,v are adjacent if there's an edge between u and v

degree (u) = # adjacent vertices

- indegree or outdegree

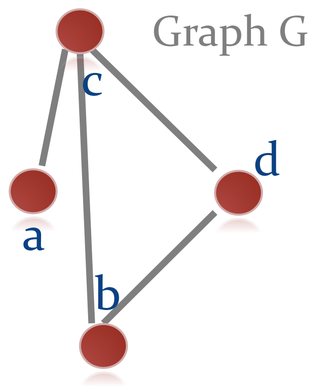
path = sequence of adjacent vertices

u,v are connected if path between u and v

- Connected component: Set of vertices connected to each other, that is not part of a larger connected set.
- Triangle: 3 vertices that are pairwise adjacent.
- Clique: Any set of vertices that are all pairwise adjacent.

https://tinyurl.com/cis545-lecture-02-21-22

Encoding Graphs as Data Structures



*We'll focus on this initially, as an **edge** Dataframe*

Adjacency list L(G)

a □	c
b □	c, d
c □	b, a, d
d □	c, b

(a,c)
(b,c) (b,d)
(c,b) (c,a) (c,d)
(d,c) (d,b)

Adjacency matrix A(G)

	a	b	c	d
a	0	0	1	0
b	0	0	1	1
c	1	1	0	1
d	0	1	1	0

We'll see this subsequently

https://tinyurl.com/cis545-lecture-02-21-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771577 (10A)

The most natural representation of a graph (ignoring unconnected nodes) in dataframes is as

a. an edge list
b. an adjacency list
c. an adjacency matrix
d. a hierarchical structure

The notion of a clique generalizes which concept?

e. indegree
f. graph
g. triangle
h. connected component

https://tinyurl.com/cis545-lecture-02-21-22

Road Map

- Simple graph analysis, centrality, and “betweenness”
- Graph exploration
- BFS in Spark
- Applications of BFS

https://tinyurl.com/cis545-lecture-02-21-22

A Brief Intro to Graph Analysis, aka Network Science

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

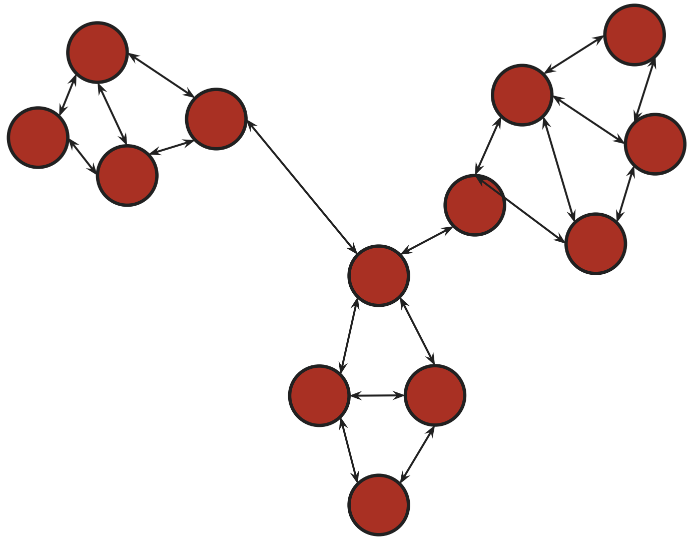
Penn Engineering

ODPi OpenDS4All

Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-21-22

A Simple Social (or Informational) Network as a Graph



Who is most influential?

What are the natural groupings?

Where should we suggest new links?

https://tinyurl.com/cis545-lecture-02-21-22

“Network Centrality”

The general question:

How do we measure the *important* nodes in a graph?

aka the “central” nodes

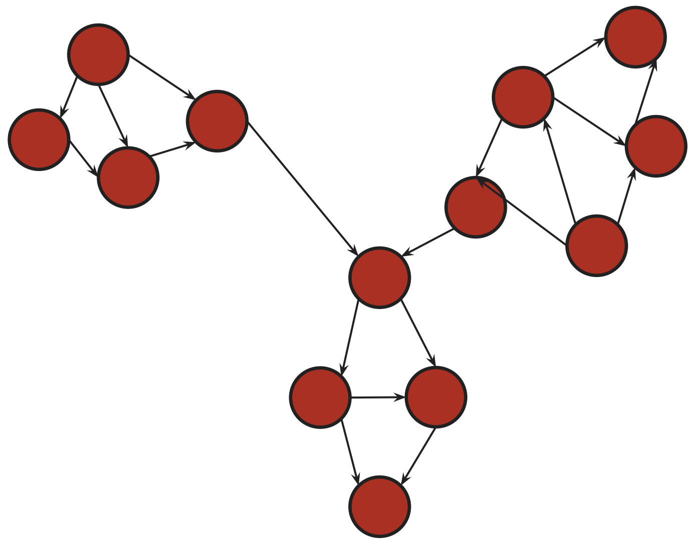
Several methods proposed in network science literature:

- *Degree centrality*: for a node, how many **other nodes is it connected to**
- *Betweenness centrality*: for a node, how many **shortest paths** go through the node
- *Eigenvector centrality*: very similar to PageRank, which we’ll discuss shortly

https://tinyurl.com/cis545-lecture-02-21-22

Degree Centrality

- For each node, compute its **degree**, i.e., the number of edges it connects to



- In a **directed graph** suppose we want **outdegree** centrality, i.e., the number of edges coming out from each node n?

- How to write in Spark, given a relation **edges(from, to)**?

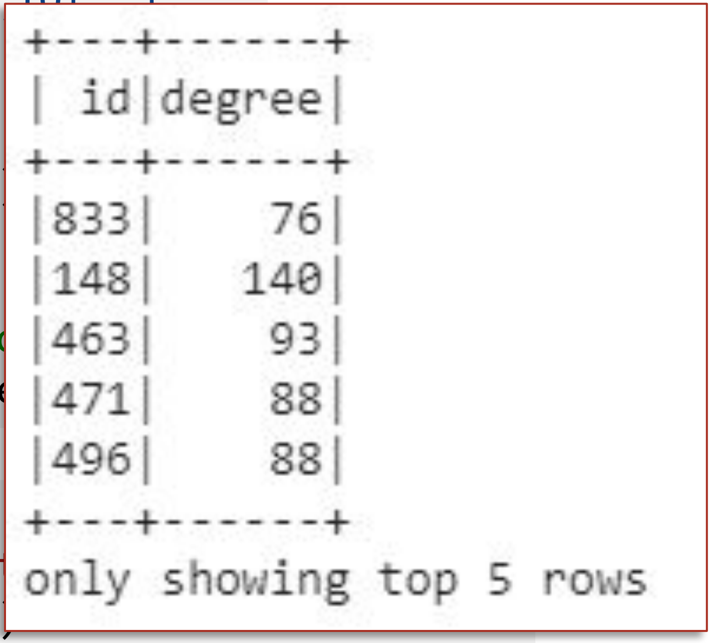
https://tinyurl.com/cis545-lecture-02-21-22

LinkedIn Example

Given a list **edges** that looks like: [[0,5], [5,10]]

```
from pyspark.sql.types import IntegerType
schema = StructType([
  StructField("from", IntegerType(), True
  StructField("to", IntegerType(), True)
])
# Load the remote data as a list of dicti
edges_df = sqlContext.createDataFrame(edge
```

```
edges_df.createOrReplaceTempView('edges')
sqlContext.sql('select from as id, count(
              \'from edges group by from')
```



```
+---+------+
| id|degree|
+---+------+
|833|    76|
|148|   140|
|463|    93|
|471|    88|
|496|    88|
+---+------+
only showing top 5 rows
```

https://tinyurl.com/cis545-lecture-02-21-22

Beyond Degree Centrality

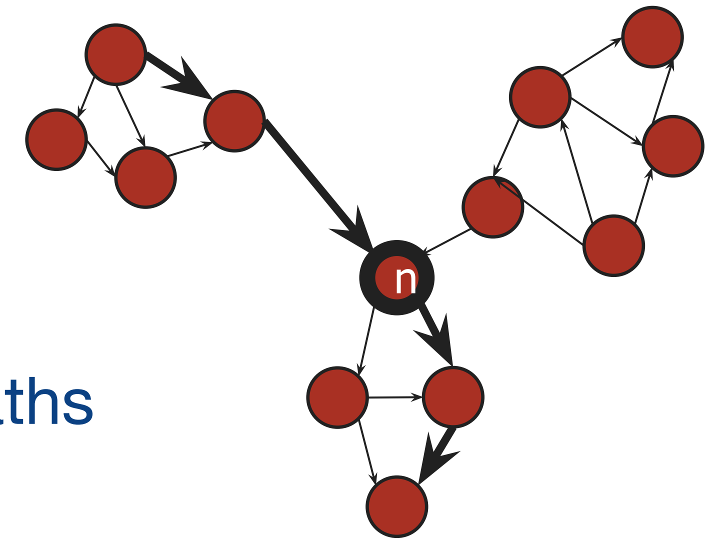
- Degree centrality is moderately useful
  - citation counts in academia
  - number of followers in Twitter
  - number of commits in GitHub

- But we may want to look at relationships to more distant nodes

https://tinyurl.com/cis545-lecture-02-21-22

Betweenness Centrality

- Some people (or entities) are important "connectors" – they bridge natural clusters
- Another measure: how many shortest paths go *through* a given node?
- To compute:
  - Find all shortest paths
  - Count how many include any node *n*



# shortest paths between every (A,B) *through* n

# shortest paths between every (A,B)

https://tinyurl.com/cis545-lecture-02-21-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771585 (10B)

Which type of centrality is reliant on computing shortest paths?

a. betweenness centrality
b. PageRank centrality
c. degree centrality
d. eigenvector centrality

When someone is proud of their number of retweets in Twitter, this is an instance of

e. degree centrality
f. like centrality
g. eigenvector centrality
h. betweenness centrality

https://tinyurl.com/cis545-lecture-02-21-22

Recap

Network *centrality* seeks to identify the influence ("centrality") of a node

- Simplest measures are based on direct connectivity
- Most measures take into account the broader graph and its paths!

So how do we explore paths?

https://tinyurl.com/cis545-lecture-02-21-22

Graph Exploration

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

https://tinyurl.com/cis545-lecture-02-21-22

Exploring a Graph

- Commonly, we will want to start at some node and look at how it relates to other nodes in the graph
  - How far away is X from Y?
  - How many nodes are within distance *k*?
  - What are the odds I can start at X and end up at Y?
  - (Some of these are the basis of ranking + recommendations)
- So how can we do this? Let's start with a single machine…

https://tinyurl.com/cis545-lecture-02-21-22

Computing Distance in a Graph

How far apart are two nodes?

Distance between two nodes =
number of edges on the **shortest path** between them.

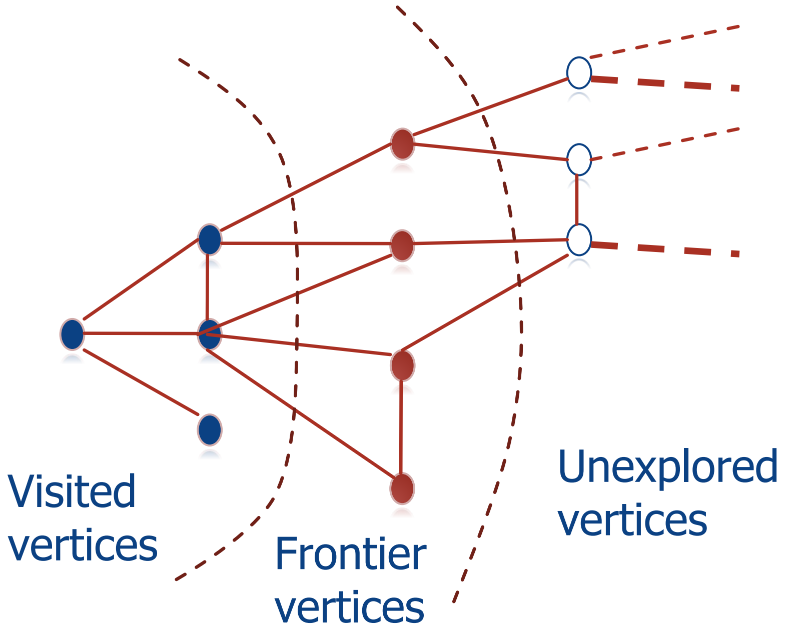
Breadth-First Search: Algorithm "pattern" for exploring at successively greater distances

Needs to remember two things:

- What you have already visited (don't want to backtrack)
- What places you've learned about but haven't visited

https://tinyurl.com/cis545-lecture-02-21-22

Breadth-First Search (BFS) for Undirected or Directed Graphs



Queue of Frontier Vertices

https://tinyurl.com/cis545-lecture-02-21-22

BFS - Centralized

```
Initialize a **frontier queue** with the origin node
While the **frontier queue** has a vertex in it
        Pick a vertex **v** from the front of the queue
        Put each unexplored neighbor of v in **queue**
```

Note closer edges are always considered before more distant edges.

Efficiency: Each edge is examined **once** (*undirected*: in each direction) (if graph given as adjacency list).

Just a small amount of work is required to examine each edge.
Running time is proportional to the number of edges.

Let's see it in Python…

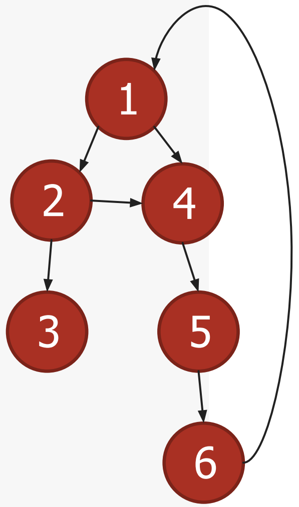
https://tinyurl.com/cis545-lecture-02-21-22

```python
graph = [(1,2),
         (1,4),
         (2,3),
         (2,4),
         (4,5),
         (5,6),
         (6,1)]

# Haven't visited anything
visited = set()

# Start at 1
frontier = [1]

while len(frontier) > 0:
    current = frontier.pop(0)
    visited.add(current)
    print ('Visiting', current)
    for item in graph:
        if item[0] == current:
            # Unexplored!
            if item[1] not in visited \
            and item[1] not in frontier:
                print (' Enqueuing', item[1])
                frontier.append(item[1])
```



```
Visiting 1
 Enqueuing 2
 Enqueuing 4
Visiting 2
 Enqueuing 3
Visiting 4
 Enqueuing 5
Visiting 3
Visiting 5
 Enqueuing 6
Visiting 6
```

https://tinyurl.com/cis545-lecture-02-21-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771595 (10D)

What data structures does breadth-first search employ?

a. queue of visited nodes only
b. queue of frontier nodes, set of visited nodes
c. queue of visited nodes, set of frontier nodes
d. set of frontier nodes only

How many times do we revisit a node in BFS?

e. we only visit each node once
f. we visit each node twice
g. we visit each node n times
h. we visit each node once for each edge

https://tinyurl.com/cis545-lecture-02-21-22

Breadth-First Search on One Computer

- Simple idea: queue enforces exploration from fewest hops to successively greater and greater distances
  - We can focus on the *frontier* which has new nodes
  - We can *prune* paths that revisit nodes we've already seen
- Requires access to a global queue, and is **inherently sequential** – so we need a different approach…

https://tinyurl.com/cis545-lecture-02-21-22

Distributed Breadth First Search

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

CIS 545 – Big Data Analytics

Portions of this lecture have been contributed to the OpenDS4All project, piloted by Penn, IBM, and the Linux Foundation

https://tinyurl.com/cis545-lecture-02-21-22

How Do We Distribute BFS?

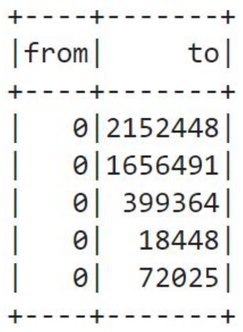
https://tinyurl.com/cis545-006

- Don’t want to traverse one node at a time
- Can’t order directly using a global queue…
- … And need to be careful about when we check for “visited” status

https://tinyurl.com/cis545-lecture-02-21-22

Suppose Our Graph is in an Edge Relation

edges_df



```
+----+-------+
|from|     to|
+----+-------+
|   0|2152448|
|   0|1656491|
|   0| 399364|
|   0|  18448|
|   0|  72025|
+----+-------+
```

Can we traverse from a subset of these nodes, via BFS, to more distant nodes?

(Later: we could count path lengths)

https://tinyurl.com/cis545-lecture-02-21-22

A Sketch of the Approach

Start with our origin nodes!

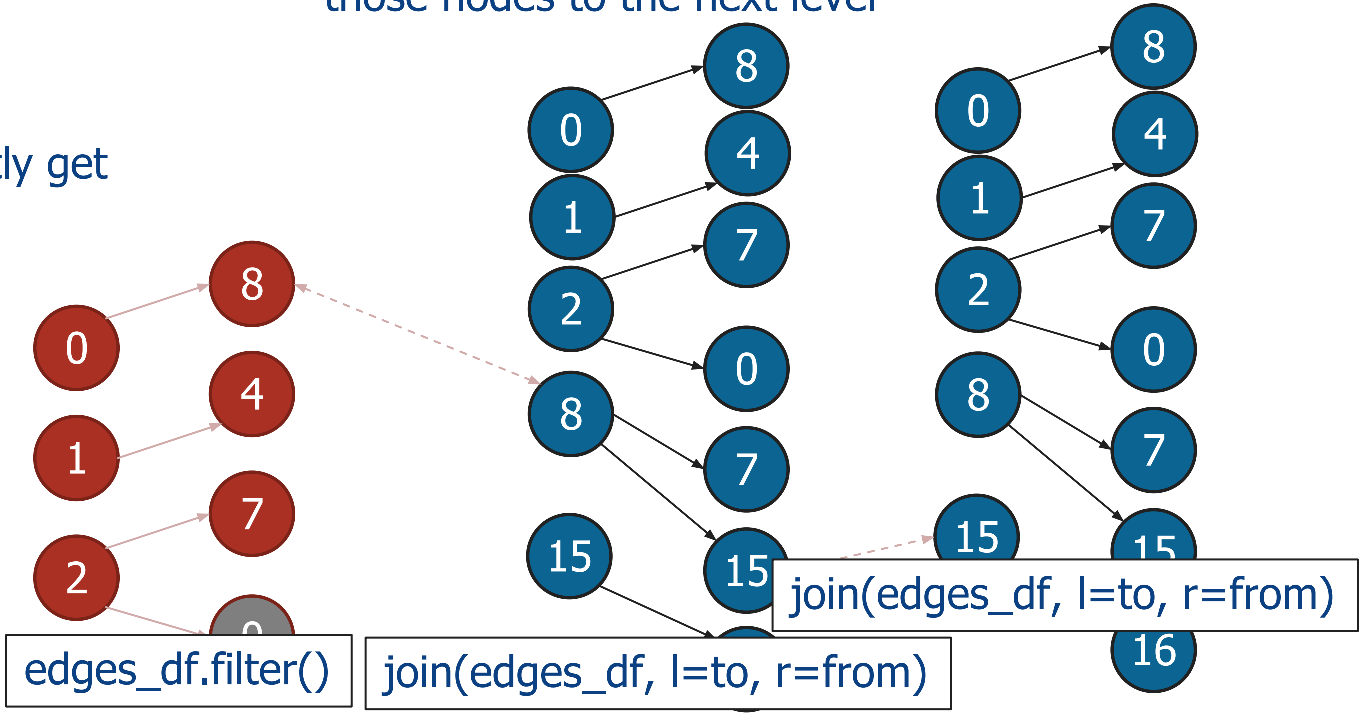
all nodes with ID < 3

From **edges_df** we can directly get the **one-hop** neighbors

from	to
0	8
1	4
2	7
2	0
8	7
8	15
15	16

Then we can traverse from those nodes to the next level

And the next

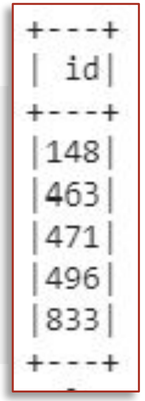


https://tinyurl.com/cis545-lecture-02-21-22

In Code, We Join in Each Traversal and Rename!

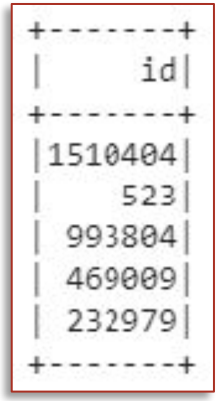
```python
from pyspark.sql.functions import col

# Start with a subset of nodes
start_nodes_df = edges_df[['from']].filter(edges_df['from'] < 1000).\
select(col('from').alias('id’)).drop_duplicates()
```



```
+---+
| id|
+---+
|148|
|463|
|471|
|496|
|833|
+---+
```

```python
neighbor_nodes_df = start_nodes_df.\
join(edges_df, start_nodes_df.id ==
edges_df['from']).\
select(col('to').alias('id'))
```



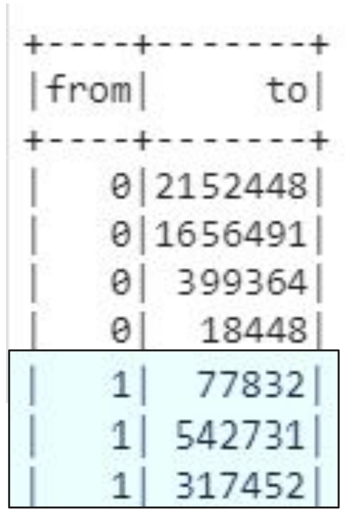
```
+-------+
|     id|
+-------+
|1510404|
|    523|
| 993804|
| 469009|
| 232979|
+-------+
```

https://tinyurl.com/cis545-lecture-02-21-22

What Happens Under the Covers?

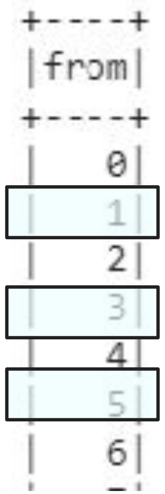
Suppose that **edges_df** is sharded by **from** and that we put even numbers on **worker 0** and odd numbers on **worker 1**

edges_df



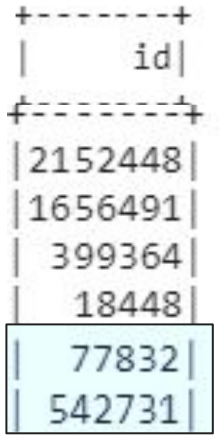
```
+----+-------+
|from|     to|
+----+-------+
|   0|2152448|
|   0|1656491|
|   0| 399364|
|   0|  18448|
|   1|  77832|
|   1| 542731|
|   1| 317452|
```

start_nodes_df



```
+----+
|from|
+----+
|   0|
|   1|
|   2|
|   3|
|   4|
|   5|
|   6|
```

neighbor_nodes_df

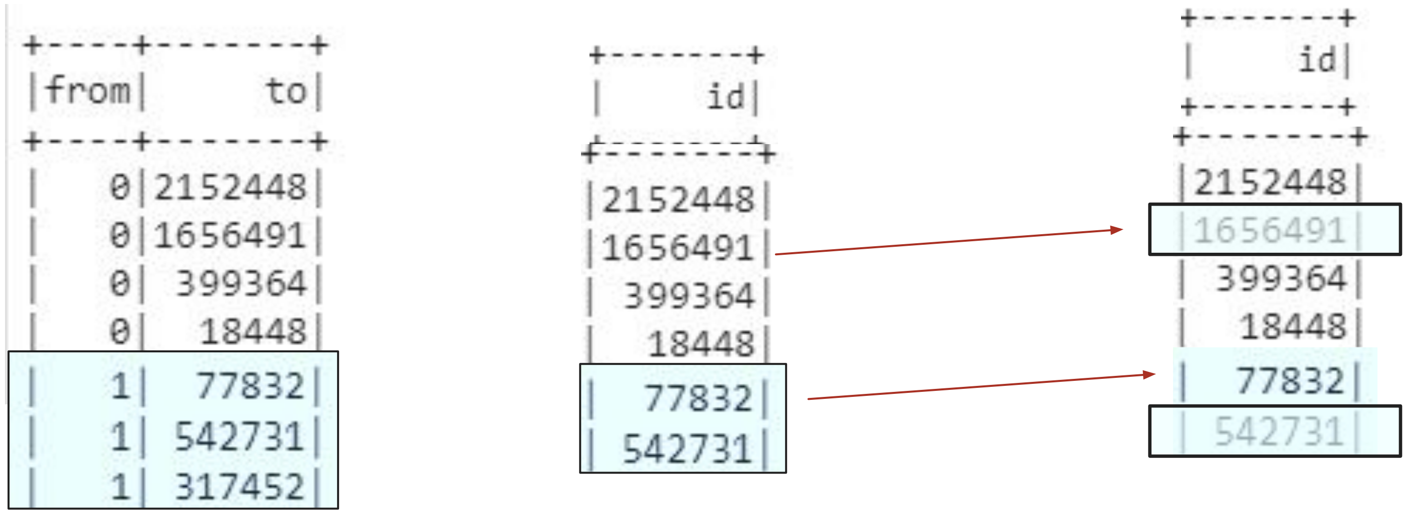


```
+-------+
|     id|
+-------+
|2152448|
|1656491|
| 399364|
|  18448|
|  77832|
| 542731|
```

https://tinyurl.com/cis545-lecture-02-21-22

Neighbor’s Neighbor?

```
neighbor_neighbor_nodes_df = neighbor_nodes_df.\
join(edges_df, neighbor_nodes_df.id == edges_df['from']).\
select(col('to').alias('id'))
```



edges_df

```
+----+-------+
|from|     to|
+----+-------+
|   0|2152448|
|   0|1656491|
|   0| 399364|
|   0|  18448|
|   1|  77832|
|   1| 542731|
|   1| 317452|
```

neighbor_nodes_df

```
+-------+
|     id|
+-------+
+-------+
|2152448|
|1656491|
| 399364|
|  18448|
|  77832|
| 542731|
```

neighbor_nodes_df ***repartitioned*** **by id**

```
+-------+
|     id|
+-------+
+-------+
|2152448|
|1656491|
| 399364|
|  18448|
|  77832|
| 542731|
```

https://tinyurl.com/cis545-lecture-02-21-22

Can We Do an *Iterative Join*?

Can we generalize from start -> neighbor -> neighbor’s neighbor in a loop?

- Base case: start with the direct edges, set distance to 1
- Iterative case:
  - start with the existing set of nodes, add an edge to get to new destinations
  - project start and end (and increment distance) – *use same schema as base case*
  - remove duplicates!

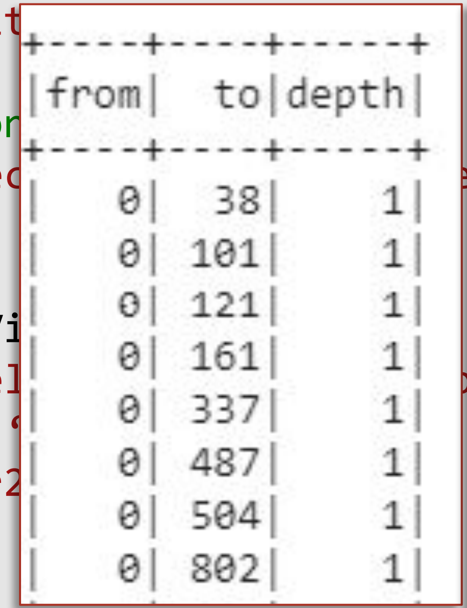
https://tinyurl.com/cis545-lecture-02-21-22

Iterative Join

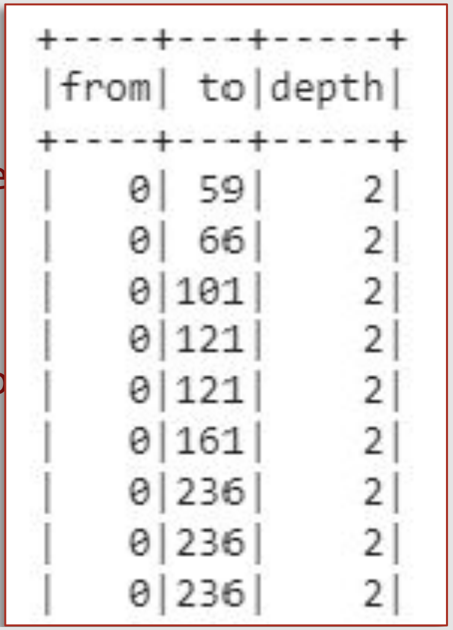
```
def iterate(df, depth):
    df.createOrReplaceTempView('it

    # Base case: direct connectio
    result = sqlContext.sql('selec

    for i in range(1, depth):
      result.createOrReplaceTempVi
      result = sqlContext.sql('sel
      as to, r1.depth+1 as depth '
      'from result r1 join iter r2
      'on r1.to=r2.from')
    return result
```



```
+----+----+-----+
|from|  to|depth|
+----+----+-----+
|   0|  38|    1|
|   0| 101|    1|
|   0| 121|    1|
|   0| 161|    1|
|   0| 337|    1|
|   0| 487|    1|
|   0| 504|    1|
|   0| 802|    1|
```



```
+----+---+-----+
|from| to|depth|
+----+---+-----+
|   0| 59|    2|
|   0| 66|    2|
|   0|101|    2|
|   0|121|    2|
|   0|121|    2|
|   0|161|    2|
|   0|236|    2|
|   0|236|    2|
|   0|236|    2|
```

```
iterate(edges_df.filter(edges_df['from'] < 1000),
2).orderBy('from','to').show()
```

https://tinyurl.com/cis545-lecture-02-21-22

What We Can Do Better

- In the loop we remove duplicate paths in each iteration
- But given paths of different lengths from s to t, we should remove the non-minimal ones!
- (You'll do this in the homework!)

https://tinyurl.com/cis545-lecture-02-21-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771531 (10E)

In a Spark-based iterative approach to BFS, we traverse edges

a. directly via a join with edges_df
b. one hop at a time via a join with edges_df
c. by grouping
d. by choosing the minimum path

Every time we do a join in a distributed BFS, there is a good chance we need to

e. group the results
f. select a subset of the matches
g. do another join
h. repartition one of the dataframes

https://tinyurl.com/cis545-lecture-02-21-22

Recap

- To do breadth-first traversals in Spark, we can iterate in "waves" from the origin(s)
  - A join at each stage
  - We may want to keep information about the distance, the path, etc.
  - We may want to prune all non-minimal paths
- Next: let's see some places BFS is used!

https://tinyurl.com/cis545-lecture-02-21-22

Applications of Breadth-First Search

Susan B. Davidson and Zachary G. Ives

University of Pennsylvania

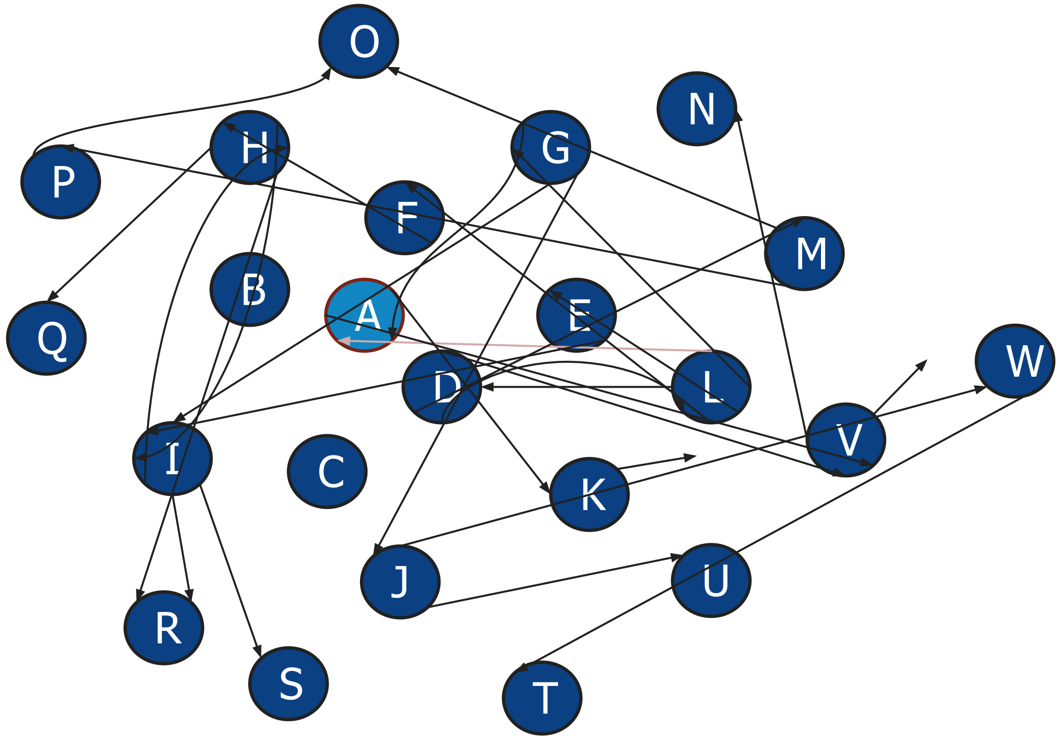
CIS 545 – Big Data Analytics

https://tinyurl.com/cis545-lecture-02-21-22

A Common Question

- How far away is V from A?
  - "Shortest path"
- Let's assume that
  1. the graph is directed and may have cycles
  2. all edges have equal ("unit") cost

Can BFS help?



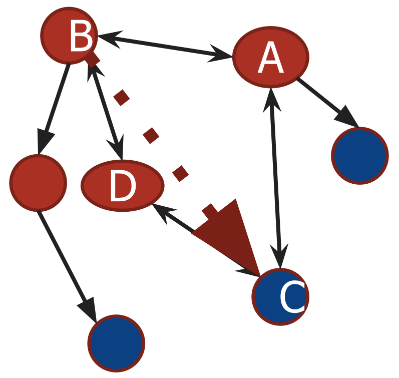
https://tinyurl.com/cis545-lecture-02-21-22

Adding Connections in Social Networks

One's friends tend to become each other's friends.
This is called Triadic Closure.

Node A violates the Triadic Closure Property if it has two friends B and C who are not each other's friends.

We often look to **complete triangles** to recommend friends – prioritize by the number of incomplete triangles



How can we use BFS/Shortest Path here?

https://tinyurl.com/cis545-lecture-02-21-22

A Sketch

- Run BFS from “us” to find **friends** (nodes at depth 1) and **friends of our friends** (nodes with min depth 2)

- Run BFS from FoF to depth 1

- For each FoF n, count how many **of our friends are in common**

- Rank each FoF *n* by how many friends we have in common

https://tinyurl.com/cis545-lecture-02-21-22

Other Common Graph Algorithms

- Algorithms for min-cost trees, traversals (minimum spanning tree, Steiner tree)
- Betweenness centrality
- As we'll see shortly, other *recursive* definitions of centrality – eigenvector centrality, PageRank, label propagation, variations thereof…

https://tinyurl.com/cis545-lecture-02-21-22

Brief Review

https://canvas.upenn.edu/courses/1636888/quizzes/2771549 (10F)

An important use case of BFS is

a. bipartite matching
b. k-means clustering
c. shortest-path computation
d. graph coloring

Triadic closure involves…

e. selecting people who resemble each other
f. adding edges to complete the most triangles
g. creating k-clusters
h. ranking friends-of-friends by strength of friendship

https://tinyurl.com/cis545-lecture-02-21-22

Summary

- Joins are a way of starting with a set of nodes and performing path traversals
- Multi-step joins achieve multi-step paths
- We can easily implement distributed BFS and use this to solve other problems

https://tinyurl.com/cis545-lecture-02-21-22