# Homework Writeup:

## //--Analyzing Co-Ocurrence Networks with GraphX--//

/\*

Data scientists nowadays come from all different types of fields.

1. The fields are often interested in relation between entities

2. Explosion of digital data, more data then we know what to use for

- Has forced these workers to pick up new skills to manage data

- Realized methods to anaylze relationships applicable to other field

- Network science is born!

- Uses graph theory (study of pairwise relationships [edges]

between set of entities [vertices])

- Used everywhere, like CS for data structures

- Big impact in business world as well. Major coprotations

abilities depend entirely on how they are able to anaylze

- Facebook, friends, Linkedin, colleges, Amazon, prods, etc.

- Companies like these drive creation of Distrubuted

Processing Framewroks like Mapreduce, and hire data scientists

- First use of Mapreduce, to solve Pagerank issue

- Over time, graph gotten much larger, need faster graph parrallel

processing frameworks (Pregel - Google, Giraph - Yahoo, Graphlab)

- "supported fault-tolerant, in-memory, iterative, and graph-

centric processing, and were capable of performing" than Mapreduce

- This chapter will introduce GraphX, let us use aforementioned frameworks ^^^

- Since in spark, easier to integrate graphx for analyze network-centric data

//-GraphX and GraphFrames--//

Designed before DF in 1.3, to work with RDDs. Working in progress to port

GraphX to new API in DF (GraphFrames). Rest of chapter uses GraphX since

GraphFrames isn't ready yet. But no worries, all ideas in GraphFrames present

in GraphX.

\*/

## //--The MEDLINE Citation Index: A Network Analysis--//

/\*

MEDLINE (Medical Literature Analysis and Retrieval System Online) database of

academic papers covering life science & medicine. Managed by US NLM. Citation index

trace back to 1879, available to med school in 1971, and public 1996. Updated 5 day

per week. Due to volumn of cits, updates, and research community developed an

extensive set of semantic tags, called MeSH (Medical Subject Headings). Tags provide

framework of relation. 2001: Pubgene released first text mining search engine to

match docs to terms.

Chapter is going to use Scale, Spark, and GraphX to analyze network of MeSH terms.

Idea is based off "https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4090190/" Use GraphX

instead of R and C++.

To feel data out, we will

1. Major topics and co-occurances (no GraphX)

2. Look for connected components (follow one path of cites to another topic?)

3. Degree distribution (how topics vary and connect similar)

4. Clustering coeefient and Average Path Length [complicated]

After these steps, will allow us to understamd how similar citation graphs work

\*/

## //--Getting the Data--//

/\*

//Grab sample of citation index

$ mkdir medline\_data

$ cd medline\_data

$ wget ftp://ftp.nlm.nih.gov/nlmdata/sample/medline/^.gz

//Uncompress data and examine

$ gunzip \*.gz

$ ls -ltr

...

total 1814128

-rw-r--r-- 1 jwills staff 145188012 Dec 3 2015 medsamp2016h.xml

-rw-r--r-- 1 jwills staff 133663105 Dec 3 2015 medsamp2016g.xml

-rw-r--r-- 1 jwills staff 131298588 Dec 3 2015 medsamp2016f.xml

-rw-r--r-- 1 jwills staff 156910066 Dec 3 2015 medsamp2016e.xml

-rw-r--r-- 1 jwills staff 112711106 Dec 3 2015 medsamp2016d.xml

-rw-r--r-- 1 jwills staff 105189622 Dec 3 2015 medsamp2016c.xml

-rw-r--r-- 1 jwills staff 72705330 Dec 3 2015 medsamp2016b.xml

-rw-r--r-- 1 jwills staff 71147066 Dec 3 2015 medsamp2016a.xml

"Each entry in the sample files is a MedlineCitation record, which contains

information about the publication of an article in a biomedical journal,

including the journal name, issue, publication date, the names of the authors,

the abstract, and the set of MeSH keywords that are associated with the article"

Also each MESH keyword has attribute to tell if keyword major topic of doc.

//First citation record in medsamp2016a.xml

<MedlineCitation Owner="PIP" Status="MEDLINE">

<PMID Version="1">12255379</PMID>

<DateCreated>

<Year>1980</Year>

<Month>01</Month>

<Day>03</Day>

</DateCreated>

...

<MeshHeadingList>

...

<MeshHeading>

<DescriptorName MajorTopicYN="N">Humans</DescriptorName>

</MeshHeading>

<MeshHeading>

<DescriptorName MajorTopicYN="Y">Intelligence</DescriptorName>

</MeshHeading>

<MeshHeading>

<DescriptorName MajorTopicYN="Y">Rorschach Test</DescriptorName>

</MeshHeading>

...

</MeshHeadingList>

...

</MedlineCitation>A

In last case Wikipedia, we wanted to extract the text. We actually here want to

extract the DescriptorName tags by parsing the XML.

//To compile code into Jar

$ cd ch07-graph/

$ mvn package

$ cd target

$ spark-shell --jars /proj/cse398-498/course/aas/ch07-graph/target/ch07-graph-2.0.0-jar-with-dependencies.jar

\*/

//Read XML MEDLINE data into shell

import edu.umd.cloud9.collection.XMLInputFormat

import org.apache.spark.sql.{Dataset, SparkSession}

import org.apache.hadoop.io.{Text, LongWritable}

import org.apache.hadoop.conf.Configuration

def loadMedline(spark: SparkSession, path: String) = {

import spark.implicits.\_

@transient val conf = new Configuration()

conf.set(XMLInputFormat.START\_TAG\_KEY, "<MedlineCitation ")

conf.set(XMLInputFormat.END\_TAG\_KEY, "</MedlineCitation>")

val sc = spark.sparkContext

val in = sc.newAPIHadoopFile(path, classOf[XMLInputFormat],

classOf[LongWritable], classOf[Text], conf)

in.map(line => line.\_2.toString).toDS()

}

val medlineRaw = loadMedline(spark, "/proj/cse398-498/course/AAS\_CH7/medline\_data")

## //--Parsing XML Documents with Scala’s XML Library--//

/\*

Scala has treated XML as first-class data type since 1.2. Folllowing is valid:

import scala.xml.\_

val cit = <MedlineCitation>data</MedlineCitation>

Support for XML literals is unusual. As of Scala 2.11, scalaxml no longer part

of core Scala libraries. Must explicitly include scala-xml to use.

\*/

//Start by taking unparsed first citation

import scala.xml.\_

val rawXml = medlineRaw.take(1)(0)

val elem = XML.loadString(rawXml) //elem variable instance of scala.XML.Elem

//used to represent individual node, have lots of builtin funcs

elem.label

/\*res18: String = MedlineCitation\*/

elem.attributes

/\*res20: scala.xml.MetaData = Status="MEDLINE" Owner="NLM"\*/

//has operators to find children of node

elem \ "MeshHeadingList" //only get node direct children

/\*

<MeshHeading>

<DescriptorName UI="D020816" MajorTopicYN="N">Amino Acid Motifs</DescriptorName>

</MeshHeading>

<MeshHeading>

<DescriptorName UI="D000595" MajorTopicYN="N">Amino Acid Sequence</DescriptorName>

</MeshHeading>

<MeshHeading>

<DescriptorName UI="D017124" MajorTopicYN="N">Conserved Sequence</DescriptorName>

</MeshHeading>

<MeshHeading>

<DescriptorName UI="D004268" MajorTopicYN="N">DNA-Binding Proteins</DescriptorName>

<QualifierName UI="Q000737" MajorTopicYN="N">chemistry</QualifierName>

<QualifierName UI="Q000378" MajorTopicYN="N">metabolism</QualifierName>

</MeshHeading>

<MeshHeading>

<DescriptorName UI="D003903" MajorTopicYN="N">Deuterium</DescriptorName>

<QualifierName UI="Q000378" MajorTopicYN="N">metabolism</QualifierNa...

\*/

//extract nondirect children using "//"

elem \\ "MeshHeading"

/\*

res22: scala.xml.NodeSeq =

NodeSeq(<MeshHeading>

<DescriptorName UI="D020816" MajorTopicYN="N">Amino Acid Motifs</DescriptorName>

</MeshHeading>, <MeshHeading>

<DescriptorName UI="D000595" MajorTopicYN="N">Amino Acid Sequence</DescriptorName>

</MeshHeading>, <MeshHeading>

<DescriptorName UI="D017124" MajorTopicYN="N">Conserved Sequence</DescriptorName>

</MeshHeading>, <MeshHeading>

<DescriptorName UI="D004268" MajorTopicYN="N">DNA-Binding Proteins</DescriptorName>

<QualifierName UI="Q000737" MajorTopicYN="N">chemistry</QualifierName>

<QualifierName UI="Q000378" MajorTopicYN="N">metabolism</QualifierName>

</MeshHeading>, <MeshHeading>

<DescriptorName UI="D003903" MajorTopicYN="N">Deuterium</DescriptorName>

<QualifierName UI="Q000378" MajorTopicYN="N">metabolism</QualifierName>

</MeshHead...

\*/

//use "\\" to get DescriptorName directly, and retrieve MESH tag within node

(elem \\ "DescriptorName").map(\_.text)

//return the name of major Mesh tags for each article

def majorTopics(record: String): Seq[String] = {

val elem = XML.loadString(record)

val dn = elem \\ "DescriptorName"

val mt = dn.filter(n => (n \ "@MajorTopicYN").text == "Y")

mt.map(n => n.text)

}

val elem = XML.loadString(rawXml)

val tempElem = elem.toString

majorTopics(tempElem)

/\*res25: Seq[String] = List(Nuclear Magnetic Resonance, Biomolecular)\*/

//now it parsing locally, apply to parse Mesh into our RDD

val medline = medlineRaw.map(majorTopics)

medline.cache()

medline.take(1)(0)

/\*res27: Seq[String] = List(Nuclear Magnetic Resonance, Biomolecular)\*/

## //--Analyzing the MeSH Major Topics and Their Co-Occurrences--//

//Do some basic mathematical calculations to get feel of data

medline.count()

val topics = medline.flatMap(mesh => mesh).toDF("topic")

topics.createOrReplaceTempView("topics")

val topicDist = spark.sql("""

SELECT topic, COUNT(\*) cnt

FROM topics

GROUP BY topic

ORDER BY cnt DESC""")

topicDist.count()

//res60: Long = 14548

topicDist.show()

/\*

+--------------------+----+

| topic| cnt|

+--------------------+----+

| Research|1649|

| Disease|1349|

| Neoplasms|1123|

| Tuberculosis|1066|

| Public Policy| 816|

| Jurisprudence| 796|

| Demography| 763|

| Population Dynamics| 753|

| Economics| 690|

| Medicine| 682|

|Socioeconomic Fac...| 655|

| Blood| 631|

| Politics| 631|

|Emigration and Im...| 601|

| Social Change| 577|

| Physicians| 560|

| Mutation| 542|

| Abortion, Induced| 503|

| Anesthesia| 483|

| Public Health| 479|

+--------------------+----+

\*/

/\*

The more general the topic, the more often it appears/ More than 13,000 different

topics in dataset. Biggest topic is only (1,649/240,000 ~ 0.7%). Relatively

long tail for overall distribution

\*/

//Test above theory

topicDist.createOrReplaceTempView("topic\_dist")

spark.sql("""

SELECT cnt, COUNT(\*) dist

FROM topic\_dist

GROUP BY cnt

ORDER BY dist DESC

LIMIT 10""").show()

/\*

+---+----+

|cnt|dist|

+---+----+

| 1|3106|

| 2|1699|

| 3|1207|

| 4| 902|

| 5| 680|

| 6| 571|

| 7| 490|

| 8| 380|

| 9| 356|

| 10| 296|

+---+----+

\*/

/\*

Primary interest is co-occuring MeSH topics. Each entry in medline is a set

of strings that are namnes of topics. To get co-occurances, need to generate

all of the 2-element subsets of this list of strings.

combinations to make generating these sublists extremely easy. combinations

returns an Iterator, meaning that the combinations need not all be held in memory

val list = List(1, 2, 3)

val combs = list.combinations(2)

combs.foreach(println)

-Results-

List(1, 2)

List(1, 3)

List(2, 3)

However, before we make a list of all combinations, we need to sort the data.

List(3, 2) == List(2, 3)

val combs = list.reverse.combinations(2)

combs.foreach(println)

-Results-

List(3, 2)

List(3, 1)

List(2, 1)

\*/

//Sort then generate two-element sublist

val topicPairs = medline.flatMap(t => {

t.sorted.combinations(2)

}).toDF("pairs") //create a sorted df of pairs

topicPairs.createOrReplaceTempView("topic\_pairs")

val cooccurs = spark.sql("""

SELECT pairs, COUNT(\*) cnt

FROM topic\_pairs

GROUP BY pairs""")

cooccurs.cache()

cooccurs.count()

//res68: Long = 213745

cooccurs.createOrReplaceTempView("cooccurs")

spark.sql("""

SELECT pairs, cnt

FROM cooccurs

ORDER BY cnt DESC

LIMIT 10""").collect().foreach(println)

/\*

[WrappedArray(Demography, Population Dynamics),288]

[WrappedArray(Government Regulation, Social Control, Formal),254]

[WrappedArray(Emigration and Immigration, Population Dynamics),230]

[WrappedArray(Acquired Immunodeficiency Syndrome, HIV Infections),220]

[WrappedArray(Antibiotics, Antitubercular, Dermatologic Agents),205]

[WrappedArray(Analgesia, Anesthesia),183]

[WrappedArray(Economics, Population Dynamics),181]

[WrappedArray(Analgesia, Anesthesia and Analgesia),179]

[WrappedArray(Anesthesia, Anesthesia and Analgesia),177]

[WrappedArray(Population Dynamics, Population Growth),174]

\*/

/\*

Nothing too interesting above. The top results are combinations of large topics

or are just words that are nearly the same like Analgesia and Anesthesia.

\*/

## //--Constructing a Co-Occurrence Network with GraphX--//

/\*

The standard tools we use aren't enough to give insight. Raw counts is not

enough, and the most co-occuring pairs are not of interest.

Need to analyze the co-occurence network: topics as vertices, and existence

of a cite that has both pairs as an edge btwn the 2. Then compute network-centric

stats, that help us understand the structure and investigate outlier vertices.

Use co-occurence network to find relationships btwn entities worth investigating.

https://bit.ly/3AdxDUj -> Figure 7-1

Similar to how MLLib provides algo for creating ML models in Spark, GraphX is

used to anaylze networks. GraphX is built ontop of Spark, capable of multi-

computer processing.

1. move from writing data-parallel ETL routines against RDDs to executing

graph-parallel algorithms against a graph

2. to analyzing and summarizing the output of the graph computation in a

data-parallel fashion again

GraphX built on RDD, specifically VertexRDD[VD] (each type per instance is long

for every vertex, while VD can be any other [vertex attribute], built on

RDD[(VertexId, VD)]), and EdgeRDD[ED] (Edge is a case class that contains two

VertexId values and an edge attribute of type ED, built on RDD[Edge[ED]]).

Both aforementioned have internal indices within each partition designed to

facilitate fast joins and attribute updates. Given both a VertexRDD and an

associated EdgeRDD, create Graph class, do a bunch of methods efficently

First requirement in creating graph is to have Long val used as identifier for

each vertex in graph. All topics rn is string. Need way to create unique

64-bit for each topic as an ID, need to be done in a distributed fasion,

done quickly for large dataset.

Use hashCode method, generate a 32-bit for any Scala object. In our case,

13000 vertices, no problem. What if tens of million? Hash code collision prob

occur. Due to this, we copy Google’s Guava Library to create a unique 64-bit

identifier for each topic using MD5 hashing algorithm

\*/

import java.nio.charset.StandardCharsets

import java.security.MessageDigest

def hashId(str: String): Long = {

val bytes = MessageDigest.getInstance("MD5").

digest(str.getBytes(StandardCharsets.UTF\_8))

(bytes(0) & 0xFFL) |

((bytes(1) & 0xFFL) << 8) |

((bytes(2) & 0xFFL) << 16) |

((bytes(3) & 0xFFL) << 24) |

((bytes(4) & 0xFFL) << 32) |

((bytes(5) & 0xFFL) << 40) |

((bytes(6) & 0xFFL) << 48) |

((bytes(7) & 0xFFL) << 56)

}

/\*

Apply hashing func to data to generate DF for basis to set of vertices in our

co-occurence graph. Also double check uniquness

\*/

import org.apache.spark.sql.Row

val vertices = topics.map{ case Row(topic: String) =>

(hashId(topic), topic) }.toDF("hash", "topic")

val uniqueHashes = vertices.agg(countDistinct("hash")).take(1)

/\*

Generate edges for graph from unique ID (hashing function to map each topic

name to its corresponding vertex ID). Good habit: left side VertexId (src)

is less than right side VertexId (dst). Most algto in GraphX don't assume

relationship btwn src and dst, only few do. Good idea to implement patten

early, dont forget

\*/

import org.apache.spark.graphx.\_

val edges = cooccurs.map{ case Row(topics: Seq[\_], cnt: Long) =>

val ids = topics.map(\_.toString).map(hashId).sorted

Edge(ids(0), ids(1), cnt)

}

//Have both vertices and edges, create Graph and cache

val vertexRDD = vertices.rdd.map{

case Row(hash: Long, topic: String) => (hash, topic)

}

val topicGraph = Graph(vertexRDD, edges.rdd)

topicGraph.cache()

/\*

The vertexRDD and edges used were regulear RDD. Didn't deduplicate. Graph API

does it for us. Converting them to VertexRDD and EdgeRDD, vertex all unique

\*/

vertexRDD.count()

//Long = 280464

topicGraph.vertices.count()

//res54: Long = 14548

/\*

Will not deduplicate EdgeRDD, GraphX create multigraphs, multiple edges with

diff values btwn same vertices. Usefull aplplcaition where verices are rich

objects (people - friends, collegues, etc) that have many diff relationships.

Allows us to treat edges as either direct or undirect (depending context)

\*/

## //--Understanding the Structure of Networks--//

/\*

When exploring the table, number of stats needed to be computed immediatly.

Same principle applies to graph, just diff stats. Easy due to Graph and RDD API

\*/

### 

### //-Connected Components-//

/\*

Most basic feature, if graph connected (travel from one vertex to any another).

If not, may be full of subgraphs. Connectedness is fundamental, graphx includes

function to do so. Number of spark jobs will appear when called, later result

\*/

val connectedComponentGraph = topicGraph.connectedComponents()

/\*

Type of object returned is another Graph class, but vertex attribute is VertexID,

unique ID for component each vertex belongs to. To get # of connected comp,

convert VertexRDD to df

\*/

val componentDF = connectedComponentGraph.vertices.toDF("vid", "cid")

val componentCounts = componentDF.groupBy("cid").count()

componentCounts.count()

componentCounts.orderBy(desc("count")).show()

/\*

Largest component had more than 90% of vertices, second is 4. Big difference.

To see names associatated with smaller components, need to join the VertexRDD

with vertices from our original concept graph. Innerjoin in graphx, faster than

normal join. Returns a new data type for the resulting VertexRDD. In this case

we want a DF with topic name and component ID.

\*/

val topicComponentDF = topicGraph.vertices.innerJoin(

connectedComponentGraph.vertices) {

(topicId, name, componentId) => (name, componentId.toLong)

}.toDF("topic", "cid")

topicComponentDF.where("cid = -2062883918534425492").show()

//topicComponentDF.where(col("cid")==="-8193948242717911820")

//Doesn't print? Data type mismatch?

/\*

"A bit of Google searching reveals that “Campylobacter” is a genus of bacteria that

is one of the most common causes of food poisoning, and “serotyping” is a technique

used for classifying bacteria based on their cell surface “antigens,” which is a

toxin that induces an immune response in the body."

\*/

//Find others that were related but were cut off.

val campy = spark.sql("""

SELECT \*

FROM topic\_dist

WHERE topic LIKE '%ampylobacter%'""")

campy.show()

/\*

+--------------------+---+

| topic|cnt|

+--------------------+---+

|Campylobacter jejuni| 3|

|Campylobacter Inf...| 2|

| Campylobacter| 1|

| Campylobacter coli| 1|

| Campylobacter fetus| 1|

+--------------------+---+

\*/

/\*

Fetus seems similar, but not reallt, occurs in cattle and shepp, not humans.

The broader pattern in our network is more citations we add, the more connected

things become.

connectedComponents method is doing iterative comps on graph to identify component

that each vertex belongs to, taking advantage of unique VertexID. During each

phase of comp, each vertex broadcast smallest VertexID value it has seen

to each neighboor. During first run, this will be vertex onw Id, but will be

generall updated in each run. Each vertex keep track of smallest IDs changes,

and when none change, the connected component comp is done, each vertex

assigned to when they were the smallest value for that component.

\*/

### //--Degree Distribution--//

/\*

Connected graphs differ drastically. Can be all points to one, but no other

connections. If delete center vertix, whole model breaks apart into indy vertex.

May have case where all connected to only 2 others, forming a big loop.

Nice insight is finding degree of each vertex, how many edges vertex belong to.

Sum of degree, when not looped, will be equal to 2x # of edges, bc each edge

connected to 2 vertices.

\*/

//Find degree of vertex, returned as a VertexRDD of integers, degree of each vertex

val degrees: VertexRDD[Int] = topicGraph.degrees.cache()

degrees.map(\_.\_2).stats()

/\*

(count: 13721, mean: 31.155892, stdev: 65.497591, max: 2596.000000, min: 1.000000)

\*/

/\*

Something pecuilar, less degree than edges. That means some are not connected. Must

be due to some entries only having one main topic and does not divilge off.

\*/

//Find instances of single topic

val sing = medline.filter(x => x.size == 1)

sing.count()

//44509

val singTopic = sing.flatMap(topic => topic).distinct()

singTopic.count()

//8243

//There is 8243 distinct topics

//Remove instances within topic pair

val topic2 = topicPairs.flatMap(\_.getAs[Seq[String]](0))

singTopic.except(topic2).count()

//827

/\*

Lil math = 14,548–827 is 13,721, the number of entries in the degrees RDD.

Analyzing other stats, we see most aren't connected to much, mean is small fraction.

Lets analyze the high degree vertixes by innerJoin degrees VertexRDD to vertices

in concept graph. This will filter out non co-occuring concepts.

\*/

val namesAndDegrees = degrees.innerJoin(topicGraph.vertices) {

(topicId, degree, name) => (name, degree.toInt)

}.values.toDF("topic", "degree")

namesAndDegrees.orderBy(desc("degree")).show()

/\*

+-------------------+------+

| topic|degree|

+-------------------+------+

| Research| 2596|

| Disease| 1746|

| Neoplasms| 1202|

| Blood| 914|

| Pharmacology| 882|

| Tuberculosis| 815|

| Toxicology| 694|

| Drug Therapy| 678|

| Jurisprudence| 661|

|Biomedical Research| 633|

| Physicians| 625|

| Public Policy| 601|

| Medicine| 590|

| Metabolism| 578|

| Social Change| 570|

|Wounds and Injuries| 570|

| Brain| 569|

| Hospitals| 557|

| Urine| 551|

| Economics| 548|

+-------------------+------+

only showing top 20 rows

\*/

/\*

Unsuprising, the most common are most vague. Next section will use GraphX api and stats

to filter out less interesting

\*/

## //--Filtering Out Noisy Edges--//

/\*

As of right now, edges are weighted on count of how often a pair is. This doesnt mean

there is a semantic relationship present. Need new edge weighting scheme that

checks interest a pair concept is, given "overall prevalence of those concepts in

the data. We will use Pearson’s chi-squared test to calculate whether the occurrence

of a particular concept is independent from the occurrence of another concept"

With a pair of concepts A and B, create a 2x2 contigency table. The entries YY, YN,

& NN is raw count presence of A and B. The entries YA and NA are the row sums for

concept A, YB and NB are the column sums for concept B, and the value T is the

total number of documents.

yes b no b a total

yes a yy yn ya

no a ny nn na

b total yb nb t

x^2 = (T(|YY \* YN -YN \* NY| - T/2)^2)/(YA \* NA \* YB \* NB)

-T/2 is Yates continuity correlation

Higher the value, the more pair concept is interesting. We will calc this per pair

\*/

//-Processing EdgeTriplets-//

//Easiest to calculate, is T, total number of docs under consideration

val T = medline.count()

//240000

//Calculate how many documents feature each concept

val topicDistRdd = topicDist.map{ //Done earlier, but redoing as hashed version

case Row(topic: String, cnt: Long) => (hashId(topic), cnt)

}.rdd

//New graph

val topicDistGraph = Graph(topicDistRdd, topicGraph.edges)

/\*

To do calc, need to combine vertices - how often each concept appears in a document

& edges - how often each pair of concepts occurs in the same document. GraphX

support this operation with EdgeTriplet[VD,ED]

\*/

def chiSq(YY: Long, YB: Long, YA: Long, T: Long): Double = {

val NB = T - YB

val NA = T - YA

val YN = YA - YY

val NY = YB - YY

val NN = T - NY - YN - YY

val inner = math.abs(YY \* NN - YN \* NY) - T / 2.0

T \* math.pow(inner, 2) / (YA \* NA \* YB \* NB)

}

val chiSquaredGraph = topicDistGraph.mapTriplets(triplet => {

chiSq(triplet.attr, triplet.srcAttr, triplet.dstAttr, T)

})

chiSquaredGraph.edges.map(x => x.attr).stats()

/\*

(count: 213745, mean: 877.956648, stdev: 5094.935171, max: 198668.408387, min: 0.000000)

\*/

/\*

Range here is huge, STD proves this. Use aggresive thresholding/filtering.

"The 99.999th percentile of the chi-squared distribution with one degree

of freedom is approximately 19.5" - Use this to thresholding/filtering

\*/

//Threholding the 99.99th percentile

val interesting = chiSquaredGraph.subgraph(

triplet => triplet.attr > 19.5)

interesting.edges.count

/\*

res85: Long = 140575 //removed about a 1/3 edges

\*/

### //-Analyzing the Filtered Graph-//

val interestingComponentGraph = interesting.connectedComponents()

val icDF = interestingComponentGraph.vertices.toDF("vid", "cid")

val icCountDF = icDF.groupBy("cid").count()

icCountDF.count()

//878

icCountDF.orderBy(desc("count")).show()

/\*

+--------------------+-----+

| cid|count|

+--------------------+-----+

|-9218306090261648869|13610|

|-8193948242717911820| 5|

|-2062883918534425492| 4|

|-7016546051037489808| 3|

| 2742772755763603550| 3|

|-8679136035911620397| 3|

|-7685954109876710390| 3|

| -784187332742198415| 3|

| 1765411469112156596| 3|

|-9211944049288765106| 2|

|-4895960388347845016| 2|

|-2317423407077322989| 2|

|-3299226677350014771| 2|

|-6541817437503372447| 2|

|-5362458719777034637| 2|

|-3191983795676547449| 2|

| -697775734067750523| 2|

|-3467839743215210439| 2|

|-4717785562675251817| 2|

|-1046815223728304871| 2|

+--------------------+-----+

only showing top 20 rows

\*/

//Removing a third of data didn't didnt impact connectedness. Everything remainds same

val interestingDegrees = interesting.degrees.cache()

interestingDegrees.map(\_.\_2).stats()

//(count: 13721, mean: 20.490489, stdev: 29.864223, max: 863.000000, min: 1.000000)

//The mean has fallen down a bit

interestingDegrees.innerJoin(topicGraph.vertices) {

(topicId, degree, name) => (name, degree)

}.values.toDF("topic", "degree").orderBy(desc("degree")).show()

/\*

+--------------------+------+

| topic|degree|

+--------------------+------+

| Research| 863|

| Disease| 637|

| Pharmacology| 509|

| Neoplasms| 453|

| Toxicology| 381|

| Metabolism| 321|

| Drug Therapy| 304|

| Blood| 302|

| Public Policy| 279|

|Evaluation Studie...| 277|

| Social Change| 277|

| Jurisprudence| 253|

| Contraception| 245|

|Socioeconomic Fac...| 245|

|Family Planning S...| 244|

| Plants| 239|

| Economics| 237|

|Diagnosis, Differ...| 234|

| Brain| 229|

| Diagnosis| 227|

+--------------------+------+

only showing top 20 rows

\*/

//Look it effects of chi square ended up working. Removed generics and left related!

## //--Small-World Networks--//

/\*

“Collective Dynamics of ‘Small-World’ Networks” outlined the first mathematical mode

to generate graphs that has “small-world” properties in real-world graphs:

1. most nodes have small degree and belongs to dense cluster

2. can travel from any node to another from sets of edges

Researchers defined metric based on how strong these properties are true

\*/

### //-Cliques and Clustering Coefficients-//

/\*

Complete if every vertex is connected. In graph, may be subset that are complete,

called clique. Presence of many large cliques similar to locally dense real

small-world networks. Finding cliques is difficult and expensive.

One option is to use triangle count. "A triangle is a complete graph on three vertices,

and the triangle count at a vertex V is simply the number of triangles that contain V.

The triangle count is a measure of how many neighbors of V are also connected to each other."

New metric: local clustering coefficent: undirected graph,

local clustering coefficient C for a vertex that has k neighbors and t triangles is:

C = 2t / k(k-1)

\*/

val triCountGraph = interesting.triangleCount()

triCountGraph.vertices.map(x => x.\_2).stats()

//(count: 14548, mean: 74.660159, stdev: 295.327094, max: 11023.000000, min: 0.000000)

//normalize triangle count

val maxTrisGraph = interestingDegrees.mapValues(d => d \* (d - 1) / 2.0)

//join triangle count and normaliztion terms

val clusterCoef = triCountGraph.vertices.

innerJoin(maxTrisGraph) { (vertexId, triCount, maxTris) => {

if (maxTris == 0) 0 else triCount / maxTris //avoid dividing by 0

}

}

clusterCoef.map(\_.\_2).sum() / interesting.vertices.count() //network average clustering coefficient

//0.30624625605188605

## //--Computing Average Path Length with Pregel--//

/\*

Calculating path length between vertices is iterative. Each phase, vertices will

remember how many it knows and how fat away. Then it will communicate with neighboors

to add new vertices it didn't know. Keep doing this until no new info in list.

1. Figure out what state we need to keep track of at each vertex.

2. Write a function that takes the current state into account and evaluates each

pair of linked vertices to determine which messages to send at the next phase.

3. Write a function that merges the messages from all of the different vertices

before we pass the output of the function to the vertex for updating.

Store this list of collection of veritxes inside a Map[VertexId, Int] (int is ID)

for each vertex. The messages passed need to be in similar lookup table as well.

Need to create 2 functions:

1. merge the infrom from new messages

2. update function with state and message

\*/

def mergeMaps(m1: Map[VertexId, Int], m2: Map[VertexId, Int])

: Map[VertexId, Int] = {

def minThatExists(k: VertexId): Int = {

math.min(

m1.getOrElse(k, Int.MaxValue), //make sre to get a return regardleess

m2.getOrElse(k, Int.MaxValue))

}

(m1.keySet ++ m2.keySet).map {

k => (k, minThatExists(k))

}.toMap

}

def update(

id: VertexId,

state: Map[VertexId, Int],

msg: Map[VertexId, Int]) = {

mergeMaps(state, msg)

}

/\*

Final step to send messages to each vertex with neighhooring info each iteration

Each vertex increment its key by one, combine the values with neighbhoors using

mergeMaps, and send result to neighboor if it differs from their internal map.

\*/

def checkIncrement(

a: Map[VertexId, Int],

b: Map[VertexId, Int],

bid: VertexId) = {

val aplus = a.map { case (v, d) => v -> (d + 1) }

if (b != mergeMaps(aplus, b)) {

Iterator((bid, aplus))

} else {

Iterator.empty

}

}

//iterate both dst and src

def iterate(e: EdgeTriplet[Map[VertexId, Int], \_]) = {

checkIncrement(e.srcAttr, e.dstAttr, e.dstId) ++

checkIncrement(e.dstAttr, e.srcAttr, e.srcId)

}

//Prepare data for BSP run

val fraction = 0.005

val replacement = false

val sample = interesting.vertices.map(v => v.\_1).

sample(replacement, fraction, 1729L)

val ids = sample.collect().toSet

/\*

ids: scala.collection.immutable.Set[Long] = Set(4362101753053852075, -7060386309451085362,

1561067361034022813, 8423631057606637304, -8735611893949641035, -2994881691215263051,...

\*/

//Create new graph object with vertex map, nonempty if vertiex is member of sampled IDS

val mapGraph = interesting.mapVertices((id, \_) => {

if (ids.contains(id)) {

Map(id -> 0)

} else {

Map[VertexId, Int]()

}

})

//Initial message to send

val start = Map[VertexId, Int]()

val res = mapGraph.pregel(start)(update, iterate, mergeMaps)

//flatmap vertices to exract tuples of (VertexId, VertexId, Int) values that represent the unique path lengths that were computed:

val paths = res.vertices.flatMap { case (id, m) =>

m.map { case (k, v) =>

if (id < k) {

(id, k, v)

} else {

(k, id, v)

}

}

}.distinct()

paths.cache()

//compute summary stats and histogram of path lengths

paths.map(\_.\_3).filter(\_ > 0).stats()

//(count: 774117, mean: 3.495234, stdev: 0.762609, max: 8.000000, min: 1.000000)

//we ended up with shorter mean than textbook, ty new versions

val hist = paths.map(\_.\_3).countByValue()

hist.toSeq.sorted.foreach(println)

/\*

(0,60)

(1,2084)

(2,54329)

(3,340159)

(4,316531)

(5,57841)

(6,3043)

(7,126)

(8,4)

\*/

/\*

The MEDLINE citation showed very similar ranges and avg path length with other

small-world networks; however very high clustering, but low avg path length.

"In general, real-world graphs should exhibit the small-world property", if not

may be evidence of issue present.

\*/

# Extension Writeups

## Small-world Networks

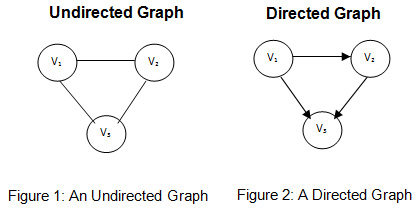
For this week, I decided I wanted to learn more about small-world networks since I have never ever been introduced to this before. After learning about implementation and how vertices and edges work in relation, I can finally understand how impressive and daunting pregel is and just how chaotic the current field is. What shocks me the most is its adaptability and growth, since all networks are constantly expanding (DRASTICALLY).

So the overarching idea in this chapter is that these networks are massive right. TOO massive, so to compensate for this, researchers want to design models that generate graphs and then researchers can compare these graphs to make sure they are similar to real-world networks. And as mentioned before, connected networks differ drastically in connectedness (all to one, or a loop). So due to this, there are 3 common models researchers use to generate small-world networks. Examples of small-worlds can be like road maps, food chains, power grid, and etc.

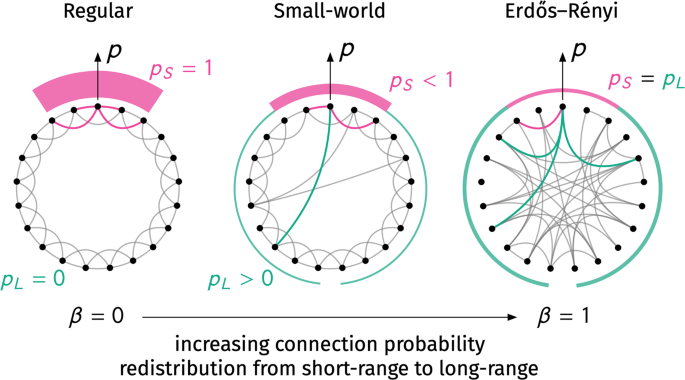
One of the biggest limitations of small-network models is that the randomly generated models are all undirected networks.

Undirected graphs edges don't have a direction. Whereas directed does.

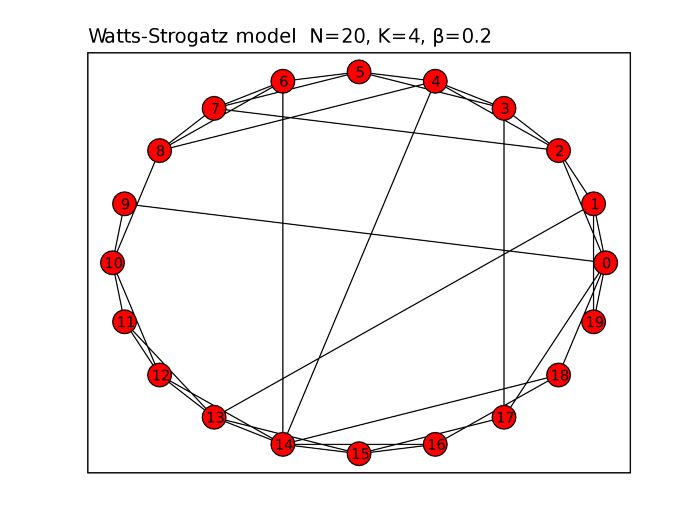
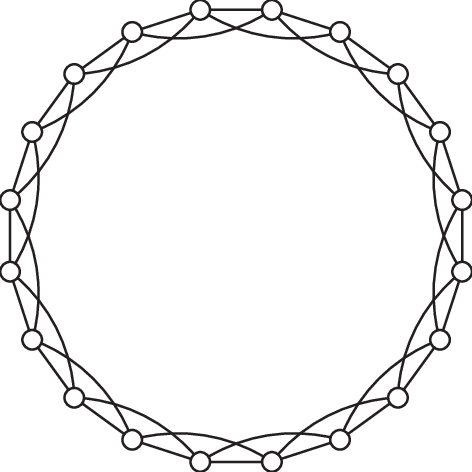
Examples of directed graphs are numerous, with one of the best examples being the world wide web. For example, hyperlinks! Great way to direct one node(website) to another(node).



1. Erdos-Renyi Model (random)
   1. Often called Random Graph Model - generates small networks randomly based on the probability distribution
      1. 2 variants
         1. G(n,p)
            1. Create an undirected network with n nodes and each edge is independently generated with probability of p
         2. G(n,m)
            1. Create an undirected network with n nodes and m edges selected at random
      2. Limitations:
         1. Due to it having a small clustering coefficient, which is quite different from real world networks, follow power-law
            1. Power laws: f(x) = C(1/x)α
         2. REAL-WORLD NETWORKS AREN”T RANDOMLY GENERATED



1. Watts-Strogatz Model (random)
   1. Have high clustering coefficient with short mean path length
      1. We get high clustering coefficients when using Regular Lattices and short average path length similar to that of an ER model.
      2. Get randomness in model by rewire with probability p while avoiding self loops and link duplication
      3. Examples: neural network of C elegans (worm), power grid, and network of movie actors.
      4. Algorithm: Number of nodes (N), mean degree K (even), and beta (0 < b < 1), construct a model with N nodes and NK/2 edges
         1. Create a regular ring lattice (with N nodes connected to K neighbours, K/2 on each side of the nodes.
         2. For every node 1, n-1, take every K/2 rightmost edges connected to target and rewire it with probability beta



* + 1. Limitations
       1. Unrealistic degree distribution which does not follow power-law
       2. Fix number of nodes (UNLIKE REAL WORLD WHERE NETWORK ALWAYS GROWING)

1. Barabasi-Albert Model
   1. The aforementioned are limited by randomness and lack of growing. Studying more created this model. Main points:
      1. Real world networks are growing, so it needs to be capable of continuous growth
      2. New nodes like to connect to nodes with high degree
   2. These 2 main points are why real world networks are scale-free property
      1. To build this network, we start with m0 nodes and links aren’t of focus, as long as each node is connected to another one.
      2. We add a new node to the network that has m < m0 links. This connects that node to the m nodes within the network, causing the network to grow continuously
   3. Preferential attachment - This allows the probability of a new nodes connection with node i directly correlates to i’s degree
   4. “The Barabasi-Albert model generates networks with scale-free property by adding continuous growth and preferential attachment characteristic in network generation.”
   5. GIF MODEL of growth-> https://miro.medium.com/max/600/1\*QuYD-YXvTdJcw0\_-9q\_fNw.gif

