Big Data Analytics

Chapter 6: Medical Network Analysis in GraphX

Following: [3] "**Advanced Analytics with Spark**", Chapter 7

Prof. Dr. Martin Theobald
Faculty of Science, Technology & Communication
University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

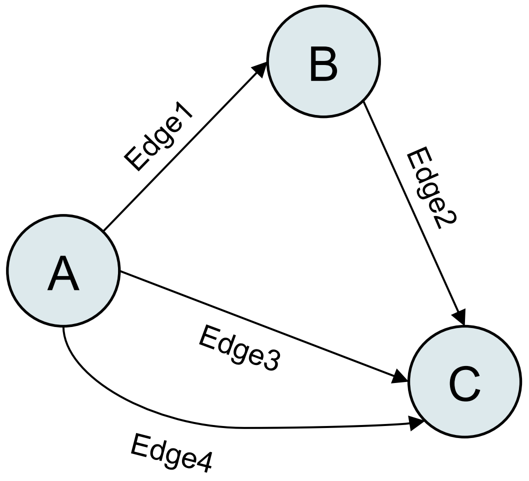
Spark's GraphX

- Just like MLlib, **GraphX** is part of the core Spark distribution.
- It provides a number of **parallel-processing algorithms** for **distributed directed labeled multi-graphs**, which are again stored as RDDs.
- **Core components** of the GraphX API:

```
val vertices: VertexRDD[<vertex-type>]
val edges: EdgeRDD[<edge-type>]

class Graph[<vertex-type>, <edge-type>] {
  val vertices: VertexRDD[<vertex-type>]
  val edges: EdgeRDD[<edge-type>] }

class EdgeTriplet[<vertex-type>, <edge-type>] {
  val srcAttr: <vertex-type>
  val dstAttr: <vertex-type>
  val attr: <edge-type> }
```



As before, instances of `VertexRDD` and `EdgeRDD` are automatically partitioned among all nodes in the Spark cluster.

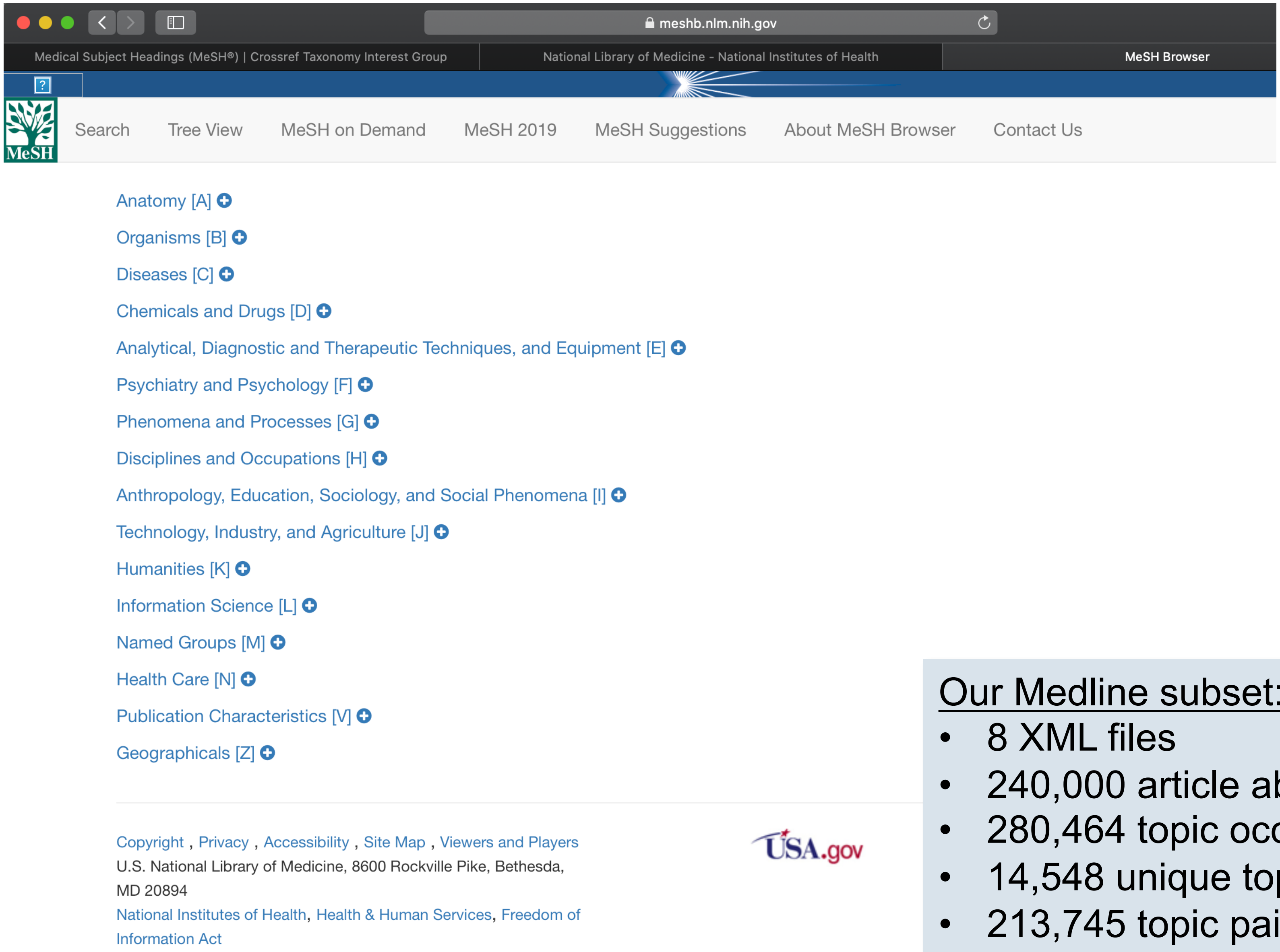
The MedLine Citation Index

- The Medline (Medical Literature Analysis and Retrieval System) is a huge database of medical research papers and clinical trials that have been published in the life sciences and in medicine.
- The main database contains more than 20 million medical publications.
- The provided dump from nlm.nih.gov which is available via FTP

```
wget ftp://ftp.nlm.nih.gov/nlmdata/sample/medline/*.gz
```

- contains metadata and abstracts of 240,000 Medline articles in XML format.
- Each paper entry is wrapped into a <MedlineCitation> element.
- The <MeshHeading> elements contain keywords capturing the major topics by which the papers were annotated according to the MeSH taxonomy and can viewed via the MeSH browser.

MeSH Taxonomy Structure


meshb.nlm.nih.gov
Medical Subject Headings (MeSH®) | Crossref Taxonomy Interest Group
National Library of Medicine - National Institutes of Health
MeSH Browser
MeSH
Search
Tree View
MeSH on Demand
MeSH 2019
MeSH Suggestions
About MeSH Browser
Contact Us

Anatomy [A]
Organisms [B]
Diseases [C]
Chemicals and Drugs [D]
Analytical, Diagnostic and Therapeutic Techniques, and Equipment [E]
Psychiatry and Psychology [F]
Phenomena and Processes [G]
Disciplines and Occupations [H]
Anthropology, Education, Sociology, and Social Phenomena [I]
Technology, Industry, and Agriculture [J]
Humanities [K]
Information Science [L]
Named Groups [M]
Health Care [N]
Publication Characteristics [V]
Geographicals [Z]

Copyright , Privacy , Accessibility , Site Map , Viewers and Players
U.S. National Library of Medicine, 8600 Rockville Pike, Bethesda, MD 20894
National Institutes of Health, Health & Human Services, Freedom of Information Act

USA.gov

Our Medline subset:

- 8 XML files
- 240,000 article abstracts
- 280,464 topic occurrences
- 14,548 unique topics
- 213,745 topic pairs

Load the XML Files into an RDD (I)

- The `cloudera.datascience` package contains (amongst others) rich APIs for parsing XML data.

```
import com.cloudera.datascience.common.XmlInputFormat

import scala.xml._
import org.apache.spark.SparkContext
import org.apache.spark.rdd._
import org.apache.hadoop.io.{Text, LongWritable}
import org.apache.hadoop.conf.Configuration
```

Load the XML Files into an RDD (II)

- The `newAPIHadoopFile` method of the `SparkContext` class supports custom delimiters for extracting parts of the contents of a text file into an RDD. We employ this to extract the Medline abstracts – one at a time – into an RDD of type `Array[String]`.

```
def loadMedline(sc: SparkContext, path: String) = {
  @transient val conf = new Configuration()
  conf.set(XmlInputFormat.START_TAG_KEY, "<MedlineCitation ")
  conf.set(XmlInputFormat.END_TAG_KEY, "</MedlineCitation>")
  val in = sc.newAPIHadoopFile(path, classOf[XmlInputFormat],
    classOf[LongWritable], classOf[Text], conf)
  in.map(line => line._2.toString) }

val medline_raw = loadMedline(sc, "./path-to-medline-files")
```

Basic XML Support in Scala

- As of version 1.2, Scala considers **XML as a first-class data type** (just like `Int`, `String`, etc.). Thus, the following code is syntactically valid:

```
import scala.xml._
val data = "My Medline Citation Node."
val citation = <MedlineCitation>data</MedlineCitation>
```

- After loading the Medline data into an XML element of type `scala.xmlElem`:

```
val raw_xml = medline_raw.take(1)(0)
val elem = XML.loadString(raw_xml)
```

the following **path operations** can be performed directly on that element:

```
elem.label
elem.attributes
elem \ "MeshHeadingList"
(elem \\ "DescriptorName").map(_.text)
```

Parse the MeSH Codes from the XML Format

- A "**major topic**" is denoted by a special markup tag in the Medline XML format. We extract all of these fields for each citation record in our RDD and cache the result:

```
def majorTopics(elem: Elem): Seq[String] = {
  val dn = elem \\ "DescriptorName"
  val mt = dn.filter(n => (n \ "@MajorTopicYN").text == "Y")
  mt.map(n => n.text) }
majorTopics(elem)
```

- And let's apply this function to the entire set of XML files via an RDD transformation:

```
val mxml: RDD[Elem] = medline_raw.map(XML.loadString)
val mesh_topics: RDD[Seq[String]] = mxml.map(majorTopics).cache()
mesh_topics.take(1)(0)
```

Analyze the MeSH Major Topics and Their Co-Occurrences

- A quick count over the topics reveals their **absolute frequencies**:

```
mesh_topics.count()
val topics: RDD[String] = mesh_topics.flatMap(mesh => mesh)
val topicCounts = topics.countByValue()
topicCounts.size

val tcSeq = topicCounts.toSeq
tcSeq.sortBy(_._2).reverse.take(10).foreach(println)
```

- First analyze the topic distribution by a simple frequency count:

```
val valueDist = topicCounts.groupBy(_._2).mapValues(_.size)
valueDist.toSeq.sorted.take(20).foreach(println)
```

- Show also some of the "long tail" topics:

```
tcSeq.sortBy(_._2).take(100).foreach(println)
```

Analyze the Co-Occurrences of Topics

- Next, we consider **pairs of topics that co-occur in a same Medline abstract**. The function `combinations(2)` achieves this effect very conveniently for us:

```
val topicPairs = mesh_topics.flatMap(t =>
     t.sorted.combinations(2))
val cooccurs = topicPairs.map(p => (p, 1)).reduceByKey(_+_)
```

- The results are up to $n\,(n-1)/2$ topic pairs. This is going to be a frequently accessed RDD, so we should cache it:

```
cooccurs.cache()
cooccurs.count()
```

Notes:

- Although the GraphX APIs support *directed* graphs, all of the following analyses consider the underlying mesh-topic graph as *undirected*.
- For an undirected graph with n vertices, the maximum amount of unique edges (excluding self-loop edges) is given by the **binomial coefficient** $n\,(n-1)/2$.

Option 1: Choose a Good Hashing Function for Vertices

- The VertexRDD and EdgeRDD objects we defined in the GraphX API require vertices to be identified by a unique `Long` value rather than by a `String`.

- Thus, for building the co-occurrence graph among major topics in the Medline articles, we need to first **turn** the **topic names** into **unique topic ids**:

```
import com.google.common.hash.Hashing

def getTopicId(str: String) = {
  Hashing.md5().hashString(str).asLong() }
```

`Hashing.md5` is an implementation of the MD5 message digest algorithm specification.

Option 2: Use a Counter for Unique Vertex Ids

- If we do not trust our hashing function, we may also **explicitly map** the topic string **to a unique id by using a counter**.

```
val topicIds =
  topics.zipWithUniqueId().collectAsMap()
val bTopicIds = sc.broadcast(topicIds).value

def getTopicId(str: String) = {
  bTopicIds(str) }
```

- Note that the mapping from the string to a unique integer is backed by a `HashMap` in Scala/Java. Thus, if there is a collision within this `HashMap`, the map interface still guarantees that the correct value is returned for each key.

- Hence, a `HashMap` actually stores a list of key/value pairs for each hash-code rather than just the hash-code/value pairs.

GraphX: Create the Graph Vertices as an RDD

- The unique MeSH topics from the Medline articles will form the vertices of our **co-occurrence network** among the major topics we found within a same Medline abstract.

- We start by importing the GraphX libraries:

```
import org.apache.spark.graphx._
```

- However, before we continue, we may wish to first verify that the assigned topic ids are indeed unique:

```
val vertices = topics.map(topic => (getTopicId(topic), topic))
val uniqueHashes = vertices.map(_._1).countByValue()
val uniqueTopics = vertices.map(_._2).countByValue()
uniqueHashes.size == uniqueTopics.size
```

GraphX: Create the Edges as another RDD

- Thus, edges are built from the unique ids and are based on the pairwise co-occurrences of MeSH topics in the Medline articles:

```
val edges = cooccurs.map(p => {
  val (topics, cnt) = p
  val ids = topics.map(getTopicId).sorted
  Edge(ids(0), ids(1), cnt) })
```

- We first initialize the `Graph` object in GraphX as a container for our vertices and edges. Also this data structure should be cached:

```
val topicGraph = Graph(vertices, edges)
topicGraph.cache()
```

- And we quickly analyze the graph's main properties:

```
vertices.count()
topicGraph.vertices.count()
topicGraph.edges.count()
```

More Advanced Network Analysis Algorithms

1. **Connected Components**

   A (strongly) connected-components analysis of a (directed) graph computes all subgraphs in which every pair of nodes is connected to each other. For both an undirected and a directed graph, these can be computed in linear time in the number of vertices and edges in the graph by using, for example, a simple a breath-first search over all edges and by storing the set of visited vertices for each intermediate vertex.

2. **Degree Distribution**

   The degree distribution is a coarse measure for how densely connected the graph is. Its average may range between 0 (no edges) and $\#vertices$ (fully connected).

3. **Cliques/Triangles & Clustering Coefficients**

   The clustering coefficient of vertices in a graph is a more detailed measure for how densely connected the graph is (based on the number of triangles each node participates in). Its average may range between 0 (no edges) and 1 (fully connected).

4. **Diameter & Average Path Length**

   Diameter and average path length require an iterative processing starting from all vertices either via breath- or depth-first search. The maximum number of iterations from any source to any (formerly unseen) target corresponds to the diameter of the graph.

1. Connected Components Analysis (I)

- A **connected component** of a graph is a subgraph in which every pair of nodes is reachable to each other.
- Fortunately, connected components can be computed very conveniently via a built-in API function of GraphX:

```
val connectedComponentGraph: Graph[VertexId, Int] =
  topicGraph.connectedComponents()
```

- We can also conveniently count in how many connected components each vertex participates:

```
def sortedConnectedComponents(
    connectedComponents: Graph[VertexId, _])
      : Seq[(VertexId, Long)] = {
      val componentCounts =
        connectedComponents.vertices.map(_._2).countByValue
      componentCounts.toSeq.sortBy(_._2).reverse }
```

1. Connected Components Analysis (II)

- We can also count the size of each connected component in the graph:

```
val componentCounts = sortedConnectedComponents(
  connectedComponentGraph)
componentCounts.size
componentCounts.take(10).foreach(println)
```

- The final result is obtained by an (inner) join between the two vertex sets:

```
val nameCID = topicGraph.vertices.
  innerJoin(connectedComponentGraph.vertices) {
    (topicId, name, componentId) => (name, componentId) }
```

Exploring the Connected Components

- Let's have a look at the second-largest component in the graph:

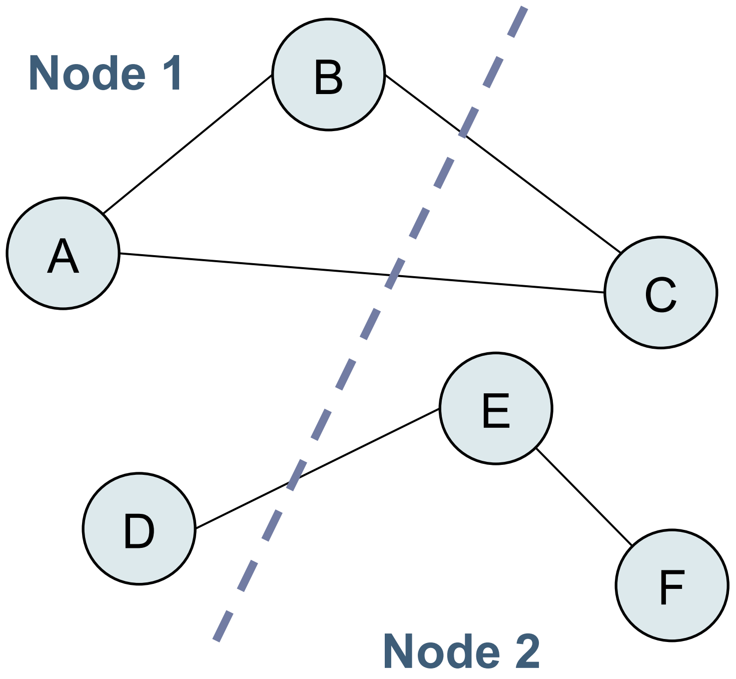
```
val c1 = nameCID.filter(x => x._2._2 == topComponentCounts(1)._2)
c1.collect().foreach(x => println(x._2._1))
```

- Compare these to the topic names containing the substring "HIV":

```
val hiv = topics.filter(_.contains("HIV")).countByValue()
hiv.foreach(println)
```

Iterative & Distributed Connected-Components Analysis

```
1.  Initialize each vertex state with
    a unique number (e.g. a counter or id);
2. Do:
3.  For all vertices in parallel do:
4.    Send own state to all neighbors;
5.  For all vertices in parallel do:
6.    Receive all neighbors' states;
7.    Set new state to be the minimum
        among own and all incoming states;
8.  While "any node state changed";
9.  Connected components are sets of
    vertices with same state;
```



Initialization

Vertex	State
A	1
B	2
C	3
D	4
E	5
F	6

Iteration 1

Vertex	State
A	1
B	1
C	1
D	4
E	4
F	5

Iteration 2

Vertex	State
A	1
B	1
C	1
D	4
E	4
F	4

Note:
In a distributed setting, this form of iterative computation requires state exchange (and hence communication) along the cut edges.

2. Analyzing the Degree Distribution (I)

- GraphX provides the `degrees` method which returns a `VertexRDD` of integers that captures the degree of each vertex:

```
val degrees: VertexRDD[Int] = topicGraph.degrees.cache()
degrees.map(_._2).stats()
```

- Compare the number of these RDDs with the number of the singleton topics, the ones that have no incoming or outgoing edges.

```
val singletonTopicGroups = mesh_topics.filter(x => x.size == 1)
singletonTopicGroups.count()

val singletonTopics = singletonTopicGroups.flatMap(mesh =>
  mesh).distinct()
singletonTopics.count()

val flatTopics = topicPairs.flatMap(p => p)
singletonTopics.subtract(flatTopics).count()
```

2. Analyzing the Degree Distribution (II)

- Let's take a closer look at the high-degree vertices by
  - joining the `degrees` VertexRDD to the vertices in the topic graph and
  - combining the topic name and the degree of the vertex into a tuple.

```
def topNamesAndDegrees(degrees: VertexRDD[Int],
    topicGraph: Graph[String, Int]): Array[(String, Int)] = {
  val namesAndDegrees = degrees.innerJoin(topicGraph.vertices) {
    (topicId, degree, name) => (name, degree) }

  val ord = Ordering.by[(String, Int), Int](_._2)
  namesAndDegrees.map(_._2).top(10)(ord) }
```

- And finally pretty-print the resulting tuples:

```
topNamesAndDegrees(degrees, topicGraph).foreach(println)
```

Digression: Filtering Out Noisy Edges

- In our current co-occurrence graph, the edges are weighted based on the **count** of **how often a pair of major topics** appears within a same abstract.
- The problem with this simple weighting scheme is that it does not **distinguish** topic pairs that occur together because they have a **meaningful semantic relationship** from topic pairs that occur together because they happen to **both occur frequently** for any type of abstract.
- We may thus apply the X^2 ("**Chi-Square**") **test of independence** to distinguish frequently co-occurring from just randomly (i.e., "**independently**") co-occurring topic pairs.
- For co-occurrences among two topics A and B into the four categories YY, YN, NY, NN, we can determine the coefficient of the X^2 test statistic as follows:

$$X^2 = N\frac{(|YY \times NN - YN \times NY| - N/2)^2}{(YA \times NA \times YB \times NB)}$$

where the values are obtained from a respective **contingency table** (see rhs.) among A and B.

	Yes B	No B	A Total
Yes A	YY	YN	YA
No A	NY	NN	NA
B Total	YB	NB	N

Digression: The X^2 Test Statistic

- The value of the test statistic is computed as follows:

```
def chiSq(YY: Int, YB: Int, YA: Int, N: Long): Double = {
  val NB = N - YB
  val NA = N - YA
  val YN = YA - YY
  val NY = YB - YY
  val NN = N - NY - YN - YY
  val inner = (YY * NN - YN * NY) - N / 2.0
  N * math.pow(inner, 2) / (YA * NA * YB * NB) }
```

- These new weights are assigned to all edges in the graph:

```
val N = mesh_topics.count()
val chiSquaredGraph = topicCountGraph.mapTriplets(triplet => {
  chiSq(triplet.attr, triplet.srcAttr, triplet.dstAttr, N) })

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

Digression: The X^2 Test

- For a contingency table with r rows and c columns, the basic parameter of the X^2 distribution, called the "**degrees of freedom**", is computed as $(r-1)(c-1)$. For our 2-by-2 contingency table, this is just 1.

- The ***p*-value** of the X^2 test is defined as one minus the area under the X^2 distribution for the computed value of the test statistic at 1 degree(s) of freedom (see next slide).

- Formally, the *p*-value is defined as the probably that the co-occurrence frequencies of topics A and B are "**at least as extreme**" as the ones we observe in our graph, given that we assume that A and B indeed occur independently of each other.

- If this *p*-value is very small, say less than 1%, we may reject the **null hypothesis** stating that A and B just occur independently and thus keep the respective edge in our graph because the pair may actually be a meaningful one.

Digression: The X^2 Distribution

1.1.2.10. Obere 100α-prozentige Werte χ^2_α der χ^2-Verteilung (s. 5.2.3.)

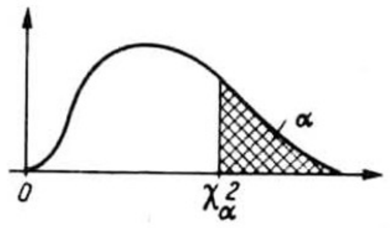
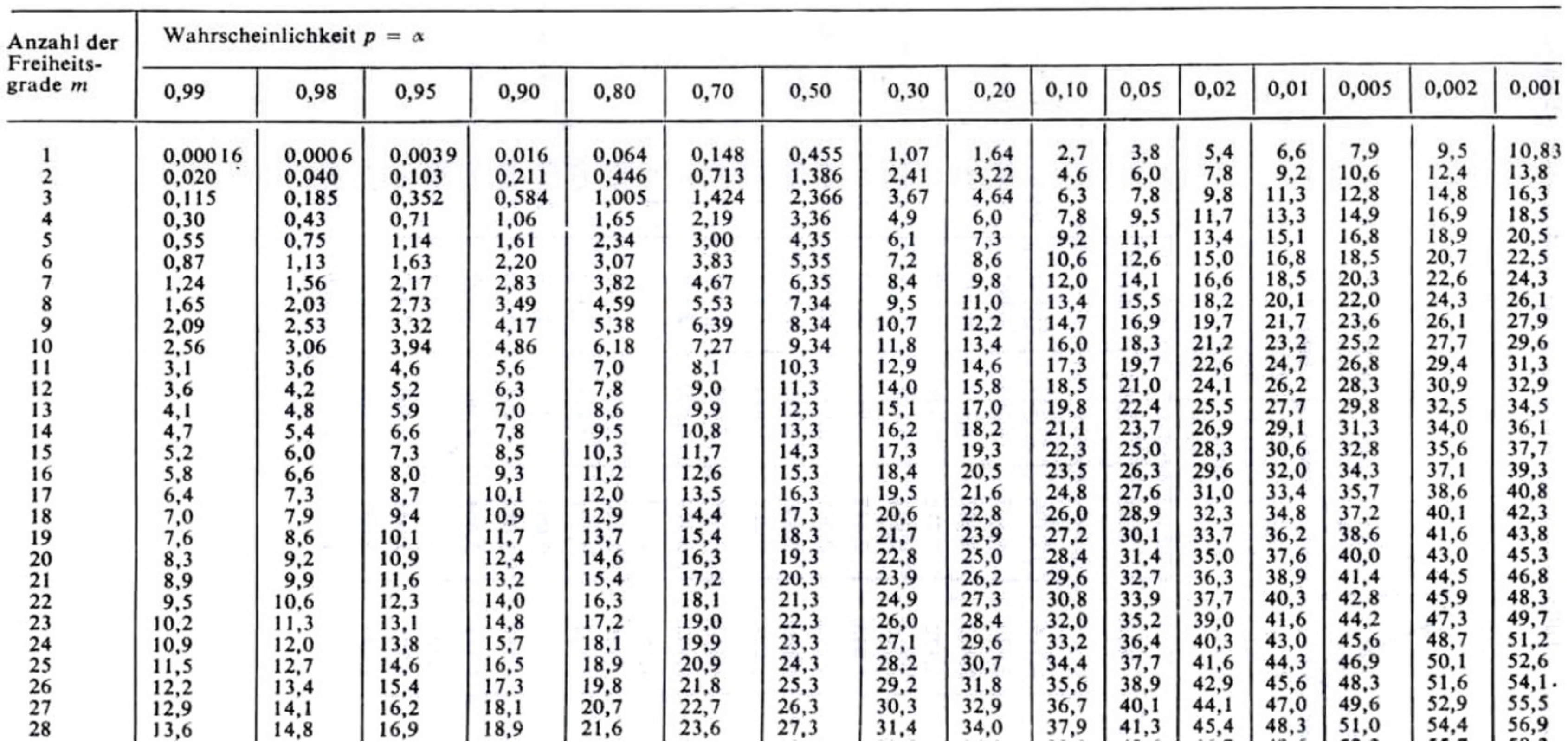


Abb. 1.4



Anzahl der Freiheits-grade m	Wahrscheinlichkeit $p = \alpha$															
	0,99	0,98	0,95	0,90	0,80	0,70	0,50	0,30	0,20	0,10	0,05	0,02	0,01	0,005	0,002	0,001
1	0,00016	0,0006	0,0039	0,016	0,064	0,148	0,455	1,07	1,64	2,7	3,8	5,4	6,6	7,9	9,5	10,83
2	0,020	0,040	0,103	0,211	0,446	0,713	1,386	2,41	3,22	4,6	6,0	7,8	9,2	10,6	12,4	13,8
3	0,115	0,185	0,352	0,584	1,005	1,424	2,366	3,67	4,64	6,3	7,8	9,8	11,3	12,8	14,8	16,3
4	0,30	0,43	0,71	1,06	1,65	2,19	3,36	4,9	6,0	7,8	9,5	11,7	13,3	14,9	16,9	18,5
5	0,55	0,75	1,14	1,61	2,34	3,00	4,35	6,1	7,3	9,2	11,1	13,4	15,1	16,8	18,9	20,5
6	0,87	1,13	1,63	2,20	3,07	3,83	5,35	7,2	8,6	10,6	12,6	15,0	16,8	18,5	20,7	22,5
7	1,24	1,56	2,17	2,83	3,82	4,67	6,35	8,4	9,8	12,0	14,1	16,6	18,5	20,3	22,6	24,3
8	1,65	2,03	2,73	3,49	4,59	5,53	7,34	9,5	11,0	13,4	15,5	18,2	20,1	22,0	24,3	26,1
9	2,09	2,53	3,32	4,17	5,38	6,39	8,34	10,7	12,2	14,7	16,9	19,7	21,7	23,6	26,1	27,9
10	2,56	3,06	3,94	4,86	6,18	7,27	9,34	11,8	13,4	16,0	18,3	21,2	23,2	25,2	27,7	29,6
11	3,1	3,6	4,6	5,6	7,0	8,1	10,3	12,9	14,6	17,3	19,7	22,6	24,7	26,8	29,4	31,3
12	3,6	4,2	5,2	6,3	7,8	9,0	11,3	14,0	15,8	18,5	21,0	24,1	26,2	28,3	30,9	32,9
13	4,1	4,8	5,9	7,0	8,6	9,9	12,3	15,1	17,0	19,8	22,4	25,5	27,7	29,8	32,5	34,5
14	4,7	5,4	6,6	7,8	9,5	10,8	13,3	16,2	18,2	21,1	23,7	26,9	29,1	31,3	34,0	36,1
15	5,2	6,0	7,3	8,5	10,3	11,7	14,3	17,3	19,3	22,3	25,0	28,3	30,6	32,8	35,6	37,7
16	5,8	6,6	8,0	9,3	11,2	12,6	15,3	18,4	20,5	23,5	26,3	29,6	32,0	34,3	37,1	39,3
17	6,4	7,3	8,7	10,1	12,0	13,5	16,3	19,5	21,6	24,8	27,6	31,0	33,4	35,7	38,6	40,8
18	7,0	7,9	9,4	10,9	12,9	14,4	17,3	20,6	22,8	26,0	28,9	32,3	34,8	37,2	40,1	42,3
19	7,6	8,6	10,1	11,7	13,7	15,4	18,3	21,7	23,9	27,2	30,1	33,7	36,2	38,6	41,6	43,8
20	8,3	9,2	10,9	12,4	14,6	16,3	19,3	22,8	25,0	28,4	31,4	35,0	37,6	40,0	43,0	45,3
21	8,9	9,9	11,6	13,2	15,4	17,2	20,3	23,9	26,2	29,6	32,7	36,3	38,9	41,4	44,5	46,8
22	9,5	10,6	12,3	14,0	16,3	18,1	21,3	24,9	27,3	30,8	33,9	37,7	40,3	42,8	45,9	48,3
23	10,2	11,3	13,1	14,8	17,2	19,0	22,3	26,0	28,4	32,0	35,2	39,0	41,6	44,2	47,3	49,7
24	10,9	12,0	13,8	15,7	18,1	19,9	23,3	27,1	29,6	33,2	36,4	40,3	43,0	45,6	48,7	51,2
25	11,5	12,7	14,6	16,5	18,9	20,9	24,3	28,2	30,7	34,4	37,7	41,6	44,3	46,9	50,1	52,6
26	12,2	13,4	15,4	17,3	19,8	21,8	25,3	29,2	31,8	35,6	38,9	42,9	45,6	48,3	51,6	54,1
27	12,9	14,1	16,2	18,1	20,7	22,7	26,3	30,3	32,9	36,7	40,1	44,1	47,0	49,6	52,9	55,5
28	13,6	14,8	16,9	18,9	21,6	23,6	27,3	31,4	34,0	37,9	41,3	45,4	48,3	51,0	54,4	56,9

Digression: Generate the Filtered Graph

- We filter out all edges whose ***p*-values are below 1%** (this confirms to a value of the test statistic of about 6.6 at 1 degree of freedom):

```
val interesting = chiSquaredGraph.subgraph(
  triplet => triplet.attr > 6.6)
interesting.edges.count
```

Digression: Further Analyze the Filtered Graph

1. **Connected components** of the filtered graph:

```
val interestingComponentCounts = sortedConnectedComponents(
  interesting.connectedComponents())
interestingComponentCounts.size
interestingComponentCounts.take(10).foreach(println)
```

2. **Degree distribution** of the filtered graph:

```
val interestingDegrees = interesting.degrees.cache()
interestingDegrees.map(_._2).stats()
topNamesAndDegrees(interestingDegrees,
  topicGraph).foreach(println)
```

3. Cliques & Clustering Coefficients (I)

- A **clique** is a fully connected subgraph in which every pair of nodes is directly connected via an edge with each other.
- A **triangle count** thus counts all cliques that consist of exactly three vertices:

```
val triangleCountGraph = topicGraph.triangleCount()
triangleCountGraph.vertices.map(x => x._2).stats()
```

3. Cliques & Clustering Coefficients (II)

- If have computed the triangle count of the graph, then the **local clustering coefficient** C for a vertex that has k neighbors (for $k > 1$) and takes part in t triangles is defined as follows:

$$C = \frac{2t}{k(k-1)}$$

- We can then compute the **average local clustering coefficient** over all vertices in the graph in Spark as follows:

```
val maxTriangleGraph = topicGraph.degrees.mapValues(
  d => d * (d - 1) / 2.0)
val clusterCoefficientGraph = triangleCountGraph.vertices.
  innerJoin(maxTriangleGraph) {
    (vertexId, triCount, maxTris) => {
      if (maxTris == 0) 0 else triCount / maxTris } }
clusterCoefficientGraph.map(_._2).sum() /
     topicGraph.vertices.count()
```

4. Average Path Length in GraphX (I)

Pregel was the first MapReduce-based graph engine that introduced the concept of **node-centric computing**. The average path length can be implemented as follows:

1. Each vertex v stores a local map of reachable vertices and their distances to v.
2. Messages sent from v to its successors contain the map of v (including v itself) with an added distance of 1.
3. Incoming messages at each vertex are merged into a new local map using the minimum distance of v to any node in the incoming maps.

```
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),
       m2.getOrElse(k, Int.MaxValue))
  }
  (m1.keySet ++ m2.keySet).map {
     k => (k, minThatExists(k)) }.toMap }
```

4. Average Path Length in GraphX (II)

- `Update` is part of the Pregel API (now simulated in GraphX) and in this case just calls the `mergeMaps` function to merge the local map with each incoming map:

```
def update(id: VertexId, state: Map[VertexId, Int],
    msg: Map[VertexId, Int]) = {
  mergeMaps(state, msg) }
```

4. Average Path Length in GraphX (III)

- The `checkIncrement` function increments the distances in the local map of each vertex v by 1, merges the local map with the one from an incoming message using `mergeMaps`, and sends the results of the `mergeMaps` function to the successors of v.

```
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
  }
}
```

4. Average Path Length in GraphX (IV)

- `iterate` is another Pregel function (now simulated in GraphX), that merges the iterators returned by `checkIncrement` in both forward and backward direction, hence we perform both a **forward and backward breadth-first search** (BFS) over the graph.

```
def iterate(e: EdgeTriplet[Map[VertexId, Int], _]) = {
  checkIncrement(e.srcAttr, e.dstAttr, e.dstId) ++
    checkIncrement(e.dstAttr, e.srcAttr, e.srcId)
}
```

4. Average Path Length in GraphX (V)

- With the afore defined functions at hand, we could now compute **the path length between all pairs of nodes** (if connected) in the entire graph, which will consume $O(\#vertices^2)$ memory in the Spark cluster.
- For a large graph, this may quickly grow very large, such that we resort to using a **vertex-induced subgraph** with consisting of 2% of randomly sampled vertices of our topic graph to approximate the average path length:

```
val sampleVertices = topicGraph.vertices.map(v =>
     v._1).sample(false, 0.02).collect().toSet

val mapGraph = interesting.mapVertices((id, _) => {
  if (sampleVertices.contains(id)) {
    Map(id -> 0)
  } else {
    Map[VertexId, Int]()
  } })
```

4. Average Path Length in GraphX (VI)

- Start the Pregel-style form of **iterative breadth-first search**:

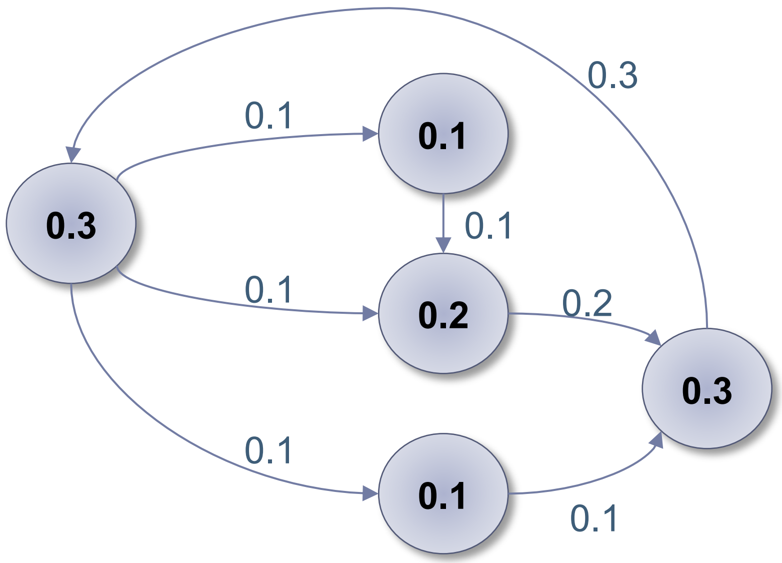
```
val start = Map[VertexId, Int]()
val res = mapGraph.pregel(start)(update, iterate, mergeMaps)
```

- And finally **extract the path lengths** (as distinct pairs of vertices with their distances) from all the local maps stored at the vertices:

```
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()
paths.map(_._3).filter(_ > 0).stats()
val hist = paths.map(_._3).countByValue()
hist.toSeq.sorted.foreach(println)
```

Side Note: PageRank in GraphX (I)

- The **PageRank algorithm** (Brin, Page: 1998) was the initial break-through for Google's ranking algorithm.
- It considers the hyperlinks in a web crawl as edges of a directed graph and measures the importance of each vertex in this graph, assuming an **edge** from u to v **represents an endorsement** of v's importance by u.
- For example, if a web page v is linked to by many other important pages $u_1, \ldots, u_n$, then v itself becomes more important.



(not showing random jumps)

- Mathematically, this recursive dependency can be expressed as follows

$$PageRank(v) = \frac{\varepsilon}{N} + \sum_{u_i \in\, inlinks(v)} \frac{PageRank(u_i)}{|outlinks(u_i)|}$$

where ε is a "**random-jump probability**" (usually 0.05 – 0.15) and N is the number of all vertices in the graph.

Side Note: PageRank in GraphX (II)

- Fortunately, PageRank is readily implemented in GraphX:

```
val ranks = graph.pageRank(0.0001, 0.15).vertices
```

  The first parameter is a threshold for the convergence (i.e., the sum of the absolute differences in PageRank values of all vertices between two iterations), the second one is the random-jump probability ε.

☞ See Moodle for a direct implementation of **PageRank in Spark using regular RDDs**!

- Once more, sort and pretty-print the results:

```
val namesAndRanks = ranks.innerJoin(topicGraph.vertices) {
  (topicId, rank, name) => (name, rank) }
val ord = Ordering.by[(String, Double), Double](_._2)
namesAndRanks.map(_._2).top(20)(ord).foreach(println)
```

Summary

- GraphX is a great extension of the Spark core libraries that is optimized for **processing distributed graph algorithms** in parallel.

- A graph is **automatically sharded** on both vertices and edges (both stored internally as RDDs).

- The term "**node-centric computing**" was first coined in the context of Pregel (and implemented in MapReduce):
  - Each vertex has a *local state* only.
  - Each vertex communicates only with its *immediate neighbors* (both successors and predecessors are allowed for communication).
  - Messages among neighboring vertices are *iteratively exchanged* and *merged* among the neighboring vertices.

- Graph-based data representations are **ubiquitous** in everyday life (social nets, citations & references, web graphs, etc.)

- See http://snap.stanford.edu for a wide collection of **real-world** (both small and large-scale) **graphs**!