PageRank applied to the 2016 TFL Santander Cycle Hire Journey Data

IS71059B Assignment 2: John Downing

Contents

PageRank applied to the 2016 TFL Santander Cycle Hire Journey Data 1
- Introduction 2
- GraphX 2
- PageRank 3
- Spark SQL 4
- Gephi/Sigmajs 4
- Putting it all together 5
- Running 7
- Results 9
- Discussion 14
- References 15
- Appendices 16
  - Code 16
  - Raw TFL data 25

Introduction

This assignment follows on from earlier efforts[1] to analyse the 2016 usage data for London's *Santander Cycles* bike-sharing scheme. The scheme, introduced in July 2010, provides self-service cycle hire for the purposes of making short-term[2], point-to-point[3], journeys. Journey history is freely available as part of TFL's Open Data programme, and although basic in nature – the raw data consisting of little more than start and end locations, and timestamps – offers the potential for a range of analyses.

Earlier efforts focused on visualisation and descriptive statistics, and despite the (relatively large) size of the dataset (some 9.45 million rows), did not make use of big data technologies. I was initially reluctant to re-use this data, wanting to avoid e.g. simply re-implementing "journey count" in MapReduce. However, a brief section in *Learning Spark* (Karau, Konwinski, Wendell & Zaharia, 2015) alerted me to the possibility of applying graph algorithms to the network of journey data – and a different kind of model. The proposal for this project, then, became to apply the PageRank algorithm to the cycle hire data – as both a ready-made use case for exploring these aspects of the Spark stack, and also in the hope of discovering some new insights along the way.

GraphX

Leaning Spark only referenced GraphX in passing, however a quick web search pointed strongly in its direction – and it immediately felt like a good fit, being (a) available out of the box, with the standard Spark distribution, and (b) having a built-in implementation of the PageRank algorithm. However, the documentation and examples on the Spark web site[4] did not offer much by way of an easy way in. Happily, Manning publish *Spark GraphX In Action* (Malak & East, 2016), which proved invaluable – not only with regards to the specifics of GraphX, but also with the basics of graph theory, with Scala (since the GraphX API is only available via Scala), and with third party visualisation tools such as Gephi. More of which later.

Malak & East (and see also e.g. Robinson, Webber & Eifrem, 2015) are keen to stress the importance of graphs as an alternative data structure to more traditional, tabular, approaches – as might be found in an RDBMS database or a Pandas DataFrame. Graphs elevate the connections between entities to primacy in a network model; an inversion of the idea that a problem domain is

---

[1] My third assignment from IS53048A: Data Visualisation. The full report of this is available at https://github.com/downinja/MSc-Data-Science/blob/master/data visualisation/Santander Cycle Hire Usage Statistics.pdf

[2] The maximum permitted rental period is 1 day; above this TFL can impose a fine.

[3] In the sense that a rental period is a single occurrence of travel between two docking stations, although these might be the same station.

[4] http://spark.apache.org/docs/latest/graphx-programming-guide.html

modelled as a collection of entity tables related by foreign keys. Although it is possible to represent connections between entities using a join table in RDBMS, the SQL involved in finding e.g. friends of friends of friends quickly becomes unwieldy – whereas a graph processing system is designed specifically for the traversal of such relationships.

PageRank

PageRank was named after Larry Page[5], the co-founder of Google, and was initially designed to determine the importance of web pages for use in displaying search results. Pages with higher rank have not just – or not even – a higher number of web links pointing to them, as compared with other pages; rather, they have a higher number of pages pointing to them which are *themselves* of higher rank. PageRank was chosen, here, primarily because of the (loose) analogies between the structure of the cycle network and the web, rather than because of any specific hypotheses about its applicability – in the sense that on the one hand we have web pages and links to other web pages, and on the other we have docking stations and journeys to other docking stations. However, there have been a number of studies (e.g. Chan & Teknomo, 2016; Crisostomi, Kirkland & Shorten, 2011) which have attempted to model the dynamics of road networks using PageRank, and so it is at least plausible that this approach may be valid.

The algorithm, in its iterative form, proceeds by

- Assigning an initial, equal, rank[6] to all nodes (vertices) in the network
- Looping around the following steps until convergence
    - Having each vertex distribute its current rank across all vertices to which it has an outgoing edge (that is, transmit a message containing the numerical value `currentRank/M` – where `M` is its "out-degree").
    - Having all vertices sum their incoming messages, which become their new ranks[7].

A complement to this is *Personalized PageRank*, which is designed to rank vertices from the perspective of a particular starting point. In this variant, rank does not propagate too many steps from the starting point – providing a more localised view of influence.

---

[5] https://web.archive.org/web/20010715123343/https://www.google.com/press/funfacts.html

[6] Malak & East (2016) suggest 1/N, where N is the number of vertices. However, Karau, Konwinski, Wendell & Zaharia (2015) simply set all initial ranks to 1.

[7] There's an additional "damping factor" involved in the calculation, so that rankings of less important pages do not shrink to zero, and so that the algorithm can escape a vertex with zero out-degree.

Spark SQL

Spark SQL is a module which sits on top of core Spark and provides a SQL-like interface to data that has a known schema. It provides a DataFrame interface, which is (presumably[8]) similar in intent to those in Python and R, and which is optimised to work with structured data using the Tungsten and Catalyst libraries – rather than operating on raw RDDs directly. The motivation for using it in this assignment was partly for comparison with Hive, in the Hadoop stack, but also to get an idea of how the results of processing might be used outside of GraphX.

Gephi/Sigmajs

Gephi is an open source desktop application for visualising graphs and networks. *Spark GraphX In Action* provides sample code for serialising to Graph Exchange XML Format (GEXF[9]), and so this seemed like a natural choice for plotting the PageRank-ed results – which, although not strictly necessary, allowed for a much easier comparison with the findings from my Data Visualisation assignment.

Sigmajs is similarly a tool for graph visualisation, however this time is a browser based Javascript library, with support for the GEFX file format. (Sigma actually provides a plug-in for Gephi which exports a complete Javascript application, ready to use). Strictly speaking, it should be possible to load hand-crafted GEFX directly into Sigma – however Gephi adds a number of sizing and positional attributes to the XML, which are not easy to pre-empt[10].

[8] DataFrame used to be called SchemaRDD but was renamed. See e.g. http://apache-spark-developers-list.1001551.n3.nabble.com/renaming-SchemaRDD-gt-DataFrame-td10271i20.html.
[9] https://gephi.org/gexf/format/
[10] I tried, but Sigma did not seem especially happy working with longitude and latitude directly.

Putting it all together

The source code for this assignment was developed in Eclipse[11], using the Scala-IDE plugin[12], and compiled and built using Maven[13] – rather than sbt. (I had prior familiarity with Eclipse and Maven, albeit not in terms of building Scala projects). This resulted in a jar file containing the compiled classes, which was invoked with spark-submit using the following command on the cluster.

```
spark-submit \
--class com.downinja.msc.bda.JourneyDataDriver \
--master yarn \
--deploy-mode client \
journey-data-driver-0.0.1-SNAPSHOT.jar \
hdfs:/user/jdown003/ /home/jdown003/sparkstuff/ RegularJourneys2016.csv
```

The first five lines just tell Spark (1.4.1) how to find and run the JourneyDataDriver class, and also to run on YARN rather than standalone. The JourneyDataDriver class then takes three program arguments telling it the input directory, output directory, and journey data file name, respectively. In the above example, the RegularJourneys2016.csv file needs to be available on HDFS before running the process, as does a file called stations.csv containing the docking stations information[14].

```
[jdown003@dsm2 sparkstuff]$ hfs -ls
Found 21 items
-rw-r--r--   3 jdown003 hadoop 1593886339 2017-03-09 18:22
RegularJourneys2016.csv
-rw-r--r--   3 jdown003 hadoop  219542814 2017-03-08 13:26
RegularJourneys2016Extract.csv
....
-rw-r--r--   3 jdown003 hadoop      76633 2017-03-08 11:56 stations.csv
```

The code then performs the following logical steps:

- Reads program arguments (input/output directories/files).
- Sets up SparkConf, SparkContext and SQLContext.
- Reads stations.csv and creates a vertex for each docking station.
- Reads RegularJourneys2016.csv, and creates an edge for each journey.

[11] https://eclipse.org/, specifically version 4.6.1 - "Neon.1"
[12] http://scala-ide.org/
[13] https://maven.apache.org/
[14] I didn't make this configurable as there was no need; I only made the journeys file configurable so that a cut-down version could be switched-in for initial debugging.

- Creates a graph object containing these vertices and edges.
- Runs the pageRank algorithm on this graph.
- Creates a Spark SQL DataFrame where each row represents a docking station together with its rank and in/out degrees (the number of incoming/outgoing edges).
- Queries the DataFrame to get the list of docking stations in order of rank, high to low – and writes this out to a csv file.
- Queries the DataFrame to get a list of *pairs* of docking stations, where the first in the pair has a higher rank despite having a lower in-degree. Writes these out, along with the numeric differences in in-degree and rank between each pair, to a csv file.
- Queries the DataFrame to find the docking station which has gained the most rank, as compared to those other docking stations which have a lower-rank, but higher in-degree, than it.
- Queries the DataFrame to find the docking station which has lost the most rank, as compared to those other docking stations which have a higher-rank, but lower in-degree, than it.
- Runs personalPageRank for each of the above two docking stations, and outputs the resulting graph in GEXF format.
- Outputs the initial pageRanked graph in GEXF format.

Note that for the GEXF output, parallel edges were removed and the remaining distinct edges given a weight attribute representing the number of journeys made along them – reducing their number from 9+ million to some 175 thousand. For the personalPageRank graphs, I additionally filtered the number of edges so that output graph only included those starting or finishing at the vertex in question, with a view to ease of later visualisation.

The resulting GEXF files were subsequently copied back to my PC, and opened in Gephi for publishing. Since in this case the data already contained longitude and latitude values, it made sense to use the GeoLayout plugin[15] so that the resulting graphs would be physically accurate depictions of the network. Gephi was also used to size the docking station vertices according to their PageRank attributes, and edges according to their weights.

Finally, the sigmajs plugin was used to create the Javascript (and supporting[16]) files needed for interactive, browser based, visualisation – and these were copied across to IGOR[17].

---

[15] https://marketplace.gephi.org/plugin/geolayout/
[16] Although sigmajs can work with GEXF, it's preferred format is (naturally) JSON.
[17] Available at http://doc.gold.ac.uk/~jdown003/network/

Running

The process reliably took ~3 minutes to complete (M=2.77, SD=0.2), based on 5 runs using the default 2 executors. I also repeated the process with 4, 6 and 10 executors, by supplying a `num-executors` parameter to the spark-submit command. Timings were taken from the Spark console log output (I took the difference between the last and first timestamps). The boxplots below show a decrease in both duration and range when using more executors, tailing off around 10.

The screenshots below show snapshots of the event log and resource usage/distribution across two of the machines in the dsm cluster, as the application was running.

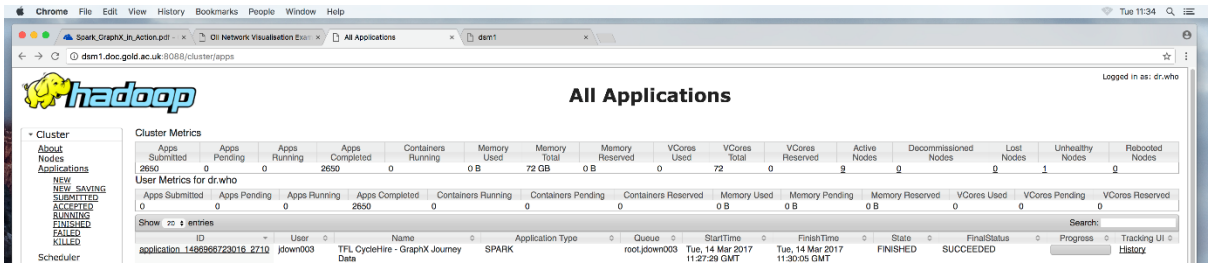

Figure 1: YARN web console showing Spark job running

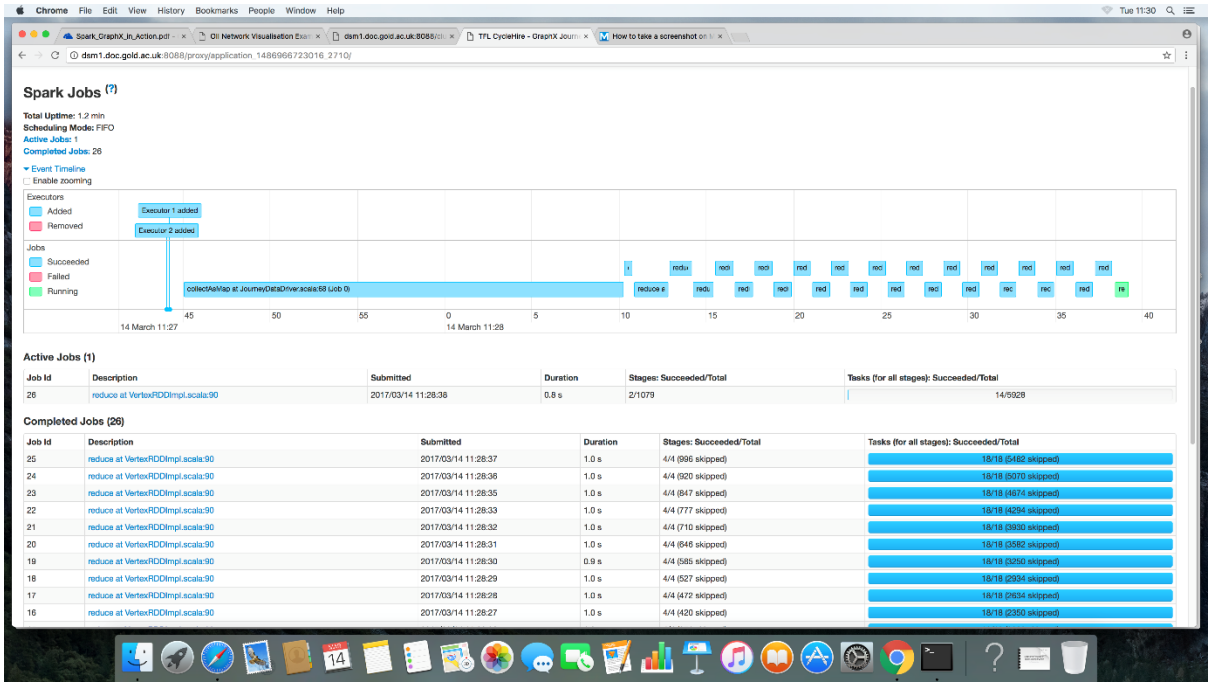

Figure 2: Spark Web Console showing Event Timeline for running job

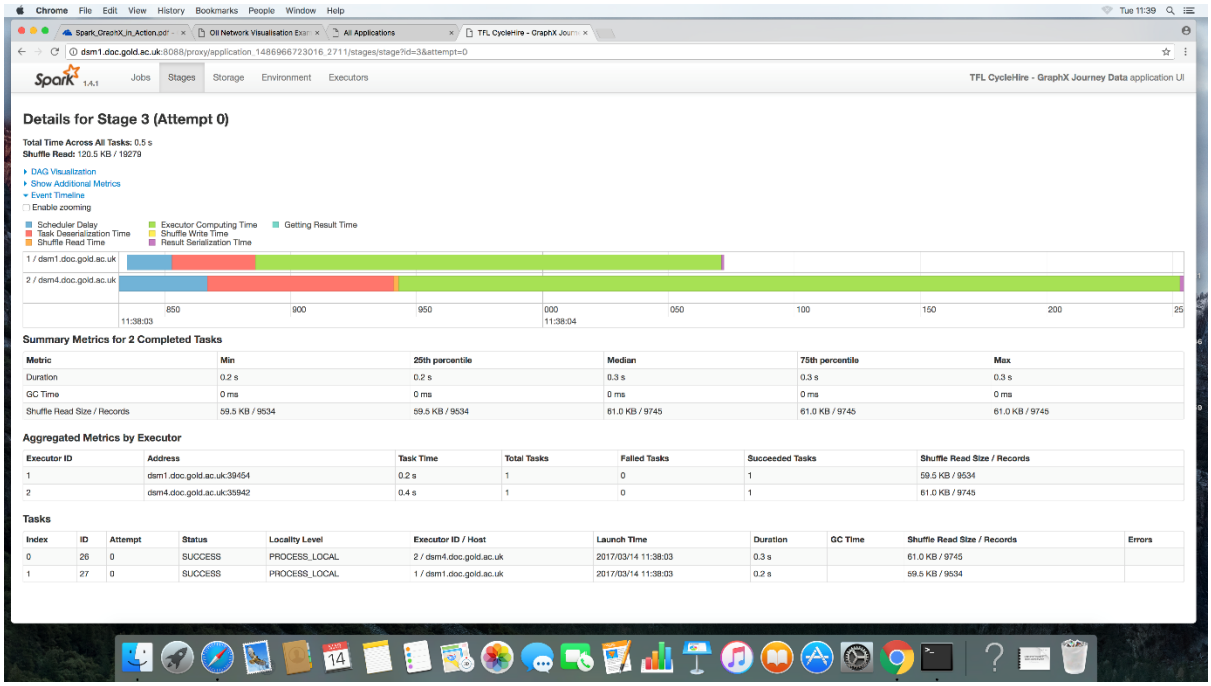

Figure 3: Spark Web Console showing resource metrics for running job across dsm1 and dsm4

Results

Belgrove Street, King's Cross, was the highest ranked docking station – which was not unexpected, given that earlier investigations had established that 180,066 journeys either started or ending there, and that of the 12708 individual bikes in operation only 201 did not check in there at some point during the year. Per the table below, of the top 20 stations by rank (leftmost two columns), and by in-degree (rightmost two columns), the top seven ranked docking stations were also the seven most popular destinations (since in-degree is a measure of inbound journey count). However, after this the pattern shifts; Black Lion Gate, Kensington Gardens, was the 8th highest ranked station, with a lower in-degree (45,156) than Brushfield Street, Liverpool Street (45,296). Generally speaking, though, docking stations were in similar positions whether based on rank or in-degree - for at least the first several hundred rows.

Top 20 docking stations by RANK		Top 20 docking stations by IN_DEGREE	
NAME	RANK	NAME	IN_DEGREE
Belgrove Street , King's Cross	6.409865	Belgrove Street , King's Cross	88743
Waterloo Station 3, Waterloo	5.391897	Waterloo Station 3, Waterloo	72922
Hop Exchange, The Borough	4.546574	Hop Exchange, The Borough	60036
Hyde Park Corner, Hyde Park	3.951513	Hyde Park Corner, Hyde Park	54566
Waterloo Station 1, Waterloo	3.888266	Waterloo Station 1, Waterloo	51754
Wormwood Street, Liverpool Street	3.608633	Wormwood Street, Liverpool Street	48065
Albert Gate, Hyde Park	3.452103	Albert Gate, Hyde Park	47250
Black Lion Gate, Kensington Gardens	3.315313	Brushfield Street, Liverpool Street	45296
Brushfield Street, Liverpool Street	3.306019	Newgate Street , St. Paul's	45180
Newgate Street , St. Paul's	3.276807	Black Lion Gate, Kensington Gardens	45156
Holborn Circus, Holborn	3.255184	Holborn Circus, Holborn	45153
Finsbury Circus, Liverpool Street	3.21077	Finsbury Circus, Liverpool Street	43912
Craven Street, Strand	3.074533	Craven Street, Strand	40050
Duke Street Hill, London Bridge	2.954753	Duke Street Hill, London Bridge	39111
Wellington Arch, Hyde Park	2.907024	Queen Street 1, Bank	38572
Queen Street 1, Bank	2.858692	Regent's Row, Haggerston	37614
William IV Street, Strand	2.751022	Wellington Arch, Hyde Park	37304
Bethnal Green Road, Shoreditch	2.71656	Bethnal Green Road, Shoreditch	37154
Regent's Row, Haggerston	2.708607	William IV Street, Strand	36601

The docking station which gained the most positions in rank, as compared to its position by in-degree, was Saunders Ness Road, Cubitt Town. This station was number 239 in the list by rank, as compared to 412 by in-degree. Relative to all docking stations which had a higher in-degree than it, Saunders Ness Road had a total rank difference over them of +25.68. Conversely, the docking station at Hoxton Station had a total rank difference of -4.71. This docking station was number 405 in the list by in-degree, but only 474 by rank. Table 1 below shows the heavily skewed distributions for overall rank, rank differences, relative rank increase and relative rank decrease.

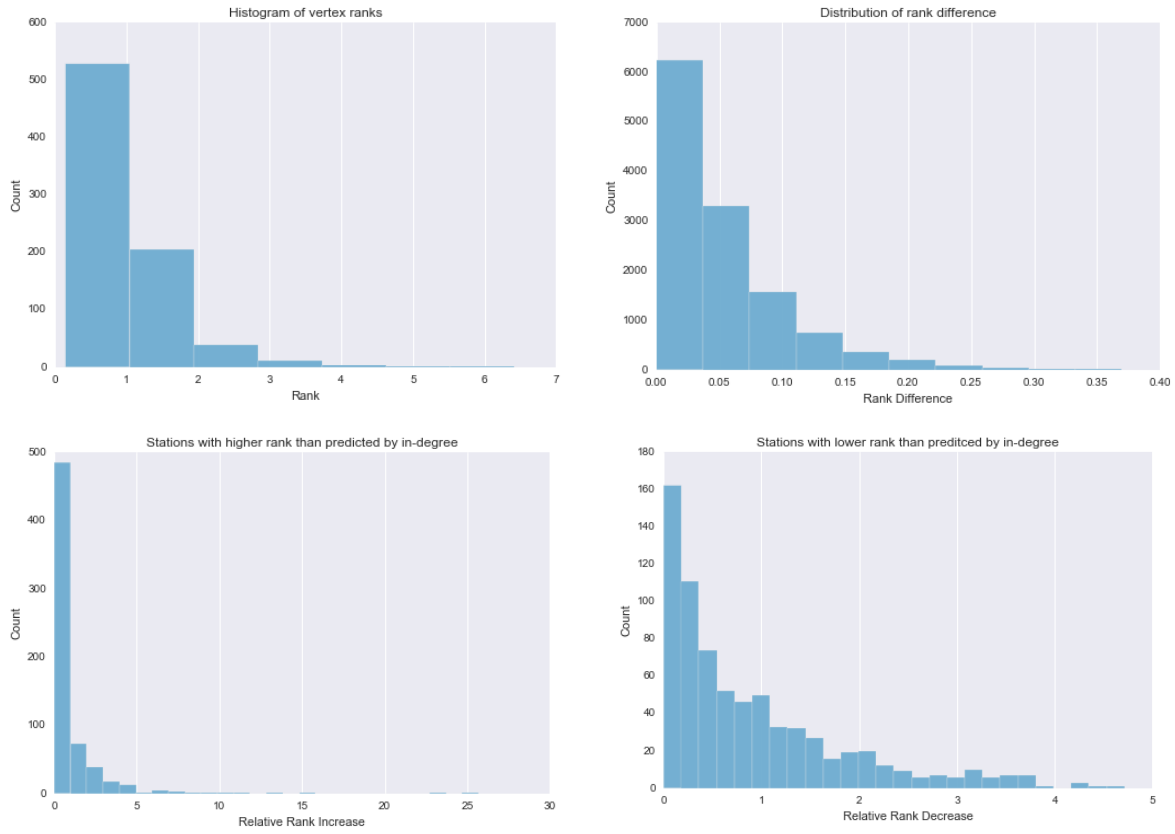

Table 1: Distribution of overall rank (top-left), and then for pairs of vertices A and B – where A has a higher rank than B despite a lower in-degree – distributions of rank differences between A and B, total increase in A's rank over all B, and total decrease of B's rank compared to all A.

These distributions suggest the following:

- Only a small number of docking stations have a high rank relative to the others – an indication that PageRank may have been successful in identifying influential nodes.
- The difference between a node's in-degree and rank is not typically large; for any given pair of nodes A and B, where A has a higher rank despite a lower in-degree, the gain in rank is never more than 0.37.

- A small number of nodes consistently "pick-up" rank from nodes which have a higher in-degree; there are only twenty with a total rank increase > 5, and only six with a total rank increase > 10.
- Nodes do not lose rank to other nodes (with lower in-degrees) by anything like the same amounts.

The Gephi visualisation of the overall network graph told a similar story to that which had been found previously[18]. The busiest docking locations tended to be mainline stations; King's Cross and Waterloo, followed by Liverpool Street and London Bridge – with some additional hotspots such as the Hop Exchange, Borough, Hyde Park, and various locations in the City. The routes with the most journey counts were also identified, including those around Kensington Gardens, Hyde Park, and the Olympic Park in Stratford. The visualisations for Saunders Ness Road and Hoxton Station are harder to interpret, in terms of trying to ascertain why the former should have a higher rank than would be predicted by its in-degree (and vice versa for the latter). Probably this is not the intent of Personalised PageRank, in which case the values given to vertices in the local area do not have any meaning in this context. Alternatively, it could be that this particular metric – i.e. whether a vertex should consistently under/over achieve compared to other vertices with lower/higher in-degrees – is not in itself useful[19].

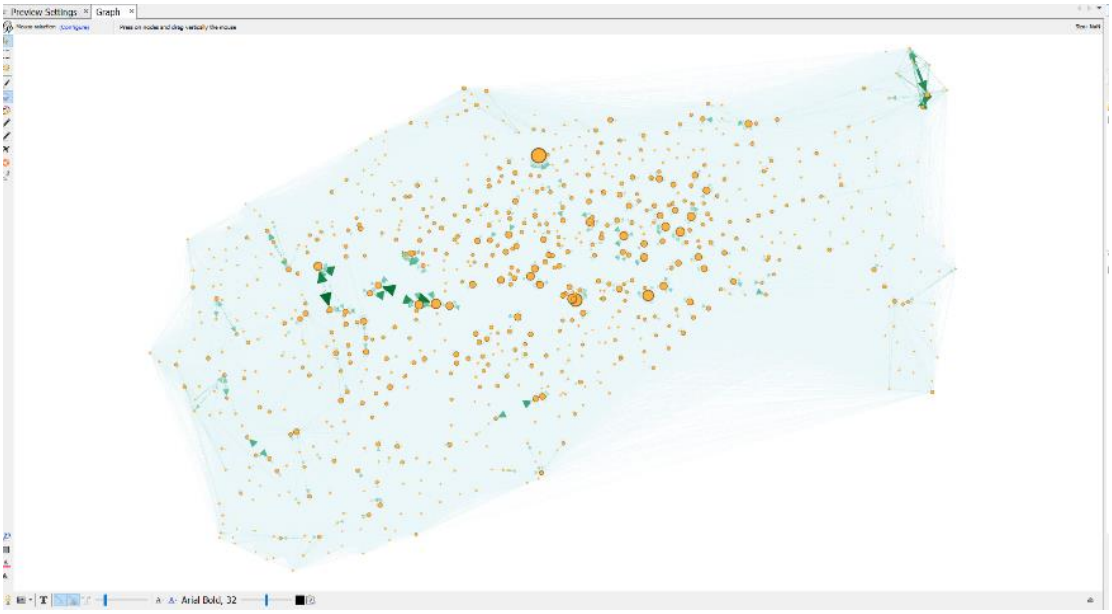

Figure 4: rankedJourneys.gexf loaded in Gephi workbench, with GeoLayout applied and vertex size / edge weight based on rank and journey count attributes respectively.

[18] albeit more extensively, and with a more accurate layout; I had previously only managed to represent the busiest docking stations & routes, rather than map the entire network.
[19] It was after all mainly chosen because the query behind it was reasonably complicated and I wanted to see if Spark SQL could handle it.

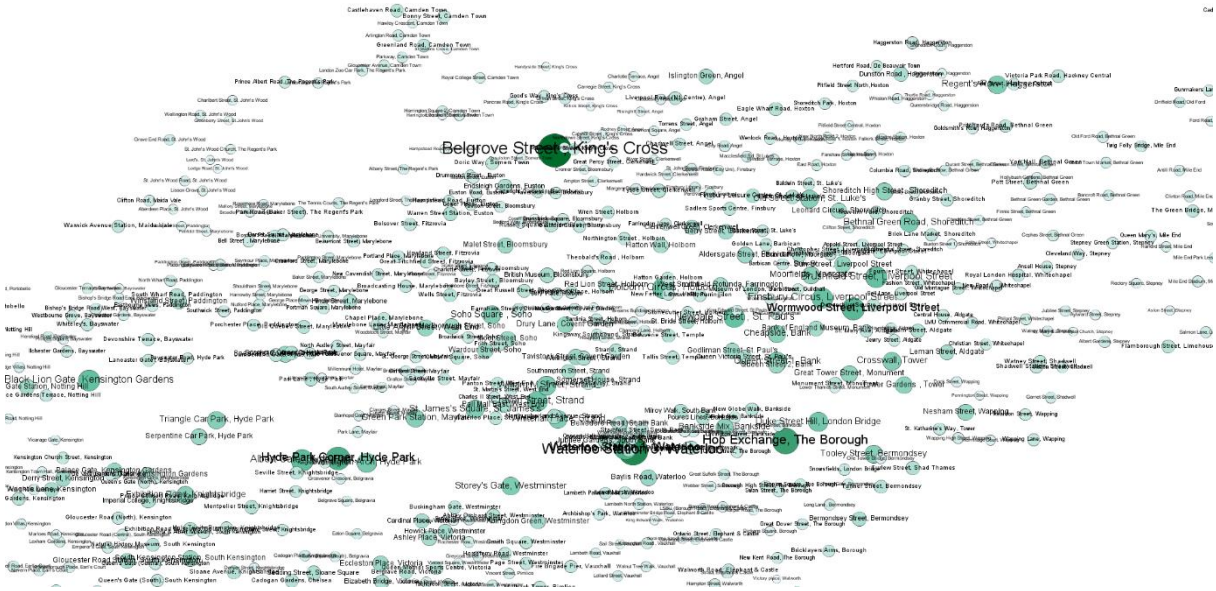

Figure 5: Preview of final output for rankedJourneys.gexf, with edges omitted for clarity.

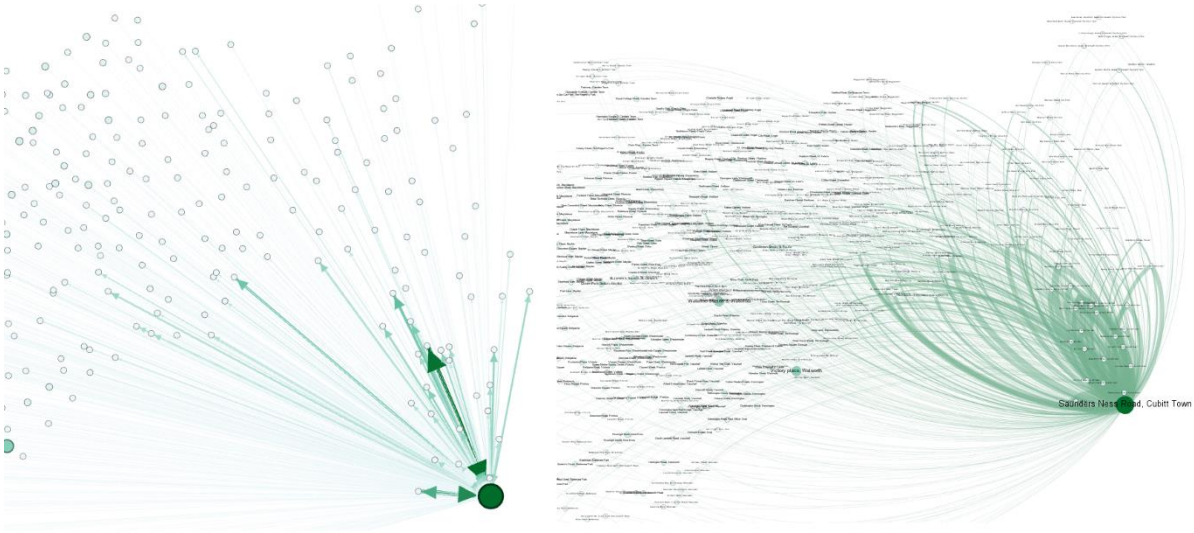

Figure 6: Gephi representation of the personalisedPageRank results for Saunders Ness Road, Cubitt Town. This was the docking station which gained the most in rank, relative to other docking stations which had a higher in-degree than it.

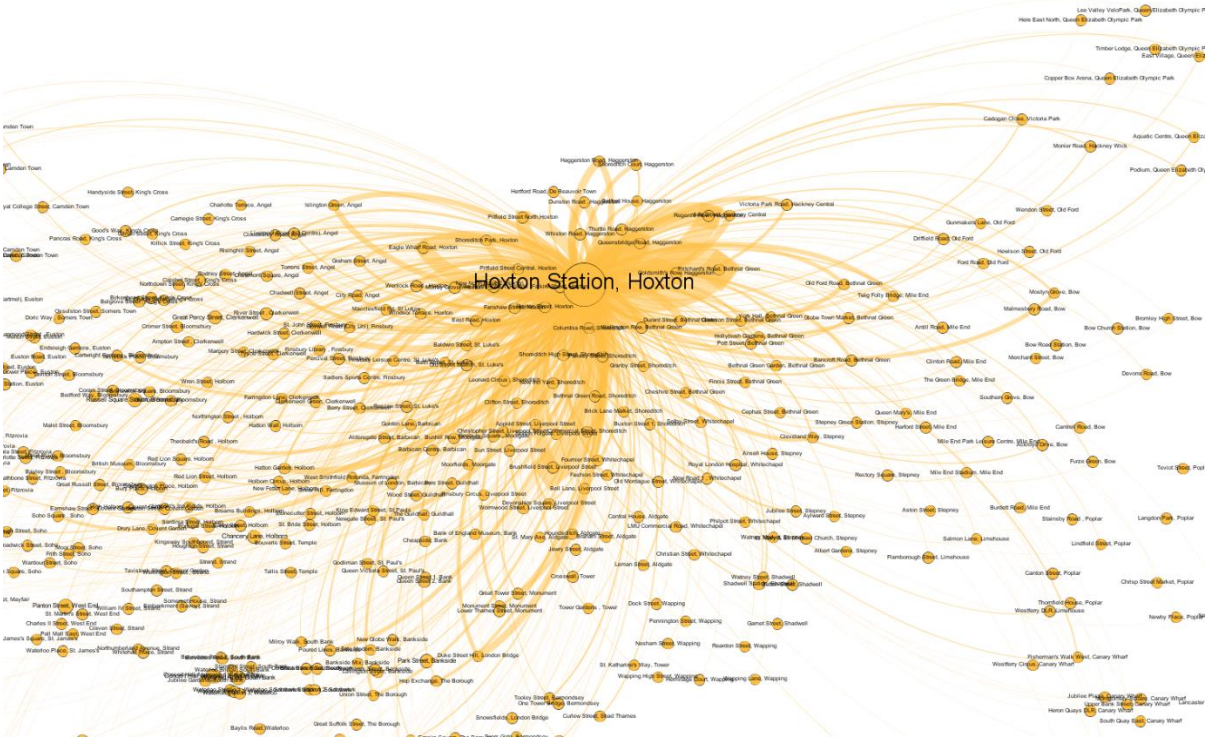

Figure 7: Figure 8: Gephi representation of the personalisedPageRank results for Hoxton Station. This was the docking station which lost the most in rank, relative to other docking stations which had a lower in-degree than it.

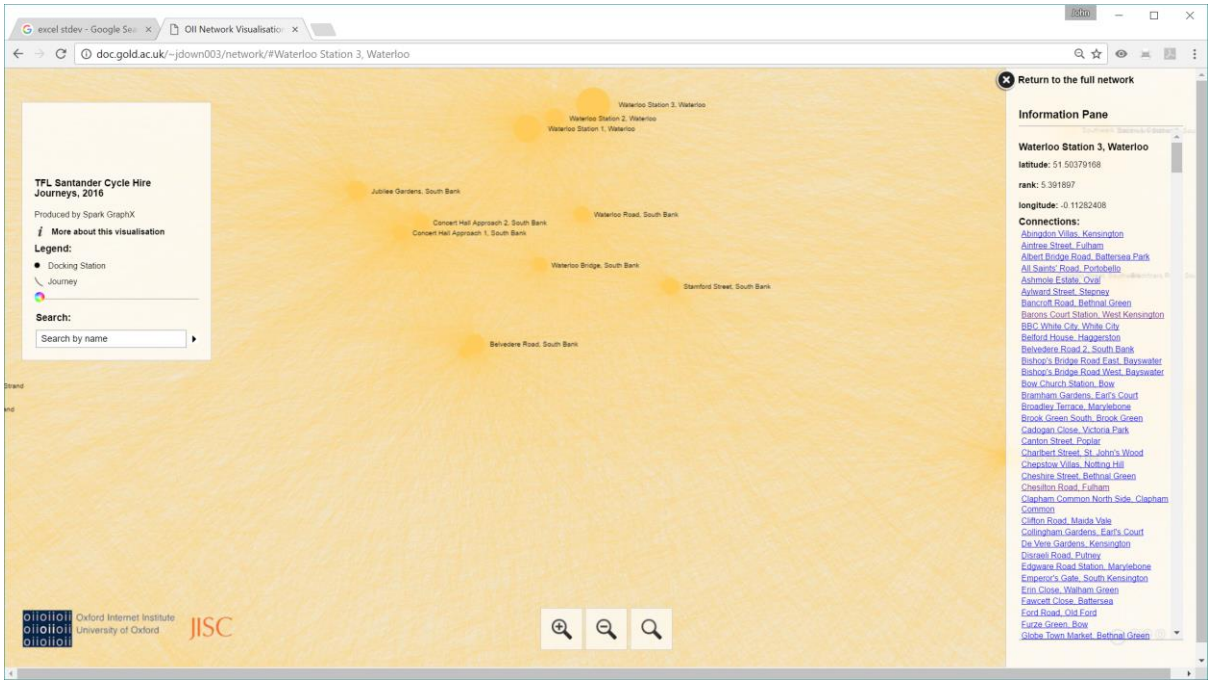

Figure 9: Screenshot from sigmajs interactive network graph - available at http://doc.gold.ac.uk/~jdown003/network

Discussion

The results of applying PageRank to the cycle data suggest that it does have utility. It identified a similar set of principal docking stations as did the earlier, tabular, approach, and moreover extended this process, holistically, to every station in the network – and with far less effort. In addition, the localised networks of influence provided by Personalised PageRank would have been much harder to generate via tabular aggregation.

It is less clear whether rank is any more meaningful than in-degree in this particular domain; most likely due to a lack of clarity on my part regards which questions to ask of the data. I was not able to establish why, for example, Hoxton Station should be ranked lower than would be suggested by in-degree alone. In one sense the answer is trivial; it is because journeys to Hoxton Station did not originate at locations with as high a rank as did journeys to some of its lower in-degree "rivals". But this does not really offer anything in terms of an explanation of a deeper or latent structure within the cycle network.

Possibly this is due to a problem of taking a "user-centric" perspective of the network. The conceptual model for PageRank is the random walk of a web surfer following links between web pages (and occasionally visiting all-new pages). The pages with the highest rank are those which the surfer is more likely to (eventually) visit. Clearly there is no direct analogue of this in the cycle hire network – since a user will only make one journey, rather than a sequence of steps, and their choice of destination is not constrained by a limited subset of hyperlinks (routes). However, from a "bike's-eye view", the analogy holds better. From the bike's perspective, every journey is a random walk from wherever it happens to be, but down previously worn paths, in terms of likelihoods of route, formed by earlier user choices. So in this sense, it more closely resembles the same kind of Markov Chain as the random web surfer model[20]. Perhaps PageRank is mapping out the likelihoods of where in the network a bike will (eventually) end up?

This being the case, then perhaps PageRank is more suited to network/capacity planning, as a means of determining, in advance (e.g. at several steps removed), locations at which there are likely to be too few, or too many, bikes. TFL do in fact provide a live feed of docking station availability statistics, and whilst this is probably not feature-rich enough to be able to tie-in with journey data[21], it would be interesting to see if capacity changes at one docking station could be used to predict capacity changes at others – if not to predict actual journeys.

---

[20] Source: https://en.wikipedia.org/wiki/PageRank

[21] Which in any event is only published retrospectively, after some months

References

Chan, J., & Teknomo, K. (2016). Hub Identification of the Metro Manila Road Network Using PageRank. *arXiv preprint arXiv:1609.01464*.

Crisostomi, E., Kirkland, S., & Shorten, R. (2011). A Google-like model of road network dynamics and its application to regulation and control. *International Journal of Control, 84*, 3, 633-651.

Karau, H., Konwinski, A., Wendell, P. & Zaharia, M. (2015). *Learning Spark: Lightning Fast Data Analysis*. Sebastopol, CA: O'Reilly.

Malak, M. S., & East, R. (2016). *Spark GraphX In Action*. New York: Manning Publications.

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph Databases*. O'Reilly Media.

Appendices

Code

JourneyDataDriver.scala

```
package com.downinja.msc.bda

import org.apache.log4j.LogManager

import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.serializer._

import org.apache.spark.sql.Row;
import org.apache.spark.sql.functions._
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.types.{DoubleType,IntegerType,LongType,StructType,StructField,StringType};

object JourneyDataDriver {

  @transient lazy val log = org.apache.log4j.LogManager.getLogger("JourneyDataDriver")

  def main(args : Array[String]) {

    val inDirectory = args(0)
    val outDirectory = args(1)
    val journeysFile = args(2)
    val pageRankTolerance = 0.0001

    log.info("Using inDirectory: " + inDirectory)
    log.info("Using outDirectory: " + outDirectory)
    log.info("Using journeysFile: " + journeysFile)
    log.info("Using pageRankTolerance: " + pageRankTolerance)

    // SparkConf needs a master URL to be specified - which is different when running on my
    // home PC (in standalone mode) than when running on the DSM cluster. To avoid having to
    // recompile/redeploy code per environment, I'm using either the "-Dspark.master=local"
    // system property, for local use, or else specifying "--master yarn" to the spark-submit
    // job, on the cluster.
    val conf = new SparkConf()
      .setAppName("TFL CycleHire - GraphX Journey Data")
```

```
      .set("spark.serializer", classOf[KryoSerializer].getName)
    log.info("created SparkConf: " + conf.toDebugString)

    // Create our SparkContext. (We're not doing this inside a REPL shell, so it doesn't get
    // done for us.)
    val sc = new SparkContext(conf)
    log.info("created SparkContext: " + sc)

    // Also create a SQLContext for querying DataFrames later.
    val sqlContext = new SQLContext(sc)
    log.info("created SQLContext")

    // Read in the static data for the docking stations (IDs, names,
    // longitudes and latitudes) - converting them to graphx Vertices
    val vertices = sc.textFile(inDirectory + "stations.csv")
      .filter(line => !line.startsWith("id,")) // skip header
      .map(JourneyDataUtil.createVertex)

    // Read in the journey data, converting each journey to a graphx Edge
    val edges = sc.textFile(inDirectory + journeysFile)
      .filter(line => !line.startsWith(",RentalId")) // skip header
      .map(JourneyDataUtil.createEdge)
    edges.cache

    // Create/cache our primary Graph object
    val journeysGraph = Graph(vertices, edges)
    journeysGraph.cache

    // Create lookup tables of the in/out degrees for each vertex,
    // as this will be more convenient later when constructing our
    // SQL DataFrame.
    val byInDegree = journeysGraph.inDegrees.collectAsMap
    val byOutDegree = journeysGraph.outDegrees.collectAsMap

    // Now call the pageRank method to rank each vertex (docking station)
    // in the graph (cycle network). This will continue to loop round the
    // algorithm unless/until the specified tolerance is met.
    val verticesWithRank = journeysGraph
      .pageRank(pageRankTolerance).vertices
      .join(journeysGraph.vertices)
    verticesWithRank.cache

    // Although we could query the graph, using a functional approach,
    // it's also useful to construct a DataFrame so that we can execute
```

```
    // SQL queries for basic ordering and aggregation (rather than
    // anything more suited to, or requiring, graph traversal).
    val schema = StructType(Array(
      StructField("ID", LongType, true),
      StructField("NAME", StringType, true),
      StructField("RANK", DoubleType, true),
      StructField("IN_DEGREE", IntegerType, true),
      StructField("OUT_DEGREE", IntegerType, true)
    ))

    // Helper function for converting verticesWithRank to a DataFrame, in
    // conjuction with the lookup tables for in/out degree by VertexId
    def createRow(vertexWithRank: (VertexId,(Double, Map[String, String]))): Row = {
      val vertexId = vertexWithRank._1.longValue
      val name = vertexWithRank._2._2.get("name").get
      val rank = vertexWithRank._2._1
      val inDegree = byInDegree.get(vertexId).getOrElse(0)
      val outDegree = byOutDegree.get(vertexId).getOrElse(0)
      return Row(vertexId, name, rank, inDegree, outDegree);
    }

    // Now we can create a DataFrame from our graph/ranked data.
    val rowRDD = verticesWithRank.map(createRow)
    val verticesDataFrame = sqlContext.createDataFrame(rowRDD, schema)
    verticesDataFrame.registerTempTable("VERTICES")

    // A simple query is to sort the vertices by rank, in descending
    // order. Then we can write this to a csv file for use outside
    // this application.
    val columns = "ID,NAME,RANK,IN_DEGREE,OUT_DEGREE"
    var sql = "SELECT " + columns + " FROM VERTICES ORDER BY RANK DESC"
    var dataFrame = sqlContext.sql(sql)
    JourneyDataUtil.dataFrameToCsv(outDirectory + "rankedJourneys.csv", dataFrame)

    // Similarly, we can use SQL to join the DataFrame to itself and calculate
    // differences in RANK and IN_DEGREE - where these are of interest (e.g.
    // where one vertex has a higher IN_DEGREE than another, but has been
    // assigned a lower RANK by the pageRank algorithm).
    sql =
      "SELECT " +
          "a.ID, " +
          "b.ID, " +
          "(b.IN_DEGREE - a.IN_DEGREE) as DEGREE_DIFF, " +
          "(a.RANK - b.RANK) AS RANK_DIFF " +
```

```
      "FROM " +
          "VERTICES a, " +
          "VERTICES b " +
      "WHERE " +
          "a.RANK > b.RANK " +
          "and b.IN_DEGREE > a.IN_DEGREE "

    dataFrame = sqlContext.sql(sql)
    JourneyDataUtil.dataFrameToCsv(outDirectory + "rankDiffs.csv", dataFrame)

    // And in order to summarise these interesting events, we can sum over the
    // IN_DEGREE and RANK values and pick out e.g. the vertex with the highest
    // total increase in rank over those vertices which have a larger value of
    // IN_DEGREE.
    sql =
      "SELECT " +
          "a.ID, " +
          "SUM(b.IN_DEGREE - a.IN_DEGREE) as TOTAL_DEGREE_DIFF, " +
          "SUM(a.RANK - b.RANK) AS TOTAL_RANK_DIFF " +
      "FROM " +
          "VERTICES a, " +
          "VERTICES b " +
      "WHERE " +
          "a.RANK > b.RANK " +
          "AND b.IN_DEGREE > a.IN_DEGREE " +
          "GROUP BY a.ID"

    dataFrame = sqlContext.sql(sql)
    dataFrame.cache
    JourneyDataUtil.dataFrameToCsv(outDirectory + "highestRankedDespiteLowerInDegree.csv", dataFrame)
    // Store the first one in the results so we can use it for personalisedPageRank later
    val highestRankedDespiteLowerInDegree = dataFrame.orderBy(desc("TOTAL_RANK_DIFF")).take(1)(0).getAs[Long](0)
    log.info("highestRankedDespiteLowerInDegree = " + highestRankedDespiteLowerInDegree)

    // Similarly, find the vertex which "loses out" the most, in terms of RANK,
    // to those vertices which have a lower value of IN_DEGREE.
    sql =
      "SELECT " +
          "b.ID, " +
          "SUM(b.IN_DEGREE - a.IN_DEGREE) as TOTAL_DEGREE_DIFF, " +
          "SUM(a.RANK - b.RANK) AS TOTAL_RANK_DIFF " +
      "FROM " +
          "VERTICES a, " +
          "VERTICES b " +
```

```
      "WHERE " +
          "a.RANK > b.RANK " +
          "AND b.IN_DEGREE > a.IN_DEGREE " +
          "GROUP BY b.ID"

    dataFrame = sqlContext.sql(sql)
    dataFrame.cache
    JourneyDataUtil.dataFrameToCsv(outDirectory + "lowestRankedDespiteHigherInDegree.csv", dataFrame)
    // Store the first one in the results so we can use it for personalisedPageRank later
    val lowestRankedDespiteHigherInDegree = dataFrame.orderBy(desc("TOTAL_RANK_DIFF")).take(1)(0).getAs[Long](0)
    log.info("lowestRankedDespiteHigherInDegree = " + lowestRankedDespiteHigherInDegree)

    // Now we move on to creating Gephi format files for visualisation. We don't want to specify
    // every journey from A to B, as plotting ~9 million lines on a chart isn't helpful. But we
    // can summarise by totting up all journeys from A to B, and using this total as the weight
    // of the respective Edge.

    val journeyCounts = edges
      // There's probably a function to do this, but am just using map/reduce to sum
      // each journey between A and B and then create a lookup table. Potentially it's
      // riskier to collect Edges in the driver app, but out of a possible ~332k
      // combinations of A and B it's unlikely that we're going to pull back enough to
      // run out of memory.
      .map(JourneyDataUtil.countJourney)
      .reduceByKey(_+_)
      .collectAsMap

    val edgesWithCounts = edges
      // Similarly there may be an "add attribute" function that I'm not aware of,
      // but I'm just using the above lookup table
      .map(edge => JourneyDataUtil.createEdgeWithJourneyCount(edge, journeyCounts))
      .distinct
    edgesWithCounts.cache

    // Output the full network graph in GEFX format. Sort of. It still needs to be loaded
    // into Gephi and exported for use by e.g. Javascript libraries - since Gephi adds
    // additional attributes for display purposes. Haven't quite figured out how to
    // calculate them directly.
    JourneyDataUtil.graphToFile(outDirectory + "rankedJourneys.gexf", Graph(verticesWithRank, edgesWithCounts))

    // Also output a cut-down network graph of just those journeys involving our "highest ranked
    // despite not having the highest in-degree" vertex, from earlier. In case it's useful.
    val ppr1 = journeysGraph.personalizedPageRank(highestRankedDespiteLowerInDegree, pageRankTolerance)
      .vertices
```

```
      .join(journeysGraph.vertices)

    val pprEdges1 = edgesWithCounts.filter(
        edge =>
           edge.srcId == highestRankedDespiteLowerInDegree ||
           edge.dstId == highestRankedDespiteLowerInDegree)

    JourneyDataUtil.graphToFile(outDirectory + "highestRankedDespiteLowerInDegree.gexf", Graph(ppr1, pprEdges1))

    // And the same for our "not highest ranked, despite having a higher in-degree than
    // more highly ranked vertices" vertex.
    val ppr2 = journeysGraph.personalizedPageRank(lowestRankedDespiteHigherInDegree, pageRankTolerance)
      .vertices
      .join(journeysGraph.vertices)

    val pprEdges2 = edgesWithCounts.filter(
        edge =>
           edge.srcId == lowestRankedDespiteHigherInDegree ||
           edge.dstId == lowestRankedDespiteHigherInDegree)

    JourneyDataUtil.graphToFile(outDirectory + "lowestRankedDespiteHigherInDegree.gexf", Graph(ppr2, pprEdges2))

  }

}
```

JourneyDataUtil.scala

```
package com.downinja.msc.bda

import org.apache.spark.graphx.Edge
import org.apache.spark.graphx.Graph
import org.apache.spark.sql.DataFrame

/*
 * Util functions for use with the JourneyDataDriver class. Note that these
 * functions are passed around with lambdas, so should be stateless (to avoid
 * passing the enclosing class around with them).
 */

object JourneyDataUtil {
```

```scala
    // This is for use inside a map/reduce implementation of journey count.
    // It just emits "1" for every combination of start and end station found
    // in the graph.
    def countJourney(journey: Edge[Int]): ((Long, Long), Int) = {
        val startStationId = journey.srcId
        val endStationId = journey.dstId
        return ((startStationId, endStationId),1)
    }

    // This creates an Edge for each line of the (csv) journeys data.
    def createEdge(line: String) : Edge[Int] = {
      val array = line.split(',')
      val sourceId = array(6).toLong
      val destId = array(4).toDouble.toLong
      // could add attributes here such as bikeId, timestamp etc
      return Edge(sourceId, destId, 1); // "1" is just a dummy attribute, since GraphX requires there to be one
    }

    // This decorates an Edge with the number of times it's been traversed in the graph - that is,
    // its journey count. Possibly inefficient to pass in a lookup table (Map) of journey counts, as
    // this will need to be replicated out to each cluster machine that uses this function (?)
    def createEdgeWithJourneyCount(edge: Edge[Int], map: scala.collection.Map[(Long,Long), Int]): Edge[Int] = {
        return Edge(edge.srcId, edge.dstId, map.get((edge.srcId, edge.dstId)).get)
    }

    // This creates a vertex for every location station in the (csv) docking stations data.
    def createVertex(line: String): (Long, Map[String, String]) = {
      // http://stackoverflow.com/questions/1757065/java-splitting-a-comma-separated-string-but-ignoring-commas-in-quotes
      val array = line.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)")
      val id = array(0).toLong
      val name = array(1)
      val lat = array(3)
      val long = array(4)
      val attributes = Map(
          "name" -> name,
          "latitude" -> lat,
          "longitude" -> long
      )
      return (id, attributes)
    }
```

```
    // This just writes out a DataFrame in csv format. Evidently this is built-in
    // from Spark 1.6, but we're running 1.4.1 on the cluster.
    def dataFrameToCsv(fileName: String, dataFrame: DataFrame) = {
      val csvFile = new java.io.File(fileName)
      csvFile.createNewFile();
      val pw = new java.io.PrintWriter(csvFile)
      pw.write(dataFrame.columns.map(columnName => columnName + ",").mkString  + "\n")
      val results = dataFrame.map(row => row.mkString("",",","")+",\n").collect.mkString;
      pw.write(results)
      pw.close
    }

    // This writes out a Graph in GEXF format (using the helper function from the
    // Manning "Spark GraphX in Action" book below).
    def graphToFile(fileName: String, graph:Graph[(Double, Map[String, String]),Int]) = {
      val gephiFile = new java.io.File(fileName)
      gephiFile.createNewFile();
      val pw = new java.io.PrintWriter(gephiFile)
      pw.write(JourneyDataUtil.toGexf(graph))
      pw.close
    }

  /*
   * Adapted from:
   * https://manning-content.s3.amazonaws.com/download/3/6311f4a-a8af-45e2-80d1-f4689c23d802/SparkGraphXInActionSourceCode.zip
   */
    def toGexf(g:Graph[(Double, Map[String, String]),Int]) =
      "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
      "<gexf xmlns=\"http://www.gexf.net/1.2draft\" version=\"1.2\">\n" +
      "  <graph mode=\"static\" defaultedgetype=\"directed\">\n" +
      "    <attributes class=\"node\">\n" +
      "      <attribute id=\"0\" title=\"latitude\" type=\"double\"/>\n" +
      "      <attribute id=\"1\" title=\"longitude\" type=\"double\"/>\n" +
      "      <attribute id=\"2\" title=\"rank\" type=\"float\"/>\n" +
      "    </attributes>\n" +
      "    <nodes>\n" +
           g.vertices.sortBy(vertex => vertex._2._1, ascending = false).map(v =>
      "      <node id=\"" + v._1 + "\" label=" +
           v._2._2.get("name").get.replace("&", "&amp;") + ">\n" +
      "        <attvalues>\n" +
      "          <attvalue for=\"0\" value=\"" + v._2._2.get("latitude").get + "\"/>\n" +
      "          <attvalue for=\"1\" value=\"" + v._2._2.get("longitude").get + "\"/>\n" +
      "          <attvalue for=\"2\" value=\"" + v._2._1 + "\"/>\n" +
```

```
      "        </attvalues>\n" +
      "      </node>\n").collect.mkString +
      "    </nodes>\n" +
      "    <edges>\n" +
         g.edges.sortBy(edge => edge.attr, ascending = false).map(e =>
      "      <edge source=\"" + e.srcId + "\" " +
           "target=\"" + e.dstId + "\" " +
           "label=\"" + e.attr + "\" " +
           "weight=\"" + e.attr.toFloat + "\"" +
         "/>\n").collect.mkString +
      "    </edges>\n" +
      "  </graph>\n" +
      "</gexf>"
  }
```

Raw TFL data

Data Dictionary: Journey Data

Rental Id	Unique ID for an observed journey. Ascending sequence. Primary Key	StartStationId	ID of the docking station at which a journey ends. Foreign key.
Duration	The journey duration. Measured in seconds, but always in whole minutes e.g. 60, 120, 180.	StartStation Name	Text description of the address of the docking station at which the journey began.
Bike Id	Unique ID for a particular bike in the cycle-hire scheme.		
End Date	The date/time at which the journey ended. Format dd/MM/yyyy hh:mm,SSS.		
EndStationId	ID of the docking station at which a journey began. Foreign key.		
EndStation Name	Text description of the docking station at which the journey ended.		
Start Date	The date/time at which the journey started. Format dd/MM/yyyy hh:mm,SSS		

Sample Journey Data

```
Rental Id,Duration,Bike Id,End Date,EndStation Id,EndStation Name,Start Date,StartStation Id,StartStation Name
50608184,4440,1104,01/01/2016 01:14,21,"Hampstead Road (Cartmel), Euston",01/01/2016 00:00,98,"Hampstead Road, Euston"
50608186,1200,529,01/01/2016 00:24,118,"Rochester Row, Westminster",01/01/2016 00:04,419,"Chelsea Bridge, Pimlico"
50608187,1200,8452,01/01/2016 00:24,118,"Rochester Row, Westminster",01/01/2016 00:04,419,"Chelsea Bridge, Pimlico"
50608188,1080,8934,01/01/2016 00:22,251,"Brushfield Street, Liverpool Street",01/01/2016 00:04,66,"Holborn Circus, Holborn"
50608189,1080,13194,01/01/2016 00:23,251,"Brushfield Street, Liverpool Street",01/01/2016 00:05,66,"Holborn Circus, Holborn”
```

Data Dictionary: Docking Stations

id	Unique ID, Primary Key
name	Description of address
terminalName	Secondary unique ID
lat	Latitude
long	Longitude
installDate	
removeDate	
installed	Data about current state of docking station, not relevant to this analysis.
locked	
temporary	
nbBikes	
nbEmptyDocks	
nbDocks	

Sample Data

```
id,name,terminalName,lat,long,installed,locked,installDate,removalDate,temporary,nbBikes,nbEmptyDocks,nbDocks
195,"Milroy Walk, South Bank",959,51.5072443,-0.106237501,TRUE,FALSE,1.28E+12,,FALSE,16,14,30
194,"Hop Exchange, The Borough",960,51.50462759,-0.091773776,TRUE,FALSE,1.28E+12,,FALSE,25,24,49
196,"Union Street, The Borough",961,51.50368837,-0.098497684,TRUE,FALSE,1.28E+12,,FALSE,8,9,17
193,"Bankside Mix, Bankside",963,51.50581776,-0.100186337,TRUE,FALSE,1.28E+12,,FALSE,16,44,60
119,"Bath Street, St. Luke's",964,51.52589324,-0.090847761,TRUE,FALSE,1.28E+12,,FALSE,0,18,18
148,"Tachbrook Street, Victoria",965,51.49211134,-0.138364847,TRUE,FALSE,,,FALSE,5,11,16
156,"New Kent Road, The Borough",966,51.49443626,-0.092921165,TRUE,FALSE,1.28E+12,,FALSE,24,9,33
```