# Computational Network Analysis - Class Project
# Wikipedia Navigation Paths

Tobias Kloht, ID: 4596192

March 7, 2014

## 1 Introduction

Wikipedia defines itself as "a collaboratively edited, multilingual, free Internet encyclopedia" [1] covering more than 4.4 million articles in the english language version. It is ranked 6th among the most popular websites worldwide [2] and has developed into the de-facto standard for free user generated encyclopedic content.

This class project aims to analyze the behaviour of users browsing Wikipedia by aggregating internal links used when searching for a specific information into a network. Apart from finding common patterns and analyzing the user behaviour in general, this could potentialy reveal problems with the site's navigation and highlight possible improvements.

## 2 Dataset

The dataset used in this project contains navigation paths on Wikipedia, collected through the game Wikispeedia [3]. The idea of this game is to navigate from one given article to another, using only internal links and as few steps as possible. It was developed as part of a research project to acquire data on commonsense knowledge [4]. The game operates on a condensed subset of Wikipedia consisting of 4600 articles from 2007.

The resulting dataset consists of 51,318 finished navigation paths and can be accessed at the Stanford Network Analysis Project [5]. Unfinished navigation paths are also available but have not been used in this project.

## 3 Adjusting the Data Format

While the dataset described above clearly represents a network with articles as vertices and links as edges, the provided data had to be adjusted for usage with the igraph library [6]. To illustrate this, the following listing is an excerpt of the data, provided as a tab-separated list.

The columns from left to right are:

- Hashed IP Address
- Timestamp
- Duration in seconds
- Path given as list of articles separated by ";"
- Rating - optinally provided by users after finishing the game

In addition to that, back clicks are represented as "<".

Excerpt: Wipispeedia Dataset

```
1 36dabfa133b20e3c  1249525912  112 14th_century;China;Gunpowder;Fire 2
2 20418ff4797f96be  1229188046  139 14th_century;Time;Isaac_Newton;Light;Color;
      Rainbow  1
3 08888b1b428dd90e  1232241510  74  14th_century;Time;Light;Rainbow 3
```

The main problem is that this does not define a network but a list of paths clicked through by users. In order to allow processing of the data with R and igraph I have decided to transform the data into a network as follows:

**Vertices:** The articles

**Edges:** The links used to navigate between two articles

**Edge Weights:** The amount of times a link has been used

In this format the representation of back clicks is problematic, because it results in a vertex labeled "<" with ingoing edges for each back click. For this project it makes more sense to replace each "<" with the article that the back click leads to. For example, an entry of the form

```
1 408362e90dc90bce  1249103594  61  Asia;Japan;Cereal;<;<;Japan;Fishing;<;Meat;
      Mammal NULL
```

would be transformed to

```
1 408362e90dc90bce  1249103594  61  Asia;Japan;Cereal;Japan;Asia;Japan;Fishing;
      Japan;Meat;Mammal  NULL
```

## 3.1 The Large Graph Layout format

The resulting network will be stored in the Large Graph Layout (LGL) format [7], which is essentially a modified adjacency list. It lists each vertex preceded by a pound sign, followed by all vertices sharing an edge with it on subsequent lines. Each neighboring vertex can optionally have a weight attached to it.

LGL format example

```
1 # vertex1name
2 vertex2name [optionalWeight]
3 vertex3name [optionalWeight]
```

### 3.2 Conversion Steps

The provided data was transformed to the format described above in the following steps:

1. Normalize navigation path by replacing "back clicks" with the actual article
2. Extract all vertices, i.e. articles
3. For each vertex, find neighboring vertices and store both in the LGL format

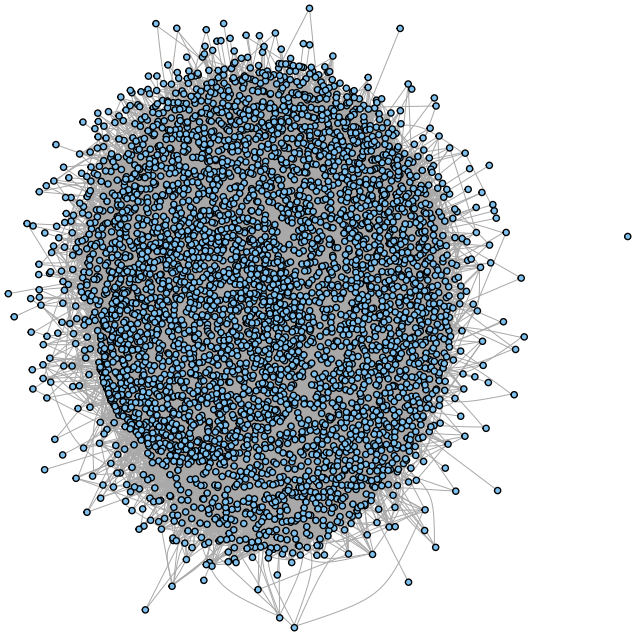
In step three the navigation paths are iterated for each article. Whenever an entry matching the current article is found, the next entry of the navigation list is added to a temporary list. After this is done for all navigation paths, the temporary list holds all neighboring articles of the current vertex and it is easy to construct the format discussed above. The other steps are straightforward and will not be discussed further.

Each step has been implemented as a Ruby script. All scripts are available in Appendix A as well as Github.

## 4 Analyzing the Network

The resulting network can be imported by igraph directly. A simple plot of the network provides the following:

```
plot.igraph(wikispeedia, vertex.size = 2, vertex.label = NA, edge.arrow.width = 0.02,
    edge.arrow.size = 0.2, edge.width = 0.2, layout = layout.kamada.kawai)
```



To get an initial overview the size of the network can be examined:

```
# number of vertices
vcount(wikispeedia)

## [1] 4170

# number of edges
ecount(wikispeedia)

## [1] 59530
```

## 4.1 Degree Distribution

The next step will be a look at the degrees and their distribution:

```r
# mean degree of the network
mean(degree(wikispeedia))
```

```
## [1] 28.55
```

```r
# median degree of the network
median(degree(wikispeedia))
```

```
## [1] 14
```

```r
# max degree of the network
max(degree(wikispeedia))
```

```
## [1] 1639
```

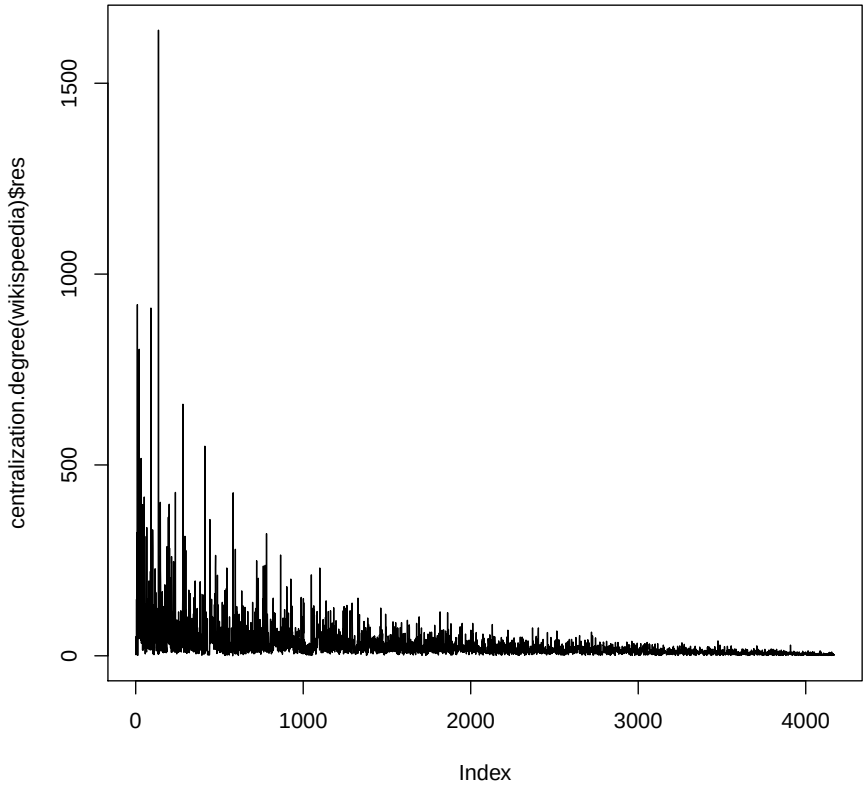
This shows that the networks degree is right-skewed, i.e. the mean degree is significantly larger than the median. This is a first hint that a small number of vertices has significantly more edges than the rest of the network. To investigate this further we will calculate the degree centralization.

```r
centralization.degree(wikispeedia)$centralization
```

```
## [1] 0.3863
```

This result is relatively high and supports our assumption above. The degree centrality is distributed as follows:

```r
plot(centralization.degree(wikispeedia)$res, type = "l")
```




```
# mean degree
mean(centralization.degree(wikispeedia)$res)

## [1] 28.55

# max degree
max(centralization.degree(wikispeedia)$res)

## [1] 1639
```

This data further exemplifies that the degree is very unevenly distributed among the network, which means that a small number of articles has been navigated frequently in the game. One article in particular has a very high degree even in comparison to all other vertices with high degrees. To find out which article is represented by this vertex the following command can be used:

```
which(degree(wikispeedia) == max(degree(wikispeedia)))

## United_States
##           136
```

The reason why the article about the United States of America has been used so frequently in this experiment is beyond the scope of this class project. However, given that the initial intention of the Wikispeedia project is to analyze commonsense knowledge, it could be assumed that the users of the game shared a basic understanding of that subject and used it to derive further information.

## 4.2 Betweenness Distribution

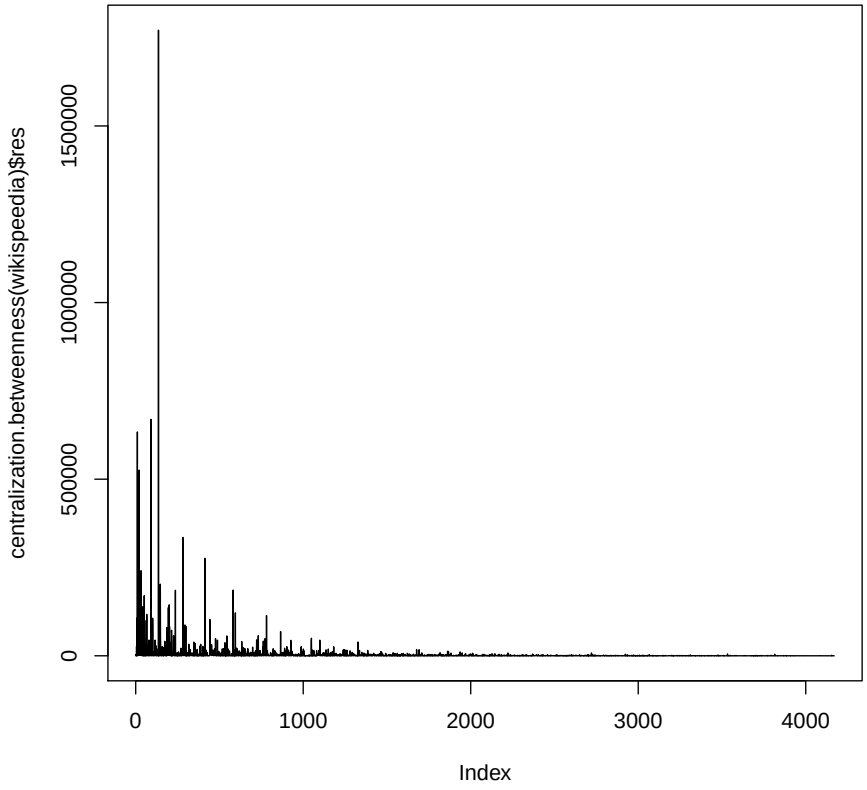
When analyzing the betweenness similar results to the degree section can be expected.

```
centralization.betweenness(wikispeedia)$centralization

## [1] 0.2035
```

This result is in line with our expectations and shows a relatively centralized distribution of the betweenness over the network.

```
plot(centralization.betweenness(wikispeedia)$res, type = "l")
```




```
# mean betweenness
mean(centralization.betweenness(wikispeedia)$res)

## [1] 3632

# max betweenness
max(centralization.betweenness(wikispeedia)$res)

## [1] 1771100
```

The Plot above further shows the similarity between degree and betweenness distribution in this graph. In the next step we will test if the same article shares maximum betweenness and degree:

```
which(betweenness(wikispeedia) == max(betweenness(wikispeedia)))

## United_States
```

```
##           136
```

This expectation is true as well. In this case this just shows that the article which is used the most also interconnects the most articles, which is only natural given the premise of this experiment.

### 4.3 Path Length

The analysis to this point makes it very likely that the small-world phenomenon applies to this network. To confirm this we can calculate the average path length of the network:

```
average.path.length(wikispeedia)

## [1] 2.743
```

This shows that the articles in the network can be reached with very few hops on average.

## 5 Conclusion

To summarize, we found that the examined network shows typical characteristics of similar networks. The relatively high centralization leads to a short path length because a small set of nodes can interconnect many other vertices.

For further analysis it would be interesting to identify communities within the network, this proved difficult during this class project due to the size of the dataset.

# References


[1] Wikipedia. *Wikipedia — Wikipedia, The Free Encyclopedia*. Mar. 2014. URL: \url{http://en.wikipedia.org/w/index.php?title=Wikipedia&oldid=598345833}.

[2] Alexa Internet, Inc. *Wikipedia.org Site Info*. Mar. 2014. URL: http://www.alexa.com/siteinfo/wikipedia.org.

[3] Alexa Bob West. *Wikispeedia*. Mar. 2014. URL: http://www.cs.mcgill.ca/~rwest/wikispeedia/.

[4] Robert West, Joelle Pineau, and Doina Precup. "Wikispeedia: An Online Game for Inferring Semantic Distances Between Concepts". In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. IJCAI'09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1598–1603. URL: http://dl.acm.org/citation.cfm?id=1661445.1661702.

[5] Stanford Network Analysis Project. *Web data: Wikispeedia navigation paths*. Mar. 2014. URL: http://snap.stanford.edu/data/wikispeedia.html.

[6] The igraph Project. *The igraph library*. Mar. 2014. URL: http://igraph.sourceforge.net/.

[7] Marcotte Lab, Institute for Cellular and Molecular Biology, University of Texas at Austin. *Large Graph Layout*. Mar. 2014. URL: http://lgl.sourceforge.net/#FileFormat.

# A Scripts for Data Transformation

Listing 1: Node Extraction

```ruby
#!/usr/bin/env ruby

# extracts all nodes/articles from paths_finished.tsv

require "csv"
#parsed_file = CSV.read("paths_finished.tsv", { :col_sep => "\t" })
nodes = Array.new

CSV.foreach("paths_finished.tsv", { :col_sep => "\t" }) do |row|
  # use row here...
  row[3].split(";").each{ |x|
    nodes.push(x)
    #puts x
  }
end
nodes.uniq!
nodes = nodes.sort_by{|name| name.downcase}

puts nodes
puts "length: "
puts nodes.length

CSV.open("nodes.csv", "wb") do |csv|
  nodes.each_with_index { |x, i|
    csv << [x, i]}
  # ...
end
```

Listing 2: Path normalization

```ruby
#!/usr/bin/env ruby

# normalizes back clicks ("<") by replacing them with the actual article.
# input paths taken from paths_finished.tsv,
# output will be paths_finished_fixed.tsv


require "csv"

nodes = Array.new

CSV.foreach("nodes.csv", { :col_sep => "," }) do |row|
  nodes.push(row[0])
end


File.open('paths_finished_fixed.tsv', 'a') do |file|
  CSV.foreach("paths_finished.tsv", { :col_sep => "\t" }) do |row|
  # use row here...
  edges = row[3].split(";")
  edges.each_with_index{ |x, i|
    if x == '<' && i < edges.length
      pivot = i-1 #save edge before first '<' as pivot element
      count = 1
      z = i
      # once a '<' edge has been found, find length of consecutive ones
      while edges[z+1] == '<'
        count += 1
        z += 1
      end
      # then replace each of these
      while count > 0
```

```
        # if you are x elements right of the pivot element, replace with
            element x steps to the left it
        edges[pivot + count] = edges[pivot - count]
        count -= 1
      end
    end
  }
  edgeString = ""
  edges.each_with_index { |e, i|
    edgeString << e
    if i < edges.length - 1
      edgeString << ";"
    end
  }
  row[3] = edgeString
  #puts "#{row[0]}\t#{row[1]}\t#{row[2]}\t#{row[3]}"
  file.puts "#{row[0]}\t#{row[1]}\t#{row[2]}\t#{row[3]}"
  end

end
```

Listing 3: Building the Network

```
#!/usr/bin/env ruby

# For each vertex, find neighboring vertices and store both in the LGL format

require "csv"

nodes = Array.new

CSV.foreach("nodes.csv", { :col_sep => "," }) do |row|
  nodes.push(row[0])

end


File.open('network_fixed', 'a') do |file|
  # for each vertex/article
  nodes.each { |x|
    nodeEdges = Array.new
    # iterate over all paths
    CSV.foreach("paths_finished_fixed.tsv", { :col_sep => "\t" }) do |row|
      # navigation paths are stored in column 4
      edges = row[3].split(";")
      edges.each_with_index{ |y, i|
        # iterate over all articles visited by user
        if x == y && i+1 < edges.length
         # if article matching current node is found, next list entry is a
             neighboring vertex -> add to temporary list
          nodeEdges.push(edges[i+1])
        end
      }
    end
    # lgl file is constructed here
    file.puts "# #{x}"
    nodeEdges.uniq.each { |e|
      # find edge weight
      z = nodeEdges.find_all { |x|  x == e }.length
      file.puts "#{e}\t#{z}"
    }


  }
end
```