

TP Final - Simulación

February 18, 2019

1 Datos

Alumno: Matías José Cano

Padrón: 97925

Paper: Sergey Brin and Lawrence Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Computer Science Department, Stanford University, Stanford, CA 94305, USA, 1998

Dataset: <https://snap.stanford.edu/data/web-Stanford.html>

2 Resumen

2.1 Introducción

Con el rápido aumento tanto de la información presente en internet, como de la cantidad de usuarios que consumían la web, las herramientas de búsqueda usadas hasta ese momento fueron perdiendo la calidad necesaria.

Los usuarios solían usar un índice de páginas mantenido (como *Yahoo!*) y a partir de este navegaban por el grafo de links. El problema que tenían estos índices es que si bien permitían cubrir temas populares de forma efectiva, tenían algunos defectos; eran subjetivos, difíciles de crear y mantener, requerían mucho tiempo para ser mejorados y no cubrían temas poco frecuentes. Asimismo, los motores de búsqueda basados en *keyword matching* generalmente devolvían muchos resultados de baja calidad, además de que podían ser “engañosos”.

Bajo este contexto, el objetivo del paper es proponer un sistema de búsqueda que resuelva los problemas de calidad y escalabilidad.

2.1.1 Calidad de la búsqueda

El problema de la calidad de las búsquedas se debía en gran medida a que el número de documentos en los índices fue aumentando en varios ordenes de magnitud, mientras que la capacidad de las personas de buscar en documentos no se incrementó. Es decir, las personas seguían buscando en los primeros resultados de la búsqueda.

Teniendo esto en cuenta, el paper propone establecer un orden de relevancia que refleje el interés de las personas. Para ello, se propone hacer uso del grafo de links (mediante la métrica de *PageRank*) y del texto de los links (esto permite obtener información valiosa de los links y además poder indexar páginas que no tienen texto).

Este trabajo se centrará principalmente en *PageRank*.

2.1.2 Escalabilidad

En cuanto a la escalabilidad el paper busca que el sistema de búsqueda pueda ser usado por una gran cantidad de personas, y que además la información de las búsquedas pueda ser usada con propósitos de investigación. Con este fin, se propone que la información de las búsquedas se guarde de forma comprimida.

2.2 PageRank

Como se mencionó anteriormente, lo que se busca reflejar es la importancia que las personas le atribuyen a cada página, para así poder priorizar los resultados de las búsquedas. Para ello el paper hace uso de PageRank, y lo define así:

"We assume page A has pages T₁...T_n which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. [...] Also C(A) is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

$$PR(A) = (1 - d) + d * \left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_n)}{C(T_n)} \right)$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one."

Esta definición se puede interpretar como el comportamiento de un "random surfer" que navega el grafo de links de forma aleatoria, clickeando un link con probabilidad d, o volviendo a empezar en una página aleatoria con probabilidad 1-d. De esta forma, el *PageRank* representa la probabilidad de que el "random surfer" visite la página.

2.3 Google más allá de PageRank

Si bien uno de los focos del paper estudiado es el algoritmo de *PageRank*, también se centra en la arquitectura empleada con el fin de que el motor de búsqueda alcance la escalabilidad y eficiencia deseada. A continuación se muestra un diagrama de la arquitectura propuesta:

URLserver: Se encarga de enviar listas de URLs a bajar a los *crawlers*.

Crawlers: Se encargan de bajar las páginas. Estas páginas luego son enviadas al *storeserver*.

Storeserver: Se encarga de comprimir y guardar las páginas en un repositorio (*repository*).

Repository: Se encarga de almacenar las páginas.

Indexer: Se encarga de leer del repositorio, descomprimir y parsear los documentos. Luego los documentos son convertidos en un conjunto de ocurrencias de palabras (llamados *hits*), los cuales son distribuidos en un conjunto de *barrels*. Además parsea todos los links que se encuentran en las páginas y guarda la información importante en los *anchors files*.

Barrels: Se encargan de almacenar los *hits*, los cuales además contienen datos sobre la posición de las palabras en los documentos, el tamaño de fuente (relativo) y el uso de mayúsculas.

Anchor files: Se encargan de almacenar a donde apunta cada link y el texto del link.

URLresolver: Se encarga de leer los *anchor files*, convertir las URLs de relativas a absolutas, y luego en *docIDs* (identificador único de cada página). Con esta información genera una base de datos de links y coloca el texto de los links en el *forward index* (índice con las palabras que aparecen en cada

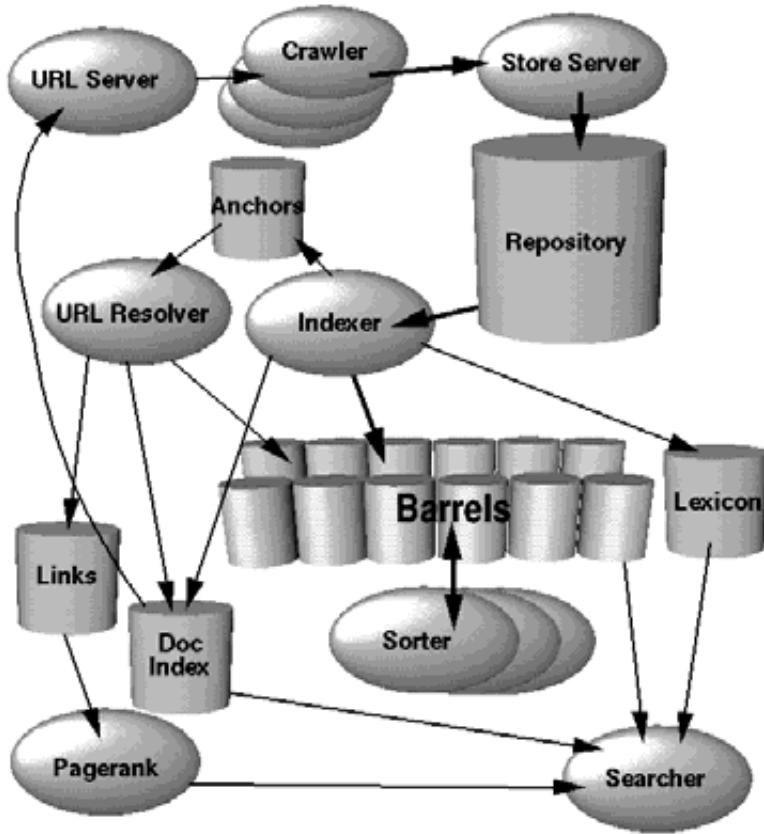


Diagrama de arquitectura de alto nivel de Google

página).

Sorter: Se encarga de reordenar los *barrels* con el fin de formar el *inverse index* (índice que referencia a las páginas mediante las palabras que aparecen en ellas, a partir de estos índices se pueden realizar las búsquedas).

3 Simulación de PageRank

A continuación se realizará el cálculo de *PageRank* para las páginas presentes en el dataset seleccionado. Algunas consideraciones a tener en cuenta:

- Se empleó un factor $\beta = 0.85$ como lo sugiere el paper.
- Se consideró que una vez que el *random surfer* llega a una página con grado de salida igual a 0, se puede continuar el camino en cualquier nodo del grafo.

Importo librerías a utilizar:

In:

```
import gzip
import random
from math import sqrt,log10
import numpy as np
```

```

import pandas as pd
import matplotlib.pyplot as plt
import datashader as ds
import datashader.transfer_functions as tf
from datashader.bundling import connect_edges
from datashader.layout import random_layout
from sklearn import linear_model
from sklearn.preprocessing import PolynomialFeatures

```

3.1 Parseo de datos

El dataset utilizado está formado por:

- Lineas comenzadas por # que representan comentarios
- Lineas con el formato <id_nodo_1>, <id_nodo_2> que representa que hay un link de la página 1 a la página 2.

Tomando esto en consideración se recorrió el archivo y se guardó el grafo.

In:

```

file = gzip.open(filename="web-Stanford.txt.gz", mode="rt")

graph = {}
edges = []

for line in file:
    if line[0] == '#': continue
    src, dst = map(lambda x : int(x.strip()), line.split('\t'))
    edges.append([src, dst])
    if (not src in graph): graph[src] = set()
    if (not dst in graph): graph[dst] = set()
    graph[src].add(dst)

edges = pd.DataFrame(edges, columns = ["source", "target"])

```

3.2 Datos básicos del grafo

In:

```

print("Cantidad de nodos: {}".format(len(graph)))
print("Cantidad de aristas: {}".format(len(edges)))

```

Out:

```

Cantidad de nodos: 281903
Cantidad de aristas: 2312497

```

Algorithm 36: PageRank v1.0

```
1  $PR[1..N] = 1/N$  while not convergencia do
2   for  $i=1..N$  do
3     for  $l$  in  $links(i)$  do
4       //transfer PR to linked pages
5        $PR[l] = PR[l] + \beta PR[i]/|l|$ 
6   // Add teleportation to ALL pages
7    $PR = PR + (1 - \beta)(1/N)$ 
```

Fuente: Apunte de Organización de Datos.

3.3 Cálculo de PageRank de forma iterativa

Para realizar el cálculo de PageRank se empleó el siguiente método iterativo:

Este algoritmo es equivalente a realizar iterativamente la multiplicación de matrices, con la salvedad de que en este caso no se realiza la multiplicación de matrices entera (lo cual computacionalmente es muy costoso).

In:

```
beta = 0.85
pagerank = {}
difference = 1
convergence = []
N = len(graph)

for node in graph:
    pagerank[node] = 1/N

i = 0
while difference>10**-10:
    i += 1
    pagerank_copy = {}
    for node in graph: pagerank_copy[node] = 0
    dead_ends = 0
    for node, links in graph.items():
        if not links:
            dead_ends += pagerank[node]
            continue
        for link in links:
            pagerank_copy[link] += beta*pagerank[node]/len(links)

    for node in graph:
        pagerank_copy[node] += ((1-beta)+beta * dead_ends)*(1/N)

    difference =
        sqrt(sum((pagerank[i]-pagerank_copy[i])**2 for node in graph))
```

```

if (i%10 == 0):
    print("Iteration {0}: {1}".format(i, difference))
convergence.append(difference)

pagerank = pagerank_copy

```

Out:

```

Iteration 10: 3.213699385255622e-06
Iteration 20: 2.3091720015225337e-08
Iteration 30: 1.533487237270756e-08

```

A continuación realizamos un gráfico de la diferencia entre iteraciones de los valores obtenidos de PageRank (junto con una regresión cuadrática que ayuda a observar la tendencia). La diferencia entre la iteración i y la $i - 1$ se obtuvo mediante:

$$diff = \sqrt{\sum_{n=0}^N (PR_i(n) - PR_{i-1}(n))^2}$$

Este valor se utilizó para garantizar la convergencia, exigiendo que se alcance un $diff < 10^{-9}$.

In:

```

plt.figure(figsize=(14, 14))

poly = PolynomialFeatures(degree=2)
regr = linear_model.LinearRegression()

x = np.fromiter(range(1,i+1), dtype=np.int)
y = np.fromiter(map(lambda x: log10(x), convergence), dtype=np.float)

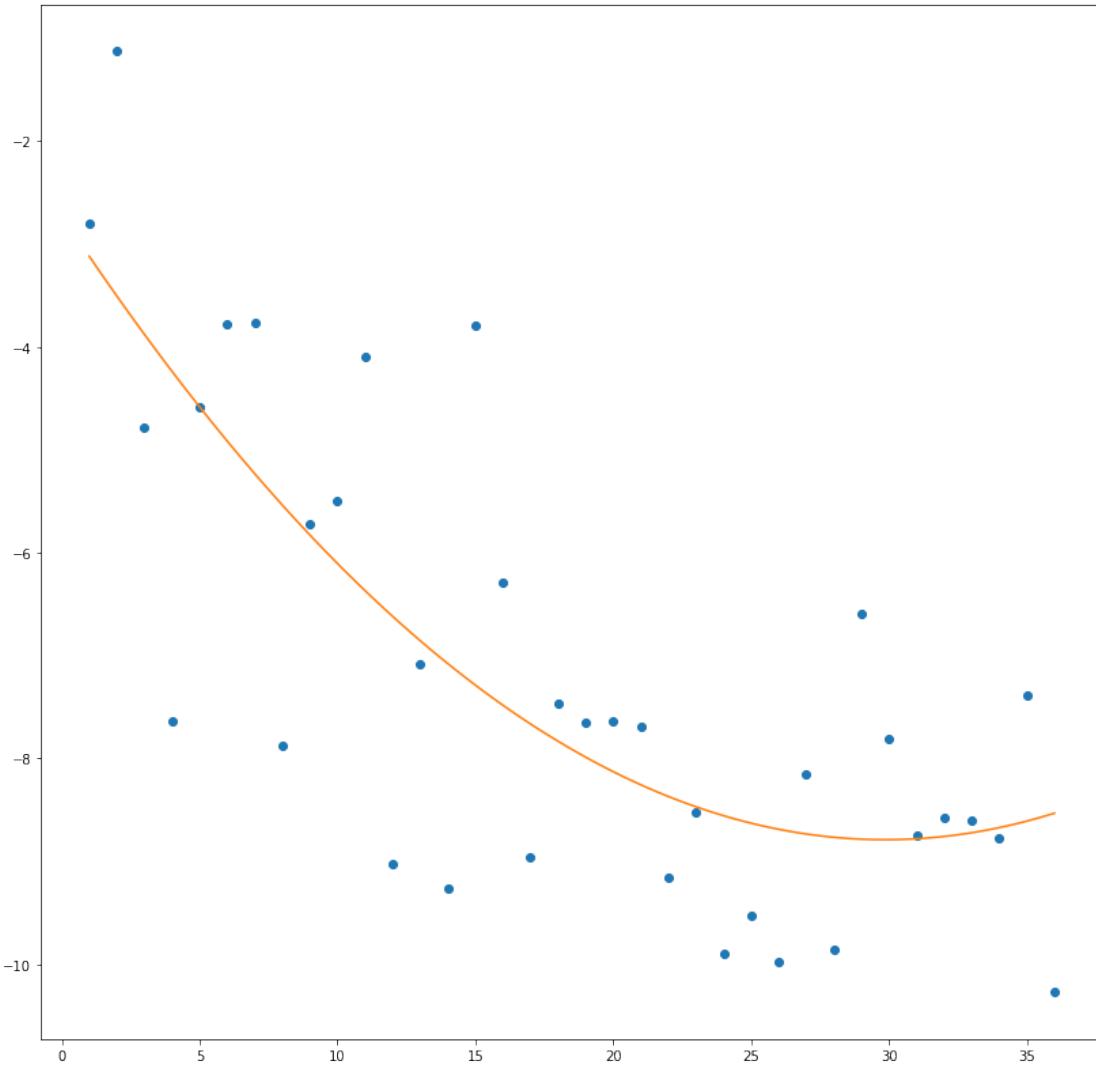
regr.fit(poly.fit_transform(x.reshape(-1,1)), y.reshape(-1,1))
x_interpolation = np.linspace(1, i, num=100, endpoint=True)
x_interpolation = x_interpolation.reshape(-1,1)

plt.plot(x, y, 'o')
plt.plot(x_interpolation,
         regr.predict(poly.fit_transform(x_interpolation))),
         '-')

plt.show()

```

Out:



3.4 Cálculo de PageRank mediante simulación de random surfer

En este caso, se realizó la simulación de un *random surfer* que parte de un nodo aleatorio y realiza caminos de largo 300. El *random surfer*, en cada paso con una probabilidad de β (0.85) elige ir a un nodo adyacente al actual, y con una probabilidad de $1-\beta$ (0.15) elige un nodo aleatorio del grafo para continuar. Se realizaron 500 iteraciones de este proceso y luego se obtuvo la probabilidad de pasar por cada nodo (*PageRank*) como:

$$PR(i) = \frac{q_i}{q_{tot}}$$

Donde q_i es la cantidad de veces que se paso por el nodo i , y q_{tot} es la cantidad de veces que se visitó algún nodo.

In:

```
beta = 0.85
```

```

iterations = 500
length = 300

total = 0
appearances = {key: 0 for key in graph.keys()}

for i in range(iterations):
    if i%100 == 0: print("Iteración {}".format(i))
    current = random.choice(tuple(graph.keys()))
    for i in range(length):
        appearances[current] += 1
        total += 1
        options = graph[current]
        if graph[current] and
           random.uniform(a=0,b=1)<0.85 else tuple(graph.keys())
        current = random.choice(tuple(options))

practical_pagerank = {node: app/total for node, app in appearances.items()}

```

Out:

```

Iteración 0
Iteración 100
Iteración 200
Iteración 300
Iteración 400

```

3.5 Comparación entre los métodos

En primer lugar, se realizó una tabla que cuenta con los valores obtenidos para los dos valores de *PageRank*.

In:

```

nodes = pd.DataFrame(
    [[str(key), pagerank[key], practical_pagerank[key]]
     for key in graph.keys()], columns=['node', 'PR_1', 'PR_2'])

nodes = nodes.sort_values(by='PR_1', ascending=False)
nodes.head()

```

Out:

	node	PR_1	PR_2
281	89073	0.011304	0.011440
67	226411	0.009269	0.009033
1352	241454	0.008299	0.008780
4898	262860	0.003024	0.002880
3367	134832	0.003001	0.003327

Luego, a partir de esta tabla se analizó la correlación de los resultados obtenidos de las dos formas. Como método de correlación se uso la correlación de *Pearson* (default en la librería de python usada). Como se puede ver la correlación se aproxima a 1, por lo que se podría decir que

los dos métodos son indistinguibles en cuanto a los resultados obtenidos. Además, esto permite mostrar, que la interpretación de *PageRank* como un *random surfer* es correcta.

In:

```
nodes['PR_1'].corr(nodes['PR_2'])
```

Out:

```
0.98886872547280991
```

4 Análisis de resultados

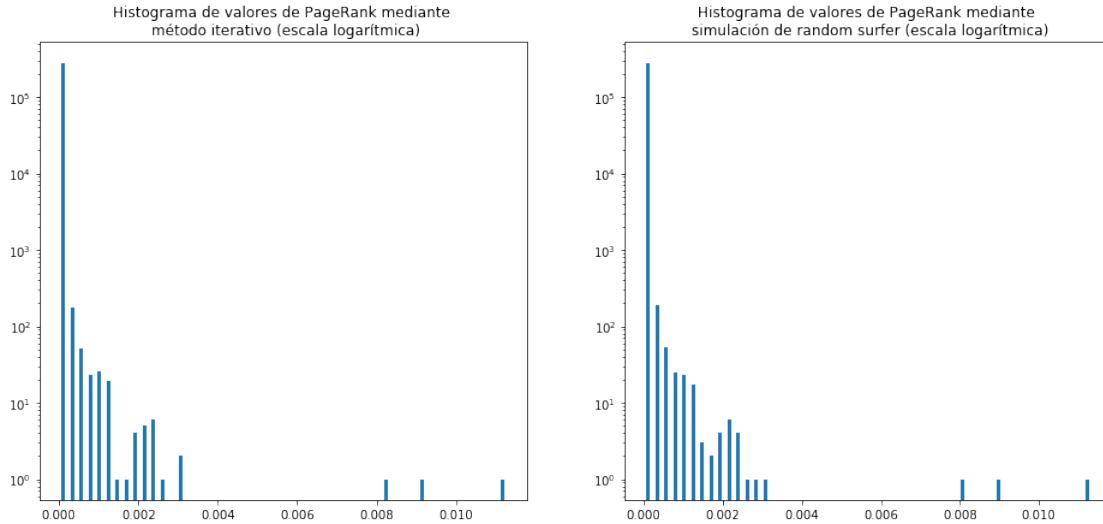
4.1 Histograma

A continuación se realizaron histogramas de los valores obtenidos de *PageRank*. Como se puede observar, hay pocas páginas con valores de *PageRank* altos, y muchas páginas con valores de *PageRank* bajos. Esto quiere decir, que en el grafo del dataset estudiado hay unas pocas páginas con un grado de centralidad y relevancia muy alto. Adicionalmente, se puede observar que los dos histogramas (correspondiente a cada uno de los métodos) son muy similares, lo cual tiene concordancia con la correlación mostrada previamente.

In:

```
plt.figure(figsize=(16, 7))
plt.subplot(1,2,1)
plt.hist(list(pagerank.values()), bins=50, log=True, rwidth=0.4)
plt.title(
    "Histograma de valores de PageRank mediante \n"
    "método iterativo (escala logarítmica)")
plt.subplot(1,2,2)
plt.hist(list(practical_pagerank.values()), bins=50, log=True, rwidth=0.4)
plt.title(
    "Histograma de valores de PageRank mediante \n"
    "simulación de random surfer (escala logarítmica)")
plt.show()
```

Out:

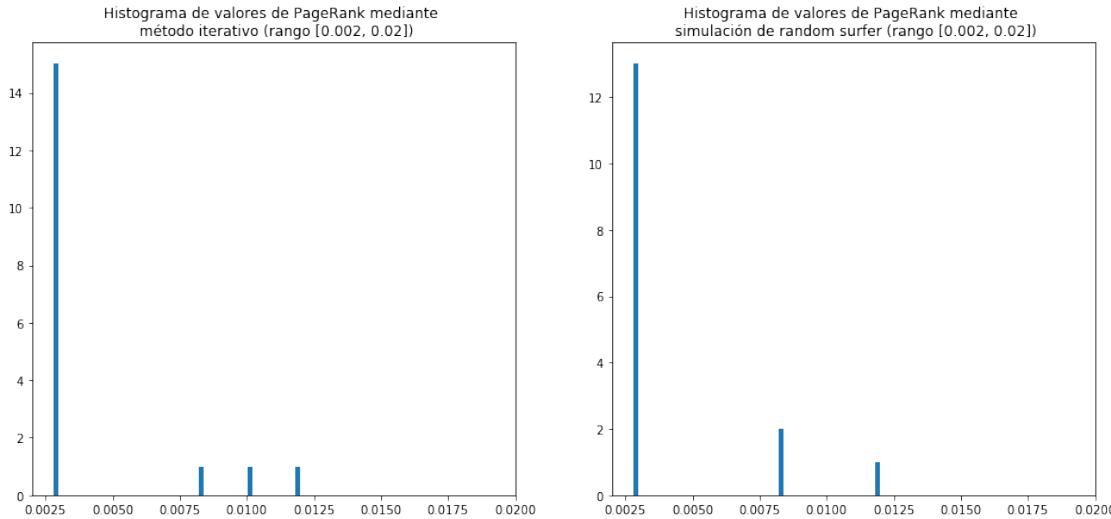


Luego, realizando un histograma para las páginas con *PageRank* entre [0.002, 0.02], podemos ver que tan solo hay 3 páginas con una relevancia alta (*PageRank* entre 0.0075 y 0.0125), seguidas por aproximadamente 15 páginas con un *PageRank* apenas mayor a 0.0025.

In:

```
plt.figure(figsize=(16, 7))
plt.subplot(1,2,1)
plt.hist(list(pagerank.values()), range=(0.002,0.02), rwidth=0.1)
plt.title(
    "Histograma de valores de PageRank mediante \n"
    "método iterativo (rango [0.002, 0.02])")
plt.subplot(1,2,2)
plt.hist(list(practical_pagerank.values()), range=(0.002,0.02), rwidth=0.1)
plt.title(
    "Histograma de valores de PageRank mediante \n"
    "simulación de random surfer (rango [0.002, 0.02])")
plt.show()
```

Out:



4.2 Visualización del Grafo

A continuación se realizaron visualizaciones del grafo de links. Para realizar estas visualizaciones se utilizó un muestreo de los nodos, ya que hay demasiados nodos para que se puedan visualizar claramente.

In:

```
cvsopts = dict(plot_height=5000, plot_width=5000)

def nodesplot(nodes, name=None, canvas=None, cat=None):
    canvas = ds.Canvas(**cvsopts) if canvas is None else canvas
    aggregator=None if cat is None else ds.count_cat(cat)
    agg=canvas.points(nodes, 'x', 'y', aggregator)
    return tf.spread(tf.shade(agg, cmap=["#FF3333"])), px=4, name=name

def edgesplot(edges, name=None, canvas=None):
    canvas = ds.Canvas(**cvsopts) if canvas is None else canvas
    return tf.shade(canvas.line(edges, 'x', 'y', agg=ds.count()), name=name)

def graphplot(nodes, edges, name="", canvas=None, cat=None):
    if canvas is None:
        xr = 0.0, 1.0
        yr = 0.0, 1.0
        canvas = ds.Canvas(x_range=xr, y_range=yr, **cvsopts)

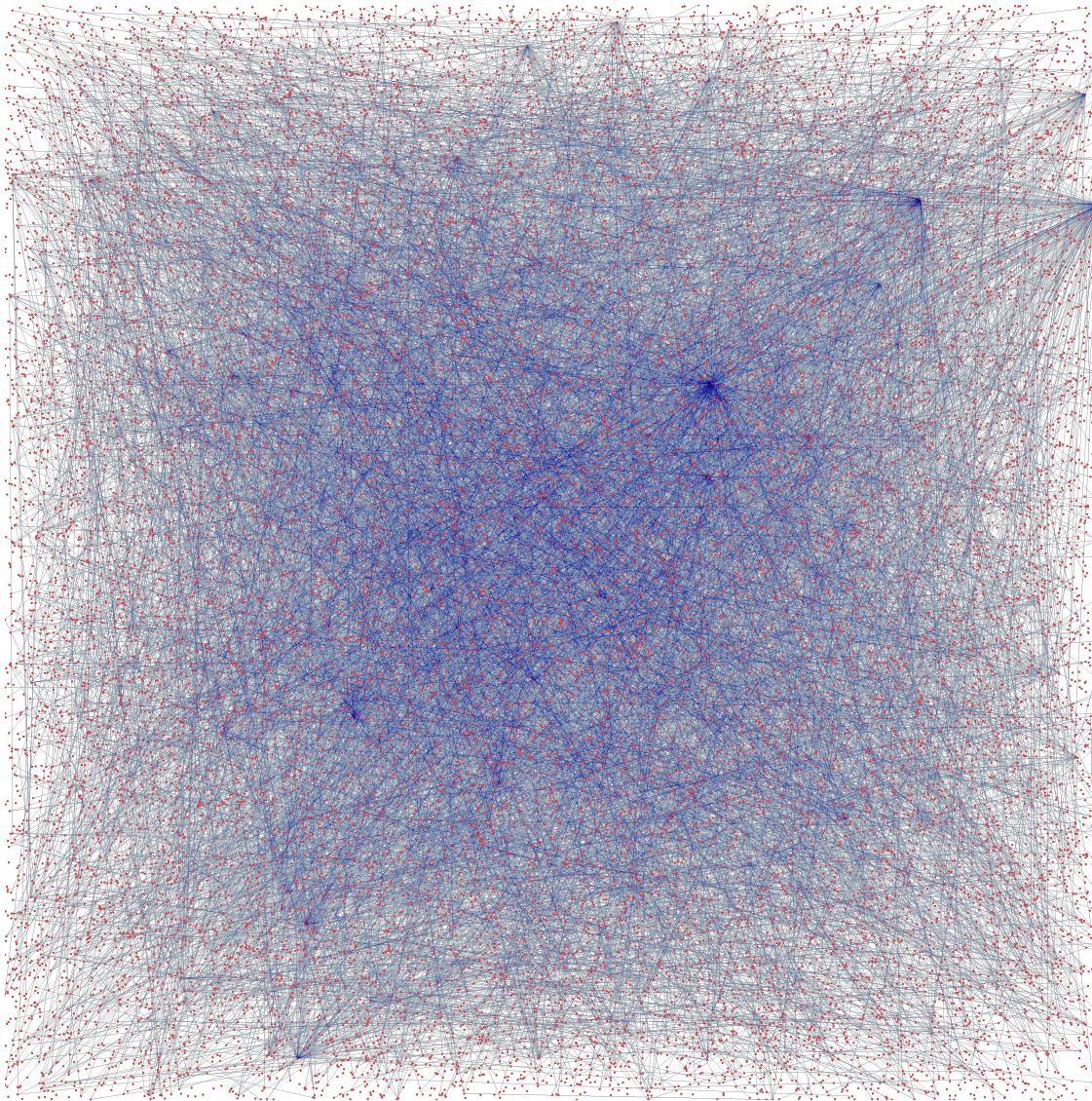
    np = nodesplot(nodes, name + " nodes", canvas, cat)
    ep = edgesplot(edges, name + " edges", canvas)
    return tf.stack(ep, np, how="over", name=name)
```

En este primer gráfico se realizó un sampleo del 7%. En él se pueden ver que hay muchos nodos con pocas conexiones y unos pocos muy conectados, lo cual está en concordancia con lo que se observa de los histogramas.

In:

```
centralalloc = random_layout(nodes.sample(frac=0.07))
graphplot(centralalloc, connect_edges(centralalloc,edges), "Random layout")
```

Out:



En la siguiente visualización del grafo están graficados solo los nodos con un *PageRank* mayor a 0.000004. En este, se puede observar que hay un nodo que se encuentra claramente más conectado que el resto, seguido de otro nodo también muy conectado, y luego un conjunto de nodos que también se destacan. Es probable que todos estos nodos sean los asociados a los de mayor *PageRank*.

In:

```
print("Porcentaje de nodos con un PageRank mayor a 0.000004: {}%"
      .format(len(nodes[nodes.PR_1>0.000004])/len(nodes)*100))
```

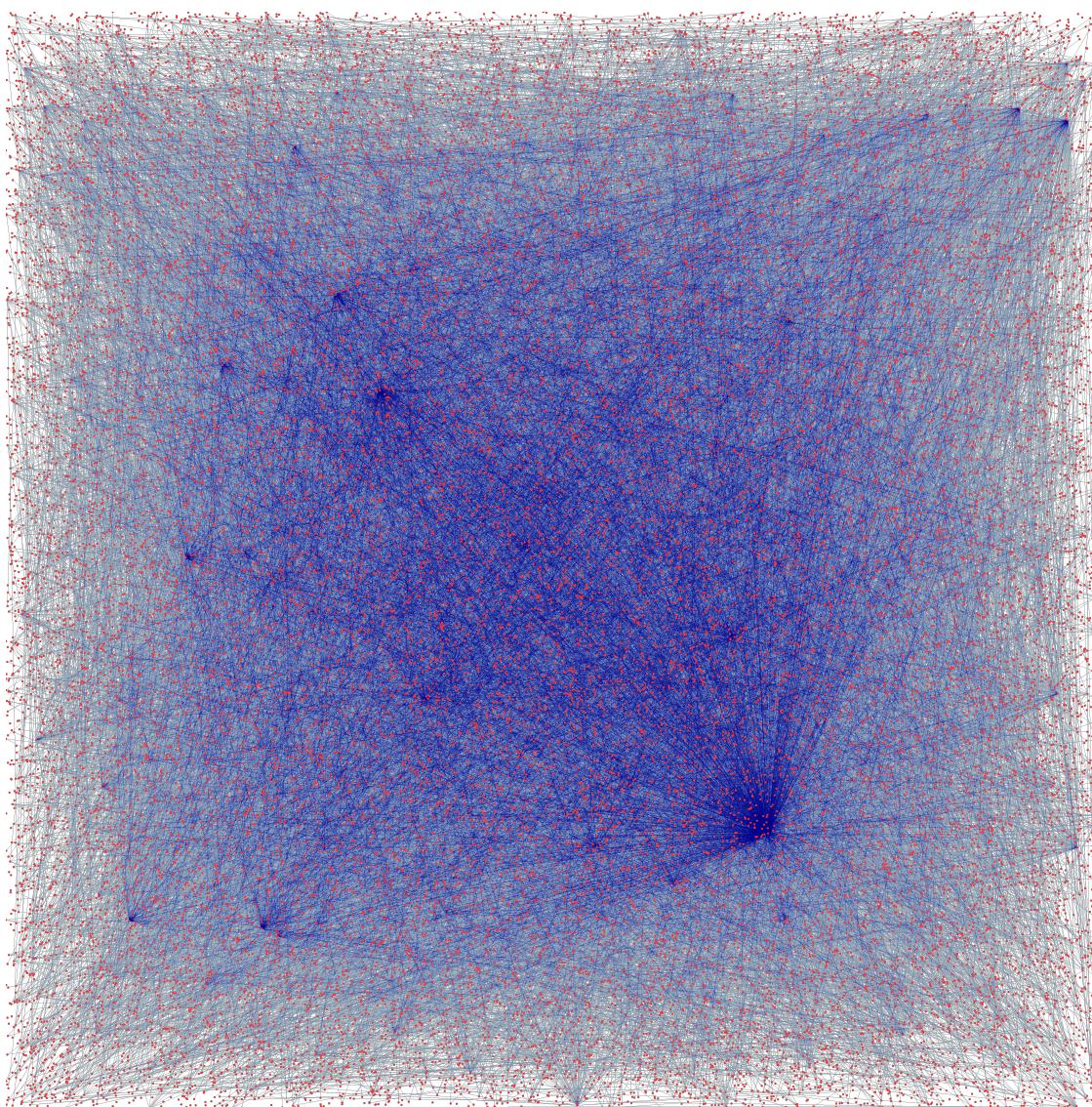
Out:

```
Porcentaje de nodos con un PageRank mayor a 0.000004: 9.972224488565216%
```

In:

```
centralalloc = random_layout(nodes[nodes.PR_1>0.000004])
graphplot(centralalloc, connect_edges(centralalloc,edges), "Random layout")
```

Out:



5 Conclusiones

A modo de conclusión se puede decir que el algoritmo de PageRank emplea el grafo de links para poder priorizar las páginas. Este algoritmo simula un viajante aleatorio que recorre el grafo, siendo el *PageRank* de una página la probabilidad de que pase por esa página. Esto puede ser interpretado como una cadena de markov, en la cual los estados son las páginas y las transiciones vienen dadas por los enlaces de una página a otra (siendo estos equiprobables). Este modelo requiere que el grafo sea irreducible y aperiodico para que converja, lo cual no puede ser garantizado, por esto es que se requiere el factor de teletransportación β .

En cuanto a los resultados obtenidos, se puede ver que los resultados obtenidos mediante el algoritmo iterativo y mediante la simulación del random surfer son indistinguibles. Asimismo, estos resultados se pueden ver reflejados en los gráficos y visualizaciones realizadas.