DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Métodos Numéricos
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
José Massigoge	954/12	jmmassigoge@gmail.com
Iván Pondal	078/14	ivan.pondal@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El objetivo principal de este Trabajo Práctico es estudiar, implementar y analizar algoritmos de Rankeo en dos escenarios distintos: tanto para páginas Web como para competencias deportivas.

Comenzaremos haciendo una breve introducción al algoritmo *PageRank*, el cual es muy conocido por ser utilizado por el buscador Google, para luego pasar a describir el Modelo que lo sostiene, donde explicaremos como resuelve este algoritmo el problema de rankear páginas web.

Luego, veremos una posible optimización de *PageRank* al modelar el problema de una manera equivalente utilizando matrices esparsas. Para esto, presentaremos y analizaremos 3 estructuras de datos distintas que nos permitirán mejorar la complejidad espacial y temporal de dicho algoritmo. Además, demostraremos que el Algoritmo 1 propuesto en Kamvar et al.[6] para trabajar con matrices esparsas es correcto.

A continuación, presentaremos el Modelo del algoritmo *GeM* (Generalized Markov chains Method) basado en *PageRank*. *GeM*, a diferencia de su padre, no se orienta a páginas web, sino a competencias deportivas, por lo que mostraremos los aspectos en los cuales difieren. En particular, mostraremos el problema de considerar deportes con empates al utilizar *GeM* y dos formas alternativas de modelar dicho escenario.

Una vez finalizada la parte del Modelo, pasaremos a describir la Implementación de los diferentes algoritmos presentados. Dichas implementaciones fueron realizadas algunas en `C++` y otras en `MATLAB/Octave`.

Ya llegando al final, pasaremos a presentar la Experimentación realizada, a la vez que iremos analizando y discutiendo los resultados obtenidos.

Los experimentos realizados para *PageRank* fueron:

- Comparación Page Rank normal vs Optimización matriz esparsa.
- Convergencia del algoritmo.
- Tiempos de ejecución.
- Calidad de los resultados.

Y para *GeM*:

- Variación del parámetro c.
- Evolución del ranking por cada iteración.

Para finalizar, cerraremos el presente informe con una conclusión, en la cual discutiremos acerca de los algoritmos vistos, así como de la experimentación realizada. También, contaremos las dificultades encontradas al realizar el Trabajo Práctico, las posibles continuaciones que se podrían realizar, y si los objetivos planteados fueron alcanzados.

2. Modelo

2.1. Rankings de Páginas Web

Hoy en día la cantidad de páginas web en todo el mundo asciende a una cantidad de 4.79 billones (sólo las indexadas). Es por esta razón que los buscadores (Search Engines) cumplen un rol tan importante en el uso diario de internet desde hace muchos años.

Dichos buscadores nos permiten realizar búsquedas de páginas web mediante distintos criterios, facilitando el acceso a la información.

Un posible criterio (bastante utilizado) es considerar que las páginas web populares son las más buscadas. De esta forma, el buscador puede ofrecernos en orden descendiente de popularidad los resultados obtenidos, esperando que encontremos más rápido lo que buscamos.

Siguiendo esta intuición, se han elaborado diferentes algoritmos para "rankear" las páginas web. A continuación presentaremos el famoso método PageRank, utilizado por Google.

2.1.1. PageRank

El algoritmo de PageRank[4] se define para un conjunto de páginas Web $= \{1, \ldots, n\}$ de forma tal de asignar a cada una de ellas un puntaje que determine la importancia relativa de la página respecto de las demás.

Llamemos x_j al puntaje asignado a la página $j \in Web$, que es lo que buscamos calcular.

Ahora, un link saliente de la página j a la página i puede significar que i es una página importante. Pero bien podría ser que j sea una página muy poco importante, por lo que deberíamos ponderar sus links salientes para decidir la importancia de las páginas a las que apunta.

Luego, vamos a considerar que la importancia de la página i obtenida mediante el link de j es proporcional a la importancia de j e inversamente proporcional al grado de j. Entonces, si $L_k \in Web$ es el conjunto de páginas web que apuntan a la página k:

$$x_k = \sum_{j \in L_k} \frac{x_j}{n_j}, \quad k = 1, \ldots, n \tag{1}$$

En este algoritmo, hallaremos los x_k modelando el problema como una cadena de Markov, a la cual llamaremos Matriz de Transición, y que construiremos de la siguiente forma:

1. Sea G el grafo de la Web, dónde cada vértice es una página web y un eje de v a u significa que la página v tiene link saliente hacia u.

2. Luego, sea $W \in \{0,1\}^{n \times n}$ la *matriz de conectividad* de G, tal que la celda $\{i, j\}$ tiene un 1 si hay un link saliente de la j-ésima página a la i-ésima página (los *autolinks* son ignorados, o lo que es lo mismo, $\forall\ 1 \leq i \leq n\ W_{i,i} = 0$).

3. Si definimos $n_j = \sum_{i=1}^{n} W_{i,j}$ como el grado de j (la cantidad de links salientes), entonces podemos definir la matriz $P \in \mathbb{R}^{n \times n}$, tal que la $P_{i,j} = 1/n_j * W_{i,j}$, y P es estocástica por columnas.

   Además, nótese que resolver el sistema dado por 1 es equivalente a encontrar un $x \in \mathbb{R}^n$ tal que $Px = x$. Es decir, encontrar el autovector asociado al autovalor 1 de P tal que $x_i > 0$ y $\sum_{i=1}^{n} x_i = 1$.

4. Ahora, puede pasar que para algún j, $n_j = 0$ lo que indicaría que la página j no tiene ningún link saliente. Para remediar estos casos, vamos a modificar P utilizando la idea del *navegante aleatorio*, de forma tal que para un j sin links salientes, la probabilidad de que el navegante salte a cualquier otra página i es $1/n$.

   Entonces, $P_1 = P + D$, dónde $D = vd^t$, $d \in \{0,1\}^n$ tal que $d_j = 1$ si $n_j = 0$, y $d_j = 0$ en caso contrario, y $v \in \mathbb{R}^n$ tal que $v_j = 1/n$.

5. Por lo tanto, P_1 es estocástica por columnas, pero puede que no sea regular. Para que sí lo sea, extendemos el concepto anterior a todas las páginas (fenómeno de *teletransportación*).

Luego, $P_2 = c * P_1 + (1 - c) * E$, donde $\forall\ 1 \leq i, j \leq n,\ E_{i,j} = 1/n$, y $c \in (0, 1)$. Llamamos a c coeficiente de teletransportación.

Lo que nos queda es un matriz P_2 estocástica por columnas y $\forall\ 1 \leq i, j \leq n,\ (P_2)_{i,j} > 0$.

Una vez que tenemos la Matriz de Transición, generaremos el puntaje para cada página buscando el autovector w del autovalor 1 de P_2, tal que $P_2 w = w$ y w sea un vector de probabilidades (normalizado con norma 1).

Es decir, generar el ranking de páginas equivale a aplicar el Método de la Potencia a la matriz P_2 y una vez hallado el w mencionado, ordenar los puntajes de mayor a menor:

$$ranking = \{p_1, \ldots, p_n\}, \text{donde } \forall\ i = 1, \ldots, n-1,\ w_{p_i} \geq w_{p_{i+1}} \tag{2}$$

Falta ver que podemos utilizar el Método de la Potencia con P_2. Pero como P_2 es una matriz de transición regular, entonces se cumple que:

- 1 es un autovalor de P_2.
- Hay un único vector de probabilidades que es el autovector asociado al autovalor 1.
- Los demás autovalores cumplen:
$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$$

Las condiciones que debe cumplir una matriz $A \in \mathbb{R}^{n \times n}$, con autovalores $\lambda_1, \lambda_2, ..., \lambda_n$, y $v_1, v_2, ..., v_n$ los autovalores asociados, para poder aplicar el método son:

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$$

$$v_1, v_2, \ldots, v_n \text{ son autovectores l.i.}$$

Por lo visto, P_2 cumple los requisitos necesarios para utilizar el Método de la Potencia. Explicaremos en detalle el Método de la Potencia en la sección 2.3.

2.1.2. PageRank con matriz Esparsa

Partiendo del hecho de que, aproximadamente, cada pagina Web tiene 7 links salientes[6] surge la posibilidad de optimizar la estructura que representa el Grafo dirigido de la Web, debido a la baja densidad de la matriz de conectividad, W, asociada a dicho Grafo. Esta optimización no es solo una posibilidad, sino una necesidad dado que la cantidad de páginas indexadas tiene un requerimiento prohibitivo en términos de memoria si la representamos, ingenuamente, como un vector de vectores. Nuestra hipótesis es que, utilizando una estructura de datos optimizada, no solo obtendremos mejoras en términos de complejidad espacial, sino también en términos de complejidad temporal.

A partir de este análisis, nos propusimos analizar tres estructuras de datos que representan matrices esparsas:

1. *Dictionary of Keys (dok)*
2. *Compressed Sparse Row (CSR)*
3. *Compressed Sparse Column (CSC)*

Sea $A \in \mathbb{R}^{n \times n}$ una matriz esparsa, y sea m es la cantidad de valores distintos de 0 de A.

1. Dictionary of Keys (dok): Esta estructura esta representada como un vector de diccionarios, en donde cada posición, i, del vector puede considerarse como una fila, y la key del diccionario contenido dentro de esa posición representa una columna, j, siendo el valor asociado a esa key, en el diccionario, el valor en la matriz contenido en esa fila y columna (A_{ij}). Solo aquellas A_{ij} que tienen valores distintos de 0 tienen una key en los diccionarios.

Veamos un ejemplo. Sea $B \in \mathbb{R}^{4\times4}$ la siguiente matriz:

$$B = \begin{pmatrix} 10 & 0 & 0 & -2 \\ 3 & 9 & 0 & 0 \\ 0 & 7 & 8 & 0 \\ 3 & 0 & 4 & 5 \end{pmatrix}$$

El dictionary of keys que representa a B es:

{[1;10], [4;-2]}	{[1;3], [2;9]}	{[2;7], [3;8]}	{[1;3], [3;4], [4;5]}

Esta claro que intercambiando columnas por filas, como posiciones del vector contenedor, y filas por columnas como keys, representamos la misma matriz.

Las ventajas de usar esta estructura de datos son las siguientes:

1. Facilidad para crearse de forma dinámicamente, es decir, si a priori no conocemos la forma de la matriz a representar.
2. La operación matriz esparsa por vector es sencilla.

Por otro lado, su principal desventaja es que:

1. Las operaciones entre matrices esparsas, sea la suma o la multiplicación, son engorrosas.

2. Compressed Sparse Row (CSR): Esta estructura utiliza tres vectores, dos de tamaño m, llamémosles *val* y *col_ind* respectivamente, y uno de tamaño $n+1$, llamémosle *row_ptr*.

- En *val* guardamos los valores de A distintos de 0, recorriendo A por filas, es decir fijando las filas y avanzado por las columnas.
- En la posición k de *col_ind* guardamos el índice j del valor A_{ij} contenido en la posición k del vector *val*.
- Por ultimo en *row_ptr* guardamos índices del vector *val* en los cuales se encuentran el primera valor de cada fila, de tal forma que para la fila i, el valor k contenido en *row_ptr[i]*, nos dice que *val[k]* corresponde al primer valor de la fila i y el valor k' contenido en *row_ptr[i+1]*, nos dice que *val[k'-1]* corresponde al ultimo valor de la fila i. Vale la pena aclarar que si $k = k'$ entonces la fila no tiene valores distintos de 0.

Utilicemos B como la matriz del punto anterior:

$$B = \begin{pmatrix} 10 & 0 & 0 & -2 \\ 3 & 9 & 0 & 0 \\ 0 & 7 & 8 & 0 \\ 3 & 0 & 4 & 5 \end{pmatrix}$$

La misma seria representado de la siguiente manera:

- *val:*

10	-2	3	9	7	8	3	4	5

- *col_ind:*

1	4	1	2	2	3	1	3	4

- *row_ptr:*

1	3	5	7	10

Las ventajas de usar esta estructura de datos son las siguientes:

1. Eficiente a la hora de realizar operaciones aritméticas entre matrices esparsas (suma, multiplicación).
2. La operación matriz esparsa por vector es sencilla.

Por otro lado, sus desventajas son:

1. Dificultad a la hora de crearse de forma dinámica, ya que debemos recalcular/redimensionar los tamaños de los vectores.
2. Cambios en la esparcidad son costosos, misma razón que el punto anterior.

3. Compressed Sparse Column (CSC): Muy parecida a la *Compressed Sparse Row (CSR)*. También utiliza tres vectores, dos de tamaño m, llamémosles *val* y *row_ind* respectivamente, y uno de tamaño $n+1$, llamémosle *col_ptr*.

- En *val* guardamos los valores de A distintos de 0, recorriendo A por columnas, es decir fijando las columnas y avanzado por las filas.
- En la posición k de *row_ind* guardamos el índice i del valor A_{ij} contenido en la posición k del vector *val*.
- Por ultimo en *col_ptr* guardamos índices del vector *val* en los cuales se encuentran el primera valor de cada columna, de tal forma que para la columna j, el valor k contenido en *col_ptr[j]*, nos dice que *val[k]* corresponde al primer valor de la fila j y el valor k' contenido en *col_ptr[j+1]*, nos dice que *val[k'-1]* corresponde al ultimo valor de la col j. Vale la pena aclarar que si $k = k'$ entonces la columna no tiene valores distintos de 0.

Nuevamente utilicemos B como ejemplo:

$$B = \begin{pmatrix} 10 & 0 & 0 & -2 \\ 3 & 9 & 0 & 0 \\ 0 & 7 & 8 & 0 \\ 3 & 0 & 4 & 5 \end{pmatrix}$$

La misma seria representado de la siguiente manera:

- *val:*

10	3	3	9	7	8	4	-2	5

- *row_ind:*

1	2	4	2	3	3	4	1	4

- *col_ptr:*

1	4	6	8	10

Las ventajas de usar esta estructura de datos son las siguientes:

1. Eficiente a la hora de realizar operaciones aritméticas entre matrices esparsas (suma, multiplicación).
2. La operación matriz esparsa por vector es sencilla.

Por otro lado, sus desventajas son:

1. Dificultad a la hora de crearse de forma dinámica, ya que debemos recalcular/redimensionar los tamaños de los vectores.
2. Cambios en la esparcidad son costosos, misma razón que el punto anterior.

Contexto de Uso Antes de definir que estructura de datos vamos a utilizar para representar matrices esparsas, debemos definir el contexto de uso de la misma. Sabemos que el algoritmo de PageRank utiliza el Método de la Potencia, usando la matriz P_2 definida previamente. Sin embargo la P_2 no es esparsa, inclusive todos sus valores son distintos de 0, por lo cual utilizar la implementación del Page Rank descripta en la Sección 2.1.1 no es posible .

Para poder hacer frente a esta dificultad, utilizamos el Algoritmo 1 propuesto por Kamvar et al.[6] para calcular $x^{(k+1)} = P_2 x^k$, cuyos pasos son los siguientes:

1. $y = cPx$
2. $w = ||x||_1 - ||y||_1$
3. $y = y + wv$

donde c, P y v corresponden al escalar, la matriz y el vector descriptos en la Sección 2.1.1.
Tenemos que ver que el algoritmo descripto calcula efectivamente $x^{(k+1)} = P_2 x^k$:

Proposición 1. *El Algoritmo 1 calcula* $x^{(k+1)} = P_2 x^{(k)}$

Demostración. Por definición sabemos que

$$\begin{aligned} x^{(k+1)} &= P_2 x^{(k)} \\ &= (cP_1 + (1-c)E)x^{(k)} \\ &= (cP + cD + (1-c)E)x^{(k)} \\ &= (cP + cD + (1-c)E)x^{(k)} \\ &= (cP + cvd^t + (1-c)v1^t)x^{(k)} \\ &= cPx^{(k)} + cvd^t x^{(k)} + (1-c)v1^t x^{(k)} \end{aligned}$$

Por otro lado, el Algoritmo 1 nos dice que:

$$x^{(k+1)} = cPx^{(k)} + (||x^{(k)}||_1 - ||cPx^{(k)}||_1)v$$

Luego basta ver que:

$$\begin{aligned} cPx^{(k)} + (||x^{(k)}||_1 + ||cPx^{(k)}||_1)v &= cPx^{(k)} + cvd^t x^{(k)} + (1-c)v1^t x^{(k)} \\ (||x^{(k)}||_1 - ||cPx^{(k)}||_1)v &= cvd^t x^{(k)} + (1-c)v1^t x^{(k)} \\ ||x^{(k)}||_1 v - ||cPx^{(k)}||_1 v &= cvd^t x^{(k)} + v1^t x^{(k)} - cv1^t x^{(k)} \end{aligned}$$

Sabemos que $\forall\ 1 \leq i \leq n\ x_i^{(k)} \geq 0$, ya que x es un vector de probabilidades. Entonces $||x^{(k)}||_1 = \sum_{i=1}^{n} x_i^{(k)}$.
También sabemos que $1^t x^{(k)} = \sum_{i=1}^{n} 1 x_i^{(k)}$, por lo tanto $||x^{(k)}||_1 = 1^t x^{(k)}$.
Por otro lado sabemos que $0 \leq c \leq 1$, lo que implica que $|c| = c$.
Teniendo en cuenta esta información, la igualdad nos queda de la siguiente forma:

$$\begin{aligned} v1^t x^{(k)} - c||Px^{(k)}||_1 v &= cvd^t x^{(k)} + v1^t x^{(k)} - cv1^t x^{(k)} \\ -c||Px^{(k)}||_1 v &= cvd^t x^{(k)} - cv1^t x^{(k)} \end{aligned}$$

Tomando como factor común v y c:

$$\begin{aligned} -c||Px^{(k)}||_1 v &= cv(d^t x^{(k)} - 1^t x^{(k)}) \\ -||Px^{(k)}||_1 &= d^t x^{(k)} - 1^t x^{(k)} \\ -||Px^{(k)}||_1 &= (d^t - 1^t) x^{(k)} \end{aligned}$$

Sabemos, por definición de d, que d^t solo tiene 1 y 0 como valores, por lo tanto al realizar $(d^t - 1^t)$ obtendremos un vector fila, llamémosle z, cuyos valores serán 0 o -1. Lo que implica que $z_j = 0$ cuando $d^t_j = 1$, situación que se da cuando, para ese j, $\sum_{i=1}^{n} P_{ij} = 0$. Caso contrario $z_j = -1$ cuando $d^t_j = 0$, situación que se da cuando, para ese j, $\sum_{i=1}^{n} P_{ij} = 1$. Por lo tanto al multiplicar $zx^{(k)}$ obtendremos un escalar que sera la suma de los opuestos de los $x^{(k)}_j$ que cumplan que, para ese j, $\sum_{i=1}^{n} P_{ij} = 1$, ya que los otros serán multiplicados por 0.

Por otro lado, partiendo del hecho de que P es estocástica por columnas, sabemos que al realizar $Px^{(k)}$, aquellas columnas, j, que cumplan que $\sum_{i=1}^{n} P_{ij} = 0$, forzaran a que los $x^{(k)}_j$ no sean parte de ninguna componente del vector resultante, ya que en todas las multiplicaciones de filas de P por el vector $x^{(k)}$, estos $x^{(k)}_j$ serán multiplicados por 0. Caso contrario cuando las columnas, j, cumplan que $\sum_{i=1}^{n} P_{ij} = 1$, estas forzaran a que la suma de todos los $x^{(k)}_j$ presentes en las distintas componentes del vector resultante sea igual a $x^{(k)}_j$. Precisamente al aplicar $||.||_1$ al vector $Px^{(k)}$, el resultado sera un escalar que sera la suma de todas los $x^{(k)}_j$ que cumplan la condición antes descripta. Por ultimo si al resultado de aplicar $||.||_1$ lo multiplicamos por -1 obtenemos un escalar que sera la suma de los opuestos de los $x^{(k)}_j$ que cumplan que, para ese j, $\sum_{i=1}^{n} P_{ij} = 1$.

Por todo lo expuesto anteriormente, podemos concluir que $-||Px^{(k)}||_1 = (d^t - 1^t)x^{(k)}$, lo que implica que El Algoritmo 1 calcula $x^{(k+1)} = P_2 x^{(k)}$. □

Luego, sabemos que la única operación que vamos a realizar con la matriz esparsa es multiplicarla por un vector. También sabemos que la construcción de la matriz es de forma dinámica, ya que vamos leyendo los links salientes de cada pagina de forma secuencial.

A partir de esta información concluimos que la estructura de datos que mas se adecua a nuestras necesidades es el *Dictionary of Keys (dok)*, en donde las posiciones del vector representan las filas y las keys de los diccionarios las columnas.

Esta elección se debe al hecho de que podemos actualizar la matriz de forma sencilla a medida de que vamos leyendo los datos de entrada, y a su vez la multiplicación por un vector se realiza de forma sencilla, véase Sección 3.1.2.

2.2. Rankings en competencias deportivas

Elegir un sistema de puntos que sea justo para todos los participantes en un deporte no es una tarea sencilla. Existen muchos factores que afectan el resultado de una competencia como lo pueden ser el orden en el que deben competir entre si los equipos, generando desbalances respecto las capacidades de cada uno. Es por esto que a continuación presentaremos el modelo GeM[5] que busca modelar los resultados de forma tal que estos factores impacten lo menos posible en el posicionamiento final de la tabla de puntajes.

Con el fin de experimentar con distintos modelos, a lo largo del informe trabajaremos sobre los resultados del Torneo de Primera División 2015[1], donde utilizaremos el sistema de ranking estándar de la AFA como punto de comparación.

2.2.1. Generalized Markov chains Method (GeM)

Definición del método

El método GeM es el resultado de tomar el algoritmo PageRank y mediante pequeñas modificaciones utilizar su potencial para establecer un ranking de equipos. Análogo a PageRank, los equipos pasan a formar parte de un grafo dirigido con pesos, donde cada nodo representa un equipo y los pesos de cada arista reflejan el resultado de los partidos jugados entre los vértices conectados.

Formalmente, el modelado se realiza de la siguiente manera:

1. Representamos el torneo como un grafo con pesos dirigidos de n nodos, donde n es igual a la cantidad de equipos que participan. Cada equipo tiene su respectivo nodo y las aristas contienen como peso la diferencia positiva entre los nodos conectados.

2. Definimos la matriz de adyacencia $A \in \mathbb{R}^{n \times n}$.

$$A_{ij} = \begin{cases} w_{ij} & \text{si el equipo } i \text{ perdió contra } j \\ 0 & \text{caso contrario} \end{cases}$$

   Donde w_{ij} es la suma total de diferencia positiva de puntaje sobre todos los partidos en los que i perdió contra j.

3. Definimos la matriz $H \in \mathbb{R}^{n \times n}$.

$$H_{ij} = \begin{cases} A_{ij} / \sum_{k=1}^{n} A_{ik} & \text{si hay un link de } i \text{ a } j \\ 0 & \text{caso contrario} \end{cases}$$

4. Definimos la matriz GeM, $G \in \mathbb{R}^{n \times n}$ con $u, v, a, e \in \mathbb{R}^n$ y $c \in \mathbb{R}$.

$$G = c(H + au^t) + (1 - c)ev^t$$

$$\sum_{k=1}^{n} v_k = 1 \qquad \sum_{k=1}^{n} u_k = 1 \qquad \forall_{i=1..n} e_i = 1$$

$$0 \leq c \leq 1 \qquad a_i = \begin{cases} 1 & \text{si la fila } i \text{ de } H \text{ es un vector nulo} \\ 0 & \text{caso contrario} \end{cases}$$

5. Por último tenemos que el ranking de los equipos estará definido por el vector $\pi \in \mathbb{R}^n$ tal que

$$\pi^t = \pi^t G$$

o si tomamos la transpuesta en ambos lados

$$G^t \pi = \pi$$

[1] Campeonato de Primera División 2015, *Julio H. Grondona*
`http://www.afa.org.ar/html/9/estadisticas-de-primera-division`

De esta forma al igual que con PageRank, calculando el autovector π obtenemos nuestro ranking. Este modelo permite cierta flexibilidad a partir del u, v y c que tomemos.

El vector de probabilidad u se aplicará en el caso de que un equipo se encuentre invicto, esto es el equivalente a que en PageRank un sitio no tenga ningún link entrante, por lo tanto su tratamiento es el mismo, se le asigna a la fila correspondiente el vector con las probabilidades de saltar a otro nodo. Por lo tanto, el vector u nos permite definir con qué probabilidades un equipo invicto perdería contra el resto de los participantes. En el caso de PageRank, este es un vector de distribución uniforme, donde es igual la posibilidad de saltar a cualquiera de los otros nodos, una posible alternativa sería definirlo como un vector cuyas probabilidades se basen en algún ranking anterior.

El vector de probabilidad v, nos da otro tipo de personalización que es la del *navegante aleatorio*. Esta es la probabilidad de que un equipo cualquiera independientemente de los resultados registrados, pierda contra el resto de los equipos. En PageRank, esto lo veíamos como la posibilidad de que estando navegando el grafo, uno se *teletransportará* a otro nodo independientemente de las conexiones de los mismos. Este vector por defecto también suele tomar el valor de la distribución uniforme.

Por último tenemos nuestro valor c que actúa como un factor de amortiguación donde lo que se modifica es cuánto afecta el *navegante aleatorio* al resultado final, donde con $c = 0$, únicamente influye el *navegante aleatorio* y con $c = 1$ se elimina el efecto del mismo.

Modelado del empate

Una particularidad de este sistema que se puede observar en la definición del mismo es que no contempla los partidos donde hubo empate. Un empate equivale a que no exista un perdedor y por ende no se modifica el peso de ningún nodo. Para deportes donde el empate no es algo frecuente esto no sería un problema, pero si tomamos como ejemplo el fútbol, donde los empates son algo mucho más común, el ignorar estos partidos afecta notablemente el ranking.

Podemos tomar como ejemplo los Torneos Argentinos de Primera División y observar el porcentaje de partidos empatados sobre el total que se jugaron.

Campeonato de Primera División	Partidos	Empates	Empates/Partidos
2015	390	126	0.32
2014	190	45	0.23
2013/14	380	117	0.31

Relación entre partidos jugados y empates totales

Como podemos ver, la proporción de partidos empatados no es menor, con lo cual analizaremos cómo impactaría esto en el método GeM. Como mencionamos en la definición del modelo, los pesos de las aristas en el grafo dependen en parte de la diferencia de puntaje con la que gana un equipo sobre otro.

Al encontrarnos con un empate no existe diferencia alguna, por lo tanto no implica ningún cambio en el sistema. Esto podría desfavorecer a los equipos que más empates tuvieran en la temporada, ya que sólo los partidos donde hubieran ganado o perdido afectarían su lugar en la tabla de puntajes. Además podría suceder que un equipo perdiese contra un contrincante que no tuviera mucho peso pero empatara contra un puntero, y esto debería verse reflejado ya que de otra manera, sólo quedaría registrado su partido perdido.

En base a esto, proponemos dos formas alternativas de modelar el empate:

- Una posible forma de representar los empates aprovechando la estructura del modelo actual sería que cuando se produce uno, reflejar en el grafo el equivalente a que ambos equipos hubieran perdido entre si, asignando como diferencia de puntaje el valor con el que empataron. De esta forma no solo incluiríamos este escenario al cálculo del ranking final, si no que además el puntaje con el que empataron tendría peso.

- Otra posibilidad sería que en caso de empate, se utilizaran otras métricas del deporte para definir el ganador, al cual luego se le podría asignar el puntaje mínimo para ganar el partido. Por ejemplo, si tomamos el caso del fútbol, en caso de empate podríamos definir el ganador como el equipo que

tuvo mayor posesión del balón durante el partido, o el equipo que tuvo menor cantidad de tarjetas, o una combinación de varios factores. Una vez determinado el ganador, el score asignado sería 1-0. De esta forma, rompemos el empate refinando el criterio de ganador, y como asignamos el mínimo puntaje para ganar, el puntaje en el ranking que obtiene el ganador no es altamente influenciado por el partido en cuestión.

2.2.2. Puntaje AFA

A modo de comparación con el método GeM, siendo que trabajaremos sobre los resultados del Torneo de Primera División, utilizaremos también el sistema de puntajes que propone la AFA[2]. El mismo es bastante sencillo, donde lo que establece es que todos los equipos comienzan con puntaje **0** y se les suma **1** punto en caso de empate, **3** por victoria y **0** por derrota.

A primera vista, dada que la modalidad de este torneo es de todos contra todos, donde cada equipo juega una vez contra el resto, el resultado debería reflejar de forma justa el desempeño de cada participante. Sin embargo, para este torneo en particular se aplicó una fecha adicional de *clásicos*, donde cada equipo vuelve a jugar contra su respectivo clásico. Este es un punto donde se desequilibran un tanto las cosas ya que un participante podría tener asignado como clásico un puntero, mientras que otro al último en la tabla de posiciones. Para este sistema de puntuación ambos partidos serían tratados al igual que el resto del torneo.

[2] `http://www.afa.org.ar/upload/reglamento/Reglamento_PrimeraDivision_2015.pdf`

2.3. Método de la potencia

El Método de la Potencia es un método iterativo que calcula sucesivas aproximaciones a los autovectores y autovalores de una matriz.

Sea $A \in \mathbb{R}^{n\times n}$ una matriz cuadrada y sean $\lambda_1, \lambda_2, ..., \lambda_n$ sus autovalores, con $v_1, v_2, ..., v_n$ los autovalores asociados. Para poder aplicar el método a la matriz A, debe valer que:

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$$

$$v_1, v_2, \ldots, v_n \text{ son autovectores l.i.}$$

A continuación demostraremos como funciona el método.

Proposición 2. *Sea $x_0 \in \mathbb{R}^n$. Luego, $x_{(k+1)} = \frac{Ax_k}{||Ax_k||} \xrightarrow{k\to\infty} v_1$.*

Demostración. Como $\{v_1, \ldots, v_n\}$ es una base de autovectores l.i., podemos escribir $x_0 = \sum_{j=1}^{n} \beta_j v_j$.

Entonces, $Ax = \sum_{j=1}^{n} \beta_j A v_j = \sum_{j=1}^{n} \beta_j \lambda_j v_j$.

Si multiplicamos una vez más por A tenemos que: $A^2x = \sum_{j=1}^{n} \beta_j \lambda_j A v_j = \sum_{j=1}^{n} \beta_j \lambda_j^2 v_j$.

Y generalizando:

$$A^k x = \sum_{j=1}^{n} \beta_j \lambda_j^k v_j$$

Reescribiendo, tenemos que:

$$\begin{aligned} A^k x &= \sum_{j=1}^{n} \beta_j \lambda_j^k v_j \\ &= \beta_1 \lambda_1^k v_1 + \sum_{j=2}^{n} \beta_j \lambda_j^k v_j \\ &= \lambda_1^k \left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j \right) \end{aligned}$$

Y es fácil ver que $\lim_{k\to\infty} \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j = 0$, ya que $\forall\ 2 \leq j \leq n,\ |\lambda_1| > |\lambda_i|$.

Luego, consideremos una función $\phi : \mathbb{R}^n \to \mathbb{R}$ tal que:

- $\phi(\alpha x) = \alpha\phi(x)$
- $\phi(0) = 0$
- ϕ continua
- $\phi(\beta_1 v_1) \neq 0 \iff \beta_1 v_1 \neq 0$

Entonces, podemos escribir:

$$\frac{\phi(A^{k+1}x)}{\phi(A^k x)} = \frac{\lambda_1^{k+1}\phi\left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^{k+1}}{\lambda_1^{k+1}} v_j\right)}{\lambda_1^k \phi\left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j\right)} = \lambda_1 \frac{\phi\left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^{k+1}}{\lambda_1^{k+1}} v_j\right)}{\phi\left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j\right)}$$

Y tomando el límite con k tendiendo a infinito:

$$\lim_{k\to\infty} \frac{\phi(A^{k+1}x)}{\phi(A^k x)} = \lambda_1 \lim_{k\to\infty} \frac{\phi\left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^{k+1}}{\lambda_1^{k+1}} v_j\right)}{\phi\left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j\right)} = \lambda_1 \frac{\phi(\beta_1 v_1)}{\phi(\beta_1 v_1)} = \lambda_1$$

Finalmente,

$$x_{(k+1)} = \frac{Ax_k}{||Ax_k||} = \frac{A^k x}{||A^k x||} = \frac{\lambda_1^k \left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j \right)}{||\lambda_1^k \left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j \right)||}$$

Ahora, tomando límites:

$$\lim_{k\to\infty} \frac{\lambda_1^k \left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j \right)}{||\lambda_1^k \left(\beta_1 v_1 + \sum_{j=2}^{n} \beta_j \frac{\lambda_j^k}{\lambda_1^k} v_j \right)||} = \lim_{k\to\infty} \frac{\lambda_1^k \beta_1 v_1}{||\lambda_1^k \beta_1 v_1||} = \pm \frac{v_1}{||v_1||}$$

Lo cual tiende a un autovector asociado al autovalor principal λ_1. Si podemos garantizar que $\lambda_1^k \beta_1 > 0$, entonces podríamos decir que $x_{(k+1)}$ tiende a $\frac{v_1}{||v_1||}$ cuando k tiende al infinito. Además, si el vector v_1 se encuentra normalizado, concluimos que $x_{(k+1)} \xrightarrow{k\to\infty} v_1$.

En la práctica se utiliza $\phi(x) = x_p$, dónde $|x_p| = ||x||_\infty$, que cumple las condiciones pedidas. □

3. Implementación

3.1. Page Rank

A la hora de implementar Page Rank, en ambas versiones, utilizamos C++, lenguaje que nos permite realizar mediciones temporales precisas, como así también utilizar nuestras propias estructuras de datos. A su vez es necesario tener en cuenta que los datos de entrada nos son provistos de la siguiente forma:

- Cantidad de nodos (n) y cantidad de links salientes (m).
- m lineas con pares i, j, donde i tiene un link hacia j.

3.1.1. Page Rank sin matriz esparsa

En esta versión del Page Rank utilizamos un vector de vector para representar la matriz P_2, la cual es necesaria para utilizar el método de la potencia en su formulación clásica. Los paso para realizar Page Rank sin matriz esparsa son los siguientes:

1. Leemos los datos de entrada y luego generamos P

```
Leer n, m del input
Crear un vector de vectores, P, de tamanio n*n
Para k=0 hasta m-1 hacer:
  Leer i,j del input
  P[j-1][i-1] = 1
Fin Para
Crear un vector, links_salientes, de tamanio n
Para j=0 hasta n-1 hacer:
  Para i=0 hasta n-1 hacer:
    links_salientes[j] = links_salientes[j] + P[i][j]
  Fin Para
  Para i=0 hasta n-1 hacer:
    Si links_salientes[j] > 0 entonces:
      P[i][j] = P[i][j] / links_salientes[j]
    Fin Si
  Fin Para
Fin Para
```

   Vale la pena aclarar que utilizamos *links_salientes* para calcular los n_j necesarios para luego calcular los valores de P.

2. A partir de P y *links_salientes* generamos P_2

```
Definir c como el factor de teletransportacion
Crear un vector, v, de tamanio n iniciliazado en 1/n
Crear un vector, d, de tamanio 0 inicializado en 0
Para j=0 hasta n-1 hacer:
  Si links_salientes[j] = 0 hacer:
    d[j] = 1
  Fin Si
Fin Para
Crear vector de vectores, D = multiplicarVectores(v, d)
Crear vector de vectores, P1 = sumaMatrices(P, D)
Crear vector, uno, inicializado en 1
Crear vector de vectores, E = multiplicarVectores(v, uno)
Crear vector de vectores, P2 = sumaMatrices(escalarPorMatriz(P1, c
   ), escalarPorMatriz(E, 1-c)
```

Los procedimientos auxiliares multiplicarVectores, sumaMatrices y escalarPorMatriz son procedimientos estándar, por lo cual no los detallamos.

3. Calculamos el autovector asociado a 1 con el método de la Potencia con P_2, normalizando el autovector obtenido.

```
Definir una precision
Crear un vector, x, con valores aleatorios de 1 a 50 de tamanio n
Crear vector de vectores, B = multiplicarMatrices(P2, P2)
Crear vector de vectores, C = P2
delta = phi(multiplicarMatrizPorVector(B, x)) / phi(
   multiplicarMatrizPorVector(C, x))
ultimo_delta = INFINITY
Mientras (delta - ultimo_delta) > precision hacer:
  C = B
  B = multiplicarMatrices(B, P2)
  ultimo_delta = delta
  delta = phi(multiplicarMatrizPorVector(B, x)) / phi(
     multiplicarMatrizPorVector(C, x))
Fin Mientras
y = multiplicarMatrizPorVector(B, x)
Normalizar y
```

El procedimiento phi consiste en calcular la norma infinito del vector pasado por parámetro, con la variación de no tomar el modulo de las componentes del vector.

Los procedimientos auxiliares multiplicarMatrices, multiplicarMatrizPorVector y phi son procedimientos estándar, por lo cual no los detallamos.

Normalizar un vector, en nuestra implementación, implica dividir cada componente del vector por la norma 1 del mismo.

4. Una vez obtenido el autovector estacionario, y, generamos el ranking de las paginas y lo imprimimos:

```
Crear un vector de pares posición-valor, llamado ranking, de
   tamanio n inicializado en 0
Para i=0 hasta n-1 hacer:
  ranking[i] = par(i, y[i])
Fin Para
Ordenar ranking según la segundo componente de los pares, valor
imprimir(ranking)
```

Para ordenar el vector ranking utilizamos el procedimiento que nos provee *C++*, `sort`[3].

El procedimiento consiste, simplemente, en recorrer el vector ranking e ir imprimiendo sus pares.

3.1.2. Page Rank con matriz esparsa

Recordemos que elegimos el *Dictionary of Keys (dok)* para representar las matrices esparsas. El mismo consiste en un vector de diccionarios, en donde las posiciones del vector representan filas y las keys de los diccionarios las columnas. Los paso para realizar Page Rank con matriz esparsa son los siguientes:

1. Leemos los datos de entrada y luego generamos P.

```
Leer n, m del input
Crear un vector de diccionarios vacíos, P, de tamano n
Crear un vector tamano n, links_salientes, inicializado en 0
```

[3]http://en.cppreference.com/w/cpp/algorithm/sort

```
Para k=0 hasta m-1 hacer:
  Leer i,j del input
  Crear la key i-1 con 1 como valor asociado en el diccionario de
     P[j-1]
  Sumar 1 al contenido de links_salientes[i-1]
Fin Para
Para i=0 hasta n-1 hacer:
  Iterar en el diccionario de P[i]:
    Actualizar los valores asociados a las keys con el valor 1 /
       links_salientes[key]
  Fin Iteracion
Fin Para
```

Al igual que en la versión sin matriz esparsa utilizamos *links_salientes* para calcular los n_j necesarios para luego calcular los valores de P.

2. Calculamos el autovector asociado a 1 con el método de la Potencia con P, normalizando el autovector obtenido.

```
Definir una precision
Definir c como el factor de teletransportacion
Crear un vector, x, con valores aleatorios de 1 a 50 de tamanio n
Crear un vector, v, con 1/n como valores de tamanio n
Crear un vector, y, igual a x
delta = INFINITY
ultimo_delta = 0.
Hacer:
  x = y
  y = esparsaPorVector(P, x)
  y = escalarPorVector(y, c)
  w = normaUno(x) - normaUno(y)
  y = sumaVectores(y + escalarPorVector(w, v))
  ultimo_delta = delta
  delta = phi(x) / phi(y)
Mientras (delta - utimo_delta) > precision
Normalizar y
```

A continuación describimos el procedimiento auxiliar esparsaPorVector:

```
esparsaPorVector(matriz esparsa P, vector x):
  Crear vector, y, de tamano n
  Para i = 0 hasta n-1 hacer:
    suma = 0
    Itero sobre las keys del diccionario en P[i]:
      entero j = key
      suma = (valor de la key)*x[j]
    Fin de la Iteracion
    y[i] = suma
  Fin del Para
  Devolver y
```

El procedimiento phi es igual de la versión sin matriz esparsa.

Los procedimientos auxiliares escalarPorVector, sumaVectores y normaUno son procedimientos estándar, por lo cual no los detallamos.

Por ultimo Normalizar un vector consiste el realizar el mismo procedimiento descripto en la versión sin matriz esparsa.

3. A partir de este punto, los pasos son idénticos a los que realizamos en la versión sin matriz esparsa.

3.2. Scripts Rankings Deportivos

Para los dos modelos siendo estudiados, GeM y el sistema de puntaje de la AFA, decidimos implementarlos con scripts para MATLAB/OCTAVE, dado que lo que nos interesaba era analizar los resultados de los mismos y no analizar su tiempo de cómputo u otro atributo relacionado a su implementación.

3.2.1. GeM

El script posee varios parámetros de entrada que pasamos a describir a continuación:

- **in_filename:** Dirección de archivo con datos de entrada.
- **out_filename:** Dirección de archivo donde escribir el resultado final.
- **team_codes_filename:** Archivo opcional con el número de equipo asociado a su nombre correspondiente.
- **c:** Parámetro de amortiguación descrito en la sección 2.2.1 (Por defecto 0.85).
- **date_limit:** Campo opcional con la cantidad de fechas a tomar en cuenta (Por defecto toma todas las fechas disponibles).
- **precision:** Campo opcional con el nivel de precisión a utilizar en las comparaciones (Por defecto 0.0001).

Ahora explicaremos el código desarrollado:

1. Leemos la cantidad de equipos y partidos para después cargar todos los partidos disponibles y generar nuestra matriz A.

```
Cargar numero de partidos y equipos
Crear matriz A de tamanio equipos*equipos llena de 0s
Mientras partidos > 0
  Cargo numero de fecha del partido junto a los equipos y el
     resultado
  Si había limite de fecha y ya se cumplió
    partidos = 0
  Si no
    Si el primer equipo perdió
      A[numero primer equipo][numero segundo equipo] += diferencia
          de puntos
    Si el segundo equipo perdió
      A[numero segundo equipo][numero primer equipo] += diferencia
          de puntos
    Fin si
  Fin si
  Decremento partidos en uno
Fin mientras
```

2. Generamos la matriz H.

```
Crear matriz H de tamanio equipos*equipos llena de 0s
Crear vector a de tamanio equipos lleno de 0s
Para i de 1 a equipos
  sumaFila = suma total de elementos en fila i de A
  Si la suma > 0
```

```
    La fila i de H serán todos los elementos de la fila i de A
    cada uno dividido por sumaFila
  Si no
    Pongo un 1 en la posición i del vector a
  Fin si
Fin para
```

3. Generamos la matriz G.

```
Crear vector e de tamanio equipos lleno de 1s
Crear vector u para equipos invictos con todos elementos 1/equipos
Crear vector v para teletransportacion con todos elementos 1/
   equipos
Crear matriz G como resultado de hacer
G = c*[H + a*transponer(u)] + (1 - c)*e*transponer(v)
```

4. Busco el autovector asociado al autovalor $\lambda = 1$.

```
Generar matrices V y l de autovectores y autovalores de mi matriz
   G transpuesta
V es una matriz con autovectores como columnas
l es una matriz con autovalores en su diagonal
i = 1
Mientras valorAbsoluto(l[i][i] - 1) > precision
  Incremento i en una unidad
Fin mientras
Crear vector x correspondiente a la columna i de la matriz V
```

5. Normalizo el vector solución.

```
x = valorAbsolutoACadaElemento(x)/sumaElementos(
   valorAbsolutoACadaElemento(x))
```

6. Genero matriz de solución.

```
Crear matriz S de tamanio equipos*2 y llena de 0s
Para i de 1 a equipos
  S[i][1] = i
  S[i][2] = x[i]
Fin para
```

7. Ordeno y escribo la matriz solución.

```
Ordenar crecientemente por segundo elemento de filas la matriz S (
   ranking)
Si tengo archivo con nombres de equipos
  Creo mapa teamcodes con cada numero de equipo asociado a su
     nombre
Fin si

Si tengo nombres de equipos cargados
  Para i de 0 a equipos - 1
    Escribo numero, nombre y ranking del equipo numero equipos - i
  Fin para
Si no
  Para i de 0 a equipos - 1
    Escribo numero y ranking del equipo numero equipos - i
```

```
  Fin para
Fin si
```

3.2.2. Puntaje AFA

Para el script utilizado para calcular el puntaje según el criterio de la AFA los parámetros son los siguientes:

- **in_filename:** Dirección de archivo con datos de entrada.
- **out_filename:** Dirección de archivo donde escribir el resultado final.
- **team_codes_filename:** Archivo opcional con el número de equipo asociado a su nombre correspondiente.
- **date_limit:** Campo opcional con la cantidad de fechas a tomar en cuenta (Por defecto toma todas las fechas disponibles).

Pasamos a describir el código:

1. Leemos la cantidad de equipos y partidos para después cargar todos los partidos disponibles y generar nuestra matriz S.

```
Cargar numero de partidos y equipos
Crear matriz S de tamanio equipos*2 llena de 0s
Mientras partidos > 0
  Cargo numero de fecha del partido junto a los equipos y el
     resultado
  Si había limite de fecha y ya se cumplió
    partidos = 0
  Si no
    Si hubo empate
      S[numero primer equipo] += 1
      S[numero segundo equipo] += 1
    Si gano el primer equipo
      S[numero primer equipo] += 3
    Si gano el segundo equipo
      S[numero segundo equipo] += 3
    Fin si
  Fin si
  Decremento partidos en uno
Fin mientras
```

2. Ordeno y escribo la matriz solución.

```
Ordenar crecientemente por segundo elemento de filas la matriz S (
   ranking)
Si tengo archivo con nombres de equipos
  Creo mapa teamcodes con cada numero de equipo asociado a su
     nombre
Fin si

Si tengo nombres de equipos cargados
  Para i de 0 a equipos - 1
    Escribo numero, nombre y ranking del equipo numero equipos - i
  Fin para
Si no
```

```
  Para i de 0 a equipos - 1
    Escribo numero y ranking del equipo numero equipos - i
  Fin para
Fin si
```

4. Experimentación

En esta sección, se detallan los diferentes experimentos que realizamos para medir la eficiencia y calidad de resultados de los algoritmos implementados.

4.1. Page Rank

4.1.1. Instancias de prueba

Las instancias de prueba utilizadas para la experimentación pueden dividirse en tres grupos:

- Instancias medianas-grandes, que fueron utilizadas para medir tiempo y la convergencia del algoritmo Page Rank. Estas instancias se obtuvieron de la base de datos de SNAP[2] y se detallan a continuación:

Nombre	Nodos	Ejes	Descripción
p2p-Gnutella08	6,301	20,777	Gnutella peer to peer network from August 8 2002
p2p-Gnutella04	10,876	39,994	Gnutella peer to peer network from August 4 2002
p2p-Gnutella30	36,682	88,328	Gnutella peer to peer network from August 30 2002
p2p-Gnutella31	62,586	147,892	Gnutella peer to peer network from August 31 2002
web-Stanford	281,903	2,312,497	Web graph of Stanford.edu

Cuadro 1: Recursos utilizados de SNAP

- Instancias chicas, que fueron utilizadas para analizar la calidad de los resultados de PageRank. Estas instancias fueron generadas utilizando las herramientas provistas por la cátedra y otras extra generadas por nosotros. Los detalles particulares se desarrollan en la sección **Calidad de los resultados**.
- Instancias aleatorias, generadas mediante nuestros propios algoritmos. Estas son detalladas en la sección que analiza los **Tiempos de Ejecución**.

4.1.2. Convergencia del algoritmo

Realizamos una serie de experimentos con el propósito de analizar la convergencia de la solución provista por el algoritmo a través de las iteraciones que el mismo realiza en su ciclo principal.
Para hacer esto, modificamos el algoritmo (no su procedimiento) con el fin de ir guardando las soluciones temporales en cada paso.
Con estos datos, medimos la Norma Manhattan (L1) entre las soluciones de cada par de iteraciones sucesivas, definida como:

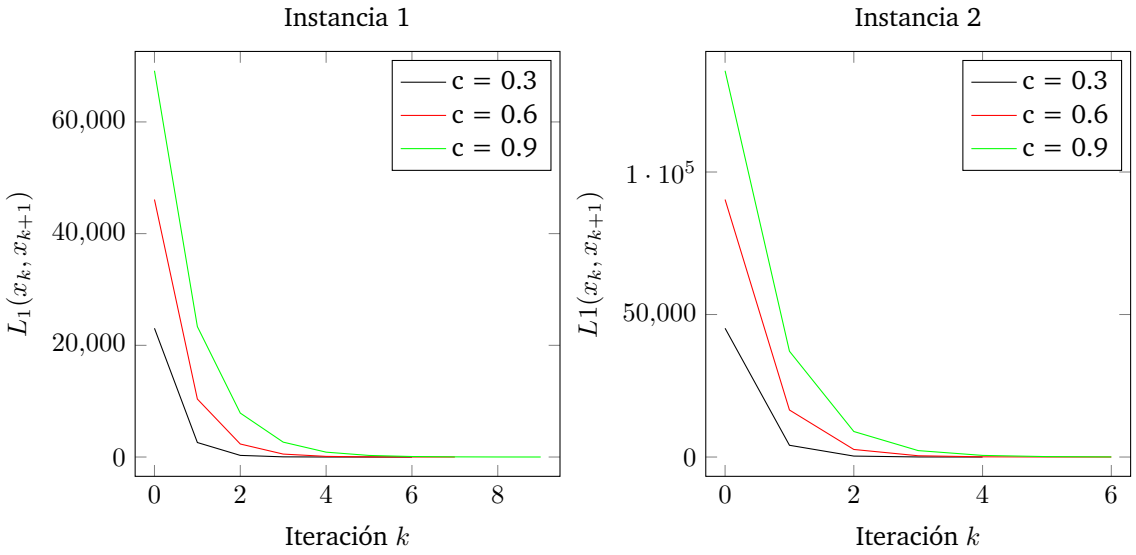
$$L_1(x, y) = \sum_{i=1}^{n} |x[i] - y[i]|$$

Siendo n la cantidad de nodos del grafo siendo estudiado, $x \in \mathbb{R}^n$ la solución del algoritmo en una iteración k e $y \in \mathbb{R}^n$ la de la iteración $k + 1$.
Esta es una forma intuitiva y lógica de estudiar la convergencia, ya que al tomar la suma de los valores absolutos entre los componentes de cada vector, lo que realmente estamos haciendo es calcular cuanto varia la solución que estamos calculando a través de las iteraciones. Además, podemos asegurar que por el procedimiento de nuestro algoritmo esta diferencia va a ir disminuyendo, y cuando sea lo suficientemente chica el algoritmo va a terminar.

Estudiamos la convergencia a través de dos instancias de tamaño mediano-grande obtenidos de SNAP, mas precisamente:

- Instancia 1: 6301 nodos y 20777 ejes.
- Instancia 2: 10876 nodos y 39994 ejes.

Por lo que podemos decir que son considerables y relativamente esparsas (por ejemplo, el primero podría tener cerca de 40 millones de ejes si fuera completo, teniendo en cuenta que es un grafo dirigido).
A su vez, fuimos variando el componente c del algoritmo Page Rank entre 0.3, 0.6 y 0.9, el cual controla la importancia del *navegante aleatorio*, es decir, a menor c aumenta la probabilidad de que el usuario vaya a una página aleatoria desde la actual.
Veamos los resultados:



Siendo $L1(x_k, x_{k+1})$ (el eje y del gráfico) la distancia Manhattan entre los autovectores generados por el algoritmo entre las iteraciones k e $k+1$.
Como podemos ver el algoritmo converge rápidamente, tomando a lo sumo 9 iteraciones para terminar de procesar instancias de datos relativamente grandes. Los resultados son parecidos para ambas instancias.
Adicionalmente, vemos que Page Rank converge de forma más rápida para valores de c más chicos, es decir, casos en los cuales es más probable que el usuario vaya a una página cualquiera desde la actual. Cuando esto sucede, la matriz de transición tiene valores más parejos dentro de cada columna y podemos decir entonces que el algoritmo soluciona el problema más rápido porque el sistema es más estable.

4.1.3. Tiempo de ejecución

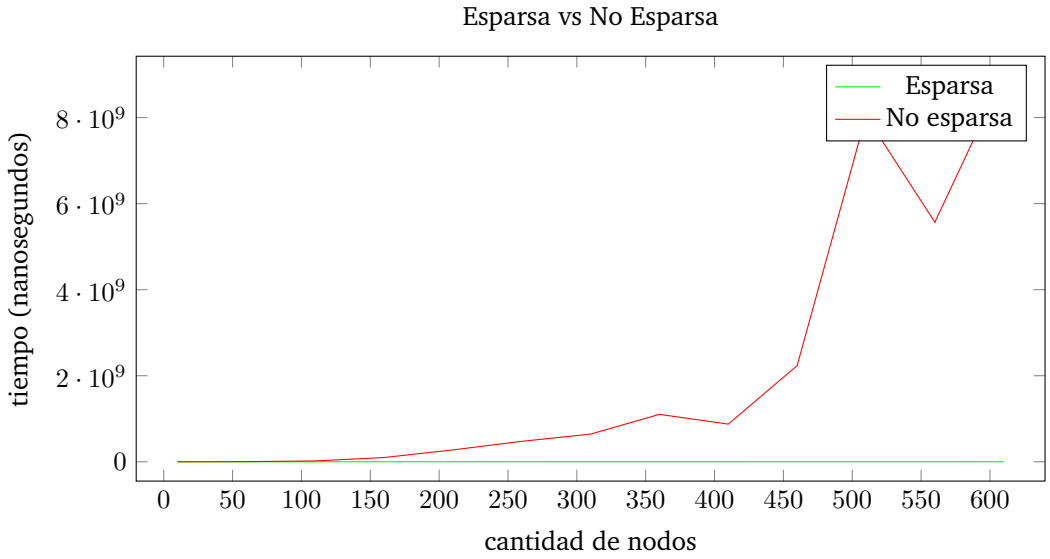
En esta sección, analizamos el tiempo de cómputo que emplea Page Rank.
Para tomar los tiempos, utilizamos la librería chronos de C++ y pasamos las mediciones a nanosegundos. Todos las medidas fueron hechas bajo las mismas condiciones (procesos abiertos, computadora, alimentación y nivel de optimizaciones del compilador).
Los experimentos pasan por tres variables: la cantidad de nodos y la cantidad de ejes del grafo siendo analizado, y la precisión utilizada en Page Rank (es decir, la diferencia que tomamos como suficiente entre dos iteraciones para dar un resultado en el método de la potencia).
Además, comparamos la versión 'esparsa' del algoritmo Page Rank (donde la matriz se guarda de forma diferente) con la versión clásica del mismo algoritmo. En este caso, la precisión del algoritmo esta fija y se va variando la cantidad de vértices.
Para los demás experimentos, cabe destacar que utilizamos la versión esparsa del algoritmo ya que por la cantidad de nodos y ejes, la versión normal es muy lenta para evaluar.
Con este fin, generamos instancias de forma dinámica y aleatoria, es decir que dada una cantidad de nodos, asignamos los ejes aleatoriamente hasta llegar a cierta cantidad a través de la librería Random de C++.
Más precisamente, los experimentos consisten en:

- Una comparación entre el algoritmo para matrices esparsas y Page Rank normal.
- Nodos del grafo: variar la cantidad de nodos entre 1000 y 100,000 con tantos ejes como 2 veces la cantidad de nodos en cuestión.

- Ejes del grafo: dado un grafo con 3000 nodos, vamos variando la cantidad de ejes entre 1000 y 8,000,000 (el grafo completo tendría 8,997,000 ejes).
- Precisión: teniendo un grafo de 3000 nodos y 1,000,000 ejes, variamos la precisión entre 0.1 y 0.000000000000001 (dividiéndola por 10 entre cada iteración).

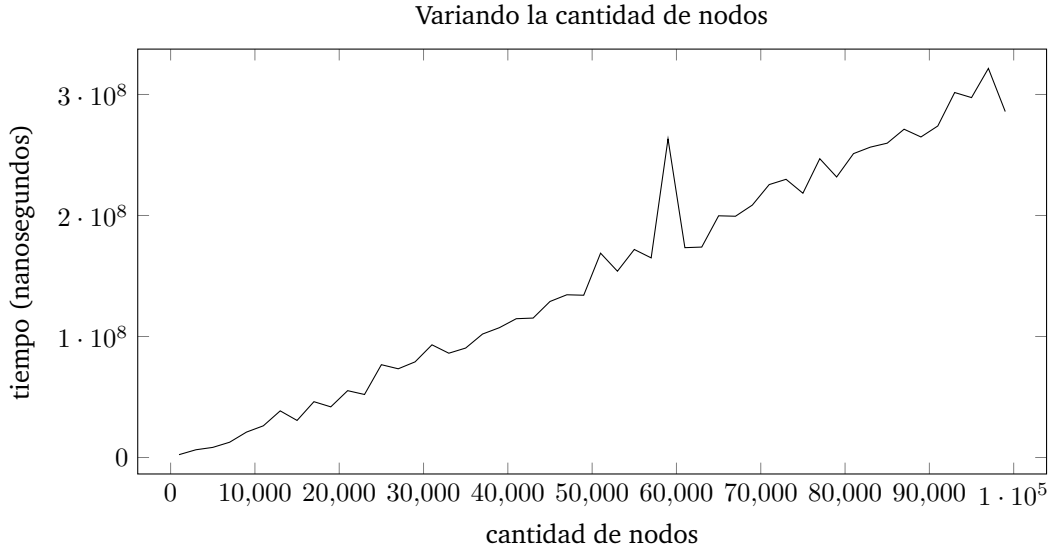
El factor c de Page Rank fue configurado en 0.85, ya que no afecta el tiempo de cómputo.

Los resultados de los experimentos fueron los siguientes:

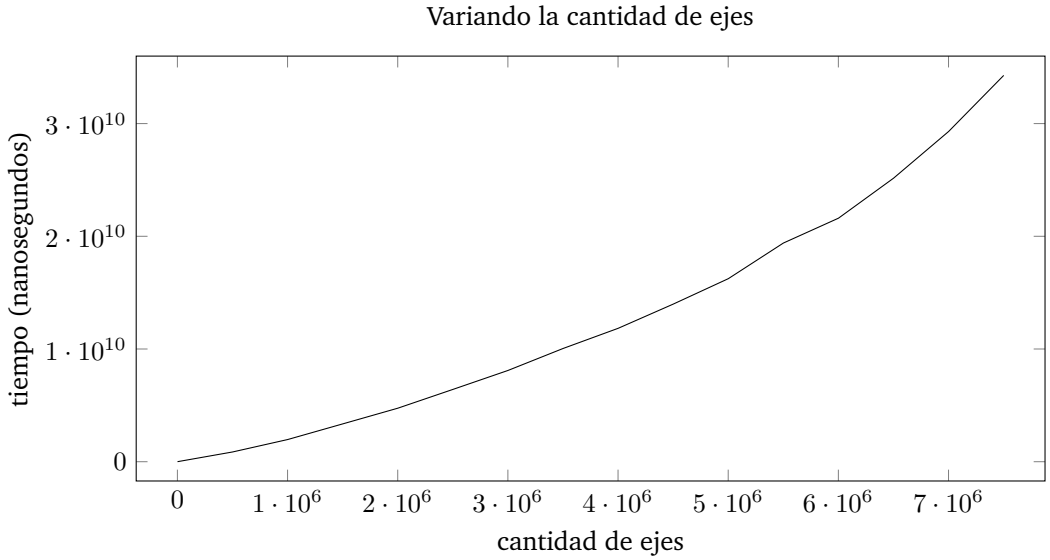


Como podemos ver, la versión esparsa del algoritmo optimiza el tiempo de cómputo de forma significativa. Por este motivo, para los demás experimentos vamos a utilizar esta versión del algoritmo.

En el siguiente experimento podemos ver como varían los tiempos según la cantidad de nodos del grafo:



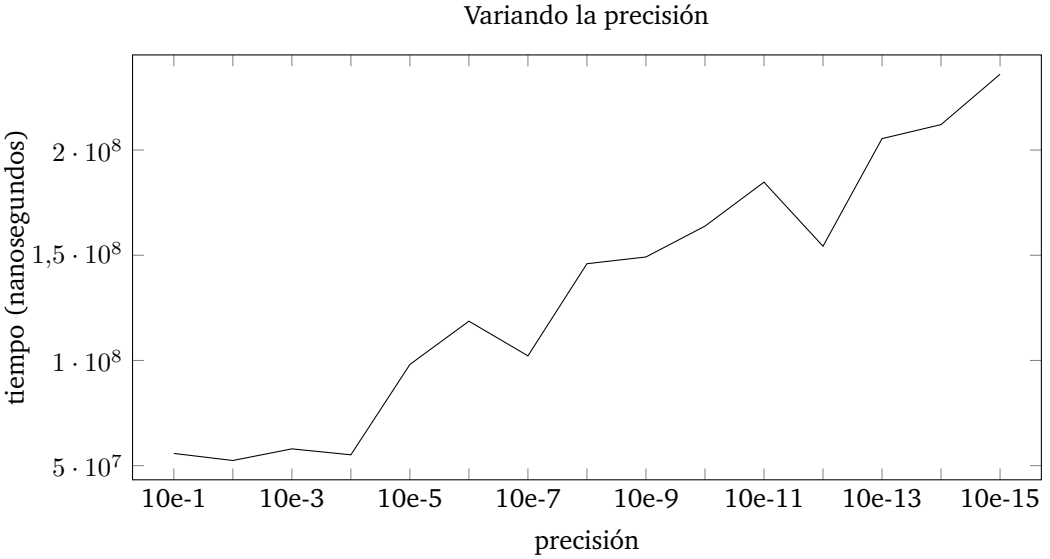
Como vemos, la cantidad de nodos del grafo afecta la rapidez del algoritmo Page Rank. A simple vista, se nota que la curva que sigue el eje 'y' del gráfico tiene pinta de superior a lineal, aun así nos limitamos a decir que es polinomial. Mientras que vamos aumentando la cantidad de nodos, los ejes empleados pasa a ser el doble de la cantidad de nodos en la iteración. Entonces la duda que queda en cuestión es saber si es la cantidad de ejes el factor clave que hace escalar los tiempos o la cantidad de nodos, lo que analizamos a continuación:



En este caso, la curva es más pronunciada. Esto es un fuerte indicador de que la cantidad de ejes afecta la rapidez del algoritmo de manera más critica que la cantidad de nodos empleados. Aun así, la curva que sigue el gráfico parece ser polinomial.

Esto resultados tienen sentido ya que por las operaciones del algoritmo, sabemos que la mayor parte de la complejidad reside en generar la matriz de transición, lo cual toma una complejidad cúbica derivada de multiplicar matrices (además de sumarlas y escalarlas, lo cuál es cuadrático).
Adicionalmente, en los tests de convergencia, el algoritmo no toma muchas iteraciones en determinar el autovector que queremos como respuesta, por lo que restaría ver como afecta al algoritmo el tiempo que se toma en usar el método de la potencia.

A continuación vemos como la precisión empleada afecta los tiempos del algoritmo:



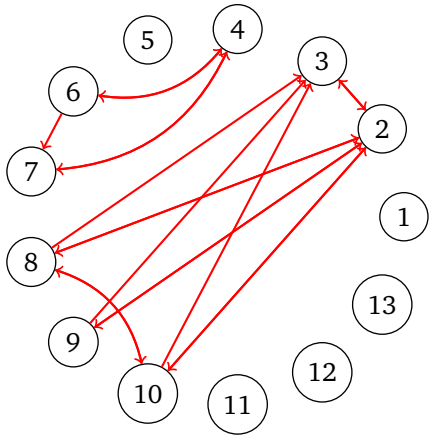
En base a esto, concluimos que la precisión también es un factor considerable al analizar la eficiencia de nuestro algoritmo. Esto era predecible porque cuanta más precisión es requerida en el cálculo del autovector generado por Page Rank, más iteraciones deberá tomar el algoritmo para dar una respuesta.

4.1.4. Calidad de los resultados

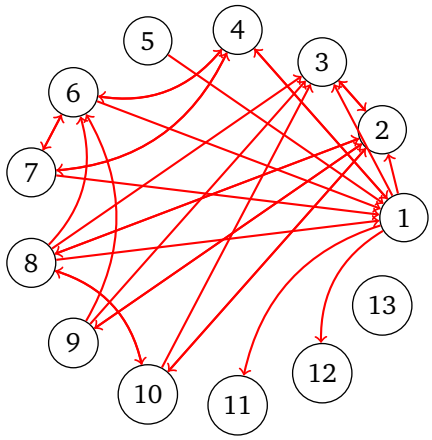
Para evaluar la calidad de las soluciones provistas por Page Rank las comparamos contra el algoritmo InDeg, que ordena las páginas según la cantidad de links que van hacia ellas.

Por el lado de Page Rank, no utilizamos la versión Esparsa del algoritmo, ya que el tamaño de la entrada no lo requiere.
Las instancias que usamos para hacer estas comparaciones fueron:

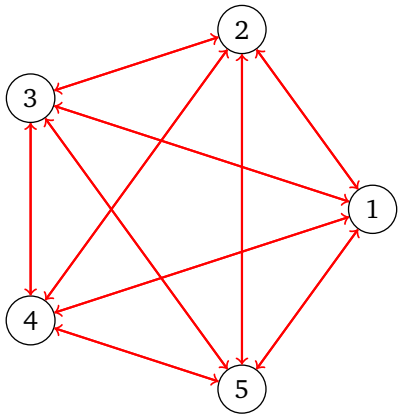
- Instancia 1: 13 nodos y 18 ejes.



- Instancia 2: 13 nodos y 31 ejes.



- Instancia 3: 5 nodos y 20 ejes (grafo completo).



De forma de poder analizar los resultados fácilmente, nos limitamos a instancias chicas.

Para la instancia 1, el valor c del algoritmo Page Rank fue configurado en 0.3, 0.6 y luego en 0.9 con el fin de introducir la posibilidad de que el navegante salte a otra página de forma aleatoria y ver cuánto afecta al resultado final. Esto no es así para la instancia 3 ya que al ser un grafo donde la probabilidad

de ir a las demás páginas es igual para todas, el concepto de navegante aleatorio es irrelevante porque es igual de probable que vaya a cualquier página desde el principio.

Los resultados fueron:

Instancia 1

Page Rank(c = 0.3)

Ranking	Página	Importancia
0	2	$8{,}95 \cdot 10^{-2}$
1	4	$8{,}38 \cdot 10^{-2}$
2	8	$8{,}01 \cdot 10^{-2}$
3	10	$8{,}01 \cdot 10^{-2}$
4	6	$8 \cdot 10^{-2}$
5	7	$7{,}62 \cdot 10^{-2}$
6	9	$7{,}61 \cdot 10^{-2}$
7	3	$7{,}42 \cdot 10^{-2}$
8	0	$7{,}2 \cdot 10^{-2}$
9	1	$7{,}2 \cdot 10^{-2}$
10	5	$7{,}2 \cdot 10^{-2}$
11	11	$7{,}2 \cdot 10^{-2}$
12	12	$7{,}2 \cdot 10^{-2}$

Page Rank(c = 0.6)

Ranking	Página	Importancia
0	2	0,14
1	4	0,11
2	10	$9{,}8 \cdot 10^{-2}$
3	8	$9{,}8 \cdot 10^{-2}$
4	6	$9{,}52 \cdot 10^{-2}$
5	7	$7{,}62 \cdot 10^{-2}$
6	9	$7{,}35 \cdot 10^{-2}$
7	3	$6{,}53 \cdot 10^{-2}$
8	0	$4{,}76 \cdot 10^{-2}$
9	1	$4{,}76 \cdot 10^{-2}$
10	5	$4{,}76 \cdot 10^{-2}$
11	11	$4{,}76 \cdot 10^{-2}$
12	12	$4{,}76 \cdot 10^{-2}$

Page Rank(c = 0.9)

Ranking	Página	Importancia
0	2	0,2
1	4	0,15
2	8	0,13
3	10	0,13
4	6	0,12
5	7	$8{,}11 \cdot 10^{-2}$
6	9	$7{,}05 \cdot 10^{-2}$
7	3	$5{,}76 \cdot 10^{-2}$
8	0	$1{,}18 \cdot 10^{-2}$
9	1	$1{,}18 \cdot 10^{-2}$
10	5	$1{,}18 \cdot 10^{-2}$
11	11	$1{,}18 \cdot 10^{-2}$
12	12	$1{,}18 \cdot 10^{-2}$

InDeg

Ranking	Página	In Degree
0	2	4
1	8	3
2	10	3
3	4	2
4	6	2
5	9	2
6	3	1
7	7	1
8	0	0
9	1	0
10	5	0
11	11	0
12	12	0

Por un lado, Page Rank obtiene un ranking con valores lógicos en el sentido de que el nodo con mas ejes hacia él es el de mayor importancia y los nodos con cantidad de links parecidos reciben un ranking similar.

Como podemos ver, los resultados de Page Rank no varían mucho, sus valores son afectados pero los rankings varían muy levemente. Esto es debido a que la cantidad de ejes no es de un tamaño tan grande como para que la variación de c afecte el ranking.

A su vez, InDeg da los resultados que esperábamos, ordenando según la cantidad de links entrantes.

En comparación, los algoritmos dan resultados parecidos de a secciones pero sí tiene diferencias en los primeros puestos. Esto se debe a que Page Rank detecta que por las probabilidades del recorrido que un navegante puede hacer, no necesariamente las páginas con mas links entrantes van a ser las más visitadas a futuro.

Aun así, veamos otro ejemplo donde la cantidad de ejes sea un poco mayor:

Instancia 2

Page Rank(c = 0.3)

Ranking	Página	Importancia
0	1	$9{,}02 \cdot 10^{-2}$
1	2	$8{,}73 \cdot 10^{-2}$
2	8	$8{,}12 \cdot 10^{-2}$
3	6	$7{,}86 \cdot 10^{-2}$
4	4	$7{,}79 \cdot 10^{-2}$
5	10	$7{,}76 \cdot 10^{-2}$
6	7	$7{,}67 \cdot 10^{-2}$
7	9	$7{,}55 \cdot 10^{-2}$
8	5	$7{,}21 \cdot 10^{-2}$
9	3	$7{,}21 \cdot 10^{-2}$
10	0	$7{,}03 \cdot 10^{-2}$
11	11	$7{,}03 \cdot 10^{-2}$
12	12	$7{,}03 \cdot 10^{-2}$

Page Rank(c = 0.6)

Ranking	Página	Importancia
0	2	0,13
1	1	0,12
2	8	0,11
3	6	$8{,}77 \cdot 10^{-2}$
4	10	$8{,}72 \cdot 10^{-2}$
5	4	$8{,}46 \cdot 10^{-2}$
6	7	$7{,}89 \cdot 10^{-2}$
7	9	$7{,}18 \cdot 10^{-2}$
8	3	$5{,}53 \cdot 10^{-2}$
9	5	$5{,}39 \cdot 10^{-2}$
10	0	$4{,}22 \cdot 10^{-2}$
11	11	$4{,}22 \cdot 10^{-2}$
12	12	$4{,}22 \cdot 10^{-2}$

Page Rank(c = 0.9)

Ranking	Página	Importancia
0	2	0,21
1	8	0,15
2	10	0,12
3	1	$9{,}94 \cdot 10^{-2}$
4	6	$8{,}73 \cdot 10^{-2}$
5	4	$8{,}22 \cdot 10^{-2}$
6	9	$7{,}56 \cdot 10^{-2}$
7	7	$7{,}36 \cdot 10^{-2}$
8	3	$4{,}71 \cdot 10^{-2}$
9	5	$2{,}83 \cdot 10^{-2}$
10	0	$1{,}04 \cdot 10^{-2}$
11	11	$1{,}04 \cdot 10^{-2}$
12	12	$1{,}04 \cdot 10^{-2}$

InDeg

Ranking	Página	In Degree
0	1	5
1	8	5
2	2	4
3	4	3
4	6	3
5	7	3
6	9	3
7	10	3
8	3	1
9	5	1
10	0	0
11	11	0
12	12	0

En este caso ya podemos ver que el ranking varía más. Esto se debe a que ahora la cantidad de ejes es mayor con respecto a la cantidad de vértices. Por lo tanto, cuando vamos variando c, el algoritmo devuelve diferentes respuestas porque ahora la distribución de los links cobra más relevancia a comparación del factor ”navegante aleatorio”.
A diferencia del caso anterior, los resultados ya no son tan parecidos a los del algoritmo InDeg. De hecho, en solo un caso coincide PageRank con InDeg en el primer puesto del ranking. Lo que sugiere que la distribución de los ejes y su cantidad afectan la impredecibilidad del resultado.
Además, los ejes están distribuidos de tal manera que no haya grupos grandes aislados o grafos completos donde un grupo de vértices tenga una distribución de probabilidad muy pareja y nada de relación con otras componentes conexas.
Para ilustrar este caso veamos el siguiente ejemplo de un grafo completo:

Instancia 3

Page Rank

Ranking	Página	Importancia
0	0	0,2
1	1	0,2
2	2	0,2
3	3	0,2
4	4	0,2

InDeg

Ranking	Página	In Degree
0	0	4
1	1	4
2	2	4
3	3	4
4	4	4

Los resultados indican lo lógico, como es un grafo donde la probabilidad de ir a cualquier lado es igual, tanto Page Rank como InDeg resuelven el problema asignándole igual importancia a cada página. Por lo tanto concluimos que tanto la cantidad de ejes como la distribución son los factores que más contribuyen a los resultados de Page Rank.

4.2. GeM

4.2.1. Instancias de prueba

La instancia de prueba utilizada para la experimentación consistió en tomar los resultados de las primeras 26 fechas del Torneo de Primera División Argentino, con la salvedad los siguientes resultados para los partidos suspendidos: Fecha 19 - Defensa y Justicia 0:1 River Plate y Fecha 22 - Godoy Cruz 0:1 Racing Club

4.2.2. Variando el parámetro c

Este experimento consiste en variar el parámetro c y analizar como impacta esta variación en los resultados obtenidos al generar el ranking con GeM.

Recordemos que c es el coeficiente de amortiguación, que regula que tanto afecta el *navegante aleatorio* al resultado final.

Como es intuitivo de pensar, nuestra hipótesis en este experimento es que al tender c a cero aumenta la influencia del *navegante aleatorio* en el puntaje de cada equipo y, por lo tanto, más se parecen los puntajes de todas los equipos. Es decir, los resultados anteriores entre de los distintas equipos dejan de importar.

Por otro lado, cuando c tiende a uno se debilita la influencia del *navegante aleatorio*, causando que el puntaje dependa solo de la matriz $H + au^t$, descrita en la sección 2.2.1.

Se muestran a continuación dichos resultados para diferentes valores de c:

Posición	Equipo	Puntaje
1	Vélez Sarsfield	0.033333
2	Unión	0.033333
3	Gimnasia y Esgrima (LP)	0.033333
4	Estudiantes (LP)	0.033333
5	Defensa y Justicia	0.033333
6	Crucero del Norte	0.033333
7	Colón	0.033333
⋮	⋮	⋮
24	Lanús	0.033333
25	San Lorenzo	0.033333
26	Rosario Central	0.033333
27	River Plate	0.033333
28	Racing Club	0.033333
29	Quilmes	0.033333
30	Olimpo	0.033333

Posición	Equipo	Puntaje
1	Boca Juniors	0.051301
2	San Lorenzo	0.044563
3	River Plate	0.044200
4	Racing Club	0.040067
5	Aldosivi	0.039451
6	Rosario Central	0.039114
7	Quilmes	0.036699
⋮	⋮	⋮
24	Godoy Cruz	0.028356
25	Argentinos Juniors	0.028208
26	Temperley	0.027904
27	Crucero del Norte	0.027353
28	Nueva Chicago	0.026346
29	Atlético de Rafaela	0.025776
30	Colón	0.025577

Cuadro 2: A izquierda: puntajes obtenidos con $c = 0$, a derecha: puntajes obtenidos con $c = 0{,}3$

Posición	Equipo	Puntaje
1	Boca Juniors	0.086019
2	Aldosivi	0.065353
3	River Plate	0.063500
4	San Lorenzo	0.062035
5	Rosario Central	0.048473
6	Racing Club	0.047878
7	San Martín (SJ)	0.043956
⋮	⋮	⋮
24	Godoy Cruz	0.017516
25	Temperley	0.016045
26	Crucero del Norte	0.016039
27	Argentinos Juniors	0.015398
28	Nueva Chicago	0.014232
29	Atlético de Rafaela	0.011388
30	Colón	0.010276

Posición	Equipo	Puntaje
1	Boca Juniors	0.095290
2	Aldosivi	0.075151
3	River Plate	0.068142
4	San Lorenzo	0.065265
5	Rosario Central	0.050875
6	Racing Club	0.048824
7	San Martín (SJ)	0.047118
⋮	⋮	⋮
24	Godoy Cruz	0.014148
25	Crucero del Norte	0.013132
26	Temperley	0.012487
27	Argentinos Juniors	0.011286
28	Nueva Chicago	0.011239
29	Atlético de Rafaela	0.007559
30	Colón	0.006003

Cuadro 3: A izquierda: puntajes obtenidos con $c = 0{,}85$, a derecha: puntajes obtenidos con $c = 1$

Analizando los resultados tenemos que:

- Para $c = 0$, la tabla de puntajes se condice con la hipótesis planteada. Los puntajes de los equipos no solo son parecidos, sino que son iguales. Y dicho puntaje es $0{,}0\hat{3} = 1/30 = 1/n$, dónde n es la cantidad de equipos totales.
- A medida que el c aumenta (desde $0{,}3$ a 1) las posiciones van convergiendo. Ya con $c = 0{,}3$, 6 de los 7 primeros equipos aparecen en los primeros 7 puestos con $c = 1$, y los últimos 7 equipos con $c = 0{,}3$ aparecen en los últimos 7 puestos con $c = 1$.
- A su vez, la diferencia entre los resultados con $c = 85$ y $c = 1$ es muy chica: de los 14 equipos mostrados en el Cuadro 3 solo dos de ellos cambian de posición (Crucero del Norte y Temperley).

En base a lo analizado, podemos concluir que los resultados del experimento corroboran las hipótesis planteadas.

4.2.3. Evolución del ranking

Una de las características que nos interesa estudiar con estos modelos de ranking es ver cómo evolucionan a lo largo del tiempo ya que es importante analizar los casos donde se producen cambios abruptos, para que no exista ninguna anomalía en el resultado final. Es por esto que en el siguiente experimento lo que hicimos es tomar los dos modelos descritos en la sección 2.2 y estudiar en paralelo su evolución a lo largo del tiempo.

Para esto tomamos los equipos que terminaron en los primeros 8 lugares según el modelo GeM utilizando un $c = 0{,}85$ (Cuadro 4).

La idea es no sólo ver cómo fueron modificándose las posiciones de los participantes si no además comparar entre GeM y el sistema establecido por AFA. Nuestra intuición es que probablemente veamos resultados que a primera vista resulten inesperados en el ranking provisto por GeM, ya que el hecho de brindarle pesos a cada equipo produce que no necesariamente el participante con más partidos ganados esté arriba en la tabla.

A continuación pasamos a mostrar los gráficos reflejando el comportamiento de ambos modelos. Cabe destacar que el sistema de puntaje de la AFA no trabaja asignando a cada equipo un peso dónde la suma total da 1. Para subsanar esto, y poder comparar los dos métodos, vamos a definir el puntaje de un equipo en una fecha particular como el porcentaje de sus puntos sobre el total de puntos en dicha fecha.

Ejemplo: Si Boca Juniors acumuló un total de 58 puntos pero el total de puntos para esa fecha es 1044, entonces el puntaje "normalizado" será de 0.0556.

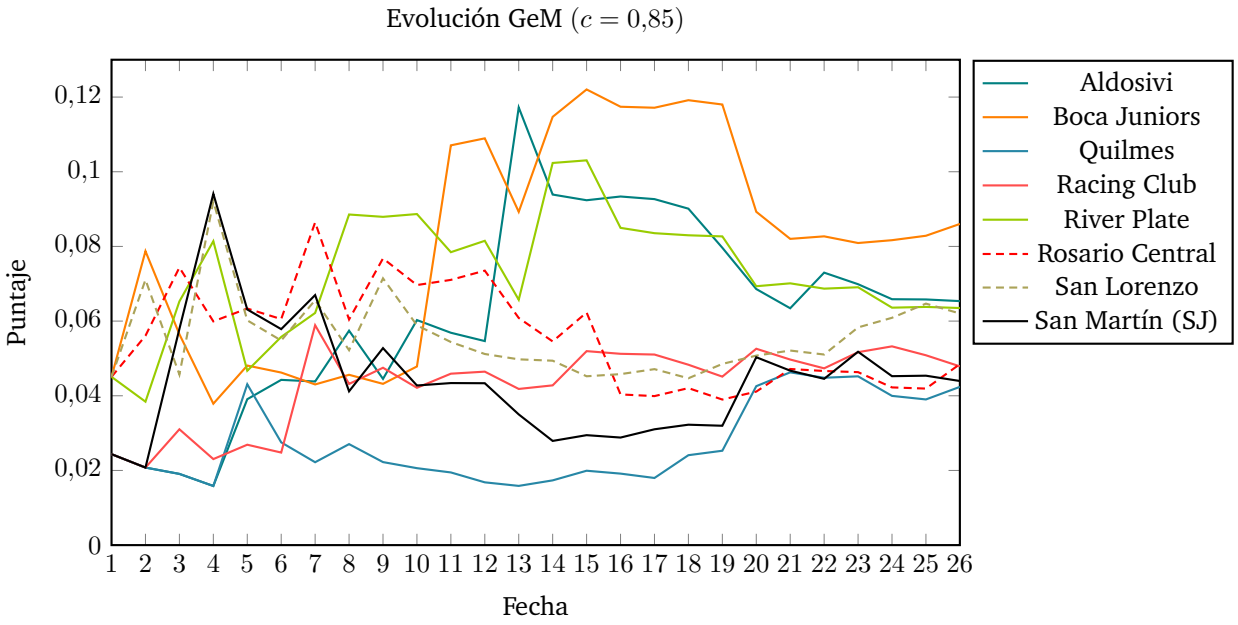


Figura 6: Evolución de los equipos que terminaron en los primeros 8 lugares según GeM.

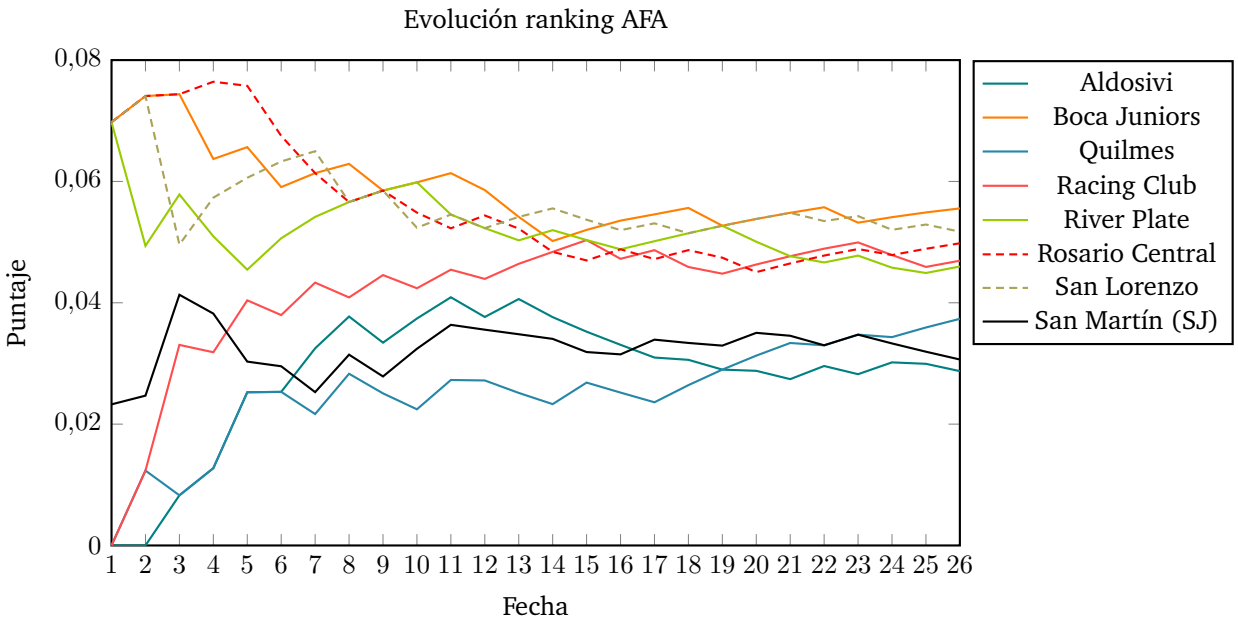


Figura 7: Evolución de los mismos 8 equipos en el modelo que utiliza la AFA.

Tenemos varios puntos para observar en los gráficos generados. Para empezar se puede ver cómo en el ranking que generó GeM hay mucha más variación que en el provisto por AFA, donde los cambios son mucho más graduales. Esto se debe a que el sistema de la AFA tiene un tope para el puntaje que cada participante puede ganar por fecha que serían los 3 puntos de ganar su partido correspondiente. Sin embargo, con GeM basta que un equipo derrote a otro de mayor peso para catapultarlo en la tabla de posiciones.

Otro detalle que podemos notar, es en el resultado final cómo quedó el orden para la AFA con respecto a GeM (Cuadro 4). Acá tenemos que por un lado 5 de los 8 primeros equipos según GeM figuran dentro de los primeros 8 para el puntaje de AFA. Esto en cierta forma nos indica que el resultado de GeM no es disparatado, incluso el orden de estos 5 equipos se mantiene bastante parecido. Sin embargo, notamos que existen algunas irregularidades, donde la que nos va interesar es la de Aldosivi.

Posición	Equipo	Puntaje
1	Boca Juniors	0.086019
2	Aldosivi	0.065353
3	River Plate	0.063500
4	San Lorenzo	0.062035
5	Rosario Central	0.048473
6	Racing Club	0.047878
7	San Martín (SJ)	0.043956
8	Quilmes	0.042382

Posición	Equipo	Puntaje
1	Boca Juniors	0.055556
2	San Lorenzo	0.051723
3	Rosario Central	0.049808
4	Racing Club	0.046935
5	River Plate	0.045977
11	Quilmes	0.037356
16	San Martín (SJ)	0.030651
18	Aldosivi	0.028736

Cuadro 4: A izquierda, primeros 8 lugares según GeM con $c = 0{,}85$. A derecha la posición de los mismos equipos según el ranking de AFA.

Aldosivi figura como segundo lugar en la primer tabla, mientras que para el ranking estándar está 18º. Si observamos en la Figura 6, Aldosivi pega un salto en la fecha 13, llegando incluso a estar por encima de Boca Juniors, pero luego se va estabilizando a lo largo de las fechas hasta quedar en su respectiva posición. Esta fecha da la casualidad de que Aldosivi jugó contra Boca Juniors, ganándole 3-0. Si observamos, en la fecha 12, Boca se encontraba primero, llevándole una diferencia apreciable en puntaje al segundo lugar, River Plate, por lo tanto como Aldosivi venció al puntero, sus goles tenían el mayor peso posible para una victoria, explicando su repentina escalada en la tabla.

Por último, podemos analizar también el salto de Boca de la fecha 10 a la 11. Como era de esperarse, en la fecha 11, Boca jugó contra River, ganando 2-0. Nuevamente se ve como en la fecha 10 River se encontraba por encima del resto, por lo cual era el equipo que más puntos generaba al ganarle.

Mediante este experimento, llegamos a observar cómo se comportaban ambos modelos a lo largo del tiempo. Por un lado, en el método GeM tenemos los cambios abruptos en la tabla de posiciones que son consecuencia de su característica principal, el puntaje por pesos. Por otra parte, con el estándar de la AFA el avance de los equipos está limitado y por ende resulta mucho más controlado. Estas son características claves de cada sistema ya que a la hora de decidir qué modelo utilizar es necesario tener en cuenta los efectos secundarios de los métodos aplicados. Es aquí donde el concepto de ranking “justo” depende de lo que consideren más importante los organizadores. Si se cree que todo equipo posee exactamente el mismo nivel, entonces el ranking estándar ofrecerá un resultado simple, controlado donde con esa ideología ganará el mejor. Sin embargo, si uno está dispuesto a sacrificar el entorno controlado a cuestas de un sistema donde se considere que no todo participante tenga las mismas posibilidades de ganar, GeM brindará un ranking considerando las diferencias entre ellos.

5. Conclusión

En este trabajo pudimos no solo modelar el problema planteado, sino apreciar y aprovechar las propiedades del mismo para así resolverlo con los métodos estudiados observando también las características de ellos.

Así mismo, cabe destacar que al realizar operaciones con aritmética finita, tanto para la solución de los sistemas como para el cálculo de los autovectores donde la reutilización de datos arrastra error, no podemos garantizar que los resultados obtenidos sean exactos, pero dado que realizamos varias instancias de prueba con distintas metodologías, pudimos ver que los valores que obtuvimos eran coherentes a su contexto.

Por un lado mediante la forma en la que construimos nuestro sistema probamos que Page Rank era un método factible para asignar puntajes a un listado de páginas web. Además produjimos una versión mejorada del algoritmo para matrices esparzas y así redujimos drásticamente la cantidad de operaciones necesarias para resolverla así como la complejidad espacial.

Mediante nuestros experimentos, corroboramos que los resultados de Page Rank varían no solamente según los parámetros pasados al algoritmo (c y precisión), sino por las características mismas del grafo. Más precisamente, la cantidad y distribución de los ejes es lo que más cambia el autovector respuesta. Esto nos da una idea de la gran utilidad que tiene Page Rank porque a diferencia de otros algoritmos como InDeg, donde el resultado es totalmente predecible, Page Rank considera las particularidades y la esparcidad del grafo de links con el método de la potencia. Por ejemplo, cuando una página importante linkea a otra página distinta, esta ultima se ve beneficiada en su propio puntaje de mayor manera que si la hubiese referenciado una página poco importante. De esta manera, el algoritmo logra predecir de una forma mucho mas rica, en cuales sitios web es más probable que el navegante esté a lo largo del tiempo.

A su vez, mediante una implementación del algoritmo GeM, pudimos ver como el concepto de Page Rank va más allá de rankear páginas web. Es decir, el modelo en grafos que plantea el algoritmo es extensible a diversas aplicaciones y en este caso lo pudimos aplicar para ligas deportivas.

Si bien tuvimos que cambiar los datos de entrada para producir un grafo con pesos que represente la situación de manera más correcta, con esta implementación logramos ver como Page Rank determina resultados lógicos aún en contexto variados.

De una forma similar a lo que pasa con las páginas web, GeM tiene la particularidad de que cuando un equipo gana contra otro con un puntaje muy alto, este se beneficia más que al hacerlo contra un equipo de bajo puntaje. En este sentido, se rompe con el concepto, talvez injusto, de que todos los partidos valen lo mismo y no importa contra quién se gane, lo único que importa son los puntos en la tabla de resultados.

Por último, podemos mencionar algunos cosas extra que podrían realizarse a futuro, como las diferentes implementaciones de matrices esparzas, junto a su correspondiente estudio de tiempo de ejecución. A su vez, podríamos implementar nuevos algoritmos para rankear páginas web y compararlos con Page Rank como también realizar implementaciones alternativas para rankear los equipos. En el caso de Page Rank, podríamos implementar las optimizaciones avanzadas del trabajo de Kamvar et al.[6] que garantizan un tiempo de convergencia mucho más rápido. Otras mejoras para incorporar a GeM podrían ser considerar otros factores de los partidos como posesión de pelota o corners a favor e ir agregándolos como peso en las aristas entre los nodos de los equipos.

6. Referencias

[1] Datahub. `http://datahub.io`.

[2] Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, April 1998.

[4] Kurt Bryan and Tanya Leise. The linear algebra behind google. *SIAM Review*, 48(3):569–581, 2006.

[5] Angela Y. Govan, Carl D. Meyer, and Rusell Albright. Generalizing google's pagerank to rank national football league teams. In *Proceedings of SAS Global Forum 2008*, 2008.

[6] Sepandar D. Kamvar, Taher H. Haveliwala, Christopher D. Manning, and Gene H. Golub. Extrapolation methods for accelerating pagerank computations. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 261–270, New York, NY, USA, 2003. ACM.

7. Enunciado

Contexto y motivación

A partir de la evolución de Internet durante la década de 1990, el desarrollo de motores de búsqueda se ha convertido en uno de los aspectos centrales para su efectiva utilización. Hoy en día, sitios como Yahoo, Google y Bing ofrecen distintas alternativas para realizar búsquedas complejas dentro de un red que contiene miles de millones de páginas web.

En sus comienzos, una de las características que distinguió a Google respecto de los motores de búsqueda de la época fue la calidad de los resultados obtenidos, mostrando al usuario páginas relevantes a la búsqueda realizada. El esquema general de los orígenes de este motor de búsqueda es brevemente explicado en Brin y Page [3], donde se mencionan aspectos técnicos que van desde la etapa de obtención de información de las páginas disponibles en la red, su almacenamiento e indexado y su posterior procesamiento, buscando ordenar cada página de acuerdo a su importancia relativa dentro de la red. El algoritmo utilizado para esta última etapa es denominado PageRank y es uno (no el único) de los criterios utilizados para ponderar la importancia de los resultados de una búsqueda. En este trabajo nos concentraremos en el estudio y desarrollo del algoritmo PageRank.

Por otro lado, las competencias deportivas, en todas sus variantes y disciplinas, requieren casi inevitablemente la comparación entre competidores mediante la confección de *Tablas de Posiciones* y *Rankings* en base a resultados obtenidos en un período de tiempo determinado. Estos ordenamientos de equipos están generalmente (aunque no siempre) basados en reglas relativamente claras y simples, como proporción de victorias sobre partidos jugados o el clásico sistema de puntajes por partidos ganados, empatados y perdidos. Sin embargo, estos métodos simples y conocidos por todos muchas veces no logran capturar la complejidad de la competencia y la comparación. Esto es particularmente evidente en ligas donde, por ejemplo, todos los equipos no juegan la misma cantidad de veces entre sí.

A modo de ejemplo, la NBA y NFL representan dos ligas con fixtures de temporadas regulares con estas características. Recientemente, el Torneo de Primera División de AFA se suma a este tipo de competencias, ya que la incorporación de la *Fecha de Clásicos* parece ser una interesante idea comercial, pero no tanto desde el punto de vista deportivo ya que cada equipo juega contra su *clásico* más veces que el resto. Como contraparte, éstos rankings son utilizados muchas veces como criterio de decisión, como por ejemplo para determinar la participación en alguna competencia de nivel internacional, con lo cual la confección de los mismos constituye un elemento sensible, afectando intereses deportivos y económicos de gran relevancia.

El problema, Parte I: PageRank y páginas web

El algoritmo PageRank se basa en la construcción del siguiente modelo. Supongamos que tenemos una red con n páginas web $Web = \{1, \ldots, n\}$ donde el objetivo es asignar a cada una de ellas un puntaje que determine la importancia relativa de la misma respecto de las demás. Para modelar las relaciones entre ellas, definimos la *matriz de conectividad* $W \in \{0,1\}^{n \times n}$ de forma tal que $w_{ij} = 1$ si la página j tiene un link a la página i, y $w_{ij} = 0$ en caso contrario. Además, ignoramos los *autolinks*, es decir, links de una página a sí misma, definiendo $w_{ii} = 0$. Tomando esta matriz, definimos el grado de la página j, n_j, como la cantidad de links salientes hacia otras páginas de la red, donde $n_j = \sum_{i=1}^{n} w_{ij}$. Además, notamos con x_j al puntaje asignado a la página $j \in Web$, que es lo que buscamos calcular.

La importancia de una página puede ser modelada de diferentes formas. Un link de la página $u \in Web$ a la página $v \in Web$ puede ser visto como que v es una página importante. Sin embargo, no queremos que una página obtenga mayor importancia simplemente porque es apuntada desde muchas páginas. Una forma de limitar esto es ponderar los links utilizando la importancia de la página de origen. En otras palabras, pocos links de páginas importantes pueden valer más que muchos links de páginas poco importantes. En particular, consideramos que la importancia de la página v obtenida mediante el link de la página u es proporcional a la importancia de la página u e inversamente proporcional al grado de u. Si la página u contiene n_u links, uno de los cuales apunta a la página v, entonces el aporte de ese link a la página v será x_u/n_u. Luego, sea $L_k \subseteq Web$ el conjunto de páginas que tienen un link a la página k. Para cada página pedimos que

$$x_k = \sum_{j \in L_k} \frac{x_j}{n_j}, \quad k = 1, \ldots, n. \tag{3}$$

Definimos $P \in \mathbb{R}^{n \times n}$ tal que $p_{ij} = 1/n_j$ si $w_{ij} = 1$, y $p_{ij} = 0$ en caso contrario. Luego, el modelo planteado en (3) es equivalente a encontrar un $x \in \mathbb{R}^n$ tal que $Px = x$, es decir, encontrar (suponiendo que existe) un autovector asociado al autovalor 1 de una matriz cuadrada, tal que $x_i \geq 0$ y $\sum_{i=1}^{n} x_i = 1$. En Bryan y Leise [4] y Kamvar et al. [6, Sección 1] se analizan ciertas condiciones que debe cumplir la red de páginas para garantizar la existencia de este autovector.

Una interpretación equivalente para el problema es considerar al *navegante aleatorio*. Éste empieza en una página cualquiera del conjunto, y luego en cada página j que visita sigue navegando a través de sus links, eligiendo el mismo con probabilidad $1/n_j$. Una situación particular se da cuando la página no tiene links salientes. En ese caso, consideramos que el navegante aleatorio pasa a cualquiera de las página de la red con probabilidad $1/n$. Para representar esta situación, definimos $v \in \mathbb{R}^{n \times n}$, con $v_i = 1/n$ y $d \in \{0,1\}^n$ donde $d_i = 1$ si $n_i = 0$, y $d_i = 0$ en caso contrario. La nueva matriz de transición es

$$\begin{aligned} D &= vd^t \\ P_1 &= P + D. \end{aligned}$$

Además, consideraremos el caso de que el navegante aleatorio, dado que se encuentra en la página j, decida visitar una página cualquiera del conjunto, independientemente de si esta se encuentra o no referenciada por j (fenómeno conocido como *teletransportación*). Para ello, consideramos que esta decisión se toma con una probabilidad $c \geq 0$, y podemos incluirlo al modelo de la siguiente forma:

$$\begin{aligned} E &= v\bar{1}^t \\ P_2 &= cP_1 + (1-c)E, \end{aligned}$$

donde $\bar{1} \in \mathbb{R}^n$ es un vector tal que todas sus componentes valen 1. La matriz resultante P_2 corresponde a un enriquecimiento del modelo formulado en (3). Probabilísticamente, la componente x_j del vector solución (normalizado) del sistema $P_2 x = x$ representa la proporción del tiempo que, en el largo plazo, el navegante aleatorio pasa en la página $j \in Web$. Denotaremos con π al vector solución de la ecuación $P_2 x = x$, que es comúnmente denominado *estado estacionario*.

En particular, P_2 corresponde a una matriz *estocástica por columnas* que cumple las hipótesis planteadas en Bryan y Leise [4] y Kamvar et al. [6], tal que P_2 tiene un autovector asociado al autovalor 1, los demás autovalores de la matriz cumplen $1 = \lambda_1 > |\lambda_2| \geq \cdots \geq |\lambda_n|$ y, además, la dimensión del autoespacio asociado al autovalor λ_1 es 1. Luego, π puede ser calculada de forma estándar utilizando el método de la potencia.

Una vez calculado el ranking, se retorna al usuario las t páginas con mayor puntaje.

El problema, Parte II: PageRank y ligas deportivas

Existen en la literatura distintos enfoques para abordar el problema de determinar el *ranking* de equipos de una competencia en base a los resultados de un conjunto de partidos. En Govan et al. [5] se hace una breve reseña de dos ellos, y los autores proponen un nuevo método basado en el algoritmo PageRank que denominan GeM[4]. Conceptualmente, el método GeM representa la temporada como un red (grafo) donde las páginas web representan a los equipos, y existe un link (que tiene un valor, llamado peso, asociado) entre dos equipos que los relaciona modelando los resultados de los posibles enfrentamientos entre ellos. En base a este modelo, Govan et al. [5] proponen calcular el ranking de la misma forma que en el caso de las páginas web.

En su versión básica, que es la que consideraremos en el presente trabajo, el método GeM (ver, e.g., [5, Sección GeM Ranking Method]) es el siguiente[5]:

1. La temporada se representa mediante un grafo donde cada equipo representa un nodo y existe un link de i a j si el equipo i perdió al menos una vez con el equipo j.
2. Se define la matriz $A^t \in \mathbb{R}^{n \times n}$

$$A^t_{ji} = \begin{cases} w_{ji} & \text{si el equipo } i \text{ perdió con el equipo } j, \\ 0 & \text{en caso contrario,} \end{cases}$$

[4] Aunque no se especifica, asumimos que el nombre se debe a las iniciales de los autores.

[5] Notar que en artículo, Govan et al. [5] lo definen sobre la traspuesta. La definición y las cuentas son equivalentes, simplemente se modifica para mantener la consistencia a lo largo del enunciado.

donde w_{ji} es la diferencia absoluta en el marcador. En caso de que i pierda más de una vez con j, w_{ji} representa la suma acumulada de diferencias. Notar que A^t es una generalización de la matriz de conectividad W definida en la sección anterior.

3. Definir la matriz $H^t_{ji} \in \mathbb{R}^{n \times n}$ como

$$H^t_{ji} = \begin{cases} A^t_{ji} / \sum_{k=1}^{n} A^t_{ki} & \text{si hay un link } i \text{ a } j, \\ 0 & \text{en caso contrario.} \end{cases}$$

4. Tomar $P = H^t$, y aplicar el método PageRank como fue definido previamente, siendo π la solución a la ecuación $P_2 x = x$. Notar que los páginas sin links salientes, en este contexto se corresponden con aquellos equipos que se encuentran invictos.

5. Utilizar los puntajes obtenidos en π para ordenar los equipos.

En función del contexto planteado previamente, el método GeM define una estructura que relaciona equipos dependiendo de los resultados parciales y obtener un ranking utilizando solamente esta información.

Enunciado

El objetivo del trabajo es experimentar en el contexto planteado utilizando el algoritmo PageRank con las variantes propuestas. A su vez, se busca comparar los resultados obtenidos cualitativa y cuantitativamente con los algoritmos tradicionales utilizados en cada uno de los contextos planteados. Los métodos a implementar (como mínimo) en ambos contexto planteados por el trabajo son los siguientes:

1. *Búsqueda de páginas web:* PageRank e IN-DEG, éste último consiste en definir el ranking de las páginas utilizando solamente la cantidad de ejes entrantes a cada una de ellas, ordenándolos en forma decreciente.

2. *Rankings en competencias deportivas:* GeM y al menos un método estándar propuesto por el grupo (ordenar por victorias/derrotas, puntaje por ganado/empatado/perdido, etc.) en función del deporte(s) considerado(s).

El contexto considerado en 1., en la búsqueda de páginas web, representa un desafío no sólo desde el modelado, si no también desde el punto de vista computacional considerando la dimensión de la información y los datos a procesar. Luego, dentro de nuestras posibilidades, consideramos un entorno que simule el contexto real de aplicación donde se abordan instancias de gran escala (es decir, n, el número total de páginas, es grande). Para el desarrollo de PageRank, se pide entonces considerar el trabajo de Bryan y Leise [4] donde se explica la intución y algunos detalles técnicos respecto a PageRank. Además, en Kamvar et al. [6] se propone una mejora del mismo. Si bien esta mejora queda fuera de los alcances del trabajo, en la Sección 1 se presenta una buena formulación del algoritmo. En base a su definición, P_2 no es una matriz esparsa. Sin embargo, en Kamvar et al. [6, Algoritmo 1] se propone una forma alternativa para computar $x^{(k+1)} = P_2 x^{(k)}$. Este resultado debe ser utilizado para mejorar el almacenamiento de los datos.

En la práctica, el grafo que representa la red de páginas suele ser esparso, es decir, una página posee relativamente pocos links de salida comparada con el número total de páginas. A su vez, dado que n tiende a ser un número muy grande, es importante tener en cuenta este hecho a la hora de definir las estructuras de datos a utilizar. Luego, desde el punto de vista de implementación se pide utilizar alguna de las siguientes estructuras de datos para la representación de las matrices esparsas: *Dictionary of Keys* (dok), *Compressed Sparse Row* (CSR) o *Compressed Sparse Column* (CSC). Se deberá incluir una justificación respecto a la elección que consdiere el contexto de aplicación. Además, para PageRank se debe implementar el método de la potencia para calcular el autovector principal. Esta implementación debe ser realizada íntegramente en C++.

En función de la experimentación, se deberá realizar un estudio particular para cada algoritmo (tanto en términos de comportamiento del mismo, como una evaluación de los resultados obtenidos) y luego se procederá a comparar cualitativamente los rankings generados. La experimentación deberá incluir como mínimo los siguientes experimentos:

1. Estudiar la convergencia de PageRank, analizando la evolución de la norma Manhattan (norma L_1) entre dos iteraciones sucesivas. Comparar los resultados obtenidos para al menos dos instancias de tamaño mediano-grande, variando el valor de c.

2. Estudiar el tiempo de cómputo requerido por PageRank.

3. Para cada algoritmo, proponer ejemplos de tamaño pequeño que ilustren el comportamiento esperado (puede ser utilizando las herramientas provistas por la cátedra o bien generadas por el grupo).

Puntos opcionales:

1. Demostrar que los pasos del Algoritmo 1 propuesto en Kamvar et al. [6] son correctos y computan P_2x.

2. Establecer una relación con la proporción entre $\lambda_1 = 1$ y $|\lambda_2|$ para la convergencia de PageRank.

El segundo contexto de aplicación no presenta mayores desafíos desde la perspectiva computacional, ya que en el peor de los casos una liga no suele tener mas que unas pocas decenas de equipos. Más aún, es de esperar que en general la matriz que se obtiene no sea esparsa, ya que probablemente un equipo juegue contra un número significativo de contrincantes. Sin embargo, la popularidad y sensibilidad del problema planteado requieren de un estudio detallado y pormenorizado de la calidad de los resultados obtenidos. El objetivo en este segundo caso de estudio es puramente experimental.

En función de la implementación, aún cuando no represente la mejor opción, es posible reutilizar y adaptar el desarrollo realizado para páginas web. También es posible realizar una nueva implementación desde cero, simplificando la operatoria y las estructuras, en C++, MATLAB o PYTHON.

La experimentación debe ser realizada con cuidado, analizando (y, eventualmente, modificando) el modelo de GeM:

1. Considerar al menos un conjunto de datos reales, con los resultados de cada fecha para alguna liga de alguún deporte.

2. Notar que el método GeM asume que no se producen empates entre los equipos (o que si se producen, son poco frecuentes). En caso de considerar un deporte donde el empate se da con cierta frecuencia no despreciable (por ejemplo, fútbol), es fundamental aclarar como se refleja esto en el modelo y analizar su eventual impacto.

3. Realizar experimentos variando el parámetro c, indicando como impacta en los resultados. Analizar la evolución del ranking de los equipos a través del tiempo, evaluando también la evolución de los rankings e identificar características/hechos particulares que puedan ser determinantes para el modelo, si es que existe alguno.

4. Comparar los resultados obtenidos con los reales de la liga utilizando el sistema estándar para la misma.

Puntos opcionales:

1. Proponer (al menos) dos formas alternativas de modelar el empate entre equipos en GeM.

Parámetros y formato de archivos

El programa deberá tomar por línea de comandos dos parámetros. El primero de ellos contendrá la información del experimento, incluyendo el método a ejecutar (`alg`, 0 para PageRank, 1 para el método alternativo), la probabilidad de teletransportación c, el tipo de instancia (0 páginas web, 1 deportes), el *path* al archivo/directorio conteniendo la definición de la red (que debe ser relativa al ejecutable, o el path absoluto al archivo) y el valor de tolerancia utilizado en el criterio de parada del método de la potencia.

El siguiente ejemplo muestra un caso donde se pide ejecutar PageRank, con una probabilidad de teletransportación de 0.85, sobre la red descripta en `test1.txt` (que se encuentra en el directorio `tests/`), correspondiente a una instancia de ranking aplicado a deportes y con una tolerancia de corte de $0{,}0001$.

```
0 0.85 1 tests/red-1.txt 0.0001
```

Para la definición del grafo que representa la red, se consideran dos bases de datos de instancias con sus correspondientes formatos. La primera de ellas es el conjunto provisto en SNAP [2] (el tipo de instancia es 0), con redes de tamaño grande obtenidos a partir de datos reales. Además, se consideran las instancias que se forman a partir de resultados de partidos entre equipos, para algún deporte elegido por el grupo.

En el caso de la base de SNAP, los archivos contiene primero cuatro líneas con información sobre la instancia (entre ellas, n y la cantidad total de links, m) y luego m líneas con los pares i, j indicando que i apunta a j. A modo de ejemplo, a continuación se muestra el archivo de entrada correspondiente a la red propuesta en Bryan y Leise [4, Figura 1]:

```
# Directed graph (each unordered pair of nodes is saved once):
# Example shown in Bryan and Leise.
# Nodes: 4 Edges: 8
# FromNodeId    ToNodeId
1   2
1   3
1   4
2   3
2   4
3   1
4   1
4   3
```

Para el caso de rankings en ligas deportivas, el archivo contiene primero una línea con información sobre la cantidad de equipos (n), y la cantidad de partidos totales a considerar (k). Luego, siguen k lńeas donde cada una de ellas representa un partido y contiene la siguiente información: número de fecha (es un dato opcional al problema, pero que puede ayudar a la hora de experimentar), equipo i, goles equipo i, equipo j, goles equipo j. A continuación se muestra el archivo de entrada con la información del ejemplo utilizado en Govan et al. [5]:

```
6 10
1 1 16 4 13
1 2 38 5 17
1 2 28 6 23
1 3 34 1 21
1 3 23 4 10
1 4 31 1 6
1 5 33 6 25
1 5 38 4 23
1 6 27 2 6
1 6 20 5 12
```

Es importante destacar que, en este último caso, los equipos son identificados mediante un número. Opcionalmente podrá considerarse un archivo que contenga, para cada equipo, cuál es el código con el que se lo identifica.

Una vez ejecutado el algoritmo, el programa deberá generar un archivo de salida que contenga una línea por cada página (n líneas en total), acompañada del puntaje obtenido por el algoritmo PageRank/IN-DEG/método alternativo.

Para generar instancias de páginas web, es posible utilizar el código Python provisto por la cátedra. La utilización del mismo se encuentra descripta en el archivo README. Es importante mencionar que, para que el mismo funcione, es necesario tener acceso a Internet. En caso de encontrar un bug en el mismo, por favor contactar a los docentes de la materia a través de la lista. Desde ya, el código puede ser modificado por los respectivos grupos agregando todas aquellas funcionalidades que consideren necesarias.

Para instancias correspondientes a resultados entre equipos, la cátedra provee un conjunto de archivos con los resultados del Torneo de Primera División del Fútbol Argentino hasta la Fecha 23. Es importante aclarar que los dos partidos suspendidos, River - Defensa y Justicia y Racing - Godoy Cruz han sido arbitrariamente completados con un resultado inventado, para simplificar la instancia. En función de datos reales, una alternativa es considerar el repositorio DataHub [1], que contiene información estadística y resultados para distintas ligas y deportes de todo el mundo.

Fechas de entrega

- *Formato Electrónico:* Martes 6 de Octubre de 2015, hasta las 23:59 hs, enviando el trabajo (informe + código) a la dirección `metnum.lab@gmail.com`. El subject del email debe comenzar con el texto `[TP2]` seguido de la lista de apellidos de los integrantes del grupo.
- *Formato físico:* Miércoles 7 de Octubre de 2015, a las 18 hs. en la clase práctica.

Importante: El horario es estricto. Los correos recibidos después de la hora indicada serán considerados re-entrega.

A. Código C++

A.1. utils.h

```
#ifndef UTILS_H_
#define UTILS_H_

#include <vector>
#include <map>
#include <iostream>
#include <fstream>
#include <utility>
#include <iomanip>
#include <stdlib.h>
#include <ctime>

using namespace std;

#define TELETRANSPORTACION_DEFAULT 0.85

namespace utils {
  static double teletransportacion = TELETRANSPORTACION_DEFAULT;

  static int random_in_range(int min, int max) {
    srand (time(NULL));
    return min + (rand() % (max - min + 1));
  }


  static void set_teletransportacion(double t) {
    teletransportacion = t;
  }

  template <typename T>
  static vector< vector<T> > vector2matrix(const vector<T> &x) {
    vector< vector<T> > aux;
    aux.push_back(x);
    return aux;
  }

  template <typename T>
  static vector< vector<T> > row2Column(const vector<T> &x) {
    vector< vector<T> > aux(x.size());
    for (unsigned int i = 0; i < x.size(); ++i) {
      vector<T> elem;
      elem.push_back(x[i]);
      aux[i] = (elem);
    }
    return aux;
  }

  template <typename T>
  static vector<T> column2Row(const vector< vector<T> > &x) {
    vector<T> aux(x.size());
```

```
  for (unsigned int i = 0; i < x.size(); ++i) {
    aux[i] = x[i][0];
  }
  return aux;
}

static double norma2(const vector<double> &x) {
  double sum = 0;
  for (unsigned int i = 0; i < x.size(); i++) {
    sum += pow(x[i],2);
  }

  return sqrt(sum);
}

static double norma1(const vector<double> &x) {
  double sum = 0;
  for (unsigned int i = 0; i < x.size(); i++) {
    sum += fabs(x[i]);
  }
  return sum;
}

static double normaManhattan(const vector<double> &x, const vector<
    double> &y) {
  double sum = 0;
  for (unsigned int i = 0; i < x.size(); i++) {
    sum += fabs(x[i]-y[i]);
  }
  return sum;
}

template <typename T>
static vector< vector<T> > sum(const vector< vector<T> > &A, const
    vector< vector<T> > &B) {
  if (A.size() != B.size() || A[0].size() != B[0].size()) {
    cout << "No coinciden los tamaños para sumar las matrices" << endl
        ;
    cout << "A: " << A.size() << "x" << A[0].size() << endl;
    cout << "B: " << B.size() << "x" << B[0].size() << endl;
    return vector< vector<T> >();
  }

  int rows = A.size();
  int cols = A[0].size();
  vector< vector<T> > resultado(rows, vector<T>(cols, 0));

  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
      resultado[i][j] += A[i][j] + B[i][j];
    }
  }
  return resultado;
}
```

```
template <typename T>
static vector<T> sumVector(const vector<T> &x, const vector<T> &y) {
  if (x.size() != y.size()) {
    cout << "No coinciden los tamaños para sumar los vectores" << endl
       ;
    cout << "x: " << x.size() << endl;
    cout << "y: " << y.size() << endl;
    return vector<T>();
  }

  vector<T> resultado(x.size(), 0);

  for (int i = 0; i < x.size(); ++i) {
    resultado[i] = x[i] + y[i];
  }
  return resultado;
}

template <typename T>
static vector<T> scaleVector(const vector<T> &A, const T &scalar) {
  int n = A.size();
  vector<T> resultado(n, 0);

  for (int i = 0; i < n; ++i) {
    resultado[i] = A[i] * scalar;
  }
  return resultado;
}

template <typename T>
static vector< vector<T> > scaleMatriz(const vector< vector<T> > &A,
   const T &scalar) {
  int rows = A.size();
  int cols = A[0].size();
  vector< vector<T> > resultado(rows, vector<T>(cols, 0));

  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
      resultado[i][j] = A[i][j] * scalar;
    }
  }
  return resultado;
}

template <typename T>
static vector< vector<T> > multiply(const vector< vector<T> > &A,
   const vector< vector<T> > &B) {
  if (A.size() == 0 || B.size() == 0 || A[0].size() != B.size()) {
    cout << "No coinciden los tamaños para multiplicar las matrices"
       << endl;
    cout << "A: " << A.size() << "x" << A[0].size() << endl;
    cout << "B: " << B.size() << "x" << B[0].size() << endl;
    return vector< vector<T> >();
```

```
    }

    int rows = A.size();
    int cols = B[0].size();
    int dim = B.size();
    vector< vector<T> > resultado(rows, vector<T>(cols, 0));

    for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < cols; ++j) {
        for (int k = 0; k < dim; ++k) {
          resultado[i][j] += A[i][k] * B[k][j];
        }
      }
    }
    return resultado;
}

/**
* Toma una matriz de conectividad (w_{i,j} tiene un 1 si j tiene un
    link saliente a i)
* y devuelve su matriz de transición asociada.
*/
static vector< vector<double> > matrizTransicion(const vector< vector<
    int> > &A) {
    int n = A.size();
    // convierto la matriz A en una matriz P, donde p_{i,j} = 1/n_{j} si
        w_{i,j} = 1, y 0 en caso contrario.
    // Y n_{j} = cantidad de links salientes desde la página j
    vector< vector<double> >P(n, vector<double>(n, 0));
    vector<int> links_salientes(n,0);
    for (int j = 0; j < n; j++) {
      // cuento los links salientes del nodo j
      for (int i = 0; i < n; i++) {
        links_salientes[j] += A[i][j];
      }

      // asigno el peso a la celda p_{i,j}
      for (int i = 0; i < n; i++) {
        P[i][j] = links_salientes[j] > 0 ? double(A[i][j])/double(
            links_salientes[j]) : 0;
      }
    }

    // convierto la matriz P en una matriz P1 = P + D, con D = v*d^{t},
        con v_{i} = 1/n y d_{i} = 1 si n_{i} = 0, y 0 en caso contrario
    vector<double> v(n, double(1)/double(n));
    vector<double> d(n, 0);
    for (int j = 0; j < n; j++) {
      if (links_salientes[j] == 0) {
        d[j] = 1;
      }
    }

    vector< vector<double> > D = multiply(row2Column(v), vector2matrix(d
```

```
        ));
    vector< vector<double> > P1 = sum(P, D);

    // agrego la probabilidad del navegante aleatorio: P2 = c*P1 + (1-c)
       *E, con E = v*1^{t}
    vector<double> unos(n, 1);
    vector< vector<double> > E = multiply(row2Column(v), vector2matrix(
       unos));

    vector< vector<double> > P2 = sum(scaleMatriz(P1, teletransportacion
       ), scaleMatriz(E, 1-teletransportacion));

    return P2;
}

static vector< vector<int> > cargarSNAP(const char* entrada, int
    offset = 1) {
    ifstream datos;
    datos.open(entrada);
    if (!datos.good()) {
      cout << endl;
      cout << "\tNo se pudo cargar el archivo: " << entrada << endl;
    }

    // skip first two lines
    string line;
    getline(datos,line);
    getline(datos,line);
    // get how many nodes and edges has the graph
    int nodes = 0, edges = 0;
    datos >> line >> line; // Skip "# Nodes:"
    datos >> nodes;
    datos >> line; // Skip " Edges:"
    datos >> edges;
    // skip another line
    datos >> line;
    getline(datos,line);
    // load the graph edges
    vector< vector<int> >A(nodes, vector<int>(nodes, 0));
    for (int e = 0; e < edges; e++) {
      int i, j;
      datos >> i >> j;
      A[j-offset][i-offset] = 1;
    }
    datos.close();

    return A;
}

static vector< map<int, double> > cargarSNAPEsparso(const char*
    entrada, int offset = 1) {
    ifstream datos;
    datos.open(entrada);
    if (!datos.good()) {
```

```
      cout << endl;
      cout << "\tNo se pudo cargar el archivo: " << entrada << endl;
    }

    // skip first two lines
    string line;
    getline(datos,line);
    getline(datos,line);
    // get how many nodes and edges has the graph
    int nodes = 0, edges = 0;
    datos >> line >> line; // Skip "# Nodes:"
    datos >> nodes;
    datos >> line; // Skip " Edges:"
    datos >> edges;
    // skip another line
    datos >> line;
    getline(datos,line);
    // load the graph edges
    vector< map<int, double> > A(nodes);
    vector<int> links_salientes(nodes);
    for (int e = 0; e < edges; e++) {
      int i, j;
      datos >> i >> j;
      A[j-offset].insert(pair<int,double>(i-offset,1));
      links_salientes[i-offset]++;
    }
    datos.close();

    // convierto la matriz A en una matriz P, donde p_{i,j} = 1/n_{j} si
        w_{i,j} = 1, y 0 en caso contrario.
    // Y n_{j} = cantidad de links salientes desde la página j
    for (int i = 0; i < nodes; i++) {
      typedef map<int, double>::iterator it_type;
      for(it_type iterator = A[i].begin(); iterator != A[i].end();
          iterator++) {
        int j = iterator->first;
        iterator->second = double(1)/double(links_salientes[j]);
      }
    }

    return A;
  }

  static vector< vector<double> > cargarLigaDeportiva(const char*
      entrada) {
    return vector< vector<double> >();
  }

  template <typename T>
  static void imprimirMatriz(const vector< vector<T> > &A) {
    for(unsigned int i = 0; i < A.size(); i++){
      cout<<"[";
      for(unsigned int j = 0; j < A[i].size(); j++){
        cout<<" "<< A[i][j];
```

```
      }
      cout<<"␣]"<<endl;
    }
  }

  template <typename T>
  static void imprimirVector(const vector<T> &A) {
    cout<<"[";
    for(unsigned int i = 0; i < A.size(); i++){
      cout<<"␣"<< A[i];
    }
    cout<<"␣]"<<endl;
  }
};

#endif // UTILS_H_INCLUDED
```

A.2. in_deg.cpp

```
#include <in_deg.h>

InDeg::InDeg(vector< vector<int> > A){
  if (A.size() == 0 || A[0].size() == 0 || A.size() != A[0].size()) {
    throw invalid_argument("InDeg␣instanciado␣con␣una␣matriz␣que␣no␣es␣
        cuadrada");
  }
  this->A = A;
}

vector< typename InDeg::rankeable > InDeg::rankear() {
  vector< typename InDeg::rankeable > ranking;

  for (unsigned int i = 0; i < A.size(); i++) {
    int grado = 0;
    for (unsigned int j = 0; j < A.size(); j++) {
      if(A[i][j]) {
        grado++;
      }
    }
    ranking.push_back(rankeable(i, grado));
  }

  sort(ranking.begin(), ranking.end());
  return ranking;
}
```

A.3. in_deg.h

```
#ifndef IN_DEG_H_
#define IN_DEG_H_

#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
```

```
#include <math.h>
#include <utils.h>
#include <stdexcept>

using namespace std;
using namespace utils;

class InDeg {
  public:
    struct rankeable {
      rankeable(int pos, double v) : posicion(pos), valor(v) {};
      bool operator<(const rankeable& other) const { return valor >
          other.valor; }

      int posicion;
      double valor;
    };
    InDeg(vector< vector<int> > A);
    vector< rankeable > rankear();
  private:
    vector< vector<int> > A;
};

#endif // IN_DEG_H_INCLUDED
```

A.4. page_rank.cpp

```
#include <page_rank.h>

PageRank::PageRank(vector< vector<double> > A){
  if (A.size() == 0 || A[0].size() == 0 || A.size() != A[0].size()) {
    throw invalid_argument("PageRank instanciado con una matriz que no
        es cuadrada");
  }
  this->A = A;
}

void PageRank::set_precision(double p) {
  precision = p;
}

vector< typename PageRank::rankeable > PageRank::rankear() {
  vector<double> autovector = metodoPotencia(); // autovector asociado
     al autovalor 1

  // cout << "Norma2: " << norma2(autovector) << endl;
  vector< typename PageRank::rankeable > ranking;
  for (unsigned int i = 0; i < autovector.size(); i++) {
    ranking.push_back(rankeable(i, autovector[i]));
  }

  sort(ranking.begin(), ranking.end());
  return ranking;
}
```

```
vector<double> PageRank::metodoPotencia() {
  int n = A.size();
  // creo el x aleatorio
  vector<double> x(n, 0);
  for (int i = 0; i < n; i++) {
    x[i] = random_in_range(1,50);
  }

  // traspongo x para poder multiplicarlo por A
  vector< vector<double> > aux = row2Column(x);

  vector< vector<double> > B = multiply(A, A);
  vector< vector<double> > C = A;
  double delta = phi(column2Row(multiply(B, aux))) / phi(column2Row(
     multiply(C, aux)));
  double last_delta = INFINITY;

  while (fabs(delta - last_delta) > precision) {
    C = B;
    B = multiply(B, A);
    last_delta = delta;
    delta = phi(column2Row(multiply(B, aux))) / phi(column2Row(multiply(
       C, aux)));
  }
  vector<double> v = column2Row(multiply(B, aux));

  return scaleVector(v, 1/norma1(v));
}

double PageRank::phi(const vector<double> &x) {
  double ret = x[0];
  double max = abs(ret);
  for (unsigned int i = 1; i < x.size(); i++) {
    if (max < abs(x[i])) {
      ret = x[i];
      max = abs(ret);
    }
  }

  return ret;
}
```

A.5. page_rank.h

```
#ifndef PAGE_RANK_H_
#define PAGE_RANK_H_

#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <math.h>
#include <utils.h>
```

```
#include <stdexcept>

using namespace std;
using namespace utils;

#define PRECISION_DEFAULT 0.001

class PageRank{
  public:
    struct rankeable {
      rankeable(int pos, double v) : posicion(pos), valor(v) {};
      bool operator<(const rankeable& other) const { return valor >
          other.valor; }

      int posicion;
      double valor;
    };

    PageRank(vector< vector<double> > A);
    vector< rankeable > rankear();
    void set_precision(double p);
  private:
    vector< vector<double> > A;
    double precision = PRECISION_DEFAULT;
    vector<double> metodoPotencia();
    double phi(const vector<double> &A);
};

#endif // PAGE_RANK_H_INCLUDED
```

A.6. page_rank_esparso.cpp

```
#include <page_rank_esparso.h>

PageRankEsparso::PageRankEsparso(vector< map<int, double> > A){
  this->A = A;
}

void PageRankEsparso::set_precision(double p) {
  precision = p;
}

vector< typename PageRankEsparso::rankeable > PageRankEsparso::rankear()
    {
  vector<double> autovector = metodoPotencia(); // autovector asociado
     al autovalor 1

  // cout << "Norma2: " << norma2(autovector) << endl;
  vector< typename PageRankEsparso::rankeable > ranking;
  for (unsigned int i = 0; i < autovector.size(); i++) {
    ranking.push_back(rankeable(i, autovector[i]));
  }

  sort(ranking.begin(), ranking.end());
```

```
  return ranking;
}

vector<double> PageRankEsparso::metodoPotencia() {
  int n = A.size();
  // creo el x aleatorio
  vector<double> x(n, 0);
  for (int i = 0; i < n; i++) {
    x[i] = random_in_range(1,50);
  }
  // creo el v aleatorio
  vector<double> v(n, double(1)/double(n));

  vector<double> y = x;
  double delta = INFINITY;
  double last_delta;
  do {
    x = y;
    y = scaleVector(multiplyEsparso(A, x), teletransportacion);
    double w = norma1(x) - norma1(y);
    y = sumVector(y, scaleVector(v, w));
    last_delta = delta;
    delta = phi(y) / phi(x);
  } while (fabs(delta - last_delta) > precision);

  return scaleVector(y, 1/norma1(y));
}

vector<double> PageRankEsparso::multiplyEsparso(const vector< map<int,
    double> > &A, const vector<double> &x) {
  vector<double> y(x.size());
  for (unsigned int i = 0; i < x.size(); i++) {
    typedef map<int, double>::const_iterator it_type;
    double sum = 0;
    for(it_type iterator = A[i].begin(); iterator != A[i].end();
        iterator++) {
      int j = iterator->first;
      sum += (iterator->second)*x[j];
    }
    y[i] = sum;
  }
  return y;
}

void PageRankEsparso::imprimirEsparso(const vector< map<int, double> > &
    A) {
  vector< vector<double> > v(A.size(), vector<double>(A.size(), 0));
  for (unsigned int i = 0; i < A.size(); i++) {
    typedef map<int, double>::const_iterator it_type;
    for(it_type iterator = A[i].begin(); iterator != A[i].end();
        iterator++) {
      v[i][iterator->first] = iterator->second;
    }
  }
```

```
  imprimirMatriz(v);
}

double PageRankEsparso::phi(const vector<double> &x) {
  double ret = x[0];
  double max = abs(ret);
  for (unsigned int i = 1; i < x.size(); i++) {
    if (max < abs(x[i])) {
      ret = x[i];
      max = abs(ret);
    }
  }

  return ret;
}
```

A.7. page_rank_esparso.h

```
#ifndef PAGE_RANK_ESPARSO_H_
#define PAGE_RANK_ESPARSO_H_

#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <math.h>
#include <utils.h>
#include <stdexcept>

using namespace std;
using namespace utils;

#define PRECISION_DEFAULT 0.001

class PageRankEsparso {
  public:
    struct rankeable {
      rankeable(int pos, double v) : posicion(pos), valor(v) {};
      bool operator<(const rankeable& other) const { return valor >
          other.valor; }

      int posicion;
      double valor;
    };

    PageRankEsparso(vector< map<int, double> > A);
    vector< rankeable > rankear();
    void set_precision(double p);
  private:
    vector< map<int, double> > A;
    double precision = PRECISION_DEFAULT;
    vector<double> metodoPotencia();
    vector<double> multiplyEsparso(const vector< map<int, double> > &A,
        const vector<double> &x);
```

```
    void imprimirEsparso(const vector< map<int, double> > &A);
    double phi(const vector<double> &A);
};

#endif // PAGE_RANK_ESPARSO_H_INCLUDED
```

A.8. footbal_rankings.m

```
1;

function res = GeM(in_filename, out_filename, team_codes_filename = '',
   c = 0.85, date_limit = 0, pres = 0.0001)
  has_date_limit = date_limit != 0;

  fid = fopen(in_filename, 'r');
  [teams, matches] = fscanf(fid, '%u␣%u' ,"C");

  # Genero la matriz A
  A = zeros(teams);
  while (matches > 0)
    [f, e0, p0, e1, p1] = fscanf(fid, '%u␣%u␣%u␣%u␣%u', "C");

    if(has_date_limit && f > date_limit)
      matches = 0;
    else
      if(p0 < p1)
        # si e0 perdió contra e1
        A(e0, e1) += p1 - p0;
      elseif(p0 > p1)
        # si e1 perdió contra e0
        A(e1, e0) += p0 - p1;
      endif

      matches--;
    endif
  endwhile
  fclose(fid);

  # Genero la matriz H
  H = zeros(teams);
  a = zeros(teams, 1);
  for i = 1:teams
    suma = sum(A(i, :));
    if(suma > 0)
      H(i, :) = A(i, :)/sum(A(i, :));
    else
      # vector para equipos invictos
      a(i) = 1;
    endif
  endfor

  # Genero la matriz G
  e = ones(teams, 1);
```

```
  # Genero vectores de personalización
  # (por defecto distribución uniforme)

  # Equipos invictos
  u = ones(teams, 1)/teams;
  # teletransportación
  v = u;

  G = c*[H + a*u'] + (1 - c)*e*v';

  # Calculo autovalores y autovectores
  [V, l] = eig(G');

  # Busco el autovalor 1
  i = 1;
  while(abs(l(i, i) - 1) > pres)
    i++;
  endwhile
  x = V(:, i);

  # Normalizo el vector solución
  x = abs(x)/sum(abs(x));

  # Ordeno las soluciones
  S = zeros(teams, 2);
  for i = 1:teams
    S(i, 1) = i;
    S(i, 2) = x(i);
  endfor

  res = S;
  if(!strcmp(out_filename, ''))
    save_solution(teams, S, out_filename, team_codes_filename);
  endif
endfunction

function GeM_evolution(in_filename, out_filename, teams, number_of_dates
   )
  season_dates = cell(1, number_of_dates);
  for i = 1:number_of_dates
    season_dates{i} = GeM(in_filename, '', '', 0.85, i);
  endfor

  fid = fopen(out_filename, 'w');
  for i = 1:number_of_dates
    % Imprimo número de fecha
    fprintf(fid, '%u', i);
    S = season_dates{i};
    % Imprimo el ranking de cada equipo en esa fecha
    for j = 1:teams
      fprintf(fid, ' %f', S(j,2));
    endfor
    fprintf(fid, '\n');
  endfor
```

```
  fclose(fid);

endfunction

function AFA(in_filename, out_filename, team_codes_filename = '',
    date_limit = 0)
  has_date_limit = date_limit != 0;

  fid = fopen(in_filename, 'r');
  [teams, matches] = fscanf(fid, '%u␣%u' ,"C");

  # Genero la matriz S
  S = zeros(teams, 2);
  S(:, 1) = 1:teams;
  while (matches > 0)
    [f, e0, p0, e1, p1] = fscanf(fid, '%u␣%u␣%u␣%u␣%u', "C");

    if(has_date_limit && f > date_limit)
      matches = 0;
    else
      if(p0 == p1)
        # si e0 empató contra e1
        S(e0, 2) += 1;
        S(e1, 2) += 1;
      elseif(p0 > p1)
        # si ganó e0
        S(e0, 2) += 3;
      else
        # si ganó e1
        S(e1, 2) += 3;
      endif

      matches--;
    endif
  endwhile
  fclose(fid);

  save_solution(teams, S, out_filename, team_codes_filename);
endfunction

function AFA_evolution(in_filename, out_filename, matches_per_date,
    normalize_score = 1)
  fid = fopen(in_filename, 'r');
  [teams, matches] = fscanf(fid, '%u␣%u' ,"C");

  # Genero la matriz S
  S = zeros(teams, 2);
  S(:, 1) = 1:teams;
  number_of_dates = matches/matches_per_date;
  season_dates = cell(1, number_of_dates);
  while (matches > 0)
    [f, e0, p0, e1, p1] = fscanf(fid, '%u␣%u␣%u␣%u␣%u', "C");

    if(p0 == p1)
```

```
      # si e0 empató contra e1
      S(e0, 2) += 1;
      S(e1, 2) += 1;
    elseif(p0 > p1)
      # si ganó e0
      S(e0, 2) += 3;
    else
      # si ganó e1
      S(e1, 2) += 3;
    endif

    matches--;
    if (mod(matches, matches_per_date) == 0)
      X = S;
      if (normalize_score == 1)
        X(:, 2) = X(:, 2)/sum(X(:, 2));
      endif
      season_dates{f} = X;
    endif
  endwhile
  fclose(fid);

  fid = fopen(out_filename, 'w');
  for i = 1:number_of_dates
    % Imprimo número de fecha
    fprintf(fid, '%u', i);
    S = season_dates{i};
    % Imprimo el ranking de cada equipo en esa fecha
    for j = 1:teams
      fprintf(fid, ' %f', S(j,2));
    endfor
    fprintf(fid, '\n');
  endfor
  fclose(fid);
endfunction

function save_solution(teams, S, out_filename, team_codes_filename)
  has_team_codes = !strcmp(team_codes_filename, '');

  S = sortrows(S, 2);

  # Si tengo el nombre de cada equipo, lo cargo
  if(has_team_codes)
    i = teams;
    fid = fopen(team_codes_filename, 'r');
    teamcodes = cell(1, teams);
    while (i > 0)
      team_code = fgetl(fid);
      codearray = strsplit(team_code, ',');
      teamcodes{str2num(codearray{1})} = codearray{2};
      i--;
    endwhile
    fclose(fid);
  endif
```

```
  # Escribo la solución
  fid = fopen(out_filename, 'w');

  if(has_team_codes)
    for i = 0:teams - 1
      fprintf(fid, '%u, %s %f\n', S(teams - i,1), teamcodes{S(teams - i
          ,1)},S(teams - i, 2));
    endfor
  else
    for i = 0:teams - 1
      fprintf(fid, '%u %f\n', S(teams - i,1), S(teams - i, 2));
    endfor
  endif

  fclose(fid);
endfunction
```