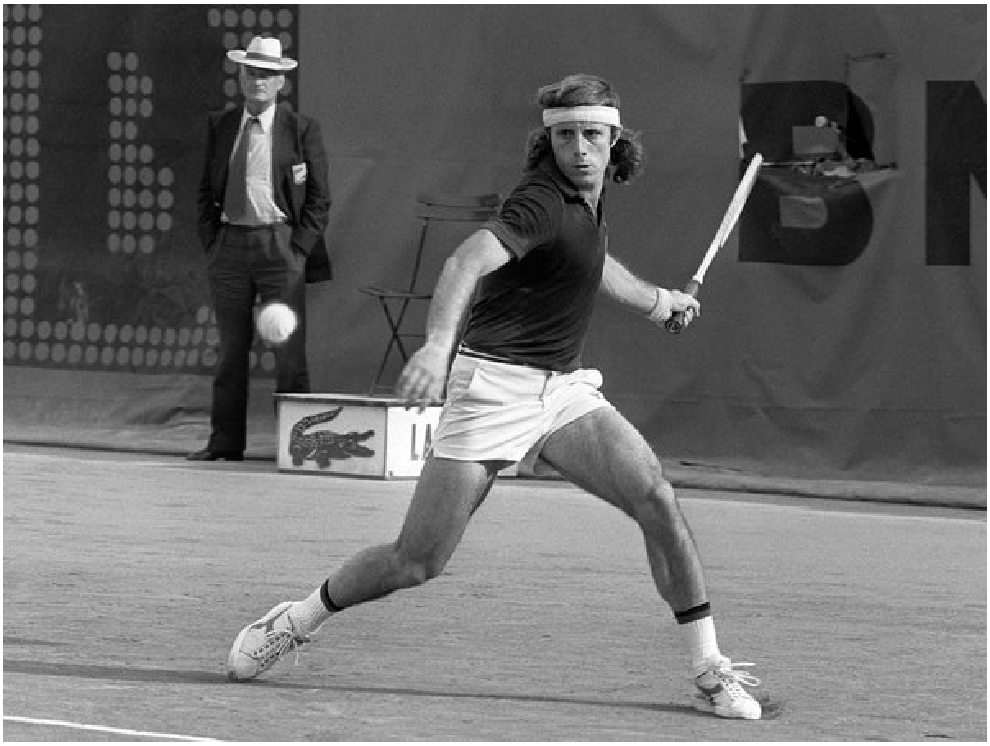
Métodos Numéricos
TP2

5 de octubre de 2015

Years Later for Guillermo Vilas, He's Still Not the One



Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com
Rodrigo Kapobel	695/12	rok_35@live.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Resumen: TODO
Keywords: TODO

Índice

1. Introduccion

El 25 de Mayo de 2015 el diario The New York Times publico un articulo titulado ”Years Later for Guillermo Vilas, He’s Still Not the One”, donde se repasa el rendimiento del tenista argentino durante los anios 1975/1976 y se discute el calculo del ranking de la ATP en ese momento. Aunque hoy en día Vilas es un icono del tenis argentino, nunca logro estar en la cima del ranking de la ATP.

Figura 1: Guillermo Vilas after winning a tournament in Stockholm in 1975. A journalist has asserted that Vilas deserved to be ranked No. 1 during that year.

En 2016, un grupo de investigadores argentinos decidió analizar el ranking de la ATP en 1975 y 1976 para determinar si Vilas debio haber sido numero 1. Dado que los rankings no se actualizaban constantemente en ese momento, los investigadores mostraron que de haberse actualizado de forma periódica, Vilas hubiese sido numero 1 por durante 7 semanas en 1975 y 1976.

Existen precedentes donde se actualizo un ranking de tenis de forma retroactiva. Este es el caso de la WTA, que determino que Evonne Goolagong Cawley debió haber sido numero 1 por dos semanas en 1976. Por esta razon el grupo de investigación argentino considera que revisar estos rankings no es un esfuerzo en vano. Cuando buscábamos los datos de la ATP en el 1975/1976, uno de los investigadores de este equipo que contactamos nos comento: “Es interesante tu decisión de indagar sobre el tema. Tal vez no estás al tanto del trabajo y lucha que estamos realizando contra la ATP, por el ranking de los 70 en el que perjudicaron a Vilas y muchísimos otros jugadores.”.

En ese momento, el calculo del ranking de la ATP era bastante rudimentario: “It was a system based on an average of a player’s results, and it often rewarded top players who played fewer tournaments. Vilas was a workhorse, which is how he managed not to reach No. 1 in the ATP rankings in 1977, when he won the French Open, the United States Open and 14 other tournaments.” [5].

Los métodos para calcular rankings no solo son relevantes para definir las posiciones de equipos y jugadores en eventos deportivos, sino que aparecen constantemente en todo tipo de situaciones donde se debe imponer algun tipo de orden. Este es el caso por ejemplo de los concursos docentes, donde se ponderan los diferentes antecedentes para decidir cual es el candidato *idoneo* para el puesto.

Otro caso sumamente relevante en cuanto algoritmos de rankeo es el de los motores de búsqueda. Los motores de búsqueda deben encontrar alguna forma de ordenar de forma relevante los sitios web que están relacionados con una consulta. El caso iconico es el de Google con su algoritmo PageRank. Los buscadores antes de 1990 eran sumamente rudimentarios, utilizaban algoritmos de rankeo vulnerables en el sentido que podían ser manipulados y no se explotaba gran parte de la estructura de la web. Esta fue una de las razones por las cuales una consulta no siempre devolvía resultados relevantes. Este fue el caso por ejemplo de algunos buscadores en ese momento como Yahoo! Search o AltaVista.

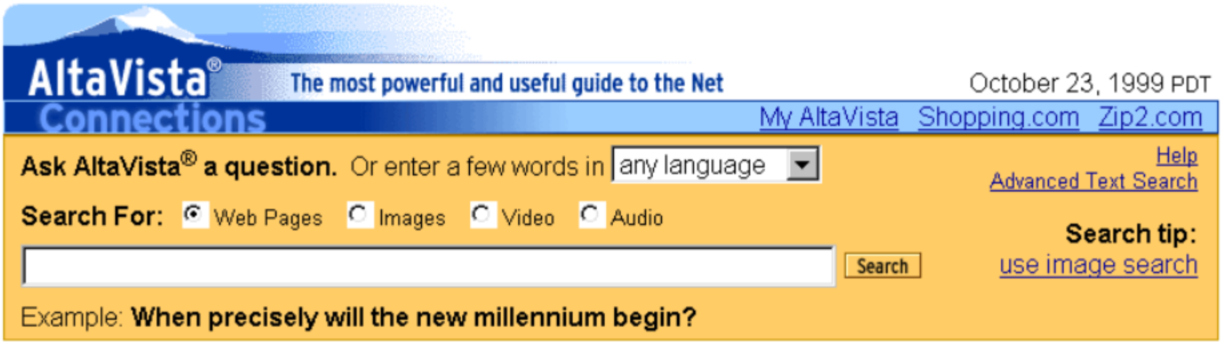


Figura 2: Sitio Web de Altavista, ano 1999.

El clásico paper de Brin y Page, “The anatomy of large-scale hypertextual Web search engine.” [3] explica brevemente el origen del motor de búsqueda de Google y del algoritmo PageRank. La idea es básicamente la siguiente, en primer lugar se implementa un crawler distribuido para poder solicitar y armar el grafo de la web. Las palabras de cada sitio son indexadas y guardadas en una base de datos. Al llegar una consulta al buscador, un programa busca la consulta en los indices de paginas. De esta forma llegamos a un conjunto de paginas que están relacionadas con la consulta. Luego, antes de devolverle al usuario los resultados, estas paginas son ordenadas utilizando el famoso algoritmo PageRank. Este algoritmo se basa en la idea de que para medir la relevancia de un sitio se puede usar como proxy la cantidad de sitios que tienen un link al mismo. Para evitar que un usuario malintencionado manipule los resultados del mismo, la relevancia otorgada por un sitio web que linkea a otro es proporcional a su propia relevancia e inversamente proporcional a la cantidad de links (o grado de salida) del mismo.

El presente trabajo practico tendrá como objetivo implementar el algoritmo PageRank para luego utilizarlo para generar rankings de todo tipo, ya sea para ordenar la relevancia de paginas webs o generar rankings deportivos. PageRank es un algoritmo que basa su ranking en encontrar el autovector de una matriz de transiciones. A priori esto puede sonar complicado, pero luego mostraremos que en realidad es bastante simple y elegante. Dado que ordenar la relevancia de millones de sitios web no es un problema trivial, en la practica este problema se resuelve utilizando álgebra lineal y métodos numéricos. Una muy buena introducción teórica se puede encontrar en el trabajo de Bryan y Leise [4]. Otros autores como Kamvar et al. [7] han buscado otros enfoques y métodos para poder acelerar este algoritmo. La idea es encontrar una forma eficiente de poder computar este modelo, calibrando sus diferentes parámetros de modelado y convergencia para lograr un orden relevante.

Una vez planteado el procedimiento, experimentaremos con la complejidad temporal de los métodos implementados y evaluaremos los diferentes parámetros a calibrar. Finalmente concluiremos si según el algoritmo PageRank y nuestra matriz de transición Vilas efectivamente debió haber estado en la punta del ATP en 1975/1976. En caso afirmativo, sin dudas nos comunicaremos con la ATP.

Figura 3: Guillermo Vilas apoya este TP.

2. PageRank

2.1. Modelado para paginas web

El algoritmo PageRank fue ideado en un principio para buscar de darle alguna medida de relevancia a los sitios web en internet. El mismo tiene dos interpretaciones equivalentes, que serán expuestas a continuación.

El problema se modela a partir de un grafo $G(Web, Links)$ donde Web es el conjunto de sitios web y $Links$ es la cantidad de conexiones entre sitios. Consideremos que toda pagina web $u \in Web$ esta representada por un vértice y la relación entre paginas por un link con una arista. Una representación posible del grafo es mediante matrices de adyacencia. Definimos la matriz de adyacencia o conectividad $W \in \{0,1\}^{n \times n}$ de forma tal que $w_{ij} = 1$ si la pagina j tiene un link a la pagina i y $w_{ij} = 0$ en caso contrario. Por lo tanto, la cantidad de paginas a las que la pagina u apunta ($d_{out}(u)$) se puede calcular como $n_j = \sum_{i=1}^{n} w_{ij}$.

2.1.1. Propiedades

Sea x_j el puntaje asignado a la pagina o vértice $j \in Web$ y otra pagina $u \in Web$. La idea es buscar una medida que cumpla con las siguientes propiedades:

- La relevancia de todo sitio web es positiva.
- La relevancia de un sitio web debe aumentar a medida que mas sitios unicos lo apuntan.
- La relevancia derivada de otro sitio web debe depender de su propia relevancia. Es decir, es mas valioso que me linkee un sitio relevante que uno no relevante. En caso de no cumplirse esta propiedad, el ranking seria fácilmente manipulable al permitir que un usuario cree muchos sitios que linkeen a uno para darle relevancia.
- La relevancia de todos los sitios web debe sumar uno. De esta manera estamos ante una distribución de probabilidad de los sitios. Mas adelante veremos que al interpretar esto mediante Cadenas de Markov existe una interpretación directa: la relevancia se puede ver como la proporción del tiempo total que un usuario pasa en ese sitio.

Por lo tanto, estamos buscando una medida de relevancia tal que la importancia obtenida por la pagina u obtenida por el link de la pagina v sea proporcional a la relevancia de v e inversamente proporcional al grado de v. El aporte del link de v a u entonces es $x_u = x_v / n_v$. Luego, sea $L_k \subseteq Web$ el conjunto de paginas que tienen un link a la pagina k. Por lo tanto, la relevancia total de un sitio sera:

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

Notar que esta es de cierta manera una definición recursiva. La relevancia de un sitio u puede depender de la relevancia de un sitio v, y luego la de v puede depender de la de u. A priori calcular la relevancia de un sitio puede parecer sumamente complicado, pero luego veremos que al plantearlo como un sistema de ecuaciones esta dificultad per se ya no se presenta.

Definimos entonces una matriz de transición o adyacencia con pesos en las aristas $P \in \mathbb{R}^{n \times n}$ tal que $p_{ij} = 1/n_j$ si $w_{ij} = 1$ y $p_{ij} = 0$ en caso contrario. Luego, el modelo planteado en (1) para toda pagina web se puede expresar $Px = x$ donde $x \in \mathbb{R}^n$. Notar que esto es equivalente a encontrar el autovector de autovalor 1 tal que $x_i > 0$ y $\sum_{n=1}^{n} x_i = 1$. Notar que si logramos probar que bajo ciertas condiciones nuestra matriz de transición tiene autovalor 1, el signo de todos los elementos de un autovector es el mismo y la dimension del autoespacio es 1 ya tenemos un ranking valido. Esto se debe a que cualquier autovector puede ser reescalado a uno de norma unitaria con $x_i \geq 0$.

2.1.2. Existencia y Unicidad

Bryan y Leise [4] analiza y prueba las condiciones bajo las que podemos garantizar que:

- La matriz de transición tiene autovalor 1.
- La dimension del autoespacio asociado al autovalor 1 es 1. Es deseable que el ranking asociado a una matriz de transición sea único.
- El signo de todos los elementos del autovector asociado al autovalor 1 es el mismo.

Veamos bajo que condiciones nuestra matriz de transición cumple con estas propiedades:

Definición Una matriz cuadrada se llama estocástica por columnas si todos sus elementos son positivos y la suma de cada columna es igual a 1.

A partir de esta definición se puede probar la siguiente proposición:

Proposición 2.1 *Toda matriz estocástica por columnas tiene a 1 como autovalor.*

Esto significa que si no existen `dangling nodes`, es decir, vértices con $d_{out} = 0$, podemos garantizar que nuestra matriz de transición es estocástica por columnas.

Notar que bajo las condiciones actuales no podemos garantizar que si existe el autoespacio asociado al autovalor 1, el mismo tenga dimension 1. Intuitivamente, esto se debe a que el grafo de la web puede tener varias componentes conexas. Como comparamos sitios web que no están relacionados? Justamente la relación, ya sea directa o indirecta mediante transitividad me da algún tipo de relación de orden. Al no tener una relación de orden entre dos sitios web bien definida, es razonable que existan múltiples autovectores, es decir, rankings. Esto se puede ver claramente en la pagina 4 del paper de Bryan y Leise [4].

Por lo tanto, la idea es básicamente buscar algún tipo de transformación relevante de mi matriz de transición que me permita garantizar que no voy a tener `dangling nodes` y ademas que solo tenga una componente conexa, es decir, que el grafo sea conexo. Definimos la siguiente matriz de transición, donde $v \in \mathbb{R}^{n\times n}$, con $v_i = 1/n$ y $d \in \{0,1\}^n$, $d_i = 1$ si $n_i = 0$ y $d_i = 0$ como:

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

De esta manera, en caso de tener una pagina web que es un `dangling node`, le asignamos un link uniforme a todos los sitios web $u \in Web$. Una interpretación equivalente es tomar a la matriz de transiciones como la matriz que describe una Cadena de Markov, donde el link pesado representa la probabilidad de dirigirse de una pagina a la otra. Por lo tanto, esta transformación se puede interpretar como que que existe una probabilidad uniforme de ir de uno de estos sitios a cualquiera de la web. Esto normalmente se conoce como el `navegante aleatorio`.

Tambien podemos considerar la posibilidad de que el navegante aleatorio se dirija a una pagina web que no esta linkeada a la pagina a la que esta actualmente. Este fenómeno se conoce como teletransportación. Para incluirlo al modelo, tomemos un numero $c \in [0,1]$ y transformemos la matriz de transiciones de la siguiente manera, donde $\bar{1} \in \mathbb{R}^n$ es un vector tal que todos sus componentes valen 1:

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

Notar que en caso de tener $c = 1$, estamos en la matriz de transición sin teletransportación. Por otro lado, si $c = 1$ estamos en el caso donde solo hay teletransportación y no importa la estructura del grafo de la web.

Esta nueva matriz de transición, dado que es estocástica por columnas y no tiene `dangling nodes`, nos garantiza que la dimension del autoespacio generado por el autovector de autovalor 1 es unitaria. Solo nos falta mostrar que todo autovector tiene todos sus elementos del mismo signo. Es facil probar la siguiente proposicion:

Proposición 2.2 *Si la matriz M es positiva y estocástica por columnas, entonces todo autovector en $V_1(M)$ tiene todos sus elementos positivos o negativos.*

Por lo tanto, ya probamos la existencia del autovector de norma 1 asociado al autovalor 1 de la matriz de transición transformada. El siguiente lema nos garantiza su unicidad. Su respectiva demostración se encuentra nuevamente en la pagina 7 del paper de Bryan y Leise [4].

Lemma 2.3

Si M es positiva y estocástica por columnas, entonces $V_1(M)$ tiene dimension 1.

2.2. Modelado para Tenis

Hacer referencia al paper de Govan et al. Explicar existencia y unicidad haciendo referencia a las pruebas de modelado de paginas web.

2.3. Representacion del grafo

Ya hemos demostrado las condiciones necesarias para poder obtener el autovector asociado al autovalor dominante de una matriz de Markov. Ahora debemos proceder a calcular el mismo. Para esto, tenemos que tener en cuenta las cualidades del sistema y el método de resolución del algoritmo. Recordemos que en general, el grafo que representa la web tenderá a ser desconexo y muy grande, es decir, que podrán existir dos o mas rankings diferentes. Por lo tanto la matriz de transiciones puede ser muy esparsa e inclusive puede suceder que una página no tenga links de salida, dando lugar a dangling nodes. Para solucionar estos inconvenientes, con lo visto anteriormente disponemos de dos soluciones. Para los dangling nodes, la solución consiste en sumar una columna con probabilidad 1/n a la columna de ceros, esto en si, se puede interpretar como la probalidad de navegación aleatoria que previamente describimos. Aunque con esto no solucionamos el problema de la esparsidad de la matriz en si y el de poder tener mas de un ranking diferente. Para esto último, se agregó la matriz de probabilidad de teletransportación.

Dada esta definición, la matriz de transiciones resultante no es esparsa. Para sistemas muy grandes, esto puede resultar contraproducente a la hora de obtener el autovector asociado, dado que la complejidad espacial y temporal aumenta considerablemente con la cantidad de información representada en la matriz. Sin embargo existe un resultado que podremos utilizar para mejorar la eficiencia del algoritmo en términos de complejidad temporal y espacial. El mismo se basa en la idea de Kamvar et al. [7, Algoritmo 1] para el calculo del autovector. Este resultado nos permite utilizar la matriz original de transiciones sin modificar en lo absoluto, pero si cambiando su representación, valiendonos de una buena estructura para almacenar las entradas de la misma.

Las cualidades de la matriz hacen que sea razonable intentar pensar en una forma de representar solo las entradas que no sean ceros, y dado que la matriz suele ser esparsa, la misma contendrá muchos ceros que podrían no ser representados. Para esto optamos por una de entre las 3 siguientes estructuras de representación:

- Dictionary of Keys (DOK)
- Compressed Sparse Row (CSR)
- Compressed Sparse Column (CSC)

De todas estas representaciones posibles, para este t.p optamos por CSR. Aún así no haremos una elección sin una justificación apropiada del porque consideramos que es la mejor para nuestro trabajo, dado que como en toda estructura de datos, siempre existen pros y contras. Nos encargaremos en lo que sigue de exponer estos detalles para dejar en claro nuestro punto de vista.

- Dictionary of Keys (DOK)

Consiste en un diccionario que mapea pares de fila-columa a la entrada. No se representan las entradas nulas. El formato es bueno para gradualmente construir una matriz esparsa en orden aleatorio, pero pobre para iterar sobre valores distintos de cero en orden lexicográfico. Uno construye típicamente una matriz en este formato y luego se convierte en otro formato más eficiente para su procesamiento.

- Compressed Sparse Row (CSR)

Pone las entradas no nulas de las filas de la matriz en posiciones de memoria contiguas. Suponiendo que tenemos una matriz dispersa no simétrica, creamos vectores: uno para los números de punto flotante (val), y los otros dos para enteros (col_ind, row_ptr). El vector val almacena los valores de los elementos distintos de cero de la matriz, de izquierda a derecha y de arriba hacia abajo. El vector col_ind almacena los índices de columna de los elementos en el vector val. Es decir, si $val(k) = a_ij$ entonces $col\_ind(k) = j$. El vector row_ptr almacena los lugares en el vector val que comienza y termina una fila, es decir, si $val(k) = a_ij$ entonces $row_ptr(i) \leq k \leq row\_ptr(i+1)$. Por convención, se define $row\_ptr(n+1) = nnz$, en donde nnz es el número de entradas no nulas en la matriz. Los ahorros de almacenamiento de este enfoque es significativo. En lugar de almacenar elementos n^2, solamente necesitamos $2nnz+n$ lugares de almacenamiento.

Veamos con un ejemplo como seria la representacion:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

Es una matrix de 4x4 con 4 entradas no nulas. Luego:
val = [5 8 3 6] row_ptr = [0 0 2 3 4] col_ind = [0 1 2 1]

- Compressed Sparse Column (CSC) La idea es analoga a CSR, pero la compresion se hace por columnas es decir, si CSR comprime A, CSC comprime A^t

Sobre la matriz definida para CSR, con CSC obtenemos lo siguiente
val = [5 8 6 3] col_ptr = [0 1 3 4 4] row_ind = [1 1 2 3]
Todos los resultados anteriores permiten evitar representar valores nulos. El motivo de nuestra elección se debe a que CSR ofrece una buena representación de las filas de la matriz y es más eficiente a la hora de hacer operaciones del tipo A*x (matriz-vector) que es lo que nos interesa en el método de la potencia que realiza pageRank. CSC en cambio, es efectiva para el producto x*A (vector-matriz) dado que la misma ofrece una mejor representación de las columnas. En contra partida, tanto CSR como CSC, no permiten construcción incremental aleatoria, que si ofrece DOK, es decir, que cambios a la esparsidad de la matriz son costosos. En general están pensadas para ser estáticas, pero esto no es un inconveniente en nuestro caso, dado que no se realizaran cambios en la esparsidad de la matriz durante el proceso.

En el presente trabajo utilizaremos la idea de Kamvar et al. [7, Algoritmo 1] para el calculo del autovector valiendonos de nuestra estructura de representación elegida y compararemos los resultados con el algoritmo standard para mostrar que al final de cuentas, si el sistema es muy grande y esparso, puede resultar muy beneficioso en terminos de complejidad espacial y temporal.

2.4. Computo: Método de la Potencia (MAURO)

2.4.1. Eliminacion Gausiana

Antes de introduciir el metodo de la potencia, en principio podriamos querer calcular este autovector utilizando el metodo de elimnacion gausiana. Sin embargo, de ser posible seria de orden cubico... blah blh blah

Esta seccion solo la pongo para que la consideres. Se podra hacer eliminacion gausiana con pivoteo para (P-I)x = 0? Igual si es posible es de orden cubico, con la web de millones de paginas se te va al carajo. Es solo para enriquecer la discusion.

2.4.2. Correctitud

La idea básicamente es generar la secuencia $x_k = Px_{k-1}$ y tomar $k \to \infty$. Se puede probar que para este caso no importa el valor inicial que asignemos a la secuencia x_0, el vector converge al autovector asociado al mayor autovalor de P. Se puede probar que todo autovalor λ de P satisface que $|\lambda| < 1$.

Otra propiedad interesante es que el método de la potencia va a converger de forma asintotica siguiendo $\|Px_k - q\|_1 \approx |\lambda_2| \, \|x - q\|_1$ donde λ_2 es el segundo autovalor mas grande de P. **Mauro revisa esto y fijate si sirve y se puede hacer algún criterio de parada copado.**

2.4.3. Valor Inicial

Elijo uno uniforme (todos con la misma relevancia y norma 1), o uno bien contra las esquinas (obvio que no)? Con esto hay que experimentar un cachito igual.

2.4.4. Criterio de Parada

Diferencias entre normas 1 o hay algo mejor?

2.4.5. Complejidad

Cual es la complejidad? Discutir que para deportes mucho no importa, pero para paginas web si.

2.4.6. Otras propiedades

Mirar esto antes de resolver lo siguiente...

Sin embargo, en Kamvar et al. se propone una forma alternativa de computar la secuencia. Este resultado debe ser utilizado para mejorar el almacenamiento de los datos. Esta relacionado con el punto de representacion del grafo esto? Ni idea.

... seguir

LO SIGUIENTE ES PARA MAURO. HACELO, NO SEAS PAJERO. SI, TE LO DIGO DENUEVO PORQUE SE QUE LO VAS A TRATAR DE EVITAR. NO TE PEDI MUCHO. ESTO ES PARA EL FINAL IGUAL, CUANDO YA TERMINASTE TODO LO DE ARRIBA.

- Demostrar que los pasos del Algoritmo 1 propuesto en Kamvar et a. son correctos y computan P_2x.
- Establecer una relacion con la proporción entre $\lambda_1 = 1$ y $|\lambda_2|$ para la convergencia de PageRank.

3. Experimentación

3.1. PageRank

3.1.1. Complejidad

tiempo de computo en funcion de size del grafo, eje x, cantidad de sitios web, eje y, tiempo en ms a convergencia.

3.1.2. Casos Patologicos

Caso particular chiquito, pagina 3. Fijate el parrafo que arranca en A simple apprroach...... y despues This approach ignores that... La idea es armar el mismo grafo y mostrar el mismo ejemplo jaja

3.2. Paginas Web

3.2.1. Comparacion PageRank vs In-Deg (RODRI)

Comparar solo los rankings, nada de complejidad. Podes mencionar que In-Deg usa un algoritmo $\mathcal{O}(n \times log(n))$, pero nada mas. Comparar top 10 con los dos y discutir diferenciias.

3.2.2. Manipulacion

Pagina 5, ejercicio 1. La idea es que plantees un caso de un tipo que quiere manipular el ranking, mostra que aunque agregues miles de nuevas paginas apuntando no podes hacer demasiado, hacelo en funcion de la cantidad de paginas que agregas?

Se puede manipular entonces o no? Agarra, en el eje x pone cantidad de sitios web que apuntan solamente al sitio u que le quiero subir el ranking, y en el eje y el ranking de ese sitio. Fijate que aumenta, y fijate si podes hacer algun tipo de curva de nivel con c (cuanto mayor c, mas manipulable es la cosa). Citar el paper de Sergei y Brin, que dicen que hacen promedios de muchas cosas en la practica para evitar este problema. Usan muchos criterios promediados.

3.3. Ranking ATP

3.3.1. Ranking ATP oficial vs Ranking PageRank/In-Deg (RODRI)

Discutir nuevamente diferencias. Un poco de chamullo, el que yo te dije rodri, sobre el cambio de calculo en el ranking del ATP y la retroactividad, etc. Acordate de escribir en la seccion del desarollo Rodri como se arma la matriz de transicion para los deportes y cual fue la motivacion/idea.

3.3.2. Eleccion del factor de 'teletransportacion' c (RODRI)

probar relevancia a medida que cambias ese valor = 0.85, creo que c.

Citar paper de google, que usan 0.85. Discutir que si c es uno, ignoras la estructura del grafo al hacer el ranking, todos rankean igual.

Pagina 6.... This is the ultimately egalitarian case: the only... blah. La idea es jugar con c aca, como dije arriba. Es un buen exp, hay que pensar bien como graficarlo y que quede lindo, creo que es facil.

3.4. Metodo de la Potencia

3.4.1. Representacion de la Matriz de Transicion (RODRI/FEDE)

Este experimento lo pueden hacer directo o usando al PageRank. Si pueden, implementen todas las representaciones de matrices y luego comparen el tiempo de computo del producto N veces. Comparen la matriz normal vs el resto. Discutan que en paginas web la cantidad de vertices del grafo se va al carajo, pero para deportes es super acotada, asi que la eleccion de estructura no afecta tanto.

Aca podes argumentar que lo que domina al metodo de la potencia es la cantidad de productos, asi que no hace falta probar PageRank directo. Igual si queres metelo con pagerank de una, a fin de cuentas es lo mismo.

3.4.2. Evolucion de la norma entre iteraciones

Como va evolucionando la norma manhattan entre dos iteraciones sucesivas. Eje x, iteraciones, eje y, norma manhattan.

3.4.3. Convergencia

Aca tienen que calcular el vector posta, y luego tomar algun tipo de norma. En el eje x van a tener la cantidad de iteraciones, y en el eje y van a tener la norma de x* - x_actual.

3.4.4. Eleccion del x_0

Aca pongan que te conviene arrancar con una buena 'adivinanza' de la solucion, asi se acerca mas rapido. Muestren la cantidad de iteraciones a la convergencia (norma manhattan ¡epsilon) dependiendo de la distanciia de la solucion inicial a la solucion posta. Si arranco con la posta de una, converge de una. Si arranco con una sol asquerosa inicial, tarda mas iteraciones en cumplir nuestro epsilon.

Mostrar dos instancias, una donde arrarnco desde el valor inicial donde todos tienen 1/n y otra donde una tiene 1 y el resto 0, mostrar la cantidad de pasos y como evoluciona la norma.

4. Conclusiones

Una vez que ya este todo lo leo y escribo esto bien a los pedos, incluyendo la caratula.

5. Apéndice A: Enunciado

Métodos Numéricos
Segundo Cuatrimestre 2015
Trabajo Práctico 2

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ohhh solo tiran π-edras...

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 $Web = \{1, \dots, n\}$ 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

$$x_k = \sum_{j\in L_k} \frac{x_j}{n_j}, \qquad k = 1, \dots, n. \tag{1}$$

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 (1) 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. [7, 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 (1). Probabilísticamente, la componente x_j del vector solución (normalizado) del sistema $P_2x = x$ 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 $P_2x = x$, 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. [7], 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. [6] se hace una breve reseña de dos ellos, y los autores proponen un nuevo método basado en el algoritmo PageRank que denominan GeM[1]. 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. [6] 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., [6, Sección GeM Ranking Method]) es el siguiente[2]:

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 $A^t \in \mathbb{R}^{n\times n}$

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

   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

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

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

[2] Notar que en artículo, Govan et al. [6] lo definen sobre la traspuesta. La definición y las cuentas son equivalentes, simplemente se modifica para mantener la consistencia a lo largo del enunciado.

4. Tomar $P = H^t$, y aplicar el método PageRank como fue definido previamente, siendo π la solución a la ecuación $P_2x = x$. 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. [7] 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. [7, Algoritmo 1] se propone una forma alternativa para computar $x^{(k+1)} = P_2x^{(k)}$. 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. [7] 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. [6]:

```
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.

6. Apéndice B: Código

6.1. system.cpp

```
#include <iostream>
#include <math.h>
#include <fstream>
#include <sstream>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <new>
#include <regex>
#include <iterator>
#include "sparseMatrix.h"

using namespace std;

struct dataNode {
    //dataNode(int n) : node(n), edgesCount(0) {} // no lo pude compilar con listas de inicializacion...

    int node;
    int edgesCount;
};

struct matchesStats {
    //matchesStats(int t) : team(t), matchesWin(0), matchesDefeat(0), pointsScored(0), pointsReceived(0) {}

    int team;
    int matchesWin;
    int matchesDefeat;
    int pointsScored;
    int pointsReceived;
};

//webs / sports
Matrix<double> pageRank(Matrix<double>& M, double c, double d, vector<int>& nodesCount);
Matrix<double> enhancementPageRank(Matrix<double>& M, double c, double d, vector<int>& nodesCount);

//webs
void in_deg(vector<dataNode>& nodesCount);

//sports
void basic_sort(vector<matchesStats>& stats);

//out data
void saveResultPageRank(FILE * pFile, Matrix<double>& data);
void saveResultInDeg(FILE * pFile, vector<dataNode>& data);
void saveResultBasicSort(FILE * pFile, vector<matchesStats>& data);

//utils
double uniform_rand(double a, double b);

const int number_line = 3;

```

```
int main(int argc, char** argv) {

    if (argc < 6) {
        printf("Usage: %s algorithm: 0 (pageRank) 1 (alternative) / teletransportation probability / instance type: 0 (web pages) 1 (deportes) / input file / tolerance / result (optional)\n", argv[0]);
        return 0;
    }

    ifstream inputFile(argv[4]);

    if (!inputFile.good()) {
        printf("Non-existant input file.\n");
        return 0;
    }

    FILE * pFile = NULL;

    if (argc == 7) {
        pFile = fopen(argv[6],"w");
        if (pFile == NULL){
            printf("can't open output file.\n");
            return 0;
        }
    }

    cout << "alg: " << (*argv[1] - '0') << endl;
    int alg = (int) (*argv[1] - '0');
    if (alg != 0 && alg != 1) {
        printf("Error: Invalid algorithm.\n");
        return 0;
    }

    cout << "c: " << atof(argv[2]) << endl;
    double c = (double) atof(argv[2]);
    if (c <= 0 || c > 1) {
        printf("Error: Invalid teletransportation probability.\n");
        return 0;
    }

    cout << "inst: " << (*argv[3] - '0') << endl;
    int inst = (int) (*argv[3] - '0');
    if (inst != 0 && inst != 1) {
        printf("Error: Invalid instance type.\n");
        return 0;
    }

    cout << "tolerance: " << atof(argv[5]) << endl;
    double e = (double) atof(argv[5]);
    if (e <= 0 || e > 1) {
        printf("Error: Invalid tolerance.\n");
        return 0;
    }

    string line;
    getline(inputFile, line);

    if (inst == 0){
        int nodes = 0;
```

```
        int edges = 0;

        sscanf(line.c_str(),"%d %d", &nodes, &edges);
        cout << "nodes: " << nodes << " edges: " << edges << endl;

        if (alg == 0){
            Matrix<double> M(nodes, nodes);

            vector<int> nodesCount(nodes);

            for(int i = 0; i < nodes; i++) {
                nodesCount[i] = 0;
            }

            int i = 0;
            while(i < edges){
                int node_from = 0;
                int node_to = 0;

                getline(inputFile, line);
                sscanf(line.c_str(),"%d %d", &node_from, &node_to);
                //cout << "node_from: " << node_from << " node_to: " << node_to <<
                    endl;

                nodesCount[node_from-1] += 1;
                M(node_to-1, node_from-1) = 1;
                i++;
            }

            Matrix<double> res = pageRank(M, c, e, nodesCount);
//          Matrix<double> res = enhancementPageRank(M, c, e, nodesCount);

            cout << "page rank result: \n" << endl;
            res.printMatrix();

             if (argc == 7) {
                saveResultPageRank(pFile, res);
             }
        }else{
             // group algorithm webs
            vector<dataNode> nodesCount(nodes);

            for(int i = 0; i < nodes; i++){
                dataNode nod;
                nod.node = i+1;
                nod.edgesCount = 0;
                nodesCount[i] = nod;
            }

            int i = 0;
            while(i < edges){
                int node_from = 0;
                int node_to = 0;

                getline(inputFile, line);
                sscanf(line.c_str(),"%d %d", &node_from, &node_to);
                //cout << "node_from: " << node_from << " node_to: " << node_to <<
                    endl;

```

```
                    dataNode nod = nodesCount[node_from -1];

                    nod.edgesCount += 1;

                    nodesCount[node_from -1] = nod;

                    i++;
                }

                in_deg(nodesCount);

                if (argc == 7) {
                    saveResultInDeg(pFile, nodesCount);
                }
            }
        }else{
            int teams = 0;
            int matches = 0;

            sscanf(line.c_str(),"%d %d", &teams, &matches);
            cout << "teams: " << teams << " matches: " << matches << endl;

            int day = 0;
            int local = 0;
            int visitor = 0;
            int local_score = 0;
            int visitor_score = 0;

            if (alg == 0){
                // page rank sports
                Matrix<double> M(teams, teams);

                vector<int> totalAbs(teams);

                for(int i = 0; i < teams; i++) {
                    totalAbs[i] = 0;
                }

                int i = 0;
                while(i < matches){
                    getline(inputFile, line);
                    sscanf(line.c_str(),"%d %d %d %d %d", &day, &local, &local_score, &
                        visitor, &visitor_score);

                    int abs_score = abs(local_score-visitor_score);
                    if (local_score > visitor_score) {
                        M(local-1, visitor-1) += abs_score;
                        totalAbs[visitor-1] += abs_score;
                    }else if(visitor_score > local_score) {
                        M(visitor-1, local-1) += abs_score;
                        totalAbs[local-1] += abs_score;
                    }

                    i++;
                }

                Matrix<double> res = pageRank(M, c, e, totalAbs);

                cout << "gem result: \n" << endl;
```

```
            res.printMatrix();

            if (argc == 7) {
                saveResultPageRank(pFile, res);
            }
        }else{
            // group algorithm sports
            vector<matchesStats> stats(teams);

            for(int i = 0; i < teams; i++){
                matchesStats teamStats;
                teamStats.team = i+1;
                teamStats.matchesWin = 0;
                teamStats.matchesDefeat = 0;
                teamStats.pointsScored = 0;
                teamStats.pointsReceived = 0;
                stats[i] = teamStats;
            }

            int i = 0;
            while(i < matches){
                getline(inputFile, line);
                sscanf(line.c_str(),"%d %d %d %d %d", &day, &local, &local_score, &
                    visitor, &visitor_score);

                matchesStats localStats = stats[local-1];
                matchesStats visitorStats = stats[visitor-1];

                if (local_score > visitor_score) {
                    localStats.matchesWin += 1;
                    visitorStats.matchesDefeat += 1;
                }else if(visitor_score > local_score) {
                    localStats.matchesDefeat += 1;
                    visitorStats.matchesWin += 1;
                }

                localStats.pointsScored += local_score;
                localStats.pointsReceived += visitor_score;
                visitorStats.pointsScored += visitor_score;
                visitorStats.pointsReceived += local_score;

                stats[local-1] = localStats;
                stats[visitor-1] = visitorStats;

                i++;
            }

            basic_sort(stats);

            if (argc == 7) {
                saveResultBasicSort(pFile, stats);
            }
        }
    }

    inputFile.close();

    if (pFile != NULL) fclose(pFile);

```

```
    //M.printMatrix();

    //Depending on input data, create a matrix with the input file and call rank with matrix a values

    return 0;
}

Matrix<double> pageRank(Matrix<double>& M, double c, double d, vector<int>& nodesCount) {
    srand(45);

    int n = M.rows();
    double dbl_n = M.rows();

    int j = 0;
    while(j < n){
        int i = 0;
        while(i < n){
            if(M(i, j) != 0){
                M(i, j) = M(i, j) / (double)nodesCount[j];
            }else if(nodesCount[j] == 0){
                M(i, j) = 1/dbl_n; // dangling node / undefeated team
            }
            i++;
        }
        j++;
    }

    Matrix<double> v(n, 1/dbl_n);

    Matrix<double> E(n, n, (1 - c)*1/dbl_n); // PRE: rows == columns

    //Salvo que sea c = 1 no tiene sentido usar Sparse Matrix
    Matrix<double> A = M*c + E;

    Matrix<double> x(n, 1, 1/dbl_n);

    //for (int i = 0; i < M.rows(); i++) {
    //    x(i) = uniform_rand(0, 1);
    //}

    Matrix<double> last_x(n);

    double delta = 0;

    do {
        last_x = x;
        x = A*x;
        delta = x.L1(last_x);
    }while (delta > d);

    printf("delta is %f\r\n", delta);

    return x;
}

Matrix<double> enhancementPageRank(Matrix<double>& M, double c, double d, vector<int>& nodesCount) {
```

```
    srand(45);

    int n = M.rows();
    double dbl_n = M.rows();

    int j = 0;
    while(j < n){
        int i = 0;
        while(i < n){
            if(M(i, j) != 0){
                M(i, j) = M(i, j) / (double)nodesCount[j];
            }
            i++;
        }
        j++;
    }

    SparseMatrix<double> A(M);

    SparseMatrix<double> x(n, 1/dbl_n);

    //for (int i = 0; i < M.rows(); i++) {
    //     x(i) = uniform_rand(0, 1);
    //}

    SparseMatrix<double> last_x(n);

    SparseMatrix<double> v(n, 1/dbl_n);

    double delta = 0;

    do {
        last_x = x;

        x = A*x;
        x = x*c;

        double w = last_x.norm1() - x.norm1();

        x = x +  v*w;

        delta = x.L1(last_x);
    }while (delta > d);

    printf("delta is %f\r\n", delta); //Deberia devolverse.

    return x.descompress();
}

void in_deg(vector<dataNode>& nodesCount) {
    sort(nodesCount.begin(), nodesCount.end(), [](dataNode a, dataNode b) {
        return b.edgesCount < a.edgesCount;
    });

    cout << "IN-DEG result \n" << endl;
    for (dataNode a : nodesCount) {
        cout << "node: " << a.node << " points: " << a.edgesCount << "\n";
    }
}
```

```
void basic_sort(vector<matchesStats>& stats) {
    sort(stats.begin(), stats.end(), [](matchesStats a, matchesStats b) {
        if (a.matchesWin - a.matchesDefeat != b.matchesWin - b.matchesDefeat) {
            return b.matchesWin - b.matchesDefeat < a.matchesWin - a.matchesDefeat;
        }else{
            return b.pointsScored - b.pointsReceived < a.pointsScored - a.
                pointsReceived;
        }
    });

    cout << "basic sort result \n" << endl;
    for (matchesStats a : stats) {
        cout << "team: " << a.team
        << " matches win: " << a.matchesWin << " matches defeat: " << a.matchesDefeat
        << " points scored: " << a.pointsScored << " points received: " << a.
            pointsReceived <<"\n";
    }
}

void saveResultPageRank(FILE * pFile, Matrix<double>& data) {
    int n = data.rows();
    int i = 0;

    while(i < n){
        fprintf(pFile, "%f\r\n", data(i));
        i++;
    }
}

void saveResultInDeg(FILE * pFile, vector<dataNode>& data) {
    for (dataNode a : data){
        fprintf(pFile, "%d %d\r\n", a.node, a.edgesCount);
    }
}

void saveResultBasicSort(FILE * pFile, vector<matchesStats>& data) {
    for (matchesStats a : data){
        fprintf(pFile, "%d %d %d %d %d\r\n", a.team, a.matchesWin, a.matchesDefeat, a
            .pointsScored, a.pointsReceived);
    }
}

double uniform_rand(double a, double b) {
    return ((b-a)*((double)rand()/RAND_MAX))+a;
}
```

6.2. matrix.h

```
/*
 * File:   matrix.h
 * Author: Federico
 *
 * Created on August 16, 2015, 9:54 PM
 */

#ifndef MATRIX_H
#define MATRIX_H

#include <algorithm>
#include <math.h>
#include <vector>
#include <stdio.h>

using namespace std;

// La matriz respeta la notacion de la catedra, es decir, el primer subindice
// es la fila y el segundo es la columna

template<class T>
class Matrix {
    public:
        Matrix();
        Matrix(int rows); // Columnas impllicitas (col = 1)
        Matrix(int rows, int col);
        Matrix(int rows, int col, const T& init);
        Matrix(const Matrix<T>& other);
        ~Matrix();

        Matrix<T>& operator=(const Matrix<T>& other);
        Matrix<T> operator*(const Matrix<T>& other);
        Matrix<T>& operator*=(const Matrix<T>& other);
        Matrix<T> operator+(const Matrix<T>& other);
        Matrix<T>& operator+=(const Matrix<T>& other);
        Matrix<T> operator-(const Matrix<T>& other);
        Matrix<T>& operator-=(const Matrix<T>& other);

        Matrix<T> operator*(const T& scalar);
        Matrix<T> operator/(const T& scalar);

        T& operator()(int a, int b);
        const T& operator()(const int a, const int b) const;
        T& operator()(int a);
        const T& operator()(const int a) const;

        //PRE are vectors
        double L1(const Matrix<T>& other);
        double norm1();

        int rows();
        int columns();

        int rows() const;
        int columns() const;

        void printMatrix();
```

```
    private:
        vector<vector<T> > _values;
        int _rows;
        int _columns;

};

template<class T>
Matrix<T>::Matrix()
    : _values(1), _rows(1), _columns(1)
{
    _values[0].resize(1);
}

template<class T>
Matrix<T>::Matrix(int rows)
    : _values(rows), _rows(rows), _columns(1)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(1);
    }
}

template<class T>
Matrix<T>::Matrix(int rows, int col)
    : _values(rows), _rows(rows), _columns(col)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(col);
    }
}

template<class T>
Matrix<T>::Matrix(int rows, int col, const T& init)
    : _values(rows), _rows(rows), _columns(col)
{
    for(int i = 0; i < rows; i++) {
        _values[i].resize(col, init);
    }
}

template<class T>
Matrix<T>::Matrix(const Matrix<T>& other)
    : _values(other._values), _rows(other._rows), _columns(other._columns)
{}

template<class T>
Matrix<T>::~Matrix() {}

template<class T>
Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
  if (&other == this)
    return *this;

  int new_rows = other._rows;
  int new_columns = other._columns;

  _rows = new_rows;
```

```
  _columns = new_columns;

  _values.resize(new_rows);
  for (int i = 0; i < new_columns; i++) {
      _values[i].resize(new_columns);
  }

  for(int i = 0; i < new_rows; i++) {
    for(int j = 0; j < new_columns; j++) {
      _values[i][j] = other(i, j);
    }
  }

  return *this;
}

template<class T>
Matrix<T> Matrix<T>::operator*(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    int innerDim = _columns; // Tambien podria ser other._rows

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = 0;
            for(int k = 0; k < innerDim; k++) {
                result(i,j) += _values[i][k] * other(k,j);
            }
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& other) {
    Matrix<T> result = (*this) * other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] + other(i,j);
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& other) {
    Matrix<T> result = (*this) + other;
```

```
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator-(const Matrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    Matrix<T> result(_rows, other._columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] - other(i,j);
        }
    }

    return result;
}

template<class T>
Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& other) {
    Matrix<T> result = (*this) - other;
    (*this) = result;
    return (*this);
}

template<class T>
Matrix<T> Matrix<T>::operator*(const T& scalar) {
    Matrix<T> result(_rows, _columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] * scalar;
        }
    }

    return result;
}

template<class T>
Matrix<T> Matrix<T>::operator/(const T& scalar) {
    Matrix<T> result(_rows, _columns);

    for(int i = 0; i < result._rows; i++) {
        for(int j = 0; j < result._columns; j++) {
            result(i,j) = _values[i][j] / scalar;
        }
    }

    return result;
}

template<class T>
T& Matrix<T>::operator ()(int a, int b) {
    return _values[a][b];
}

template<class T>
const T& Matrix<T>::operator ()(const int a, const int b) const {
    return _values[a][b];
```

```cpp
}

template<class T>
T& Matrix<T>::operator ()(int a) {
    return _values[a][0];
}

template<class T>
const T& Matrix<T>::operator ()(const int a) const {
    return _values[a][0];
}

template<class T>
double Matrix<T>::L1(const Matrix<T>& other) {
    Matrix<T> vectorSubs = *this - other;

    double res = 0;
    for(int i = 0; i < rows(); i++){
        res = res + abs(vectorSubs(i));
    }

    return res;
}

template<class T>
double Matrix<T>::norm1() {
    double res = 0;
    for(int i = 0; i < rows(); i++){
        res = res + abs(_values[i][0]);
    }

    return res;
}


template<class T>
int Matrix<T>::rows() {
    return _rows;
}

template<class T>
int Matrix<T>::columns() {
    return _columns;
}

template<class T>
int Matrix<T>::rows() const{
    return _rows;
}

template<class T>
int Matrix<T>::columns() const{
    return _columns;
}

template<class T>
void Matrix<T>::printMatrix() {
    for(int i = 0; i < _rows; i++) {;
        for(int j = 0; j < _columns; j++) {
```

```
                printf("%f ", _values[i][j]);
                // cout << _values[i][j] << " ";
            }
            cout << endl;
        }
        cout << endl;
}

#endif  /* MATRIX_H */
```

6.3. sparseMatrix.h

```
/*
 * File:   SparseMatrix.h
 * Author: Rodrigo Kapobel
 *
 * Created on August 21, 2015, 23:05 PM
 */

#ifndef SparseMatrix_H
#define SparseMatrix_H

#include <algorithm>
#include <math.h>
#include <vector>
#include <stdio.h>
#include <cmath>
#include "matrix.h"

using namespace std;

//CSR implementation for sparse matrix. For more information check the next link: https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_row_Storage_.28CRS_or_CSR.29

template<class T>
class SparseMatrix {
public:
    SparseMatrix();
    SparseMatrix(int rows); // column vector with 0's
    SparseMatrix(int rows, T value);
    SparseMatrix(vector<T>& values, vector<int>& iValues, vector<int>& jValues, int columns); // PRE: jValues::size == values::size & values of iValues[0..iValues::size-2] are indices of values & iValue[iValues::size-1] == values::size
    SparseMatrix(const SparseMatrix<T>& other); // compress matrix
    SparseMatrix(const Matrix<T>& other);
    ~SparseMatrix();

    SparseMatrix<T>& operator=(const SparseMatrix<T>& other);
    SparseMatrix<T> operator*(const SparseMatrix<T>& other); // performs a matrix-vector multiplication
    SparseMatrix<T>& operator*=(const SparseMatrix<T>& other); // performs a matrix-vector multiplication
    SparseMatrix<T> operator+(const SparseMatrix<T>& other); // performs a vector-vector sum
    SparseMatrix<T>& operator+=(const SparseMatrix<T>& other); // performs a vector-vector sum
    SparseMatrix<T> operator-(const SparseMatrix<T>& other); // performs a vector-vector subs
    SparseMatrix<T>& operator-=(const SparseMatrix<T>& other); // performs a vector-vector subs

    SparseMatrix<T> operator*(const T& scalar);
    SparseMatrix<T> operator/(const T& scalar);

    T& operator()(int a, int b);
    const T& operator()(const int a, const int b) const;
    T& operator()(int a);
    const T& operator()(const int a) const;
```

```
    Matrix<T> descompress();
    //PRE are vectors
    double L1(const SparseMatrix<T>& other);
    double norm1();

    int rows();
    int columns();
    int rows() const;
    int columns() const;
    void printSparseMatrix();

private:
    vector<T> _values;
    vector<int> _iValues;
    vector<int> _jValues;

    int _columns;
};

template<class T>
SparseMatrix<T>::SparseMatrix()
: _values(1), _iValues(1), _jValues(1), _columns(1)
{}

template<class T>
SparseMatrix<T>::SparseMatrix(int rows)
: _values(rows), _iValues(1), _jValues(rows), _columns(1)
{
    for(int i = 0; i <= rows; i++){
        _iValues.resize(i+1, i);
    }
}

template<class T>
SparseMatrix<T>::SparseMatrix(int rows, T value)
: _values(rows), _iValues(1), _jValues(rows), _columns(1)
{
    for(int i = 0; i <= rows; i++){
        _iValues.resize(i+1, i);
        if(i < rows){
            _values[i] = value;
        }
    }
}

template<class T>
SparseMatrix<T>::SparseMatrix(vector<T>& values, vector<int>& iValues, vector<int>&
    jValues, int columns)
: _values(values), _iValues(iValues), _jValues(jValues), _columns(columns)
{}

template<class T>
SparseMatrix<T>::SparseMatrix(const SparseMatrix<T>& other)
: _values(other._values), _iValues(other._iValues), _jValues(other._jValues),
    _columns(other._columns)
{}

template<class T>
```

```
SparseMatrix<T>::SparseMatrix(const Matrix<T>& other)
: _values(0), _iValues(0), _jValues(0), _columns(other.columns())
{
    int new_rows = other.rows();
    int new_columns = other.columns();

    for(int i = 0; i < new_rows; i++) {
        _iValues.resize(_iValues.size()+1, _values.size());
        for(int j = 0; j < new_columns; j++) {
            if (other(i, j) != 0) {
                _values.resize(_values.size()+1, other(i, j));
                _jValues.resize(_jValues.size()+1, j);
            }
        }
    }
    _iValues.resize(_iValues.size()+1, _values.size());
}

template<class T>
SparseMatrix<T>::~SparseMatrix() {}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator=(const SparseMatrix<T>& other) {
    if (&other == this)
        return *this;

    _values = other._values;
    _iValues = other._iValues;
    _jValues = other._jValues;
    _columns = other.columns();

    return *this;
}

template<class T>
SparseMatrix<T> SparseMatrix<T>::operator*(const SparseMatrix<T>& other) {
    // ASUME QUE LAS DIMENSIONES DAN
    SparseMatrix<T> result(rows());

    for (int i = 0; i < rows(); i++) {
        for (int j = _iValues[i]; j < _iValues[i+1]; j++) {
            result(i) += _values[j]*other(_jValues[j]);
        }
    }

    return result;
}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator*=(const SparseMatrix<T>& other) {
    SparseMatrix<T> result = (*this) * other;
    (*this) = result;
    return (*this);
}

template<class T>
SparseMatrix<T> SparseMatrix<T>::operator+(const SparseMatrix<T>& other) {
    SparseMatrix<T> result(rows());

```

```
    for(int i = 0; i < rows(); i++) {
        result(i) = _values[i] + other(i);
    }

    return result;
}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator+=(const SparseMatrix<T>& other) {
    SparseMatrix<T> result = (*this) + other;
    (*this) = result;
    return (*this);
}

template<class T>
SparseMatrix<T> SparseMatrix<T>::operator-(const SparseMatrix<T>& other) {
    SparseMatrix<T> result(rows());

    for(int i = 0; i < rows(); i++) {
        result(i) = _values[i] - other(i);
    }

    return result;
}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator-=(const SparseMatrix<T>& other) {
    SparseMatrix<T> result = (*this) - other;
    (*this) = result;
    return (*this);
}

template<class T>
SparseMatrix<T> SparseMatrix<T>::operator*(const T& scalar) {

    vector<T> resValues(_values.size());

    for(int i = 0; i < _values.size(); i++) {
       resValues[i] = _values[i] * scalar;
    }

    SparseMatrix<T> result(resValues, _iValues, _jValues, _columns);

    return result;
}

template<class T>
SparseMatrix<T> SparseMatrix<T>::operator/(const T& scalar) {
    vector<T> resValues(_values.size());

    for(int i = 0; i < _values.size(); i++) {
       resValues[i] = _values[i] / scalar;
    }

    SparseMatrix<T> result(resValues, _iValues, _jValues, _columns);

    return result;
}

```

```
template<class T>
T& SparseMatrix<T>::operator ()(int a) {
    return _values[a];
}

template<class T>
const T& SparseMatrix<T>::operator ()(const int a) const {
    return _values[a];
}

template<class T>
Matrix<T> SparseMatrix<T>::descompress() {
    Matrix<T> result(rows(), _columns, 0);
    for (int i = 0; i < rows(); i++) {
        for (int j = _iValues[i]; j < _iValues[i+1]; j++) {
            result(i, _jValues[j]) = _values[j];
        }
    }

    return result;
}

template<class T>
double SparseMatrix<T>::L1(const SparseMatrix<T>& other) {
    SparseMatrix<T> vectorSubs = *this - other;
    double res = 0;
    for(int i = 0; i < rows(); i++){
        res = res + abs(vectorSubs(i));
    }

    return res;
}

template<class T>
double SparseMatrix<T>::norm1() {
    double res = 0;
    for(int i = 0; i < rows(); i++){
        res = res + abs(_values[i]);
    }

    return res;
}

template<class T>
int SparseMatrix<T>::rows() {
    return _iValues.size()-1;
}

template<class T>
int SparseMatrix<T>::columns() {
    return _columns;
}

template<class T>
int SparseMatrix<T>::rows() const{
    return _iValues.size()-1;
}

template<class T>
```

```
int SparseMatrix<T>::columns() const{
    return _columns;
}

template<class T>
void SparseMatrix<T>::printSparseMatrix() {
    Matrix<T> descompressedMatrix = descompress();
    descompressedMatrix.printMatrix();
}

#endif  /* SparseMatrix_H */
```

Referencias

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

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

[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] Christopher Clarey. Years later for guillermo vilas, he's still not the one. *The New York Times*.

[6] 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.

[7] 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.