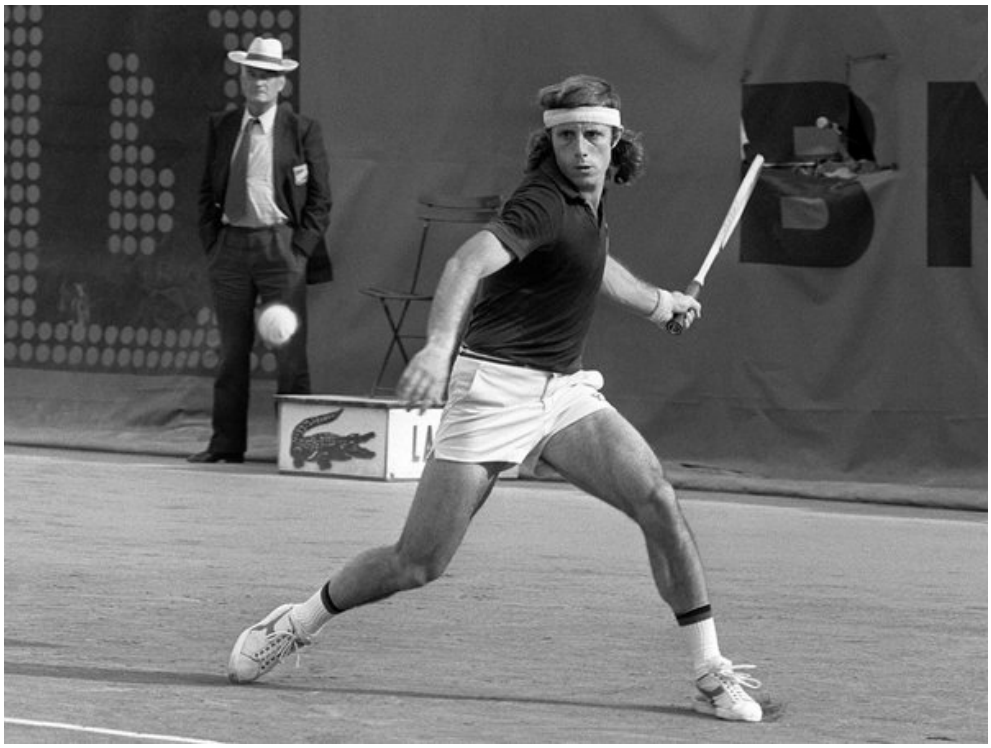
Métodos Numéricos
TP2

7 de noviembre 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: Existe una gran polémica en torno a los rankings de la ATP entre los años 1975 y 1977. Para enriquecer la discusión, en este trabajo se decidió analizar y recomputar estos rankings utilizando PageRank. En un primer momento, se analiza la motivación original de este algoritmo y se discuten sus diferentes cualidades numéricas y métodos de calculo. Se discute la calibración del algoritmo y su robustez ante intentos de manipulación. Finalmente, concluimos que si la ATP hubiese utilizado PageRank, efectivamente Vilas hubiese sido líder del ranking de la ATP en el año 1977, convirtiéndose en el primer argentino en lograr semejante hazaña.

Keywords: PageRank, Método de la Potencia, ATP Ranking, Vilas.

Índice

1. Introducción

El 25 de Mayo de 2015 el diario The New York Times publicó un artículo titulado "Years Later for Guillermo Vilas, He's Still Not the One", donde se repasa el rendimiento del tenista argentino durante los años 1975/1976 y se discute el calculo del ranking de la ATP en ese momento. Aunque hoy en día Vilas es un ícono del tenis argentino, nunca logró 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 y periodistas deportivos argentinos decidieron analizar el ranking de la ATP en 1975 y 1977 para determinar si Vilas debió haber sido número 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 número 1 durante 7 semanas entre 1975 y 1977.

Existen precedentes donde se actualizó un ranking de tenis de forma retroactiva. Este es el caso de la WTA, que determinó que Evonne Goolagong Cawley debió haber sido número 1 por dos semanas en 1976. Por esta razón el grupo de investigación argentino considera que revisar estos rankings no es un esfuerzo en vano. Cuando buscábamos los datos de la ATP entre 1975/1977, 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." [**?**].

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 algún 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 icónico 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.

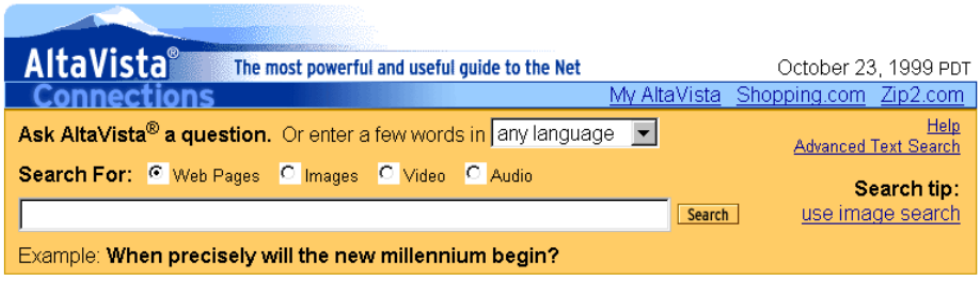


Figura 2: Sitio Web de Altavista, ano 1999.

El clásico paper de Brin y Page, "The anatomy of large-scale hypertextual Web search engine." [**?**] 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 índices de páginas. De esta forma llegamos a un conjunto de páginas que están relacionadas con la consulta. Luego, antes de devolverle al usuario los resultados, estas páginas 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 práctico tendrá como objetivo implementar el algoritmo PageRank para luego utilizarlo para generar rankings de todo tipo, ya sea para ordenar la relevancia de páginas webs o generar rankings deportivos. PageRank es un algoritmo que basa su ranking en la idea abstracta del navegante aleatorio. Este problema en general se modela con Cadenas de Markov, y en última instancia consiste en encontrar el autovector de una matriz de transiciones, que es similar a una matriz de adyacencia en teoría de grafos. A priori esto puede sonar complicado, pero luego mostraremos que en realidad es bastante simple y elegante.

Debido a su relevancia, PageRank es un algoritmo que ha sido ampliamente estudiado en la literatura. Una muy buena introducción teórica se puede encontrar en el trabajo de Bryan y Leise [**?**]. Otros autores como Kamvar et al. [**?**] 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. Otros autores como Govan et al. han buscado modelar la matriz de transiciones para aplicar este algoritmo en eventos deportivos [**?**]. Es importante mencionar que en este caso la complejidad temporal de los algoritmos utilizados para calcular PageRank no tendra tanta relevancia dado que la cantidad de equipos es acotada. La forma de calibrar el modelo también ha sido amplicamente estudiada, en especial para el caso de las páginas web. Un estudio muy interesante es el Constantine et al. de Microsoft Research, donde utilizando los registros del toolbar de Internet Explorer y un modelo basado en la distribución Beta generalizan la matriz de transiciones para poder modelar un parámetro de teletransportación variable [**?**].

Una vez planteado el procedimiento para computar el PageRank, experimentaremos con la complejidad temporal de los métodos implementados y evaluaremos los diferentes parámetros a calibrar. Finalmente concluiremos si según el algoritmo PageRank y nuestro modelado Vilas efectivamente debió haber estado en la punta del ATP en periodo 1975/1977. En caso afirmativo, sin dudas nos comunicaremos con la ATP.

Figura 3: Guillermo Vilas apoya este TP.

2. PageRank

2.1. La motivación original: páginas web

El algoritmo PageRank fue ideado en un principio para buscar una medida de relevancia con el objetivo de ordenar consultas sobre el grafo de la web. El mismo explota la estructura de este grafo, y como veremos mas adelante luego de explicar la idea general del algoritmo, se calcula buscando el autovector de norma unitaria que resuelve un sistema de la forma $Px = x$ con $P \in \mathbb{R}^{n \times n}$, es decir, el autovector asociado al autovalor 1. A su vez, existen dos interpretaciones equivalentes de este modelo, que serán expuestas a continuación.

2.1.1. Modelado

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 página web $u \in Web$ esta representada por un vértice y la relación entre páginas 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 página j tiene un link a la página i y $w_{ij} = 0$ en caso contrario. Por lo tanto, la cantidad de páginas a las que la página u apunta ($d_{out}(u)$) se puede calcular como $n_j = \sum_{i=1}^{n} w_{ij}$.

2.1.2. Propiedades

Sea x_j el puntaje asignado a la página o vértice $j \in Web$ y sea otra página $u \in Web$. La idea es buscar una medida de relevancia 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 únicos lo apuntan.
- La relevancia derivada de otro sitio web debe depender de su propia relevancia. Es decir, es mas valioso que nos 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. Esto nos permite a su vez utilizar muchos teoremas estudiados en procesos estocásticos. 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 página u obtenida por el link de la página 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 páginas que tienen un link a la página k. Por lo tanto, la relevancia total de un sitio sera:

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

Notar que esta es de cierta manera una definición autoreferencial. 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 página 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 $x_i \geq 0$ y $\sum_{i=1}^{n} x_i = 1$. 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 que cumple con estas propiedades puede ser reescalado a uno de norma unitaria con $x_i \geq 0$.

2.1.3. Existencia y Unicidad

Bryan y Leise [**?**] 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 nos 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 página 4 del paper de Bryan y Leise [**?**].

Por lo tanto, la idea es básicamente buscar algún tipo de transformación relevante de la 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 resultante 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 $d_i = 0$ en caso contrario, como:

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

De esta manera, en caso de tener una página web que es un `dangling node`, le asignamos un link con probabilidad o peso uniforme a todos los otros 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 página 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. Luego el famoso `navegante aleatorio` simplemente recorre el grafo utilizando estas probabiblidades.

Tambien podemos considerar la posibilidad de que el navegante aleatorio se dirija a una página web que no está linkeada a la página 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 $c = 0$ estamos en el caso donde solo hay teletransportación y no importa la estructura del grafo de la web al momento de calcular el ranking.

Observar además que la elección de este parámetro c no es trivial. En general, este parámetro se calcula haciendo análisis empíricos sobre los datos de navegadores y toolbars. Un trabajo interesante sobre este tema es el de Constantine et al. de Microsoft Research, donde utilizando los registros del toolbar de Internet Explorer y un modelo basado en la distribución Beta generalizan la matriz de transiciones para poder modelar un parámetro de teletransportación variable [**?**]. Para este trabajo, simplemente utilizaremos la elección original constante del trabajo de Page & Brin [**?**] y fijaremos $\alpha = 0{,}85$.

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 página 7 del paper de Bryan y Leise [**?**].

Lemma 2.3

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

Proposición 2.4 *Sea M una matriz real de n x n positiva y estocástica por columnas, y V el subespacio de $\mathbb{R}^n$ que consiste de aquellos vectores v tales que la suma de sus componentes sea 0, entonces $Mv \in V$ y $||Mv||_1 \leq \alpha||v||_1$ donde $\alpha = max_{1\leq j\leq n}|1 - 2min_{1\leq i\leq n}M_{ij}| < 1$.*

2.2. Tenis

2.2.1. Modelado

El modelo GeM es un método de rankeo basado en PageRank para rankear equipos en ligas deportivas planteado por Govan et al. en "Generalizing Google's PageRank to Rank National Football League Teams"[**?**]. Utilizaremos el mismo para analizar los rankings del ATP entre 1975 y 1977. A continuación explicaremos como los autores modelan la matriz de transiciones que utiliza el algoritmo. Antes de comenzar con las definiciones, vale aclarar que para mantener la consistencia de la notación utilizada en este paper trabajaremos sobre la transpuesta de la matriz que se usa en el paper de Govan et al.

Se representa una temporada como un grafo dirigido con n nodos. Cada nodo representa un participante o equipo y cada partido entre dos participantes representa una arista desde el perdedor al ganador igual a la diferencia de puntos obtenidos por cada participante en el partido.

La matriz A^t de adyacencias queda definida de la siguiente manera, donde cada w_{ij} representa la diferencia positiva del puntaje de cada participante en ese enfrentamiento.

$$A^t = \begin{cases} w_{ij} \text{ si el participante } j \text{ pierde con el participante } i \\ 0 \text{ en cualquier otro caso} \end{cases} \tag{2}$$

En caso de que un equipo pierda mas de una vez en el mismo torneo, será la suma de las diferencias de cada partido entre i y j. Esta es la mayor generalización en cuanto a PageRank.

Luego se define la matriz H de la siguiente forma:

$$H = \begin{cases} w_{ij} / \sum_{k=1}^{n} A^t_{kj} \text{ si hay un link de } j \text{ a } k \\ 0 \text{ en cualquier otro caso} \end{cases} \tag{3}$$

Finalmente definimos G a la matriz resultante de:

$$G = \alpha(A^t + ua^t) + (1 - \alpha)ev^t \tag{4}$$

Donde $0 < \alpha < 1$, v es un vector de probabilidades, a es tal que a_j es 1 si la columna j de H es el vector nulo y a_j es 0 en en cualquier otro caso. u es un vector de probabilidad de n x 1. e se define como un vector de n x 1 con todas sus entradas igual a 1. Luego, el vector que contendrá los puntajes de cada equipo será π tal que: $G\pi = \pi$.

Cada entrada H_{ij} de H se puede interpretar como la probabilidad de que el participante j pierda contra el participante i. Para los participantes invictos un simple ajuste que se propone es elegir un vector u donde todas las entradas son $1/n$. Esto significa cambiar la probabilidad de los invictos a que puedan perder contra cualquiera de los otros participantes (incluido si mismo) con probabilidad uniforme. El modelo básico utiliza α de la misma forma que en PageRank y v como vector de personalización del sistema. Una simple elección puede ser $v = (1/n)e$, con e al igual que antes, como un vector de n x 1 con todas sus entradas igual a 1. Aunque v ofrece mucha más flexibilidad. Podría usarse con el resultado estadistico de un ranking previo, aumentando así las probabilidades de que cierto participante gane el presente torneo. La elección del α determina la importancia de la matriz de personalización ev^t. A diferencia del caso de las paginas web, la elección de este parámetro puede ser difícil de medir empíricamente y en última instancia debe ser elegida por el organizador del torneo. En nuestro caso, decidimos elegir $\alpha = 0{,}85$.

2.2.2. Sistema de puntos y consideraciones

Para procesar los rankings, obtuvimos todos los partidos realizados entre 1975 y 1977 de todas las competencias a nivel internacional de tenis que sirven para clasificar al ATP [1]. (Australian Open, Indianapolis, Roma, Roland Garros, US Open, Wimbledon, Davis Cup, y muchos más) junto con los rankings de cada año. Uno a mitad de año y otro a finales de año.

Los datos están completos a nivel enfrentamientos y participantes, pero carecen de los puntajes que cada jugador obtuvo por cada partido ganado, por avanzar de ronda y/o ganar un torneo. Esta es una parte importante a aclarar del sistema de rankings de la ATP. Se obtienen puntos por cada partido ganado, ronda superada y torneo ganado, los cuales además varían por torneo en importancia, siendo los Grand Slam los de mayor jerarquía. Ademas, en esa época el sistema estaba basado en el puntaje promedio obtenido por el jugador.

Utiliza un sistema de defensa de puntos que fue variando con los años. En aquella época, la defensa se hacia por año. Es decir, los puntos realizados durante un año, se defienden al año siguiente.

Igualmente esto no supone una limitación grave para computar los rankings y comprobar si Guillermo Vilas fue realmente o no 1ro en al menos uno de esos años.

Además, dada la naturaleza del comportamiento de PageRank, no solo importa a cuantos contrincantes derrotó un participante dado, si no a quienes derrotó. Ésta es la primera consideración a tener en cuenta y una carta a favor que será utilizada por el algoritmo. Dado que no disponemos del sistema de puntuación real y no sabemos a priori cuantos partidos jugó exactamente cada participante, es esencial que además de la puntuación que definamos por cada partido, un jugador obtenga un extra por el rango del rival derrotado. Esto mismo se puede interpretar como que ese jugador ganó un partido importante, dado que los partidos importantes son los de instancias decisivas (cuartos, semifinales o una final), y generalmente, al que solo llegan los mejores jugadores.

Dadas estas consideraciones podemos definir un sistema de puntos con el que poder computar partidos. El mismo es muy sencillo. 3 puntos al ganador y 1 punto al perdedor.

Solo utilizaremos el ranking de fin de año, totalizando por los puntos acumulados en el mismo, obteniendo así un ranking por cada año.

Veremos luego en la experimentación, como este sistema resuelve el cálculo de los rankings de manera apropiada y explicaremos que es lo que sucede en cada año computado observando algunos detalles importantes y comparándolos con los rankings oficiales de la ATP para luego concluir si Vilas fue o no 1ro en algún momento entre 1975 y 1977 según PageRank.

[1] Los datos los obtuvimos de https://github.com/JeffSackmann/tennis_atp, un repositorio con todos los rankings historicos de la ATP. La veracidad de los mismos es discutible, pero fue lo mejor que conseguimos dado que los investigadores que contactamos no nos quisieron proveer los datos: "Lamentablemente, toda esa información está hoy en pleno ida y vuelta legal con la ATP y no puedo bajo ningún concepto filtrarla."

2.3. Computo: Método de la Potencia

Habiendo definido la matriz de transiciones como la matriz P_2 de la sección anterior, debemos hallar el vector de relevancias x que define en x_j la relevancia de la $j - esima$ pagina $\forall 0 \leq j \leq n$. Debido a la definición recursiva de este vector, x debe satisfacer que $P_2 x = x$ o lo que es equivalente, que x sea un autovector asociado al autovalor 1 (i.e $x \in V_1(P_2)$). Para este tipo de problemas es que contamos con el método de la potencia, un algortimo iterativo que nos devuelve una aproximación lineal al x deseado.

2.3.1. Correctitud

La idea detras de este algoritmo consiste en generar a partir de un vector inicial x_0, una secuencia $x_k = \frac{P_2 x_{k-1}}{||P_2 x_{k-1}||}$ que aproxime al autovector q (en nuestro caso el vector x de relevancias) asociado al autovalor 1. Siguiendo esta propuesta elegimos nuestro x_0 de manera que satifaga que todas sus componetes sean positivas y $||x_0||_1 = 1$, definiendolo además como $x_0 = q + v$, en donde q es el único vector unitario que satisface que $q \in V_1(P_2)$ y que posee todas sus componentes positivas; y v siendo un vector tal que la suma de sus componentes sea 0. De esta menera, dado que P_2 es una matriz positiva y estocástica por columnas, y x_0 un vector unitario con todas sus componentes también positivas, resulta que $P_2 x_0$ también es un vector unitario con todas sus componentes positivas. Esto nos facilita la secuencia de la siguiente forma $x_k = P_2 x_{k-1} = P_2^k x_0$, pues desde un principio $x_1 = P_2 x_0$ ($||P_2 x_0|| = 1$); luego $x_k = P_2^k x_0 = P_2^k q + P_2^k v = q + P_2^k v \implies P_2^k v = P_2^k x_0 - q$. Ahora bien, por la desigualdad de la proposición 2.4, $||P_2^k x_0 - q||_1 = ||P_2^k v||_1 \leq \alpha^k ||v||_1$, haciendo tender $k \to \infty$, $\alpha \to 0$, de lo que se deduce que $P_2^k x_0 - q \to 0$ o equivalentemente $P_2^k x_0 \to q$, es decir al x buscado.

2.3.2. Complejidad

En cuanto a complejidad temporal es evidente que el costo del algoritmo radica en cuantas operaciones elementales nos conlleva calcular $P_2^k x$, en especial si se requiere realizar una gran cantidad de iteraciones. Recordando como la definimos, $P_2 = cP_1 + (1-c)E$ donde E es la matriz con todos sus elementos iguales a $\frac{1}{n}$ y P_1 es una matriz positiva y estocástica por columnas, de lo que podemos concluir que todos los elementos de P_2 son estrictamente positivos (si $c \in (0,1)$). Esta conclusión no es muy alentadora, pues sólo tener que hacer $P_2 x$ tiene una complejidad espacial y temporal de $\Theta(n^2)$. No obstante, sabiendo que $||x||_1 = 1$, resulta que $P_2 x = (cP_1 + (1-c)E)x = cP_1 x + (1-c)Ex = cP_1 x + (1-c)e$ en donde e ahora es un vector con todos sus elementos iguales a $\frac{1}{n}$, luego la atención se concentra en el costo de $P_1 x$. Esta última matriz ya no sólo no tiene todos sus elementos estrictamente positivos, sino que suele tratarse de una matriz esparsa, ahorrandonos varias multiplicaciones (las que corresponderian a los elementos iguales a cero) aventajando en velocidad a la anterior situación; a su vez podremos aprovechar la redundancia de ceros para utilizar alguna estructura más eficiente a la hora de almacenar la matriz de la actual iteración. En conclusión dandonos una complejidad temporal total de $\mathcal{O}(k * n^2)$, pero un caso promedio notablemente menor.

2.4. Representación 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 disconexo 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. [**?**, 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 optaremos por 2 de entre las 3 siguientes estructuras de representación:

2.4.1. 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. Se suele utilizar para construir la matriz en este formato y luego se convierte en otro formato más eficiente para su procesamiento.

2.4.2. 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 esparsa 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]

2.4.3. Compressed Sparse Column (CSC)

La idea es análoga a CSR, pero la compresión 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. Para este trabajo práctico optaremos por CSR y DOK y las utilizaremos en conjunto. 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.

Debido a que la matriz involucrada en el proceso puede ser muy esparsa y en general con muchos nodos involucrados, las dimensiones pueden ser extremadamente grandes para trabajar con la representación usual de matriz, es decir, la matriz sin comprimir. Para tener una idea de esto, si el grafo posee 100 mil nodos y apenas 10 mil conexiones obtendriamos una matriz de 10 mil millones de posiciones. Si cada posición se representa con un double, tenemos 8 bytes por cada una. Esto daria un total de 80 mil millones de bytes, que dividido por 2^{30} bytes da un total de 74.5 gb que tendriamos que disponer de memoria ram para poder procesar la matriz. En este trabajo práctico vamos a trabajar con una matriz que es incluso casi 3 veces más grande y mucho más esparsa.

Queda claro que no parece viable siquiera intentar utilizar la representación estándard de matriz, al menos hasta disponer de un ordenador de la nasa. Por suerte como vimos, disponemos de dos estructuras de datos para poder solucionar este inconveniente. La manera de utilizarlas será DOK para crear la matriz estocástica y CSR para realizar el calculo de page rank dado que las operaciones involucradas serán matriz por vector y para esto convendrá tener una buena representación de las filas.

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

3. Experimentación

3.1. PageRank: Complejidad

En el caso de PageRank, la complejidad del algoritmo esta dictada por el tamaño del grafo que representa la red, con lo cual vamos a estudiar el tiempo de ejecución en función de dicho factor.

Para generar las instancias tomamos igual cantidad de nodos y de ejes, la primera tomando valor 30, cada instancia sucesiva aumentamos la cantidad de nodos en 15, hasta llegar a las 150 iteraciones. Esto lo hacemos utilizando un $\alpha = 0{,}85$ como factor de teletransportacion y un $\epsilon = 10^{-5}$ como factor de convergencia. En la matriz el tamaño esta dado en función de la cantidad de nodos, mientras que los ejes son utilizados en el paso inicial para determinar las probablidades iniciales, con lo cual la cantidad de ejes no tiene la misma importancia que la cantidad de nodos al calcular el método de la potencia.

El siguiente gráfico muestra los tiempos de convergencia según el tamaño, para suavizar el ruido se tomaron 5 muestras por cada instancia y se luego tomo el promedio.

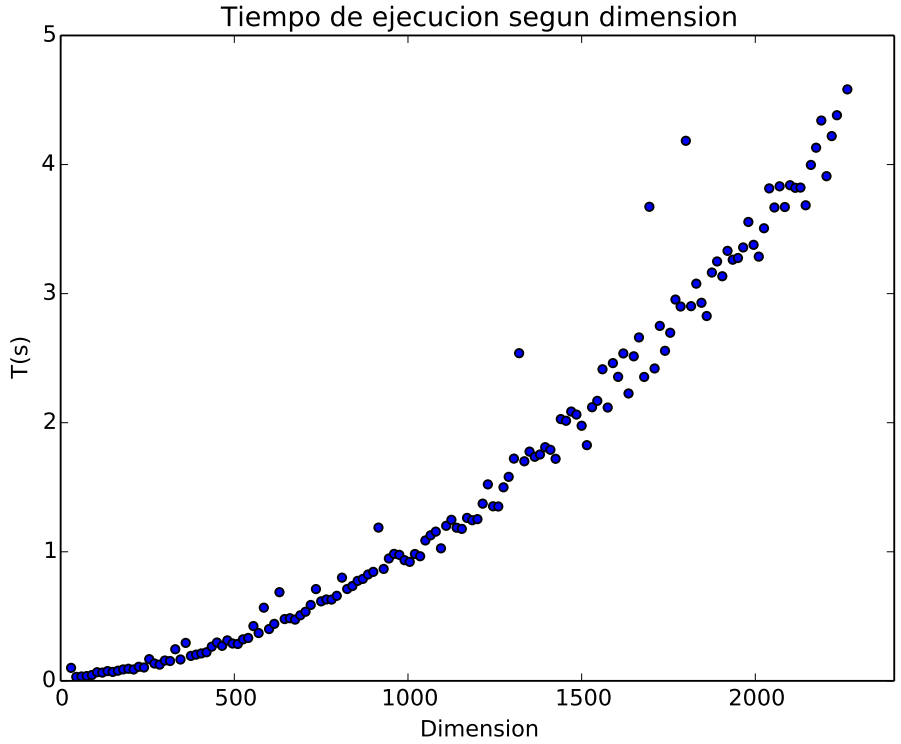


Figura 4: Tiempos de ejecución según cantidad de webs.

El método de la potencia hace k productos de matrices en $\mathcal{O}(n^3)$, donde k es la cantidad de iteraciones que toma cada instancia en converger y n es la cantidad de vértices del grafo. Como podemos observar, la tendencia es no lineal. Esto tiene sentido, dado que el producto de matrices es de orden cubico. Luego experimentaremos para ver si la dimension del grafo efectivamente tiene un gran impacto sobre la cantidad de iteraciones que necesita el método para converger. Sin embargo, a priori conjeturamos que la tendencia cubica de este gráfico esta dominada por el orden del producto entre matrices.

3.2. Convergencia

En este experimento buscamos estudiar la convergencia del método de la potencia. Para ello generamos dos instancias. La primera instancia fue generada con 2.500 vértices y 2.500 aristas. La segunda fue generada con 5000 vértices, utilizando las mismas aristas. Ambas instancias fueron generadas con un $\alpha = 0{,}85$. Como criterio de parada, se utilizo que la diferencia entre las normas 1 de dos vectores en sucesiones consecutivas sea menor a $\epsilon = 10^{-5}$. A continuación mostramos los resultados:

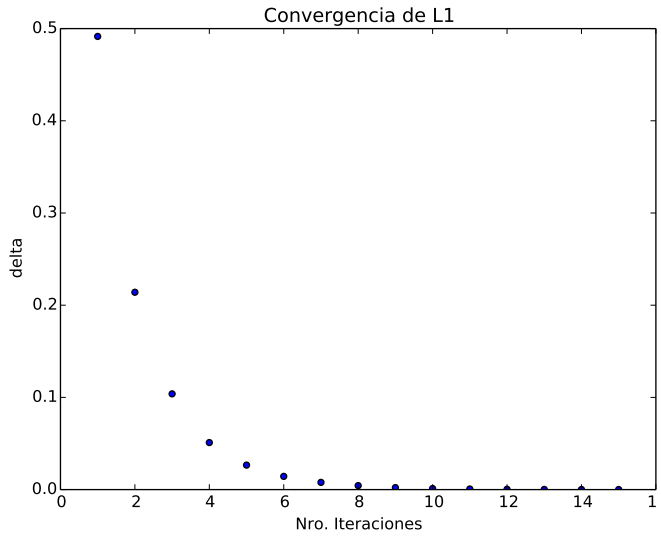


Figura 5: 2500 vértices

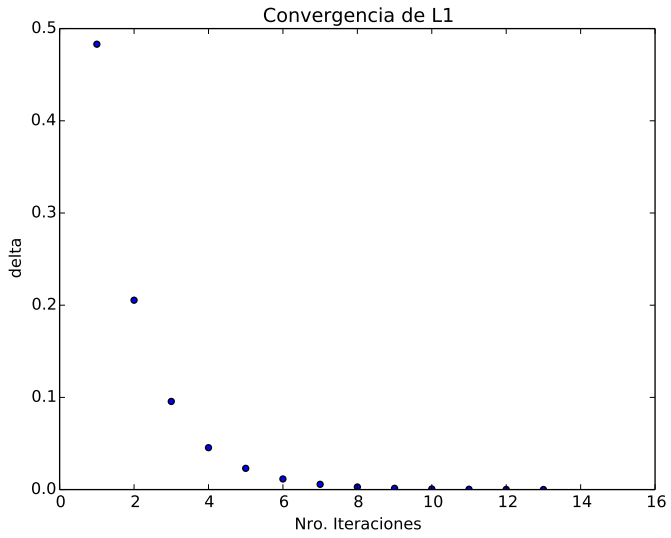


Figura 6: 5000 vértices

Como podemos observar en ambos gráficos, a medida que aumenta el numero de iteraciones el delta baja de forma monótona. A su vez, en general ambos algoritmos toman aproximadamente 13 iteraciones en converger. Notar lo siguiente, que la cantidad de vértices se mayor no implica necesariamente que el método de la potencia tome mas iteraciones en converger. De hecho, podemos observar que la instancia con 5000 vértices toma 13 iteraciones, mientras que con 2500 vértices toma 15 iteraciones. Esto apoya la hipotesis del punto anterior, donde argumentamos que la complejidad del metodo de la potencia estaba dominada por el orden del producto entre matrices y no por el cambio en la cantidad de iteraciones necesarias para que el método converja.

3.3. Comparación PageRank vs In-Deg

Consideremos el siguiente grafo que representa cierta web.

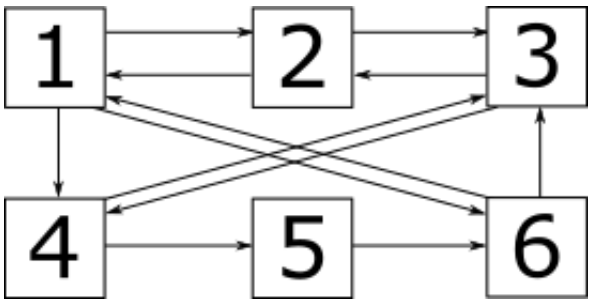


Figura 7: Caso de estudio

Computemos el ranking de esta web utilizando PageRank ($\alpha = 0{,}85$) y utilizando el orden por grado de entrada.

Nro. Web	PageRank	Por grado
1	0.164204	3
2	0.172456	2
3	0.237500	2
4	0.172456	2
5	0.098296	1
6	0.155089	2

Cuadro 1: PageRank vs Grado de Entrada

Como podemos observar, la web con mayor ranking en PageRank es la web número 3, mientras que al utilizar el grado como medida de relevancia la web con mayor ranking es la número 1.

Intuitivamente, notemos que el algoritmo del grado ignora el hecho de que una pagina relevante que linkea a otro sitio debe contribuirle mas a su relevancia que una que otra que no es tan relevante. Este algoritmo es claramente manipulable, dado que yo puedo crear muchos sitios fácilmente que linkeen a uno que quiero favorecer en el ranking. Aquí podemos observar nuevamente una de las ventajas de PageRank, al elegir un α suficientemente grande, el poder de manipulación que existe es muy bajo.

3.4. Manipulación

A continuación vamos a experimentar que tan manipulable es el algoritmo PageRank a medida que va cambiando el factor de teletransportacion. La idea es la siguiente, un algoritmo con un factor de teletransportacion bajo pondera menos en la matriz de transición el componente que representa la estructura del grafo. Por lo tanto, un Search Engine que tenga este factor de teletransportacion bajo va a ser sumamente manipulable, dado que puedo crear muchisimos nodos e inflar el ranking de cualquier sitio. Cuanto mayor es este factor, conjeturamos que vamos a observar que inflar cualquier sitio sera mucho mas costoso en termino de cantidad de sitios únicos que debo crear. De hecho, Page & Brin en su paper consideran esto y mencionan que en Google se ponderan muchos factores para evitar lo que hoy se conoce como SEO (Search Engine Optimization).

Primero vamos a ver las posiblidadades de manipulación con $c = 0{,}85$. Para hacer esto se fijo el valor de c, y se tomo una red inicial de 30 nodos y 30 ejes, y a la primer pagina se le agrego una pagina que la apunte por cada iteracion, este proceso se repitio hasta agregar 80 enlaces a la primer pagina. El resultado es:

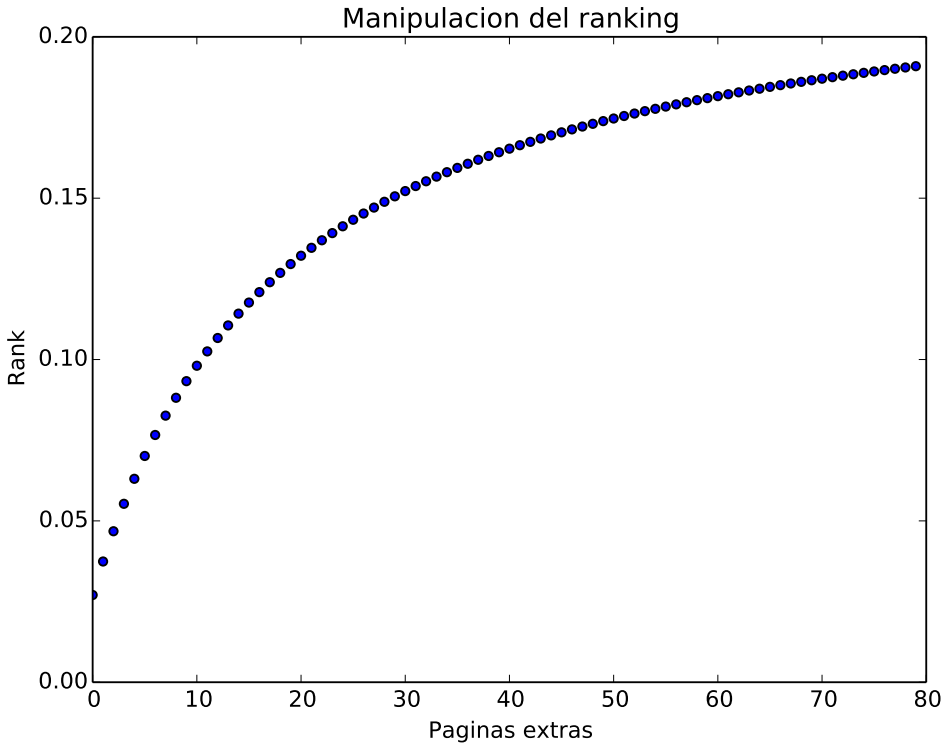


Figura 8: Score de una pagina web a medida que creamos mas paginas que lo apuntan.

Como era de esperar, a medida que aumentamos el numero de sitios web que apuntan a otro, su ranking comienza a subir lentamente. Sin embargo, el beneficio marginal de agregar nuevos sitios web es decreciente. Esto se debe a que la relevancia que aporta un nuevo sitio al que nadie lo linkea depende de forma decreciente de la cantidad de sitios web totales.

El aporte de un nuevo sitio web de este estilo siempre mejorara o mantendra el ranking de un sitio web. Aunque la relevancia total depende de todos los sitios que lo linkean, y al agregar un sitio le estoy quitando relevancia a los sitios web originales que lo linkeaban, el incremento en la relevancia al agregar un nuevo sitio web con c fijo compensa la caida en la relevancia de los sitios originales. Por eso observamos una curva creciente. Sin embargo, el beneficio marginal de agregar un sitio web luego solo compensa por la perdida en relevancia del resto de los sitios, razon por la cual la sucesion dada por la cantidad de nuevos sitios web parece converger.

Para el ultimo caso (es decir, el caso con los nuevos 80 enlaces extras apuntando a la primer pagina), decidimos ver como influyen los valores de c en su ranking. Se tomo $c = 0$, se lo aumento en 0,1, hasta llegar a 0,9. La idea es que a medida que aumenta c, el factor de teletransportacion tiene una mayor relevancia en el ranking de un sitio web. Por ejemplo, sabemos que si $c = 1$, se ignora la estructura del grafo y todos los sitios web tendran la misma relevancia, dado que todos linkean a todos con la misma probabilidad. El resultado fue el siguiente:

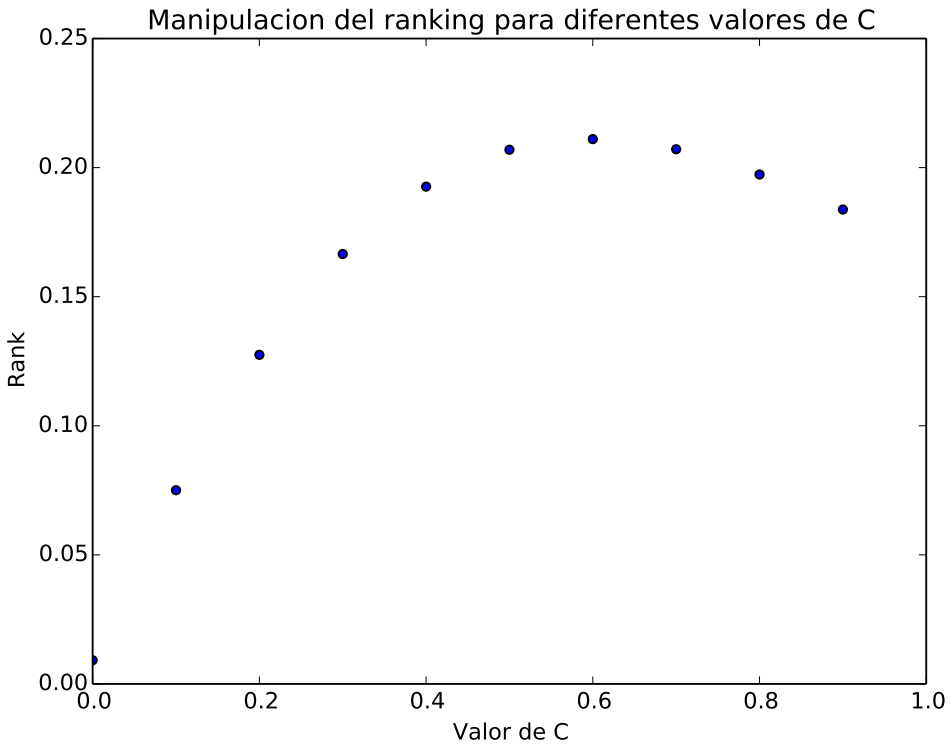


Figura 9: Cambio en el score de una pagina a la que la apuntaban otros sitios web falsos. A medida que aumenta c, notemos como cae su score.

En un comienzo, podemos observar que el ranking del sitio web mejora a medida que sube c. Esto se debe a que los sitios web 'falsos' que lo apuntan toman mayor relevancia e inflan al sitio web original. Sin embargo, luego observamos que la curva es decreciente. Esto se debe a que al aumentar demasiado c la relevancia del sitio web depende mas de cuantos sitios totales lo linkean y menos de la relevancia del mismo. A su vez, al ignorar la estructura del grafo, los nuevos sitios que agregue comienzan a linkear a todos por teletransporacion y no exclusivamente al sitio web original. Si un sitio web sumamente relevante linkeaba a mi sitio, el valor de ese link baja a medida que sube c. La curva es decreciente dado que el efecto de la perdida en relevancia de sitios entrantes le gana al efecto de los sitios web 'falsos' y ademas porque se comienza a ignorar la estructura del grafo para calcular los rankings.

3.5. Ranking ATP

Empezemos con el apartado que seguro el lector más esperaba de todo el trabajo: ¿Vilas fue o no 1ro entre 1975 y 1977? Esta pregunta podemos contestarla. Pero previo a esto, necesitamos poner al lector al tanto de la situación. Empecemos viendo los rankings oficiales de la época.

En 1975, Vilas llegó a la primera final de un Grand Slam, el Roland Garros, en donde fue derrotado por el sueco Björn Borg y cuartos de final de Wimbledon.

Este es el top 10 para 1975 segun la ATP, donde Guillermo Vilas se hubica 2do:

Nombre
Jimmy Connors
Guillermo Vilas
Bjorn Borg
Arthur Ashe
Manuel Orantes
Ken Rosewall
Ilie Nastase
John Alexander
Roscoe Tanner
Rod Laver

Cuadro 2: Ranking ATP 1975

En 1976 la ATP ubica a Vilas 6to dentro del top 10:

Nombre
Jimmy Connors
Bjorn Borg
Ilie Nastase
Manuel Orantes
Raul Ramirez
Guillermo Vilas
Adriano Panatta
Harold Solomon
Eddie Dibbs
Brian Gottfried

Cuadro 3: Ranking ATP 1976

En 1977 Vilas se ubica segundo, por debajo de Connors. Recordemos que esto hasta el día de hoy sigue creando polémicas debido a los muy buenos resultados obtenidos por Vilas en aquel año comparados con los del estadounidense. Veamos el top 10 oficial para este año:

Nombre
Jimmy Connors
Guillermo Vilas
Bjorn Borg
Vitas Gerulaitis
Brian Gottfried
Eddie Dibbs
Manuel Orantes
Raul Ramirez
Ilie Nastase
Dick Stockton

Cuadro 4: Ranking ATP 1977

Todos los años tienen como lider indiscutido al estadounidense Jimmy Connors.

3.6. Ranking ATP oficial vs. Ranking PageRank vs. Sort por diferencia de victorias/-derrotas

Introducimos aquí un algoritmo de rankeo con un criterio de ordenamiento basado en victorias/derrotas, que además, si hay empate define por diferencia de puntos, aunque para el caso de tenis y por como esta definido el sistema de puntos del mismo no tiene ninguna relevancia. Nos servirá para poder hacer un análisis cualitativo de las virtudes de PageRank sobre algoritmos más básicos y comparar estos resultados con el ranking oficial y poder obtener así alguna conclusión sobre los mótivos principales de la investigación.

Veamos que obtuvimos en cada año con este algoritmo, empezando por 1975:

Nombre	PG	PP
Arthur Ashe	96	18
Manuel Orantes	90	20
Guillermo Vilas	86	18
Jimmy Connors	79	11
Ilie Nastase	88	22
Bjorn Borg	82	17
Raul Ramirez	70	28
John Alexander	65	23
Roscoe Tanner	64	24
Jan Kodes	60	21

Cuadro 5: Ranking 1975 PG(partidos ganados) / PP(partidos perdidos)

Podemos ver que Vilas no aparece segundo, si no tercero. Connors fue desplazado al 4to lugar y en primer lugar aparece Arthur Ashe quien ocupaba el lugar que Connors ocupa ahora. Podemos ver varios cambios relacionados al top 10 oficial.

Todos ellos particularmente relacionados al hecho de que como se habia anticipado la cantidad de victorias sobre derrotas conformaria otro ranking diferente en el cual no importa exactamente que clase de victorias ha conseguido o derrotas sufrido un determinado participante. Podemos ver que Vilas con más victorias sobre derrotas que Connors, se encontraba por debajo de él en el ranking oficial. Esto se debe al sistema de puntos manejados por el ATP donde se suma más puntaje cuanto más avancemos en un torneo y cuantas más finales ganemos y por la jerarquía de ese torneo. Claramente no alcanza con ganar un solo torneo importante, para sumar se tiene que jugar.

Observemos el ranking de 1976:

Nombre	PG	PP
Jimmy Connors	86	9
Guillermo Vilas	83	20
Ilie Nastase	73	15
Raul Ramirez	91	33
Manuel Orantes	76	19
Eddie Dibbs	83	28
Bjorn Borg	57	12
Roscoe Tanner	71	27
Wojtek Fibak	75	32
Harold Solomon	70	27

Cuadro 6: Ranking 1976 PG(partidos ganados) / PP(partidos perdidos)

Vemos que Vilas avanzó del 6to lugar al 2do puesto que era ocupado por Bjorn Borg, que a su vez, fue desplazado a la 7ma posición.

Veamos que sucede en 1977:

Nombre	PG	PP
Guillermo Vilas	126	14
Brian Gottfried	105	22
Bjorn Borg	71	8
Jimmy Connors	69	15
Eddie Dibbs	77	28
Vitas Gerulaitis	60	15
Manuel Orantes	65	25
Horold Solomon	64	27
Raul Ramirez	61	24
Wojtek Fibak	66	30

Cuadro 7: Ranking 1977 PG(partidos ganados) / PP(partidos perdidos)

Ok. Tomemoslo con calma. Vilas aparece primero pero esto no es indicador absoluto de que la ATP cometió un error, debido a la falta de información que provee. Aunque si nos da indicios de lo que realmente pasó.

La diferencia de partidos ganados sobre perdidos con respecto a cualquier otro competidor es bastante significativa. Podemos ver que Connors fué desplazado al 4to lugar. Segundo se ubica Brian Gottfried, que durante ese año tuvo un gran desempeño, entre los que se encuentra su victoria sobre Vilas en la final del Roland Garros.

A priori...cantidad de victorias está relacionada con cantidad de puntos, pero esto no es una regla general y depende como mencionamos del sistema de puntajes. Para poder asemejar a la relevancia que la ATP le da a los torneos tenemos que utilizar un algoritmo que aproveche esa carácteristica lo mejor posible. Para esto haremos uso del modelo GeM y analizaremos sus resultados. Esperamos que los mismos se parezcan al ranking oficial, obviamente, con ciertas variaciones.

Utilizaremos los siguientes parámetros para generar los 3 rankings:

$c = 0.85$ $\delta = 0.00001$

Además, como indicamos en el diseño del sistema, usaremos una matriz de personalización uniforme, dado que no nos interesa que influyan sobre los resultados ninguna información estadistica de un torneo anterior, por el simple hecho de que evaluamos a los jugadores desde cero cada año.

Avancemos sobre los resultados, comenzando con 1975:

Nombre	Puntaje
Arthur Ashe	0.033172
Bjorn Borg	0.030089
Manuel Orantes	0.026483
Ilie Nastase	0.026254
Guilermo Vilas	0.023572
Jimmy Connors	0.021918
John Alexander	0.017379
Roscoe Tanner	0.015914
Brian Gottfried	0.011614
Ken Rosewall	0.011055

Cuadro 8: Ranking 1975 GeM

Vemos que Vilas se ubica en el 4to puesto, Connors fué desplazado al 5to puesto y por arriba se ubican Ashe en primer puesto y Borg en segundo lugar.

Si analizamos los partidos podremos saber que Ashe se enfrentó en una sola oportunidad a Connors y logró derrotarlo, en lo que fué la final Wimbledon. Además se enfrento en 7 oportunidades a Borg y logró vencerlo en 4 (Wimbledon, Barcelona WCT, Dallas WCT, Munich WCT), siendo Wimbledon el más importante de los 7 enfrentamientos.

Borg obtuvo la final de Roland Garros frente a Vilas como máximo logro.

Mientras tanto Manuel Orantes hizo lo propio para obtener el 3er lugar derrotando a Vilas en las 4 oportunidades que se encontraron y derrotando a Connors en su único enfrentamiento. No es una tarea fácil determinar todas estas congruencias pero con un simple vistazo a los partidos jugados y los torneos en los que se enfrentaron parece claro que el ranking es elocuente.

Veamos el ranking de 1976:

Nombre	Puntaje
Jimmy Connors	0.033300
Ilie Nastase	0.031773
Raul Ramirez	0.029343
Eddie Dibbs	0.026328
Brian Gottfried	0.025002
Harold Solomon	0.024189
Guillermo Vilas	0.024006
Bjorn Borg	0.023806
Manuel Orantes	0.020587
Adriano Panatta	0.014804

Cuadro 9: Ranking 1976 GeM

No parace haber mucho que analizar con respecto al ranking oficial. Vilas se encuentra una posición más abajo. Connors sigue siendo lider indiscutido y la otras posiciones relativas no han cambiado demasiado. Podriamos resaltar el caso particular de Borg que no tuvo malos resultados durante ese año, destacando su final ganada contra Ilie Nastase en Wimbledon, una final perdida en el US Open contra Connors y Cuartos de final en un Roland Garros. Pero si miramos el mismo ranking pero por diferencia de ganados/perdidos veremos que Borg se encuentra casi en la misma posición debido a la baja diferencia de partidos ganados sobre perdidos.

Concluyamos con el análisis de 1977:

Nombre	Puntaje
Guillermo Vilas	0.037046
Brian Gottfried	0.035463
Bjorn Borg	0.029461
Jimmy Connors	0.027319
Harold Solomon	0.024189
Eddie Dibbs	0.023230
Dick Stockton	0.021816
Vitas Gerulaitis	0.017483
Raul Ramirez	0.016667
Manuel Orantes	0.016318

Cuadro 10: Ranking 1977 GeM

Por una diferencia significativa, Vilas desplaza del primero puesto a Connors para quedarse el con el trono. Pero hay una razón bastante justificada para que esto suceda. En 1977 Vilas conquistó 17 torneos (récord todavía vigente), se consagró en Roland Garros frente a Brian Gottfried por 6-0 6-3 6-0 y el US Open frente a Jimmy Connors por 2-6 6-3 7-6 6-0, fue finalista de Australia donde cayó frente a Roscoe Tanner por 3-6 3-6 3-6, logró una seguidilla de 50 partidos consecutivos sin conocer la derrota y, durante esos doce meses, ganó 145 de los 159 encuentros que jugó (91,1 %). Jimmy Connors, en 1977 no obtuvo ningún torneo de Grand Slam y ganó ocho torneos, menos de la mitad de los logrados por Vilas. Parece lógico que Connors incluso esté unos escalones más abajo. Además el ranking por partidos ganados/perdidos para los primeros 4 es idéntico. Esto no puede hacer más que confirmar que Vilas fué el primero indiscutido en 1977 por una amplia diferencia, sobre todo por el bajo rendimiento de Connors durante ese año.

Como comentamos al principio, la cantidad de partidos ganados y los rivales derrotados son factores importantes a la hora de generar el ranking con GeM. Si la ATP hubiese cambiado a este método y fuera retroactivo, hoy el número 1 indiscutido de 1977 sería sin lugar a dudas Guillermo Vilas.

4. Conclusiones

Los algoritmos de rankeo son sumamente relevantes. Se utilizan en todo tipo de situaciones donde es necesario asignar un orden, ya sea entre personas, equipos o países. En el presente trabajo se decidió analizar el algoritmo PageRank. Este algoritmo fue un algoritmo icónico que marco el inicio de lo que hoy conocemos como Google en 1998. Hoy en día se sigue utilizando el algoritmo original y muchas variantes del mismo en la practica.

Como ya hemos mostrado, computar el algoritmo PageRank se reduce a encontrar un autovector de norma unitaria correspondiente al autovalor 1 del sistema, es decir, buscamos x tal que $Ax = x$, donde A es la matriz de transición y x representa un ranking valido. Una de las primeras cuestiones que nos encontramos al tener que aplicar el algoritmo en la práctica es el modelado de la matriz de transiciones. Se debe demostrar que nuestro modelado cumple con que la matriz es estocástica positiva por columnas. De esta manera mostramos que el Método de la Potencia el cual utilizamos para encontrar un autovector válido converge y además resulta en un ranking válido.

Un factor sumamente relevante antes de considerar computar el problema es la representación de la matriz de transición. Esto es sumamente importante, dado que en problemas con un grafo de gran tamaño la matriz de transición puede ocupar mucho espacio en memoria. Aquí es donde entran las representaciones alternativas de matrices esparsas, que nos permiten superar este inconveniente en casos como el de los sitios web. Este problema no surge al modelar competencias deportivas, dado que en general el tamaño de la matriz de transición esta acotado.

Uno de los experimentos mas interesantes que llevamos a cabo fue el de la manipulacion del PageRank. Como era de esperar, notamos que a medida que el factor de teletransportación es más bajo, más fácil es para un usuario malintencionado manipular el ranking. A su vez, observamos que el ranking se puede manipular pero hasta cierto punto. Hay un momento que marginalmente agregar mas sitios web que apuntan a uno que queremos inflar ya no tiene efecto.

Sin embargo, mostramos que el algoritmo PageRank se comporta mucho mejor que otros métodos 'naive', como lo es ordenar vertices por grado de entrada. Este tipo de métodos son sumamente manipulables y no capturan la relevancia relativa al computar los rankings.

Finalmente, la pregunta más importante del trabajo. ¿Fue Vilas líder del ranking de la ATP entre 1975 y 1977?

Concluimos que si la ATP hubiese utilizado PageRank, **EFECTIVAMENTE** Vilas hubiese sido líder del ranking de la ATP en el año 1977, convirtiéndose en el primer argentino en lograr semejante hazaña. Ya estamos haciendo una campaña en change.org para que actualicen el ranking.

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 [**?**], 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

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

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 [**?**] y Kamvar et al. [**?**, 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 [**?**] y Kamvar et al. [**?**], 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. [**?**] se hace una breve reseña de dos ellos, y los autores proponen un nuevo método basado en el algoritmo PageRank que denominan GeM[2]. 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. [**?**] 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., [**?**, Sección GeM Ranking Method]) es el siguiente[3]:

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

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

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

[3] Notar que en artículo, Govan et al. [**?**] 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 $P_2 x = x$. 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 [**?**] donde se explica la intución y algunos detalles técnicos respecto a PageRank. Además, en Kamvar et al. [**?**] 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. [**?**, 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. [**?**] son correctos y computan $P_2 x$.

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 [**?**] (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 [**?**, 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 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 [**?**], 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

```
1  #include <iostream>
2  #include <math.h>
3  #include <fstream>
4  #include <sstream>
5  #include <stdio.h>
6  #include <string.h>
7  #include <time.h>
8  #include <new>
9  #include <regex>
10 #include <iterator>
11 #include "sparseMatrix.h"
12
13 using namespace std;
14
15 struct dataNode {
16     int node;
17     int edgesCount;
18 };
19
20 struct matchesStats {
21     int team;
22     int matchesWin;
23     int matchesDefeat;
24     int pointsScored;
25     int pointsReceived;
26 };
27
28 //webs / sports
29 Matrix<double> pageRank(Matrix<double>& M, double c, double d, vector<int>&
       nodesCount);
30 Matrix<double> enhancementPageRank(dok& diccMatrix, double c, double d, vector<int>&
       nodesCount);
31
32 //webs
33 void in_deg(vector<dataNode>& nodesCount);
34
35 //sports
36 void basic_sort(vector<matchesStats>& stats);
37
38 //out data
39 void saveResultPageRank(FILE * pFile, Matrix<double>& data);
40 void saveResultInDeg(FILE * pFile, vector<dataNode>& data);
41 void saveResultBasicSort(FILE * pFile, vector<matchesStats>& data);
42
43 //utils
44 double uniform_rand(double a, double b);
45
46 int main(int argc, char** argv) {
47
48     if (argc < 3) {
49         printf("Usage %s: parametros.in salida.out \n", argv[0]);
50         return 0;
51     }
52
53     ifstream inputFile(argv[1]);
```

```
54
55      if (!inputFile.good()) {
56          printf("can't open input file.\n");
57          return 0;
58      }
59
60      FILE * outputFile = NULL;
61
62      outputFile = fopen(argv[2],"w");
63      if (outputFile == NULL){
64          printf("can't open output file.\n");
65          return 0;
66      }
67
68      int alg = 0;
69      double c = 0;
70      int inst = 0;
71      double e = 0;
72      char testFileName[100];
73
74      string line;
75      getline(inputFile, line);
76
77      sscanf(line.c_str(),"%d %lf %d %s %lf", &alg, &c, &inst, testFileName, &e);
78
79      ifstream testFile(testFileName);
80
81      if (inst == 0){
82          int nodes = 0;
83          int edges = 0;
84
85          getline(testFile, line);
86          getline(testFile, line);
87          getline(testFile, line); //3rd line: nodes and edges count
88
89          char * cstrhead = new char [line.length()+1];
90          strcpy(cstrhead, line.c_str());
91
92          char * chr_numhead = strtok(cstrhead," \t");
93          strtok(NULL," \t");
94          chr_numhead = strtok(NULL," \t");
95          nodes = atoi(chr_numhead);
96          strtok(NULL," \t");
97          chr_numhead = strtok(NULL," \t");
98          edges = atoi(chr_numhead);
99
100         cout << "nodes: " << nodes << " edges: " << edges << endl;
101
102         getline(testFile, line); //4th line
103
104       if (alg == 0){
105             //Matrix<double> M(nodes, nodes); //with A
106             dok diccMatrix; //with B
107
108             vector<int> nodesCount(nodes);
109
110             for(int i = 0; i < nodes; i++) {
111                 nodesCount[i] = 0;
112             }
```

```
113
114             int i = 0;
115             while(i < edges){
116                 int node_from = 0;
117                 int node_to = 0;
118
119                 getline(testFile, line); //5th line: graph data
120
121                 char * cstr = new char [line.length()+1];
122                 strcpy(cstr, line.c_str());
123
124                 char * chr_num = strtok(cstr," \t");
125
126                 node_from = atoi(chr_num);
127                 chr_num = strtok(NULL," \t");
128                 node_to = atoi(chr_num);
129
130                 //cout << "node from " << node_from << " node to " << node_to << endl
                        ;
131
132                 nodesCount[node_from-1] += 1;
133
134                 //M(node_to-1, node_from-1) = 1; //with A
135                 rowCol p(node_to-1,node_from-1); //with B
136                 diccMatrix.insert(dok::value_type(p,1)); //with B
137                 i++;
138             }
139
140             //Matrix<double> res = pageRank(M, c, e, nodesCount);
141             Matrix<double> res = enhancementPageRank(diccMatrix, c, e, nodesCount);
142
143             cout << "page rank result: \n" << endl;
144             //res.printMatrix();
145
146             saveResultPageRank(outputFile, res);
147         }else{
148            // group algorithm webs
149            vector<dataNode> nodesCount(nodes);
150
151            for(int i = 0; i < nodes; i++){
152                 dataNode nod;
153                 nod.node = i+1;
154                 nod.edgesCount = 0;
155                 nodesCount[i] = nod;
156            }
157
158            int i = 0;
159            while(i < edges){
160                 int node_from = 0;
161                 int node_to = 0;
162
163                 getline(testFile, line); //5th line: graph data
164
165                 char * cstr = new char [line.length()+1];
166                 strcpy(cstr, line.c_str());
167
168                 char * p = strtok(cstr," \t");
169
170                 node_from = atoi(p);
```

```
171                 p = strtok(NULL," \t");
172                 node_to = atoi(p);
173
174                 dataNode nod = nodesCount[node_from-1];
175
176                 nod.edgesCount += 1;
177
178                 nodesCount[node_from-1] = nod;
179
180                 i++;
181             }
182
183             in_deg(nodesCount);
184
185             saveResultInDeg(outputFile, nodesCount);
186         }
187     }else{
188         int teams = 0;
189         int matches = 0;
190
191         getline(testFile, line);
192
193         sscanf(line.c_str(),"%d %d", &teams, &matches);
194         cout << "teams: " << teams << " matches: " << matches << endl;
195
196         int day = 0;
197         int local = 0;
198         int visitor = 0;
199         int local_score = 0;
200         int visitor_score = 0;
201
202         if (alg == 0){
203             // page rank sports
204             Matrix<double> M(teams, teams);
205
206             vector<int> totalAbs(teams);
207
208             for(int i = 0; i < teams; i++) {
209                 totalAbs[i] = 0;
210             }
211
212             int i = 0;
213             while(i < matches){
214                 getline(testFile, line);
215                 sscanf(line.c_str(),"%d %d %d %d %d", &day, &local, &local_score, &
                        visitor, &visitor_score);
216
217                 int abs_score = abs(local_score-visitor_score);
218                 if (local_score > visitor_score) {
219                     M(local-1, visitor-1) += abs_score;
220                     totalAbs[visitor-1] += abs_score;
221                 }else if(visitor_score > local_score) {
222                     M(visitor-1, local-1) += abs_score;
223                     totalAbs[local-1] += abs_score;
224                 }
225
226                 i++;
227             }
228
```

```
229             Matrix<double> res = pageRank(M, c, e, totalAbs);
230
231             cout <<  "gem result: \n" << endl;
232             res.printMatrix();
233
234             saveResultPageRank(outputFile, res);
235         }else{
236             // group algorithm sports
237             vector<matchesStats> stats(teams);
238
239             for(int i = 0; i < teams; i++){
240                 matchesStats teamStats;
241                 teamStats.team = i+1;
242                 teamStats.matchesWin = 0;
243                 teamStats.matchesDefeat = 0;
244                 teamStats.pointsScored = 0;
245                 teamStats.pointsReceived = 0;
246                 stats[i] = teamStats;
247             }
248
249             int i = 0;
250             while(i < matches){
251                 getline(testFile, line);
252                 sscanf(line.c_str(),"%d %d %d %d %d", &day, &local, &local_score, &
                        visitor, &visitor_score);
253
254                 matchesStats localStats = stats[local-1];
255                 matchesStats visitorStats = stats[visitor-1];
256
257                 if (local_score > visitor_score) {
258                     localStats.matchesWin += 1;
259                     visitorStats.matchesDefeat += 1;
260                 }else if(visitor_score > local_score) {
261                     localStats.matchesDefeat += 1;
262                     visitorStats.matchesWin += 1;
263                 }
264
265                 localStats.pointsScored += local_score;
266                 localStats.pointsReceived += visitor_score;
267                 visitorStats.pointsScored += visitor_score;
268                 visitorStats.pointsReceived += local_score;
269
270                 stats[local-1] = localStats;
271                 stats[visitor-1] = visitorStats;
272
273                 i++;
274             }
275
276             basic_sort(stats);
277
278             saveResultBasicSort(outputFile, stats);
279         }
280     }
281
282     testFile.close();
283
284     inputFile.close();
285
286     if (outputFile != NULL) fclose(outputFile);
```

```
287
288     return 0;
289 }
290
291 Matrix<double> pageRank(Matrix<double>& M, double c, double d, vector<int>&
        nodesCount) {
292     srand(45);
293
294     int n = M.rows();
295     double dbl_n = M.rows();
296
297     int j = 0;
298     while(j < n){
299         int i = 0;
300         while(i < n){
301             if(M(i, j) != 0){
302                 M(i, j) = M(i, j) / (double)nodesCount[j];
303             }else if(nodesCount[j] == 0){
304                 M(i, j) = 1/dbl_n; // dangling node / undefeated team
305             }
306             i++;
307         }
308         j++;
309     }
310
311     Matrix<double> v(n, 1/dbl_n);
312
313     Matrix<double> E(n, n, (1 - c)*1/dbl_n); // PRE: rows == columns
314
315     //Salvo que sea c = 1 no tiene sentido usar Sparse Matrix
316     Matrix<double> A = M*c + E;
317
318     Matrix<double> x(n, 1, 1/dbl_n);
319
320     Matrix<double> last_x(n);
321
322     double delta = 0;
323
324     do {
325         last_x = x;
326         x = A*x;
327         delta = x.L1(last_x);
328     }while (delta > d);
329
330     printf("delta is %f\r\n", delta);
331
332     return x;
333 }
334
335 Matrix<double> enhancementPageRank(dok& diccMatrix, double c, double d, vector<int>&
        nodesCount) {
336     srand(45);
337
338     int n = nodesCount.size();
339     double dbl_n = nodesCount.size();
340
341     for (dok::iterator it= diccMatrix.begin(); it!=diccMatrix.end(); ++it) {
342         if (it->second > 0) {
343             it->second = it->second / (double)nodesCount[it->first.second];
```

```
344             }else if (nodesCount[it->first.second] == 0) {
345                 it->second = 1/dbl_n; // dangling node / undefeated team
346             }
347         }
348 
349         SparseMatrix<double> A(diccMatrix, n, n);
350 
351         SparseMatrix<double> x(n, 1/dbl_n);
352 
353         SparseMatrix<double> last_x(n);
354 
355         SparseMatrix<double> v(n, 1/dbl_n);
356 
357         double delta = 0;
358 
359         do {
360             last_x = x;
361 
362             x = A*x;
363             x = x*c;
364 
365             double w = last_x.norm1() - x.norm1();
366 
367             x = x +  v*w;
368 
369             delta = x.L1(last_x);
370         }while (delta > d);
371 
372         printf("delta is %f\r\n", delta); //Deberia devolverse.
373 
374         return x.descompress();
375     }
376 
377     void in_deg(vector<dataNode>& nodesCount) {
378         sort(nodesCount.begin(), nodesCount.end(), [](dataNode a, dataNode b) {
379             return b.edgesCount < a.edgesCount;
380         });
381 
382         cout << "IN-DEG result \n" << endl;
383         for (dataNode a : nodesCount) {
384             cout << "node: " << a.node << " points: " << a.edgesCount << "\n";
385         }
386     }
387 
388     void basic_sort(vector<matchesStats>& stats) {
389         sort(stats.begin(), stats.end(), [](matchesStats a, matchesStats b) {
390             if (a.matchesWin - a.matchesDefeat != b.matchesWin - b.matchesDefeat) {
391                 return b.matchesWin - b.matchesDefeat < a.matchesWin - a.matchesDefeat;
392             }else{
393                 return b.pointsScored - b.pointsReceived < a.pointsScored - a.
                        pointsReceived;
394             }
395         });
396 
397         cout << "basic sort result \n" << endl;
398         for (matchesStats a : stats) {
399             cout << "team: " << a.team
400             << " matches win: " << a.matchesWin << " matches defeat: " << a.matchesDefeat
```

```
401         << " points scored: " << a.pointsScored << " points received: " << a.
                pointsReceived <<"\n";
402     }
403 }
404
405 void saveResultPageRank(FILE * pFile, Matrix<double>& data) {
406     int n = data.rows();
407     int i = 0;
408
409     while(i < n){
410         fprintf(pFile, "%f\r\n", data(i));
411         i++;
412     }
413 }
414
415 void saveResultInDeg(FILE * pFile, vector<dataNode>& data) {
416     for (dataNode a : data){
417         fprintf(pFile, "%d %d\r\n", a.node, a.edgesCount);
418     }
419 }
420
421 void saveResultBasicSort(FILE * pFile, vector<matchesStats>& data) {
422     for (matchesStats a : data){
423         fprintf(pFile, "%d %d %d %d %d\r\n", a.team, a.matchesWin, a.matchesDefeat, a
                .pointsScored, a.pointsReceived);
424     }
425 }
426
427 double uniform_rand(double a, double b) {
428     return ((b-a)*((double)rand()/RAND_MAX))+a;
429 }
```

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();
```

```
58
59      private:
60          vector<vector<T> > _values;
61          int _rows;
62          int _columns;
63
64  };
65
66  template<class T>
67  Matrix<T>::Matrix()
68      : _values(1), _rows(1), _columns(1)
69  {
70      _values[0].resize(1);
71  }
72
73  template<class T>
74  Matrix<T>::Matrix(int rows)
75      : _values(rows), _rows(rows), _columns(1)
76  {
77      for(int i = 0; i < rows; i++) {
78          _values[i].resize(1);
79      }
80  }
81
82  template<class T>
83  Matrix<T>::Matrix(int rows, int col)
84      : _values(rows), _rows(rows), _columns(col)
85  {
86      for(int i = 0; i < rows; i++) {
87          _values[i].resize(col);
88          //cout << "row " << i << endl;
89      }
90  }
91
92  template<class T>
93  Matrix<T>::Matrix(int rows, int col, const T& init)
94      : _values(rows), _rows(rows), _columns(col)
95  {
96      for(int i = 0; i < rows; i++) {
97          _values[i].resize(col, init);
98      }
99  }
100
101 template<class T>
102 Matrix<T>::Matrix(const Matrix<T>& other)
103     : _values(other._values), _rows(other._rows), _columns(other._columns)
104 {}
105
106 template<class T>
107 Matrix<T>::~Matrix() {}
108
109 template<class T>
110 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& other) {
111   if (&other == this)
112     return *this;
113
114   int new_rows = other._rows;
115   int new_columns = other._columns;
116
```

```
117     _rows = new_rows;
118     _columns = new_columns;
119 
120     _values.resize(new_rows);
121     for (int i = 0; i < new_columns; i++) {
122         _values[i].resize(new_columns);
123     }
124 
125     for(int i = 0; i < new_rows; i++) {
126       for(int j = 0; j < new_columns; j++) {
127         _values[i][j] = other(i, j);
128       }
129     }
130 
131     return *this;
132 }
133 
134 template<class T>
135 Matrix<T> Matrix<T>::operator*(const Matrix<T>& other) {
136     // ASUME QUE LAS DIMENSIONES DAN
137     Matrix<T> result(_rows, other._columns);
138 
139     int innerDim = _columns; // Tambien podria ser other._rows
140 
141     for(int i = 0; i < result._rows; i++) {
142         for(int j = 0; j < result._columns; j++) {
143             result(i,j) = 0;
144             for(int k = 0; k < innerDim; k++) {
145                 result(i,j) += _values[i][k] * other(k,j);
146             }
147         }
148     }
149 
150     return result;
151 }
152 
153 template<class T>
154 Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& other) {
155     Matrix<T> result = (*this) * other;
156     (*this) = result;
157     return (*this);
158 }
159 
160 template<class T>
161 Matrix<T> Matrix<T>::operator+(const Matrix<T>& other) {
162     // ASUME QUE LAS DIMENSIONES DAN
163     Matrix<T> result(_rows, other._columns);
164 
165     for(int i = 0; i < result._rows; i++) {
166         for(int j = 0; j < result._columns; j++) {
167             result(i,j) = _values[i][j] + other(i,j);
168         }
169     }
170 
171     return result;
172 }
173 
174 template<class T>
175 Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& other) {
```

```
176     Matrix<T> result = (*this) + other;
177     (*this) = result;
178     return (*this);
179 }
180
181 template<class T>
182 Matrix<T> Matrix<T>::operator-(const Matrix<T>& other) {
183     // ASUME QUE LAS DIMENSIONES DAN
184     Matrix<T> result(_rows, other._columns);
185
186     for(int i = 0; i < result._rows; i++) {
187         for(int j = 0; j < result._columns; j++) {
188             result(i,j) = _values[i][j] - other(i,j);
189         }
190     }
191
192     return result;
193 }
194
195 template<class T>
196 Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& other) {
197     Matrix<T> result = (*this) - other;
198     (*this) = result;
199     return (*this);
200 }
201
202 template<class T>
203 Matrix<T> Matrix<T>::operator*(const T& scalar) {
204     Matrix<T> result(_rows, _columns);
205
206     for(int i = 0; i < result._rows; i++) {
207         for(int j = 0; j < result._columns; j++) {
208             result(i,j) = _values[i][j] * scalar;
209         }
210     }
211
212     return result;
213 }
214
215 template<class T>
216 Matrix<T> Matrix<T>::operator/(const T& scalar) {
217     Matrix<T> result(_rows, _columns);
218
219     for(int i = 0; i < result._rows; i++) {
220         for(int j = 0; j < result._columns; j++) {
221             result(i,j) = _values[i][j] / scalar;
222         }
223     }
224
225     return result;
226 }
227
228 template<class T>
229 T& Matrix<T>::operator ()(int a, int b) {
230     return _values[a][b];
231 }
232
233 template<class T>
234 const T& Matrix<T>::operator ()(const int a, const int b) const {
```

```
235     return _values[a][b];
236 }
237
238 template<class T>
239 T& Matrix<T>::operator ()(int a) {
240     return _values[a][0];
241 }
242
243 template<class T>
244 const T& Matrix<T>::operator ()(const int a) const {
245     return _values[a][0];
246 }
247
248 template<class T>
249 double Matrix<T>::L1(const Matrix<T>& other) {
250     Matrix<T> vectorSubs = *this - other;
251
252     double res = 0;
253     for(int i = 0; i < rows(); i++){
254         res = res + abs(vectorSubs(i));
255     }
256
257     return res;
258 }
259
260 template<class T>
261 double Matrix<T>::norm1() {
262     double res = 0;
263     for(int i = 0; i < rows(); i++){
264         res = res + abs(_values[i][0]);
265     }
266
267     return res;
268 }
269
270
271 template<class T>
272 int Matrix<T>::rows() {
273     return _rows;
274 }
275
276 template<class T>
277 int Matrix<T>::columns() {
278     return _columns;
279 }
280
281 template<class T>
282 int Matrix<T>::rows() const{
283     return _rows;
284 }
285
286 template<class T>
287 int Matrix<T>::columns() const{
288     return _columns;
289 }
290
291 template<class T>
292 void Matrix<T>::printMatrix() {
293     for(int i = 0; i < _rows; i++) {;
```

```
294             for(int j = 0; j < _columns; j++) {
295                 printf("%f ", _values[i][j]);
296                 // cout << _values[i][j] << " ";
297             }
298             cout << endl;
299         }
300         cout << endl;
301     }
302
303     #endif  /* MATRIX_H */
```

6.3. sparseMatrix.h

```
1  /*
2   * File:   SparseMatrix.h
3   * Author: Rodrigo Kapobel
4   *
5   * Created on August 21, 2015, 23:05 PM
6   */
7
8  #ifndef SparseMatrix_H
9  #define SparseMatrix_H
10
11 #include <algorithm>
12 #include <math.h>
13 #include <vector>
14 #include <stdio.h>
15 #include <cmath>
16 #include "matrix.h"
17 #include <map>
18
19 using namespace std;
20
21 //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
22
23 typedef pair<int, int> rowCol;
24
25 struct comparator {
26     bool operator() (const rowCol p1, const rowCol p2) const
27     {
28         if (p1.first == p2.first) {
29             return p1.second < p2.second;
30         }
31
32         return p1.first < p2.first;
33     }
34 };
35
36 typedef map< rowCol, double, comparator > dok;
37
38 template<class T>
39 class SparseMatrix {
40 public:
41     SparseMatrix();
42     SparseMatrix(int rows); // column vector with 0's
43     SparseMatrix(int rows, T value);
44     SparseMatrix(vector<T>& values, vector<int>& iValues, vector<int>& jValues, int
          rows, int columns); // PRE: jValues::size == values::size & values of iValues
          [0..iValues::size-2] are indices of values & iValue[iValues::size-1] == values
          ::size
45     SparseMatrix(const SparseMatrix<T>& other); // compress matrix
46     SparseMatrix(const Matrix<T>& other);
47     SparseMatrix(const dok, int num_rows, int num_cols); //create sparse matrix from
          a dok map
48     ~SparseMatrix();
49
50     SparseMatrix<T>& operator=(const SparseMatrix<T>& other);
```

```
51      SparseMatrix<T> operator*(const SparseMatrix<T>& other); // performs a matrix-
            vector multiplication
52      SparseMatrix<T>& operator*=(const SparseMatrix<T>& other); // performs a matrix-
            vector multiplication
53      SparseMatrix<T> operator+(const SparseMatrix<T>& other); // performs a vector-
            vector sum
54      SparseMatrix<T>& operator+=(const SparseMatrix<T>& other); // performs a vector-
            vector sum
55      SparseMatrix<T> operator-(const SparseMatrix<T>& other); // performs a vector-
            vector subs
56      SparseMatrix<T>& operator-=(const SparseMatrix<T>& other); // performs a vector-
            vector subs
57
58      SparseMatrix<T> operator*(const T& scalar);
59      SparseMatrix<T> operator/(const T& scalar);
60
61      T& operator()(int a, int b);
62      const T& operator()(const int a, const int b) const;
63      T& operator()(int a);
64      const T& operator()(const int a) const;
65
66      Matrix<T> descompress();
67      //PRE are vectors
68      double L1(const SparseMatrix<T>& other);
69      double norm1();
70
71      int rows();
72      int columns();
73      int rows() const;
74      int columns() const;
75      void printSparseMatrix();
76
77  private:
78      vector<T> _values;
79      vector<int> _iValues;
80      vector<int> _jValues;
81
82      int _rows;
83      int _columns;
84  };
85
86  template<class T>
87  SparseMatrix<T>::SparseMatrix()
88  : _values(1), _iValues(1), _jValues(1), _columns(1)
89  {}
90
91  template<class T>
92  SparseMatrix<T>::SparseMatrix(int rows)
93  : _values(rows), _iValues(1), _jValues(rows), _rows(rows), _columns(1)
94  {
95      for(int i = 0; i <= rows; i++){
96          _iValues.resize(i+1, i);
97      }
98  }
99
100 template<class T>
101 SparseMatrix<T>::SparseMatrix(int rows, T value)
102 : _values(rows), _iValues(1), _jValues(rows), _rows(rows), _columns(1)
103 {
```

```
104     for(int i = 0; i <= rows; i++){
105         _iValues.resize(i+1, i);
106         if(i < rows){
107             _values[i] = value;
108         }
109     }
110 }
111
112 template<class T>
113 SparseMatrix<T>::SparseMatrix(vector<T>& values, vector<int>& iValues, vector<int>&
        jValues, int rows, int columns)
114 : _values(values), _iValues(iValues), _jValues(jValues), _rows(rows), _columns(
        columns)
115 {}
116
117 template<class T>
118 SparseMatrix<T>::SparseMatrix(const SparseMatrix<T>& other)
119 : _values(other._values), _iValues(other._iValues), _jValues(other._jValues), _rows(
        other.rows()), _columns(other.columns())
120 {}
121
122 template<class T>
123 SparseMatrix<T>::SparseMatrix(const Matrix<T>& other)
124 : _values(0), _iValues(0), _jValues(0), _rows(other.rows()), _columns(other.columns()
        )
125 {
126     for(int i = 0; i < _rows; i++) {
127         _iValues.resize(_iValues.size()+1, _values.size());
128         for(int j = 0; j < _columns; j++) {
129             if (other(i, j) != 0) {
130                 _values.resize(_values.size()+1, other(i, j));
131                 _jValues.resize(_jValues.size()+1, j);
132             }
133         }
134     }
135     _iValues.resize(_iValues.size()+1, _values.size());
136 }
137
138 template<class T>
139 SparseMatrix<T>::SparseMatrix(const dok dicc, int num_rows, int num_cols)
140 : _values(0), _iValues(0), _jValues(0), _rows(num_rows), _columns(num_cols)
141 {
142     int actual_row = -1;
143
144     for (dok::const_iterator it= dicc.begin(); it!=dicc.end(); ++it) {
145
146         if (actual_row != it->first.first) {
147             //this is because csr need to represent empty rows
148             while (actual_row < it->first.first) {
149                 _iValues.resize(_iValues.size()+1, _values.size());
150                 actual_row++;
151             }
152             actual_row = it->first.first;
153         }
154
155         if (it->second != 0) {
156             _values.resize(_values.size()+1, it->second);
157             _jValues.resize(_jValues.size()+1, it->first.second);
158         }
```

```
159     }
160
161     _iValues.resize(_iValues.size()+1, _values.size());
162 }
163
164 template<class T>
165 SparseMatrix<T>::~SparseMatrix() {}
166
167 template<class T>
168 SparseMatrix<T>& SparseMatrix<T>::operator=(const SparseMatrix<T>& other) {
169     if (&other == this)
170         return *this;
171
172     _values = other._values;
173     _iValues = other._iValues;
174     _jValues = other._jValues;
175     _columns = other.columns();
176
177     return *this;
178 }
179
180 template<class T>
181 SparseMatrix<T> SparseMatrix<T>::operator*(const SparseMatrix<T>& other) {
182     // ASUME QUE LAS DIMENSIONES DAN
183     SparseMatrix<T> result(rows());
184
185     for (int i = 0; i < rows(); i++) {
186         for (int j = _iValues[i]; j < _iValues[i+1]; j++) {
187             result(i) += _values[j]*other(_jValues[j]);
188         }
189     }
190
191     return result;
192 }
193
194 template<class T>
195 SparseMatrix<T>& SparseMatrix<T>::operator*=(const SparseMatrix<T>& other) {
196     SparseMatrix<T> result = (*this) * other;
197     (*this) = result;
198     return (*this);
199 }
200
201 template<class T>
202 SparseMatrix<T> SparseMatrix<T>::operator+(const SparseMatrix<T>& other) {
203     SparseMatrix<T> result(rows());
204
205     for(int i = 0; i < rows(); i++) {
206         result(i) = _values[i] + other(i);
207     }
208
209     return result;
210 }
211
212 template<class T>
213 SparseMatrix<T>& SparseMatrix<T>::operator+=(const SparseMatrix<T>& other) {
214     SparseMatrix<T> result = (*this) + other;
215     (*this) = result;
216     return (*this);
217 }
```

```
218
219 template<class T>
220 SparseMatrix<T> SparseMatrix<T>::operator-(const SparseMatrix<T>& other) {
221     SparseMatrix<T> result(rows());
222
223     for(int i = 0; i < rows(); i++) {
224         result(i) = _values[i] - other(i);
225     }
226
227     return result;
228 }
229
230 template<class T>
231 SparseMatrix<T>& SparseMatrix<T>::operator-=(const SparseMatrix<T>& other) {
232     SparseMatrix<T> result = (*this) - other;
233     (*this) = result;
234     return (*this);
235 }
236
237 template<class T>
238 SparseMatrix<T> SparseMatrix<T>::operator*(const T& scalar) {
239
240     vector<T> resValues(_values.size());
241
242     for(int i = 0; i < _values.size(); i++) {
243        resValues[i] = _values[i] * scalar;
244     }
245
246     SparseMatrix<T> result(resValues, _iValues, _jValues, _rows, _columns);
247
248     return result;
249 }
250
251 template<class T>
252 SparseMatrix<T> SparseMatrix<T>::operator/(const T& scalar) {
253     vector<T> resValues(_values.size());
254
255     for(int i = 0; i < _values.size(); i++) {
256        resValues[i] = _values[i] / scalar;
257     }
258
259     SparseMatrix<T> result(resValues, _iValues, _jValues, _columns);
260
261     return result;
262 }
263
264 template<class T>
265 T& SparseMatrix<T>::operator ()(int a) {
266     return _values[a];
267 }
268
269 template<class T>
270 const T& SparseMatrix<T>::operator ()(const int a) const {
271     return _values[a];
272 }
273
274 template<class T>
275 Matrix<T> SparseMatrix<T>::descompress() {
276     Matrix<T> result(rows(), _columns, 0);
```

```
277     for (int i = 0; i < rows(); i++) {
278         for (int j = _iValues[i]; j < _iValues[i+1]; j++) {
279             result(i, _jValues[j]) = _values[j];
280         }
281     }
282 
283     return result;
284 }
285 
286 template<class T>
287 double SparseMatrix<T>::L1(const SparseMatrix<T>& other) {
288     SparseMatrix<T> vectorSubs = *this - other;
289     double res = 0;
290     for(int i = 0; i < rows(); i++){
291         res = res + abs(vectorSubs(i));
292     }
293 
294     return res;
295 }
296 
297 template<class T>
298 double SparseMatrix<T>::norm1() {
299     double res = 0;
300     for(int i = 0; i < rows(); i++){
301         res = res + abs(_values[i]);
302     }
303 
304     return res;
305 }
306 
307 template<class T>
308 int SparseMatrix<T>::rows() {
309     return _rows;
310 }
311 
312 template<class T>
313 int SparseMatrix<T>::columns() {
314     return _columns;
315 }
316 
317 template<class T>
318 int SparseMatrix<T>::rows() const{
319     return _rows;
320 }
321 
322 template<class T>
323 int SparseMatrix<T>::columns() const{
324     return _columns;
325 }
326 
327 template<class T>
328 void SparseMatrix<T>::printSparseMatrix() {
329     Matrix<T> descompressedMatrix = descompress();
330     descompressedMatrix.printMatrix();
331 }
332 
333 #endif  /* SparseMatrix_H */
```