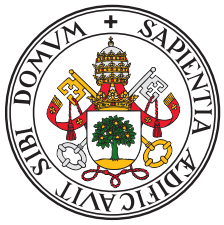


Universidad de Valladolid

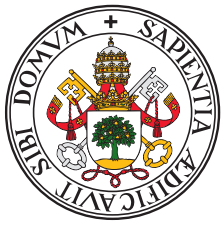
E.T.S Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática,
mención en Computación

Algoritmos para Big Data

Autor:
Sergio García Prado



Universidad de Valladolid

E.T.S Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática,
mención en Computación

Algoritmos para Big Data

Autor:
Sergio García Prado

Tutor:
Manuel Barrio Solórzano

Abstract

[TODO]

Resumen

[TODO]

Agradecimientos

[TODO]

Prefacio

Para entender el contenido de este documento así como la metodología seguida para su elaboración, se han de tener en cuenta diversos factores, entre los que se encuentran el contexto académico en que ha sido redactado, así como el tecnológico y social. Es por ello que a continuación se expone una breve descripción acerca de los mismo, para tratar de facilitar la compresión sobre el alcance de este texto.

Lo primero que se debe tener en cuenta es el contexto académico en que se ha llevado a cabo. Este documento se ha redactado para la asignatura de **Trabajo de Fin de Grado (mención en Computación)** para el *Grado de Ingeniería Informática*, impartido en la *E.T.S de Ingeniería Informática* de la *Universidad de Valladolid*. Dicha asignatura se caracteriza por ser necesaria la superación del resto de las asignaturas que componen los estudios del grado para su evaluacion. Su carga de trabajo es de **12 créditos ECTS**, cuyo equivalente temporal es de *300 horas* de trabajo del alumno, que se han llevado a cabo en un periodo de 4 meses.

La temática escogida para realizar dicho trabajo es **Algoritmos para Big Data**. El Big Data es la disciplina que se encarga de "todas las actividades relacionadas con los sistemas que manipulan grandes conjuntos de datos. Las dificultades más habituales vinculadas a la gestión de estas cantidades de datos se centran en la recolección y el almacenamiento, búsqueda, compartición, análisis, y visualización. La tendencia a manipular enormes cantidades de datos se debe a la necesidad en muchos casos de incluir dicha información para la creación de informes estadísticos y modelos predictivos utilizados en diversas materias."[Wik17a]

Uno de los puntos más importantes para entender la motivación por la cual se ha escogido dicha temática es el contexto tecnológico en que nos encontramos. Debido a la importante evolución que están sufriendo otras disciplinas dentro del mundo de la informática y las nuevas tecnologías, cada vez es más sencillo y económico recoger gran cantidad de información de cualquier proceso que se dé en la vida real. Esto se debe a una gran cantidad de factores, entre los que se destacan los siguientes:

- **Reducción de costes derivados de la recolección de información**: Debido a la constante evolución tecnológica cada vez es más barato disponer de mecanismos (tanto a nivel de hardware como de software), a partir de los cuales se puede recabar datos sobre un determinado suceso.
- **Mayor capacidad de cómputo y almacenamiento**: La recolección y manipulación de grandes cantidades de datos que se recogen a partir de sensores u otros métodos requieren por tanto del apoyo de altas capacidades de cómputo y almacenamiento. Las tendencias actuales se están apoyando en técnicas de virtualización que permiten gestionar sistemas de gran tamaño ubicados en distintas zonas geográficas como una unidad, lo cual proporciona grandes ventajas en cuanto a reducción de complejidad algorítmica a nivel de aplicación.

- **Mejora de las telecomunicaciones**: Uno de los factores que ha permitido una gran disminución de la problemática relacionada con la virtualización y su capacidad de respuesta ha sido el gran avance a nivel global que han sufrido las telecomunicaciones en los últimos años, permitiendo disminuir las barreras geográficas entre sistemas tecnológicos dispersos.

Gracias a este conjunto de mejoras se ha llegado al punto en que existe la oporturnidad de poder utilizar una gran cantidad de conocimiento, que individualmente o sin un apropiado procesamiento, carece de valor a nivel de información.

El tercer factor que es necesario tener en cuenta es la tendencia social actual, que cada vez más, está concienciada con el valor que tiene la información. Esto se ve reflejado en un amplio abanico de aspectos relacionados con el comportamiento de la población:

- **Monitorización de procesos laborales**: Muchas empresas están teniendo en cuenta la mejora de la productividad de sus empleados y máquinas. Por tanto, buscan nuevas técnicas que les permitan llevar a cabo dicha tarea. En los últimos años se ha dedicado mucho esfuerzo en implementar sistemas de monitorización que permitan obtener información para después procesarla y obtener resultados valiosos para dichas organizaciones.
- **Crecimiento exponencial de las plataformas de redes sociales**: La inherente naturaleza social del ser humano hace necesaria la expresión pública de sus sentimientos y acciones, lo cual, en el mundo de la tecnología se ha visto reflejado en un gran crecimiento de las plataformas de compartición de información así como de las de comunicación.
- **Iniciativas de datos abiertos por parte de las administraciones públicas**: Muchas insitituciones públicas están dedicando grandes esfuerzos en hacer visible la información que poseen, lo que conlleva una mejora social aumentando el grado de transparencia de las mismas, así como el nivel de conocimiento colectivo, que puede ser beneficioso tampo para ciudadanos como para empresas.

Como consecuencia de este cambio social, posiblemente propiciado por el avance tecnológico anteriormente citado, la población tiene un mayor grado de curiosidad por aspectos que antes no tenia la capacidad de entender, debido al nivel de complejidad derivado del tamaño de los conjuntos de muestra necesarios para obtener resultados fiables.

En este documento no se pretenden abordar temas relacionados con las técnicas utilizadas para recabar nuevos datos a partir de los ya existentes. A pesar de ello se realizará una breve introducción sobre dicho conjunto de estrategias, entre las que se encuentran: *Heurísticas*, *Regresión Lineal*, *Árboles de decisión*, *Máquinas de Vector Soporte (SVM)* o *Redes Neuronales Artificiales*.

Por contra, se pretende realizar un análisis acerca de los diferentes algoritmos necesarios para manejar dichas cantidades ingentes de información, en especial de su manipulación a nivel de operaciones básicas, como operaciones aritméticas, búsqueda o tratamiento de campos ausentes. Para ello, se tratará de acometer dicha problemática teniendo en cuenta estrategias de paralelización, que permitan aprovechar en mayor medida las capacidades de cómputo existentes en la actualidad.

Otro de los aspectos importantes en que se quiere orientar este trabajo es el factor dinámico necesario para entender la información, lo cual conlleva la búsqueda de nuevas estrategias algorítmicas de procesamiento en tiempo real. Por lo tanto, se pretende ilustrar un análisis acerca de las soluciones existentes en cada caso con respecto a la solución estática indicando las ventajas e inconvenientes de la versión dinámica según corresponda.

Índice general

Capítulo 1

Introducción

1.1. Ideas Iniciales

[TODO]

1.2. Motivación

[TODO]

1.3. Big Data

[TODO]

1.4. Grafos

[TODO]

1.5. Objetivos

[TODO]

Capítulo 2

Algoritmos para Streaming

2.1. Introducción

Los *algoritmos para streaming* son una estrategia de diseño algoritmica basada en el procesamiento secuencial de la entrada, lo cual encaja en marcos en los cuales los datos tienen una componente dinámica. Además, este contexto se amolda perfectamente en los casos en que el tamaño de los mismos es tan elevado que no es posible mantenerlos de manera completa en la memoria del sistema. Dicha problemática es precisamente la que surge con el denominado *Big Data*, que al trabajar con conjuntos masivos de datos en el orden de gigabytes, terabytes o incluso petabytes, no se pueden procesar utilizando estrategias clásicas que presuponen que se dispone de todos los datos de manera directa e inmediata.

Por tanto, en dicho contexto se presupone un espacio de almacenamiento en disco de tamaño infinito, mientras que se restringe el espacio de trabajo o memoria a un tamaño limitado y mucho menor que el del conjunto de datos con el que se trabaja. Mediante estas presuposiciones fijadas a priori cobra especial importancia el diseño de algoritmos en el modelo en streaming, que tratan de reducir el número de peticiones al espacio de almacenamiento o disco, lo cual genera una gran reducción en el tiempo de procesamiento.

Además, bajo este modelo es trivial realizar una extensión de los algoritmos y técnicas para su uso en entornos dinámicos en los cuales el conjunto de datos varía con respecto del tiempo, añadiendo y eliminando nuevos datos. Debido a estas carácterísticas la investigación en el campo de los *algoritmos para streaming* a cobrado una gran importancia. En este capítulo se pretende realizar una introducción conceptual acerca de los mismos, además de realizar una exposición acerca de los algoritmos más relevantes en dicha área.

2.1.1. Computación en Tiempo Real

El primer concepto del que hablaremos es **Computación en Tiempo Real**, que tal y cómo describen Shin y Ramanathan [SR94] se carácteriza por tres términos que se exponen a continuación:

- **Tiempo***(time)*: En la disciplina de *Computación en Tiempo Real* el tiempo de ejecución de una determinada tarea es especialmente crucial para garantizar el correcto desarrollo del cómputo, debido a que se asume un plazo de ejecución permitido, a partir del cual la solución del problema deja de tener validez. Shin y Ramanathan[SR94] diferencian entre tres categorías dentro de dicha restricción, a las cuales denominan *hard*, *firm* y *soft*, dependiendo del grado de relajación de la misma.

- **Confiabilidad***(correctness)*: Otro de los puntos cruciales en un sistema de *Cómputación en Tiempo Real* es la determinación de una unidad de medida o indicador acerca de las garantias de una determinada solución algorítmica para cumplir lo que promete de manera correcta en el tiempo esperado.
- **Entorno***(environment)*: El último factor que indican Shin y Ramanathan[SR94] para describir un sistema de *Computación en Tiempo Real* es el entorno del mismo, debido a que este condiciona el conjunto de tareas y la periodicidad en que se deben llevar a cabo. Por esta razón, realizan una diferenciación entre: *a*) tareas periódicas *periodic tasks* las cuales se realizan secuencialmente a partir de la finalización de una ventana de tiempo, y *b*) tareas no periódicas *periodic tasks* que se llevan a cabo debido al suceso de un determinado evento externo.

2.1.2. Problemas Dinámicos

Una vez completada la descripción acerca de lo que se puede definir como *Computación en Tiempo Real*, conviene realizar una descripción desde el punto de vista de la *teoría de complejidad computacional*. Para definir este tipo de problemas, se utiliza el término *problemas dinámicos*, los cuales consisten en aquellos en los cuales es necesario recalcular su solución conforme el tiempo avanza debido a variaciones en los parámetros de entrada del problema (Nótese que dicho término no debe confundirse con la estrategia de *programación dinámica* para el diseño de algoritmos).

Existen distintas vertientes dependiendo del punto de vista desde el que se estudien, tanto de la naturaleza del problema (soluciones dependientes temporalmente unas de otras o soluciones aisladas) como de los parámetros de entrada (entrada completa en cada nueva ejecución o variación respecto de la anterior). Los *Algoritmos para Streaming* están diseñados para resolver *problemas dinámicos*, por lo que en la sección 2.2, se describe en profundidad el modelo en que se enmarcan.

A continuación se indican los principales indicadores utilizados para describir la complejidad de una determinada solución algorítmica destinada a resolver un problema de dicha naturaleza:

- Espacio: Cantidad de espacio utilizado en memoria durante la ejecución del algoritmo.
- Inicialización: Tiempo necesario para la inicialización del algoritmo.
- Procesado: Tiempo necesario para procesar una determinada entrada.
- Consulta: Tiempo necesario para procesar la solución a partir de los datos de entrada procesados hasta el momento.

2.1.3. Algoritmos Online vs Algoritmos Offline

Una vez descrita la problemática de *Computación en Tiempo Real* en la sección 2.1.1 y la categoría de *Problemas Dinámicos* en la sección 2.1.2, en esta sección se pretende ilustrar la diferencia entre los *Algoritmos Online* y los *Algoritmos Offline*. Para ello, se ha seguido la diferenciación propuesta por Karp [Kar92], en la cual se plantea el problema de la siguiente manera (Se utilizará la misma notación que sigue Muthukrishnan[Mut05] para tratar mantener la consistencia durante todo el documento): Sea A el conjunto de datos o eventos de entrada, siendo cada $A[i]$ el elemento *i-ésimo* del conjunto, y que en el caso de los *Algoritmos Online* supondremos que es el elemento recibido en el instante i. A continuación se muestran las características de cada subgrupo:

- **Algoritmos Offline**: Esta categoría contiene todos los algoritmos que realizan el cómputo suponiendo el acceso a cualquier elemento del conjunto de datos A durante cualquier momento de su ejecución. Además, en esta categoría se impone la restricción de que el A debe ser invariante respecto del tiempo, lo que conlleva que para la adaptación del resultado a cambios en la entrada, este tenga que realizar una nueva ejecución desde su estado inicial. Nótese por tanto, que dentro de este grupo se engloba la mayoría de algoritmos utilizados comunmente.

- **Algoritmos Online**: Son aquellos que calculan el resultado a partir de una secuencia de sucesos $A[i]$, los cuales generan un resultado dependiente de la entrada actual, y posiblemente de las anteriores. A partir de dicha estrategia, se añade una componente dinámica, la cual permite que el tamaño del conjunto de datos de entrada A no tenga impuesta una restricción acerca de su longitud *a-priori*. Por contra, en este modelo no se permite conocer el suceso $A[i+1]$ en el momento i. Esto encaja perfectamente en el modelo que se describirá en la sección 2.2.

Según la diferenciación que se acaba de indicar, estas dos estrategias de diseño de algoritmos encajan en disciplinas distintas, teniendo una gran ventaja a nivel de eficiencia en el caso estático los *Algoritmos Offline*, pero quedando prácticamente inutilizables cuando la computación es en tiempo real, donde es mucho más apropiado el uso de estrategias de diseño de *Algoritmos Online*.

Como medida de eficiencia para los *Algoritmos Online*, Karp [Kar92] propone el **Ratio Competitivo**, el cual se define como la cota inferior del coste de cualquier nueva entrada con respecto de la que tiene menor coste. Sin embargo, dicha medida de eficiencia no es comúnmente utilizada en el caso de los *Algoritmos para Streaming* por la componente estocástica de los mismos, para los cuales son más apropiadas medidas probabilistas. A continuación se describen las ventajas de estos respecto de su vertiente determinista.

2.1.4. Algoritmos Probabilistas

Los *Algoritmos Probabilistas* son una estrategia de diseño que emplea en un cierto grado de aleatoriedad en alguna parte de su lógica. Estos utilizan distribuciones uniformes de probabilidad para tratar de conseguir un incremento del rendimiento en su caso promedio. A continuación se describen los dos tipos de algoritmos probabilísticos según la clasificación realizada por Babai [Bab79]:

- **Algoritmos Las Vegas**: Devuelven un resultado incorrecto con una determinada probabilidad, pero avisan del resultado incorrecto cuando esto sucede. Para contrarrestrar este suceso basta con llevar a cabo una nueva ejecución del algoritmo, lo cual tras un número indeterminado de ejecuciones produce un resultado válido.

- **Algoritmos Monte Carlo**: Fallan con un cierto grado de probabilidad, pero en este caso no avisan del resultado incorrecto. Por lo tando, lo único que se puede obtener al respecto es una indicador de la estimación del resultado correcto hacia la que converge tras varias ejecuciones. Además, se asegura una determinada cota del error ϵ, que se cumple con probabilidad δ.

La razón anecdótica por la cual Babai [Bab79] decidió denominar dichas categorías de algoritmos de esta manera se debe a lo siguiente (teniendo en cuenta el contexto de lengua inglesa): cuando se va a un casino en *Las Vegas* y se realiza una apuesta el *croupier* puede decir si se ha ganado o perdido porque habla el mismo

idioma. Sin embargo, si sucede la misma situación en *Monte Carlo*, tan solo se puede conocer una medida de probabilidad debido a que en este caso el *croupier* no puede comunicarlo por la diferencia dialéctica.

2.1.5. Algoritmos Online Probabilistas vs Deterministas

La idea subyacente acerca del diseño de los *Algoritmos Online* es la mejora de eficiencia con respecto de sus homónimos estáticos cuando el conjunto de valores de entrada es dependiente de los resultados anteriores. Sin embargo, existen casos en que la frecuencia de ejecución del algoritmo, debido a una alta tasa de llegada de valores en la entrada, las soluciones deterministas se convierten en alternativas poco escalables.

Dicha problemática se ha incrementado de manera exponencial debido al avance tecnológico y la gran cantidad de información que se genera en la actualidad, que sigue creciendo a un ritmo desorbitado. Este fenómeno ha convertido en algo necesario el diseño de estrategias basadas en técnicas probabilísticas que reduzcan en gran medida el coste computacional que como consecuencia eliminan el determinismo de la solución.

2.2. Modelo en Streaming

En esta sección se describen los aspectos formales del *Modelo en Streaming*. Para ello se ha seguido la representación definida por Muthukrishnan [Mut05]. Lo primero por tanto, es definir un flujo de datos o *Data Stream* como una "secuencia de señales digitalmente codificadas utilizadas para representar una transmisión de información" [Ins17]. Muthukrishnan [Mut05] hace una aclaración sobre dicha definición y añade la objeción de que los datos de entrada deben tener un ritmo elevado de llegada. Debido a esta razón existe complejidad a tres niveles:

- **Transmisión**: Ya que debido a la alta tasa de llegada es necesario diseñar un sistema de interconexiones que permita que no se produzcan congestiones debido a la obtención de los datos de entrada.
- **Computación**: Puesto que la tarea de procesar la gran cantidad de información que llega por unidad de tiempo produce cuellos de botella en el cálculo de la solución por lo que es necesario implementar técnicas algorítmicas con un reducido nivel de complejidad computacional para contrarrestar dicha problemática.
- **Almacenamiento**: Debido a la gran cantidad de datos que se presentan en la entrada, deben existir técnicas que permitan almacenar dicha información de manera eficiente. Esto puede ser visto desde dos puntos de vista diferences: *a*) tanto desde el punto de vista del espacio, tratando de minimizar el tamaño de los datos almacenados, maximizando la cantidad de información que se puede recuperar de ellos, *b*) como desde el punto de vista del tiempo necesario para realizar operaciones de búsqueda, addición, eliminación o edición.. Además, se debe prestar especial atención en la información que se almacena, tratando de reducirla al máximo prescindiendo de datos redundantes o irrelevantes.

2.2.1. Formalismo para Streaming

Una vez descritos los niveles de complejidad a los que es necesario hacer frente para abordar problemas en el *Modelo en Streaming*, se realiza una descripción de los distintos modelos que propone Muthukrishnan [Mut05] en los apartados 2.2.2, 2.2.3 y 2.2.4. La especificación descrita en dichos apartados será seguida durante el resto del capítulo. Para ello nos basaremos en el siguiente formalismo:

Sea $a_1, a_2, ..., a_t, ...$ un flujo de datos de entrada (*Input Stream*), de tal manera que cada elemento debe presentar un orden de llegada secuencial respecto de $t \in \mathbb{M}$. Esto también se puede ver de la siguiente manera: el elemento siguiente a la llegada de a_{t-1} debe ser a_t y, por inducción, el próximo será a_{t+1}. Es necesario aclarar que t no se refiere a unidades propiamente temporales, sino a la posición en la entrada.

$$\boldsymbol{A}_t : [1...N] \rightarrow \mathbb{R}^2 \tag{2.1}$$

El siguiente paso para describir el formalismo es añadir la función $\boldsymbol{A}_t$, cuyo dominio e imagen se muestran en la ecuación (2.1). Esta función tiene distintas interpretaciones dependientes del *Modelo en Streaming* bajo el cual se esté trabajando en cada caso, pero la idea subyacente puede resumirse asumiendo que la primera componente almacena el valor, mientras que la segunda almacena el número de ocurrencias de dicho valor. Algo común a todos ellos es la variable t, que se corresponde con resultado de la función en el instante de tiempo t. Por motivos de claridad, en los casos en que nos estemos refiriendo un único momento, dicha variable será obviada en la notación.

2.2.2. Modelo de Serie Temporal

El *Modelo de Serie Temporal* o *Time Series Model* se refiere, tal y como indica su nombre, a una serie temporal, es decir, modeliza los valores que toma la variable i respecto de t, codificados en el modelo como $a_t = (i, 1)$. Nótese que se utiliza el valor 1 en la segunda componente de a_t, la razón de ello se debe a la definición de la imagen de $\boldsymbol{A}$ en la ecuación (2.1). A pesar de ello, dicho campo es irrelevante en este modelo, por lo que se podría haber escogido cualquier otro arbitrariamente. La razón por la cual se ha utilizado el valor 1 ha sido el refuerzo de la idea de que en este caso, el valor que toma a_t en un determinado momento, no volverá a variar su valor, pero quedará obsoleto con la llegada de a_{t+1}.

El modelo se describe de manera matemática mediante la función $\boldsymbol{A}$, tal y como se ilustra en la ecuación (2.2). Textualmente, esto puede traducirse diciendo que la función $\boldsymbol{A}$ representa una estructura de datos que almacena el valor de todos los elementos recibidos en la entrada hasta el instante de tiempo t, es decir, actúa como un historial. Un ejemplo de este modelo son los valores en bolsa que toma una determinada empresa a lo largo del tiempo.

$$\boldsymbol{A}(t) = a_t \tag{2.2}$$

2.2.3. Modelo de Caja Registradora

El *Modelo de Caja Registradora* o *Cash Register Model* consiste en la recepción de incrementos de un determinado valor i. El nombre del modelo hace referencia al funcionamiento de una caja registradora (suponiendo que el pago se realiza de manera exacta), que recibe billetes o monedas de tipos diferentes de manera secuencial.

Para describir dicho modelo, previamente hay que realizar una aclaración acerca del contenido del elemento $a_t = (i, I_t)$, de manera que i representa el valor recibido, mientras que $I_t \geq 0$ indica el incremento en el instante t. Una vez aclarada esta definición, la función $\boldsymbol{A}_t$, se construye tal y como se indica en la ecuación (2.3).

$$\boldsymbol{A}_t(i) = A_{t-1}(i) + I_t \tag{2.3}$$

El *Modelo de Caja Registradora* es ampliamente utilizado en la formalización de problemas reales debido a que muchos fenómenos siguen esta estructura. Un ejemplo de ello es el conteo de accesos a un determinado sitio web, los cuales se corresponden con incrementos I_t, en este caso de caracter unitario realizados por un determinado usuario i en el momento t.

2.2.4. Modelo de Molinete

El *Modelo de Molinete* o *Turnstile Model* se corresponde con el caso más general, en el cual no solo se permiten incrementos, sino que también se pueden realizar decrementos en la cuenta. El nombre que se le dió a este modelo se debe al funcionamiento de los molinetes que hay en las estaciones de metro para permitir el paso a los usuarios, que en la entrada incrementan la cuenta del número de personas, mientras que en la salida los decrementan. La relajación originada por la capacidad de decremento ofrece una mayor versatilidad, que permite la contextualización de un gran número de problemas en este modelo. Por contra, añade un numerosas complicaciones a nivel computacional, tal y como se verá a lo largo del capítulo.

Al igual que ocurre en el caso anterior, para describir este modelo, lo primero es pensar en la estructura de los elementos en la entrada, que están formados por $a_t = (i, U_t)$, algo muy semejante a lo descrito en el *Modelo de Caja Registradora*. Sin embargo, en este caso U_t no tiene restricciones en su imagen, sino que puede tomar cualquier valor tanto positivo como negativo, lo cual añade el concepto de decremento. La construcción de la función $\boldsymbol{A}_t$ se describe en la ecuación (2.4).

$$\boldsymbol{A}_t(i) = A_{t-1}(i) + U_t \tag{2.4}$$

Muthukrishnan [Mut05] hace una diferenciación dentro de este modelo dependiendo del nivel de exigencia que se le pide al modelo, se dice que es un *Modelo de Molinete estricto* cuando se añade la restricción $\forall i, \forall t\ \boldsymbol{A}_t(i) \geq 0$, mientras que se dice que es un *Modelo de Molinete relajado* cuando dicha restricción no se tiene en cuenta.

Un ejemplo de este modelo es el conteo del número de usuarios que están visitando un determinado sitio web, tomando U_t el valor 1 en el caso de una nueva conexión y -1 en el caso de de una desconexión. En este ejemplo el valor i representa una determinada página dentro del sitio web.

2.3. Estructura básica

Puesto que la naturaleza intríseca de los *Algoritmos para Streaming* hace que procesen los elementos de entrada según van llegando, esto presenta peculiaridades con respecto de otras categorías algorítmicas más asentadas y utilizadas en la actualidad. Por tanto, primero se describirá la estructura básica que siguen los algoritmos más comunmente utilizados para después mostrar la estrategia seguida en el caso de Streaming.

Los algorítmos clásicamente estudiados para resolver la mayoría de problemas se basan la idea de funciones matemáticas. Es decir, se les presenta un conjunto de valores en la entrada, y a partir de ellos, realizan una

determinada transformación sobre los datos, que genera como resultado una salida. Nótese que esta idea no impone ninguna restricción acerca de lo que puede suceder en dicho proceso, es decir, no se restringe el uso de estructuras de datos auxiliares o técnicas similares.

Esta visión no se enmarca correctamente en el contexto de los *Algoritmos para Streaming*. La razón se debe a que la entrada no es propiamente un conjunto de datos, sino que se refiere a un flujo en sí mismo. Esta característica tiene como consecuencia que en un gran número de ocasiones ya no sea necesario obtener los resultados tras cada llamada al algoritmo, ya que estos podrían carecer de interés o requerir un sobrecoste innecesario. Por lo tanto, el concepto de función matemática pierde el sentido en este caso, ya que estas exigen la existencia de un valor como resultado.

Un concepto más acertado para modelizar un *Algoritmo para Streaming* podría ser lo que en los lenguajes de programación derivados de *Fortran* se denomina subrutina, es decir, una secuencia de instrucciones que realizan una tarea encapsulada como una unidad. Sin embargo, para poder describir correctamente la estructura de un *Algoritmo para Streaming* hace falta algo más. La razón de ello es que a partir de dicho modelo de diseño no sería posible realizar peticiones sobre el resultado calculado, es decir, sería una estrategia puramente procedural. Para corregir dicha problemática surge el concepto de consulta o *query*. A través de dicha idea se pretende representar la manera de obtener un resultado a partir del cómputo realizado hasta el momento actual.

En resumen, con dicha estrategia de diseño se consigue separar la parte de procesado de la entrada de la parte de consulta del resultado, lo cual proporciona amplias ventajas para el modelo seguido por los *Algoritmos para Streaming*. Sin embargo, dicha estrategia produce un sobrecoste espacial con respecto del modelo de algoritmo clásico. Este se debe a la necesidad de mantener una estructura de datos en la cual se almacenen los resultados parciales referentes al flujo de entrada.

Los algoritmos *Algoritmos para Streaming* se componen por tanto de algoritmo de procesamiento del flujo de datos, una estructura de datos que almacena dichos resultados, y por último, un algoritmo de procesamiento de la consulta o *query* necesaria para obtener los resultados requeridos. a continuación se dividen las fases para el funcionamiento de un algoritmo de dichas caraterísticas.

- **Inicialización**: En esta fase se llevan a cabo el conjunto de tareas necesarias para inicializar la estructura de datos que actuará como almacen de información durante el procesamiento del flujo de datos de entrada. Generalmente esto consite en el proceso de reservar memoria, inicializar a un valor por defecto la estructura de datos, etc. Sin embargo, existen técnicas más sofisticadas que requieren de una mayor carga computacional en esta fase.

- **Procesado**: Se corresponde con el procesamiento del flujo de datos de manera secuencial. La idea subyacente en esta fase es la de realizar una determinada operación sobre la estructura de datos y el elemento de entrada actual, de manera que se lleve a cabo una actualización sobre la misma. Nótese en que la manera en que se manipula dicha estructura de datos condiciona en gran medida el conjunto de peticiones que se podrán realizar sobre ella.

- **Consulta**: La fase de consulta se diferencia con respecto de la anterior por ser de carácter consultivo. Con esto nos estamos refiriendo a que dicha tarea no modifica el estado actual de la estructura de datos,

sino que recoge información de la misma, que posiblemente transforma mediante alguna operación, para despues obtener un valor como resultado de dicha petición.

2.4. Medidas de Análisis y Conceptos Matemáticos

Los *Algoritmos para Streaming* se caracterizan por utilizar propiedades estadísticas en alguna parte (generalmente en el procesado) de su cómputo para obtener la solución con un menor coste computacional. En este caso, el coste que se pretende minimizar es el referido al espacio necesario para almacenar la estructura de datos auxiliar. Tal y como se ha dicho anteriormente, la razón de ello es debida a que se presupone un conjunto masivo de datos en la entrada, por lo que se pretende que el orden de complejidad espacial respecto de la misma sea de carácter sublineal ($o(N)$).

El objetivo es encontrar soluciones con un intervalo de error acotado que permitan llegar a la solución en un orden espacial de complejidad logarítmica ($O(log(N))$). Sin embargo, existen ocasiones en que no es posible llegar a una solución en dicho orden de complejidad, como es el caso de *Algoritmos para Streaming* aplicados a problemas de *Grafos*, en los cuales se relaja dicha restricción a un orden de complejiddad *poli-logarítmico* ($O(polylog(N))$).

El órden de complejidad *poli-logarítmico* engloba el conjunto de funciones cuyo orden de complejidad presenta un crecimiento acorde a una función polinomial formada logaritmos. Matemáticamente esto se modeliza a través de la ecuación (2.5)

$$a_k \log^k(N) + \cdots + a_1 \log(N) + a_0 = O(polylog(N)) \in o(N). \tag{2.5}$$

En esta sección se muestran distintas estrategias para poder llevar a cabo la demostración de pertenencia a un determinado orden de complejidad de un *Algoritmo para Streaming*. Debido a la elevada base estadística que requieren dichas demostraciones, a continuación se definen algunos conceptos básicos relacionadas con estimadores estadísticos, para después realizar una breve demostración acerca de distintas cotas de concentración de valores en una distribución en las subsecciones 2.4.2, 2.4.3 y 2.4.4. Las definiciones que se exponen a continuación han sido extraidas de los apuntes del curso sobre *Randomized Algorithms* [Asp16] impartido por Aspnes en la *Universidad de Yale* así como las de la asignatura de *Estadística*[SJNSB16] impartida en el Grado de Ingeniería Informática de la *Universidad de Valladolid*.

2.4.1. Conceptos básicos de Estadística

Denotaremos como x_1 a una observación cualquiera contenida en el espacio de todas los posibles. Al conjunto de todas las observaciones posibles lo denotaremos como Ω y lo denominaremos espacio muestral, por lo tanto, $x \in \Omega$. Este concepto se puede entender de manera más sencilla mediante el siguiente ejemplo. Supongamos el lanzamiento de una moneda, que como resultado puede tomar los valores cara o cruz. Definiremos entonces $x_1 =$ cara y $x_2 =$ cruz como los sucesos posibles de lanzar una moneda. Por tanto el espacio Ω se define como $\Omega = \{x_1, x_2\} = \{\text{cara}, \text{cruz}\}$.

El siguiente paso es definir el concepto de **Variable Aleatoria**, que representa una función que mapea la realización de un determinado suceso sobre el espacio Ω. Dicha función se denota con letras mayúsculas y

puesto que sus parámetros de entrada son desconocidos, estos se ignoran en la notación. Por tanto denotaremos las variables aleatorias como ($\boldsymbol{E}, \boldsymbol{X}, \boldsymbol{Y}$, etc.). Para la variable aleatoria $\boldsymbol{X}$, sean $x_1, x_2, ..., x_i, ...$ cada una de las observaciones posibles. Siguiendo el ejemplo anterior, se puede modelizar el lanzamiento de una moneda como X. Nótese por tanto, que una variable aleatoria puede definirse de manera textual como la modelización del resultado de un suceso *a-priori* desconocido.

Definiremos probabilidad como la medida de certidumbre asociada a un suceso o evento futuro, expresada como un valor contenido en el intervalo $[0, 1]$, tomando el valor 0 un suceso imposible y 1 un suceso seguro. La notación seguida para representar esto será $Pr[\boldsymbol{X} = x_i]$. Suponiendo la equiprobabilidad en el ejemplo de la moneda, podemos definir sus valores de probabilidad como $Pr[\boldsymbol{X} = \text{cara}] = \frac{1}{2}$ y $Pr[\boldsymbol{X} = \text{cruz}] = \frac{1}{2}$

Una vez descritos estos conceptos simples, a continuación hablaremos sobre distintos conceptos estadísticos utilizados en el análisis de algorítmos probabilisticos tales como *Esperanza*, *Varianza*, *Variables Independientes* y *Probabilidad Condicionada*.

Denominaremos **Esperanza Matemática** al valor medio o más probable que se espera que tome una determinada variable aleatoria. La modelización matemática de dicho concepto se muestra en la ecuación (2.6). Además, la esperanza matemática es de carácter lineal, por lo que se cumplen las ecuaciones (2.7) y (2.8)

$$\mathbb{E}[\boldsymbol{X}] = \sum_{i=1}^{\infty} x_i \cdot Pr[\boldsymbol{X} = x_i] \tag{2.6}$$

$$\mathbb{E}[c\boldsymbol{X}] = c\mathbb{E}[\boldsymbol{X}] \tag{2.7}$$

$$\mathbb{E}[\boldsymbol{X} + \boldsymbol{Y}] = \mathbb{E}[\boldsymbol{X}] + \mathbb{E}[\boldsymbol{Y}] \tag{2.8}$$

La **Varianza** se define como una medida de dispersión de una variable aleatoria. Dicho estimador representa el error cuadrático respecto de la esperanza. Su modelización matemática se muestra en la ecuación (2.9). Aplicando propiedades algebraicas se puede demostrar la veracidad de las propiedades descritas en las ecuaciones (2.10) y (2.11).

$$Var[\boldsymbol{X}] = \mathbb{E}[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])^2] \tag{2.9}$$

$$Var[\boldsymbol{X}] = \mathbb{E}[\boldsymbol{X}^2] - \mathbb{E}^2[\boldsymbol{X}] \tag{2.10}$$

$$Var[c\boldsymbol{X}] = c^2 Var[\boldsymbol{X}] \tag{2.11}$$

A continuación se describe el concepto de **Independencia** entre dos variables aleatorias $\boldsymbol{X}, \boldsymbol{Y}$. Se dice que dos variables son independientes cuando los sucesos de cada una de ellas no están condicionados por los de otras. Esto puede verse a como el cumplimiento de la igualdad de la ecuación (2.12).

$$Pr[\boldsymbol{X} = x \cap \boldsymbol{Y} = y] = Pr[\boldsymbol{X} = x] \cdot Pr[\boldsymbol{Y} = y] \tag{2.12}$$

Cuando nos referimos al concepto de independencia referido a un conjunto n variables aleatorias $\boldsymbol{X_1}, \boldsymbol{X_2}, ..., \boldsymbol{X_n}$ lo denominaremos **Independencia Mutua**, que impone la restricción descrita en la ecuación (2.13).

$$Pr\left[\bigcap_{i=1}^{n} \boldsymbol{X_i} = x_i\right] = \prod_{i=1}^{n} Pr[\boldsymbol{X_i} = x_i] \tag{2.13}$$

También es de especial interés en el campo de los algoritmos probabilísticos el caso de la **k-independencia** sobre un conjunto de n variables aleatorias $\boldsymbol{X_1}, \boldsymbol{X_2}, ..., \boldsymbol{X_n}$. Dicha idea se puede resumir como la independencia de todas las variables en grupos de k variables. Este concepto tiene mucha importancia en el ámbito de los *Sketches*, tal y como se verá en la sección 3.2.4. El caso más simple es para $k = 2$, el cual se denomina **independencia pareada**, cuya modelización matemática se muestra en la ecuación (2.14).

$$\forall i, \forall j \; Pr[\boldsymbol{X_i} = x_i \cap \boldsymbol{X_j} = x_j] = Pr[\boldsymbol{X_i} = x_i] \cdot Pr[\boldsymbol{X_j} = x_j] \tag{2.14}$$

Desde el punto de vista de conjuntos de n variables aleatorias $\boldsymbol{X_1}, \boldsymbol{X_2}, ..., \boldsymbol{X_n}$, existen distintas propiedades de linearidad que se cumplen entre ellas a nivel del cálculo de la *Esperanza* y la *Varianza*. En el caso de la *Esperanza*, la linealidad respecto de la suma (ecuación (2.15)) se cumple para variables dependientes e independientes. Sin embargo, en el caso de la *Varianza*, la linealidad respecto de la suma (ecuación (2.16)) se cumple tan solo para variables **independientes pareadas**.

$$\mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}\right] = \sum_{i=1}^{n} \mathbb{E}[\boldsymbol{X_i}] \tag{2.15}$$

$$Var\left[\sum_{i=1}^{n} \boldsymbol{X_i}\right] = \sum_{i=1}^{n} Var[\boldsymbol{X_i}] \tag{2.16}$$

La **Probabilidad Condicionada** entre dos variables aleatorias $\boldsymbol{E_1}$ y $\boldsymbol{E_2}$ se puede definir como la medida de verosimilitud de la ocurrencia del suceso $\boldsymbol{E_1}$ sabiendo que ya ha ocurrido $\boldsymbol{E_2}$. Esto se puede modelizar matemáticamente tal y como se muestra en la ecuación (2.17).

$$Pr[\boldsymbol{E_1}|\boldsymbol{E_2}] = \frac{Pr[\boldsymbol{E_1} \cap \boldsymbol{E_2}]}{Pr[\boldsymbol{E_1}]} \tag{2.17}$$

En el caso de la **Probabilidad Condicionada** sobre variables independientes, surge la propiedad descrita en la ecuación (2.18). Es fácil entender la razón, que se apoya en la idea de que si dos variables aleatorias no guardan relación, entonces la ocurrencia de una de ellas, no condicionará el resultado de la otra.

$$Pr[\boldsymbol{X_1} = x_1|\boldsymbol{X_2} = x_2] = \frac{Pr[\boldsymbol{X_1} = x_1 \cap \boldsymbol{X_2} = x_2]}{Pr[\boldsymbol{X_1} = x_1]} = \frac{Pr[\boldsymbol{X_1} = x_1 \cdot \boldsymbol{X_2} = x_2]}{Pr[\boldsymbol{X_1} = x_1]} = Pr[\boldsymbol{X_2} = x_2] \tag{2.18}$$

Una vez descritos los conceptos estadísticos básicos para el análisis de algoritmos probabilísticos, lo siguiente es realizar una exposición acerca de las distintas cotas de concentración de valores, lo cual permite obtener resultados aproximados acerca de los resultados esperados por dichos algoritmos, así como sus niveles de complejidad. Primero se describirá *Desigualdad de Boole*, para después tratar las desigualdades de *Markov*(2.4.2), *Chebyshev*(2.4.3) y *Chernoff* (2.4.4)

La **Desigualdad de Boole** consiste en una propiedad básica que indica que la probabilidad de que se cumpla la ocurrencia de un suceso es menor o igual que ocurrencia de la suma de todas ellas. Esto se modeliza matemáticamente en la ecuación (2.19).

$$Pr\left[\bigcup_{i=1}^{n} \boldsymbol{E_i}\right] \leq \sum_{i=1}^{n} Pr[\boldsymbol{E_i}] \tag{2.19}$$

2.4.2. Desigualdad de Markov

La *Desigualdad de Markov* es la técnica base que utilizan otras desigualdades más sofisticadas para tratar de acotar la *Esperanza* de una determinada *Variable Aleatoria*. Proporciona una cota superior de probabilidad respecto de la *Esperanza* tal y como se muestra en la ecuación (2.20). Tal y como se puede intuir, dicha cota es muy poco ajustada, sin embargo, presenta una propiedad muy interesante como estructura base. Sea $f : \boldsymbol{X} \rightarrow \mathbb{R}^+$ una función positiva, entonces también se cumple la desigualdad de la ecuación (2.21). El punto interesante surge cuando se escoge la función f de tal manera que sea estrictamente creciente, entonces se cumple la propiedad descrita ecuación (2.22), a través de la cual podemos obtener cotas mucho más ajustadas. Dicha idea se apoya en la *Desigualdad de Jensen*.

$$\forall \lambda \geq 0,\ Pr[\boldsymbol{X} \geq \lambda] \leq \frac{\mathbb{E}[\boldsymbol{X}]}{\lambda} \tag{2.20}$$

$$\forall \lambda \geq 0,\ Pr[f(\boldsymbol{X}) \geq f(\lambda)] \leq \frac{\mathbb{E}[f(\boldsymbol{X})]}{f(\lambda)} \tag{2.21}$$

$$\forall \lambda \geq 0,\ Pr[\boldsymbol{X} \geq \lambda] = Pr[f(\boldsymbol{X}) \geq f(\lambda)] \leq \frac{\mathbb{E}[f(\boldsymbol{X})]}{f(\lambda)} \tag{2.22}$$

2.4.3. Desigualdad de Chebyshev

La *Desigualdad de Chebyshev* utiliza la técnica descrita en la subsección anterior apoyandose en al idea de la función f para obtener una cota de concentración mucho más ajustada basandose en la *Varianza*. Dicha propiedad se muestra en la ecuación (2.23). En este caso se utiliza $f(\boldsymbol{X}) = \boldsymbol{X}^2$, que es estrictamente creciente en el dominio de aplicación de $\boldsymbol{X}$. Además, se selecciona como variable aleatoria $|\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}]$, es decir, el error absoluto de una $\boldsymbol{X}$ respecto de su valor esperado. La demostración de esta idea se muestra en la ecuación (2.24).

$$\forall \lambda \geq 0,\ Pr[|\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}]| \geq \lambda] \leq \frac{Var[\boldsymbol{X}]}{\lambda^2} \tag{2.23}$$

$$\forall \lambda \geq 0,\ Pr[|\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}]| \geq \lambda] = Pr[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])^2 \geq \lambda^2] \leq \frac{\mathbb{E}[(\boldsymbol{X} - \mathbb{E}[\boldsymbol{X}])^2]}{\lambda^2} = \frac{Var[\boldsymbol{X}]}{\lambda^2} \tag{2.24}$$

2.4.4. Desigualdad de Chernoff

En este apartado se realiza una descripción acerca de la *Desigualdad de Chernoff*. Dicha descripción ha sido extraida de los apuntes de la asignatura de Algoritmos Probabilísticos (*Randomized Algorithms*) [Cha04] impartida por Shuchi Chawla en la *Carnegie Mellon University* de Pennsylvania.

La *Desigualdad de Chernoff* proporciona cotas mucho más ajustadas que por contra, exigen unas presunciones más restrictivas para poder ser utilizada. La variable aleatoria en este caso debe ser de la forma $\boldsymbol{S} = \sum_{i=1}^{n} \boldsymbol{X_i}$ donde cada $\boldsymbol{X_i}$ es una variable aleatoria uniformemente distribuida e independiente del resto. También describiremos la esperanza de cada una de las variables $\boldsymbol{X_i}$ como $\mathbb{E}[\boldsymbol{X}] = p_i$.

Denotaremos como μ a la esperanza de $\boldsymbol{S}$, tal y como se describe en la ecuación (2.25). También se define la función f como $f(\boldsymbol{S}) = e^{t\boldsymbol{S}}$.

$$\mu = \mathbb{E}\left[\sum_{i=1}^{n} \boldsymbol{X_i}\right] = \sum_{i=1}^{n} \mathbb{E}[\boldsymbol{X_i}] = \sum_{i=1}^{n} p_i \tag{2.25}$$

El siguiente paso es utilizar la ecuación (2.22) de la *Desigualdad de Markov* con la función f, que en este caso es posible puesto que es estrictamente creciente. En este caso en lugar de utilizar λ como constante, se prefiere $\delta \in [0, 1]$, que se relacionada con la anterior de la siguiente manera: $\lambda = (1 + \delta)$. Entonces la ecuación (2.26) muestra el paso inicial para llegar a la *Desigualdad de Chernoff*.

$$Pr[\boldsymbol{X} > (1 + \delta)\mu] = Pr[e^{t\mathbb{X}} > e^{(1+\delta)t\mu}] \leq \frac{\mathbb{E}[e^{t\boldsymbol{X}}]}{e^{(1+\delta)t\mu}} \tag{2.26}$$

Aplicando operaciones aritméticas y otras propiedades estadísticas, se puede demostrar la veracidad de las ecuaciones (2.27) y (2.28), que proporcionan cotas mucho muy ajustadas de concentración de la distribución de una *variable aleatoria* formada por la suma de n variables aleatorias independientes uniformemente distribuidas.

$$\forall \delta \geq 0,\ Pr[\boldsymbol{X} \geq (1 + \delta)\mu] \leq e^{\frac{-\lambda^2 \mu}{2+\lambda}} \tag{2.27}$$

$$\forall \delta \geq 0,\ Pr[\boldsymbol{X} \leq (1 - \delta)\mu] \leq e^{\frac{-\lambda^2 \mu}{2+\lambda}} \tag{2.28}$$

2.4.5. Funciones Hash

Las funciones hash son transformaciones matemáticas basadas en la idea de trasladar un valor en un espacio discreto con n posibles valores a otro de m valores de tamaño menor tratando de evitar que se produzcan colisiones en la imagen. Por tanto son funciones que tratan tratan de ser inyectivas en un subespacio de destino menor que el de partida. Sin embargo, tal y como se puede comprender de manera intuitiva es una propiedad imposible de cumplir debido a los tamaños del espacio de partida y de destino.

Las familias de funciones hash universales se refieren a distintas categorías en las cuales se pueden agrupar las funciones hash según el nivel de propiedades que cumplen. La categorías en las que se agrupan se denominan *funciones hash k-universales*, de tal manera que el valor k indican la dificultad de aparición de colisiones. Las funciones *2-universales* se refieren a aquellas en las cuales se cumple que $Pr[h(x) = h(y)] \leq 1/m$ siendo h la función hash, x e y dos posibles entradas tales que $x \neq y$ y m el cardinal del conjunto de todas las posibles claves. Se dice que una función hash es *fuertemente 2-universal* (*strongly 2-universal*) si cumple que $Pr[h(x) = h(y)] \leq 1/m^2$ y genéricamente se dice que una función es *k-universal* si cumple que $Pr[h(x) = h(y)] \leq 1/m^{-}k$. A continuación se describen dos estrategias básicas de diseño de funciones hash. Posteriormente se realiza una descripción acerca de las funciones hash sensibles a la localización.

2.4.5.1. Hash basado en Congruencias

Las funciones hash basadas en congruencias poseen la propiedad de ser *2-universales*. Se basan en la idea de utilizar como espacio de destino aquel formado por $\mathbb{Z}_p$, es decir, todos los enteros que son congruentes con p siendo $p \geq m$ un número primo. Además, se utilizan los enteros $a, b \in \mathbb{Z}_p$ con $a \neq 0$. La función hash entonces se describe tal y como se indica en la ecuación (2.29).

$$h_{ab}(x) = (ax + b) mod p \tag{2.29}$$

2.4.5.2. Hash basado en Tabulación

El hash basado en tabulación consiste en un método de generación de valores hash que restringe su dominio de entrada a cadenas de texto de tamaño fijo (u otras entradas que puedan codificarse de dicha forma). Denominaremos c al tamaño fijo y $T_i, i \in [1, c]$ a vectores que mapean el caracter i-ésimo de manera aleatoria. Entonces la función hash realiza una operación *or-exclusiva* sobre los $T_i[x_i]$ valores tal y como se indica en la ecuación (2.30). Las funciones hash que se construyen siguiendo esta estrategia poseen la propiedad pertenecer a la categoría de las *3-universales*.

$$h(x) = T_1[x_1] \oplus T_2[x_2] \oplus ... \oplus T_c[x_c] \tag{2.30}$$

2.4.5.3. Funciones Hash sensibles a la localización

Una categoría a destacar son las *funciones hash sensibles a la localización*. Estas poseen la propiedad de distribuir los valores cercanos en el espacio de destino tratando mantener las propiedades de cercanía entre los valores. El primer artículo en que se habla de ellas es *Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality* [IM98] de *Indyk* y *Motwani* inicialmente para resolver el problema de la *búsqueda del vecino más cercano (Nearest Neighbor Search)*. Se trata de funciones hash multidimensionales, es decir, en la entrada están compuestas por más de un elemento. Estas funciones han cobrado especial importancia en los últimos años por su uso en problemas de dimensión muy elevada, ya que sirven como estrategia de reducción de la dimensionalidad del conjunto de datos, que como consecuencia reduce el coste computacional del problema.

Indyk indica que para que una función hash sea sensible a la localización o (r_1, r_2, p_1, p_2)-sensible, para cualquier par de puntos de entrada p, q y la función de distancia d se cumpla la ecuación (2.31). Las funciones hash que cumplen esta propiedad son interesantes cuando $p_1 > p_2$ y $r_1 < r_2$, de tal forma que para puntos cercanos en el espacio, el valor hash obtenido es el mismo, por lo que se asume que dichos puntos se encuentran cercanos en el espacio de partida.

$$\text{if } d(p, q) \leq r_1, \text{ then } Pr[h(x) = h(y)] \geq p_1 \tag{2.31}$$

$$\text{if } d(p, q) \geq r_2, \text{ then } Pr[h(x) = h(y)] \leq p_2 \tag{2.32}$$

Una vez descritos los conceptos básicos acerca de lo que son los *Algoritmos para Streaming* y las bases estadísticas necesarias para poder entender el funcionamiento de los mismo y sus niveles de complejidad así como de precisión, en las siguientes secciones se realiza una descripción sobre algunos de los algoritmos más

relevantes en este área. En especial se explica el algoritmo de *Morris* en la sección 2.5, el de *Flajolet-Martin* en la 2.5 y por último se hablará de la *Estimación de Momentos de Frecuencia* en el modelo en streaming en la sección 2.7.

2.5. Algoritmo de Morris

El *Algoritmo de Morris* fue presentado por primera vez en el artículo *Counting Large Numbers of Events in Small Registers* [Mor78] redactado por *Robert Morris*. En dicho documento se trata de encontrar una solución al problema de conteo de ocurrencias de un determinado suceso teniendo en cuenta las restriciones de espacio debido a la elevada tasa de ocurencias que se da en muchos fenómenos. El problema del conteo (**Count Problem**) de ocurrencias también se denomina el momento de frecuencia F_1 tal y como se verá en la sección 2.7.

Por tanto, *Morris* propone realizar una estimación de dicha tasa para reducir el espacio necesario para almacenar el valor. Intuitivamente, a de partir dicha restricción se consigue un orden de complejidad espacial sublineal ($o(N)$) con respecto del número de ocurrencias. Se puede decir que el artículo publicado por *Morris* marcó el punto de comienzo de este área de investigación. El conteo probabilista es algo trivial si se restringe a la condición de incrementar el conteo de ocurrencias siguiendo una distribución de *Bernoulli* con un parámetro p prefijado previamente. Con esto se consigue un error absoluto relativamente pequeño con respecto al valor p escogido. Sin embargo, el error relativo que se obtiene cuando el número de ocurrencias es pequeño es muy elevado, lo cual lo convierte en una solución impracticable.

Para solucionar dicha problemática y conseguir una cota del error relativo reducida, la solución propuesta por *Morris* se basa en la selección del parámetro p variable con respecto del número de ocurrencias, con lo cual se consigue que la decisión de incrementar el contador sea muy probable en los primeros casos, lo cual elimina el problema del error relativo. *Morris* propone aumentar el contador X con probabilidad $\frac{1}{2^X}$. Tras n ocurrencias, el resultado que devuelve dicho algoritmo es $\widetilde{n} = 2^X - 1$. El pseudocódigo se muestra en el algoritmo 1.

Algorithm 1: Morris-Algorithm

Result: $\widetilde{n} = 2^X - 1$

```
X ← 0;
for cada evento do
    X ← X + 1 con probabilidad 1/2^X;
end
```

A continuación se realiza un análisis de la solución. Esta ha sido extraida de los apuntes de la asignatura *Algorithms for Big Data* [Nel15] impartida por *Jelani Nelson* en la *Universidad de Harvard*. Denotaremos por X_n el valor del contador X tras n ocurrencias. Entonces se cumplen las igualdades descritas en las ecuaciones (2.33) y (2.34). Esto se puede demostrar mediante técnicas inductivas sobre n.

$$\mathbb{E}[2^{X_n}] = n + 1 \tag{2.33}$$

$$\mathbb{E}[2^{2X_n}] = \frac{3}{2}n^2 + \frac{3}{2}n + 1 \tag{2.34}$$

Por la *Desigualdad de Chebyshev* podemos acotar el error cometido tras n repeticiones, dicha formulación se muestra en la ecuación (2.35).

$$Pr[|\widetilde{n} - n| > \epsilon n] < \frac{1}{\epsilon^2 n^2} \cdot \mathbb{E}[\widetilde{n} - n]^2 = \frac{1}{\epsilon^2 n^2} \cdot \mathbb{E}[2^X - 1 - n]^2 \tag{2.35}$$

$$= \frac{1}{\epsilon^2 n^2} \cdot \frac{n^2}{2} \tag{2.36}$$

$$= \frac{1}{2\epsilon^2} \tag{2.37}$$

La ventaja de esta estrategia algorítmica con respecto de la trivial es la cota del error relativo producido en cada iteracción del algoritmo, lo cual aporta una mayor genericidad debido a que esta se mantiene constante con respecto del número de ocurrencias. Sin embargo, se han propuesto otras soluciones para tratar de reducir en mayor medida dicha cota de error. El algoritmo *Morris+* se basa en el mantenimiento de s copias independientes de *Morris* para después devolver la media del resultado de cada una de ellas. A partir de esta estrategia se consiguen las tasas de error que se indican en la ecuación (2.38).

$$Pr[|\widetilde{n} - n| > \epsilon n] < \frac{1}{2s\epsilon^2} \qquad\qquad s > \frac{3}{2\epsilon^2} \tag{2.38}$$

2.6. Algoritmo de Flajolet-Martin

En esta sección se describe el *Algoritmo de Flajolet-Martin*, cuya descripción aparece en el artículo *Probabilistic Counting Algorithms for Data Base Applications* [FM85] redactado por *Philippe Flajolet* y *G. Nigel Martin*. En este caso, la problemática que se pretende resolver no es el número de ocurrencias de un determinado suceso, sino el número de sucesos distintos (**Count Distinct Problem**) en la entrada. Este problema también se conoce como el cálculo del momento de frecuencia F_0 tal y como se verá en la sección 2.7. Al igual que en el caso del algoritmo de *Morris*, se apoya en estrategias probabilistas para ajustarse a un orden de complejidad espacial de carácter sublineal ($o(N)$)) manteniendo una cota de error ajustada.

La intuicción en la cual se basa el *Algoritmo de Flajolet-Martin*, es la transformación de los elementos de entrada sobre una *función Hash* universal binaria con distribución uniforme e independiente de probabilidad. La propiedad de distribución uniforme permite entonces preveer que la mitad de los elementos tendrán un 1 en el bit menos significativo, que una cuarta parte de los elementos tendrán un 1 en el segundo bit menos significativo y así sucesivamente. Por tanto, a partir de esta idea se puede realizar una aproximación probabilista del número de elementos distintos que han sido presentados en la entrada. Requiere de L bits de espacio para el almacenamiento del número de elementos distintos. Por la notación descrita en anteriores secciones $L = log(n)$, donde n es el número máximo de elementos distintos en la entrada. A continuación se explica esta estrategia, para ello nos apoyaremos en las siguientes funciones:

- $hash(x)$ Es la función hash con distribución uniforme e independiente de probabilidad que mapea una entrada cualquiera a un valor entero en el rango $[0, ..., 2^L - 1]$.

- $bit(y,k)$ Esta función devuelve el bit *k-ésimo* de la representación binaria de y, de tal manera que se cumple que $y = \sum_{k\geq 0} bit(y,k)2^k$
- $\rho(y)$ La función ρ devuelve la posición en la cual se encuentra el bit con valor 1 empezando a contar a partir del menos significativo. Por convenio, devuelve el valor L si y no contiene ningún 1 en su representación binaria, es decir, si $y = 0$. Esto se modeliza matemáticamente en la ecuación (2.39).

$$\rho(y) = \begin{cases} min_{k\geq 0} bit(y,k) \neq 0 & y \geq 0 \\ L & y = 0 \end{cases} \tag{2.39}$$

Flajolet y *Martin* se apoyan en una estructura de datos indexada a la cual denominan **BITMAP**, de tamaño $[0...L-1]$ la cual almacena valores binarios $\{0,1\}$ y se inicializa con todos los valores a 0. Nótese por tanto, que esta estructura de datos puede ser codificada como un string binario de longitud L. La idea del algoritmo es marcar con un 1 la posición $\mathbf{BITMAP}[\rho(hash(x))]$. Seguidamente, queda definir el resultado de la consulta sobre cuántos elementos distintos han aparecido en el flujo de datos de entrada. Para ello se calcula $2^{\rho(\mathbf{BITMAP})}$. El pseudocódigo se muestra en el algoritmo 2.

Algorithm 2: FM-Algorithm

Result: $2^{\rho(\mathbf{BITMAP})}$

```
for i ∈ [0, ..., L − 1] do
    BITMAP[i] ← 0;
end
for cada evento do
    if BITMAP[ρ(hash(x))] = 0 then
        BITMAP[ρ(hash(x))] ← 1;
    end
end
```

El análisis de esta solución ha sido extraido de los apuntes del libro *Mining of massive datasets* [LRU14] de la *Universidad de Cambridge*. En este caso lo representa teniendo el cuenta el número de 0's seguidos en la parte menos significativa de la representación binaria de $h(x)$. Nótese que esto es equivalente al valor de la función $\rho(h(y))$, por tanto, adapteremos dicho análisis a la solución inicial propuesta por *Flajolet* y *Martin*. La probabilidad de que se cumpla $\rho(h(x)) = r$ es 2^{-r}. Supongamos que el número de elementos distintos en el stream es m. Entonces la probabilidad de que ninguno de ellos cumpla $\rho(h(x)) = r$ es al menos $(1-2^{-r})^m$ lo cual puede ser reescrito como $((1-2^{-r})^{2^r})^{m2^{-r}}$. Para valores suficientemente grandes r se puede asumir que dicho valor es de la forma $(1-\epsilon)^{1/\epsilon} \approx 1/\epsilon$. Entonces la probabilidad de que no se cumpla que $\rho(h(x)) = r$ cuando han aparecido m elementos distintos en el stream es de $e^{-m2^{-r}}$

La problemática de este algoritmo deriva de la suposición de la capacidad de generación de claves Hash totalmente aleatorias, lo cual no se ha conseguido en la actualidad. Por lo tanto posteriormente, *Flajolet* ha seguido trabajando el problema de conteo de elementos distintos en *Loglog counting of large cardinalities* [DF03] y *Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm* [FFGM07] para tratar

de mejorar el grado de precisión de su estrategia de conteo. En el artículo *An optimal algorithm for the distinct elements problem* [KNW10] *Daniel Kane y otros* muestran un algoritmo óptimo para el problema. Los resultados de dichos trabajos se discuten en la sección 3.6 por su cercana relación con las *estructuras de datos de resumen.*

2.7. Aproximación a los Momentos de Frecuencia

La siguiente idea de la que es interesante hablar para terminar la introducción a los *Algoritmos para Streaming* son los *Momentos de Frecuencia.* Una generalización de los conceptos del número de elementos distintos (F_0) y el conteo de elementos (F_1) que se puede extender a cualquier F_k para $k \geq 0$. La definición matemática del momento de frecuencia k-ésimo se muestra en la ecuación (2.40). Nótese el caso especial de F_∞ que se muestra en la ecuación (2.41) y se corresponde con el elemento más veces común en el *Stream.* Estas ideas han sido extraidas del documento *Frequency Moments* [Woo09] redactado por *David Woodruff.*

$$F_k = \sum_{i=1}^{n} m_i^k \tag{2.40}$$

$$F_\infty = max_{1 \leq i \leq n} m_i \tag{2.41}$$

El resto de la sección trata sobre la exposición de los algoritmos para el cálculo de los momentos de frecuencia descritos en el articulo *The space complexity of approximating the frequency moments* [AMS96] redactado por *Noga Alon*, *Yossi Matias* y *Mario Szegedy*, por el cual fueron galardonados con el premio *Gödel* en el año 2005. En dicho trabajo además de presentar *Algoritmos para Streaming* para el cálculo de F_k (cabe destacar su solución para F_2), también presentan cotas inferiores para el problema de los *Momentos de Frecuencia.* Posteriormente *Piotr Indyk* y *David Woodruff* encontraron un algoritmo óptimo para el problema de los*Momentos de Frecuencia* tal y como exponen en *Optimal Approximations of the Frequency Moments of Data Streams* [IW05]. A continuación se discuten los resultados de dichos trabajos.

Para el cálculo de F_k para $k \geq 0$ *Alon*, *Matias* y *Szedgedy* proponen un enfoque similar a los propuestos en algoritmos anteriores (la definición de una variable aleatoria X tal que $\mathbb{E}[X] = F_k$). Sin embargo, la novedad en este caso es que su algoritmo no está restringido a un k concreto, sino que en su caso es generalizable para cualquier entero positivo, sin embargo, en este caso la exposición es a nivel teórico.

Definiremos las constantes $S_1 = O(n^{1-1/k}/\lambda^2)$ y $S_2 = O(\log(1/\varepsilon))$. El algoritmo utiliza S_2 variables aleatorias denominadas Y_1, Y_2, Y_{S_2} y devuelve la mediana de estas denominandola Y. Cada una de estas variables Y_i está formada por la media de X_{ij} variables aleatorias tales que $1leqjleqS_1$. La forma en que se actualiza el estado del algoritmo tras cada nueva llegada se apoya en el uso de S_2 funciones hash uniformemente distribuidas e independientes entre si que mapean cada símbolo a un determinado indice j.

Para el análisis del algoritmos supondremos que el tamaño n del *Stream* es conocido *a-priori.* La demostración se apoya en una variable aleatoria X construida de la siguiente manera:

- Seleccionaremos de manera aleatoria el elemento $a_{p \in (1,2,...,m)}$ del Stream, siendo $a_p = l \in (1, 2, ...n)$. Es decir, el elemento procesado en el momento p representa la llegada del símbolo l.

- Definiremos $r = |\{q : q \geq p, a_p = l\}|$ como el número de ocurrencias del símbolo l hasta el momento p.
- La variable aleatoria X se define como $X = m(r^k - (r-1)^k)$

Desarrollando la Esperanza Matemática de la variable aleatoria X se puede demostrar que esta tiende al momento de frecuencia k tal y como se muestra en la ecuación (2.42).

$$\begin{aligned}
\mathbb{E}(X) &= \textstyle\sum_{i=1}^{n} \sum_{i=1}^{m_i} (j^k - (j-1)^k) \\
&= \tfrac{m}{m}[(1^k + (2^k - 1^k) + \ldots + (m_1^k - (m_1 - 1)^k)) \\
&\quad + (1^k + (2^k - 1^k) + \ldots + (m_2^k - (m_2 - 1)^k)) + \ldots \\
&\quad + (1^k + (2^k - 1^k) + \ldots + (m_n^k - (m_n - 1)^k))] \\
&= \textstyle\sum_{i=1}^{n} m_i^k = F_k
\end{aligned} \tag{2.42}$$

En cuanto al coste espacial del algoritmo, se puede demostrar tal y como indican *Alon*, *Matias* y *Szedgedy* en su artículo original [AMS96] que este sigue el orden descrito en la ecuación (2.43) puesto que es necesario almacenar a_p y r lo cual requiere de $log(n) + log(m)$ bits de memoria, además de $S_1 x S_2$ variables aleatorias para mantener X.

$$O\left(\frac{k \log \frac{1}{\varepsilon}}{\lambda^2} n^{1-\frac{1}{k}} (\log n + \log m)\right) \tag{2.43}$$

2.8. Conclusiones

Tal y como se ha ilustrado a lo largo del capítulo, los *algoritmos para streaming* son una solución adecuada tanto a nivel conceptual como práctica para problemas en los cuales el tamaño del conjunto de datos de entrada es tan elevado que no se puede hacer frente mediante estrategias clásicas. Por contra, estas soluciones presentan dificultades debido al elevado peso de la componente matemática y estadística que presentan. Además, la imprecisión en sus resultados restringe su uso en casos en los cuales la precisión es un requisito imprescindible por lo que tan solo deben ser utilizados en casos en los cuales no existan otras soluciones que calculen una solución con las restricciones de tiempo y espacio impuestas.

En este capítulo se ha realizado una introducción superficial acerca de este modelo, sin embargo las implementaciones y descripciones que se muestran tan solo gozan de importancia a nivel teórico y conceptual. Por lo tanto, en el capítulo 3 se continua la exposición de técnicas de tratamiento de grandes cantidades de datos desde una perspectiva más práctica hablando de las *estructuras de datos de resumen*. Se describen en especial detalle las estructuras basadas en *sketches*, que internamente utilizan *algoritmos para streaming*.

Capítulo 3

Estrategias de Sumarización

3.1. Introducción

El gran crecimiento tecnológico que se está llevando a cabo en la actualidad a todos los niveles está propiciando además un aumento exponencial en cuanto a la cantidad de información que se genera. La reducción de costes en cuanto a la instalación de sensores que permiten recoger información de muchos procesos productivos, así como la obtenición de metadatos a partir del uso de internet y las redes sociales por parte de los usuarios hace que el ritmo de crecimiento en cuanto a cantidad información generada por unidad de tiempo haya crecido a un ritmo frenético.

Una de las razones que han facilitado dicha tendencia es la disminución de costes de almacenamiento de información, a la vez que han aumentado las capacidades de cómputo necesarias para procesarla. Sin embargo, debido al crecimiento exponencial de los conjuntos de datos, es necesario investigar nuevas técnicas y estrategias que permitan obtener respuestas satisfactorias basadas en la gran cantidad de información de la que se dispone en un tiempo razonable.

Tradicionalmente, la investigación en el campo de las *bases de datos* se ha centrado en obtener respuestas exactas a distintas consultas, tratando de hacerlo de la manera más eficiente posible, así como de tratar de reducir el espacio necesario para almacenar la información. *Acharya y otros* proponen en el artículo *Join synopses for approximate query answering* [AGPR99] el concepto de *Approximate Query Processing*. Dicha idea se expone en la subsección 3.1.1.

3.1.1. Approximate Query Processing

El *procesamiento de consultas aproximado*, (*Approximate Query Processing* o **AQP**) se presenta como una estrategia de resolución de consultas basada en conceptos y propiedades estadísticas que permiten una gran reducción en la complejidad computacional y espacial necesaria para la resolución de las mismas por una base de datos. Por contra, dicha reducción a tiene como consecuencia la adicción de un determinado nivel de imprecisión en los resultados a la cual se denomina error. Se pretende que dicho error pueda ser acotada en un intervalo centrado en el valor verdadero con una desviación máxima determinada por ϵ, y que la pertenencia de la solución a este intervalo se cumpla con una probabilidad δ. Al igual que en el anterior capítulo, en este caso también se presta especial importancia a la minimización del error relativo, lo cual consigue que las soluciones mediante el *procesamiento de consultas aproximado* sean válidas tanto para consultas de tamaño

reducido como de gran tamaño.

Durante el resto del capítulo se describen y analizan distintas estrategias que permiten llevar a cabo implementaciones basadas en *procesamiento de consultas aproximado* centrando especial atención en los *Sketches* por su cercanía respecto del *Modelo en Streaming* descrito en el capítulo 2. En la sección 3.2 se realiza una descripción a partir de la cual se pretende aclarar las diferencias entre las distintas estrategias de sumarización de grandes conjuntos de datos. Las estrategias que se describen son *muestreo probabilístico*, mantenimiento de un *histograma*, utilización de *wavelets* y por último se describen en profundidad conceptos referidos a *Sketches*. En las secciones 3.3, 3.4, 3.5 y 3.6 se habla del *Bloom-Filter Count-Min Sketch*, *Count Sketch* y *HyperLogLog* respectivamente.

3.2. Tipos de Estrategias de Sumarización

Para el diseño de soluciones basadas en *procesamiento de consultas aproximado* existen distintas estrategias, cada una de las cuales presentan ventajas e inconvenientes por lo que cada una de ellas es más conveniente para una determinada tarea, sin embargo en ocasiones surgen solapamientos entre ellas tal y como se pretende mostrar en esta sección. Dichas descripciones han sido extraidas del libro *Synopses for massive data* [CGHJ12] redactado por *Cormode y otros*. En las secciones 3.2.1, 3.2.2, 3.2.3 y 3.2.4 se habla de *Sampling*, *Histogram*, *Wavelet* y *Sketches* respectivamente.

3.2.1. Sampling

El *Sampling* o *muestreo probabilístico* es la estrategia más consolidada de entre las que se presentan. Las razones se deben a su simplicidad conceptual así como su extendido uso en el mundo de la estadística. Uno de los primeros artículos en que se trata el muestreo aplicado a bases de datos es *Accurate estimation of the number of tuples satisfying a condition* [PSC84] redactado por *Piatetsky-Shapiro* y *Connell*. La intuición en que se basa esta solución es la selección de un subconjunto de elementos denominado *muestra* extraida del conjunto global al cual se denomina *población*. Una vez obtenida la *muestra* del conjunto de datos global, cuyo tamaño es significativamente menor respecto del global (lo cual reduce drásticamente el coste computacional), se realizan los cálculos que se pretendía realizar sobre toda la *población* para después obtener un estimador del valor real que habría sido calculado al realizar los cálculos sobre el conjunto de datos global.

Para que las estrategias de sumarización de información obtengan resultados válidos o significativos respecto del conjunto de datos, es necesario que se escojan adecuadamente las instancias de la *muestra*, de manera que se maximice la similitud del resultado respecto del que se habría obtenido sobre toda la población. Para llevar a cabo dicha labor existen distintas estrategias, desde las más simples basadas en la selección aleatoria sin reemplazamiento como otras mucho más sofisticadas basadas en el mantenimiento de *muestras* estratificadas. Sea R la población y $|R|$ el tamaño de la misma. Denominaremos t_j al valor j-ésimo de la población y X_j al número de ocurrencias del mismo en la *muestra*. A continuación se describen distintas técnicas de muestreo:

- **Selección Aleatoria Sin Reemplazamiento**: Consiste en la estrategia más simple de generación de *muestras*. Se basa en la selección aleatoria de un valor entero r en el rango $[1, |R|]$ para después añadir el elemento localizado en la posición r de la *población* al subconjunto de la *muestra*. Este proceso se repite durante n veces para generar una *muestra* de tamaño n. A modo de ejemplo se muestra el

estimador para la operación *SUMA* en la ecuación (3.1), además se muestra la fórmula de la desviación para dicho estimador en la ecuación (3.2).

$$Y = \frac{|R|}{n} \sum_j X_j t_j \tag{3.1}$$

$$\sigma^2(Y) = \frac{|R|^2 \sigma^2(R)}{n} \tag{3.2}$$

- **Selección Aleatoria Con Reemplazamiento**: En este caso se supone que la selección de una instancia de la población tan solo se puede llevar a cabo una única vez, por lo tanto se cumple que $\forall X_j \in 0, 1$. La selección se lleva a cabo de la siguiente manera: se genera de manera aleatoria un valor entero r en el rango $[1, |R|]$ para después añadir el elemento localizado en la posición r de la *población* al subconjunto de *muestra* si este no ha sido añadido ya, sino volver a generar otro valor r. Después repetir dicha secuencia durante n veces para generar una *muestra* de tamaño n. Al igual que en la estrategia anterior, en este caso también se muestra el estimador para la operación *SUMA* en la ecuación (3.3). Nótese que el cálculo es el mismo que en el caso de la estrategia sin reemplazamiento. Sin embargo, la varianza obtenida a partir de dicha estrategia es menor tal y como se muestra en la ecuación (3.4).

$$Y = \frac{|R|}{n} \sum_j X_j t_j \tag{3.3}$$

$$\sigma^2(Y) = \frac{|R|(|R| - n)\sigma^2(R)}{n} \tag{3.4}$$

- **Bernoulli y Poisson**: Mediante esta alternativa de muestreo se sigue una estrategia completamente distinta a las anteriores. En lugar de seleccionar la siguiente instancia aleatoriamente de entre todas las posibles, se decide generar $|R|$ valores aleatorios r_j independientes en el intervalo $[0, 1]$ de tal manera que si r_j es menor que un valor p_j fijado a priori, la instancia se añade al conjunto de *muestra*. Cuando se cumple que $\forall i, j\ p_i = p_j$ se dice que es un muestreo de *Bernoulli*, mientras que cuando no se cumple dicha condición se habla de muestreo de *Poisson*. El cálculo del estimador para la *SUMA* en este caso es muy diferente de los ilustrados anteriormente tal y como se muestra en la ecuación (3.5). La desviación de este estimador se muestra en la ecuación (3.6), que en general presenta peores resultados (mayor desviación) que mediante otras alternativas, sin embargo, esta posee la cualidad de poder aplicar distintos pesos a cada instancia de la población, lo que puede traducirse en que una selección adecuada de los valores p_j, lo cual puede mejorar significativamente la precisión de los resultados si estos se escogen de manera adecuada.

$$Y = \sum_{i \in muestra} \frac{t_i}{p_i} \tag{3.5}$$

$$\sigma^2(Y) = \sum_i (\frac{1}{p_i} - 1) t_i^2 \tag{3.6}$$

- **Muestreo Estratificado**: El muestreo estratificado trata de minimizar al máximo las diferencias entre la distribución del conjunto de datos de la *población* respecto de la *muestra* que se pretende generar. Para ello existen distintas alternativas entre las que se encuentra una selección que actualiza los pesos p_j tras cada iteracción, lo que reduce drásticamente la desviación de la *muestra*, sin embargo produce un elevado coste computacional para su generación. Por tanto, existen otras estrategia más intuitivas

basada en la partición del conjunto de datos de la *población* en subconjuntos disjuntos que poseen la cualidad de tener varianza mínima a los cuales se denomina *estratos*. Posteriormente, se selecciona mediante cualquiera de los métodos descritos anteriormente una *muestra* para cada *estrato*, lo cual reduce en gran medida la desviación típica global del estimador.

La estrategia de sumarización de información mediante *muestreo* tiene como ventajas la independencia de la complejidad con respecto a la dimensionalidad de los datos (algo que como se ilustrará con en posteriores secciones no sucede con el resto de alternativas) además de su simplicidad conceptual. También existen cotas de error para las consultas, para las cuales no ofrece restricciones en cuanto al tipo de consulta (debido a que se realizan sobre un subconjunto con la misma estructura que el global). El muestre es apropiado para conocer información general acerca del conjunto de datos. Además, presenta la cualidad de permitir su modificación y adaptación en tiempo real, es decir, se pueden añadir o eliminar nuevas instancias de la muestra conforme se añaden o eliminan del conjunto de datos global.

Sin embargo, en entornos donde el ratio de addiciones/eliminaciones es muy elevado el coste computacional derivado del mantenimiento de la muestra puede hacerse poco escalable. El *muestreo* es una buena alternativa para conjuntos de datos homogéneos, en los cuales la presencia de valores atípicos es irrelevante. Tampoco obtiene buenos resultados en consultas relacionadas con el conteo de elementos distintos. En las siguientes secciones se describen alternativas que resuelven de manera más satisfactoria estas dificultades y limitaciones.

3.2.2. Histogram

Los *histogramas* son estructuras de datos utilizadas para sumarizar grandes conjuntos de datos mediante el mantenimiento de tablas de frecuencias, por lo que tienen un enfoque completamente diferente al que siguen las estrategias de *muestreo* de la sección anterior. En este caso, el concepto es similar a la visión estadística de los histogramas. Consiste en dividir el dominio de valores que pueden tomar las instancias del conjunto de datos en intervalos o contenedores disjuntos entre si de tal manera que se mantiene un conteo del número de instancias pertenecientes a cada partición.

Durante el resto de la sección se describen de manera resumida distintas estrategias de estimación del valor de las particiones, así como las distintas estrategias de particionamiento del conjunto de datos. Para llevar a cabo dicha tarea, a continuación se describe la notación que se ha seguido en esta sección: Sea D el conjunto de datos e $i \in [1, M]$ cada una de las categorías que se presentan en el mismo. Denotaremos por $g(i)$ al número de ocurrencias de la categoría i. Para referirnos a cada uno de las particiones utilizaremos la notación S_j para $j \in [1, B]$. Nótese por tanto que M representa el cardinal de categorías distintas mientras que B representa el cardinal de particiones utilizadas para "comprimir" los datos. La mejora de eficiencia en cuanto a espacio se consigue devido a la eleccion de $B \ll M$

Cuando se hablamos de *esquemas de estimación* nos estamos refiriendo a la manera en que se almacena o trata el contenido de cada una de las particiones S_j del histograma. La razón por la cual este es un factor importante a la hora de caracterizar un histograma es debida a que está altamente ligada a la precisión del mismo.

- **Esquema Uniforme**: Los esquemas que presuponen una distribución uniforme de las instancias dentro del contenedor se subdividen en dos categorías: *a*) *continous-value asumption* que presupone que

todas las categorías i contenidas en la partición S_j presentan el mismo valor para la función $g(i)$ y *b*) *uniform-spread asumption* que presupone que el número de ocurrencias de la partición S_j se localiza distribuido uniformemente al igual que en el caso anterior, pero en este caso entre los elementos de un subconjunto P_j generado iterando con un determinado desplazamiento k sobre las categorías i contenidas en S_j. El segundo enfoque presenta mejores resultados en el caso de consultas de cuantiles que se distribuyen sobre más de una partición S_j

- **Esquema Basado en Splines**: En la estrategia basada en splines se presupone que los valores se distribuyen conforme una determinada función lineal de la forma $y_j = a_j x_j + b_j$ en cada partición S_j de tal manera que el conjunto total de datos D puede verse como una función lineal a trozos y continua, es decir, los extremos de la función en cada partición coinciden con el anterior y el siguiente. Nótese que en este caso se ha descrito la estrategia suponiendo el uso de una función lineal, sin embargo esta puede extenderse a funciones no lineales.

- **Esquema Basado en Árboles**: Consiste en el almacenamiento de las frecuencias de cada partición S_j en forma de árbol binario, lo cual permite seleccionar de manera apropiada el nivel del árbol que reduzca el número de operaciones necesarias para obtener la estimación del conteo de ocurrencias según el la amplitud del rango de valores de la consulta. La razón por la cual se escoje un árbol binario es debida a que se puede reducir en un orden de 2 el espacio necesario para almacenar dichos valores manteniendo únicamente los de una de las ramas de cada sub-árbol. La razón de ello es debida a que se puede recalcular el valor de la otra mediante una resta sobre el valor almacenado en el nodo padre y la rama que si contiene el valor.

- **Esquema Heterogéneo**: El esquema heterogéneo se basa la intuición de que la distribución de frecuencias de cada una de las particiones S_j no es uniforme y tiene peculiaridades propias, por lo tanto sigue un enfoque diferente en cada una de ellas tratanto de minimizar al máximo la tasa de error producida. Para ello existens distintas heurísticas basadas en distancias o teoría de la información entre otros.

Una vez descritas distintas estrategias de estimación del valor de frecuencias de una determinada partición S_j, el siguiente paso para describir un *histograma* es realizar una descripción acerca de las distintas formas de generación de las particiones o contenedores. Para tratar de ajustarse de manera más adecuada a la distribución de los datos se puede realizar un *muestreo* a partir del cual se generan las particiones. A continuación se describen las técnicas más comunes para la elaboración de dicha tarea:

- **Particionamiento Heurístico**: Las estrategias de particionamiento heurístico se basan en el apoyo sobre distintas presuposiciones que en la práctica han demostrado comportamientos aceptables en cuanto al nivel de precisión que se obtiene en los resultados, sin embargo, no proporcionan ninguna garantía desde el punto de vista de la optimalidad. Su uso está ampliamente extendido debido al reducido coste computacional. Dentro de esta categoría las heurísticas más populares son las siguientes:

  - **Equi-Width**: Consiste en la división del dominio de categorías $[1, M]$ en particiones equi-espaciadas unas de otras. Para dicha estrategia tan solo es necesario conocer *a-priori* el rango del posible conjunto de valores. Es la solución con menor coste computacional, a pesar de ello sus resultados desde el punto de vista práctico son similares a otras estrategias más sofisticadas cuando la distribución de frecuencias es uniforme.

- **Equi-Depth**: Esta estrategia de particionamiento requiere conocer la distribución de frecuencias *a-priori* (o una aproximación que puede ser obtenida mediante alguna estrategia como el muestreo). Se basa en la división del dominio de valores de tal manera que las particiones tengan la misma frecuencia. Para ello se crean particiones de tamaños diferentes.
- **Singleton-Bucket**: Para tratar de mejorar la precisión esta estrategia de particionamiento se basa en la utilización de dos particiones especiales, las cuales contienen las categorías de mayor y menor frecuencia respectivamente para después cubrir el resto de categorías restante mediante otra estrategia (generalmente *equi-depth*) lo cual se basa en la idea de que estas particiones especiales contendrán los valores atípicos y el resto de la muestra será más uniforme.
- **Maxdiff**: En este caso, el método de particionamiento se basa en la idea de utilizar los puntos de mayor variación de frecuencias mediante la medida $|g(i+1) - g(i)|$, para dividir el conjunto de categorías en sus respectivas particiones, de tal manera que las frecuencias contenidas en cada partición sean lo más homogéneas posibles entre sí.

- **Particionamiento con Garantías de Optimalidad**: En esta categoría se enmarcan las estrategias de generación de particiones que ofrecen garantías de optimalidad a nivel de la precisión de resultados en las consultas. Para ello se utilizan técnicas de *Programación Dinámica* (DP), de tal manera que la selección de las particiones se presenta como un problema de *Optimización*. Sin embargo, dichas estrategias conllevan un elevado coste computacional que muchas veces no es admisible debido al gran tamaño del conjunto de datos que se pretende sumarizar. Como solución ante dicha problemática se han propuesto distintas estrategias que se basan en la resolución del problema de optimización, pero sobre una *muestra* del conjunto de datos global, lo cual anula las garantías de optimalidad pero puede ser una buena aproximación si la muestra seleccionada es altamente representativa respecto de la población.

- **Particionamiento Jerárquico**: Las estrategias de particionamiento jerárquico se basan en la utilización de particiones siguiendo la idea de utilizar un árbol binario. Por lo tanto, las particiones no son disjuntas entre ellas, sino que se contienen unas a otras. Esto sigue la misma idea que se describió en el apartado de *Esquemas de estimación Basados en Árboles*. Apoyandose en esta estrategia de particionamiento se consigue que las consultas de rangos de frecuencias tengan un coste computacional menor en promedio (aún en el casos en que el rango sea muy amplio). En esta categoría destacan los histogramas *nLT* (n-level Tree) y *Lattice Histograms*. Estos últimos tratan de aprovechar las ventajas a nivel de flexibilidad y precisión que presentan los histogramas, además de las estrategias jerárquicas de sumarización en que se apoyan las *Wavelets* tal y como se describe en la siguiente sección.

Las ideas descritas en esta sección sobre los *histogramas* son extrapolables conforme se incrementa la dimensionalidad de los datos. En el caso de los esquemas de estimación, esto sucede de manera directa. Sin embargo, en el caso de los esquemas de particionamiento surgen distintos problemas debido al crecimiento exponencial tanto a nivel de espacio como de tiempo conforme aumenta el número de dimensiones de los datos por lo cual se convierten en soluciones impracticables para conjuntos de datos con elevada dimensionalidad

Los *Histogramas* representan una estrategia sencilla, tanto a nivel de construcción como de consulta, la cual ofrece buenos resultados en un gran número de casos. Dichas estructuras han sido ampliamente probadas para aproximación de consultas relacionadas con suma de rangos o frecuencias puntuales. Tal y como se ha dicho previamente, su comportamiento en el caso unidimensional ha sido ampliamente estudiado, sin

embargo, debido al crecimiento exponencial a nivel de complejidad conforme las dimensiones del conjunto de datos aumentan estas estrategias son descartadas en muchas ocasiones. Los *Histogramas* requieren de un conjunto de parámetros fijados *a-priori*, los cuales afectan en gran medida al grado de precisión de los resultados (pero cuando se seleccionan de manera adecuada esta solución goza de una gran cercanía al punto de optimalidad), por tanto, en casos en que la estimación de dichos valores necesarios *a-priori* se convierte en una labor complicada, existen otras técnicas que ofrecen mejores resultados.

3.2.3. Wavelet

Las estructuras de sumarización denominadas *Wavelets*, a diferencia de las anteriores, han comenzado a utilizarse en el campo del *procesamiento de consultas aproximado* desde hace relativamente poco tiempo, por lo que su uso no está completamente asentado en soluciones comerciales sino que todavía están en fase de descubrimiento e investigación. Las *Wavelets* (u *ondículas*) se apoyan en la idea de representar la tabla de frecuencias del conjunto de datos como una función de ondas discreta. Para ello, se almacenan distintos valores (dependiendo del tipo de *Wavelet*) que permiten reconstruir la tabla de frecuencias tal y como se describirá a continuación cuando se hable de la *transformada de Haar*, la mejora de eficiencia en cuanto a espacio a partir de esta estructura de sumarización se basa en el mantenimiento aproximado de los valores que permiten reconstruir el conjunto de datos.

A continuación se describe la *transformada de Haar*, a partir de la cual se presentan las distintas ideas en que se apoyan este tipo de estructuras de sumarización. En los últimos años se ha trabajado en estrategias más complejas como la *Daubechies Wavelet* [AH01] de *Akansu y otros* o la *transformada de Wavelet basada en árboles duales completos* [SBK05] de *Selesnick y otros*.

3.2.3.1. Haar Wavelet Transform

La *Haar Wavelet Transform* (**HWT**) consiste en una construcción de carácter jerárquico que colapsa las frecuencias de las distintas categorías de manera pareada recursivamente hasta llegar a un único elemento. Por tanto, la idea es la similar a la creación de un árbol binario desde las hojas hasta la raiz. Esta estrategia es similar a la que siguen los *Histogramas jerárquicos* de la sección anterior. Además, se aprovecha de manera equivalente al caso de los histogramas jerárquicos para optimizar el espacio, consistente en almacenar únicamente la variación de uno de los nodos hoja con respecto del padre, lo cual permite reconstruir el árbol completo mediante una simple operación.

Para simplificar el entendimiento de la construcción de la *transformada de Haar* se describe un ejemplo extraido del libro *Synopses for massive data* [CGHJ12] de *Cormode y otros*. Supongamos los valores de frecuencias recogidos en $A = [2, 2, 0, 2, 3, 5, 4, 4]$. Para construir la transformada realizaremos la media de los elementos contiguos dos a dos recursivamente, de tal manera que para los distintos niveles obtenemos los resultados de la tabla 3.1. Además, se muestran los coeficientes de detalle, los cuales se obtienen tras calcular la diferencia entre el primer y segundo elemento contiguos del nivel anterior.

Nivel	Medias	Coeficientes de Detalle
3	$[2, 2, 0, 2, 3, 5, 4, 4]$	–
2	$[2, 1, 4, 4]$	$[0, -1, -1, 0]$
1	$[3/2, 4]$	$[1/2, 0]$
0	$[11/4]$	$[-5/4]$

Tabla 3.1: Ejemplo de construcción de *Haar Wavelet Transform*

Nótese que a partir de la media de nivel 0 que denominaremos $c_0 = 11/4$ así como el conjunto de coeficientes de detalle, que denotaremos por $c_1 = -5/4, c_2 = 1/2, ..., c_7 = 0$ y los coeficientes de detalle es posible reconstruir la tabla de frecuencias A.

Una vez entendida la estrategia de construcción en que se apoya la *transformada de Haar*, se puede apreciar que esta no ofrece ventajas a nivel de coste de almacenamiento respecto del conjunto de frecuencias respecto del cual ha sido construida. Sin embargo, posee la siguiente cualidad, sobre la cual se apoya esta estrategia de sumarización: *Para las categorías contiguas en que la variación de frecuencias es muy reducida, los coeficientes de detalle tienden a aproximarse a* 0.

Por la razón descrita en el parrafo anterior, se intuye que dichos coeficientes de detalle pueden ser obviados, de tal manera que el espacio utilizado para el almacenamiento de la *Wavelet* se convierte en sublineal ($o(N)$), en lugar de lineal ($O(N)$) respecto del espacio del conjunto de datos. Para elegir qué coeficientes de detalle se pueden utiliza estrategias que tratan de minimizar el error. Comúnmente, las *Wavelets* han sido construidas a partir del *error quadrático medio* o *norma-L_2*, la cual se describe en la ecuación (3.7). Sin embargo, distintos estudios como el realizado en el artículo *Probabilistic wavelet synopses* [GG04] de *Garofalakis y otros* muestran como esta medida del error obtiene malos resultados cuando se aplica a la sumarización de datos mediante *Wavelets*.

Por tanto, se proponen otras medidas de error como la minimización del máximo error absoluto o relativo, que se ilustran en las ecuaciones (3.8) y (3.9). También se propone como alternativa la minimización de la *norma-L_p* que se muestra en la ecuación (3.10). Dicha medida de error es una generalización del *error cuadrático medio* (caso $p = 2$) a cualquier valor de $p \geq 0$. Por último se muestra en la ecuación (3.11) el caso del cálculo del error mediante la *norma-L_p* con pesos o ponderada, lo cual permite fijar el grado de importancia para cada categoría en la representación mediante *Wavelets* permitiendo aumentar la precisión de las categorías más significativas.

$$||A - \widetilde{A}||_2 = \sqrt{\sum_i (A[i] - \widetilde{A}[i])^2} \tag{3.7}$$

$$max_i\{absErr_i\} = max_i\{|A[i] - \widetilde{A}[i]|\} \tag{3.8}$$

$$max_i\{relErr_i\} = max_i\left\{\frac{|A[i] - \widetilde{A}[i]|}{|A[i]|}\right\} \tag{3.9}$$

$$||A - \widetilde{A}||_p = (\sum_i (A[i] - \widetilde{A}[i])^p)^{\frac{1}{p}} \tag{3.10}$$

$$||A - \widetilde{A}||_{p,w} = (\sum_i w_i \cdot (A[i] - \widetilde{A}[i])^p)^{\frac{1}{p}} \tag{3.11}$$

Al igual que en el caso de los *Histogramas*, las *Wavelets* presentan problemas de eficiencia cuando se usan en entornos en los cuales el conjunto de datos está compuesto por una gran número de atributos. Por tanto, se dice que sufren la *Maldición de la dimensionalidad* (*Curse of Dimensionality*), que provoca un crecimiento de orden exponencial en el coste tanto en espacio como en tiempo.

Tal y como se puede apreciar, esta estrategia es muy similar a la basada en *Histogramas*, dado que ambas se apoyan en el almacenamiento de valores que tratan de describir o resumir la tabla de frecuencias de los datos de manera similar. Sin embargo, mientras que en el caso de los *Histogramas* estos destacan cuando se pretende conocer la estructura general de los datos, las *Wavelets* ofrecen muy buenos resultados cuando se pretenden conocer valores atípico o extremos (a los cuales se denomina *Heavy Hitters*).

Por su estrategia de construcción, las *Wavelets* permiten sumarizar una mayor cantidad de información utilizando un espacio reducido. Además, en el caso de la *transformada de Haar*, que posee la característica de linealidad, se puede adaptar de manera sencilla al *modelo en Streaming*. Tal y como se ha dicho en el párrafo anterior, las desventajas de esta alternativa derivan en gran medida de los problemas relacionados con el incremento de la dimensionalidad de los datos.

3.2.4. Sketch

Las estructuras de sumarización conocidas como *Sketches* son las que menos tiempo de vida tienen de entre las descritas, por lo tanto, aún están en una fase muy temprana por lo que su uso en sistemas reales todavía es anecdótico. Sin embargo poseen disintas características por las que se piensa que en el futuro las convertirán en estrategias muy importantes el el ámbito del *procesamiento aproximado de consultas*. Los *Sketches* se amoldan perfectamente al modelo en streaming del cual se habla en el capítulo anterior en la sección 2.2. Este modelo se amolde perfectamente a muchos sucesos cambiantes que se dan en la actualidad y que requieren de la otención de analíticas. Un ejemplo de ello es un sistema de transacciones financieras, que suceden con una frecuencia elevada y para las cuales sería apropiado obtener métricas en tiempo real para la mejora de la toma de decisiones. También encajan de manera apropiada en el modelo de transacciones de una base de datos, la cual es modificada constantemente mediante insercciones, modificaciones y eliminaciones.

Los *Sketches* se corresponden con estructuras de datos que funcionan manteniendo estimadores sobre cada una de las instancias que han sido procesadas hasta el momento, es decir, realizan una modificación interna por cada entrada. Esto se opone a las estrategias descritas en anteriores secciones, que procesan

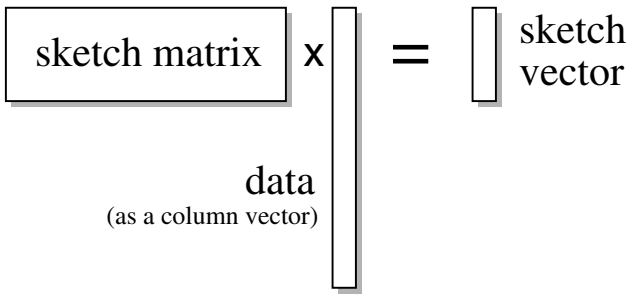


Figura 3.1: Abstracción del *Sketch Lineal. (Extraida de [CGHJ12])*

una parte del conjunto de datos o requieren que el mismo tenga un carácter estático. Estas modificaciones pueden enmarcarse bajo distintas especificaciones del modelo en streaming tal (serie temporal, modelo de caja registradora o modelo de molinete) teniendo que realizar pequeñas adaptaciones en algunos algunos *Sketches*, pero siendo muy poco escalable la implementación del modelo de molinete (que permite eliminaciones) en otros.

Los *Sketches Lineales* son un subconjunto de *Sketches* que pueden ser vistos como una transformación lineal de la estructura de sumarización, la cual puede ser interpretada como un vector de longitud $1 * n$ al cual denominaremos S. Para generar dicho vector es nos apoyamos en la matriz de tamaño $n * m$ a la cual denominaremos A y que representa la transformación lineal del conjunto de datos, el cual puede ser codificado como un vector al cual denominaremos D de tamaño $1xm$. Dicha transformación lineal se representa en la ecuación (3.12) y de manera gráfica en la figura 3.1.

$$A * D = S \tag{3.12}$$

Puesto que la transformación que se muestra es de carácter lineal, entonces posee la propiedad de que siendo S_1 y S_2 dos *Sketches* lineales, entonces se cumple la propiedad de que $S_1 + S_2 = S_{sum}$, lo que puede entenderse como la combinación de los estimadores de los dos *Sketches*. Por tanto, se puede decir que el cada actualización se procesa como una transformación de la nueva instancia codificada en el vector D para después combinar el *Sketch* resultante con el generado anteriormente.

Por motivos de eficiencia a nivel de espacio, la matriz de transformación A no se utiliza de manera explicita, sino que por contra, en muchos casos se utilizan distintas *funciones hash* (las cuales se describen en la sección 2.4.5), que realizan la transformación de la misma forma que la matriz A, solo que el coste computacional para utilizarlas es significativamente menor que el coste necesario para el almacenamiento y multiplicación de una matriz de tal tamaño.

Tal y como se puede apreciar en la figura 3.1 la ventaja de sumarización de esta estrategia se obtiene en la reducción de tamaño obtenida en la transformación lineal. Posteriormente, cada estructura de *Sketch* tiene asociadas un conjunto de *consultas o queries* para las cuales extraer la información valiosa contenida en ellas. Puesto que la forma en que se llevan a cabo dichas consultas varía según la alternativa a la que nos estamos refiriendo, estas técnicas se expondrán en profundidad en sus correspondientes secciones.

Siguiendo con las restricciones descritas anteriormente, existe una implementación trivial de las estructuras de *Sketch* consistente en mantener una tabla de frecuencias sobre cada posible instancia de entrada de tal manera que a partir de esta pueden resolver consultas como sumas de frecuencias, conteo de elementos distintos, búsqueda del máximo o mínimo, media o varianza del conjunto de datos. Sin embargo, esta solución no proporciona ninguna ventaja a nivel de espacio, por tanto, la tarea es encontrar técnicas que permitan "colapsar" dicha tabla de frecuencias para tratar de minimizar dicho coste asumiendo un cierto grado de imprecisión.

Existen dos categorías principales según la naturaleza de las consultas que el *Sketch* puede responder. Dichas categorías se describen a continuación:

- **Estimación de Frecuencias**: Los *Sketches* basados en estimación de frecuencias se corresponden con aquellos que tratan de recoger estimadores que se aproximen a la frecuencia de cada elementos $f(i)$. En esta categoría se enmarcan el *Count-Min Sketch* y *Count Sketch*.

- **Elementos Distintos**: Los *Sketches* basados en el conteo de elementos distintos (y que también pueden responder consultas referidas a la presencia de un determinado elemento) se basan en el mantenimiento de distintos indicadores que permiten aproximar dichos valores. Entre los más destacados en esta categoría se encuentran el *Bloom Filter* y el *HyperLogLog*.

A pesar de que todos los *Sketches* siguen una estructura básica similar, existen distintos parámetros diferenciadores que caracterizan unas alternativas frente a otras entre las que destacan las consultas soportadas, el tamaño de los mismos (algunos necesitan menor espacio para obtener un grado de precisión similar a otros), el tiempo de procesamiento de cada nueva actualización (el cual se pretende que sea muy reducido debido al modelo en streaming), el tiempo de resolución de consultas o la estrategia de inicialización del mismo.

La extensión de los *Sketches* sobre conjuntos de datos de caracter multidimensional puede resolverse utilizando funciones hash que mapeen una entrada multidimensional sobre un espacio unidimensional para después utilizarlos de la manera que para los casos de una dimensión. Esta solución no presenta problemas en la estimación de frecuencias puntuales o búsqueda de máximos o mínimos, sin embargo, no es factible para consultas de sumas de rangos utilizando *sketches* multidimensionales o asumiendo la independencia entre dimensiones y manteniendo una estructura de *sketch* por cada dimensión.

Los *Sketches* presentan un gran nivel de simplicidad en sus implementaciones, lo que les hace muy eficientes y válidos para modelos que requieren de actualizaciones constantes como el descrito en el modelo en streaming. La dificultad de los mismos se basa en los conceptos matemáticos subyacentes, que complica en gran medida la demostración de sus características de convergencia hacia el valor real que estiman. Generalmente para reducir su espacio se apoyan en la utilización de funciones hash con distintas propiedades, pero que también se pueden calcular eficientemente. La mayor limitación se refiere al rango de consultas que cada tipo de *Sketch* puede resolver de manera eficiente, estando especialmente limitados en el ámbito de múltiples consultas anidadas.

Los *Sketches* son un ámbito interesante dentro del mundo de la investigación, con una visión de futuro muy prometedora que les coloca como la mejor solución a medio-largo plazo para la tarea del *procesamiento de consultas aproximadas* por su reducido coste computacional, tanto a nivel de espacio como de tiempo de procesamiento. Actualmente universidades prestigiosas como la *Universidad de California, Berkeley* o el *MIT* están trabajando en el diseño de bases de datos que se apoyan fuertemente en el uso de estas estructuras. Dicha universidad está trabajando en una base de datos a la cual denominan *BlinkDB* y la cual describen en el artículo *BlinkDB: queries with bounded errors and bounded response times on very large data* [AMP+13]

En las siguientes secciones se describen en profundidad distintas estructuras de datos basadas en *Sketches* tratando de prestar especial importancia en una visión conceptual de la misma pero sin dejar de lado la parte práctica de las mismas. Además, se trata de incluir demostraciones acerca de la precisión de las mismas respecto del espacio utilizado para su almacenamiento respecto del tamaño del conjunto de datos global.

3.3. Bloom Filter

El *Bloom-Filter* se corresponde con la primera estructura de datos basada en la idea de *Sketch* por procesar cada nueva entrada de la misma forma y tener un coste a nivel de espacio sublineal ($o(N)$) respecto del cardinal de posibles valores que pueden ser procesados. Esta estructura de datos fue diseñada incialmente por *Burton Howard Bloom* la cual describe en el artículo *Space/time trade-offs in hash coding with allowable errors* [Blo70] publicado en 1970.

La funcionalidad que suministra dicha estructura de datos es la consulta acerca de presencia de un determinado elemento. Para la implementación descrita en el artículo inicial no se permiten incrementos ni eliminaciones, tan solo la existencia o inexistencia del elemento, por tanto esta estructura de datos se enmar en el marco de *serie temporal* del *modelo en streaming*. El *Bloom-Filter* asegura la inexistencia de falsos negativos (elementos que se presupone que no han sido introducidos cuando en realidad si que lo fueron) pero por contra, se admite una determinada tasa de falsos positivos (elementos que se presupone que han sido introducidos cuando en realidad no lo fueron).

Debido al tiempo de vida de dicha estrategia, su uso está muy asentado y se utiliza en distintos sistemas para tareas en las cuales se pretende optimizar el tiempo, pero sin embargo son admisibles fallos. Es comúnmente utilizado en bases de datos comerciales para limitar el número de accesos al disco durante búsquedas de elementos. Esto se lleva a cabo manteniendo un *Bloom-Filter* al cual se consulta sobre la presencia del elemento a buscar y en el caso de que la respuesta sea negativa, entonces se prescinde de dicho acceso a la unidad de almacenamiento. Mientras que si es positiva se realiza el acceso a disco. Nótese por tanto que en algunos ocasiones dicho acceso será innecesario, sin embargo las consecuencias de dicha tarea tan solo tienen consecuencias negativas a nivel de eficiencia y no de resultados.

Una vez descrita la funcionalidad que proporciona el *Bloom-Filter* así como un caso práctico de uso del mismo ya se tiene una idea a grandes rasgo acerca del mismo. Por tanto, lo siguiente es describir su composición. La estructura de datos se basa en el mantenimiento de un *mapa de bits* S de longitud n ($|S| = n$) de tal manera que se cumpla que $n \ll m$ siendo m el cardinal de elementos distintos que se pueden presentar en la entrada. Además, se utilizan k funciones hash denominadas $h_1, h_2, ..., h_j, ..., h_k$ que

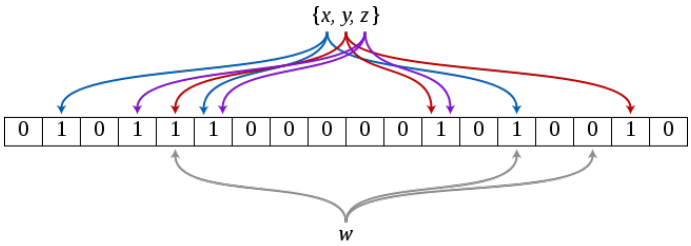


Figura 3.2: Modo de funcionamiento del *Bloom Filter*, que inserta los valores $\{x, y, z\}$ y comprueba la existencia de w. La imagen ha sido extraida de [Wik17b].

distribuyen de manera independiente el elemento i-ésimo en S. El mapa de bits S se inicializa con todas sus posiciones tomando el valor 0.

El modo de funcionamiento del *Bloom-Filter* se lleva a cabo de la siguiente manera: Cada vez que se presenta un nuevo elemento en la entrada (al cual denominaremos $i \in [1, m]$) se tomaran los k valores de las funciones funciones hash indicadas anteriormente y se asignará el valor 1 a dichas posiciones del mapa de bits S. Es decir, se realiza la operación descrita en la ecuación (3.13). La consulta acerca de la presencia del elemento i-ésimo se realiza por tanto, consultando dichas posiciones, de tal manera que si $\forall j \in [1, k],\ S[h_j(i)]$ toma el valor valor 1 se dice que el elemento ha aparecido en la entrada mientras que si alguna posición es distina de 1 (toma el valor 0) se dice que el elemento no ha sido introducido.

$$S[h_j(i)] = 1 \qquad \forall j \in [1, k] \tag{3.13}$$

Nótese que la restricción acerca de la presencia de un determinado elemento es mucho más débil que la de no existencia. La razón de ello se debe a que pueden existir colisiones entre las distintas funciones hash que conviertan dicho resultado en érroneo. Sin embargo, tal y como se ha indicado anteriormente, en caso de que el *Bloom-Filter* indique la no presencia del elemento dicha respuesta será válida. A continuación se realiza un análisis acerca del índice de error para el caso de los falsos positivos la cual se basa en el artículo *Network applications of bloom filters: A survey* [BM04] de *Broder* y *Mitzenmacher*. Nótese que dicho análisis depende de 3 parámtros: el tamaño n del mapa de bits, el cardinal m del conjunto de posibles elementos en la entrada y el número k de funciones hash utilizadas.

Para el análisis se presupone que las funciones hash h_j son totalmente aleatorias, es decir, distribuyen los valores de manera totalmente uniforme en el intervalo $[1, n]$ y son totalmente independientes entre sí. La probabilidad p' de que cualquier posición de S siga siendo igual a cero después de la aparición de l elementos distintos se muestra en la ecuación (3.14) siendo la base del lado derecho de la igualdad la probabilidad de que cada función hash mantenga con el valor 0 una posición.

$$p' = \left(1 - \frac{1}{m}\right)^{kl} \tag{3.14}$$

La probabilidad de que un elemento no introducido en la entrada tome todas sus correspondientes posiciones de S con el valor 1 se da con probabilidad $(1-p)^k$ y dado que p' es un buen estimador para p, entonces

podemos aproximar la tasa de falsos positivos. Tras desarrollar dicha expresión se puede llegar a la ecuación (3.15), que aproxima de manera apropiada la tasa de falsos positivos.

$$f = (1 - e^{-kn/m})^k \tag{3.15}$$

El *Bloom-Filter* es una buena aproximación para los casos en que es necesario reducir el sobre coste de comprobación de accesos a medios costosos, sin embargo, su utilización puede utilizarse en filtros anti-spam u otros entornos en que una determinada tasa de error sea admisible. Nótese que este típo de *Sketches* no puede agruparse en el subconjunto de *Sketches Lineales* puesto que la operación de asignar el valor 1 no es lineal.

3.4. Count-Min Sketch

El *Count-Min Sketch* es otra estructura de datos con la característica de presentar un coste espacial de carácter sublineal ($o(N)$) respecto del cardinal de posibles elementos de entrada. El *Count-Min Sketch* (en adelante *CM Sketch* por abreviación) fue descrito por primera vez por *Cormode* y *Muthukrishnan* en el artículo *An improved data stream summary: the count-min sketch and its applications* [CM05] publicado en 2005. Nótese por tanto que el tiempo de vida de dicha estructura de datos es muy corto.

El propósito del *CM Sketch* es la estimación de frecuencias para el rango de posibles valores que se presentan en la entrada. En la formulación inicial se enmarcaba modelo de caja registradora, que tan solo permite adicciones, sin embargo posteriormente se han propuesto mejoras para ampliar su uso en entornos en que también se permitan eliminaciones (modelo en molinete). Tal y como se verá a continuación, el nombre de este *Sketch* se refiere a las operaciones de operaciones que utiliza durante su funcionamiento, es decir, el conteo o suma y la búsqueda del mínimo.

La estimación de frecuencias del *CM Sketch*, tal y como se puede apreciar debido al carácter sublineal del mismo, no garantiza la exactitud en de los mismos, sino que al igual que en el caso del *Bloom-Filter* devuelve como resultado una aproximación. En este caso dicha aproximación garantiza que el resultado estimado siempre será mayor o igual que el verdadero, es decir, realiza una sobre-estimación del valor real. La razón por la cual el *CM Sketch* utiliza la operación de búsqueda del mínimo es para tratar de reducir dicho efecto en la medida de lo posible.

Debido al corto periodo de vida su uso todavía no está asentado en sistemas reales, sin embargo existen una gran variedad de situaciones que se enmarcan sobre el modelo en streaming en las cuales la precisión de frecuencias no tienen grandes efectos perjudiciales. Un ejemplo de ello podría ser el conteo de visitas a un determinado sitio web formado por distintas páginas. La estimación sobre el número de visitas para cada una de estas páginas podría llevarse a cabo utilizando esta estructura de datos, ya que es una tarea que puede admitir una determinada tasa de error y requiere de actualizaciones constantes.

Una vez descrita la funcionalidad que suministra el *CM Sketch* se describirá la estructura interna del mismo: Esta estructura de datos está formada por una matriz S de estructura bidimensional con w columnas y d filas, de tal manera que se mantienen $w * d$ contadores. Cada una de las d filas tiene asociada una función hash h_j que distribuye elementos pertenecientes al dominio $[1, m]$ (siendo m el cardinal de elementos

distintos) sobre la fila de dicha matriz, es decir, sobre $[1, w]$. Para que las estimaciones acerca de la precisión del *CM Sketch* sean correctas cada función hash h_j debe cumplir la propiedad de ser independiente del resto además de seguir una distribución uniforme. En cuanto a la inicialización de la estructura de datos, todas las posiciones toman el valor cero, esto se muestra en la ecuación (3.16).

$$S[k, j] = 0 \qquad \forall k \in [1, w],\ \forall j \in [1, d] \tag{3.16}$$

En cuanto al modo de funcionamiento del *CM Sketch*, tal y como se ha indicado anteriormente, se enmarca en el modelo de caja registradora, es decir, cada actualización se corresponde con una tupla formada por el identificador del elemento al que se refiere y un determinado valor positivo que indica el incremento c al cual se refiere la actualización. El funcionamiento es similar al del *Bloom-Filter* en el sentido de que por cada nueva entrada se realizan d actualizaciones (una para cada fila). La operación de conteo se refiere a dicha actualización, que se ilustra de manera matemática en la ecuación (3.17). Una representación gráfica de dicha operación se muestra en la figura 3.3. Nótese por tanto que el coste de actualización es de $O(d)$. Nótese que esta operación es de carácter lineal por lo que el *CM Sketch* se agrupa en el subconjunto de *Sketches Lineales*.

$$S[j, h_j(i)] = S[j, h_j(i)] + c \qquad \forall j \in [1, d] \tag{3.17}$$

En cuanto al proceso de consulta sobre la frecuencia del elemento i-ésimo, esto se lleva a cabo recogiendo el valor almacenado en cada fila y devolviendo el menor, de tal manera que se pretende reducir el efecto de las colisiones generados sobre las funciones hash al distribuir elementos sobre un espacio de menor tamaño, ya que se presupone que $w * d \ll m$ para que la ventaja a nivel de espacio se produzca. Dicha operación se muestra en la ecuación (3.18)

$$\widetilde{f}(i) = min_{j \in [1,d]}\{S[j, h_j(i)]\} \tag{3.18}$$

A nivel de análisis sobre el coste espacial necesario para almacenar en memoria el *CM Sketch*, este es dependiente del grado de precisión que se pretenda asegurar mediante el uso del mismo. Sin embargo, el análisis no se realiza sobre el cardinal de posibles elementos que pueden aparecer en la entrada, sino que depende del sumatorio del conteo de estos. Este valor se denomina F_1 (o primer momento de frecuencia), que se define tal y como se indica en la ecuación (3.19).

$$F_1 = \sum_{i=1}^{m} f(i) \tag{3.19}$$

Por tanto, la precisión de esta estructura de datos probabilística se indica diciendo que $\widetilde{f}(i)$ tendrá un error máximo de ϵN que el cual se cumplirá con probabilidad $1 - \delta$. Estos parámetros se fijan en el momento de la inicialización del *CM Sketch* fijando el número de filas y columnas de tal manera que $d = log(1/\delta)$ y $w = 2/\epsilon$. La demostración acerca de la elección de estos valores puede encontrarse en el artículo original [CM05].

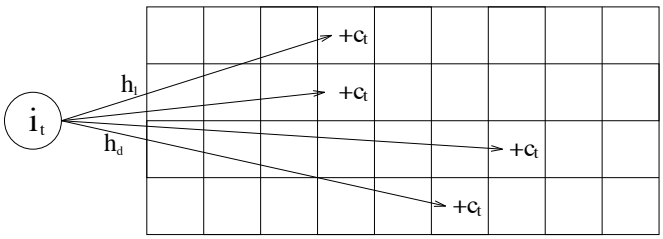


Figura 3.3: Modo de funcionamiento del *Count-Min Sketch* durante el proceso de inserción de un nuevo elemento. La imagen ha sido extraida de [CM05].

En cuanto a la estimación de frecuencias obtenida por el *CM Sketch*, en sentido estricto se dice que esta es sesgada respecto de su valor real. La razón se debe a que siempre se produce una sobre-estimación, a pesar de tratar de reducir los efectos negativos de la misma tratando de seleccionar el mínimo de las estimaciones siempre será mayor o igual (nunca menor). Para tratar de solventar esta problemática se han propuesto distintas heurísticas que tratan de contrarrestar esta problemática en el modelo de caja de registradora (tan solo se permiten inserciones).

Sin embargo, dicho problema no sucede en el caso del modelo general o de molinete, sobre el cual si que están permitidas las eliminaciones. En este caso es más apropiado utilizar como estimación la mediana de los valores almacenados en cada columna, puesto que se escoge el mínimo para un elemento que tan solo ha recibido actualizaciones negativas probablemente este será muy diferente del valor real. La razón por la cual se escoge la mediana y no la media es por sus propiedades de resistencia ante valor atípicos u *outliers*. En este caso la demostración se puede llevar a cabo apoyandose en la desigualdad de Chernoff descrita en la sección 2.4.4. Esta demostración se puede encontrar en el artículo *Selection and sorting with limited storage* [MP80] de *Munro* y *Paterson*.

El *Count-Min Sketch* consiste en una estrategia apropiada para el conteo de ocurrencias en un espacio de orden sublineal. Además, la implementación del mismo es simple en contraposición con otras estrategias más sofisticadas. En posteriores secciones se hablará del *Count Sketch*, que proporcionan valores de precisión equivalentes en espacios de menor tamaño añadiendo una mayor complejidad conceptual.

3.5. Count Sketch

El *Count Sketch* se refiere a una estructura muy similar al *Count-Min Sketch* ya que sigue la misma estructura e ideas muy similares en para la actualización y consulta de valores, sin embargo proporciona ventajas a nivel de coste espacial respecto del anterior reduciendo en un orden de dos el espacio necesario para mantener la estructura de datos. La definición acerca del *Count Sketch* se indicó por primera vez en el trabajo *Finding frequent items in data streams* [CCFC02] desarrollado por *Charikar y otros*. Nótese que este trabajo fue llevado a cabo en el año 2002, es decir, es anterior a la publicación del referido al *Count-Min Sketch* (2005). La razón por la cual se ha seguido este orden y no el temporal se debe a que el *Count Sketch* puede ser visto como una mejora sobre el descrito en la sección anterior, lo que simplifica el entendimiento de los mismos.

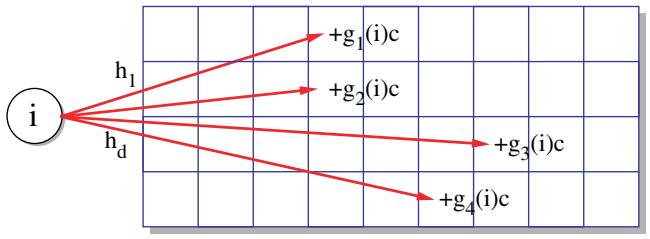


Figura 3.4: Modo de funcionamiento del *Count Sketch* durante el proceso de inserción de un nuevo elemento. La imagen ha sido extraida de [CGHJ12].

El *Count Sketch* se basa en los mismos elementos para su funcionamiento que su homónimo más simple, por lo tanto se utilizará la definición utilizada previamente de la matriz S compuesta por w columnas y d filas además de las funciones hash $h_1, h_2, ..., h_j, ..., h_d$ independientes entre si, y cuya distribución es uniforme. La variación surge a nivel de las operaciones que se realizan en el proceso de actualización y consulta sobre la estructura de datos. El funcionamiento del *Count Sketch* puede ser visto de varias formas, primero describiremos la versión extendida del mismo para después describir una versión que resuce a la mitar el espacio necesario para su mantenimiento en memoria.

$$h'_j(i) = \begin{cases} h_j(i) - 1, & h_j(i) \ mod \ 2 = 0 \\ h_j(i) + 1, & h_j(i) \ mod \ 2 = 1 \end{cases} \tag{3.20}$$

A continuación se describe la versión extendida, que se apoya en la utilización de la función hash $h'_j(i)$, la cual se define en la ecuación (3.20), que consiste en una variación sobre la función hash base $h_j(i)$. Esta función se utiliza únicamente en el momento de la estimación, es decir, no se utiliza para la actualización tras la llegada de elementos. La intuición en la que se basa su utilización es la reducción del sesgo producido por las colisiones de otros elementos en cada fila, que siempre sobre-estima los resultados (en promedio la sobre-estimación de cada fila es de $\frac{\sum_{k=1}^{w} S[j,k]}{w}$). De esta manera se pretende contrarrestar dicho efecto por lo que ya no es apropiada la utilización de la búsqueda del mínimo, sino que se puede llevar a cabo la estimación mediante el uso de la mediana, tal y como se indica en la ecuación (3.21).

$$\widetilde{f}(i) = median_{j \in [1,d]} \{S[j, h_j(i)] - S[j, h'_j(i)]\} \tag{3.21}$$

La versión reducida de esta alternativa se apoya en la utilización de d funciones hash uniformes e independientes entre sí con dominio de entrada $[1, m]$ e imagen en el subconjunto $\{-1, 1\}$. Si se sustituye el método de actualización de elementos por el descrito en la ecuación (3.22) donde i identifíca el i-ésimo elemento y c la variación del mismo correspondiente a dicha actualización, entonces para conseguir la misma precisión que en el caso anterior se requiere de la mitad de espacio. Puesto que esto es equivalente a mantener $S[j, h_j(i)] - S[j, h'_j(i)]$ en una única posición de S, entonces la estimación de la frecuencia se sustituye por la ecuación (3.23).

$$S[j, h_j(i)] = S[j, h_j(i)] + g_j(i)c \tag{3.22}$$

$$\widetilde{f}(i) = median_{j \in [1,d]} \{S[j, h_j(i)]\} \tag{3.23}$$

Para el análisis sobre la precisión del *Count Sketch*, se utiliza un enfoque similar al del *CM Sketch*. Sin embargo, en este caso el valor ϵ se escoge teniendo en cuante el segundo momento de frecuencias, es decir, $F_2 = \sum_{i=1}^{m} f(i)^2$, de tal manera que para estimación obtenida sobre cada fila, esta se encuentra en el intervalo $[f(i) - \sqrt{F_2/w}, f(i) + \sqrt{F_2/w}]$. La estimación global por tanto tiene un error máximo de $\epsilon\sqrt{F_2}$ con probabilidad $1 - \delta$.

Para que se cumpla dicha estimación del error el tamaño de la matriz S debe escogerse de tal forma que el número de columnas sea $w = O(1/\epsilon^2)$ y el número de filas sea $d = O(log(1/\delta))$. La demostración completa acerca de la elección puede encontrarse en el artículo original [CCFC02]. Nótese la diferencia en cuanto a la estimación del error del *Count Sketch* con respecto del *CM Sketch*, se debe a que en el primer caso se utiliza el segundo momento de frecuencias (F_2) mientras que en el *CM Sketch* la estimación de la tasa de error se apoya en el primer momento de frecuencias (F_1).

En esta implementación del *Count Sketch* tan solo se ha impuesto la restricción a nivel de funciones hash de que estas sean de carácter *2-universales*, lo cual es una condición suficiente para estimar frecuencias puntuales. Sin embargo, esta estructura de datos se puede extender para la estimación del segundo momento de frecuencias (F_2) mediante la técnica descrita en la sección 2.7 extraida del artículo de *Alon y otros* [AMS96]. Para la estimación de F_2 basta con calcular la mediana de las sumas de los elementos de cada fila al cuadrado. Esto se indica en la ecuación (3.24). Sin embargo, para que esta solución sea correcta desde el punto de vista de la varianza del estimador es necesario que las funciones hash utilizadas sean *4-universales* (la razón se debe al cálculo de las potencias de grado 2). Cuando se cumple esta propiedad entonces la técnica se denomina **AMS Sketch**.

$$\widetilde{F}_2 = median_{j\in[1,d]}\{\sum_{k=1}^{w} S[j,k]^2\} \tag{3.24}$$

A nivel práctico estas dos alternativas ofrecen resultados muy similares cuando se imponen las mismas restricciones a nivel de espacio. Sin embargo, ambos poseen una carácterística destacada: En muchos casos, la estimación obtenida es mucho más precisa puesto que muchas distribuciones de probabilidad tienen formas asimétricas, es decir, algunos elementos presentan una frecuencia de aparición elevada mientras que otros aparecen en contadas ocasiones, lo cual mejora la precisión de los estimadores obtenidos por estos *sketches*.

3.6. HyperLogLog

En esta sección se habla del *Sketch HyperLogLog*, que en este caso tiene tanto un enfoque como una utilidad distintas a las alternativas descritas en secciones anteriores. El *HyperLogLog* se utiliza para tratar de estimar de manera precisa el número de elementos distintos que han aparecido sobre un stream de elementos. Esta estructura de datos es una evolución de la versión básica de la cual se cual se habló en el capítulo de *Algoritmos para Streaming* en la sección 2.6.

Esta versión fue ampliada posteriormente en *Loglog counting of large cardinalities* [DF03] y para finalmente en 2007 se presentar el *HyperLogLog*. La descripción del *HyperLogLog* se indica en el trabajo *Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm* [FFGM07] llevado a cabo por *Flajolet*. Para las explicaciones indicadas en esta secciones se ha seguido el artículo *HyperLogLog in practice: algorithmic*

engineering of a state of the art cardinality estimation algorithm [HNH13] en el cual se exponen distintas mejoras sobre el *HyperLogLog* que se describirán posteriormente.

Al igual que se indicó en la descripción del *Algorimo de Flajolet-Martin* (Sección 2.6), la tarea que el *HyperLogLog* pretende resolver es el conteo de elementos distintos o cardinalidad de un conjunto de elementos. Para ello, esta estrategia se apoya en la idea de mapear cada elemento que se presente en la entrada sobre una función hash binaria que cumpla una distribución uniforme de probabilidad. Bajo estas condiciones, la intuición en que se basa el *HyperLogLog* es que en el subespacio de destino de la función hash, el 50 % de los elementos se codificarán de tal manera que el bit menos significativo sea un 0, el 25 % se codificarán de tal manera que los dos bits más significativos sean 00 y así sucesivamente, de tal manera que $1/2^k$ tendrán k ceros como bits más significativos.

Por tanto, el *HyperLogLog* se basa en la utilización de la función $\rho(x)$, que indica el número de ceros contiguos más significativos mas uno de en la representación binaria generada por la función hash. Hasta este punto, la estrategia es equivalente al *Algorimo de Flajolet-Martin*, sin embargo, a continuación se indica una alternativa a esta a partir de la cual se obtiene una estimación con varianza mucho más reducida. Para ello en lugar de tratar el stream de elementos S de manera conjunta, se divide dicho stream en m sub-streamns de tal manera que todos tengan longitudes similares a los cuales denotaremos por S_i con $i \in [1, m]$.

$$M[i] = max_{xinS_i}\rho(x) \tag{3.25}$$

Para mantener la cuenta de cada estimador se utiliza un vector M que almacena dicho valor. La estrategia de construcción del mismo se muestra en la ecuación (3.25). En cuanto a la estimación de resultados, se utiliza la estrategia que se describe en la ecuación (3.26), que se apoya en el término α_m descrito en la ecuación (3.27). Nótese que el último término se corresponde con una media armónica de la cada contador. La elección de dichos operadores así como el término α_m se describe en el artículo original [FFGM07] y la razón por la cual se escogen de dicha manera es la de tratar la varianza de la estimación y, por tanto, mejorar su precisión.

$$card(S) = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right) \tag{3.26}$$

$$\alpha_m = \left(m \cdot \int_0^\infty \left(log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1} \tag{3.27}$$

En cuanto a las mejoras propuestas en [HNH13] sobre el *HyperLogLog* original, estas se basan en la utilización de una función hash de *64 bits* (en lugar de 32 como la propuesta original), la utilización de un estimador diferente cuando se cumple que $card(S) < \frac{5}{2}m$ y una estrategia de representación dispersa que trata de aprovechar en mayor medida el espacio en memoria.

3.7. L_p-Samplers

En esta sección se describe el concepto de L_p-*Sampler*, para lo que se han seguido las ideas expuestas en *Tight bounds for Lp samplers, finding duplicates in streams, and related problems* [JST11] de *Jowhari y otros*. L_p-*Samplers* se basan en estructuras de datos que procesan un stream de elementos definidos sobre el

conjunto $N = \{1, 2, ..., i, ...n\}$ sobre el cual están permitidas tanto addiciones como eliminaciones, es decir, se enmarcan dentro del modelo en molinete (*turnstile model*). Denotaremos por $x_i \in x$ al número de ocurrencias del elemento i-ésimo sobre el stream.

Los L_p-*Samplers* devuelven un subconjunto de elementos N' extraido del conjunto global N de tal manera que cada elemento es seleccionado con probabilidad $\frac{|x_i|^p}{||x||_p^p}$ siendo $p \geq 0$. De esta manera, se obtienen una muestra del conjunto global que mantiene la misma proporción de elementos con respecto del global pero en un espacio menor utilizando como medida de estimación para el muestreo la norma p.

La dificultad del problema se basa en el mantenimiento de la muestra teniendo en cuenta que los elementos pueden ser tanto añadidos como eliminados de la misma según varía su cuenta x_i conforme se procesa el stream.

De esta manera, cuando escogemos $p = 1$ se seleccionarán los elementos pertenecientes a la muestra con una probabilidad proporcional al número de ocurrencias de los mismos en el stream. En [JST11] se describe un algoritmo para L_p-*Sampler* con $p \in [0, 2]$, ajustandose a las restricciones de espacio del modelo en streaming. Para ello se apoyan en la utilización del *Count-Sketch* (sección 3.5). A continuación se describe en más detalle el caso de los L_0-*Samplers*.

3.7.1. L_0-Samplers

Los L_0-*Samplers* siguen la misma idea de muestreo descrita en esta sección. En este caso utilizan la norma 0, por lo tanto, seleccionan los elementos pertenecientes a la muestra con probabilidad uniforme de entre los que han aparecido en el stream. Este es por tanto el caso más básico del problema debido a que tan solo es necesario mantener un contador que indique si el elemento está presente. Los L_0-*Samplers* son de gran utilidad en el contexto de grafos, ya que encajan de manera adecuada en dicho modelo, permitiendo mantener un subconjunto de aristas del grafo basandose únicamente en si estas han aparecido o no en el stream.

Esto es sencillo cuando solo están permitidas adicciones, ya que una vez aparecido el elemento este no desaparecerá. Sin embargo, en el caso de las eliminaciones la tarea se vuelve más complicada ya que tal y como se ha indicado en el párrafo anterior, es necesario mantener la cuenta de ocurrencias para saber si el elemento a desaparecido o únicamente ha decrementado su cuenta.

A continuación se describe la estructura básica de estos algoritmos siguiendo el trabajo desarrollado por *Cormode y Fermain* en *On unifying the space of L0-sampling algorithms* [CF13]. Estos indican que todas las propuestas de L_0-*Samplers* se basan en tres fases:

- *Sampling*: Se mantiene el conteo de ocurrencias en una estructura de datos auxiliar con ideas similares a las del *Count-Sketch*.
- *Recovery*: Se realizan peticiones a la estructura de datos auxiliar para recuperar el conteo de ocurrencias de cada elemento en el stream (saber si ha apararecido en el mismo).
- *Selection*: Se realiza una selección de entre los elementos aparecidos en el stream siguiendo una distribución uniforme de probabilidad.

3.8. Conclusiones

En este capítulo se han descrito distintas estrategias y técnicas para tratar de estimar indicadores estadísticos que ayuden a entender y comprender mejor conjuntos de datos de tamaño masivo. Estos estimadores tratan de agilizar la obtención de métricas sobre el conjunto de datos que debido a gran tamaño, muchas veces hace impracticable su obtención de manera determinística por el coste en complejidad tanto espacial como temporal. Para ello, se han descrito distintas soluciones, desde la extracción de muestras, así como histogramas o wavelets, hasta técnicas más novedosas y que encajan de manera más apropiada en el entorno cambiante sobre el que en muchas ocasiones se trabaja.

Por contra, las soluciones basadas en *Sketches* tienen un escaso tiempo de vida, por lo que todavía se encuentran en fase de investigación. A pesar de ello existen soluciones para la comprobación de la aparición de elementos mediante el *Bloom Filter*, la estimación de frecuencias mediante el *Count-Min Sketch* o el *Count Sketch*, y el conteo de elementos distintos a partir del *HyperLogLog*.

Se cree que el conjunto de estructuras de datos probabilísticas basadas en *Sketches* aumentará conforme el paso del tiempo y en el futuro se diseñarán alternativas más sofisticadas que permitan estimar un gran número de parámetros.

Capítulo 4

Algoritmos aplicados a Grafos

4.1. Introducción

En las últimas décadas se han dedicado grandes esfuerzos en el estudio de sucesos basados en interelaciones entre elementos. Esto puede entenderse como el análisis de una red de interconexiones modelada como flechas que relacionan un conjunto de puntos. Una gran cantidad de situaciones de la vida cotidiana puede ser representada de esta manera, mediante la cual, se consigue un formalismo matemático enriquecedor sobre el cual resolver distintos problemas que surgen sobre dichas redes.

Estas redes de interconexiones se pueden apreciar en ámbitos muy dispares, como las ciudades y las carreteras que conectan unas con otras, las personas y las amistades entre si, los teléfonos y las llamadas de unos a otros o las páginas web de internet y los enlaces para navegar de unas a otras. El estudio de estas situaciones es de gran interés en las sociedades modernas, permitiendo una mejora del rendimiento a partir de la extracción de información poco obvia mediante distintas técnicas, lo cual conlleva reducción de costes y un aumento del grado de satisfacción para los usuarios de dichos servicios.

Sin embargo, utilizar un lenguaje de representación más enriquecedor conlleva sobrecostes asociados que en otros modelos más simples no se dan, lo cual requiere de técnicas sofisticadas para tratar de hacer frente a la resolución de los problemas que se pretende resolver. Además, el crecimiento exponencial en la infraestructura tecnológica ha traido como consecuencia un gran aumento a nivel de tamaño en estos. Algunos ejemplos destacados en el ámbito de internet se dan el protocolo IPv6, que cuenta con 2^128 posibles direcciones hacia las que poder comunicarse, o el caso de la red social *Facebook*, que cuenta con 1 trillón de relaciones de amistad, tal y como se indica en *One trillion edges: Graph processing at facebook-scale* [CEK+15].

Conforme el tamaño de las redes aumenta, una estrategia razonable es la de utilizar soluciones aproximadas para la resolución de los problemas que se plantean sobre ellas, a través de los cuales se pretende encontrar una solución cercana a la exacta (admitiendo una desviación máxima de ϵ, que se cumple con probabilidad δ), lo que otorga como ventaja una reducción significativa tanto desde la perspectiva del coste temporal como del espacial en la resolución del problema.

En este capítulo se pretende realizar una descripción acerca de las distintas alternativas posibles para tratar de agilizar la resolución de problemas sobre redes de tamaño masivo utilizando técnicas aproximadas.

Por tanto, primero se ha realizado una descripción formal sobre el modelo de representación de grafos en la sección 4.2. A continuación, se ha descrito un modelo sobre el cual diseñar algoritmos que resuelvan problemas aplicados a grafos tratando de aprovechar al máximo el espacio en memoria (que se considera limitada) y reduciendo el número de accesos sobre el espacio de almacenamiento mediante el *modelo en semi-streaming*, del cual se habla en la sección 4.3. El siguiente tema que se trata en este capítulo se refiere a técnicas que tratan de reducir la complejidad de una red de tamaño masivo mediante la eliminación de relaciones entre sus punto mediante la utilización de *Spanners* y *Sparsifiers* en la sección 4.4. Después, se ha realizado una breve descripción acerca de distintos problemas bien conocidos para los cuales se han encontrados soluciones sobre el *modelo en semi-streaming* en la sección 4.5. Finalmente se ha realiza una breve conclusión en la sección 4.6.

4.2. Definición Formal

En esta sección se describen los conceptos básicos necesarios para entender el estudio de problemas modelados como *Grafos*. Para la descripción formal sobre dichas estructuras se han utilizado las notas de clase de la asignatura de *Matemática Discreta* [LMPMSB14] impartida en la *Universidad de Valladolid* así como las de la asignatura equivalente (*Discrete Mathematics CS 202* [Asp13]) impartida por *Aspnes* en la *Universidad de Yale*.

La **Teoría de Grafos** (*Graph Theory*) es la disciplina encargada del estudio de estructuras compuestas por vértices y aristas desde una persepectiva matemática. Los vértices representan objetos o elementos, mientras que las aristas se corresponden con las relaciones que se dan entre vértices. Un grafo G se define por tanto como la tupla del conjunto de vértices $V = \{v_{1,2}, ..., v_n\}$ y el conjunto de aristas $E = \{e_1, e_2, ..., e_m\}$, de tal manera que $e_j = (v_{i_1}, v_{i_2})$ representa el arista que conecta el vértice v_{i_1} con el vértice v_{i_2}. Nótese por tanto, que el grafo está compuesto por n vértices y m aristas. El grafo G se puede modelizar por tanto como $G = (V, E)$. En la figura 4.1 se muestra una representación gráfica de un determinado grafo no dirigido compuesto por 6 vértices y 7 aristas.

Aquellos grafos para los cuales la arista e_j representa los dos sentidos de la relación, es decir, v_{i_1} está relacionado con v_{i_2} y v_{i_2} está relacionado con v_{i_1} se denominan *Grafos no nirigidos*, mientras que en los casos en que esta relación no es recíproca se habla de *Grafos dirigidos*. Cuando cada arista e_j tiene asociado un determinado peso $w_j \in W = \{w_1, w_2, ..., w_m\}$ se dice entonces que G es un *grafo ponderado* y se denota

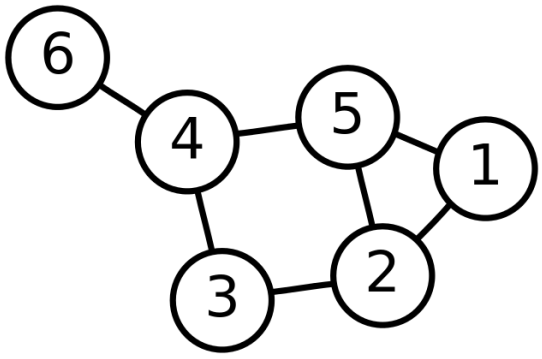


Figura 4.1: Ejemplo de *Grafo No Dirigido*. (Extraido de [Wik17e])

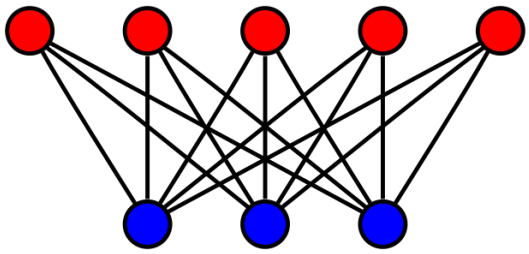


Figura 4.2: Ejemplo de *Grafo Bipartito*. (Extraido de [Wik17e])

como $G = (V, E, W)$, mientras que cuando se presupone que todas las aristas tienen el mismo peso W se omite de la notación.

Cuando un vértice denominado $v_{i_1} \in V$ está directamente relacionado con otro $v_{i_2} \in V$, es decir, existe una arista $e_j \in E$ que los conecta ($e_j = (v_{i_1}, v_{i_2})$) se dice que son e_j es *incidente* sobre dichos vértices. De la misma manera se dice que v_{i_1} y v_{i_2} son *adjacentes* entre sí.

Respecto del conjunto de aristas incidentes sobre cada vértice, se denomina *grado* al cardinal dicho conjunto, diferenciando en los grafos no dirigidos entre *in-grado* a las de entrada y *out-grado* a las de salida. Se utiliza la notación $d(v_i)$ para referirse al grado del vértice i-ésimo, $d^+(v_i)$ al *in-grado* y $d^-(v_i)$ al *out-grado* de dicho vértice. Nótese por tanto, que se cumple la siguiente propiedad: $d(v_i) = d^+(v_i) + d^-(v_i)$.

Un *camino* es un conjunto de aristas $P_p = \{e_{k_1}, e_{k_2}, ..., e_{k_p}\}$, tales que el arista k-ésimo tiene como vértice de destino el mísmo que utiliza el arista $k + 1$ como vércice origen. Nótese que el valor p indica la *longitud* del camino. Cuando el vértice de destino de la arista e_{k_p} es el mismo que el de origen de e_{k_1} se denomina *ciclo* y se denomina C_p.

Se denota como K_n al grafo compuesto por n vértices y $n*(n-1)$ aristas, de tal manera que estas conectan cada vértice con todos los demás. Los grafos que cumplen esta propiedad se denominan grafos completos de grado n. Nótese que el cardinal de aristas se reduce a la mitad en el caso de los grafos no dirigidos.

Cuando al estudiar la estructura de un grafo, se comprueba que el conjunto de vértices puede dividirse en dos subconjuntos disjuntos V_1 y V_2, de tal manera que para todas las aristas $e_j = (v_{i_1}, v_{i_2})$ el vértice v_{i_1} se encuentra en el subconjunto V_1 y el vértice v_{i_2} se encuentra en el subconjunto V_2, entonces se habla de un *Grafo Bipartito*. Un ejemplo de esta situación se muestra en la figura 4.2. Nótese que el concepto de grafo bipartito puede extenderse fácilmente a más de dos subconjuntos, denominandose entonces *Grafo k-partito*. Estos grafos son de gran utilidad para modelizar algunos problemas tales como los que se dan en empresas como *Netflix*, para los cuales V_1 puede estar formado por el conjunto de usuarios mientras que V_2 representa el contenido multimendia. Por tanto, cada arista puede entenderse como una visualización del usuario $v_{i_1} \in V_1$ sobre el contenido $v_{i_2} \in V_2$.

Un *subgrafo* H es el grafo compuesto por un subconjunto de vectores y aristas del grafo G. Nótese que en el caso de que se eliminen vértices, es necesario eliminar también todas sus aristas incidentes. Esto se puede indicar de manera matemática de la siguiente forma: $H \subseteq G$ por lo que $H_V \subseteq G_V$ y $H_E \subseteq G_E$.

Desde el punto de vista de las transformaciones que se pueden realizar sobre grafos, se denomina *isomorfismo* a una transformación biyectiva sobre el grafo $G = (V_G, E_G)$ al grafo $H = (V_H, E_H)$ que se realiza sobre los vértices de manera que $f : V_G \to V_H$ y por cada arista $(e_1, e_2) \in E_G$, entonces $(f(e_1), f(e_2)) \in E_H$ de tal manera que la estructura de G se conserva en H. Entonces se dice que G y H son isomorfos entre si. Si se modifica la condición de biyectividad del *isomorfismo* y tan solo se requiere la propiedad de inyectividad entonces se habla de *homomorfismo*. Por tanto, esto puede ser visto como una transformación sobre la cual no se mantiene la estructura de G, entonces H ya no es equivalente a G, sino que es un *subgrafo* de este.

Las transformaciones son interesantes desde el punto de vista del tratamiento de grafos de tamaño masivo, dado que a partir de estas se trata de reducir la complejidad cuando el tamaño de estos los hace intratables. Por lo tanto, en este caso interesa conseguir transformaciones que reduzcan el tamaño del grafo, pero que tratan de mantener su estructura lo más semejante posible. Destacan transformaciones conocidas como *Spanners* (de los que se hablará en la sección 4.4.1) y *Sparsifiers* (en la sección 4.4.2).

4.2.1. Métodos de Representación

Existen diversas estrategias para representar un grafo sobre estructuras de datos, las cuales presentan distintas ventajas e inconvenientes, tanto a nivel de espacio como de tiempo de acceso. Dichas estrategias se escogen teniendo en cuenta la estructura del grafo. En esta sección se habla de matrices de adyacencia, de listas de adyacencia y de la matriz laplaciana, la cual contiene un conjunto de propiedades interesantes que pueden ser muy útiles en la resolución de algunos problemas.

4.2.1.1. Matriz de Adyacencia

Se denomina matriz de adyacencia A del grafo $G = (V, E)$ compuesto por $n = |V|$ vértices, $m = |E|$ aristas y W el conjunto de pesos de los aristas a una matriz de tamaño $n * n$ construida tal y como se indica en la ecuación (4.1). Nótese que esta definición es válida tanto para grafos ponderados como para no ponderados suponiendo que $w_k = 1, k \in [1, m]$.

$$A_{i,j} = \begin{cases} w_k, & (v_i, v_j) = e_k \in E \\ 0, & \text{en otro caso} \end{cases} \tag{4.1}$$

Esta estrategia de representación es apropiada cuando se trabaja sobre grafos altamente conexos (con un gran número de aristas), ya que el número de posiciones nulas en la matriz es reducido y la estructura matricial indexada proporciona un tiempo de acceso de $O(1)$ a cada posición. Sin embargo, se requiere de $O(n^2)$ para almacenar dicha matriz, algo admisible cuando $n^2 \approx m$ para lo que esta representación se acerca a su óptimo espacial.

4.2.1.2. Lista de Adyacencia

La alternativa a la codificación del grafo como una matriz de adyacencia se conoce como *lista de adyacencia*. En este caso, se mantiene una lista (posiblemente ordenada u otras estrategias estructuradas como listas enlazadas o árboles, para tratar de reducir los tiempos de acceso) para almacenar el conjunto de aristas E. Nótese por tanto, que en este caso la codificación es óptima a nivel de espacio $O(m)$, no existiendo una estrategia de representación que pueda almacenar la estructura del mismo de manera exacta utilizando un

tamaño menor. Sin embargo, tal y como se puede intuir, el tiempo de acceso es de $O(m)$ en el peor caso. Esta solución es apropiada para grafos muy dispersos (aquellos en que $n \ll m$).

4.2.1.3. Matriz Laplaciana

La *matriz laplaciana* consiste en una estrategia de representación de grafos que cumple importantes propiedades, a partir de las cuales se facilita en gran medida la resolución de distintos problemas, entre los que se encuentran *árboles de recubrimiento* (sección 4.5.3), aunque también se utiliza en la estimación de la distribución de probabilidad de *paseos aleatorios* (sección 4.5.5).

El método de construcción de la *matriz laplaciana* se indica en la ecuación (4.2), el cual genera una matriz L de tamaño $n * n$ (al igual que en el caso de la matriz de adjacencia) que aporta información sobre el grafo subyacente. La diagonal de la matriz laplaciana contiene el grado del vértice referido a dicha posición, mientras que el resto de las celdas se construyen colocando el valor $-w_k$ cuando existe un arista entre el vértice v_i y el vértice v_j ($e_k = (v_i, v_j) \in E$) y 0 cuando no existe. La matriz laplaciana también puede entenderse como $L = D - A$ donde D representa una matriz diagonal de tamaño $n * n$ que almacena el grado del vértice v_i en la posición $D_{i,i}$ y 0 en otro caso, mientras que A se corresponde con la matriz de adyacencia descrita anteriormente.

$$L_{i,j} = \begin{cases} d(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

A partir de esta representación se facilita la resolución de distintos problemas tal y como se ha indicado anteriormente. También existen distintas variaciones respecto de la descrita en esta sección, entre las que se encuentran la *matriz laplaciana normalizada* o la *matriz laplaciana de caminos aleatorios*, de la cual se hablará en el capítulo 5 para el cálculo del ranking de importancia *PageRank*.

En la práctica, cuando se trabaja con grafos de tamaño masivo, es muy común que estos sean muy dispersos. Algunos ejemplos de ello son el *grafo de la web* (grafo dirigido), en el cual cada vértice representa sitio web y cada arista un enlace hacia otro sitio web. Es fácil comprobar que este grafo es altamente disperso ya que es muy poco probable que un sitio web contenga enlaces al resto de sitios de la red cuando se habla de millones de vértices. Algo similar ocurre en el caso de redes sociales como *Facebook* (grafo no dirigido debido a relaciones de amistad) o *Twitter* (grafo dirigido debido a relaciones seguimiento). Por tanto, la representación mediante listas de adyacencia es una herramienta útil a nivel conceptual pero que en la práctica no se utiliza por la inviabilidad derivada del gran coste espacial para su almacenamiento.

4.3. Modelo en Semi-Streaming

Al igual que sucede en el caso de conjuntos de datos de carácter numérico y categoríco, en el modelo de grafos, también es necesario hacer frente al elevado tamaño del problema mediante nuevas estrategias de diseño de algoritmos. El *modelo en streaming*, del cual se habló en la sección 2.2 es una buena alternativa para tratar de agilizar la búsqueda de soluciones que varían con respecto del tiempo, además de trabajar

sobre un espacio reducido lo cual aprovecha en mayor medida las capacidades del hardware subyacente. Tal y como se indicó anteriormente, esta estrategia permite reducir el número de acceso sobre el dispositivo de almacenamiento tratando de trabajar únicamente con los estimadores que se mantienen en la memoria del sistema, cuyo tiempo de acceso es mucho más redudido.

Sin embargo, en el caso de los problemas referidos a grafos, este modelo presenta mayores dificultades, tanto a nivel de espacio como del número de accesos sobre el stream de datos por la estructura enlazada de la representación de grafos. Por lo tanto, se han definido variaciones sobre el *modelo en streaming* original para tratar de hacer frente a los requisitos característicos de este tipo de problemas. En los artículos *On graph problems in a semi-streaming model* [FKM$^+$05] y *Analyzing graph structure via linear measurements* [AGM12a] los autores describen dicho modelo, al cual denominan **Modelo en Semi-Streaming**.

Este se corresponde con una relajación del *modelo en streaming* estándar, el cual permite mayor libertad tanto a nivel de espacio como de pasadas permitidas sobre el stream de datos. Por esta razón, cuando el número de pasadas p es superior a 1 ($p > 1$), entonces ya no es posible su uso en entornos en que se requiere que el procesamiento de la información y las consultas sean en tiempo real, algo que, por contra, si sucedía en el caso del *modelo en streaming* definido anteriormente. Por lo tanto, el *modelo en semi-streaming* se presenta como una estrategia de diseño de algoritmos que trata de obtener estimaciones sobre grafos en un espacio reducido y con un buen planteamiento a nivel de accesos a disco cuando el grafo completo no puede mantenerse en la memoria del sistema de manera completa.

El *modelo en semi-streaming* impone la forma en que el grafo es recibido bajo la idea de stream de datos, lo cual se describe a continuación: Sea $G = (V, E)$ un grafo dirigido (su adaptación al caso de grafos no dirigidos es trivial) compuesto por $n = |V|$ vértices y un número indeterminado de arístas desde el punto de vista del algoritmo que procesará el stream. Se presupone que se conoce *a-priori* el número de vértices que forman el grafo, mientras que el stream consiste en el conjunto de tuplas que representan las aristas. El conjunto de aristas E se define como $E = \{e_{i_1}, e_{i_2}, ..., e_{i_j}, ..., e_{i_m}\}$, tal y como se ha hecho en secciones anteriores. Por tanto el grafo G está formado por $m = |E|$ aristas (es necesario remarcar que el algoritmo que procesa el stream no conoce dicho valor). Dichas aristas son procesadas en un orden desconocido de manera secuencial marcado por el conjunto de permutaciones arbitrarias $\{i_1, i_2, i_j, i_m\}$ sobre $[1, m]$.

Una vez descrita la estrategia de procesamiento sobre el *modelo en semi-streaming*, lo siguiente es indicar las unidades de medida sobre las cuales realizar el análisis sobre la complejidad de los algoritmos que se desarrollan sobre este modelo. En [FKM$^+$05], los autores definen $S(n, m)$ como el espacio utilizado para procesar el stream de aristas, $P(n, m)$ el número de pasadas sobre dicho stream y $T(n, m)$ el tiempo necesario para procesar cada arista. Sobre esta contextualización se requiere que para que un algoritmo sea válido en el *modelo en semi-streaming* $S(n, m)$ esté contenido en el orden de complejidad $O(n \cdot polylog(n))$. Tal y como se puede apreciar, esta restrición a nivel de espacio es mucho más relajada que la impuesta sobre el *modelo en streaming* estándar, que requiere de $o(N)$. Sin embargo, es necesario tener en cuenta que gran cantidad de problemas sobre grafos requieren un coste espacial de $O(n^2)$, por lo que tratar de encontrar una solución en $O(n \cdot polylog(n))$ representa una tarea compleja, pero conlleva una mejora significativa.

Al igual que sucede con en el caso del *modelo en streaming* estándar, al utilizar un orden espacial menor del necesario para encontrar soluciones exactas, las soluciones encontradas admiten la existencia de una

determinada tasa de error máxima delimitada por ϵ, la cual se debe cumplir con una probabilidad δ. Para conjuntos de datos sobre los cuales no es admisible la búsqueda de una solución exacta o para los cuales sea admisible una reducida tasa de error, esta estrategia de diseño de algoritmos se presenta por tanto como una alternativa acertada.

4.4. Spanners y Sparsifiers

Para tratar de agilizar los cálculos necesarios para la resolución de problemas sobre grafos, se han propuesto distintas alternativas, muchas de las cuales son aplicables únicamente a problemas concretos. Sin embargo, estas ideas se discutirán en la sección 4.5. En esta sección se describen distintas técnicas utilizadas para tratar de "sumarizar" o disminuir el espacio necesario para almacenar un determinado grafo G (al igual que ocurría con las técnicas descritas en el capítulo anterior para valores numéricos) transformándolo en un nuevo grafo H, de tal manera que la estructura del mismo siga siendo lo más similar a la del original respecto de distintas medidas.

La descripción de estas técnicas se ha basado en las ideas recogidas en los artículos *Graph stream algorithms: a survey* [McG14] de *McGregor* y *Graph sketches: sparsification, spanners, and subgraphs* [AGM12b] de *Ahn y otros*, así como las notas de la *Clase 11* de la asignatura *Randomized Algorithms* [Har11] impartida en la *Universidad de Columbia Británica*. Tal y como se ha indicado en el párrafo anterior, estas técnicas consisten en la eliminación de vértices y/o aristas, de tal manera que la distorsión producida tras dichas modificaciones sea mínima.

Existe un enfoque trivial para la reducción del tamaño en grafos, sin embargo, este tan solo ofrece resultados aceptables sobre grafos densos (aquellos similares al grafo completo K_n). La técnica consiste en la eliminación de aristas a partir de una determinada tasa de probabilidad α. Mediante esta estrategia se reduce el espacio necesario para almacenar las aristas de $(1 - \alpha)$. Tal y como se ha dicho, esta solución tan solo es válida en aquellos casos en que el grafo sea denso. Cuando el grafo es disperso, por contra, la eliminación de aristas mediante la selección uniforme puede producir una gran distorsión respecto del grafo original.

En la figura 4.4 se muestra un grafo disperso, que además forma 3 comunidades (conjuntos de vértices altamente conexos entre sí). Nótese que en aquiellos casos en que el grafo posea una estructura similar a

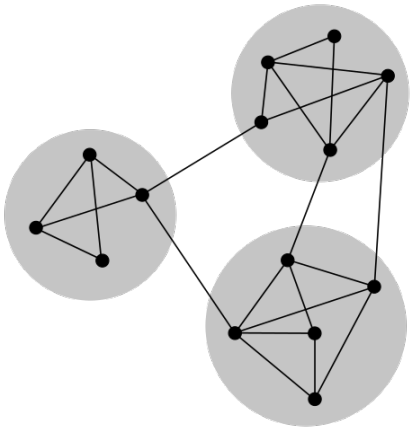


Figura 4.3: Ejemplo de *Grafo Disperso*. (Extraido de [Wik17d])

la del de la figura, ya no es conveniente utilizar la estrategia descrita en el párrafo anterior, puesto que para tratar de preservar la estructura del mismo, no todas las aristas tienen la misma relevancia. Esto puede entenderse de manera sencilla si al comprobar la variación del grafo al eliminar una de las aristas que conectan dos comunidades. Dicha variación estructural es significativamente mayor que la ocurrida tras eliminar una arista que conecta dos vértices pertenecientes a una misma comunidad.

Por esta razón, distintos autores han trabajado en técnicas para tratar de mantener la estructura del subgrafo generado lo más semejante posible a la del grafo original. Para ello, las estrategias más populares se basan en la utilización de *Spanners* y *Sparsifiers*, los cuales se describirán a continuación en las secciones 4.4.1 y 4.4.2 respectivamente. Estas técnicas han sido ampliamente estudiadas, sin embargo, en este caso se pretende orientar la descripción de las mismas sobre el *modelo en semi-streaming* del cual se habló anteriormente. En este modelo se han encontrado soluciones eficientes para *grafos no dirigidos*. Por contra, para el caso de los *grafos dirigidos*, la búsqueda de soluciones eficientes para este problema continua siendo un problema abierto.

4.4.1. Spanners

Se denomina *Spanner* a un determinado subgrafo que mantiene las propiedades de distancia entre todos sus vértices respecto del original con una tasa máxima de variación acotada por α. Por tanto, se denomina α-*spanner* del grafo $G = (V, E)$ a un subgrafo $H = (V, E')$ donde $E' \subset E$, construido de tal manera que se cumpla la ecuación (4.3) representando $d_G(v_{i_1}, v_{i_2})$ la distancia del camino más corto entre los vértices v_{i_1} y v_{i_2} sobre el grafo G.

$$\forall v_{i_1}, v_{i_2} \in V,\ d_G(v_{i_1}, v_{i_2}) \leq d_H(v_{i_1}, v_{i_2}) \leq \alpha \cdot d_G(v_{i_1}, v_{i_2}) \tag{4.3}$$

Tal y como se indica en la ecuación (4.3), lo que se pretende acotar mediante esta estrategia es, por tanto, el error desde el punto de vista de la distancia entre vértices. Tal y como se puede intuir, mediante el cumplimiento de esta propiedad se soluciona el problema descrito anteriormente que surge sobre grafos dispersos, dado que si se elimina el único arista que conecta dos comunidades distintas, entonces la distancia del camino más corto entre los vértices de comunidades distintas variará en gran medida, posiblemente superando la acotada del valor α.

Para la construcción de un α-*spanner* sobre el modelo en streaming existe un algoritmo sencillo que resuelve este problema para el caso del modelo de caja registradora (sección 2.2.3), es decir, en el cual tan solo estén permitadas adicciones. Esta estrategia se ilustra en el algoritmo 3. En este caso, la solución es trivial y consiste en añadir únicamente al conjunto de aristas E' del grafo H, aquellas cuya distancia del camino más corto entre sus vértices en H sea mayor que α, con lo cual se garantiza la propiedad del α-*spanner*.

Sin embargo, esta técnica requiere del cálculo del camino más corto entre los vértices u y v en cada actualización, lo cual genera un coste de $O(card(E'))$ en tiempo, o el mantenimiento de una estructura de datos auxiliar que almacene dichas distancias, lo que requiere de un $O(n^2)$ en espacio. El mejor resultado encontrado para este problema es la construcción de un $(2k-1)$-*spanner* utilizando $O(n^(1+1/k)$ de espacio.

Algorithm 3: Basic Spanner

```
Result: E'
E' ← ∅;
for cada (u, v) ∈ E do
    if d_H(u, v) > α then
        E' ← E' ∪ {(u, v)};
    end
end
```

Esta solución se ha demostrado que es óptima respecto de la precisión utilizando tan solo una pasada sobre el stream de aristas. La descripción completa de la misma se encuentra en el trabajo *Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners* [Elk07] de *Elkin*.

Para el caso general, en el cual están permitidas tanto adicciones como eliminaciones (modelo en molinete descrito en la sección 2.2.4) la solución básica no es trivial. El algoritmo que se describe en [Elk07] se basa en la generación de árboles incrementales que se construyen a partir de la selección aleatoria de vértices. Sin embargo, esto no es sencillo cuando se permiten las eliminaciones. Por lo tanto, para la adaptación de dichas técnicas al modelo en molinete una solución es la utilización de L_0-*Samplers*, que se describieron en la sección 3.7, lo que requiere de múltiples pasadas sobre el stream de aristas.

Tal y como se ha visto en esta sección, la construcción de *Spanners* añade un sobrecoste a la resolución de problemas sobre grafos, junto con una determinada tasa de error desde el punto de vista de la distancia entre vértices. Sin embargo, dichos inconvenientes se ven recompensados en muchos casos por la reducción en tiempo y espacio de la resolución del problema sobre el subgrafo resultante.

4.4.2. Sparsifiers

Otra alternativa para la generación del subgrafo H construido sobre el mismo conjunto de vértices y un subconjunto de aristas respecto del grafo G son los *Sparsifiers*. En este caso, en lugar de tratar de mantener la propiedad de la distancia del camino más corto entre pares de vértices, se basa en la minimización del número de cortes mínimos (eliminación de aristas) para separar el conjunto de vértices del grafo en dos subconjuntos disjuntos para cada par de vértices. A este concepto se lo denomina *corte mínimo* (o *Minimun Cut*) y se denota por $\lambda_{u,v}(G)$ para indicar el número de aristas a eliminar para formar dos subconjuntos disjuntos V_1, V_2 de tal manera que $u \in V_1$ y $v \in V_2$. Nótese por tanto, que en un grafo dirigido el *corte mínimo* puede tomar valores en el intervalo $[1, n \cdot (n-1)]$. En la ecuación (4.4) se muestra la definición formal de *Sparsifier* donde A representa todas las combinaciones de pares de vértices y ϵ la desviación máxima permitida por el *Sparsifier*, de tal manera que el resultado se corresponde con un $(1+\epsilon)$-*Sparsifier*.

$$\forall A \in V \ (1-\epsilon)\lambda_A(G) \leq \lambda_A(H) \leq (1+\epsilon)\lambda_A(G) \tag{4.4}$$

A continuación se describe una definición para entender las ideas subyacentes en que se basan los *Sparsifiers*. Para construir un $(1+\epsilon)$-*Sparsifier* de G, el valor p debe ser escogido tal y como se indica en la ecuación (4.5) donde $\lambda(G)$ representa el corte mínimo de G. La demostración se encuentra en el artículo *On graph problems in a semi-streaming model* [FKM$^+$05] desarrollado por *Feigenbaum y otros*. Por tanto, el problema se reduce

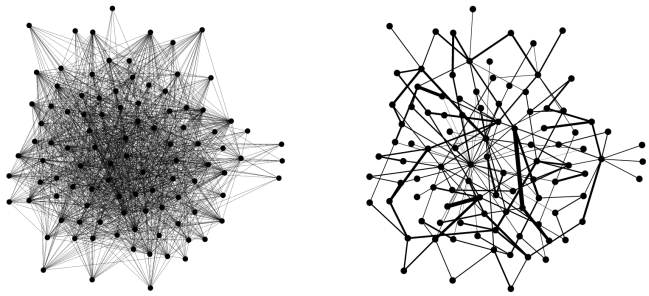


Figura 4.4: Ejemplo de *Sparsifier*. (Extraido de [Har11])

al mantenimiento de un L_0-*Sampler* que devuelva un subconjunto de aristas seleccionados con probabilidad p del stream de aristas.

$$p \geq min\{6\lambda(G)^{-1}\epsilon^{-2}log(n), 1\} \tag{4.5}$$

A continuación se describe una estrategia sencilla de construcción de $(1+\epsilon)$-*Sparsifiers* para grafos no dirigidos. Esta se basa en la selección de aristas con probabilidad p definido tal y como se indicó anteriormente en la ecuación (4.5).

Algorithm 4: Basic Sparsifier

```
Result: E'
E' ← ∅;
for cada (u, v) ∈ E do
    r ← Uniform(0, 1)
    if r < p then
        E' ← E' ∪ {(u, v)};
    end
end
```

Otro enfoque más general para la construcción de *Sparsifiers* se basa en la utilización de la *Matriz Laplaciana* (definida en la sección 4.2.1.3) del grafo H, que denotaremos como L_H. A los *Sparsifiers* construidos manteniendo la propiedad definida en la ecuación (4.6) se los denomina *Spectral Sparsifiers*. Desarollando la ecuación obtenida si restringimos el rango de valores de x al subconjunto $\{0,1\}^n$ entonces podemos obtener la ecuación 4.4. Dado que el vector x modeliza el conjunto A en dicha ecuación. Por tanto, los *Spectral Sparsifiers* pueden ser vistos como una generalización de los anteriores.

$$\forall x \in \mathbb{R} \leq (1-\epsilon)x^T L_G x \leq x^T L_H x \leq (1+\epsilon)x^T L_G x \tag{4.6}$$

Mediante la estrategia de los *Spectral Sparsifiers*, además de aproximar el corte mínimo $\lambda(G)$ se pueden aproximar otras propiedades como propiedades de los paseos aleatorios (que se describen en la sección 4.5.5). En el trabajo *Twice-Ramanujan Sparsifiers* [BSS12] redactado por *Batson y otros* se describe una estrategia de construcción de $(1+\epsilon)$-*Spectral Sparsifiers* en un espacio de $O(\epsilon^{-2}n)$.

Las estrategias descritas en esta sección para la reducción de la complejidad de grafos ajustandose al *modelo en semi-streaming* proporcionan una ventaja significativa a nivel de espacio mediante la redución del conjunto de aristas para la resolución de otros problemas a partir de ellas. Por contra, estas generan una determinada tasa de error.

Cuando se ha hablado de optimalidad en esta sección, ha sido desde el punto de vista de los *grafos no dirigidos*, dado que para el caso de los *grafos dirigidos*, aún no se han encontrado métodos de generación de *Spanners* o *Sparsifiers* óptimos debido a la mayor complejidad que estos conllevan. En la siguiente sección se describen distintos problemas para los cuales se ha encontrado una solución sobre el *modelo en semi-streaming* y sus correspondientes restricciónes.

4.5. Problemas sobre Grafos

El modelo de representación de grafos proporciona un marco de trabajo flexible sobre el cual se pueden modelizar un gran número de situaciones. Un ejemplo característico de esta situación son las redes sociales, en las cuales cada persona representa un vértice en el grafo mientras que las aristas representan las relaciones entre estas. El conjunto de relaciones generadas a partir de los enlaces entre páginas web (*Web Graph*) también son un claro ejemplo de problemas que se pueden representar mediante un grafo. Sin embargo, este modelo permite representar otras muchas situaciones como planificación de tareas donde cada vértice representa una tarea y las aristas marcan el orden de precedencia de las mismas. Los grafos también se pueden utilizar para modelizar el problema de encontrar determinados objetos o estructuras en imágenes.

Muchos de estos problemas pueden extenderse sobre un entorno dinámico, en el cual la estructura del grafo varía con respecto del tiempo. Algunos ejemplos son la conexión con nuevos amigos que se conectan entre sí en un grafo que modele redes sociales, la eliminación de enlaces entre webs en el caso del grafo de la red (*Web Graph*), cambios de última hora debido a factores externos en cuanto a planificación de tareas, o la extensión del reconocimiento de estructuras en vídeos, que pueden ser vistos como la variación del estado de la imagen con respecto del tiempo.

Por estas razones, en esta sección se pretende realizar una descripción acerca de distintos problemas aplicados a grafos sobre el *modelo en semi-streaming*. Los problemas que se describen, generalmente son de carácter básico. Sin embargo, son de vital importancia puesto que a partir de ellos pueden plantearse soluciones a otros de mayor complejidad.

El resto de la sección se organiza de la siguiente manera: En el apartado 4.5.1 se describirá el problema de *Verificación de Grafos Bipartitos*, a continuación se habla del *problema de Conteo de Triángulos* en el apartado 4.5.2, posteriormente se expone el problema de encontrar el *Arbol Recubridor Mínimo* en el apartado 4.5.3, en el apartado 4.5.4 se discute sobre el problema de *Componentes Conectados* y finalmente, en el apartado 4.5.5 se realiza una breve introducción acerca de los *Paseos Aleatorios*, que será extendida en profundidad en el capítulo siguiente, destinado exclusivamente al *Algoritmo PageRank*. Se vuelve a remarcar que estos problemas se describen desde la persepectiva del *modelo en semi-streaming*.

4.5.1. Verificación de Grafos Bipartitos

En la sección 4.2 en la cual se realizó una descripción formal acerca de las definiciones sobre grafos que se utilizarían durante el resto del capítulo se hablo de *Grafos Bipartitos*, que tal y como se indicó, estos se refieren a aquellos grafos que cumplen la propiedad de formar dos subconjuntos disjuntos de vértices que se relacionan entre si mediante aristas incidentes sobre 2 vértices cada uno perteneciente a un subconjunto distinto. Tal y como se indicó anteriormente, en la figura 4.2 se muestra un ejemplo de grafo bipartito.

En el trabajo *On graph problems in a semi-streaming model*[FKM$^+$05] (en el cual se expone por primera vez el modelo en semi-streaming) los autores ilustran un algoritmo con un coste espacial de $O(n \cdot log(n))$ procesando cada arista en $O(1)$ y realizando $O(log(1/\epsilon))\epsilon$ pasadas sobre el stream de aristas que indica si se cumple la propiedad de grafo bipartito.

El algoritmo se basa en lo siguiente: Una fase inicial de construcción de un *Matching Maximal* (selección de aristas de tal manera que ninguna sea indicente del mismo vértice que otra). Esta tarea se realiza sobre la primera pasada del stream de aristas. El resto de pasadas se basan en la adicción de aristas para hacer conexo el grafo de tal manera que se no se formen ciclos. Si esta estrategia es posible para todas las aristas del stream, entonces se cumple la propiedad de grafo bipartito.

Existen otras estrategias para probar que un grafo cumple la propiedad de grafo bipartito. Cuando se hable de problemas de conectividad en la sección 4.5.4 se expondrá otra estrategia para la validación de la propiedad de grafos bipartitos.

4.5.2. Conteo de Triángulos

Uno de los problemas que se ha tratado en profundidad sobre el *modelo en semi-streaming* aplicado a grafos es el *conteo de triángulos*. Este problema se define como el número de tripletas de vértices conectadas entre sí mediante tres aristas de tal manera que estas sean incidentes a dichos vértices. Una modelización más precisa se define a continuación sobre el grafo $G = (V, E)$.

Sean $v_{i_1}, v_{i_2}, v_{i_3} \in V$ tres dístintos vértices del grafo G y $e_{j_1}, e_{j_2}e_{j_3} \in E$ tres aristas disintos de dicho grafo. Entónces se denomina triángulo a la tripleta $\{e_{j_1}, e_{j_2}e_{j_3}\}$ si se cumple que $e_{j_1} = (v_{i_1}, v_{i_2})$, $e_{j_2} = (v_{i_2}, v_{i_3})$ y $e_{j_3} = (v_{i_3}, v_{i_1})$. Al cardinal del conjunto de tripletas distintas sobre el grafo G se lo denomina T_3. Esta definición puede extenderse a figuras de con mayor número de vértices modificando el valor 3 por otro mayor. Sin embargo, estos casos son de menor interés puesto que aportan información similar sobre la estructura del grafo pero requieren un mayor coste computacional para su cálculo.

Se han encontrado estrategias para soluciones aproximadas al problema del *conteo de triángulos* sobre las restricciones del *modelo en semi-streaming*. La primera propuesta expuesta en *Reductions in streaming algorithms, with an application to counting triangles in graphs* [BYKS02] por *Bar-Yossef y otros* fue la modelización del problema como el conteo de aristas que unen todas aquellas tripletas de nodos, denotadas como $x_{\{v_{i_1}, v_{i_2}, v_{i_3}\}}$ cuyo valor es 3.

Mediante la estimación de momentos de frecuencia (de los cuales se habló en la sección 2.7) se puede calcular el número de triángulos distintos tal y como se indica en la ecuación 4.7. En el trabajo [BYKS02] se muestra una descripción acerca del algoritmo para encontra una $(1+\epsilon)$-*estimación* basada en esta técnica con un coste computacional de $O(\alpha^{-2})$ en espacio donde $\alpha = \epsilon/(8 \cdot m \cdot n)$.

$$T_3 = F_0 + -1.5F_1 + 0.5F_2 \tag{4.7}$$

Otra propuesta es la estimación de T_3 mediante el uso de un L_0-*Sampler* (del cual se habló en la sección 3.7). De esta técnica se habla en [AGM12b] y presupone que si el cumplimiento de la propiedad $x_{\{v_{i_1},v_{i_2},v_{i_3}\}} = 3$ se denota por Y y representa un *proceso de Bernoulli*, entonces esta se puede extender a una distribución binomial al probar todas las combinaciones posibles. Los autores exponen que $\mathbb{E}[Y] = T_3/F_0$ y dado que existen técnicas para la estimación de F_0 (como el algoritmo de *Flajolet-Martin*), entonces es posible obtener T_3.

La razón por la cual resulta interesante el conteo de triángulos es que se utiliza para el cálculo del *coeficiente de agrupamiento* que denotaremos como C. La forma de calcular dicha propiedad se muestra en la ecuación (4.8). A partir de dicho indicador se puede obtener una estimación sobre la estructura del grafo, es decir, si es altamente conexo o por contra es disperso.

$$C = \frac{1}{n} \sum_{v \in V} \frac{T_3(v)}{\binom{deg(v)}{2}} \tag{4.8}$$

4.5.3. Árbol Recubridor Mínimo

El problema de la búsqueda del *árbol recubridor mínimo* (*Minimum Spanning Tree*) es uno de los casos más estudiados en problemas de grafos. Se refiere a la búqueda del subgrafo $H = (V, E')$ respecto del grafo $G = (V, E)$ formado por todos los vértices del grafo G y un subconjunto de aristas del mismo.

La propiedad que debe cumplir el subconjunto de aristas E' es que tan solo debe existir un único camino para llegar desde un vértice cualquiera al resto de vértices del grafo y, además, el sumatorio de los pesos de las aristas contenidas en dicho camino debe ser el mínimo respecto de todos los posible en G. En la figura 4.5 se muestra un ejemplo del *árbol recubridor mínimo* sobre un grafo ponderado

Nótese que para que exista un *árbol recubridor mínimo* el grafo G debe ser conexo. En el caso de que G no sea conexo entonces se habla de *bosque recubridor mínimo* (*Minimum Spanning Forest*). El problema del *árbol recubridor mínimo* no siempre tiene una solución única, sino que pueden encontrarse distintos subgrafos H_i que cumplan la propiedad utilizando subconjuntos de aristas diferentes. El caso más carácterístico de esta situación es cuando se busca el *árbol recubridor mínimo* sobre un grafo no ponderado, es decir, aquel en el cual todas las aristas tienen el mismo peso.

Este problema se ha definido comúnmente sobre grafos no dirigidos, por su mayor simplicidad y aplicación práctica. Sin embargo, la modelización es extensible a grafos dirigidos, para los cuales el problema es mucho más complejo dado que por cada vértice es necesario mantener un arista de entrada y otro de salida.

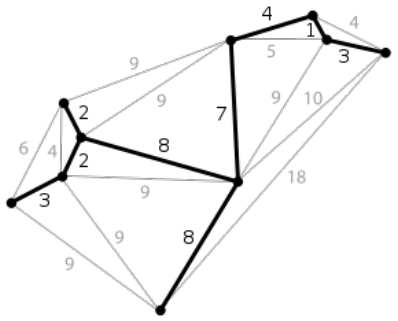


Figura 4.5: Ejemplo de *Árbol Recubridor Mínimo*. (Extraido de [Wik17f])

La versión estática del problema del *árbol recubridor mínimo* ha sido ampliamente estudiada en la literatura, existiendo un gran número de alternativas, entre las que destaca históricamente el *Algoritmo de Kruskal* descrito en el artículo *On the shortest spanning subtree of a graph and the traveling salesman problem* [Kru56] cuya complejidad temporal es de $O(n + log(m))$ sobre un espacio de $O(n + m)$.

También se han propuesto soluciones basadas en algoritmos probabilistas, que ofrecen garantias de optimalidad en su solución. La versión más eficiente encontrada hasta la fecha se describe en el trabajo *An optimal minimum spanning tree algorithm* [PR02] de *Pettie y otros*. Sin embargo, debido a las técnicas que utiliza (mediante reducción a *Árboles de Decisión*) no se puede representar mediante una función, pero sí que se ha demostrado su optimalidad. El problema también se ha estudiado desde la perspectiva de la parelelización. En el trabajo *Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs* [BC06] *Bader y otros* muestran un algoritmo que mediante el multiprocesamiento consiguen soluciones 5 veces más eficientes que la versión optimizada secuencial.

Una vez descrito el problema y su estado actual sobre el modelo estático, el resto del apartado se basará en la descripción del mismo sobre las restricciones del *modelo en semi-streaming*. Para ello, se realiza una descripción del algoritmo exacto propuesto por *Ahn y otros* en *Analyzing graph structure via linear measurements* [AGM12a]. Dicha estrategia es capaz de encontrar el *árbol recubridor mínimo* en $O(log(n)/log(log(n)))$ pasadas sobre el stream de aristas ajustandose a las restricciones espaciales de $O(polylog(n))$.

El planteamiento del algoritmo en semi-streaming se basa inicialmente en el *algoritmo de Boruvka's* (en [Wik17c] se encuentra una breve descripción), cuyo planteamiento es el siguiente: se realizan $O(log(n))$ de tal manera que en cada una de ellas se seleccionan las aristas de menor peso que conecten vértices aún no marcados como conectados. De manera incremental se construye el *árbol recubridor mínimo* hasta que todos los vértices del grafo están conectados.

Tal y como se indica en el documento, el *algoritmo de Boruvka's* puede emularse fácilmente sobre el modelo en semi-streaming emulando cada fase en $O(log(n))$ pasadas por el stream (para encontrar el arista de menor peso), lo cual conlleva $O(log(log(n)))$ pasadas en total. La idea en que se basa el algoritmo para reducir el número de pasadas sobre el streaming es realizar la búsqueda del arista de menor peso de cada nodo en "paralelo", teniendo en cuenta las limitaciones espaciales ($O(polylog(n))$) lo cual es posible en $O(log(n)/log(log(n)))$.

El problema del *árbol recubridor mínimo* tiene aplicaciones prácticas en muchos entornos reales, como el diseño de redes de telecomunicaciones o de transporte. También se han encontrado reducciones sobre el *Problema del Viajante* y el *Problema del corte mínimo*. Otros usos son el análisis de estructuras de grafos, problemas de clustering o reconocimiento de estructuras sobre imágenes.

4.5.4. Componentes Conectados

El problema de *Componentes Conectados* se refiere a la búsqueda del cardinal de subconjuntos de vértices para los cuales existe un camino entre todos los vértices del subconjunto. Mediante este resultado se puede determinar por tanto si existe un camino que entre dos vértices simplemente comprobando si pertenecen al mismo subconjunto de componentes conectados. Nótese por tanto que un grafo conexo tan solo posee un único componente conectado que contiene todos los vértices del grafo.

Se denota como $cc(G)$ al cardinal de componentes conectados del grafo G. El algoritmo clásico para resolver este problema se describe a continuación. Este requiere de $O(log(n))$ fases para su finalización y se basa en la fusión de vértices conectados. En cada fase se fusiona cada vértice con otro que posea una arista incidente a el para después crear un *super-vértice* con el cual se relacionen las aristas de los dos vértices fusionados. Tras realizar esta operación repetidas veces, el algoritmo termina cuando ya no es posible fusionar mas vértices. El número de *super-vértices* resultantes, por tanto es equivalente a $cc(G)$.

En el trabajo *Analyzing graph structure via linear measurements* [AGM12a] se describe un algoritmo para realizar dicha tarea sobre el *modelo en semi-streaming* en una única pasada y con una complejidad espacial de $O(n \cdot log(n))$. Para ello, se basa en la construcción de sketches a partir del L_0-*Samplers* denotados como $S_1, S_2, ..., S_t$ con $t = O(log(n))$. Esto conlleva un coste espacial de $O(n \cdot t \cdot log(log(n)))$, por lo que es válido sobre el *modelo en semi-streaming*. La idea es la construcción jerárquica de estos sketches, de tal manera que S_1 represente el sketch de las aristas de los vértices del grafo G, S_2 las aristas de los *super-vértices* generados en la primera iteracción, y así sucesivamente, de tal manera que a partir de S_t se puede obtener $cc(G)$ al igual que en el algoritmo básico.

En [AGM12a], además, se extiende dicho algoritmo para el problema de *k-aristas conectadas*, que indica el número mínimo de aristas incidentes sobre cada vértice. Las aplicaciones prácticas tanto de este problema como el de *componentes conectados* son de utilizad para conocer la estructura del grafo. Un ejemplo práctico se da en grafos referidos a redes sociales, en las cuales se pretende encontrar el número de agrupaciones (o *clusters*) de usuarios aislados del resto, así como el número mínimo de amistades que presenta cada usuario.

4.5.5. Paseos Aleatorios

Un paseo aleatorio se define como la distribución de probabilidad referida a la realización de un camino de longitud l desde un determinado vértice de inicio fijado *a-priori*. Suponiendo que en cada vértice se tiene una distribución de probabilidad sobre sus aristas incidentes, entonces este problema está intimamente relacionado con las *cadenas de Markov*.

El algoritmo PageRank se refiere a la obtención del estado estacionario de la distribución de probabilidad de los paseos aleatorios con algunas modificaciones. Tanto los *paseos aleatorios* como las cadenas de Markov se describen en detalle en el capítulo 5 destinado al algoritmo PageRank.

4.6. Conclusiones

A lo largo del capítulo se han citado distintas estrategias mediante las cuales se pretende reducir el grado de complejidad para la resolución de problemas sobre grafos, lo cual implica una reducción del tiempo de computo y del espacio necesario. Estas técnicas están ganando una gran relevancia en la actualidad debido a la necesidad de obtener respuestas en un periodo corto de tiempo, lo cual permite mejorar la toma de decisiones.

Debido al dinamismo del entorno en que vivimos, en un gran número de ocasiones es más beneficioso encontrar soluciones aproximadas que a pesar de no otorgar el resultado óptimo, son obtenidas en un tiempo mucho menor, lo cual permite una rápida adaptación a los cambios. Por lo cual, a través de este punto de vista, se pueden obtener mayores beneficios en promedio, puesto que los sobrecostes temporales propiciados por el tiempo de respuesta en soluciones exactas muchas veces conllevan una rápida obsolescencia de los resultados.

Se cree que en los próximos años, el estudio e investigación en el ámbito de la resolución de problemas sobre grafos de tamaño masivo será creciente, al igual que sucedia en el caso numérico como se indicó en anteriores capítulos. A pesar de ello, debido a su mayor grado de dificultad, dichos avances son lentos y actualmente se encuentran en fases muy tempranas de desarrollo, pero tal y como se ha indicado, se espera que esta situación cambie en el futuro.

Capítulo 5

Algoritmo PageRank

5.1. Introducción

El algoritmo *PageRank* fue nombrado por primera vez en el trabajo *The PageRank citation ranking: Bringing order to the web* [PBMW99] publicado por *Larry Page* y *Sergey Brin*. La motivación del mismo fue tratar de realizar un *ranking de importancia* (o relevancia) sobre los nodos de un *grafo dirigido no ponderado* de tal manera que este esté basado únicamente en la estructura de dicho grafo.

La motivación de dicho trabajo fue la de tratar de mejorar el ranking de resultados del buscador de sitios web *Google* en que estaban trabajando. Hasta la publicación de dicho trabajo, los sistemas de búsqueda se basaban en heurísticas como el número de ocurrencias de la palabra clave sobre la cual se basaba la búsqueda o el número de enlaces hacia dicha página.

Sin embargo, los rankings basados en este tipo de estrategias podían ser fácilmente manipulables con el fin de tratar de conseguir posicionarse en las primeras posiciones del sistema de búsqueda. Por ejemplo, una página web que se basara en la repetición de la misma palabra muchas veces, entonces aparecería en primer lugar en rankings basados en el número de ocurrencias para dicha palabra clave. En el caso de rankings basados en el número de enlac hacia dicha página, tampoco sería complejo manipular el resultado creando un número elevado de páginas web que contuvieran links hacia la página para la cual se pretende mejorar su posicionamiento.

La solución propuesta por *Page* y *Brin* para tratar de solucionar dicha problemática se basa en la generación de un ranking sobre los sitios web basado en la estructura del grafo subyacente, de tal manera que los vértices (sitios web) sobre los cuales existan aristas (enlaces) que provengan de otros vértices relevantes, tendrán una puntuación mayor que la de vértices que cuyo subconjunto de aristas los relacione con otros vértices menos relevantes.

La idea en que se basa el ranking se refiere por tanto a que los sitios web a los cuales se puede acceder a partir de otros sitios web considerados como importantes, entonces deberan encontrarse en primeras posiciones. Esta idea se extiende de manera inductiva sobre todos los vértices del grafo, puesto que tal y como veremos en las siguientes secciones converge hacia un estado estable (o *distribución estacionaria* desde el punto de vista estadístico)

Para tratar de facilitar la comprensión acerca de esta idea a continuación se expone un ejemplo: Supongamos que en una red social como *Twitter* (la cual se puede entender como un conjunto de usuarios que se relacionan entre si mediante relaciones de seguimiento, por lo que se puede ver como un grafo dirigido no poderado donde el conjunto de usuarios se refiere a los vértices y el conjunto de relaciones de seguimiento con las aristas) un usuario habitual (el cual no tiene un número elevado de seguidores) comienza a seguir a un antiguo amigo de la universidad, el cual tampoco tiene un gran número de seguidores.

La red social *Twitter* envía una notificación a todos seguidores del usuario indicando que este ha empezado a seguir a su amigo de la universidad. Puesto que su número de seguidores es bajo dicha acción no tendrá una gran relevancia y muy probablemente nadie más comience a seguir al amigo de la universidad. Sin embargo, supongamos que nuestro usuario, en lugar de tener un conjunto reducido de seguidores, es una persona influyente en la red social, a la cual siguen millones de personas, entonces la notificación del nuevo seguimiento le llegará a muchos más usuarios y probablemente el amigo de la universidad verá como su número de seguidores aumenta notablemente.

A grandes rasgos, esta es la idea en que se basa el algoritmo *PageRank*, es decir, la puntuación de un vértice del grafo se basa en la relevancia de los vértices que contienen aristas que apuntan hacia el propio vértice.

La idea inicial del algoritmo *PageRank* era la de realizar un ranking basado en la estructura del grafo de la web (*Web Graph*), sin embargo, tal y como se verá a lo largo del capítulo, los conceptos matemáticos en que se basa dicho ranking son extrapolables a cualquier entorno que se pueda representar a partir de una red o grafo. En el trabajo *PageRank beyond the Web*[Gle15] *Gleich* realiza un estudio acerca de los distintos entornos sobre los cuales se han aplicado estas ideas.

Entre otros, se han realizado trabajos sobre los cuales se ha aplicado el algoritmo *PageRank* en áreas tan dispares como la Biología, para analizar las células más importantes a partir de las interrelaciones entre ellas. También se ha aplicado en el sector de la neurociencia por razones similares. En el caso de la literatura, se ha aplicado sobre el grafo generado a partir del sistema de citaciones de artículos de investigación. Otros ámbitos de aplicación han sido sistemas de planificación de tareas o incluso estudios acerca de resultados en deportes, para conocer los encuentros más relevantes.

El resto del capítulo se organiza como sigue: Lo primero será hablar de *Paseos Aleatorios* en la sección 5.2, lo cual permitirá comprender en mayor medida las idea sobre las cuales se basa el ranking *PageRank*. A continuación se realizará una definición formal acerca del problema y lo que se pretende resolver en la sección 5.3. Una vez entendido el problema sobre el que se está tratando, en la sección 5.4 se describen las formulaciones para resolver el problema desde un punto de vista *Algebraico* (sección 5.4.1), *Iterativo* (sección 5.4.2) y *basado en Paseos Aleatorios* (sección 5.4.3). El siguiente paso es discutir cómo se puede añadir personalización al ranking, lo cual se lleva a cabo en la sección 5.5. Por último, se presentan distintas alternativas al algoritmo *PageRank* en la sección 5.6, en la cual se habla de *HITS* (sección 5.6.1), *SALSA* (sección 5.6.2) y *SimRank* (sección 5.6.3). Finalmente se realiza un breve resumen del capítulo en la sección 5.7.

5.2. Paseos Aleatorios

En esta sección se trata el concepto de *Paseos Aleatorios*, el cual está intimamente relacionado con el algoritmo *PageRank*. Para el desarrollo de esta sección se ha utilizado como herramienta bibliográfica el libro *Randomized algorithms* [MR10] de *Motwani* y *Raghavan*, concretamente se ha prestado especial atención al *capítulo 6: Markov Chains and Random Walks*. El resto de la sección se basará en la descripción de propiedades relacionadas con *Paseos Aleatorios* para finalizar ilustrando la relacion de estos con *PageRank*

Lo primero que haremos será describir en qué consiste un *Paseo Aleatorio*. Para ello nos referiremos al grafo $G = (V, E)$ dirigido y no ponderado, el cual esta compuesto por $n = card(V)$ vértices y $m = card(E)$ aristas. Un paseo aleatorio se refiere entonces a un camino de longitud l con origen en el vértice v_{i_1} y final en v_{i_l}. Para que dicho camino constituya un paseo aleatorio, cada paso debe haber sido generado seleccionando de manera uniforme el siguiente vértice a visitar de entre los adyacentes al vértice actual. Nótese que este comportamiento puede ser visto como el promedio del modo en que los usuarios navegan por internet, de tal manera que acceden a páginas web mediante los enlaces que encuentran en la página que están visualizando. A continuación se describen las *Cadenas de Markov* por su relación como herramienta de estudio para los paseos aleatorios.

5.2.1. Cadenas de Markov

Para el estudio de paseos aleatorios, es apropiado utilizar la abstracción conocida como *Cadenas de Markov*, las cuales están intimamente relacionadas con el concepto de grafo y máquina de estados. Una *Cadena de Markov M* se define como un proceso estocástico que consta de n posibles estados, los cuales tienen asociadas un conjunto de probabilidades denotadas como $p_{ij} = \frac{A_{ij}}{d^-(i)}$ para indicar la probabilidad con la cual se pasará del estado i al estado j.

Dichas probabilidades se pueden representar de manera matricial sobre una matriz de transiciones P de tamaño $n * n$, de tal manera que la posición (i, j) contenga el valor p_{ij} construido tal y como se indica en el párrafo anterior. Notese por tanto, que $\sum_j p_{ij} = 1$ para que la distribución de probabilidades sea válida, es decir, la suma de probabilidades para las transiciones sobre cada estado deben sumar 1.

Supongase que se realiza un paseo aleatorio sobre la *Cadena de Markov M* cuya longitud l es muy elevada ($l \gg n^2$), entonces, es fácil intuir que se visitará más de una vez cada estado. Sin embargo, el ratio de veces que se visitará cada estado muy probablemente no se distribuirá de manera uniforme, sino que habrá estados que serán visitados muchas más veces que otros. Esto depende en gran medida de la matriz de transicciones P. A la distribución de probabilidad generada sobre el ratio de visitas sobre cada nodo tras aplicar un paseo aleatorio de longitud elevada se le conoce como *distribución estacionaria* y se denota como π.

En la figura 5.1 (Extraido de [SSCRGM12]) se muestra una cadena de markov constituida por 3 estados, tando en su forma de grafo dirigido como de forma matricial.

La distribución estacionaria π existe siempre que la *Cadena de Markov M* permita que a partir de un determinado estado i, se pueda llegar al menos a otro estado j. La razón se debe a que si se llega al estado i y este no contiene más posibles estados de salida, entonces el resto de épocas se seguirá en el mismo estado. A los estados que poseen esta característica se los denomina sumideros. La segunda restricción para que se

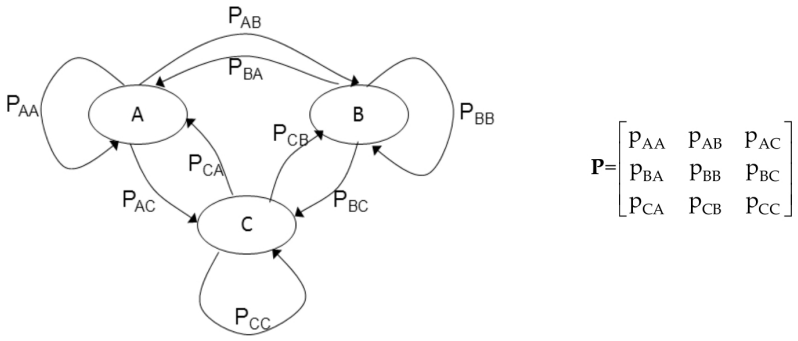


Figura 5.1: Ejemplo de *Cadena de Markov*. (Extraido de [SSCRGM12])

pueda calcular la distrbución estacionaria π es que la matriz de transiciones P no debe ser periodica, es decir, no debe contener ciclos de probabilidad constante sobre los cuales el paseo aleatorio se quedaría iterando de manera indefinida.

Las definiciones descritas en este apartado se pueden extender de manera trivial al caso de grafos dirigidos sin más que utilizar cada vértice del grafo como un estado y construir la matriz P de tal manera que $p_{ij} = \frac{A_{ij}}{d^-(i)}$ donde $d^-(i)$ representa el cardinal de aristas cuyo origen es el vértice i. Tal y como se verá posteriormente, el vector π se corresponde con el resultado obtenido por el algoritmo *PageRank* sobre una matriz P de transicciones modificada.

5.2.2. Matriz Laplaciana de Paseos Aleatorios Normalizada

En la sección 4.2.1.3 se habló sobre la *Matriz Laplacina*, la cual es una estrategia de representación, que ilustra distintas propiedades sobre el grafo subyacente. En este caso se describe una variación de la misma que es más apropiada para problemas relacionados con *Paseos Aleatorios*. Esta se denota como L^{rw} y se denomina *Matriz Laplaciana de Paseos Aleatorios Normalizada*. La estrategia de construcción de la misma se indica en la ecuación (5.1). Esto consiste en asignar a la posición (i, j) el opuesto de la probabilidad de transicción del vértice i al vértice j. Además, en la diagonal (i, i) se asigna el valor 1 cuando el grado del vértice es mayor que 0.

$$L_{i,j}^{\text{rw}} := \begin{cases} 1 & \text{if } i = j \text{ and } d(v_i) \neq 0 \\ -P_{ij} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

Una vez descritos los *Paseos Aleatorios*, junto con las *Cadenas de Markov* y la *Matriz Laplaciana de Paseos Aleatorios Normalizada*, ya se está en condiciones suficientes como para describir de manera formal el *PageRank* de un determinado grafo, que se realizará en la siguiente sección. Para ello, se indicarán las dificultades que surgen sobre este problema en grafos reales, así como las soluciones utilizadas para poder hacer frente a estas.

5.3. Definición Formal

Se define el *PageRank* como la *distribución estacionaria* π de un determinado grafo dirigido no ponderado G sobre el cual, la matriz de transicciones P, ha sido ligeramente modificada. Tal y como se ha visto en la sección anterior, la *distribución estacionaria* consiste en la probabilidad de encontrarse en el estado i durante un paseo aletorio de longitud elevada sobre la *Cadena de Markov* M. Tal y como se ha indicado, para que una cadena de markov sea válida, entonces no deben existir estados *sumideros* (no tienen posibles estados próximos).

Por estas razones, obtener la *distribución estacionaria* de un grafo G, este no debe contener vértices *sumideros*. La solución que proponen *Page* y *Brin* en [PBMW99] para encontrar la *distribución estacionaria* o *PageRank* del grafo generado por los enlaces de la web (*Web Graph*) es añadir un determinado índice de probabilidad sobre el cual los usuarios dejan de seguir los enlaces entre páginas web para acceder a otra distinta introducciendo la URL directamente. El apoyo en esta estrategia soluciona por tanto el problema de los vértices *sumidero*, además de asemejarse en un mayor grado al comportamiento real de un usario que navega por internet.

En [PBMW99] *Page* y *Brin* proponen la modificación de la matriz de transiciones P para la adaptación descrita en el párrafo superior, la cual se indica en la ecuación (5.2). De esta manera, se representa la situación en la cual un determinado usuario que llega a una página web sin enlaces hacia otras (*sumidero*), accede a otra seleccionada de manera uniforme (esto se modeliza mediante el vector p construido de tal manera que $p_i = \frac{1}{n},\ \forall i \in [1, n]$). Además, se añade un determinado índice β, que se corresponde con la probabilidad de que el usuario continue seleccionando enlaces en la página actual o ,por contra, acceda a otra seleccionandola de manera uniforme. Típicamente el valor β se fija a 0.85, sin embargo, admite cualquier valor contenido en el intervalo $[0, 1]$.

$$p'_{ij} = \begin{cases} \beta * \frac{A_{ij}}{d^-(i)} + (1 - \beta) * p_i & \text{if } d^-(i) \neq 0 \\ p_i & \text{otherwise} \end{cases} \tag{5.2}$$

Tal y como se ha indicado anteriormente, el vector p representa la distribución de probabilidad referida a los saltos que un usuario lleva a cabo entre sitios web sin tener en cuenta los enlaces del sitio web actual. En el párrafo anterior se ha indicado que este vector es construido siguiendo una distribución uniforme, por tanto, esto puede ser visto de tal manera que la probabilidad de saltar de un sitio web a otro es la misma. Sin embargo, dicha acción podría seguir una distribución de probabilidad distinta dependiendo de cada usuario de la red. Por tanto, en [PBMW99] se habla de *PageRank Personalizado* cuando el vector p sigue una distribución de probabilidad distinta de la uniforme (desviada hacia los sitios web a los que más accede el usuario).

En el trabajo *Topic-sensitive pagerank* [Hav02] *Haveliwala* propone la generación de 16 *distribuciones estacionarias* (*PageRanks*) distintas mediante la personalización del vector v, para después realizar una combinación de estas y así conseguir que el ranking final sea personalizado.

Una vez descritas las transformaciones necesarias a realizar sobre la matriz de transiciones P para que esta se adapte a la estructura de grafos con vértices *sumidero*, y que además emule de manera más apropiada el

comportamiento de un determinado usuario sobre el grafo de la web (*Web Graph*), lo siguiente es explicar cómo se puede obtener la *distribución estacionaria* o *PageRank* del grafo. Para ello, a continuación se describe el *Teorema de Perron–Frobenius*, que aporta una idea acerca de la manera en que se calcula, además de asegurar la convergencia de la matriz de transiciones hacia un estado estacionario del vector π.

5.3.1. Teorema de Perron–Frobenius

El *teorema de Perron–Frobenius* se refiere a la existencia de un **único** *vector propio* (*eigenvector*) para las matrices cuadradas reales positivas. Dicha descripción ha sido extraida del documento *Notes on the perron-frobenius theory of nonnegative matrices* [Boy05] de *Boyle* (profesor de matemáticas de la *Universidad de Marylan*). En primer lugar es necesario describir los conceptos de *vector propio* como de *matriz cuadrada real positiva* para después ver que la *distribución estacionaria* y la *matriz de transiciones* referidos a una *Cadena de Markov* M pueden ser vistos de esta manera.

Una *matriz cuadrada real positiva* A es aquella formada por n filas y n columnas ($n * n$ celdas) para las cuales $\forall i, j \in [1, n]$ se cumple que $A_{ij} \in \mathbb{R} \geq 0$. Tal y como se puede apreciar, la *matriz de transiciones modificada* P' del grafo G cumple esta propiedad ya que $\forall i, j \in [1, n]$ $P'_{ij} \in [0, 1]$.

En cuanto al concepto de *vector propio* λ de un matriz, se refiere a un vector de n columnas ($1 * n$) tal que cuando es multiplado por una determinada matriz A, el resultado sigue siendo el mismo. Es decir, se cumple que $\lambda = \lambda * A$. Notese por tanto, que esta idea es equivalente a la *distribución estacionaria* desde el punto de vista de llegar a un estado estable.

El teorema de *teorema de Perron–Frobenius* asegura por tanto, que para una *matriz cuadrada real positiva* A tan solo existe un único *vector propio* λ y el conjunto de valores de este se es estrictamente positivo, es decir, $\forall i \in [1, n]$ $\lambda_i \geq 0$. La demostración de dicho teorema puede encontrarse en [Boy05].

Además, en el caso de que la matriz A haya sido normalizada por columna, es decir, se cumpla que $\forall i \in [1, n]$ $\sum_j A_{ij} = 1$, entonces el autovector λ también seguirá la propiedad de normalización ($\sum_i \lambda_i = 1$). Gracias a este resultado es posible calcular la *distribución estacionaria* π de una *Cadena de Markov* como el *vector propio* de su matriz de transiciones.

Una vez descrito el *teorema de Perron–Frobenius* ya se está en condiciones suficientes para describir las distintas alternativas para calcular el *PageRank* de un determinado grafo, el cual se calcula tal y como se ha indicado en esta sección, encontrando el *vector propio* de la matriz de transición modificada de la cadena de markov. Las distintas estrategias para obtener este resultado se describen en la siguiente sección.

5.4. Algoritmo Básico

En esta sección se describe el método para obtener el vector *PageRank* sobre un determinado grafo G. Para ello, es necesario fijar 3 parámetros los cuales se indican a continuación:

- Matriz de Adjacencia A, que a partir de la cual se obtiene la estructura del grafo (Se habló de ella en la sección 4.2.1.1).

- El valor de probabilidad β de seguir el paseo aleatorio a partir de la distribución del vértice actual (el cual se comentó en la sección anterior).

- El vector de personalización p referido a la distribución de probabilidad de los saltos aleatorios entre vértices (también se habló en la sección anterior).

Para calcular el *vector propio* λ existen distintas estrategias matemáticas. En esta sección se habla de dos estrategias, la primera de ellas basada en la resolución de un sistema de ecuaciones lineales mientras que la segunda se basa en acercamiento a la solución de manera iterativa. Tal y como se verá a continuación, la estrategia algebraica conlleva un coste computacional muy elevado, por lo que no es admisible sobre grafos de tamaño masivo. En estos casos se utiliza la estrategia iterativa u otras alternativas basadas en la generación de *Paseos Aleatorios*.

5.4.1. Estrategia Algebraica

La idea de la estrategia algebraica se refiere a la búsqueda del vector λ que resuelva la ecuación $\lambda = \lambda * P'$ como un sistema de ecuaciones lineales. Esto se puede llevar a cabo siguiendo el desarrollo de la ecuación (5.3). Nótese que para ello no se utiliza la *matriz de transiciones modificada* P' explicitamente. En su lugar, esta es representada implicitamente a partir de las operaciones de (5.4) y (5.5).

Para entender estas ecuaciones, lo primero es indicar la notación que se ha utilizado así como la interpretación de algunas operaciones: El símbolo $\boldsymbol{I}$ representa la matriz identidad (1's en la diagonal y 0's en el resto) de tamaño n. El símbolo d^- representa un vector columna de tamaño n que representa en su posición j el cardinal de aristas cuyo origen es el vértice j. Esto puede ser visto de la siguiente manera: $\forall j \in [1, n]\ \sum_i A_{ij} = d_j^-$. A nivel de operaciones es necesario resaltar el caso de la división $\frac{A}{d^-}$ por su carácter matricial. Esta se lleva a cabo realizando la división elemento a elemento por columnas.

$$\lambda = \lambda * P' \tag{5.3}$$

$$= \lambda * \beta * \frac{A}{d^-} + (1 - \beta) * p \tag{5.4}$$

$$= \left(\boldsymbol{I} - \beta * \frac{A}{d^-}\right)^{-1} * (1 - \beta) * p \tag{5.5}$$

A partir de las operaciones descritas en la ecuación (5.5), se obtiene por tanto el *vector propio* λ, que en este caso se refiere a la *distribución estacionaria* de la *cadena de markov* descrita por la matriz de transiciones modificada P', por lo que es equivalente al vector *PageRank*. Sin embargo, el calculo del vector *PageRank* siguiendo esta estrategia conlleva un elevado coste computacional derivado de la necesidad de invertir una matriz de tamaño $n * n$, algo inadmisible para grafos de tamaño masivo.

5.4.2. Estrategia Iterativa

La *estrategia iterativa* para el cálculo del vector *PageRank* se basa en la aproximación a este mediante la multiplicación del mismo por la matriz de transiciones modificada P' de manera repetida hasta que este llegue a un estado estable desde el punto de vista de una determinada norma vectorial.

El primer paso es calcular la *matriz de transiciones modificada* P' a partir de los tres parámetros de entrada (A, β, v) siguiendo la definición de la ecuación (5.2). Una vez obtenida dicha matriz se está en condiciones de calcular el vector *PageRank* siguiendo la idea del *vector propio* único expuesta en la sección anterior.

El algoritmo para dicha tarea se muestra en la figura referida al *Algorimo 5*. Este toma como argumentos de entrada la matriz de transición aproximada P' junto con un determinado valor de convergencia $\in (0, 1)$, que condiciona la precisión del resultado así como el número de iteracciones necesarias para llegar a él.

Algorithm 5: Iterative PageRank

Result: $\pi(t)$

1 $t \leftarrow 0$;
2 $\pi(t) \leftarrow \frac{1}{n} * \mathbf{1}$;
3 **do**
4 $\quad t \leftarrow t + 1$;
5 $\quad \pi(t) = \pi(t-1) * P'$;
6 **while** $||\pi(t) - \pi(t-1)|| > conv$;

En cuanto al vector *PageRank* π, este debe ser inicializado antes de comenzar el bucle de iteracciones. La única restriccione que se pide es que la suma de las posiciones del mismo sea 1, es decir, que la norma 1 del sea igual a 1 ($||\pi||_1 = \sum_i \pi_i = 1$) para mantener la propiedad de distribucción de probabilidad. Existen distintas heurísticas destinadas a reducir el número de iteraciones del algoritmo mediante la inicialización de este vector, sin embargo, en este caso se ha preferido inicializarlo siguiendo una distribucción uniforme. Esta inicialización no determina el resultado, tan solo el tiempo de convergencia hacia este.

Tal y como se puede apreciar, el bucle de iteraciones consiste únicamente en la multiplicación del vector *PageRank* por la matriz P' junto con la actualización del índice referido a la iteracción. El siguiente punto a discutir es el criterio de convergencia del resultado. Para ello existen distintas normas vectoriales, entre ellas la norma uno (descrita en el párrafo anterior) o la norma infinito ($||\pi||_\infty = max_i\{\pi_i\}$). En este caso se ha creido más conveniente la utilización de la *norma 1* para el criterio de convergencia puesto que se pretenden reducir todos los valores del vector y no únicamente el máximo de todos ellos, por lo que de esta manera se consigue una aproximación más homogenea respecto del *PageRank Exacto*.

5.4.3. Estrategia Basada en Paseos Aleatorios

También existe una estrategia basada en paseos aleatorios para el cálculo del vector *PageRank*, la cual sigue una estrategia muy diferente de las descritas en anteriores secciones. En este caso la idea principal se basa en realizar paseos aleatorios sobre los vértices del grafo siguiendo la distribución de probabilidades descrita a partir de la matriz de transiciones modificada P' para despues estimar el vector *PageRank* a partir del número de veces que cada vértice ha aparecido en los paseos aleatorios.

La dificultad algoritmica en este caso se refiere a la generación de valores aleatorios a partir de una distribución ponderada (obtenida de la matriz de transiciones P') ya que es necesario mantener n generadores de valores aleatorios que indiquen el siguiente vértice a visitar en cada caso. Obviando esta dificultad, el resto

del algoritmo es bastante simple y se basa en el mantenimiento del ratio de visitas sobre cada vértice conforme el paseo aleatorio aumenta su longitud. Como en el caso iterativo, también se basa en la repetición de la misma operación hasta llegar a un criterio de convergencia (idéntico a la solución iterativa).

La figura correspondiente al Algoritmo 6, que se basa en la actualización del vector *PageRank* conforme se obtiene el resultado del próximo vértice a visitar. De esta manera en cada iteración se realizan n paseos aleatorios de longitud 1 que se combinan para llegar al estado estacionario. La dificultad conceptual en este caso se refiere tanto a la generación de números aleatorios ponderados y el mantenimiento de una media incremental normalizada sobre el vector *PageRank* π. Tal y como se ha indicado en el párrafo anterior, el criterio de convergencia es equivalente al caso iterativo.

Algorithm 6: Random Walks PageRank

Result: $\pi(t)$
$t \leftarrow 0$;
$\pi(t) \leftarrow \frac{1}{n} * \mathbf{1}$;
do
  $t \leftarrow t + 1$;
  $\pi_u(t) \leftarrow \frac{t}{t+n} * \pi_u(t)$;
  for *cada* $v \in V$ **do**
    $u \leftarrow randomWeighted(v)$;
    $\pi_u(t) \leftarrow \pi_u(t) + \frac{1}{t+n}$;
  end
while $||\pi(t) - \pi(t-1)|| > conv$;

Las estrategias descritas en este capítulo para la obtención de la distribución estacionaria π sobre la matriz de transiciones modificada P' del grafo G se han hecho desde una perspectiva básica. Existen gran cantidad de trabajos sobre disintas variaciones de estos métodos para tratar de reducir el tiempo de convergencia.

5.5. PageRank Personalizado

Tal y como se ha indicado a lo largo del capítulo, el algoritmo *PageRank* genera un ranking referido al ratio de importancia de cada vértice perteneciente a un determinado grafo dirigido no ponderado G. Dicha medida para el vértice v se calcula como la probabilidad de que tras un paseo aleatorio por los vértices del grafo siguiendo las aristas del mismo, el vértice final del camino sea v.

Para hacer frente a aquellos casos en los cuales se llega a un vértice que no contiene aristas (*vértice sumidero*) sobre las cuales acceder a otros vértices entonces es necesario realizar un salto hacia otro vértice del grafo sin tener en cuenta las aristas. Tal y como se ha indicado, este proceso se lleva a cabo tanto en los *vértices sumidero*, para los cuales es necesario para poder calcular el ranking, como en el resto de vértices con probabilidad $1 - \beta$.

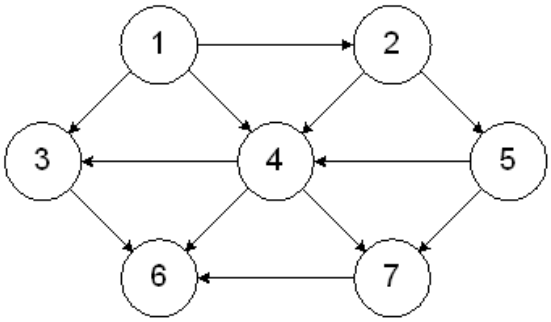


Figura 5.2: Ejemplo de *Grafo Dirigido No Ponderado.* (Extraido de [Fre10])

En esta sección se habla de la distribución de probabilidad que sigue el proceso de *saltar* de un vértice a otro. Dicha distribución se codifica mediante el vector $p = [p_1, p_2, ..., p_i, ..., p_n]$, cuya única restricción se refiere a la suma de valores, que debe ser 1, es decir, p debe estar normalizado ($||p||_1 = 1$).

El caso básico se refiere a la distribucción uniforme, para la cual todas las posiciones del vector toman el mismo valor, es decir, $\forall i \in [1, n],\ p_i = \frac{1}{n}$. De esta manera se modeliza la situación en la cual cuando se lleva a cabo un salto, todos los vértices tienen la misma probabilidad de ser el vértice de destino. Nótese por tanto, que en este caso no existe no existe ningún grado de personalización sobre el resultado, por lo que se conoce como *PageRank General* ya que aporta una estimación acerca de la importancia de los vértices del grafo desde una perspectiva generalizada, lo cual es interesante desde el punto de vista de la obtención de analíticas del grafo.

La situación descrita en el párrafo anterior, sin embargo, no se asemeja al comportamiento real de un usuario que navega por la red. La razón se debe a que cuando alguien accede directamente a un sitio web escribiendo la url del mismo en su navegador favorito, este no selecciona una al azar de entre los millones de páginas posibles, sino que la escoge de entre un reducido subconjunto de ellas, que además no es uniforme, dado que los usuarios no visitan las páginas web el mismo número de veces.

A través de dicha idea, el resultado del algoritmo *PageRank* cambia y en lugar de generar un ranking general del grafo G, ahora realiza un ranking sesgado otorgando una mayor puntuación a los vértices cercanos a los vértices de destino de los saltos, lo cual altera en gran medida el resultado anterior.

Para comprender esta idea se ha incluido en la figura 5.2 un ejemplo de *grafo dirigido no ponderado.* Supongase que se calcula el *PageRank* seleccionando el vector p de tal manera que $p_3 = 1$ y $\forall i \in [1, 7] - [3],\ p_i = 0$. Mediante esta situación se consigue que todos los saltos (los referidos a vértices sumidero y los relizados con probabilidad $1 - \beta$) tengan como vértice destino el 3. Por esta razón con total seguridad el vértice 3 tendrá la mejor puntuación *PageRank* del grafo. Además, la puntuación del resto de vértices también se habrá modificado, con altas probabilidades de que el vértice 6 se encuentre en segunda posición.

A través de la modificación de la distribución de probabilidad del vector p se consigue por tanto la adaptación del ranking *PageRank* desde una perspectiva de localidad, lo cual se convierte en una clasificación mucho más interesante desde el punto de vista del usuario, que en el caso del grafo de la web, conoce de manera mucho más fiel las páginas web que son importantes desde su perspectiva individual.

Sin embargo, la personalización del ranking *PageRank* conlleva distintos problemas a nivel computacional. Esto se debe a que en este caso en lugar de tener que mantener un único ranking para todo el grafo es necesario mantener un ranking para cada usuario que pretenda consultar el pagerank del grafo. Este problema no es tan grave como parede puesto que el vector de la distribución estacionaria π posee la propiedad de linealidad, lo cual simplifica dicha problematica, reduciendo el problema a la necesidad de mantener un vector π para cada vértice del grafo, es decir, hay que generar n vectores *PageRank*.

Ahora supongamos que un determinado usuario desea calcular su propio ranking personalizado. La tarea se reduce a realizar una media ponderada sobre los vectores π_i escogiendo los pesos según el vector p de personalización para dicho usuario. A pesar de ello, esta tarea continua siendo compleja, tanto a nivel temporal como espacial, puesto que se incrementa el coste respecto del *PageRank General* en un orden de n.

Debido al tamaño de grafos como el de la web (*Web Graph*), cuyo número de vértices es tan elevado que no es posible mantener un vector *PageRank* para cada uno de ellos en memoria. Esta razón fificulta la tarea de calcular el ranking personalizado para un determinado usuario en tiempo de consulta. Por tanto, se ha se han propuesto distintas soluciones para este problema.

Una de las primeras es la descrita en [Hav02], que se basa en el mantenimiento de *16* vectores *PageRank* de referidos a temas distintos entre si, que despues se combinan en tiempo de consulta para generar un ranking personalizado. En [KHMG03] los autores proponen el cálculo de vectores *PageRank* teniendo en cuenta la estructura de bloques que se genera entre vértices cuyo ranking es similar. Esto se lleva a cabo a partir de un algoritmo de 3 fases (búsqueda de vértices similares, cálculo del vector *PageRank* para los conjuntos resultantes y, en tiempo de consulta, combinación de los mismos para obtener el ranking personalizado).

En [JW03] se explica una técnica que permite calcular el vector *PageRank* personalizado utilizando una técnica incrementa o jerarquica basada en *Programación Dinámica*, lo cual permite hacer calcular el ranking basado en la combinación de cien mil vectores en tiempo de consulta. En [SBC+06] se propone una solución semejante en un espacio menor mediante la utilización del *Count-Min Sketch* (Sección 3.4).En el trabajo *Estimating pagerank on graph streams* [SGP11] *Sarma y otros* proponen un algoritmo basado en la generación de paseos aleatorios para estimar el ranking *PageRank* sobre las restricciones que impone el *modelo en semi-streaming*.

5.6. Alternativas a PageRank

El algoritmo *PageRank* se ha convertido en la alternativa más usada para el cálculo del ránking de vértices sobre un grafo basandose únicamente en la estructura del mismo (relaciones a partir de aristas). Su popularidad se debe en gran medida tanto a su sencillez desde el punto de vista algorítmico (a pesar de que su coste computacional es elevado), como a la gran fama del motor de búsquedas *Google*, que desde sus comienzos otorgaba resultados muy buenos apoyandose en la utilización de dicho ranking.

Sin embargo, *PageRank* no es la única alternativa sobre la que se ha trabajado en el ámbito del cálculo de importancia para los vértices de grafos. En esta sección se describen brevemente distintos algoritmos cuya finalidad es semejante. Además se hablará de una extensión del *PageRank* conocida como *SimRank* que obtiene el grado de similitud entre los vértices del grafo desde un punto de vista estructural. En la

sección 5.6.1 se describe *HITS*, posteriormente en la sección 5.6.2 se describe *SALSA* (una mejora respecto del anterior), y finalmente, en la sección 5.6.3 se hablará de *SimRank*.

5.6.1. HITS

El algoritmo *HITS* surge en paralelo junto con *PageRank*, puesto que los artículos en los que se habla de dichos algoritmos fueron publicados en *1999*. *HITS* fue descrito por primera vez en el trabajo *Authoritative sources in a hyperlinked environment* [Kle99] de *Kleinberg*.

Este algoritmo, a diferencia del *PageRank*, se basa en la generación del ranking en tiempo de consulta. Para ello utiliza un subconjunto de vértices inicial obtenido a partir del ranking del ranking de similitud basada en texto respecto de la consulta en cuestión. A partir del subgrafo generado por este subconjunto. Las iteraciones del algoritmo se basan la generación de dos puntuaciones para cada vértice. Estas puntuaciones se conocen como *Autority score* y *Hub score*. Las cuales se construyen como la suma de *Hub scores* de los vértices que apuntan hacia el vértice para el caso del *Autority score* y la suma de *Autority scores* de los vértices hacia los que apunta el vértice para el *Hub score*. Dichos valores son normalizados en cada iteración. Este proceso se repite hasta llegar a un determinado grado de convergencia.

Las diferencias respecto del *PageRank* por tanto se basan en la generación del ranking en tiempo de consulta en lugar de en tiempo de indexación o estático, por tanto este ranking varía respecto de cada búsqueda. En este caso, en lugar de obtener un único ranking se obtienen dos, indicando los vértices más importantes y indicando los vértices que más importancia generan. Además, en lugar de generar el ranking sobre el grafo completo, *HITS* lo lleva a cabo sobre un subgrafo del mismo (obtenido mediante búsqueda textual tal y como se ha indicado anteriormente).

5.6.2. SALSA

El algoritmo *SALSA* se refiere a una combinación de *PageRank* y *HITS* (descrito en la anterior sección), por tanto, tiene características semejantes de ambas alternativas. La descripción del mismo fue llevada a cabo en el documento *SALSA: the stochastic approach for link-structure analysis*[LM01] de *Lempel y Moran*.

Debido a que este algoritmo consiste en una combinación de los citados previamente, a continuación se indican las semejanzas que tiene respecto de estos. *SALSA* se basa en la misma idea que *HITS*, en el sentido de que genera dos ranking, referidos a autoridades y generadores de autoridad (*Autority* y *Hub*). Además, también realiza dicho ranking en tiempo de consulta utilizando un subgrafo generado a partir del ranking de similitud textual. La diferencia respecto de *HITS* se basa en la estrategia utilizada para calcular dichos ranking. En este caso utiliza el mismo enfoque de *paseos aleatorios* del *PageRank* (por eso se dice que es una combinación de ambos).

La generación de rankings a través de paseos aleatorios le otorgo una ventaja significativa a nivel de coste computacional respecto de *HITS*, además de ofrecer resultados en tiempo de consulta, lo cual le diferencia de *PageRank*. En el documento *WTF: The Who to Follow Service at Twitter* [GGL$^+$13] los autores describen cómo la red social *Twitter* ha desarrollado una variación de este algoritmo para su sistema de recomendación de usuarios a los que comenzar a seguir.

5.6.3. SimRank

El algoritmo *SimRank* tiene un propósito distinto respecto de los descritos anteriormente. En este caso, en lugar de tratar de conocer el grado de importancia de un determinado vértice del grafo, lo que se pretende es obtener un ranking de simitud del resto de vértices del grafo respecto de uno concreto. En el trabajo *SimRank: a measure of structural-context similarity* [JW02] de *Jeh y Widom* se describe de manera el completa el modo de funcionamiento del algoritmo, así como la demostración acerca de la corrección del mismo. A continuación se realiza una breve descripción acerca de dicho modo de funcionamiento.

Al igual que el *PageRank*, en este caso el algoritmo también se basa en el cálculo iterativo del ranking hasta llegar a un determinado índice de convergencia. La primera iteración se basa en el cálculo de un *PageRank Personalizado* desde el vértice sobre el cual se pretende basar la comparación. Tras esta inicialización, el resto del algoritmo se basa en la repetición de la ecuación (5.6) hasta llegar a un índice de convergencia.

$$Sim_{u_1}^{(k)}(u_2) = \begin{cases} (1-c) * \frac{\sum_{\{(u_1,v_1),(u_2,v_2)\} \in E} Sim_{v_1}^{(k-1)}(v_2)}{d^-(u_1) * d^-(u_2)}, & \text{if } u_1 \neq u_2 \\ 1, & \text{if } u_1 = u_2 \end{cases} \tag{5.6}$$

La ecuación (5.6) calcula por tanto el grado de similitud del vértice u_2 respecto del vértice u_1 en la iteración k, lo cual se puede denotar como $Sim_{u_1}^{(k)}(u_2)$. Tal y como se ha indicado anteriormente, dicho valor se incrementa conforme los vértices u_1 y u_2 se relacionan con conjuntos similares de vértices desde el punto de vista de que estos sean los mismos. Esto no debe confuncirse con otros problemas como el de *Matchings* (referido a encontrar subestructuras con la misma forma dentro de un grafo).

Existen numerosas aplicaciones prácticas para este algoritmo como sistema de recomendaciones sobre conjuntos de datos con estructura de grafo. Algunos ejemplos donde podría ser utilizado es en problemas como la generación de anuncios de sitios web en un buscador, la generación de listas de reproducción de video a partir de un determinado video teniendo en cuenta la navegación que los usuarios han llevado a cabo previamente entre estos, o un sistema de recomendación de compras en una tienda virtual basandose en la misma idea.

5.7. Conclusiones

Tal y como se ha visto a lo largo del capítulo, el cálculo de importancia sobre los vértices grafos es una tarea compleja, cuya dificultad se incrementa en gran medida cuando el tamaño del grafo es de tamaño masivo, lo cual dificulta la tarea de contenerlo de manera completa en memoria. Sin embargo, en este capítulo no se han discutido soluciones para dicho problema, sino que se ha realizado un estudio acerca del algoritmo *PageRank*, el cual goza de gran popularidad gracias a motor de búsquedas *Google*.

A partir de la descripción de este algoritmo se ha podido comprender la perspectiva algebraica para la resolución de un problema sobre grafos, que puede entenderse como uno problema de matrices. Además, se ha ilustrado la cercania de este ranking respecto de las *Cadenas de Markov* y los paseos aleatorios, que convergen hacia la *distribución estacionaria*. Después se ha indicado cómo calcularla a partir de distintas estrategias (*Algebraica*, *Iterativa* o *basada en paseos aleatorios*)

Posteriormente también se habló del *PageRank Personalizado* y la problematica referida a la cantidad de rankings que sería necesario mantener para llevar a cabo una solución exacta para dicho problema. Por último se ha hablado de estrategias similares para el ranking de vértices así como el algoritmo *SimRank*, que indica el grado de similitud de vértices respecto de un determinado vértice.

Gracias a este algoritmo se puede obtener un indicador acerca de la importancia de los vértices de un determinado grafo, lo cual es una tarea interesante y aplicable sobre un gran conjunto de ámbitos para los cuales el problema puede ser modelizado como una red. La observación acerca de la importancia de los vértices desde el punto de vista estructural es un factor interesante que puede mejorar la toma de desisiones en un gran número de casos.

Capítulo 6

Implementación, Resultados y Trabajo Futuro

6.1. Introducción

[TODO]

6.2. Implementación

[TODO]

6.3. Resultados

[TODO]

6.4. Trabajo Futuro

[TODO]

6.5. Conclusiones

[TODO]

Apéndice A

Metodología de Trabajo

A.1. Introducción

[TODO]

A.2. Contexto

[TODO]

A.3. Trabajo de Investigación

[TODO]

A.4. Trabajo Disperso

[TODO]

A.5. Conclusiones

[TODO]

Apéndice B

Código Fuente

B.1. `main class`

```python
import tensorflow as tf
import numpy as np
import timeit

import tfg_big_data_algorithms as tf_G


def main():
    beta: float = 0.85
    convergence: float = 0.01

    wiki_vote_np: np.ndarray = tf_G.DataSets.wiki_vote()
    followers_np: np.ndarray = tf_G.DataSets.followers()

    with tf.Session() as sess:
        writer: tf.summary.FileWriter = tf.summary.FileWriter(
            'logs/tensorflow/.')

        graph: tf_G.Graph = tf_G.GraphConstructor.from_edges(sess,
                                                             "Gfollowers",
                                                             followers_np,
                                                             writer,
                                                             is_sparse=False)
        '''
        pr_alge: tf_G.PageRank = tf_G.NumericAlgebraicPageRank(sess, "PR1",
                                                    graph,
                                                    beta)
        '''
        pr_iter: tf_G.PageRank = tf_G.NumericIterativePageRank(sess,
                                                               "PR1",
                                                               graph,
                                                               beta)
        '''
        pr_random: tf_G.PageRank = tf_G.NumericRandomWalkPageRank(sess, "PR3",
                                                    graph,
                                                    beta)
        '''
        g_updateable: tf_G.Graph = tf_G.GraphConstructor.empty(sess,
                                                               "Gfollowers",
                                                               7, writer)
        pr_updateable: tf_G.PageRank = tf_G.NumericIterativePageRank(sess,
                                                                     "PRfollowers",
                                                                     g_updateable,
                                                                     beta)

        '''
        for r in followers_np:
            start_time = timeit.default_timer()
            g_updateable.append(r[0],r[1])
            print(timeit.default_timer() - start_time)
            print()
            writer.add_graph(sess.graph)
        g_updateable.remove(followers_np[0,0], followers_np[0,1])
        g_updateable.append(followers_np[0,0], followers_np[0,1])
        '''
```

Figura B.1: `main.py` Parte (a)

B.2. `graph` package

```python
import tensorflow as tf
import numpy as np

from tfg_big_data_algorithms.utils.tensorflow_object import TensorFlowObject, \
    TF_type
from tfg_big_data_algorithms.utils.update_edge_notifier import \
    UpdateEdgeNotifier


class Graph(TensorFlowObject, UpdateEdgeNotifier):
    def __init__(self, sess: tf.Session, name: str,
                 writer: tf.summary.FileWriter = None,
                 edges_np: np.ndarray = None, n: int = None,
                 is_sparse: bool = False) -> None:
        TensorFlowObject.__init__(self, sess, name, writer, is_sparse)
        UpdateEdgeNotifier.__init__(self)

        if edges_np is not None:
            self.n = int(edges_np.max(axis=0).max() + 1)
            self.m = int(edges_np.shape[0])
            A_init = tf.scatter_nd(edges_np.tolist(), self.m * [1.0],
                                   [self.n, self.n])
        elif n is not None:
            self.n = n
            self.m = 0
            A_init = tf.zeros([self.n, self.n])
        else:
            raise ValueError('Graph constructor must be have edges or n')

        self.n_tf: TF_type = tf.Variable(float(self.n), tf.float32,
                                         name=self.name + "_n")
        self.A_tf: TF_type = tf.Variable(A_init, tf.float64,
                                         name=self.name + "_A")
        self.out_degrees_tf: TF_type = tf.Variable(
            tf.reduce_sum(self.A_tf, 1, keep_dims=True),
            name=self.name + "_d_out")
        self.in_degrees_tf: TF_type = tf.Variable(
            tf.reduce_sum(self.A_tf, 0, keep_dims=True),
            name=self.name + "_d_in")
        self.L_tf: TF_type = tf.Variable(
            tf.diag(self.get_out_degrees_tf()) - self.A_tf,
            name=self.name + "_L")
        self.run(tf.variables_initializer([self.A_tf, self.n_tf]))
        self.run(tf.variables_initializer([
            self.out_degrees_tf, self.in_degrees_tf]))
        self.run(tf.variables_initializer([self.L_tf]))

    @property
    def is_not_sink_tf(self) -> tf.Tensor:
        return tf.not_equal(self.get_out_degrees_tf(), 0)

    @property
    def out_degrees_np(self) -> np.ndarray:
        return self.run(self.get_out_degrees_tf())

```

Figura B.2: `graph.py` Parte (a)

```
56      @property
57      def out_degrees_np(self) -> np.ndarray:
58          return self.run(self.out_degrees_tf)
59
60      @property
61      def edge_list_np(self) -> np.ndarray:
62          return self.run(self.edge_list_tf)
63
64      @property
65      def edge_list_tf(self) -> tf.Tensor:
66          return tf.cast(tf.where(tf.not_equal(self.A_tf, 0)), tf.int64)
67
68      @property
69      def L_pseudo_inverse_tf(self) -> tf.Tensor:
70          return tf.py_func(np.linalg.pinv, [self.L_tf], tf.float32)
71
72      def A_tf_vertex(self, vertex: int) -> tf.Tensor:
73          return tf.gather(self.A_tf, [vertex])
74
75      def out_degrees_tf_vertex(self, vertex: int) -> tf.Tensor:
76          return tf.gather(self.out_degrees_tf, [vertex])
77
78      def is_not_sink_tf_vertex(self, vertex: int) -> TF_type:
79          return tf.not_equal(
80              tf.reshape([self.out_degrees_tf_vertex(vertex)], [1]), 0)
81
82      def get_in_degrees_tf(self, keep_dims=False) -> TF_type:
83          if keep_dims is False:
84              return tf.reshape(self.in_degrees_tf, [self.n])
85          else:
86              return self.in_degrees_tf
87
88      def get_out_degrees_tf(self, keep_dims=False) -> TF_type:
89          if keep_dims is False:
90              return tf.reshape(self.out_degrees_tf, [self.n])
91          else:
92              return self.out_degrees_tf
93
94      def __str__(self) -> str:
95          return str(self.run(self.L_tf))
96
97      def append(self, src: int, dst: int) -> None:
98          if src and dst is None:
99              raise ValueError(
100                 "tfg_big_data_algorithms and dst must not be None ")
101         self.run([tf.scatter_nd_add(self.A_tf, [[src, dst]], [1.0]),
102                   tf.scatter_nd_add(self.out_degrees_tf, [[src, 0]], [1.0]),
103                   tf.scatter_nd_add(self.in_degrees_tf, [[0, dst]], [1.0]),
104                   tf.scatter_nd_add(self.L_tf, [[src, src], [src, dst]],
105                                     [+1.0, -1.0])])
106         self.m += 1
107         self._notify([src, dst], 1)
108
109     def remove(self, src: int, dst: int) -> None:
110         if src and dst is None:
111             raise ValueError(
112                 "tfg_big_data_algorithms and dst must not be None ")
113         self.run([tf.scatter_nd_add(self.A_tf, [[src, dst]], [-1.0]),
114                   tf.scatter_nd_add(self.out_degrees_tf, [[src, 0]], [-1.0]),
115                   tf.scatter_nd_add(self.in_degrees_tf, [[0, dst]], [-1.0]),
116                   tf.scatter_nd_add(self.L_tf, [[src, src], [src, dst]],
117                                     [-1.0, +1.0])])
118         self.m -= 1
119         self._notify([src, dst], -1)
```

Figura B.3: `graph.py` Parte (b)

```python
import tensorflow as tf
import numpy as np

from tfg_big_data_algorithms.graph.graph import Graph
from tfg_big_data_algorithms.graph.graph_sparsifier import GraphSparsifier


class GraphConstructor:
    @staticmethod
    def from_edges(sess: tf.Session, name: str, edges_np: np.ndarray,
                   writer: tf.summary.FileWriter = None,
                   is_sparse: bool = False) -> Graph:
        return Graph(sess, name, edges_np=edges_np, writer=writer,
                     is_sparse=is_sparse)

    @staticmethod
    def empty(sess: tf.Session, name: str, n: int,
              writer: tf.summary.FileWriter = None,
              sparse: bool = False) -> Graph:
        return Graph(sess, name, n=n, writer=writer, is_sparse=sparse)

    @staticmethod
    def unweighted_random(sess: tf.Session, name: str, n: int, m: int,
                          writer: tf.summary.FileWriter = None,
                          is_sparse: bool = False) -> Graph:
        if m > n * (n - 1):
            raise ValueError('m would be less than n * (n - 1)')
        edges_np = np.random.random_integers(0, n - 1, [m, 2])

        cond = True
        while cond:
            # remove uniques from: https://stackoverflow.com/a/16973510/3921457
            edges_np = np.concatenate((edges_np,
                                       np.random.random_integers(0, n - 1, [
                                           m - len(edges_np), 2])), axis=0)
            _, unique_idx = np.unique(np.ascontiguousarray(edges_np).view(
                np.dtype(
                    (np.void, edges_np.dtype.itemsize * edges_np.shape[1]))),
                return_index=True)
            edges_np = edges_np[unique_idx]
            edges_np = edges_np[edges_np[:, 0] != edges_np[:, 1]]
            cond = len(edges_np) != m

        return Graph(sess, name, edges_np=edges_np, writer=writer,
                     is_sparse=is_sparse)

    @staticmethod
    def as_naive_sparsifier(sess: tf.Session, graph: Graph, p: float,
                            is_sparse: bool = False) -> Graph:
        boolean_distribution = tf.less_equal(
            tf.random_uniform([graph.m], 0.0, 1.0), p)
        edges_np = graph.edge_list_np[sess.run(boolean_distribution)]
        return Graph(sess, graph.name + "_sparsifier",
                     edges_np=edges_np, is_sparse=is_sparse)

    @classmethod
    def as_other_sparsifier(cls, sess, graph, p, is_sparse=False):
        return GraphSparsifier(sess, graph, p, is_sparse)
```

Figura B.4: graph_constructor.py

B.3. `pagerank` package

```python
import warnings

import tensorflow as tf
import numpy as np

from typing import List
from tfg_big_data_algorithms.utils.tensorflow_object import TensorFlowObject
from tfg_big_data_algorithms.utils.vector_convergence import \
    ConvergenceCriterion


class PageRank(TensorFlowObject):
    def __init__(self, sess: tf.Session, name: str) -> None:
        TensorFlowObject.__init__(self, sess, name)

    def error_vector_compare_tf(self, other_pr: 'PageRank',
                                k: int = -1) -> tf.Tensor:
        raise NotImplementedError(
            'subclasses must override compare()!')

    def error_vector_compare_np(self, other_pr: 'PageRank',
                                k: int = -1) -> np.ndarray:
        return self.run(self.error_vector_compare_tf(other_pr, k))

    def error_ranks_compare_tf(self, other_pr: 'PageRank',
                               k: int = -1) -> np.ndarray:
        raise NotImplementedError(
            'subclasses must override compare()!')

    def error_ranks_compare_np(self, other_pr: 'PageRank',
                               k: int = -1) -> np.ndarray:
        return self.run(self.error_ranks_compare_tf(other_pr, k=k))

    def pagerank_vector_np(self, convergence: float = 1.0, steps: int = 0,
                           topics: List[int] = None,
                           c_criterion=ConvergenceCriterion.ONE) -> np.ndarray:
        return self.run(
            self.pagerank_vector_tf(convergence, steps, topics,
                                    c_criterion))

    def pagerank_vector_tf(self, convergence: float = 1.0, steps: int = 0,
                           topics: List[int] = None,
                           c_criterion=ConvergenceCriterion.ONE) -> tf.Tensor:
        if 0.0 < convergence < 1.0:
            return self._pr_convergence_tf(convergence, topics=topics,
                                           c_criterion=c_criterion)
        elif steps > 0:
            return self._pr_steps_tf(steps, topics=topics)
        else:
            return self._pr_exact_tf(topics=topics)

    def ranks_np(self, convergence: float = 1.0, steps: int = 0,
                 topics: List[int] = None) -> np.ndarray:
        raise NotImplementedError(
            'subclasses must override ranks_by_rank()!')

    def _pr_convergence_tf(self, convergence: float, topics: List[int],
                           c_criterion) -> tf.Tensor:
        raise NotImplementedError(
            'subclasses must override page_rank_until_convergence()!')

    def _pr_steps_tf(self, steps: int, topics: List[int]) -> tf.Tensor:
        raise NotImplementedError(
            'subclasses must override page_rank_until_steps()!')

    def _pr_exact_tf(self, topics: List[int]) -> tf.Tensor:
        raise NotImplementedError(
            'subclasses must override page_rank_exact()!')
```

```python
import warnings
from typing import List

import numpy as np
import tensorflow as tf

from tfg_big_data_algorithms.pagerank.transition.transition import Transition
from tfg_big_data_algorithms.graph.graph import Graph
from tfg_big_data_algorithms.pagerank.pagerank import PageRank
from tfg_big_data_algorithms.utils.utils import Utils
from tfg_big_data_algorithms.utils.vector_norm import VectorNorm


class NumericPageRank(PageRank):
    def __init__(self, sess: tf.Session, name: str, graph: Graph, beta: float,
                 T: Transition) -> None:
        PageRank.__init__(self, sess, name)
        self.G = graph

        self.beta = beta
        self.T = T
        self.T.attach(self)
        self.v = tf.Variable(tf.fill([1, self.G.n], tf.pow(self.G.n_tf, -1)),
                             name=self.G.name + "_" + self.name + "_v")
        self.run(tf.variables_initializer([self.v]))

    def ranks_np(self, convergence: float = 1.0, steps: int = 0,
                 topics: List[int] = None) -> np.ndarray:
        self.pagerank_vector_tf(convergence, steps, topics)
        ranks = tf.map_fn(
            lambda x: [x, tf.gather(tf.reshape(self.v, [self.G.n]), x)],
            tf.transpose(
                tf.py_func(Utils.ranked,
                           [tf.scalar_mul(-1, self.v)], tf.int64)),
            dtype=[tf.int64, tf.float32])
        return np.concatenate(self.run(ranks), axis=1)

    def error_vector_compare_tf(self, other_pr: 'NumericPageRank',
                                k: int = -1) -> tf.Tensor:
        if 0 < k < self.G.n - 1:
            if 0 < k < self.G.n - 1:
                warnings.warn('k-best error comparison not implemented yed')

        return tf.reshape(
            VectorNorm.ONE(tf.subtract(self.v, other_pr.v)), [])

    def error_ranks_compare_tf(self, other_pr: 'NumericPageRank',
                               k: int = -1) -> tf.Tensor:
        if 0 < k < self.G.n - 1:
            warnings.warn('k-best error comparison not implemented yed')

        return tf.div(tf.cast(tf.reduce_sum(tf.abs(
            tf.py_func(Utils.ranked, [
                tf.py_func(Utils.ranked, [tf.scalar_mul(-1, self.v)],
                           tf.int64)], tf.int64) -
            tf.py_func(Utils.ranked, [
                tf.py_func(Utils.ranked, [tf.scalar_mul(-1, other_pr.v)],
                           tf.int64)], tf.int64))), tf.float32),
            (self.G.n_tf * (self.G.n_tf - 1)))
```

Figura B.6: numeric_pagerank.py

```python
import warnings
from typing import List

import tensorflow as tf

from tfg_big_data_algorithms.pagerank.transition.transition_matrix import \
    TransitionMatrix
from tfg_big_data_algorithms.pagerank.numeric_pagerank import NumericPageRank
from tfg_big_data_algorithms.graph.graph import Graph


class NumericAlgebraicPageRank(NumericPageRank):
    def __init__(self, sess: tf.Session, name: str, graph: Graph,
                 beta: float) -> None:
        T = TransitionMatrix(sess, name + "_alge", graph)
        NumericPageRank.__init__(self, sess, name, graph, beta, T)

    def _pr_exact_tf(self, topics: List[int] = None) -> tf.Tensor:
        if topics is not None:
            warnings.warn('Personalized PageRank not implemented yet!')
        a = tf.fill([1, self.G.n], (1 - self.beta) / self.G.n_tf)
        b = tf.matrix_inverse(
            tf.eye(self.G.n, self.G.n) - self.beta * self.T())
        self.run(self.v.assign(tf.matmul(a, b)))
        return self.v

    def _pr_convergence_tf(self, convergence: float, topics: List[int] = None,
                           c_criterion=None) -> tf.Tensor:
        if topics is not None:
            warnings.warn('Personalized PageRank not implemented yet!')
        warnings.warn('NumericPageRank not implements iterative PageRank! ' +
                      'Using exact algorithm.')
        return self._pr_exact_tf(topics)

    def _pr_steps_tf(self, steps: int, topics: List[int]) -> tf.Tensor:
        if topics is not None:
            warnings.warn('Personalized PageRank not implemented yet!')
        warnings.warn('NumericPageRank not implements iterative PageRank! ' +
                      'Using exact algorithm.')
        return self._pr_exact_tf(topics)
```

Figura B.7: numeric_algebraic_pagerank.py

```python
import warnings
from typing import List

import tensorflow as tf
import numpy as np

from tfg_big_data_algorithms.pagerank.transition.transition_reset_matrix import \
    TransitionResetMatrix
from tfg_big_data_algorithms.pagerank.numeric_pagerank import NumericPageRank
from tfg_big_data_algorithms.utils.vector_convergence import \
    ConvergenceCriterion
from tfg_big_data_algorithms.graph.graph import Graph


class NumericIterativePageRank(NumericPageRank):
    def __init__(self, sess: tf.Session, name: str, graph: Graph,
                 beta: float) -> None:
        T = TransitionResetMatrix(sess, name + "_iter", graph, beta)
        NumericPageRank.__init__(self, sess, name + "_iter", graph, beta, T)
        self.iter = lambda i, a, b=self.T(): tf.matmul(a, b)

    def _pr_convergence_tf(self, convergence: float, topics: List[int] = None,
                           c_criterion=ConvergenceCriterion.ONE) -> tf.Tensor:
        if topics is not None:
            warnings.warn('Personalized PageRank not implemented yet!')

        self.run(
            self.v.assign(
                tf.while_loop(c_criterion,
                              lambda i, v, v_last, c, n:
                              (i + 1, self.iter(i, v), v, c, n),
                              [0.0, self.v, tf.zeros([1, self.G.n]),
                               convergence,
                               self.G.n_tf], name=self.name + "_while_conv")[
                    1]))
        return self.v

    def _pr_steps_tf(self, steps: int, topics: List[int]) -> tf.Tensor:
        if topics is not None:
            warnings.warn('Personalized PageRank not implemented yet!')

        self.run(
            self.v.assign(
                tf.while_loop(lambda i, v: i < steps,
                              lambda i, v: (i + 1.0, self.iter(i, v)),
                              [0.0, self.v], name=self.name + "_while_steps")[
                    1]))
        return self.v

    def _pr_exact_tf(self, topics: List[int]) -> None:
        if topics is not None:
            warnings.warn('Personalized PageRank not implemented yet!')

        raise NotImplementedError(
            str(self.__class__.__name__) + ' not implements exact PageRank')

    def update_edge(self, edge: np.ndarray, change: float) -> None:
        warnings.warn('PageRank auto-update not implemented yet!')

        print("Edge: " + str(edge) + "\tChange: " + str(change))
        self.run(self._pr_convergence_tf(convergence=0.01))
```

Figura B.8: `numeric_iterative_pagerank.py`

```python
import tensorflow as tf
import numpy as np

from tfg_big_data_algorithms.pagerank.transition.transition import Transition
from tfg_big_data_algorithms.graph.graph import Graph


class TransitionMatrix(Transition):
    def __init__(self, sess: tf.Session, name: str, graph: Graph) -> None:
        Transition.__init__(self, sess, name, graph)

        self.transition = tf.Variable(
            tf.where(self.G.is_not_sink_tf,
                     tf.div(self.G.A_tf, self.G.out_degrees_tf),
                     tf.fill([self.G.n, self.G.n], 1 / self.G.n)),
            name=self.name)
        self.run(tf.variables_initializer([self.transition]))

    def __call__(self, *args, **kwargs):
        return self.transition

    def update_edge(self, edge: np.ndarray, change: float) -> None:

        # print("Edge: " + str(edge) + "\tChange: " + str(change))

        if change > 0.0:
            self.run(tf.scatter_nd_update(
                self.transition, [[edge[0]]],
                tf.div(self.G.A_tf_vertex(edge[0]),
                       self.G.out_degrees_tf_vertex(edge[0]))))
        else:
            self.run(tf.scatter_nd_update(
                self.transition, [[edge[0]]],
                tf.where(self.G.is_not_sink_tf_vertex(edge[0]),
                         tf.div(self.G.A_tf_vertex(edge[0]),
                                self.G.out_degrees_tf_vertex(edge[0])),
                         tf.fill([1, self.G.n], tf.pow(self.G.n_tf, -1)))))
        self._notify(edge, change)
```

Figura B.9: `transition_matrix.py`

```python
import tensorflow as tf
import numpy as np

from tfg_big_data_algorithms.pagerank.transition.transition import Transition
from tfg_big_data_algorithms.graph.graph import Graph


class TransitionResetMatrix(Transition):
    def __init__(self, sess: tf.Session, name: str, graph: Graph,
                 beta: float) -> None:
        Transition.__init__(self, sess, name, graph)

        self.beta = beta
        self.transition = tf.Variable(
            tf.where(self.G.is_not_sink_tf,
                     tf.add(
                         tf.scalar_mul(beta, tf.div(self.G.A_tf,
                                                    self.G.out_degrees_tf)),
                         (1 - beta) / self.G.n_tf),
                     tf.fill([self.G.n, self.G.n], tf.pow(self.G.n_tf, -1))),
            name=self.name)
        self.run(tf.variables_initializer([self.transition]))

    def __call__(self, *args, **kwargs):
        return self.transition

    def update_edge(self, edge: np.ndarray, change: float) -> None:

        # print("Edge: " + str(edge) + "\tChange: " + str(change))

        if change > 0.0:
            self.run(tf.scatter_nd_update(
                self.transition, [[edge[0]]],
                tf.add(
                    tf.scalar_mul(
                        self.beta,
                        tf.div(
                            self.G.A_tf_vertex(edge[0]),
                            self.G.out_degrees_tf_vertex(edge[0]))),
                    (1 - self.beta) / self.G.n_tf)))
        else:
            self.run(tf.scatter_nd_update(
                self.transition, [[edge[0]]],
                tf.where(self.G.is_not_sink_tf_vertex(edge[0]),
                         tf.add(
                             tf.scalar_mul(
                                 self.beta,
                                 tf.div(
                                     self.G.A_tf_vertex(edge[0]),
                                     self.G.out_degrees_tf_vertex(edge[0]))),
                             (
                                 1 - self.beta) / self.G.n_tf),
                         tf.fill([1, self.G.n], tf.pow(self.G.n_tf, -1)))))
        self._notify(edge, change)
```

Figura B.10: `transition_reset_matrix.py`

```python
import tensorflow as tf
import numpy as np

from tfg_big_data_algorithms.pagerank.transition.transition_reset_matrix import \
    TransitionResetMatrix
from tfg_big_data_algorithms.graph.graph import Graph


class TransitionRandom(TransitionResetMatrix):
    def __init__(self, sess: tf.Session, name: str, graph: Graph,
                 beta: float) -> None:
        TransitionResetMatrix.__init__(self, sess, name + "_log", graph, beta)

        self.run(self.transition.assign(tf.log(self.transition)))

    def __call__(self, *args, **kwargs):
        return self.get_tf(*args)

    def get_tf(self, t: int):
        return (tf.scatter_nd(
            tf.reshape(tf.multinomial(self.transition, num_samples=1),
                       [self.G.n, 1]),
            tf.fill([self.G.n], 1 / (self.G.n + t)), [self.G.n]))

    def update_edge(self, edge: np.ndarray, change: float) -> None:
        # TODO
        pass
```

Figura B.11: `transition_random.py`

B.4. `utils` package

```python
import pandas as pd
import numpy as np


class DataSets:
    @staticmethod
    def _get_path() -> str:
        return "./../datasets"

    @staticmethod
    def _name_to_default_path(name: str) -> str:
        return DataSets._get_path() + '/' + name + '/' + name + ".csv"

    @staticmethod
    def _permute_edges(edges_np: np.ndarray) -> np.ndarray:
        return np.random.permutation(edges_np)

    @staticmethod
    def _compose_from_path(path: str, index_decrement: bool) -> np.ndarray:
        data = pd.read_csv(path)
        if index_decrement:
            data -= 1
        return DataSets._permute_edges(data.as_matrix())

    @staticmethod
    def _compose_from_name(name: str, index_decrement: bool) -> np.ndarray:
        return DataSets._compose_from_path(DataSets._name_to_default_path(name),
                                           index_decrement)

    @staticmethod
    def followers(index_decrement: bool = True) -> np.ndarray:
        return DataSets._compose_from_name('followers', index_decrement)

    @staticmethod
    def wiki_vote(index_decrement: bool = True) -> np.ndarray:
        return DataSets._compose_from_name('wiki-Vote', index_decrement)

    @staticmethod
    def generate_from_path(path: str, index_increment=True) -> np.ndarray:
        return DataSets._compose_from_path(path, index_increment)

    @staticmethod
    def naive_4() -> np.ndarray:
        """
            url: http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html
        """
        return DataSets._permute_edges(np.array([
            [0, 1],
            [0, 2],
            [0, 3],
            [1, 2],
            [1, 3],
            [2, 0],
            [3, 0],
            [3, 2]]))

    @staticmethod
    def naive_6() -> np.ndarray:
        """
            url: https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/moler/exm/chapters/pagerank.pdf
        """
        return DataSets._permute_edges(np.array([
            [1, 2],
            [1, 6],
            [2, 3],
            [2, 4],
            [3, 4],
            [3, 5],
            [3, 6]
```

```python
class UpdateEdgeNotifier:
    def __init__(self):
        self._listeners = set()

    def attach(self, observer):
        self._listeners.add(observer)

    def detach(self, listener):
        self._listeners.discard(listener)

    def _notify(self, edge, change):
        for observer in self._listeners:
            observer.update_edge(edge, change)
```

Figura B.13: update_edge_notifier.py

```python
import warnings
from typing import TypeVar

import tensorflow as tf


TF_type = TypeVar('TF_type', tf.Variable, tf.Tensor)


class TensorFlowObject(object):
    """
    This is a proof of class documentation

    """
    def __init__(self, sess: tf.Session, name: str,
                 writer: tf.summary.FileWriter = None,
                 is_sparse: bool = False) -> None:
        """
        This is a proof of __init__ documentation

        :param sess:
        :param name:
        :param writer:
        :param is_sparse:
        """
        self.sess = sess
        self.name = name
        self.writer = writer
        if is_sparse:
            warnings.warn('TensorFlow not implements Sparse Variables yet!')
            # self.is_sparse = is_sparse

    def run(self, input_to_run):
        """
        This is a proof of method documentation

        :param input_to_run:
        :return:
        """
        return self.sess.run(input_to_run)
```

Figura B.14: tensorflow_object.py

```python
import numpy as np


class Utils:
    @staticmethod
    def ranked(x: np.ndarray, axis: int = 1) -> np.ndarray:
        return np.argsort(x, axis=axis)

    @staticmethod
    def save_ranks(filename: str, array: np.ndarray,
                   index_increment: bool = True) -> None:
        if index_increment:
            array[:, 0] += 1
        np.savetxt(filename, array, fmt='%i,%f',
                   header='vertex_id,page_rank', comments='')
```

Figura B.15: `utils.py`

```python
from enum import Enum
import tensorflow as tf

from tfg_big_data_algorithms.utils.vector_norm import VectorNorm


class ConvergenceCriterion(Enum):
    ONE = lambda i, x, y, c, n, dist=None: tf.reshape(
        VectorNorm.ONE(tf.subtract(x, y)) > [c], [])

    INFINITY = lambda i, x, y, c, n,dist=None: tf.reshape(VectorNorm.INFINITY(
        tf.subtract(x, y)) > [c / n], [])

    def __call__(self, *args, **kwargs):
        return self.value[0](*args, **kwargs)
```

Figura B.16: `vector_convergence.py`

```python
from enum import Enum
import tensorflow as tf


class VectorNorm(Enum):
    ONE = lambda x: tf.reduce_sum(tf.abs(x), 1)
    TWO = lambda x, y: tf.reduce_sum(tf.multiply(x, y), 1)
    EUCLIDEAN = lambda x: VectorNorm.P_NORM(x, 2)
    P_NORM = lambda x, p: tf.pow(tf.reduce_sum(tf.pow(x, p)), 1 / p)
    INFINITY = lambda x: tf.reduce_max(tf.abs(x), 1)

    def __call__(self, *args, **kwargs):
        return self.value[0](*args, **kwargs)
```

Figura B.17: `vector_norm.py`

Apéndice C

¿Cómo ha sido generado este documento?

En este apéndice se describen tanto la estructura como las tecnologías utilizadas para redactar este documento. El estilo visual que se ha aplicado al documento se ha tratado de almoldar lo máximo posible a las especificaciones suministradas en la *guía docente* de la asignatura *Trabajo de Fin de Grado* [uva17].

Este documento ha sido redactado utilizando la herramienta de generación de documentos LaTeX[toog], en concreto se ha utilizado la distribución para sistemas *OS X* denominada *MacTeX* [tooh] desarrollada por la organización *TEX User Group*. Mediante esta estrategia todas las labores de compilación y generación de documentos *PDF* (tal y como se especifica en la guía docente) se realizan de manera local. Se ha preferido esta alternativa frente a otras como la utilización de plataformas online de redacción de documentos LaTeX como *ShareLateX* [tooj] u *Overleaf* [tooi] por razones de flexibilidad permitiendo trabajar en lugares en que la conexión a internet no esté disponible. Sin embargo, dichos servicios ofrecen son una buena alternativa para redactar documentos sin tener que preocuparse por todos aquellos aspectos referidos con la instalación de la distribución u otros aspectos como un editor de texto. Además garantizan un alto grado de confiabilidad respecto de pérdidas inesperadas.

Junto con la distribución LaTeX se han utilizado una gran cantidad de paquetes que extienden y simplifican el proceso de redactar documentos. Sin embargo, debido al tamaño de la lista de paquetes, esta será obviada en este apartado, pero puede ser consultada visualizando el correspondiente fichero `thestyle.sty` del documento.

Puesto que la alternativa escogida ha sido la de generar el documento mediante herramientas locales es necesario utilizar un editor de texto así como un visualizador de resultados. En este caso se ha utilizado *Atom* [tooa], un editor de texto de propósito general que destaca sobre el resto por ser desarrollado mediante licencia de software libre (*MIT License*) y estar mantenido por una amplia comunidad de desarrolladores además de una extensa cantidad de paquetes con los cuales se puede extender su funcionalidad. En este caso, para adaptar el comportamiento de *Atom* a las necesidades de escritura de texto con latex se han utilizados los siguientes paquetes: *latex* [tooc], *language-latex* [toob], *pdf-view* [tood] encargados de añadir la capacidad de compilar ficheros latex, añadir la sintaxis y permitir visualizar los resultados respectivamente.

Puesto que el *Trabajo de Fin de Grado* se refiere a algo que requiere de un periodo de tiempo de elaboración largo, que además sufrirá una gran cantidad de cambios, se ha creido conveniente la utilización de una herramienta de control de versiones que permita realizar un seguimiento de los cambios de manera organizada.

Para ello se ha utilizado la tecnología *Git* [tooe] desarrollada originalmente por *Linus Torvalds*. En este caso en lugar de confiar en el entorno local u otro servidor propio se ha preferido utilizar la plataforma *GitHub* [toof], la cual ofrece un alto grado de confiabilidad respecto de posibles perdidas además de alojar un gran número de proyectos de software libre. A pesar de ofrecer licencias para estudiantes que permiten mantener el repositorio oculto al público, no se ha creido necesario en este caso, por lo cual se puede acceder al través de la siguiente url: `https://github.com/garciparedes/tfg-big-data-algorithms` [GP17]

Una vez descritas las distintas tecnologías y herramientas utilizadas para la elaboración de este trabajo, lo siguiente es hablar sobre la organización de ficheros. Todos los ficheros utilizados para este documento (obviando las referencias bibliográficas) han sido incluidos en el repositorio indicado anteriormente [GP17].

Para el documento, principal alojado en el directorio `/document/` se ha seguido una estructura modular, dividiendo los capítulos, apéndices y partes destacadas como portada, bibliografía o prefacio entre otros en distintos ficheros, lo cual permite un acceso sencillo a los mismos. Los apéndices y capítulos se han añadido en los subdirectorios separados. Para la labor de combinar el conjunto de ficheros en un único documento se ha utilizado el paquete *subfiles*. El fichero raiz a partir del cual se compila el documento es `document.tex`. La importación de los distintos paquetes así como la adaptación del estulo del documento a los requisitos impuestos se ha realizado en `thestyle.sty` mientras que el conjunto de variables necesarias como el nombre de los autores, del trabajo, etc. se han incluido en `thevars.sty`.

En cuanto al documento de resumen, en el cual se presenta una vista panorámica acerca de las distintas disciplinas de estudio relacionadas con el *Big Data* se ha preferido mantener un único fichero debido a la corta longitud del mismo. Este se encuentra en el directorio `/summary/`.

Por último se ha decidido añadir otro directorio denominado `/notes/` en el cual se han añadido distintas ideas de manera informal, así como enlaces a distintos cursos, árticulos y sitios web en que se ha basado la base bibliográfica del trabajo. En la figura C.1 se muestra la estructura del repositorio en forma de árbol.

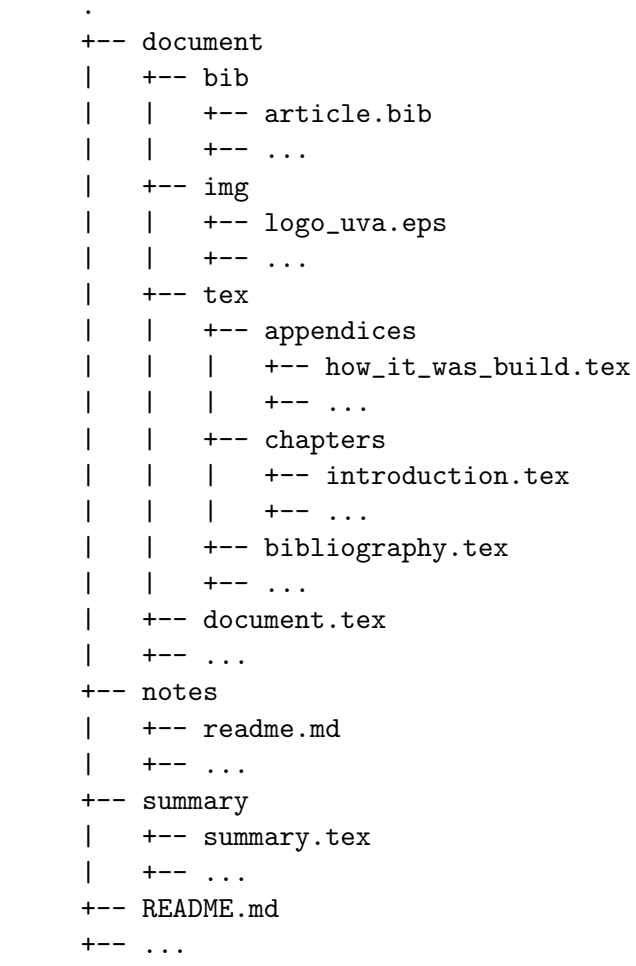


Figura C.1: Árbol de directorios del repositorio

Apéndice D

Guía de Usuario

D.1. Requisitos de Uso

[TODO]

D.2. Guía de Instalación

[TODO]

D.3. Documentación

[TODO]

D.4. Preguntas Frecuentes

[TODO]

Bibliografía

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. Society for Industrial and Applied Mathematics, 2012.

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14. ACM, 2012.

[AGPR99] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD Record*, volume 28, pages 275–286. ACM, 1999.

[AH01] Ali N Akansu and Richard A Haddad. *Multiresolution signal decomposition: transforms, subbands, and wavelets*. Academic Press, 2001.

[AMP+13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[Asp13] James Aspnes. Notes on discrete mathematics cs 202: Fall 2013. 2013.

[Asp16] James Aspnes. Notes on randomized algorithms cpsc 469/569. Fall 2016.

[Bab79] László Babai. Monte-carlo algorithms in graph isomorphism testing, 1979.

[BC06] David A Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM04] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[Boy05] MIKE Boyle. Notes on the perron-frobenius theory of nonnegative matrices. *Dept. of Mathematics, University of Maryland, College Park, MD, USA, Web site: http://www. math. umd. edu/˜ mboyle/courses/475sp05/spec. pdf*, 2005.

[BSS12] Joshua Batson, Daniel A Spielman, and Nikhil Srivastava. Twice-ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.

[BYKS02] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[CCFC02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[CEK+15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[CF13] Graham Cormode and Donatella Firmani. On unifying the space of l0-sampling algorithms. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 163–172. Society for Industrial and Applied Mathematics, 2013.

[CGHJ12] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[Cha04] Shuchi Chawla. Notes on randomized algorithms: Chernoff bounds. October 2004.

[CM05] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[DF03] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[Elk07] Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. In *International Colloquium on Automata, Languages, and Programming*, pages 716–727. Springer, 2007.

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms 2007 (AofA07)*, pages 127–146, 2007.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[FM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[Fre10] Reva Freedman. Notes on data structures and algorithm analysis: Graphs. Spring 2010.

[GG04] Minos Garofalakis and Phillip B Gibbons. Probabilistic wavelet synopses. *ACM Transactions on Database Systems (TODS)*, 29(1):43–90, 2004.

[GGL+13] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. ACM, 2013.

[Gle15] David F Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[GP17] Sergio García Prado. Trabajo de Fin de Grado: Algoritmos para Big Data, 2017. `https://github.com/garciparedes/tfg-big-data-algorithms`.

[Har11] Nick Harvey. Cpsc 536n: Randomized algorithms. December 2011.

[Hav02] Taher H Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, pages 517–526. ACM, 2002.

[HNH13] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[Ins17] Institute for Telecommunication Sciences. Definitions: Data stream, March 2017. `https://www.its.bldrdoc.gov/fs-1037/dir-010/_1451.htm`.

[IW05] Piotr Indyk and David Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 202–208. ACM, 2005.

[JST11] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 49–58. ACM, 2011.

[JW02] Glen Jeh and Jennifer Widom. Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 538–543. ACM, 2002.

[JW03] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, pages 271–279. ACM, 2003.

[Kar92] Richard M. Karp. On-line algorithms versus off-line algorithms: How much is it worth to know the future? pages 416–429, 1992.

[KHMG03] Sepandar Kamvar, Taher Haveliwala, Christopher Manning, and Gene Golub. Exploiting the block structure of the web for computing pagerank. Technical report, Stanford, 2003.

[Kle99] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[KNW10] Daniel M Kane, Jelani Nelson, and David P Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 41–52. ACM, 2010.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[LM01] Ronny Lempel and Shlomo Moran. Salsa: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems (TOIS)*, 19(2):131–160, 2001.

[LMPMSB14] Esperanza Larrinaga Miner, María Felisa Pérez Martínez, and Araceli Suárez Barrio. Grado en ingeniería informática, uva: Matemática discreta. 2014.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[McG14] Andrew McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[MP80] J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[MR10] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[Mut05] S. Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, August 2005.

[Nel15] Jelani Nelson. Algorithms for big data. Fall 2015.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM (JACM)*, 49(1):16–34, 2002.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2):256–276, 1984.

[SBC+06] Tamás Sarlós, Adrás A Benczúr, Károly Csalogány, Dániel Fogaras, and Balázs Rácz. To randomize or not to randomize: space optimal summaries for hyperlink analysis. In *Proceedings of the 15th international conference on World Wide Web*, pages 297–306. ACM, 2006.

[SBK05] Ivan W Selesnick, Richard G Baraniuk, and Nick C Kingsbury. The dual-tree complex wavelet transform. *IEEE signal processing magazine*, 22(6):123–151, 2005.

[SGP11] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *Journal of the ACM (JACM)*, 58(3):13, 2011.

[SJNSB16] Luis Augusto San José Nieto and Araceli Suárez Barrio. Grado en ingeniería informática, uva: Estadística. 2016.

[SR94] K. G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, Jan 1994.

[SSCRGM12] Diana Alejandra Sánchez-Salas, José Luis Cuevas-Ruíz, and Miguel González-Mendoza. Wireless channel model with markov chains using matlab. *Edited by Vasilios N. Katsikis*, page 235, 2012.

[tooa] Atom. `https://atom.io`.

[toob] Atom Package: Language LaTeX. `https://atom.io/packages/language-latex`.

[tooc] Atom Package: LaTeX. `https://atom.io/packages/latex`.

[tood] Atom Package: PDF View. `https://atom.io/packages/pdf-view`.

[tooe] Git. `https://www.git-scm.com`.

[toof] GitHub. `https://github.com`.

[toog] LaTeX. `https://www.latex-project.org`.

[tooh] MacTex. `http://www.tug.org/mactex`.

[tooi] Overleaf. `https://www.overleaf.com`.

[tooj] ShareLaTeX. `https://www.sharelatex.com`.

[uva17] Trabajo de Fin de Grado (Mención en Computación), 2016/17. `https://www.inf.uva.es/wp-content/uploads/2016/06/G46978.pdf`.

[Wik17a] Wikipedia. Big data — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Big%20data&oldid=782715255`, 2017. [Online; accessed February-2017].

[Wik17b] Wikipedia. Bloom filter — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Bloom%20filter&oldid=776072844`, 2017. [Online; accessed 03-June-2017].

[Wik17c] Wikipedia. Borůvka's algorithm — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Bor%C5%AFvka's%20algorithm&oldid=773835173`, 2017. [Online; accessed 16-June-2017].

[Wik17d] Wikipedia. Community structure — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Community%20structure&oldid=781588402`, 2017. [Online; accessed 12-June-2017].

[Wik17e] Wikipedia. Graph (discrete mathematics) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Graph%20(discrete%20mathematics)&oldid=779199071, 2017. [Online; accessed 07-June-2017].

[Wik17f] Wikipedia. Minimum spanning tree — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Minimum%20spanning%20tree&oldid=782387339, 2017. [Online; accessed 15-June-2017].

[Woo09] David Woodruff. Frequency moments. *Encyclopedia of Database Systems*, pages 1169–1170, 2009.