PROYECTO DE GRADO

CÓDIGOS DE PARIDAD DE BAJA DENSIDAD (LOW-DENSITY PARITY-CHECK CODES, LDPC)

Alix Lhéritier

Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Tutor: Dr. Alfredo Viola

RESUMEN

En 1948, C. E. Shannon propuso una teoría matemática para el problema general de la comunicación confiable de información a través de medios no confiables que, en particular, abarca problemas como el de transmitir información de un lugar a otro o el de almacenar información a ser recuperada más tarde. Además de dar las bases matemáticas para modelar el problema, Shannon demostró que, si la información es representada y procesada en forma adecuada, la confiabilidad puede ser arbitrariamente alta si y sólo si la eficiencia con la cual se utiliza el medio no supera un cierto límite que depende del mismo. Sin embargo, el teorema de Shannon no dice nada sobre la complejidad de los algoritmos que representan y procesan la información. La búsqueda de cómo representar y procesar la información para comunicar con eficiencia cercana al límite, en distintos tipos de medios, con alta confiabilidad y con algoritmos de baja complejidad ha resultado un verdadero desafío, desde ese entonces, para matemáticos e ingenieros en comunicaciones. Los Códigos de Paridad de Baja Densidad (Low-Density Parity-Check, LDPC) fueron descubiertos en 1963 por R. G. Gallager pero, debido a las limitaciones existentes en la capacidad de cómputo y de almacenamiento de los equipos disponibles en la época, fueron abandonados y, recién en la década de los 90', fue descubierto su verdadero potencial para acercarse al límite de Shannon, con confiabilidad arbitrariamente alta y con baja complejidad en los algoritmos. En este proyecto, presentamos un estudio de la teoría fundamental de los Códigos LDPC junto con los elementos básicos de la Teoría de la Información y de la Codificación. Además, presentamos los resultados de un trabajo de experimentación con el objetivo de verificar empíricamente algunos resultados importantes de la teoría. Finalmente, proponemos varios trabajos a realizar en el futuro, relacionados con el tema de los Códigos LDPC.

AGRADECIMIENTOS

Quiero agradecer al Dr. Alfredo Viola y al Dr. Gadiel Seroussi por apoyarme, estimularme y enriquecerme con sus experiencias y sus conocimientos, a lo largo del proyecto, y por introducirme a las áreas fascinantes de la Teoría de la Información y de la Codificación.

TABLA DE CONTENIDO

1 INTRODUCCIÓN

En [30], C. E. Shannon presenta como **problema fundamental de la comunicación** el de reproducir en un destino, de manera exacta o aproximada, un mensaje enviado por una fuente y transmitido a través de un medio que puede distorsionar o dañar los mensajes enviados. El planteo de Shannon es suficientemente genérico como para abarcar una amplia variedad de problemas concretos. En particular, la fuente y el destino pueden ser distantes tanto en el espacio como en el tiempo, es decir que podemos considerar tanto el problema de transmitir información de un lugar a otro (por ejemplo, dos personas que se comunican a través de una pieza ruidosa) como el problema de almacenar información que se desea recuperar más tarde (por ejemplo, escribir un libro que va a ser leído muchos años después).

Se supone que la manera en la cual el medio daña la información escapa a nuestro control. Sin embargo, se supone que es posible elegir el mecanismo que permite representar (o "codificar") la información antes de ser enviada y el mecanismo que maneja (o "decodifica") la información recibida antes de ser entregada en su forma final al destinatario. Son estos dos mecanismos los que permiten encontrar soluciones al problema fundamental.

La comunicación de información ha tomado una variedad de sofisticadas formas en las cuales la codificación ha sido usada en forma exitosa:

- Una de las primeras aplicaciones que existió tenía el objetivo de corregir los errores ocurridos al transmitir por largas líneas telefónicas que se veían expuestas a tormentas eléctricas y a otras interferencias.
- La capacidad de transmisión y de recepción de la mayoría de los modems se ve incrementada por el uso de sistemas de codificación embebidos en su hardware.
- La comunicación a través del Espacio se ve afectada por condiciones atmosféricas y por los rayos solares y es por eso que, desde hace años, la información transmitida en las misiones espaciales viene siendo codificada.

- También, la codificación ha encontrado su aplicación en Internet, por ejemplo, para permitir recuperar paquetes que se pierden en la red.
- La información almacenada en medios magnéticos está expuesta a rayos gama y a interferencia magnética que la pueden dañar; por lo tanto, en estos casos también es de gran utilidad codificar la información.
- Cuando Phillips diseñó la tecnología del disco compacto, un requisito era que la información almacenada en el disco fuera resistente a diferentes tipos de daños como, por ejemplo, el que ocurre cuando se raya el disco, y es por eso que incorporó codificación en sus discos compactos.

Shannon presentó dos resultados fundamentales que indican los límites que se pueden alcanzar en cuanto a la eficiencia en el uso del medio cuando se comunica información en forma confiable. En particular, estos resultados dan una cota superior a la velocidad que se puede alcanzar, cuando se transmite información de un lugar a otro, y una cota inferior a la cantidad de espacio necesario para almacenar información.

También, es deseable que los mecanismos de codificación y de decodificación tengan la menor complejidad en tiempo de cómputo y espacio de almacenamiento. En algunos casos, como el de la comunicación (a través del espacio) en tiempo real[1], el tiempo de cómputo es un factor primordial, dado que si no se quiere desperdiciar ancho de banda, los algoritmos deben ser suficientemente veloces para no dejar ocioso al medio de comunicación.

La prueba de Shannon sobre los límites de eficiencia alcanzables se basó en sistemas de codificación difíciles de usar en la práctica debido a su complejidad. Por lo tanto, estos resultados plantearon el desafío de encontrar sistemas de codificación que alcancen esos límites y al mismo tiempo sea posible usarlos en la práctica.

[1] La comunicación en tiempo real exige que la información llegue al destino en poco tiempo (milisegundos o microsegundos) después de que haya sido generada por la fuente.

En su tesis de doctorado de 1961 (publicada en 1963), Gallager ([9]) inventó los códigos LDPC y la decodificación iterativa. Pero debido a limitaciones existentes en los equipos de la época y a ciertas confusiones (ver, por ejemplo, [17]), salvo algunas excepciones, los sistemas de codificación iterativos fueron olvidados durante décadas. En 1993, se inventó otro sistema de codificación iterativa: los códigos Turbo ([2]). A la luz de este descubrimiento, los códigos LDPC fueron redescubiertos por MacKay y Neal ([17]) en 1995. Luego de ese redescubrimiento, han surgido y siguen surgiendo una cantidad importante de trabajos relacionados con el tema.

El gran interés por los sistemas de codificación basados en códigos LDPC se debe a que permiten comunicar con eficiencia muy cercana al límite establecido por Shannon, con confiabilidad arbitrariamente grande y con muy baja complejidad para una gran variedad de medios de comunicación.

Algunas de las aplicaciones posibles de los códigos LDPC son:

- recuperación de paquetes perdidos en la distribución de datos masivos a varios clientes en forma simultánea a través de Internet ([3])
- almacenamiento en medios magnéticos ([34])
- almacenamiento distribuido de información ([21])
- corrección de errores en telefonía común o inalámbrica y en modems.

En este proyecto, lo que hicimos fue:

- Hacer un estudio de los fundamentos teóricos de los códigos LDPC junto con los trabajos más importantes de los últimos años sobre el tema. También, estudiamos los fundamentos de la teoría de códigos y de la teoría de la información para poder situar los códigos LDPC en su contexto.
- Implementar un simulador para evaluar empíricamente el rendimiento de un sistema de comunicación basado en códigos LDPC para comparar con los resultados teóricos.
- Buscar trabajos relacionados con los códigos LDPC para realizar en el futuro.

Esta monografía se divide de la siguiente forma. En el capítulo 2, presentamos algunos fundamentos de la teoría de la codificación y de la teoría de la información necesarios para el estudio de los códigos LDPC. En el capítulo 3, presentamos un estudio de la teoría de los códigos LDPC poniendo énfasis en la teoría disponible para el canal de borradura, el cual es un modelo simple de medio de comunicación. A pesar de su simplicidad, muchos de los resultados se pueden extender con relativa facilidad para otros medios más complejos. Además, el canal de borradura tiene su propio interés práctico debido a que modela bastante bien la Internet y las pérdidas de paquetes que ocurren en ella. En el capítulo 4, describimos las pruebas realizadas con un sistema implementado que permitió la evaluación empírica de algunos de los aspectos de los códigos LDPC usando el canal de borradura. En el capítulo 5, proponemos tres trabajos a realizar en el futuro, relacionados con los códigos LDPC. Finalemente, en el capítulo 6, damos las conclusiones del proyecto.

2 TEORÍA DE LA INFORMACIÓN Y CODIFICACIÓN

El trabajo de Shannon abrió dos áreas científicas fuertemente relacionadas que son la Teoría de la Información y la Teoría de la Codificación. Ésta última tiene como objetivo el de encontrar maneras prácticas de representar la información (o de "codificarla") para acercarse a los límites que establece la Teoría de la Información[2].

En este capítulo, presentamos los elementos esenciales de ambas teorías, necesarios para el estudio de los códigos LDPC. Por más información, consultar, por ejemplo, [4], [26] o [10].

2.1 Un modelo para la comunicación

El modelo matemático propuesto por Shannon se ilustra en la Figura 1.

Figura 1

Sus componentes son los siguientes:

- Una **fuente de información**: es un proceso que genera símbolos pertenecientes a un alfabeto finito, en forma discreta, o valores reales, en forma continua. Los símbolos o los valores generados son de interés para un **destino**.
- Un **codificador**: es un mecanismo que opera sobre la salida de la fuente para ponerla en una forma adecuada a la transmisión.
- Un **canal**: es un modelo del medio usado para la transmisión de los mensajes. En particular, puede modelar un cable eléctrico, una banda de frecuencias de radio, un rayo de luz, una superficie magnética, etc.

[2] La Teoría de la Información actualmente abarca problemas más generales aún que el problema de la comunicación y está relacionada con áreas tan diferentes como la Física, las Ciencias Económicas y las Ciencias de la Computación (por más información consultar, por ejemplo, [4]).

- Un **decodificador**: es un mecanismo que normalmente realiza la operación inversa del codificador, intentando recuperar en la forma más exacta posible el mensaje originalmente emitido por la fuente.

En este proyecto, sólo estarán en cuestión sistemas de comunicación de tiempo discreto, en los cuales cada una de las partes anteriores trabaja con secuencias discretas de símbolos pertenecientes a un alfabeto finito.
A continuación describimos más precisamente cada uno de los componentes.

2.2 Fuentes de Información

Una **fuente de información** es un proceso que genera una secuencia de símbolos. Una fuente de información puede tomar diversas formas, por ejemplo: un libro, un reporte financiero, una danza, una música, una ecuación matemática, etc. En particular, es de interés el caso en el cual el proceso que genera los símbolos es estocástico y markoviano. Cada vez que el proceso pasa de un estado a otro se genera un símbolo dependiente de la transición y los símbolos generados pertenecen a un alfabeto finito. En [30], Shannon justificó el estudio de las fuentes de información gobernadas por procesos de Markov mostrando que se puede aproximar razonablemente bien una fuente de información que produce lenguaje natural con un proceso de Markov.

Por ejemplo, si modelamos el estado del tiempo en Montevideo a las 8:00 AM como un proceso de Markov que toma valores en un alfabeto finito que puede ser, por ejemplo, {Soleado, Nublado, Lluvia, Niebla}, entonces tenemos una fuente de información que todos los días a las 8:00 AM genera un símbolo que corresponde al estado del tiempo.

Son de especial interés los procesos de Markov que son ergódicos. Informalmente, el interés por los procesos ergódicos se debe a que nos aseguran que la mayoría de las secuencias producidas tienen las mismas propiedades estadísticas. En esos casos, las frecuencias de aparición de los distintos símbolos en secuencias particulares de cierto largo tienden, a medida que el largo crece, a límites definidos independientes de la secuencia

particular. En realidad, esto no es verdadero para todas las secuencias pero el conjunto de las secuencias para las cuales esto es falso tiene probabilidad cero de ocurrir.

Queremos medir la **cantidad de información** que recibimos en promedio por cada símbolo cuando una fuente produce una secuencia de símbolos. Intuitivamente, si podemos prescindir de ver los símbolos que genera la fuente para saber cuales son, entonces la fuente no nos brinda información alguna. Por ejemplo, si el estado del tiempo fuera siempre Niebla a las 8:00 AM en Montevideo, entonces no tendríamos ninguna **incertidumbre** sobre lo que va a suceder cada día y, por lo tanto, daría lo mismo mirar por la ventana y evaluar el estado del tiempo que suponer que es Niebla y prescindir de verlo. En un caso menos extremo, en el cual sabemos que con alta probabilidad el estado es Niebla podemos prescindir también de la evaluación y suponer que el estado es Niebla y así tener baja probabilidad de equivocarnos y, por lo tanto, podemos decir que la fuente nos brinda poca información (y tenemos poca incertidumbre). El caso en el cual la fuente nos brinda máxima información es, intuitivamente, cuando la incertidumbre es máxima lo cual sucede cuando los símbolos se emiten con igual probabilidad y, por lo tanto, si prescindimos de la salida de la fuente y hacemos alguna suposición entonces nos equivocaremos con alta probabilidad.

En el ejemplo anterior, vimos, intuitivamente, que la incertidumbre que tenemos frente a lo que la fuente puede emitir es igual a la cantidad de información que se espera que la fuente emita. Es decir que la cantidad de información sólo depende de la cantidad de símbolos y de sus probabilidades de ocurrir y no depende de la forma de los símbolos en sí. Por ejemplo, en vez de mirar por la ventana el estado del tiempo, alguien nos podría decir la palabra del conjunto {Soleado, Nublado, Lluvia, Niebla} que describe el estado del tiempo o nos podría decir un número entre 0 y 3 si previamente nos pusimos de acuerdo en la correspondencia de los números con los estados posibles del tiempo.

Shannon buscó una función H que permitiera medir la **incertidumbre** que tenemos frente a lo que puede suceder si consideramos un conjunto de n eventos posibles acerca de los

cuales conocemos únicamente sus probabilidades respectivas de ocurrir $p_1, p_2, ..., p_n$. En el caso de existir tal medida, que denotamos $H(p_1, p_2, ..., p_n)$, Shannon sugirió que debiera satisfacer tres propiedades que se detallan en el Apéndice A. Shannon demostró que, para satisfacer esas tres propiedades, H debe ser de la forma:

$$H = -K\sum_{i=1}^{n} p_i \log p_i$$

donde K es una constante positiva que corresponde simplemente a elegir las unidades y se puede tomar igual a 1. Shannon llamó **entropía** a esta función en analogía a la mecánica estadística.

Es posible definir la **entropía condicional** de un conjunto y sabiendo que ocurrió algún evento de x como la suma ponderada de la entropía de y para cada valor de x. Esto es: $H(y \mid x) \coloneqq \sum_{i \in x} \Pr(i) H(y \mid i) = \sum_{i \in x, j \in y} \Pr(i, j) \log \Pr(j \mid i)$

En el Apéndice A, se enuncian algunas propiedades más que posee la función entropía y que la sustentan aún más como medida de la incertidumbre o de la cantidad de información.

Volviendo a las fuentes de información, observamos que para cada estado i posible hay un conjunto de probabilidades $\Pr(j \mid i)$ de que se genere cada símbolo j y, por lo tanto, hay una entropía H_i para cada estado. La **entropía de una fuente de información** mide el valor esperado de la cantidad de información generada por la fuente y se define como el promedio de las entropías H_i ponderando de acuerdo a la probabilidad P_i de estar en cada estado i, es decir $H = \sum_i P_i H_i$.

La unidad de la cantidad de información depende de la base del logaritmo. Si el logaritmo es en base r entonces hablaremos de **unidades $r-$arias de información**. Si el logaritmo es en base 2, la unidad es el **bit de información**. Por lo tanto, si el logaritmo es en base r, la entropía (o la cantidad esperada de información) de una fuente se expresa en unidades

$r-$arias de información por símbolo de la fuente. A los símbolos que pertenecen a un alfabeto binario se les suele llamar bits, sin embargo queda claro que, sólo cuando los símbolos binarios se generan en forma equiprobable, se genera un bit de información por símbolo binario.

Si el proceso de Markov que genera los símbolos tiene una velocidad asociada a su funcionamiento, entonces es posible expresar la entropía en unidades de información por segundo (por más información, consultar [1]).

En el ejemplo inicial, si suponemos que el proceso está gobernado por un proceso de Markov de un solo estado con $\Pr(\text{Nublado})=1/5, \Pr(\text{Soleado})=1/2, \Pr(\text{Niebla})=1/10, \Pr(\text{Lluvia})=1/5$, la entropía de la fuente es:

$$H = -1/5\log_2 1/5 - 1/2\log_2 1/2 - 1/10\log_2 1/10 - 1/5\log_2 1/5$$
$$\approx 1.76 \text{ bits por símbolo}$$

Si los símbolos fueran equiprobables la entropía sería de 2 bits por símbolo.

2.3 Canales

Un canal es un modelo del medio usado para la transmisión de información. Un **canal discreto** es un sistema que opera en instantes de tiempo discretos y que consiste en un alfabeto de entrada $\mathcal{X}$, un alfabeto de salida $\mathcal{Y}$ y un conjunto de probabilidades de transición $p(y \mid x)$ que describen la probabilidad de obtener el símbolo y a la salida, sabiendo que entró el símbolo x. Las distribuciones de probabilidad pueden depender de los símbolos que ingresaron o salieron previamente. En este proyecto, nos interesan solamente los canales **sin memoria** que son aquellos cuya salida en el tiempo t depende únicamente de la entrada en el tiempo t.

Ejemplo: en el **canal binario de borradura** (Binary Erasure Channel, BEC) los símbolos que entran se "pierden" con probabilidad ε. Formalmente, el alfabeto de entrada puede ser representado como $\{0,1\}$ y el alfabeto de salida como $\{0,1,?\}$. Las probabilidades de

transición se muestran en la Figura 2. La propiedad esencial que distingue a este canal de otros más comunes en la práctica (ver el Apéndice B), es la certeza que hay a la salida sobre la presencia de un error o no.

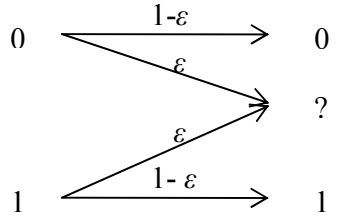

Figura 2

El canal de borradura, introducido por P. Elias en [7], fue considerado durante mucho tiempo como un modelo esencialmente teórico. Con el desarrollo de las redes de computadoras, el modelo resultó de utilidad práctica para modelar la pérdida de paquetes de datos que ocurre en las mismas. Los paquetes están numerados y, a veces, por distintas razones (por ejemplo, cuando se satura un nodo intermedio) los paquetes se pierden. Si suponemos que los paquetes tienen un mecanismo interno (un código para corrección de errores) que asegura que cuando llegan, llegan bien o se detectan los errores y se marcan los paquetes como perdidos, entonces se cumple la propiedad esencial del canal de borradura que es independiente del tamaño del alfabeto. Es necesario, asegurarse también de que las borraduras ocurran en forma independiente para que el canal sea sin memoria, lo cual no es siempre cierto (por ejemplo, debido a que las pérdidas de paquetes pueden ocurrir por problemas, como la saturación, que implican la pérdida de varios paquetes consecutivos) pero es posible simularlo modificando el orden de los paquetes antes de transmitirlos y reordenándolos al llegar (ver, por ejemplo, [16]).

El modelo del canal de borradura ha permitido desarrollar gran parte de la teoría disponible para los códigos LDPC debido a su simplicidad que hace más fácil el análisis del comportamiento de los códigos en este canal. Sin embargo, a pesar de su peculiar simplicidad, muchos de los resultados pudieron ser generalizados para otros canales y en forma más abstracta para una amplia clase de canales denominada **canales sin memoria de**

entrada binaria y salida simétrica (Memoryless Binary-Input Output-Symmetric, MBIOS) que se define de la forma siguiente:

- El alfabeto de entrada de este canal resulta conveniente considerarlo como $\{\pm 1\}$ y la salida del canal puede tomar valores discretos o continuos.
- Como su nombre lo dice, el canal es sin memoria.
- Para que la salida sea simétrica se debe cumplir que $\Pr(\text{salga } Y|\text{entró } 1) = \Pr(\text{salga } -Y|\text{entró } -1) \ \forall Y$.

El canal binario de borradura pertenece a esta clase[3]. En el Apéndice B, describimos dos canales que también pertenecen a esta clase y que, además, son de gran importancia práctica.

Dado que la salida de un canal es una variable aleatoria es posible calcular su entropía, $H(Y)$, así como la entropía de la salida sabiendo la entrada, $H(Y|X)$ y la de la entrada sabiendo la salida $H(X|Y)$. Decimos que un canal es **sin ruido** si $H(X|Y)=0$, es decir que, si conocemos la salida del canal, entonces no tenemos ninguna incertidumbre en cuanto a la entrada. De lo contrario decimos que es un canal **con ruido**. Cuanto más grande es $H(X|Y)$, más incertidumbre tenemos sobre lo que ingresó al canal conociendo la salida, por lo tanto, $H(X|Y)$ (que se denomina **equivocación),** intuitivamente, mide la cantidad de información que se pierde al pasar por el canal.

La **información mutua** se define como $I(X;Y)=H(X)-H(X|Y)$ y se interpreta como la cantidad de información por símbolo que efectivamente se transmite por el canal en cuestión.

[3] Para verlo más fácilmente, se puede transformar los alfabetos de entrada y de salida de la forma siguiente: $1 \to -1,\ 0 \to 1,\ ? \to 0$

Se define la **capacidad del canal** (que se interpreta como la máxima cantidad de información por símbolo que se puede transmitir por el canal) como $C = \max_{p(x)} I(X;Y)$ (el máximo se toma respecto a todas las frecuencias posibles de los símbolos de entrada).

En [7], se demuestra que la capacidad del canal de borradura es igual a $1-\varepsilon$ bits por símbolo binario, obteniendo este valor únicamente con una distribución uniforme de los símbolos de entrada. Por lo tanto, para transmitir la máxima cantidad de información por símbolo es necesario que los símbolos, que entran al canal de borradura, tengan la misma frecuencia.

Estas definiciones toman valor gracias al segundo teorema de Shannon que veremos más adelante pero que, en esencia, dice que es posible transmitir información con equivocación arbitrariamente chica si se codifica adecuadamente y la entropía a la salida del codificador es menor que la capacidad del canal.

2.4 Codificadores y decodificadores

El teorema de la separación fuente-canal (ver el Apéndice C) nos permite dividir el problema de la comunicación en dos subproblemas y resolverlos en forma independiente sabiendo que asintóticamente no perdemos nada respecto a una codificación conjunta. Por lo tanto, es posible dividir el mecanismo de codificación (y de decodificación) en dos partes: una que intenta representar la información emitida por la fuente en la forma más compacta posible (es decir, usando la menor cantidad promedio de símbolos por unidad de información) y otra que agrega símbolos que no agregan información (se dice que agrega **redundancia**) para permitir corregir o detectar los errores introducidos por el canal. Estos dos partes, se llaman respectivamente **codificación de fuente** y **codificación de canal.**

Esto es de gran importancia práctica en situaciones en las cuales se desea diseñar un código de canal independientemente de lo que pueda ser la fuente o, análogamente, diseñar un código de fuente independientemente del canal por el cual se vaya a transmitir la información. Sin embargo, cuando el largo es corto y tanto la fuente como el canal están

determinados y fijos puede resultar mejor hacer una codificación conjunta. Otra, alternativa es hacer la codificación separada pero teniendo en cuenta las probabilidades de la fuente (en el Apéndice D, se describe una estrategia de decodificación que toma en cuenta dichas probabilidades).

La codificación de fuente, cuyo objetivo es la compresión de la información, no es de mucho interés en el estudio de los códigos LDPC, sin embargo, se incluyen algunas ideas así como el resultado fundamental de Shannon, en el Apéndice C, para brindar una visión más completa del problema de la comunicación.

Un **código de canal** (M,n) (de cardinalidad M y de **largo** n) para el canal $(\mathcal{X}, p(y\,|\,x), \mathcal{Y})$ consiste en lo siguiente:

1. Una **función de codificación** que toma **vectores de información** y devuelve **palabras de código** $X^n : \mathcal{X}^k \to \mathcal{X}^n$ donde $n \geq k$. El conjunto de todas las palabras de código se denomina libro de código o, simplemente, **código**.
2. Una **función de decodificación** $g : \mathcal{Y}^n \to \mathcal{X}^n$, la cual es una regla determinística que estima la palabra de código enviada a partir del vector recibido con errores. En algunos casos, la función de decodificación intenta estimar individualmente cada uno de los símbolos de la palabra de código enviada lo cual puede resultar en un vector que no sea una palabra de código. También existen funciones de decodificación que además de corregir pueden detectar errores y, por lo tanto, son de la forma $g_d : \mathcal{Y}^n \to \left\{\mathcal{X} \cup \{e\}\right\}^n$, donde el símbolo $'e'$ representa un error. Para intentar recuperar el vector de información original es posible aplicarle la función $\left(X^n\right)^{-1}$ si la estimación corresponde a una palabra de código.

Decimos que el código es **separable** si los símbolos del vector de información se incluyen directamente en la palabra de código. Si las palabras de información son de la forma $(s\,|\,p) = X^n(s)$ se dice que el código es **sistemático**, s siendo la **parte sistemática** y p la

paridad. En estos casos, no es necesario aplicar la función $\left(X^n\right)^{-1}$ para recuperar la información original ya que se conocen las posiciones en la palabra de código donde se encuentra la información.

La **distancia mínima** d de un código C es la mínima distancia de Hamming[4] que existe entre dos palabras distintas cualesquiera de C:

$$d = \min_{c_1, c_2 \in C : c_1 \neq c_2} d(c_1, c_2).$$

La distancia mínima es importante en la teoría de códigos en general porque la tarea del decodificador es decidir qué palabra de código fue enviada a partir de lo que recibe (que puede tener bits invertidos, bits borrados, etc.) por lo tanto es importante, intuitivamente, que la palabra con errores se "parezca" lo más posible a la palabra realmente enviada y no a otra, por eso es bueno que las palabras estén lo más lejos posible unas de otras para que no haya ambigüedades. En el estudio de los códigos LDPC, la distancia mínima juega un rol menor debido al tipo de decodificación generalmente usada[5].

En este proyecto, consideraremos códigos cuyo objetivo es corregir errores. La detección de errores simplemente la veremos como un valor agregado que pueda tener cierto código, dado que permite que el destino no tome como correcta información que no se decodificó exitosamente y, además, permite iniciar alguna estrategia de recuperación como la retransmisión de la información. Por lo tanto, a los efectos de medir la confiabilidad que permite lograr un cierto código, consideraremos que los mensajes o los símbolos individuales se transmiten exitosamente si el decodificador los estima correctamente. Además, notamos que existen canales como, por ejemplo, el de borradura, en los cuales el problema de la detección es trivial.

[4] Dos vectores están a una distancia de Hamming d si difieren en d posiciones.

[5] Esto se verá con más detalle en el análisis de largo finito de los códigos LDPC, en el canal de borradura.

La **probabilidad de error por bloque de la palabra de código** X_i^n se define como $\Pr_B^{(i)} \coloneqq \Pr\left(g(Y^n) \neq X_i^n \mid X_i^n \text{ fue enviada}\right)$, es decir, la probabilidad de estimar mal la palabra enviada X_i^n (que incluye los casos en los cuales se detectan errores).

Definimos la **probabilidad de error por bloque** como el valor esperado de la probabilidad anterior, es decir $\Pr_B \coloneqq \sum_{i=1}^{|\mathcal{X}|^k} \Pr\left(X_i^n\right) \Pr_B^{(i)}$, donde $\Pr(X_i^n)$ es la probabilidad de que se envíe la palabra de código X_i^n.

Dado que los códigos LDPC disponen de algoritmos que estiman cada símbolo individualmente, es más natural medir la confiabilidad considerando la probabilidad de error por símbolo. Por lo tanto, definimos **la probabilidad de error del símbolo** j **para la palabra de código** X_i^n como $\Pr_b^{(i)}(j) \coloneqq \Pr(\hat{X}^n(j) \neq X_i^n(j) \mid X_i^n \text{ fue enviada})$ donde $\hat{X}^n = g(Y^n)$, es decir, la probabilidad de estimar mal el símbolo j sabiendo que se envió X_i^n. La **probabilidad de error por símbolo** se define como el valor esperado de la probabilidad de error por símbolo promedio es decir $\Pr_b \coloneqq \sum_{i=1}^{|\mathcal{X}|^k} \Pr\left(X_i^n\right) \frac{1}{n} \sum_j \Pr_b^{(i)}(j)$.

Cuando el código es separable tiene más sentido considerar únicamente los errores que quedan en los símbolos de información ya que son estos los que se desea realmente transmitir (ver, por ejemplo, [11]).

En [11], los autores sugieren que la probabilidad de error por símbolo de información es en general una mejor medida de la confiabilidad cuando la información que se transmite no se agrupa naturalmente en bloques de largo fijo, dado que lo que interesa realmente es que se estimen correctamente los símbolos individuales de los vectores de información y no interesa tanto si el bloque llega bien o no en su totalidad (que es lo que mide la probabilidad de error por bloque).

Existen varias estrategias de decodificación. En particular, destacamos las reglas de decodificación de **máxima verosimilitud** (maximum likelihood, ML) por bloque (respectivamente por símbolo) que minimiza la probabilidad de error por bloque (respectivamente por símbolo) siempre y cuando la distribución de los vectores de información (respectivamente de los símbolos de las palabras de código) sea uniforme[6] (ver [26]) (respectivamente [11]). La función de decodificación para la regla de **máxima verosimilitud por bloque** es $\hat{x}^{ML}(y) \coloneqq \arg\max_{x \in C} \Pr(y \mid x)$, es decir que la regla considera cada palabra de código y calcula la probabilidad de que se haya recibido lo que efectivamente se recibió suponiendo que se mandó esa palabra de código y devuelve la palabra que maximiza esa probabilidad.
La función de decodificación para la regla de **máxima verosimilitud por símbolo** es $\hat{x}_i^{ML}(y) \coloneqq \arg\max_{\alpha \in \{0,1\}} \sum_{x \in C \mid x_i = \alpha} \Pr(y \mid x)$, es decir que para cada posición de la palabra de código y para cada símbolo posible del alfabeto, considera las palabras de código que tienen ese símbolo en esa posición y acumula, sumando, la probabilidad de haber recibido lo que se recibió suponiendo que se envió cada una de las palabras consideradas y devuelve la que tiene mayor suma.

Es necesario, definir una política para los empates (cuando dos o más palabras o cuando dos o más símbolos maximizan las expresiones) que puede ser, por ejemplo, elegir, arbitrariamente, uno de los argumentos que maximizan o devolver un error (el símbolo 'e').

Es conveniente notar que la probabilidad de error de un código de canal depende no solamente del código, de las funciones de codificación y de decodificación sino que también depende del canal en el cual se va a usar el código de canal.

[6] En el Apéndice D, se describe una regla de decodificación que toma en cuenta las probabilidades de la fuente y que minimiza la probabilidad de error para cualquier distribución de la fuente

Para medir la eficiencia del código en el uso del canal es necesario definir la **tasa del código** como $R = k/n$. Este cociente mide cuanta redundancia agrega el codificador de canal.

Un ejemplo simple de código de canal es el **código de repetición**. Para evaluar su confiabilidad en forma comparativa, consideremos primero la transmisión de información sin codificar, por el canal binario de borradura de parámetro p. A la salida del canal, la probabilidad de error por bit es $\Pr_b = p$. Por lo tanto, si no codificamos, logramos una transmisión con tasa 1 (máxima) y probabilidad de error por bit p. Si esta probabilidad de error es demasiado alta para nuestros propósitos, necesitamos usar algún tipo de codificación que incluya redundancia. La idea es repetir cada bit una cantidad de veces n. Cuando el decodificador recibe un mensaje del canal de largo n (correspondiente a un solo bit de información), le alcanza con que haya por lo menos un bit no borrado para poder decodificar exitosamente. Por lo tanto, la probabilidad de error por bit de información es

$\Pr_b = \Pr(\text{ocurran } n \text{ errores}) = p^n$

Por lo tanto, para hacer tender a 0 la probabilidad de error por bit, es necesario hacer tender n a infinito pero el problema es que la tasa $(1/n)$ tiende a 0 al mismo tiempo.

Shannon demostró que esto no es así para todos los códigos y, de hecho, los códigos LDPC permiten lograr probabilidades de error arbitrariamente bajas a tasas muy cercanas al límite establecido por Shannon.

2.5 Transmisión confiable de información

En el ejemplo del código de repetición, vimos que para alcanzar una probabilidad de error arbitrariamente baja es necesario que la tasa del código tienda a cero. Si siempre fuera así, no tendría sentido hablar de una capacidad del canal sino que sería necesario definir una capacidad por cada nivel de confiabilidad posible. Sin embargo, no todos los códigos se comportan de esta manera y de eso trata el segundo teorema de Shannon. Shannon enunció

el resultado considerando en forma conjunta la codificación de la fuente y del canal. A continuación enunciamos el teorema de codificación de canal en la forma independiente de la fuente tal como se presenta en [4].

Teorema de codificación de canal: Dado un canal con capacidad C, para cada tasa R, $R < C$, existe una secuencia de códigos $\left(2^{nR}, n\right)$ tal que la máxima probabilidad de error (o sea, $\max_i \Pr_B^{(i)}$) tiende a cero. Cualquier secuencia de códigos $\left(2^{nR}, n\right)$ cuya máxima probabilidad de error tiende a cero debe tener $R \leq C$.

Como veremos más adelante, no es posible usar en la práctica los códigos aleatorios en los cuales se basó Shannon para demostrar el teorema. Por lo tanto, con este teorema, Shannon planteó el desafío de encontrar códigos con tasas cercanas a la capacidad del canal, con probabilidades de error cercanas a cero y, a la vez, que sea posible usarlos en la práctica.

Para medir un código respecto al límite establecido por Shannon, decimos que un código está a una **distancia δ de la capacidad para una probabilidad de error π** si el código tiene una probabilidad de error (por bloque o por símbolo, según el contexto) π y una tasa $R = C(1-\delta)$

Dada una probabilidad de error π, existen cotas que le impiden a los códigos acercarse a la capacidad, sin embargo estas cotas tienden a la capacidad cuando el largo tiende a infinito ([18]) (ver Apéndice F, para el caso del canal de borradura).

El segundo teorema de Shannon sugiere que para alcanzar una probabilidad de error arbitrariamente baja es necesario que los códigos sean suficientemente largos. En el caso de la comunicación en el espacio, dado que el decodificador debe recibir el bloque completo para poder decodificarlo y entregarle el vector de información al destino, el largo del código determina el retraso de la comunicación. Muchas veces, en la práctica, es imperativo que el retraso sea lo más corto posible (por ejemplo, en aplicaciones interactivas o de tiempo real). Además, pueden existir restricciones sobre el largo de las palabras de código

debido a diversas razones como, por ejemplo, limitaciones de memoria de los equipos. La teoría de los códigos LDPC ha permitido desarrollar muy buenos códigos pero de largos relativamente grandes, lo cual limita un poco las aplicaciones posibles.

2.6 Complejidad en la codificación: códigos lineales

El teorema de codificación de canal asegura la existencia de códigos buenos pero no dice como codificarlos de manera práctica. Si queremos intentar usar códigos aleatorios como los que usó Shannon en su demostración, podemos usar una tabla que haga la correspondencia entre los vectores de información y las palabras de código. Sin embargo, el teorema dice que si queremos una probabilidad de error arbitrariamente baja el largo tiene que ser suficientemente grande. El problema es que la cantidad de palabras de código crece exponencialmente con el largo del código (manteniendo la tasa fija) y por lo tanto el tamaño del almacenamiento requerido para la tabla de codificación también crece exponencialmente.

Una solución posible a este problema consiste en restringirse al conjunto de los **códigos lineales** que son aquellos cuyos símbolos pertenecen a cuerpos finitos y para los cuales se cumple que, si sumamos dos palabras de código, obtenemos otra palabra de código, es decir que son cerrados frente a la suma y, por lo tanto, constituyen un espacio vectorial. Esta definición hace que su codificación se pueda realizar mediante una multiplicación de una matriz de tamaño $k \times n$ por el vector de información de largo k. Por lo tanto, el tamaño del almacenamiento requerido, a lo sumo, crece en forma cuadrática con el largo del código y el tiempo de cómputo del algoritmo de codificación también crece, a lo sumo, en forma cuadrática.

Los códigos LDPC son códigos lineales y más adelante veremos que existen métodos aún más eficientes para codificarlos, más precisamente de tiempo y almacenamiento lineal en el largo del código.

Para terminar de justificar el interés por los códigos lineales resta averiguar si existen códigos en ese conjunto que alcanzan la capacidad del canal con probabilidad de error

arbitrariamente chica. El siguiente teorema, enunciado en [26], da la respuesta para la clase de canales MBIOS:

Teorema: Existen códigos lineales que permiten transmitir con probabilidad de error arbitrariamente chica por canales MBIOS con tasas hasta la capacidad del canal.

Otra forma de caracterizar un código lineal es mediante una matriz de paridad. Una **matriz de paridad** para un código C cuyo alfabeto es un cuerpo F, es una matriz $H_{(n-k)\times n}$ sobre F tal que un vector $x \in F^n$ es una palabra perteneciente al código C si y sólo si $Hx^T = 0$.

Ejemplo: **Código de Hamming**

El código $[7,4,3]$ [7] de Hamming sobre GF(2) se define mediante la matriz de paridad:

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

El rango de la matriz de paridad puede ser menor que $n-k$, en cuyo caso decimos que k/n es la **tasa de diseño** del código mientras que la verdadera tasa es $1-\frac{\text{rango}(H)}{n}$.

Por más información, consultar el Apéndice E.

2.7 Complejidad en la decodificación

El teorema de codificación de canal tampoco dice como encontrar códigos de canal que se decodifiquen eficientemente. Por ejemplo, para un código lineal, en general, la decodificación ML por bloque en el canal binario simétrico es un problema NP-completo ([1]). En el canal binario de borradura, la situación no es tan grave ya que la decodificación ML (por bloque o por símbolo) es, en general, $O\left(n^3\right)$ ([26]).

[7] Esta notación indica que el código tiene largo 7, dimensión 4 y distancia mínima 3

Existe una subclase de códigos lineales para los cuales la decodificación ML por bloque es $O\left(n^2\right)$ ([13]) y la decodificación ML por bit es $O(n)$, para una gran variedad de canales. Ambos tipos de decodificación están basadas en un mismo algoritmo genérico llamado **Sum-Product** que describimos con detalle en el Apéndice H. Sin embargo, la subclase de códigos antes mencionada tiene deficiencias en su estructura que le impiden a los códigos acercarse a la capacidad ([8]).

Como veremos en el capítulo 3, al aplicar el algoritmo de decodificación por bit (basado en el algoritmo Sum-Product) a códigos LDPC que no pertenecen a esta subclase de códigos lineales, no se obtiene una decodificación ML pero es posible acercarse a la capacidad de varios canales, con una complejidad lineal en el largo del código.

Por más detalles sobre el problema de la complejidad en la decodificación, consultar el Apéndice G.

2.8 Largo, tasa, probabilidad de error y complejidad: balances

En [26], los autores presentan un relevamiento de los distintos balances existentes entre los parámetros más importantes de los códigos de canal, en general y en algunos casos particulares.

Los parámetros más importantes para el problema de la transmisión de información son: la tasa R (que representa la eficiencia en el uso del canal), la probabilidad de error por bit o por bloque $\Pr$ (que mide la confiabilidad lograda), la complejidad de la codificación χ_E y de la decodificación χ_D (que mide el esfuerzo necesario para comunicar en forma confiable) y el largo del código n (que determina el mínimo retraso posible si hablamos de comunicación en el espacio). Idealmente, sería deseable conocer todas las tuplas $(R, \Pr, \chi_E, \chi_D, n)$ que se puede lograr pero, actualmente, la teoría está lejos de una caracterización tan completa. Sin embargo, la tarea es más sencilla cuando se ignora alguna de estas cantidades y se estudia los balances que existen entre las demás. Claramente, el

problema se vuelve trivial si no ponemos cotas en la tasa o en la probabilidad de error. Sin embargo, el problema permanece no trivial aunque no pongamos cotas en la complejidad o en el largo.

En vez de trabajar con la tasa es conveniente trabajar con la distancia (multiplicativa) a la capacidad ($\delta := 1 - \frac{R}{C}$).

Si consideramos una secuencia, en el largo n, de códigos lineales con distancia a la capacidad δ y un canal binario sin memoria[8] con capacidad C, $C > 0$, entonces se cumple que la probabilidad de error (decodificando con la regla ML) de los códigos de la secuencia tiende a cero exponencialmente en el largo del código, con un coeficiente en el exponente que tiende a cero cuando la distancia a la capacidad tiende a cero. Más precisamente se cumple que $e^{-n(E(\delta)+o_n(1))} \le \Pr_B^{ML}(n,R) \le e^{-nE(\delta)}$ donde $E(\delta) = \delta^2 C\alpha + O\left(\delta^3\right),\ \alpha > 0,\ \delta \in [0,1]$.

Por lo tanto, si se desea usar un canal eficientemente (es decir, con una distancia a la capacidad chica), el largo del código debe ser suficientemente grande para obtener una confiabilidad satisfactoria.

Si consideramos códigos lineales generales con probabilidad de error p fija, la complejidad en la decodificación ML en un canal binario sin memoria es, cuando $\delta \to 0$,

$$\chi_D(\delta, p) = 2^{O\left(1/\delta^2\right)}$$

Mientras que la complejidad de la codificación es:

$$\chi_E(\delta, p) = O\left(1/\delta^2\right)$$

En el caso particular de un canal de borradura la complejidad de la decodificación se comporta mejor ([19]) :

$$\chi_D(\delta, p) = O\left(1/\delta^4\right)$$

[8] Esta clase de canales es aún más general que la clase MBIOS, dado que no se exige simetría

Si dejamos de lado la decodificación ML y consideramos algoritmos subóptimos es posible encontrar mejores balances entre la complejidad y la distancia a la capacidad. Para la clase de códigos de canal iterativos, a la cual pertenecen los códigos LDPC, no se conoce exactamente el balance entre complejidad y distancia a la capacidad, sin embargo, se conjetura ([26]) que:

$\chi_D(\delta, p) = \chi_E(\delta, p) = K\frac{1}{\delta}\log\frac{1}{\delta}$, para una constante K, pequeña, que depende del canal

Para el caso particular del canal de borradura, está demostrado que:

$$\chi_D(\delta, p) = \chi_E(\delta, p) = K\log\frac{1}{\delta}$$

La conjetura anterior y el resultado para el canal de borradura justifican el interés por el estudio de los sistemas de codificación iterativos y en particular por los códigos LDPC, dado que permiten un muy buen balance (se cree que es el mejor balance posible) entre complejidad y distancia a la capacidad.

3 CÓDIGOS LDPC

En el capítulo anterior, vimos que el teorema de codificación de canal asegura la existencia de códigos con tasas cercanas a la capacidad del canal y con probabilidad de error arbitrariamente baja. En este capítulo, presentamos un estudio de los códigos LDPC. Estos códigos permiten acercarse a la capacidad de varios canales con probabilidad de error arbitrariamente baja con algoritmos de codificación y de decodificación con complejidad muy baja.

3.1 Definición

Los códigos LDPC son códigos lineales cuya propiedad esencial es la de tener por lo menos una matriz de paridad de baja densidad, es decir con "pocos" elementos distintos de cero. Formalmente, decimos que una **secuencia (en el largo n) de códigos es LDPC** si cada código tiene por lo menos una matriz de paridad en la cual la cantidad de elementos distintos de cero es $O(n)$.

Es debido a este tipo de estructura que los algoritmos de codificación y de decodificación tienen complejidad lineal en el largo del código. Si esos mismos algoritmos se usaran con matrices densas, es decir, matrices que tienen una cantidad $O(n^2)$ de elementos mayores que cero, entonces los algoritmos tendrían complejidad de orden cuadrático en el largo del código.

En este proyecto, consideraremos únicamente códigos LDPC binarios. Sin embargo, es posible usar códigos LDPC en cuerpos más grandes y obtener buenos resultados (por más información ver [9] y [5]).

3.2 Relación con grafos

Es muy importante el hecho de poder representar los códigos lineales mediante grafos bipartitos dado que los algoritmos eficientes de decodificación se basan en esta representación.

Si consideramos la caracterización de los códigos lineales mediante el sistema de ecuaciones $Hx^T = 0$, vemos que los elementos distintos de cero de cada fila de la matriz de paridad determinan cuales posiciones de las palabras de código pertenecen a cada ecuación. Por lo tanto, si definimos el conjunto V de los índices de las palabras de código y el conjunto C de los índices de las ecuaciones definidas por H, tenemos una relación entre elementos de C y elementos de V. Esta relación se puede representar mediante un grafo bipartito y la matriz de paridad se puede ver como una matriz de adyacencia del grafo. Este tipo de representación de los códigos lineales se llama **grafo de Tanner** ([36]).

Por ejemplo, consideremos la matriz de paridad del código de Hamming siguiente:

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

El grafo bipartito correspondiente se muestra en la Figura 3:

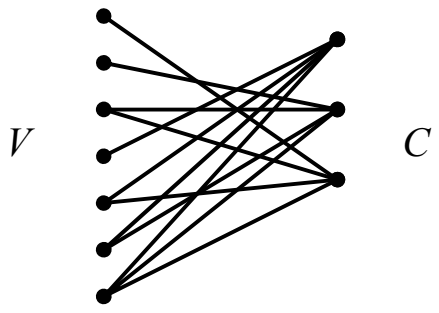

Figura 3

Los elementos de V se denominan **variables o nodos izquierdos** y los elementos de C se denominan **checks o nodos derechos**.

Los grafos que usó originalmente Gallager eran **regulares**, es decir que todos los nodos de la izquierda tenían el mismo grado y los de la derecha también. Sin embargo, como veremos más adelante, el uso de grafos **irregulares** permite, en general, acercarse mucho más a la capacidad.

El análisis de los códigos LDPC se hace, por lo general, sobre familias de códigos (o de grafos) que se especifican dando la distribución de los grados de los nodos derechos e izquierdos. La distribución puede ser dada del punto de vista de los nodos o de las aristas, es decir que para cada grado se especifica cuantos nodos hay de dicho grado o cuantas aristas hay conectadas a nodos de dicho grado, respectivamente. En muchos casos, resulta conveniente utilizar las distribuciones en forma independiente del largo del código y para eso se dan en forma normalizada.

Sea Λ_i la cantidad de variables de grado i y P_i la cantidad de checks de grado i. Tenemos entonces, las funciones generatrices asociadas para las **distribuciones del punto de vista de los nodos**:

$$\Lambda(x) \coloneqq \sum_{i=1}^{\infty} \Lambda_i x^i \quad \mathrm{P}(x) \coloneqq \sum_{i=1}^{\infty} \mathrm{P}_i x^i$$

Por lo tanto, se cumple que $\Lambda(1) = n$ y $\mathrm{P}(1) = n(1-r)$, donde r es la tasa de diseño. Para que el grafo sea válido es necesario que la cantidad total de aristas que inciden en cada lado sea igual, es decir $\Lambda'(1) = \mathrm{P}'(1)$.

Las **distribuciones normalizadas del punto de vista de los nodos** se definen como:

$$L(x) \coloneqq \Lambda(x)/\Lambda(1) \text{ y } R(x) \coloneqq \mathrm{P}(x)/\mathrm{P}(1)$$

Para el análisis asintótico es más conveniente la perspectiva de las aristas. Sea λ_i la fracción de aristas que están conectadas a variables de grado i y ρ_i la fracción de aristas conectadas a checks de grado i, las **distribuciones normalizadas del punto de vista de las aristas** están dadas por:

$$\lambda(x) = \sum_i \lambda_i x^{i-1} := \frac{\Lambda'(x)}{\Lambda'(1)} \quad \rho(x) = \sum_i \rho_i x^{i-1} := \frac{P'(x)}{P'(1)}$$

(Λ, P) y (λ, ρ, n) contienen información equivalente y ambas definen una familia de grafos cuyos nodos tienen grados de acuerdo a las distribuciones. Para el análisis asintótico, resulta útil fijar un par (λ, ρ) y hacer tender el largo n a infinito, lo cual define una secuencia de familias de códigos.

Es fácil de ver que el grado promedio de los nodos derechos (es análogo para los izquierdos) se calcula como:

$$r_{avg} = \left(\int_0^1 \rho(x) dx \right)^{-1}$$

Si una secuencia de códigos lineales, de tasa de diseño r fija, se especifica mediante una distribución normalizada, entonces para un código de la secuencia de largo n, la cantidad de aristas (o sea, de elementos distintos de cero en la matriz de paridad) es $n(1-r)r_{avg} = O(n)$. Por lo tanto, la secuencia es LDPC.

En la teoría, muchas veces se trabaja con familias de grafos que tienen determinados pares de distribuciones (λ, ρ) de largo n, esas familias se denotan $C(n, \lambda, \rho)$. Estas familias incluyen todos los grafos bipartitos de n nodos de variable, cuyas aristas tienen grados según las distribuciones (λ, ρ) y, en particular, incluyen multigrafos. La correspondencia de los multigrafos con matrices de paridad es la siguiente: cuando dos nodos están conectados por una cantidad impar de aristas, hay un 1 en la posición correspondiente de la matriz de paridad y, de lo contrario, hay un 0.

3.3 Decodificación eficiente para todos los canales

La decodificación eficiente de los códigos LDPC se logra mediante algoritmos iterativos que hacen un trabajo lineal en la cantidad de aristas y dado que la cantidad de aristas es $O(n)$, la complejidad de estos algoritmos resulta $O(n)$. En general, estos algoritmos se pueden ver como **algoritmos de envío de mensajes**, llamados así porque en cada iteración se envía un mensaje desde cada check a cada variable relacionada y luego un mensaje de cada variable a cada check relacionado. Uno de los más importantes de esta clase es el algoritmo de **Belief Propagation** que puede ser usado independientemente del canal. En algunos canales, existen otros algoritmos que se pueden ver como variantes del anterior y que permiten reducir la cantidad de cómputo necesario a cambio de una menor confiabilidad (ver [26], por ejemplo).
El algoritmo de Belief Propagation fue propuesto por Gallager ([9]), pero luego fue analizado como una instancia del algoritmo genérico **Sum-Product** (ver Apéndice H). A continuación damos una intuición de cómo funciona el algoritmo.

Los mensajes que se pasan por las aristas son funciones de probabilidad. Más precisamente, el mensaje enviado desde una variable v hacia un check c es la función de probabilidad del valor enviado por el codificador para la variable v, dado el valor observado de la variable (es decir, el valor que se obtuvo a la salida del canal) y dados todos los mensajes que fueron enviados hacia v en la iteración anterior salvo el que provino de c. El mensaje enviado desde c hacia v es la función de probabilidad del valor enviado por el codificador para la variable v, dados todos los mensajes enviados hacia c en la iteración anterior salvo el que provino de v.

Bajo ciertas hipótesis de independencia entre las variables aleatorias, es fácil encontrar las fórmulas[9] para los mensajes de salida combinando adecuadamente los mensajes de entrada. Para que se cumpla la hipótesis de independencia todo a lo largo del algoritmo, es necesario que el grafo no tenga ciclos. En ese caso, este algoritmo implementa la regla de máxima

[9] Por más detalles sobre la derivación de las reglas, consultar el Apéndice H.

verosimilitud (ver Apéndice H). Pero dado que los códigos sin ciclos no permiten alcanzar la capacidad ([8]), la idea es usar las mismas fórmulas sobre grafos con ciclos e intentar obtener un buena aproximación de la regla ML.

Dado que los mensajes son funciones de probabilidad, en principio, habría que mandar un vector con las probabilidades de que la variable tome cada valor posible. Dado que la suma de las probabilidades es 1 y tenemos sólo dos valores posibles, se puede representar la función de probabilidad mediante el cociente $\frac{\Pr(x=0)}{\Pr(x=1)}$ y, tal como sugirió Gallager ([9]), es mejor tomar el logaritmo del cociente anterior. El logaritmo de este cociente se denomina logaritmo del cociente de verosimilitud (**log-likelihood ratio, llr)**. Cuando hay total certidumbre sobre el valor de x, el llr es $\pm\infty$ y, por lo contrario, cuando hay total incertidumbre el valor es 0.

Usando la representación logarítmica, el **algoritmo de Belief Propagation** queda de la forma siguiente ([31]):

Entrada: una palabra de código recibida con errores, el grafo G que representa al código al cual pertenece la palabra, el canal usado

Salida: una estimación de la palabra de código enviada (posiblemente, con algunos bits en error, es decir, con el valor 'e')

1. Se inicializa el contador de iteración, $l := 0$.
2. Se calcula el mensaje que cada variable v le envía a cada check c, que denotamos $m_{vc}^{(l)}$, mediante la fórmula:

$$m_{vc}^{(l)} := \begin{cases} m_v, & \text{si } l = 0 \\ m_v + \sum_{c' \in C_v \setminus \{c\}} m_{c'v}^{(l-1)}, & \text{si } l \geq 1 \end{cases}$$

donde C_v representa al conjunto de vecinos de v en G y m_v representa al llr inicial basado únicamente en el valor recibido y en el canal[10].

[10] Si el algoritmo se usa sobre un canal de borradura, los mensajes iniciales posibles son $\{-\infty, 0, +\infty\}$, dado que cuando se recibe un 1 o un 0 hay certidumbre total y cuando se recibe un ? hay total incertidumbre. En el

3. Se calcula el mensaje que cada check c envía a cada variable vecina v, que denotamos $m_{cv}^{(l)}$ mediante la fórmula:

$m_{cv}^{(l)} = \ln \frac{1+\prod_{v' \in V_c \setminus \{c\}} \tanh\left(m_{v'c}^{(l)}/2\right)}{1-\prod_{v' \in V_c \setminus \{c\}} \tanh\left(m_{v'c}^{(l)}/2\right)}$ donde V_c representa al conjunto de vecinos de c en G

4. $l := l+1$
5. Se calcula para cada variable, el llr que permite estimar su valor:

$$llr_v := m_v + \sum_{c' \in C_v} m_{c'v}^{(l)}$$

6. Si se cumple el criterio de parada (ver más adelante), ir a 7. Sino volver a 2.
7. Calcular la estimación de la palabra de código enviada (ver más adelante).

Si el algoritmo se aplica sobre un grafo sin ciclos, el criterio de parada es que hayan pasado dos mensajes por cada arista (uno en cada sentido). El criterio de parada a tomar cuando el algoritmo se usa con el canal de borradura, se presenta en la sección siguiente. En los demás casos, usualmente se usa el criterio siguiente: si llr_v calculado en el paso 6 es positivo, se asume el valor 1 para la variable correspondiente, sino se asume el valor -1; luego se verifica si la palabra estimada es una palabra de código; si es una palabra de código el algoritmo se detiene; si se alcanzó una cantidad máxima de iteraciones también se detiene el algoritmo.

Como calcular la estimación de la palabra de código enviada por un canal de borradura, se ve en la sección siguiente. En los demás casos, se puede:

- devolver todo el bloque en error si se llegó a la cantidad máxima de iteraciones sin obtener una palabra de código
- devolver la palabra que se obtiene tomando para cada variable el valor 1 si el llr_v correspondiente es positivo y sino -1

---

caso del $BSC(p)$, el mensaje incial es $\ln(1-p) - \ln p$ si el valor recibido es 0 sino, si el valor recibido es 1, es el negativo de esta expresión.

- devolver la palabra que se obtiene tomando para cada variable el valor 1 si el llr_v correspondiente es positivo y sino -1; siempre y cuando el valor absoluto del llr_v sea mayor que un cierto nivel determinado y sino se toma el valor 'e' para la variable correspondiente.

Una característica importante de este algoritmo es la posibilidad de realizar los cálculos de cada nodo en forma paralela, dado que los nodos no interactúan con sus pares cuando se calculan los mensajes (por más detalles ver, por ejemplo, [9]).

3.4 Decodificación eficiente para el canal de borradura

Cuando se utiliza el algoritmo Belief Propagation para un canal de borradura los mensajes posibles enviados a lo largo del algoritmo pertenecen al conjunto $\{-\infty, 0, +\infty\}$. Esta particularidad, hace que se pueda expresar el algoritmo en forma más sencilla.

Algoritmo **Belief Propagation para el canal de borradura (**[26]**)**:
Entrada: una palabra de código recibida con borraduras, el grafo G que representa al código al cual pertenece la palabra
Salida: una aproximación $\hat{x}^n$ de la palabra de código enviada

1. A cada nodo de variable se le asocia su valor recibido.
2. Cada nodo de variable que no está borrado propaga su valor hacía sus checks vecinos. En cada check se hace la suma (modulo 2) de los valores que llegan por las aristas y se guarda el valor de la suma parcial en una celda de memoria asociada con cada check. Luego se borran las aristas a través de las cuales fueron enviados mensajes.
3. Se propaga hacia las variables el valor de la suma parcial de los checks que tienen grado 1 (o sea, que recibieron el valor de todas las variables que participan en la ecuación menos una, por lo tanto la faltante queda determinada). El valor de las variables que reciben valores por alguna de sus aristas queda determinado por el valor recibido y se borran las aristas por las cuales se enviaron valores.
4. Si, en el paso 3, se enviaron mensajes, volver a 2.

5. En $\hat{x}^n$ se devuelven los valores asociados a cada variable (algunos pueden estar determinados y otros borrados)

Dado que las aristas se usan a lo sumo una vez, el algoritmo tiene complejidad lineal en la cantidad de aristas, por lo tanto, $O(n)$ para los códigos LDPC ([14],[26]).

Belief Propagation para el canal de borradura da el mismo resultado que se obtiene resolviendo el sistema de ecuaciones definido por la matriz de paridad y por las variables borradas (que se toman como incógnitas), de la siguiente forma ([26]):

1. Resolver todas las ecuaciones en las cuales haya una sola incógnita.
2. Actualizar el sistema a resolver utilizando los valores hallados.
3. Si quedan ecuaciones con una sola incógnita volver a 1 sino FIN.

A la luz del algoritmo de decodificación, daremos una intuición, basada en [15], de porqué los códigos irregulares permiten mejores resultados que los códigos regulares. Supongamos que queremos construir un código con un grafo regular. Es conveniente considerar al proceso de construcción del código como un juego en el cual los jugadores son las variables y los checks. Los jugadores intentan determinar la cantidad adecuada de aristas. Una restricción en el juego es que los jugadores deben estar de acuerdo en la cantidad total de aristas. Desde el punto de vista de las variables, es mejor tener muchas aristas ya que pueden tener más chances de estar en una ecuación que las pueda corregir, mientras que desde el punto de vista de las ecuaciones es mejor tener pocas aristas así tienen más chances de tener pocas variables en error y así poder corregirlas. Estos dos requerimientos competitivos deben ser balanceados adecuadamente. Si permitimos un amplio rango de grados en los nodos de variable (es decir, permitimos que el grafo sea irregular del lado izquierdo) entonces las variables de grado alto van a tender a corregirse más rápidamente y permitiendo luego a las variables de grado más bajo corregirse y así sucesivamente. Este efecto de ola se observa en la práctica y de hecho se observa empíricamente que las variables que quedan sin corregir, al final del algoritmo, son de grado bajo.

3.5 Simplificaciones básicas para el análisis en el canal de borradura

Es fácil de ver, en la versión de Belief Propagation para el canal de borradura, que el éxito en la recuperación de variables borradas sólo depende del patrón de las borraduras y no de la palabra enviada. Por lo tanto, es posible analizar la probabilidad de error del decodificador asumiendo que se envió la palabra toda compuesta de ceros. Esto también es válido en el caso general de Belief Propagation ([26]).

Para simplificar el análisis se estudia la probabilidad de error promedio de familias de códigos debido a la gran dificultad de estudiarlos en forma individual. Esta simplificación tiene sentido y utilidad debido a que la probabilidad de error de los códigos individuales está concentrada alrededor del promedio en forma exponencial, tal como lo dice el siguiente teorema ([14],[25]):

Teorema de Concentración para el canal de borradura[11]: Para cualquier $\delta > 0$, existe un $\alpha(\delta) > 0$ tal que $\Pr\left\{\left|\mathrm{P}_b^{IT}(G,\varepsilon) - \mathrm{E}_{C(n,\lambda,\rho)}\left[\mathrm{P}_b^{IT}(G,\varepsilon)\right]\right| > \delta\right\} \le e^{-\alpha(\delta)n}$ para todo $G \in C(n,\lambda,\rho)$. Donde $P_b^{IT}(G,\varepsilon)$ representa a la probabilidad de error por bit, bajo decodificación iterativa, del código representado por el grafo G cuando se usa un canal de borradura con parámetro ε y $C(n,\lambda,\rho)$ es la familia de grafos con distribuciones (λ,ρ) y largo n.

Por lo tanto, si elegimos al azar un código de la familia entonces, con alta probabilidad, su probabilidad de error por bit estará cerca del promedio.

3.6 Análisis asintótico para el canal de borradura

Informalmente, el objetivo del análisis asintótico que vamos a presentar es el siguiente: dada una familia de códigos definida por un par de distribuciones (λ,ρ), la familia de

[11] Este resultado se generaliza para otros canales, ver [25],[27]

canales[12] $BEC(\varepsilon)_{\varepsilon\in[0,1]}$ y el algoritmo Belief Propagation queremos determinar el parámetro crítico del canal llamado umbral, tal que casi todos los códigos de la familia, tomando un largo suficientemente grande, permitan lograr una transmisión de información arbitrariamente confiable a condición de utilizar un canal cuyo parámetro esté por debajo del umbral. A la inversa, si se intenta usar un canal cuyo parámetro esté por encima del umbral entonces, la mayoría de los códigos de la familia no va a permitir transmitir confiablemente.

Un par de distribuciones (λ, ρ) determina la tasa de diseño r [13] de los códigos de la familia (los códigos de la familia tiene tasa real de por lo menos r). Por lo tanto, en vez de fijar el canal e intentar lograr códigos con tasa lo más cercana a la capacidad posible, se fija la tasa y se buscan códigos que soporten al peor canal posible (el límite siendo el que impone el teorema de codificación de canal y en el caso del canal de borradura es $C^{-1}(r) \coloneqq 1-r$).

Una vez que tengamos una manera de caracterizar ese umbral, la idea será encontrar pares de distribuciones (o secuencias de pares de distribuciones) que puedan tener un umbral lo más cercano posible al límite de Shannon

La herramienta fundamental para encontrar dicho umbral, es una fórmula recursiva que describe la evolución, a lo largo de las iteraciones, de la proporción de mensajes de valor 0 (incertidumbre total) enviados desde las variables hacia los checks, en la versión original de Belief Propagation. La fórmula se denomina **density evolution** y, para el canal de borradura[14] ([14]), tiene la forma:

$$x_l = x_0 \lambda(1-\rho(1-x_{l-1})),\ l \geq 1,\ x_0 = \varepsilon\text{, donde } l \text{ es el número de iteración}$$

---

[12] Estas ideas se extienden a otros canales, ver [25]

[13] $r(\lambda,\rho) \coloneqq 1 - \dfrac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}$

[14] Existe una generalización de la fórmula para la clase de canales MBIOS [25],[27]

Esta fórmula es válida siempre y cuando la vecindad de cada nodo sea un árbol hasta la capa $2l$.

Se demuestra ([26]) que si x_l tiende a 0 entonces también tiende a 0 la probabilidad de error por bit y vice-versa, cuando l tiende a infinito. Por lo tanto, el objetivo es encontrar bajo qué condiciones x_l tiende a 0 cuando l tiende a infinito.

El problema es que cuando hacemos tender l a infinito, en un grafo finito con ciclos, a partir de un momento la fórmula no va a ser válida. Sin embargo, observando que, con probabilidad $1-O(1/n)$ ([27]), la vecindad de un nodo es un árbol, si hacemos tender n a infinito podemos considerar que el grafo no tiene ciclos y la fórmula de density evolution es válida para cualquier iteración[15]. Más precisamente, tenemos el siguiente resultado de convergencia ([25]) que nos dice que podemos tener una probabilidad de error esperada tan cercana como se desee a la probabilidad de error esperada, cuando el largo es infinito, a condición de tomar n suficientemente grande.

Teorema de convergencia para el canal de borradura[16]

Existe una constante β tal que:

$$\left|\mathrm{E}_{C(n,\lambda,\rho)}\left[\mathrm{P}_b^{IT}(G,\varepsilon)\right]-\mathrm{E}_{C(\infty,\lambda,\rho)}\left[\mathrm{P}_b^{IT}(G,\varepsilon)\right]\right|\le\frac{\beta}{n}$$

Se demuestra la siguiente propiedad de **monotonicidad** ([26]):

Dado un par distribuciones (λ,ρ) y $\varepsilon\in[0,1]$, si $x_l\to 0$ cuando $x_0=\varepsilon$ y $l\to\infty$ entonces cuando $x_0=\varepsilon',\varepsilon'\le\varepsilon$ tenemos que $x_l\to 0$ también.

[15] En ese caso límite, el decodificador es equivalente a la regla de máxima verosimilitud

[16] Este resultado se generaliza para otros canales, ver [25],[27]

Entonces tiene sentido definir el concepto de umbral como el "peor" parámetro del canal "soportado" por la familia de códigos. Más precisamente, el **umbral** $\varepsilon^{IT}(\lambda,\rho)$ asociado al par de distribuciones (λ,ρ) se define como ([26]):

$$\varepsilon^{IT}(\lambda,\rho) := \sup\left\{\varepsilon \in [0,1] : x_l(\varepsilon) \overset{l\to\infty}{\to} 0\right\}$$

Esta definición no resulta fácil de manejar para determinar, en la práctica, el umbral de un par de distribuciones. Sin embargo, se demuestra ([26]) que la evolución de la probabilidad de error siempre converge a un punto fijo. Por lo tanto, si definimos $f(y,x) := y\lambda(1-\rho(1-x))$ obtenemos las siguientes **caracterizaciones de punto fijo** del umbral de un par de distribuciones (λ,ρ) ([26]):

$1)\varepsilon^{IT}(\lambda,\rho) := \sup\{\varepsilon \in [0,1] : x = f(x,\varepsilon)$ no tiene solución en $(0,\varepsilon]\}$
$2)\varepsilon^{IT}(\lambda,\rho) := \inf\{\varepsilon \in [0,1] : x = f(x,\varepsilon)$ tiene solución en $(0,\varepsilon]\}$

Esta caracterización permite hallar el umbral en forma gráfica ([26]), evaluando $f(\varepsilon,x)-x$ como una función de x, $x \in [0,1]$: el umbral es el máximo ε tal que la curva sea negativa.

Con este método gráfico, es posible determinar el umbral con cualquier grado de exactitud. En algunos casos, también es posible determinar el umbral en forma analítica (ver [26]).

3.7 Secuencias que alcanzan la capacidad del canal de borradura

Consideremos al par de distribuciones (λ,ρ) con tasa de diseño $r = r(\lambda,\rho)$ y con umbral ε^{IT}. Sea $\delta := \dfrac{1-\varepsilon^{IT}-r}{1-\varepsilon^{IT}}$, entonces decimos que el par (λ,ρ) alcanza una fracción $(1-\delta)$ de la capacidad ($r = (1-\delta)(1-\varepsilon^{IT})$) o que **el par tiene una distancia δ a la capacidad**. De la definición se desprende que $\delta \geq 0$. En otras palabras, δ es la distancia a la capacidad del canal $BEC\left(\varepsilon^{IT}\right)$ de los códigos de la familia definida por el par de distribuciones cuando el largo tiende a infinito.

El siguiente teorema asegura que ningún par fijo (λ, ρ) puede tener distancia cero a la capacidad:

Teorema: Dado un par de distribuciones (λ, ρ) con tasa $r \in (0,1)$ y con grado derecho promedio r_{avg} entonces se cumple que $\delta(\lambda, \rho) \geq \frac{r^{r_{avg}-1}(1-r)}{1+r^{r_{avg}-1}(1-r)}$.

El teorema anterior indica que para hacer tender la distancia a 0, es necesario hacer tender el grado promedio derecho a infinito.

Por lo tanto, lo mejor que se puede esperar es construir **secuencias de pares de distribuciones** $\{(\lambda^{(N)}, \rho^{(N)})\}_{N \geq 1}$ que alcancen la capacidad de un cierto canal $BEC(\varepsilon)$, más precisamente, esto se define como:

$\lim_{N \to \infty} r\left(\lambda^{(N)}, \rho^{(N)}\right) = 1 - \varepsilon$ y

$\lim_{N \to \infty} \delta\left(\lambda^{(N)}, \rho^{(N)}\right) = 0$

Lo anterior es equivalente a requerir que la distancia a la capacidad del canal $BEC\left(\varepsilon^{IT}\right)$ tienda a 0 y que ε^{IT} tienda a ε.

En [20], se describe un método para construir tales secuencias. En particular, destacamos dos secuencias.

La primera secuencia que se encontró (antes de conocer el método descrito antes mencionado) fue propuesta en [14] y es la base de los códigos Tornado ([3]), se llama **secuencia Heavy –Tail Poisson** y se define mediante las distribuciones siguientes:

$\lambda^N(x) = \frac{1}{H(N-1)} \sum_{k=1}^{N-1} \frac{x^k}{k}$, donde $H(N) := \sum_{i=1}^{N} 1/i$

$\rho_\alpha^N(x) = \rho_\alpha(x) := e^{-\alpha} \sum_{i=0}^{\infty} \frac{\alpha^i x^i}{i!}$

Los parámetros N y α definen la tasa y la distancia a la capacidad del par de distribuciones (o, equivalentemente, la tasa y el umbral). Si se desea construir una secuencia que alcance la capacidad del canal $BEC(\varepsilon)$, se elige $\alpha(N) = \frac{H(N-1)}{\varepsilon}$, lo cual hace que:

$$r(N) = 1 - \varepsilon + O(1/N)$$
$$\delta(N) \leq O(1/N)$$

Para esta secuencia, es fácil de ver ([26]) que $\varepsilon^{IT}(N) \geq \varepsilon \ \forall N$, por lo tanto, cualquier elemento de la secuencia es apto para ser usado en el canal $BEC(\varepsilon)$. En la Figura 4 , se muestra un ejemplo de secuencia que alcanza la capacidad del canal $BEC(0.4)$.

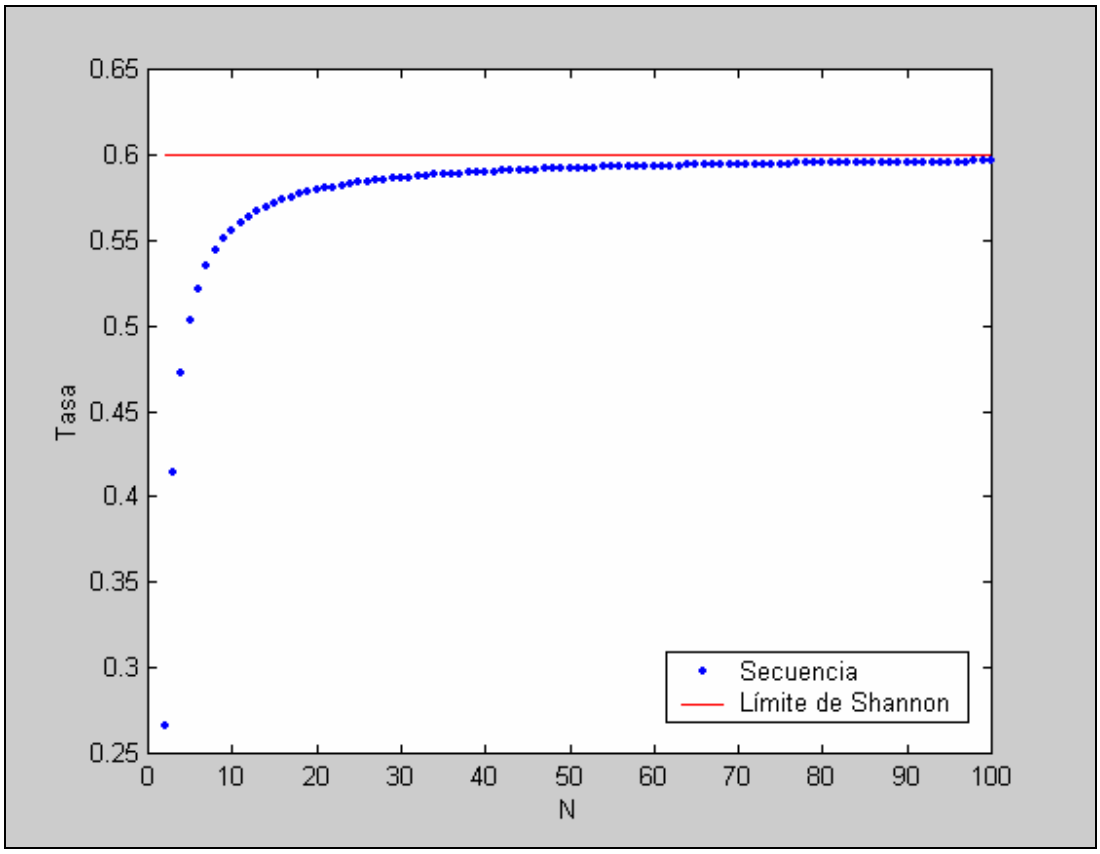

Figura 4: evolución de la tasa para una secuencia Heavy-Tail Poisson que alcanza la capacidad del canal BEC(0.4)

También es posible eligiendo α adecuadamente, fijar la tasa r y construir una secuencia cuyo umbral tienda a $1-r$ tal como lo ilustra la Figura 5 cuando la tasa es 0.6.

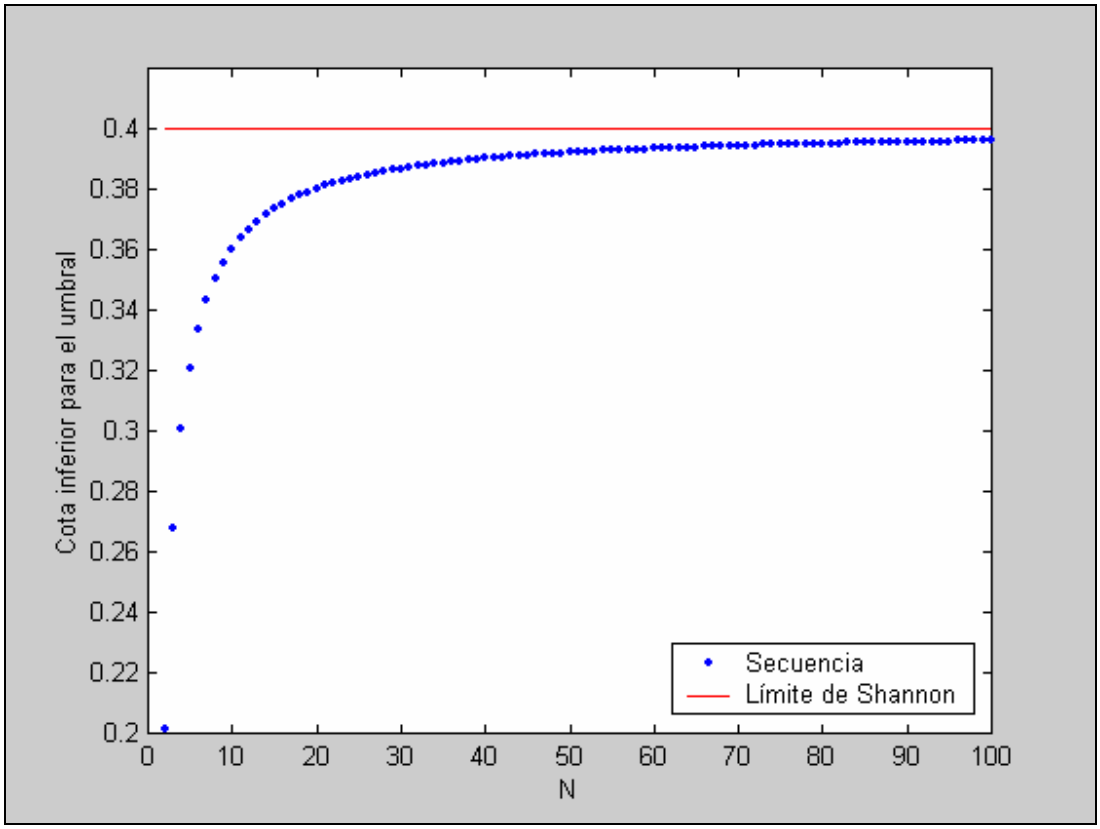

Figura 5: evolución del umbral[17] para una secuencia Heavy-Tail Poisson de tasa fija 0.6 que alcanza la capacidad

La segunda secuencia se llama **Right-regular**, dado que los checks tienen todos el mismo grado, y, como veremos más adelante, es óptima respecto al balance entre distancia a la capacidad y complejidad. La secuencia se define mediante las siguientes distribuciones:

[17] Los valores graficados corresponde a cotas inferiores para las umbrales de los pares de la secuencia

$$\lambda_{\alpha}^{N}(x) \coloneqq \alpha \frac{\sum_{k=1}^{N-1} \binom{\alpha}{k} (-1)^{k+1} x^{k}}{\alpha - N \binom{\alpha}{N} (-1)^{N+1}}$$

$$\rho_{a}(x) \coloneqq x^{\frac{1}{\alpha}}$$

En este caso, si se desea construir una secuencia que alcance la capacidad del canal $BEC(\varepsilon)$, se elige $\alpha(N) = \dfrac{\ln \dfrac{1}{1-\varepsilon}}{\ln N}$, lo cual hace que:

$$r(N) = 1 - \varepsilon + O\left(1/\ln N\right)$$
$$\delta(N) \leq O\left(1/N\right)$$

Al igual que en el caso anterior, también es posible, eligiendo α adecuadamente, fijar la tasa r y construir una secuencia cuyo umbral tienda a $1-r$.

Dado que se pueden construir una infinidad de secuencias, es necesario algún criterio para compararlas. Una manera importante de compararlas es respecto al balance existente entre distancia a la capacidad y complejidad. En base a este criterio, en [29], se muestra que la secuencia Right-Regular es óptima (en un sentido de optimalidad que ahí se define) respecto a este balance. La secuencia Heavy-Tail Poisson es óptima en un sentido más débil definido en [20]. La figura, muestra el balance para las dos distribuciones. Observamos la dependencia logarítmica de la complejidad con la distancia a la capacidad (el eje de las equis está en escala logarítmica) de ambas distribuciones, sin embargo el balance es mejor para la distribución Right-Regular.

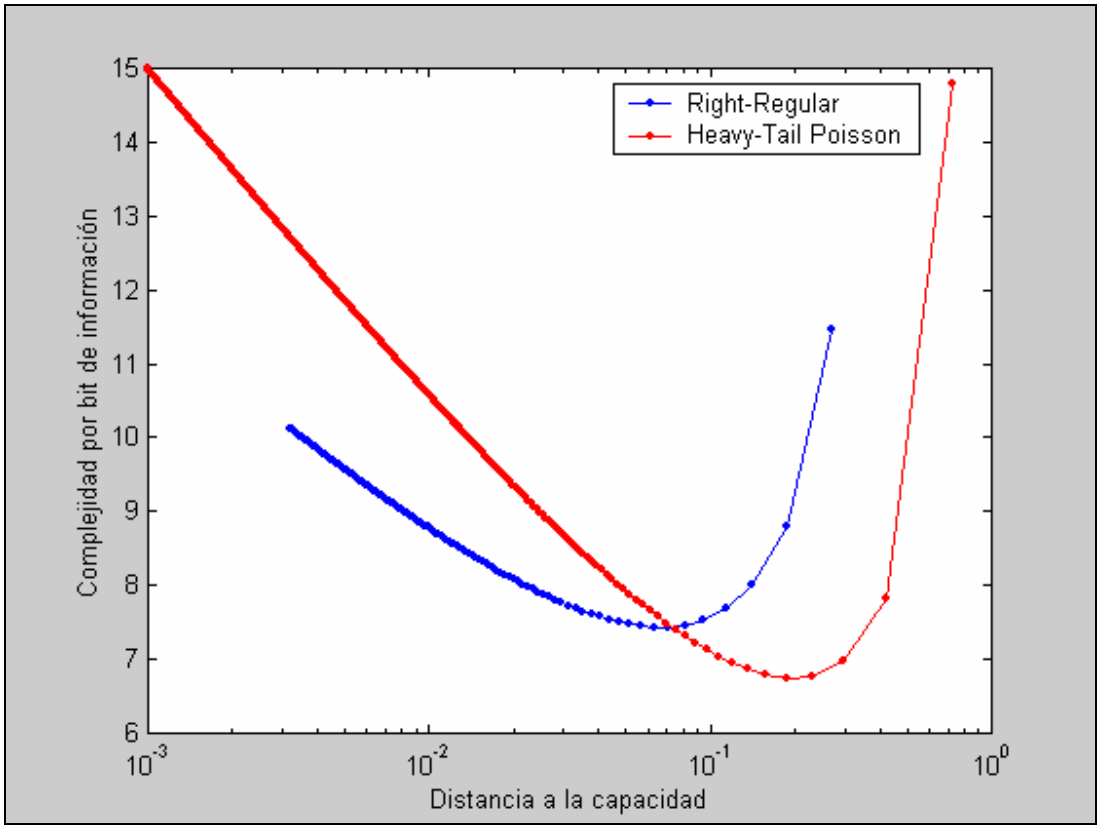

Figura 6: balance entre distancia a la capacidad[18] y complejidad para secuencias Right-Regular y Heavy-Tail Poisson que alcanzan la capacidad del canal BEC(0.5)

3.8 Optimización asintótica para el canal de borradura

Recapitulando, tenemos, para el canal de borradura, secuencias que nos permiten ubicarnos (de manera óptima) tan cerca de la capacidad como queremos a cambio de mayor complejidad. Puede haber otros criterios para seleccionar distribuciones, por ejemplo, dado que la distancia a la capacidad considerada es cuando el límite tiende a infinito, se podría tomar en cuenta la velocidad de convergencia al caso límite. Para eso es posible usar un método alternativo, basado en optimización lineal heurística, para hallar buenas distribuciones con ciertas restricciones. En efecto, cuanto más grande es el grado máximo

[18] Los valores usados para la distancia a la capacidad son cotas superiores que se vuelven ajustadas a medida que N tiende a infinito

de las distribuciones más lenta es la convergencia al comportamiento de largo infinito ([26]). En el caso de las secuencias Right-Regular, el máximo grado crece como $1/\delta$.

Para otros canales, este método se generaliza ([27]) y resulta imprescindible dado que no se conocen secuencias de distribuciones que alcancen la capacidad. Este método también permite imponer otro tipo de restricciones, en los coeficientes y los grados de las distribuciones, de utilidad práctica. Por ejemplo, como se verá más adelante, cuando se codifica en forma sistemática, puede ser conveniente poner las variables de grado bajo (en particular las de grado 2) en la parte de la paridad porque, intuitivamente y en la práctica, tienen más chances de no ser recuperadas si se borran.
Para el canal de borradura, el método fue originalmente propuesto por Luby et al. ([14]) sin embargo lo presentaremos en la forma dada en [26].

En esencia, el procedimiento consiste en ir optimizando la distribución izquierda dada una distribución de derecha y vice-versa, alternando estos dos problemas hasta encontrar un par suficientemente bueno. Es una técnica heurística y no hay pruebas de convergencia. En los dos problemas, se intenta maximizar la tasa dado un ε mínimo que el par debe soportar.
Por lo tanto, dados una distribución derecha ρ y una cota inferior ε para el umbral, tenemos que encontrar una distribución izquierda λ tal que se satisfaga la condición (que asegura que el par tiene umbral de por lo menos ε):
$\varepsilon\lambda(1-\rho(1-x))-x \le 0, x \in [0,1]$.
Esta condición representa un conjunto infinito de restricciones lineales en los coeficientes de λ. En la práctica se discretiza x eligiendo una precisión suficientemente chica. Es necesario fijar un conjunto finito de grados permitidos en el polinomio lambda para tener un conjunto finito de variables. La tasa de diseño es una función creciente en $\sum \lambda_i \frac{1}{i}$ (para ρ fija), por lo tanto es posible tomar esta expresión como función objetivo a maximizar. En resumen, queda el siguiente programa lineal:

$$\max \sum_{i \in G} \lambda_i \frac{1}{i}$$

$$s.a.$$

$$\lambda_i \geq 0$$

$$\sum_{i \in G} \lambda_i = 1$$

$$\varepsilon \sum_{i \in G} \lambda_i (1-\rho(1-x))^{i-1} - x \leq 0, \;\; x = k(1/N), \; k \in [0, N]$$

siendo G el conjunto de grados elegido y N el parámetro de discretización.

Una vez resuelto el programa lineal, intercambiando los roles de las variables y de los checks, se obtiene la siguiente restricción equivalente (lineal en los coeficientes de ρ):

$$(1-x) - \rho(1-\varepsilon\lambda(x)) \leq 0, \;\; x \in [0,1]$$

Se plantea un programa lineal análogo al anterior y se resuelve. Y así sucesivamente hasta encontrar un par satisfactorio (es decir, con tasa suficientemente cercana a $1-\varepsilon$).

En [26], los autores sugieren que, en la práctica, para lograr buenas distribuciones, alcanza con fijar un grado promedio derecho r_{avg}, tomar una distribución derecha concentrada a la derecha, es decir de la forma $\rho(x) = \frac{\mathrm{r}\left(\mathrm{r}+1-r_{avg}\right)}{r_{avg}} x^{\mathrm{r}-1} + \frac{r_{avg} - \mathrm{r}\left(\mathrm{r}+1-r_{avg}\right)}{r_{avg}} x^{\mathrm{r}-1}$, $\mathrm{r} = \lfloor r_{avg} \rfloor$ y luego optimizar la distribución izquierda. Finalmente, los autores sugieren repetir el procedimiento con diferentes grados promedios derechos hasta encontrar un par satisfactorio.

3.9 Análisis de largo finito para el canal de borradura

El análisis asintótico se basa en las dos propiedades siguientes

1. La probabilidad de error de un código particular está concentrada en el promedio del conjunto de códigos con las mismas distribuciones y el mismo largo. La concentración es exponencial en el largo del código

2. La esperanza de la probabilidad de error de un par de distribuciones tiende con el largo del código a la esperanza de la probabilidad de error del código con las mismas distribuciones pero con largo infinito. La convergencia es de orden por lo menos $1/n$, siendo n el largo de código.

Esto sugiere lo siguiente: si fijamos un largo n moderado y consideramos elementos individuales de $C(n,\lambda,\rho)$, el comportamiento de esos códigos individuales puede diferir considerablemente del comportamiento asintótico (dado que la convergencia es lenta) pero es alta la probabilidad de que el comportamiento esté cerca del promedio del conjunto. La Figura 7 ilustra ese fenómeno para una distribución regular (la curva que tiene un tramo recto vertical es la curva límite, las curvas con trazo intermitente son muestras de la familia y la curva con trazo continuo es el promedio).

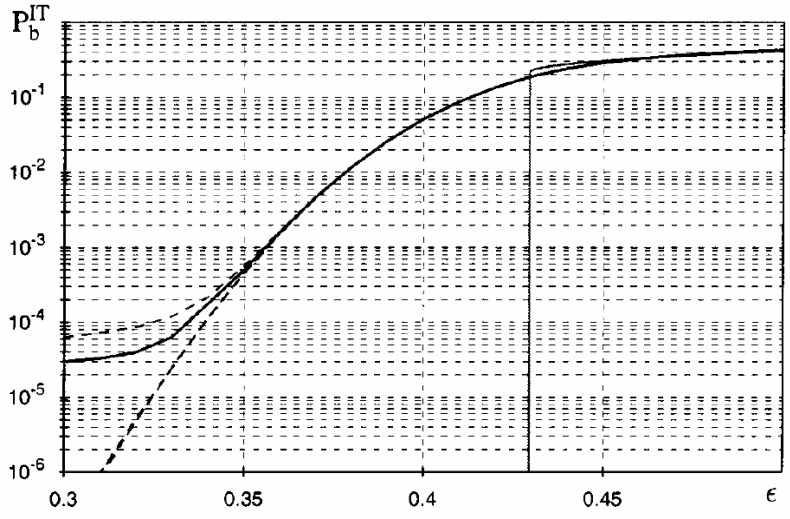

Figura 7

El hecho de que el comportamiento de largo finito difiera considerablemente del caso asintótico hace que density evolution no dé un resultado muy preciso para el caso de largo finito. Por lo tanto, la idea del análisis de largo finito consiste en estudiar el promedio de la

probabilidad de error, considerando conjuntos de códigos definidos por pares de distribuciones pero de largo finito.

La herramienta clave, introducida en [6], es una caracterización de tipo combinatoria de las fallas de decodificación. Un conjunto de parada (**stopping set)** S es un subconjunto de V, el conjunto de los nodos de variable, tal que todos los vecinos de S están conectados a S por lo menos dos veces.

Observaciones:

- el conjunto vacío es un stopping set
- el espacio de los stopping sets es cerrado bajo la unión de conjuntos

Esto implica que cada subconjunto de V contiene a un único stopping set máximo (que puede ser el conjunto vacío).

Se demuestra fácilmente el siguiente lema:
Lema: el conjunto de borraduras que queda cuando Belief Propagation termina es igual al máximo stopping set contenido en el conjunto original de borraduras.

También se puede ver fácilmente que el conjunto de las posiciones que valen 1 (**support set**) de cualquier palabra de código genera un stopping set. Sin embargo, no todos los stopping sets corresponden a support sets. Esto da una caracterización de la suboptimalidad del decodificador dado que el decodificador ML falla si y sólo si el conjunto de borraduras incluye un support set[19], sin embargo, el decodificador iterativo queda atrapado en una clase más amplia, el stopping set.

[19] Esto es porque al tratarse de un código lineal, si se borran, en una palabra x, todas las posiciones correspondientes al support set de una palabra y entonces el decodificador no puede distinguir entre la palabra x y la palabra $x+y$.

La idea básica del análisis de largo finito, si consideramos la probabilidad de error por bloque, es la siguiente. Fijamos un cierto patrón de borradura, sobre el conjunto de todos los grafos de cierta distribución de grados, determinamos la cantidad de constelaciones[20] que contienen stopping sets dentro del support set del patrón de borraduras (las denominamos "malas constelaciones"). Dado que, por definición, todas las constelaciones tienen la misma probabilidad, la probabilidad (promediada sobre el conjunto y condicionada al patrón especifico de borradura) de que el patrón de borradura no pueda ser corregido es simplemente el cociente entre la cantidad de malas constelaciones y la cantidad total. Para llegar a la probabilidad incondicional es necesario promediar sobre todos los patrones de borradura. A partir de este método, también es posible determinar la probabilidad de error por bit. Por más detalles ver el Apéndice I.

El problema combinatorio de determinar las malas constelaciones resulta, en general, en expresiones bastante complejas de evaluar y es por eso que se han buscado buenas cotas o aproximaciones más fáciles de evaluar (ver [26]).

El objetivo final de este tipo de análisis es el de lograr optimizar las distribuciones para obtener la menor probabilidad de error esperada para conjuntos de largo finito.

3.10 Codificación sistemática eficiente

En esta sección, abordaremos el problema de la codificación. Recordamos que la codificación consiste en obtener la palabra de código correspondiente a un vector de información dado. El método que vamos a describir es válido para cualquier canal.

Como se dijo en el capítulo 2, es muy importante lograr una codificación eficiente ya que para lograr una probabilidad de error arbitrariamente chica, es necesario que el largo sea suficientemente grande.

[20] Una constelación es una realización de un grafo aleatorio

Luby et al. ([14]) sugirieron usar una cascada de grafos bipartitos para lograr una codificación en tiempo lineal en el largo del código. Esta técnica tiene ciertos inconvenientes (ver [23]o [31]) que llevaron a Richardson y Urbanke ([23]) a desarrollar otro método que es el que vamos a presentar. Tanto la técnica de Luby et al. como la que presentamos logran una codificación sistemática eficiente.

Dada una matriz de chequeo de paridad H, un código se define como el conjunto de palabras tales que $Hx^T = 0$.

Si la matriz $H_{m\times n}$ es triangular inferior con unos en la diagonal, se puede codificar en forma sistemática eficientemente mediante substitución hacia atrás. Más precisamente, para $l \in [1..m]$ se calcula el $l-$ésimo símbolo de paridad como:

$$p_l = \sum_{j=1}^{n-m} H_{l,j} s_j + \sum_{j=1}^{l-1} H_{l,j+k} p_j$$

Normalmente, la matriz H no se encuentra originalmente en forma triangular inferior. Para ponerla en esa forma se puede usar la eliminación gaussiana pero, luego, la sustitución hacia atrás se hará en una cantidad $O\left(n^2\right)$ operaciones, ya que en general la matriz resultante no será de baja densidad.

La idea es llevar la matriz H a una forma triangular inferior pero conservando su baja densidad. Para eso, efectuamos únicamente permutaciones de las filas y de las columnas de la matriz. Sin embargo, no siempre es posible y fácil llevarla a una forma triangular inferior por este procedimiento, es por eso que nos conformaremos con una forma triangular inferior aproximada como la que se ilustra en la Figura 8.

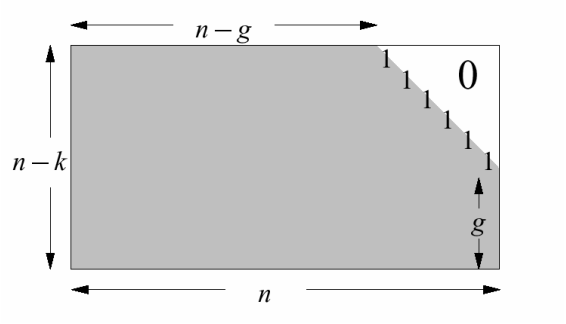

Figura 8

Más precisamente, la matriz de paridad queda con la forma:

$$H_{tri} = \begin{pmatrix} A_{(m-g)\times(n-m)} & B_{(m-g)\times g} & T_{(m-g)\times(m-g)} \\ C_{g\times(n-m)} & D_{g\times g} & E_{g\times(m-g)} \end{pmatrix}$$

A partir de esta última matriz, es posible codificar con complejidad $O\left(n+g^2\right)$ (ver el Apéndice J). Por lo tanto, la idea es hacer el **gap** g lo más chico posible y, si podemos hacer que sea de orden menor o igual que $O\left(\sqrt{n}\right)$, entonces lograremos una codificación en tiempo lineal en el largo del código. Por los tanto, nos queda ver como se lleva una matriz cualquiera de baja densidad a una forma triangular aproximada con menor g posible.

En el trabajo de Richardson et al., se sugiere usar un algoritmo voraz (greedy) para hallar una buena solución al problema. Está basado en la idea de **extensión de la diagonal**:
Dada una matriz A y un conjunto de columnas $c_1,...,c_k$ que tienen unos en las filas $r_1,...,r_k$ de grado 1 (es decir, que tienen un solo 1 y el resto son ceros). Entonces permutando filas y columnas es posible lograr una matriz de la forma ilustrada en la Figura 9.

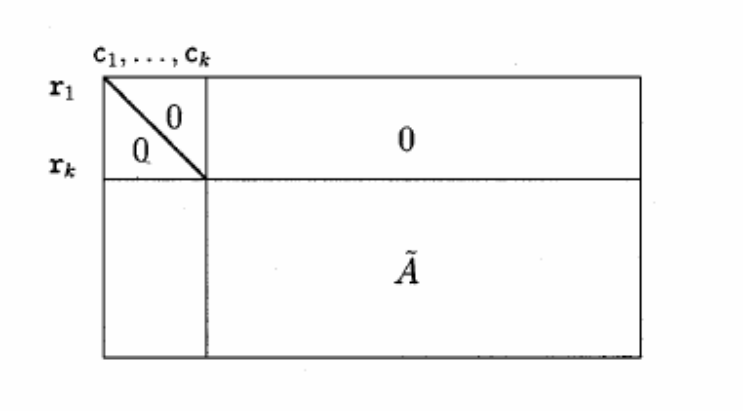

Figura 9

En el caso que no haya filas de grado 1, se busca una fila con menor grado posible y se permutan filas y columnas de forma tal de dejar $\tilde{A}$ en la forma ilustrada por la Figura 10. De esta forma, g aumenta en la cantidad de unos que quedan a la derecha (o sea, el grado de la fila menos uno).

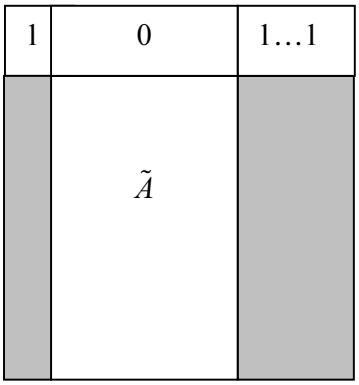

Figura 10

Luego se busca, nuevamente, filas de grado uno en la matriz residual y así sucesivamente. Por lo tanto, el **algoritmo greedy de triangulación inferior aproximada** queda:

Entrada: una matriz $H_{m\times n}$, $n > m$

Salida: una matriz $H^{tri}_{m\times n}$ en forma triangular inferior aproximada con gap g

1. $A := H^T$

2. Si la matriz A es vacía, ir a 3. Si hay filas de grado 1 en la matriz A, realizar la extensión de la diagonal, $A \coloneqq \tilde{A}$ e ir a 2.
3. Buscar una fila de menor grado posible y realizar el procedimiento ilustrado en la Figura 10. $A \coloneqq \tilde{A}$ .Volver a 2.
4. Se obtiene una matriz de la forma ilustrada en la Figura 11, por lo tanto es necesario pasar la submatriz P para la izquierda, luego hacer una simetría vertical y finalmente rotar la matriz 90° en sentido horario para obtener H^{tri}.

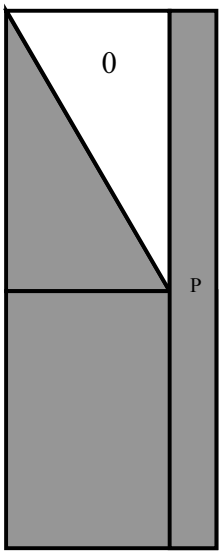

Figura 11

Es importante notar que cuando se realizan permutaciones, toda la matriz debe ser tomada en cuenta y no solo la parte residual para que el código definido por la matriz resultante sea equivalente al original.

El siguiente teorema ([26]) dice que, si el par de distribuciones que define una familia de códigos cumple ciertas restricciones, entonces con alta probabilidad los códigos de la familia pueden codificarse con complejidad lineal en el largo del código.

Teorema de codificación en tiempo lineal: Dado un par de distribuciones que cumple las condiciones:

$$(1-x) > \rho(1-\lambda(x)) \ \forall x \in (0,1)$$
$$r_{\min} > 2,\ r_{\min} \text{ siendo el grado derecho mínimo}$$
$$\rho'(1)\lambda_2 > 1$$

Sea un grafo G elegido al azar de la familia $C_m(\lambda,\rho)$[21], siendo m la cantidad de checks, entonces el algoritmo anterior aplicado a la matriz de G, produce una matriz con gap g menor que $m^{1/3}$ con probabilidad de, por lo menos, $1-e^{-m^{1/4}}$.

En la práctica[22], se observa que g toma valores entre 1 y 3, independientemente del largo. Otra importante observación es que, en general, las distribuciones que resultan de la optimización asintótica que vimos anteriormente cumplen las restricciones del teorema. En otras palabras, códigos buenos se codifican eficientemente.

En el caso, en el que no se cumplen las condiciones del teorema, la complejidad de la codificación resulta cuadrática pero, en general, con coeficiente muy chico ([23]). Por ejemplo, en el caso del par distribuciones regular (x^2, x^5), el gap es $g = 0.017n$ y por lo tanto la complejidad de la codificación queda $n + 0.000289n^2$.

Un interesante efecto colateral de este método es que permite pasar los nodos de mensaje de bajo grado para la parte de la redundancia, lo cual en la práctica es muy bueno ya que en general se observa que los bits que quedan sin recuperar al final del proceso de decodificación son de grado bajo. En realidad, es necesario hacerle una pequeña modificación al algoritmo de triangulación propuesto para darle prioridad a las variables de grado bajo para que pertenezcan a la parte triangular[23].

---

[21] Es la familia de los códigos con distribuciones (λ,ρ) y m checks

[22] Esto fue observado en [23] y también fue observado en las pruebas realizadas en este proyecto

[23] La idea es no tomar cualquier fila con grado 1 en la matriz residual, sino que, preferentemente, seleccionar las que tienen menor grado en la matriz original.

Es fácil de ver que, si el subgrafo inducido por las variables de grado 2 no tiene ciclos, entonces todas esas variables pueden ser pasadas para la parte de la paridad usando el algoritmo modificado de triangulación ([23]).

3.11 Comparación con códigos Reed-Solomon en el canal de borradura

Los códigos **Reed-Solomon** pueden ser considerados como los códigos "tradicionales" para el canal de borradura. En [3], hay una comparación entre estos códigos y los códigos Tornado que son códigos basados en LDPC y, en particular, en la secuencia Heavy-Tail Poisson. En ese artículo, se consideró un alfabeto extendido, para que cada símbolo del alfabeto represente un paquete entero de datos (de largo fijo P) a ser enviado por una red de datos.

Los códigos Tornado no son exactamente códigos LDPC como los que vimos hasta ahora. En particular, difieren en que el grafo que los representa está compuesto por una cascada de grafos bipartitos. Sin embargo, conservan la esencia de la decodificación LDPC, es decir: una codificación en tiempo lineal y una decodificación subóptima (mediante Belief Propagation) en tiempo lineal debido al uso de grafos con baja densidad de aristas. Además, el uso de la secuencia Heavy-Tail Poisson permite acercarse arbitrariamente a la capacidad. Por lo tanto, los resultados presentados en ese artículo resultan ilustrativos del rendimiento de los códigos LDPC en comparación con códigos tradicionales para el canal de borradura.

Supongamos que queremos codificar k símbolos y le agregamos l símbolos de redundancia, enviamos entonces $n = k + l$ paquetes (símbolos). Por ser MDS, los códigos Reed-Solomon garantizan que si se reciben correctamente k símbolos entonces es posible recuperar toda la información exitosamente. La decodificación de los códigos Reed-Solomon está basada en resolver un sistema de ecuaciones pero, debido a la densidad del mismo, la complejidad es mayor que para los códigos Tornado. Más precisamente, la codificación y la decodificación de los códigos Reed-Solomon, se puede realizar en tiempo $O\left(n \log^2 n \log\log n\right)$([14]). Sin embargo, la constante multiplicativa que esconde la

expresión anterior es grande y puede ser conveniente usar un algoritmo de tiempo cuadrático disponible para los códigos Reed-Solomon cuando n es chico.

El precio que se paga a cambio de la baja complejidad que se obtiene usando los códigos Tornado, es la necesidad de una mayor cantidad de símbolos recibidos correctamente para decodificar exitosamente. Por lo tanto, una manera de medir la confiabilidad es comparando respecto a los códigos MDS, diciendo que un código tiene **ineficiencia en la decodificación** $(1+\varepsilon)$ si $(1+\varepsilon)k$ símbolos son necesarios para recuperar toda la información. Sin embargo, para los códigos Tornado, la ineficiencia en la decodificación no es una cantidad fija dado que depende del patrón de las borraduras y del código que se elige aleatoriamente. A pesar de esto último, en [3], los autores muestran, empíricamente, que la ineficiencia en la decodificación está concentrada alrededor de un cierto valor.

La tabla siguiente compara las propiedades de los códigos Tornado con las de los códigos Reed-Solomon:

	Códigos Tornado	Códigos Reed-Solomon
Ineficiencia en la decodificación	$(1+\varepsilon)$	1
Tiempo de codificación	$(k+l)\ln(1/\varepsilon)P$	$k(1+l)P$
Tiempo de decodificación	$(k+l)\ln(1/\varepsilon)P$	$k(1+x)P$ [24]
Operación básica	XOR	Operaciones en cuerpos finitos

En definitiva, la ventaja de los códigos Tornado es que permiten una mejora substancial en la complejidad de la decodificación y de la codificación a cambio de una pequeña ineficiencia en la decodificación.

[24] x es la cantidad de paquetes de redundancia que llegan correctamente

3.12 Problemas abiertos

A continuación, enumeramos algunos de los problemas abiertos importantes relativos al tema de los códigos LDPC que encontramos en los trabajos relevados:

- Encontrar secuencias que alcancen la capacidad para canales diferentes del de borradura (por ejemplo, para los canales gaussiano y binario simétrico) ([31]).
- Generalizar el concepto de stopping set a otros canales para permitir hacer un análisis de largo finito como en el caso del canal de borradura ([6]).
- Encontrar buenas aproximaciones o cotas para la probabilidad de error promedio en el caso del canal de borradura para poder hacer una optimización de largo finito ([24]).

4 EXPERIMENTACIÓN

En este capítulo, presentamos cuatro experimentos que pretenden ilustrar los conceptos estudiados en el capítulo 3. Por más detalles sobre los programas desarrollados para realizar los experimentos, consultar el Apéndice K.

4.1 Ejemplo de codificación y de decodificación de una imagen

Objetivos

El objetivo de este experimento es mostrar visualmente el efecto de usar un sistema de codificación para transmitir información en forma confiable sobre en un canal de borradura.

Procedimiento

A modo de ejemplo, tomamos, como información a transmitir, la matriz de ceros y unos de tamaño 500×500 ilustrada en la Figura 12[25].

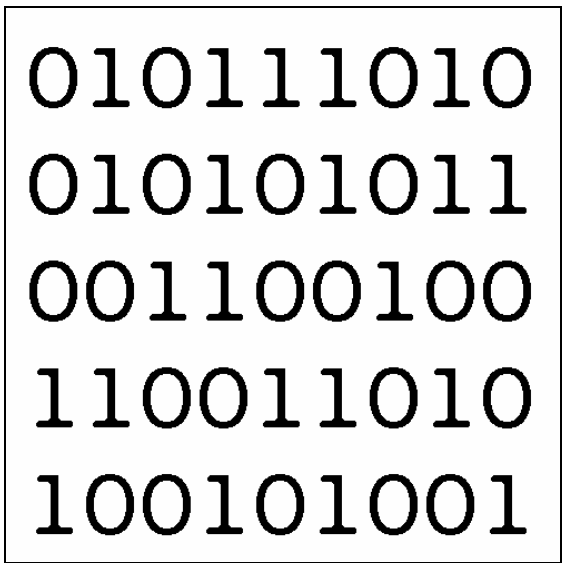

Figura 12

[25] Los ceros se muestran como píxeles de color negro y los unos como píxeles de color blanco.

Tomamos cada fila de píxeles como un vector de información. Para codificar, elegimos un código sistemático de tasa $1/2$ (con umbral asintótico aproximadamente 0.451)[26]. La matriz correspondiente a un grafo generado por el algoritmo de generación de grafos[27] se muestra en la Figura 13.

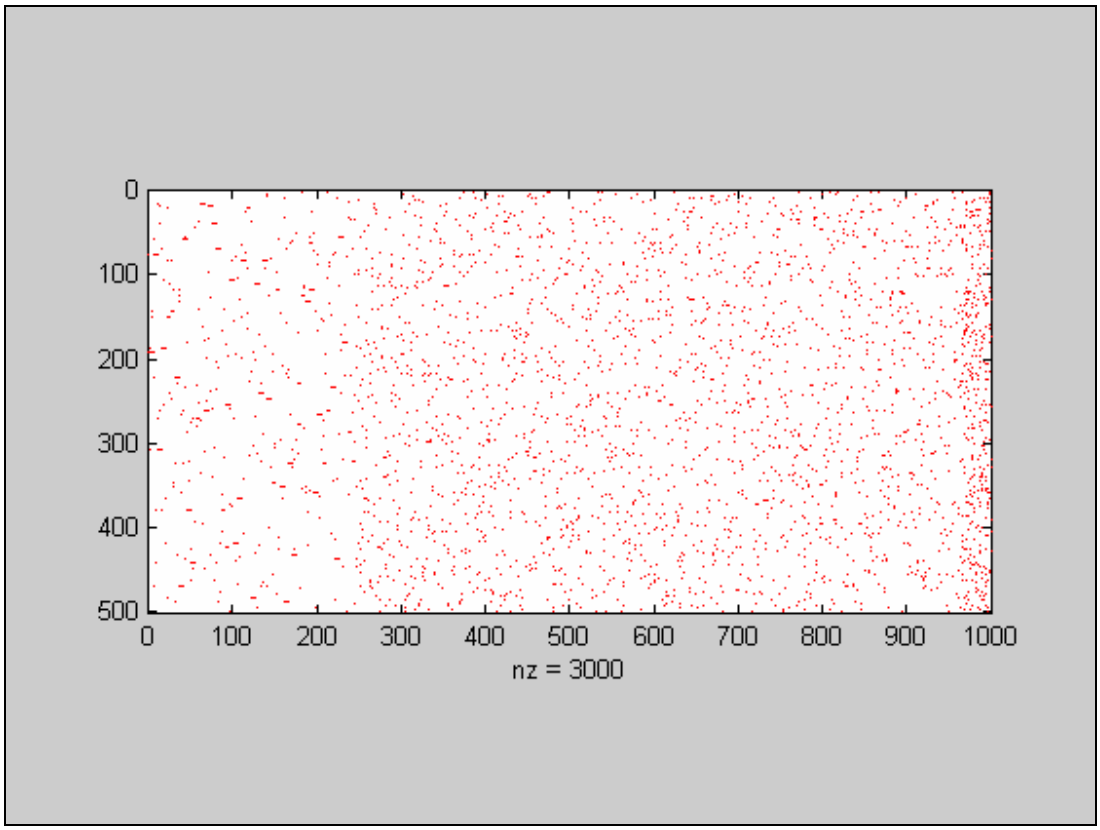

Figura 13

Luego de aplicarle a la matriz anterior el algoritmo de triangulación aproximada, se obtiene la matriz de la Figura 14.

[26] El par de distribuciones es $\lambda(x)=0.166483x+\ 0.714601x^2+0.118916x^9$ y $\rho(x)=x^5$

[27] El grafo fue generado por el algoritmo que genera grafos sin ciclos en las variables de grado 2

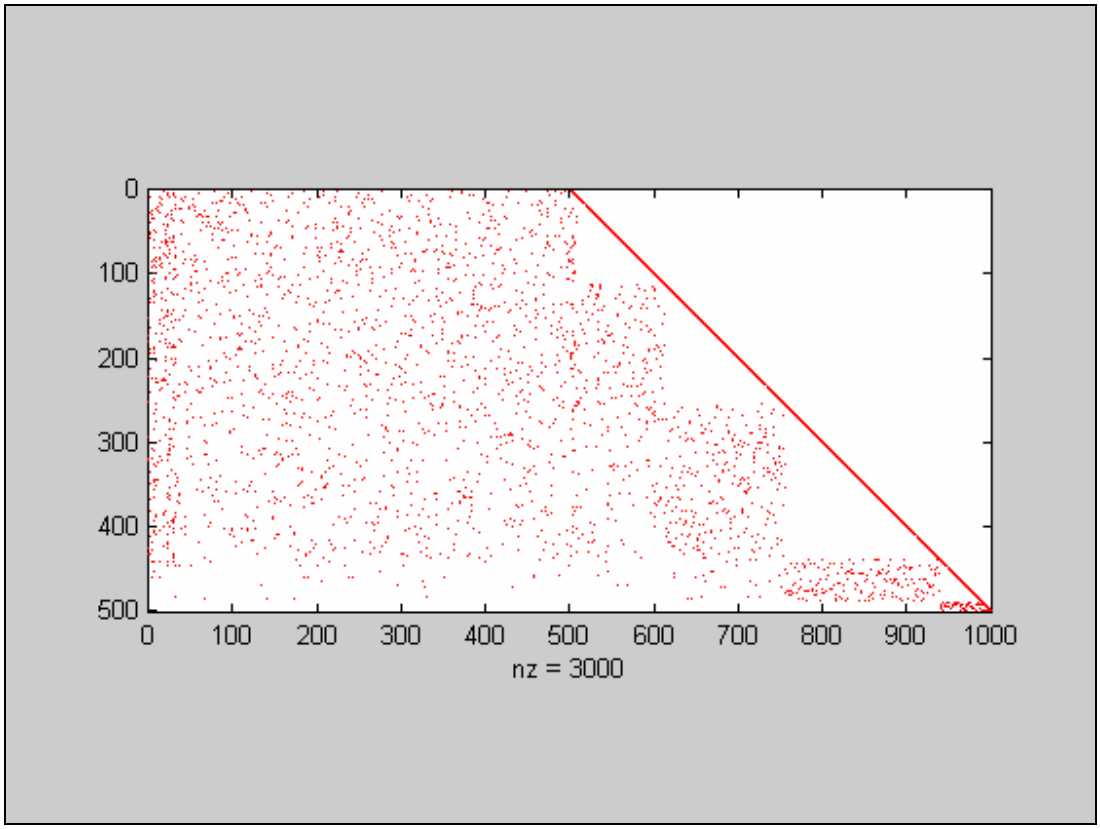

Figura 14

Es fácil de verificar que el par de distribuciones no cumple las condiciones del teorema de codificación eficiente, sin embargo, el gap resultante es 1 y, por lo tanto, es posible codificar en forma rápida, en este caso.

Codificando con la matriz de la Figura 14, obtenemos las palabras de código ilustradas por la Figura 15 (cada fila es una palabra de código), donde observamos que la información se encuentra a la izquierda y la paridad a la derecha.

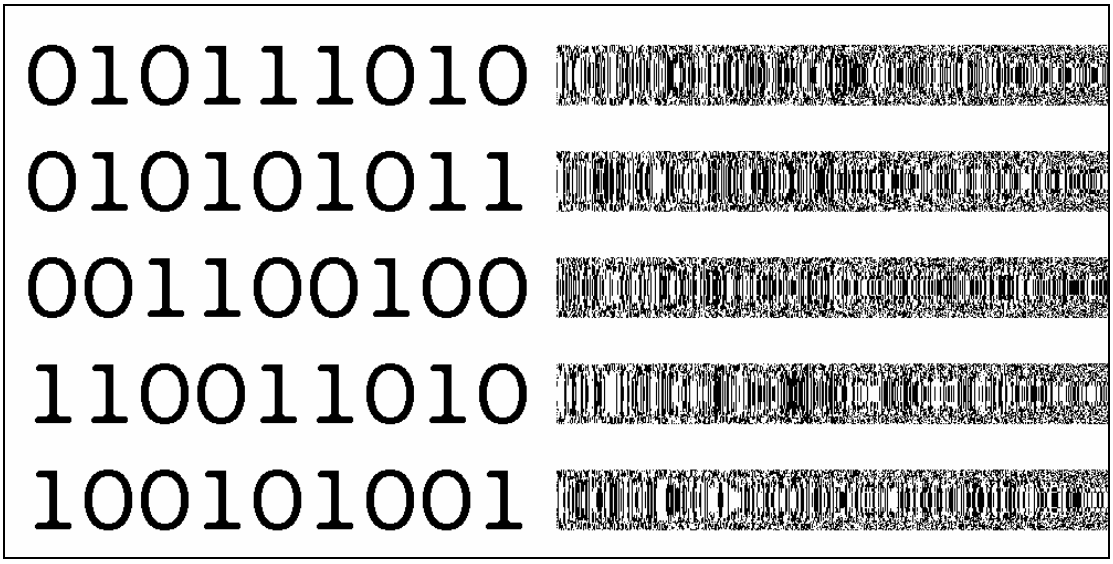

Figura 15

Luego, pasamos la misma información codificada por dos canales distintos: el $BEC(0.2)$ y el $BEC(0.43)$, los resultados correspondientes se muestran en la Figura 16 y en la Figura 17[28], respectivamente.

Figura 16

[28] Las borraduras se muestran como píxeles de color rojo

Figura 17

Finalmente, usamos el algoritmo Belief Propagation para recuperar los bits borrados.

Resultados

Los resultados obtenidos luego de decodificar la información obtenida a la salida de los canales *BEC*(0.2) y *BEC*(0.43) se ilustran en la Figura 18 y en la Figura 19, respectivamente.

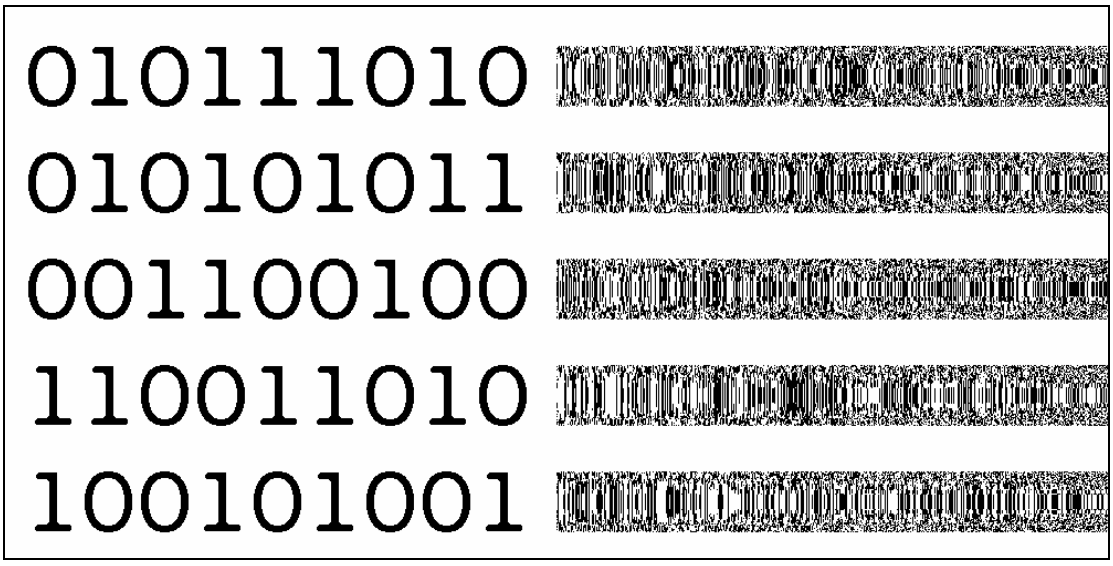

Figura 18

Figura 19

En la Figura 18, no queda ninguna borradura mientras que en la Figura 19, proveniente de un canal con parámetro cercano al umbral del código, permanecen algunas. Recordamos que el precio que pagamos por lograr tal capacidad de recuperación de borraduras es el envío del doble de la cantidad de símbolos que realmente se desea enviar.

4.2 Concentración y convergencia al caso asintótico

Objetivos

A través de este experimento, deseamos ver si se cumplen, para algunos casos, los teoremas de concentración y de convergencia al caso límite, presentados en las secciones 3.5 y 3.6 respectivamente. Los casos elegidos corresponden a los pares de distribuciones siguientes, ambos de tasa $2/3$:

1. Regular-(3,9): todas las variables tienen grado 3 mientras que los checks tienen grado 9.
2. Heavy-Tail Poisson N=100

En el límite, la probabilidad de error para ε's menores que el umbral debe ser 0 mientras que, para ε's mayores, la probabilidad de error debe estar acotada por debajo por valores mayores que 0. Nos interesa la convergencia al caso límite en el rango de ε's menores que el umbral (y relativamente cercanos al umbral).

Procedimiento

Para mostrar la evolución de la concentración y la convergencia, usamos 3 largos de código: 999, 9999 y 50001. Para cada par de distribuciones y largo considerados, generamos 5 grafos en forma aleatoria. Para cada uno de esos grafos, hacemos una estimación empírica de la probabilidad de error por bit para diferentes probabilidades de borradura del canal. Para hacer esa estimación, se simulamos el envío de palabras, a través del canal considerado, una cierta cantidad de veces y para cada palabra enviada se evaluamos las borraduras que permanecen al final de la decodificación. El procedimiento se resume en el seudo-código la Figura 20 :

```
generar los grafos aleatoriamente
para cada grafo
     para cada probabilidad de borradura del canal considerada
          repetir N(perr) veces
               generar la palabra compuesta únicamente por ceros
               enviarla por el canal de borradura considerado
               decodificarla
               acumular la cantidad de bits borrados
          fin repetir

          estimador Pbit = # bits borrados/# bits total

     fin para
fin para

calcular las prob. de error promedio considerando todos los grafos
```

Figura 20

Notar que, dada la independencia de la probabilidad de error por bit respecto a la palabra enviada, se usa siempre la palabra compuesta únicamente por ceros para evitar usar el algoritmo de codificación). La cantidad de muestras N(perr) se ajusta para que el estimador tenga siempre un dígito de exactitud con 95% de probabilidad según la técnica propuesta en [33][29].

Por detalles sobre la generación del par 2, consultar la sección K.2.

[29] Se busca obtener una cantidad fija mínima de borraduras con, a lo sumo, un cantidad especificada de muestras. Por más detalle, consultar el código fuente en el Apéndice L.

Resultados

3. Regular-(3,9)

La Figura 21 muestra el resultado de la simulación (la Figura 22 muestra una vista ampliada de la Figura 21). Las curvas de color rojo corresponden a códigos de largo 999, las de color azul a códigos de largo 9999 y las de color verde a códigos de largo 50001. Cuando una curva se corta (a la izquierda) significa que, en el punto anterior al corte, el algoritmo que hace la simulación tomó la cantidad máxima de muestras pero no logró obtener ningún error[30]. Las curvas negras representan el promedio de las curvas rojas y de las azules. Por claridad de la gráfica, no representamos el promedio de las curvas verdes. La curva violeta representa el umbral del par de distribuciones y la curva vertical negra representa el límite de Shannon.

[30] Es de esperar que pase lo mismo (o sea, que el algoritmo de simulación no obtenga ningún error usando la cantidad máxima de muestras) para todos los puntos a la izquierda del punto de corte

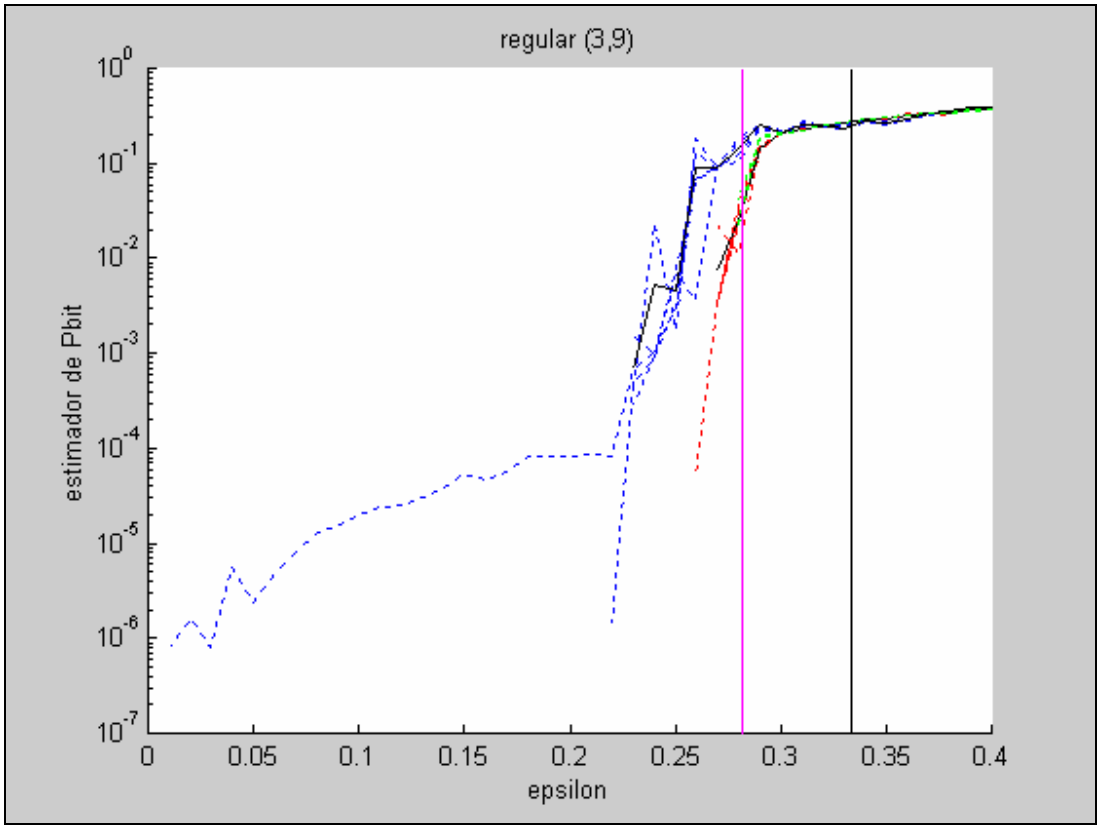
regular (3,9)
estimador de Pbit
epsilon
10^0
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4

Figura 21

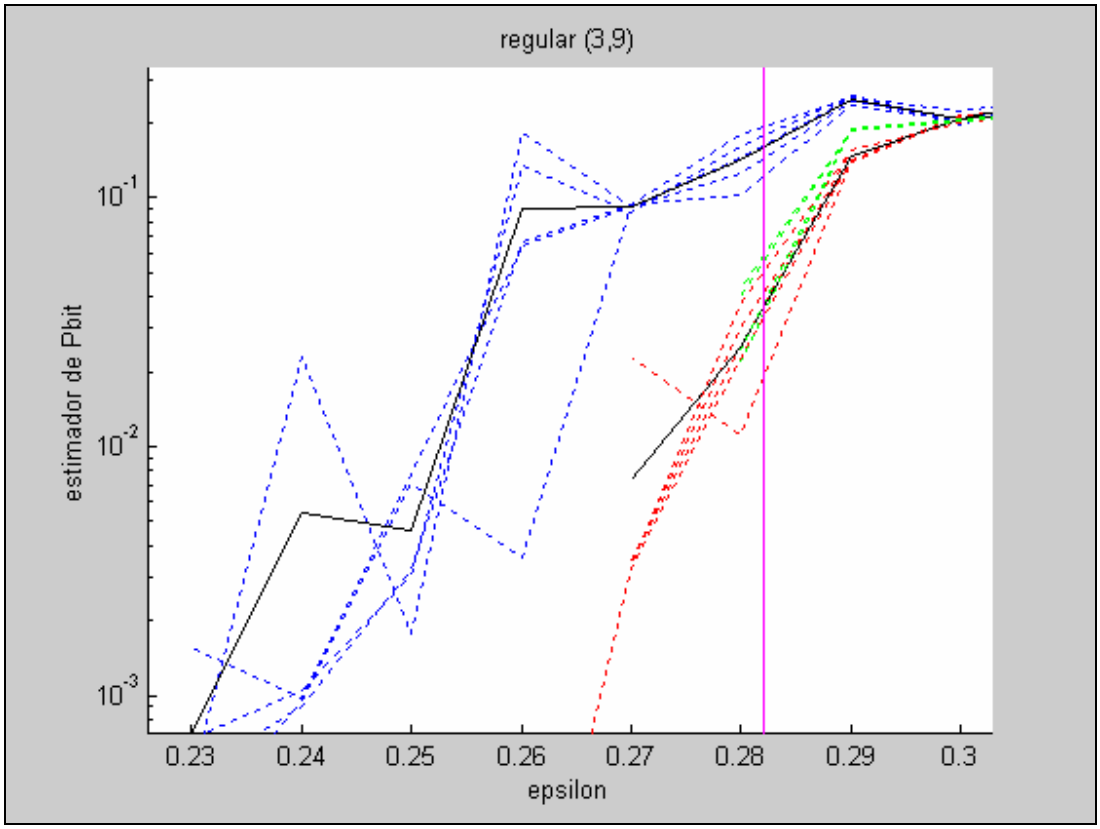

Figura 22

Observamos que la concentración aumenta a medida que el largo aumenta. También notamos la convergencia al caso límite. En el caso de las curvas verdes no se ve tan claramente pero el hecho de que las curvas estén cortadas significa que para los ε's menores que el ε de corte, la probabilidad de error por bit es menor que 7.999840×10^{-9} con 95% de probabilidad.

4. Heavy-Tail Poisson N=100

Los resultados de la simulación aparecen en la Figura 23. Los colores fueron elegidos de la misma forma que en el caso anterior.

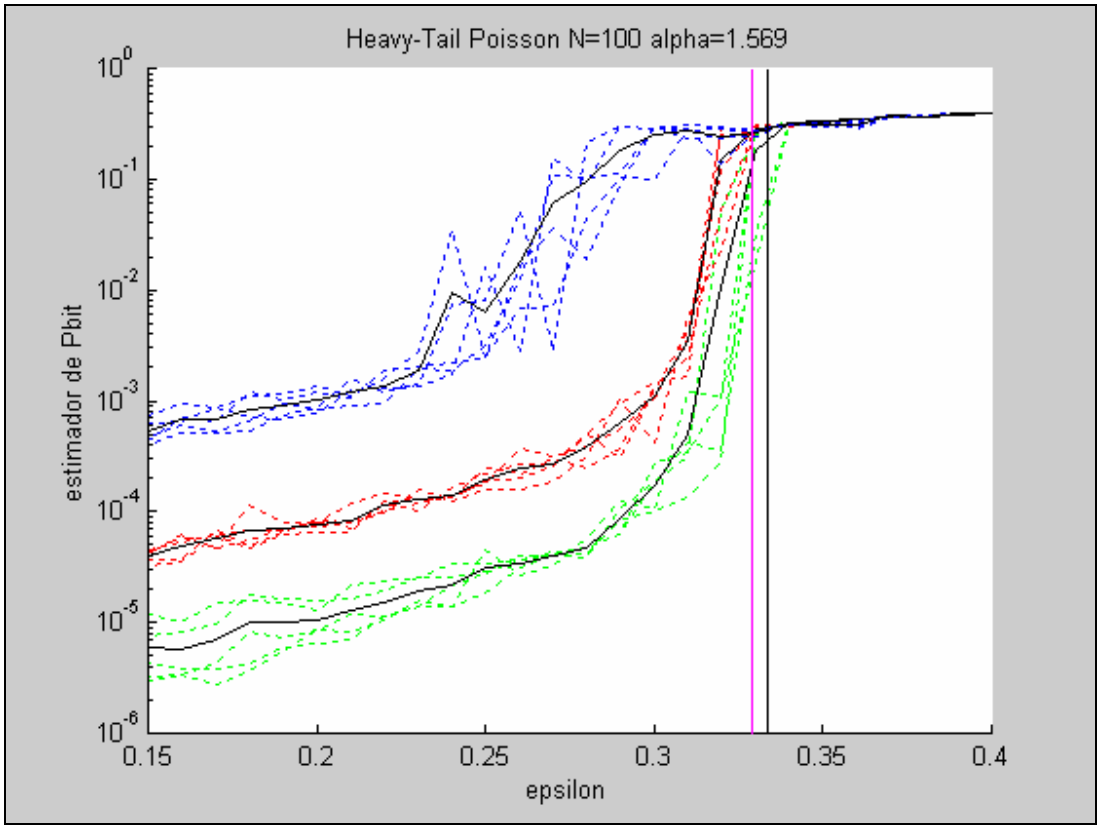

Figura 23

En este caso, también observamos la convergencia al caso límite pero la velocidad de convergencia es menor que en el caso anterior. También notamos que las curvas están concentradas pero la concentración, en general, es menor que en el caso anterior.

Los resultados de este experimento, nos muestran que es viable, en la práctica, el procedimiento de buscar familias de códigos con buenos umbrales y luego tomar, al azar, un código de la familia, suficientemente largo, para lograr códigos cercanos a la capacidad para probabilidades de error arbitrariamente bajas.

4.3 Comparación entre códigos de igual largo, tasa y complejidad

Objetivo

En este experimento, deseamos comparar empíricamente pares de distribuciones con las características siguientes:

- Grado derecho promedio: lo más cercano a 6 posible
- tasa: lo más cercana a ½ posible
- largo:10000

Dado que fijamos la tasa y el grado derecho promedio, la complejidad queda fija. Dado que además fijamos el largo, los códigos van a ser comparados por su probabilidad de error por bit empírica.

Los pares de distribuciones elegidos son los siguientes:

- Regular (3,6): como vimos en el experimento anterior, los pares regulares presentan una rápida convergencia al límite y una alta concentración, sin embargo no pueden acercarse a la capacidad. El par Regular(3,6) es el par regular de tasa $1/2$ con mejor umbral posible ([16]). El umbral es aproximadamente 0.429.
- Heavy-Tail Poisson con parámetros $N = 8$ y $\alpha \approx 5.9105$: la tasa es ½, el grado derecho promedio obtenido es aproximadamente 5.93 y el umbral es 0.438 (no hay mucha mejora respecto al Regular).
- Right Regular con parámetros $N = 13$ y $\alpha = 0.2$: la tasa es aproximadamente 0.499, el grado derecho promedio es 6 y el umbral aproximadamente 0.480. Aquí notamos el excelente umbral de esta distribución comparado con las dos anteriores de misma complejidad lo cual concuerda con la optimalidad respecto al balance entre complejidad y distancia a la capacidad mencionada en la sección 3.7.
- Optimizada: Imponemos la restricción de que el grado izquierdo máximo sea menor que 8 para que la convergencia al comportamiento asintótico sea más rápida que para el par anterior (el par anterior tiene grado izquierdo máximo 13). Esta distribución es regular del lado derecho al igual que la anterior. Su tasa es $1/2$, su grado derecho promedio 6 y su umbral aproximadamente 0.481. Notamos que tiene un umbral aún mayor que la anterior.

Procedimiento

El procedimiento para hacer las simulaciones es el mismo que en el experimento anterior. En la sección K.2, explicamos como construimos los pares de distribuciones Heavy-Tail

Poisson y Right-Regular. El par optimizado se obtuvo a partir de la aplicación disponible en http://lthcwww.epfl.ch/research/ldpcopt.

Resultados

En la Figura 24, mostramos los resultados de las simulaciones. Las curvas de color azul corresponden al par Heavy-Tail Poisson, las de color violeta al par Regular(3,6), las de color rojo al par Right-Regular y las de color verde al par optimizado. Las curvas verticales con los colores anteriores representan los umbrales correspondientes. La curva vertical negra representa el límite de Shannon.

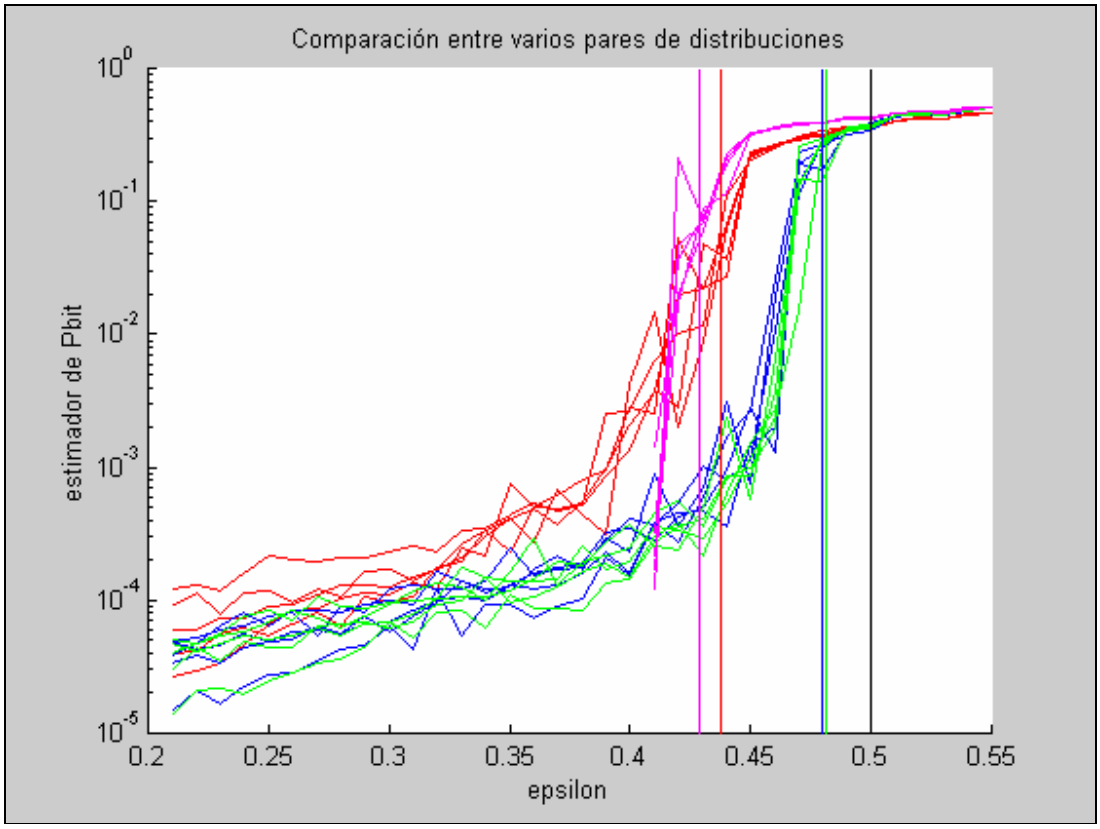

Figura 24

Debido a su cercanía al caso asintótico[31], los códigos regulares son mejores que los demás para un gran rango de ε's (entre 0 y 0.41 aproximadamente). Sin embargo, cuando estamos más cerca de la capacidad son ampliamente mejores los códigos provenientes del par Right-Regular y del par optimizado. Si comparamos los resultados de estos últimos códigos, su rendimiento resultó bastante similar salvo en la región de "cascada" (entre 0.45 y 0.48 aproximadamente), donde algunos códigos resultantes del par optimizado son mejores que los Right-Regular.

Los resultados de este experimento nos sugieren que el umbral asintótico de un par de distribuciones es un buen indicador del rendimiento de largo finito de un código perteneciente a una familia representada por el par, sobre todo en las regiones cercanas a los umbrales de los códigos.

4.4 Codificación sistemática con un código optimizado específico

Objetivo

Este experimento se basa en una observación empírica de que, en general, las borraduras que permanecen al final de la decodificación se encuentran en las variables de grado bajo. El objetivo es mostrar, con un ejemplo de codificación sistemática, como se puede aprovechar el pasaje de las variables de grado 2 para la paridad, con el objetivo de reducir la probabilidad de error en los bits de información.

Procedimiento

Se obtuvo un código optimizado (a partir de la aplicación disponible en http://lthcwww.epfl.ch/research/ldpcopt) con la restricción que la cantidad de variables de grado 2 sea menor que la mitad de la cantidad de checks. Las restricciones anteriores son para poder generar un grafo sin ciclos en esas variables y para que, luego, el algoritmo de

[31] En este caso, el hecho de que las curvas violetas estén cortadas significa que para ε's menores al ε de corte, la probabilidad de error por bit es menor que 4×10^{-8} con 95% de probabilidad

triangulación aproximada[32] pase todas las variables de grado 2 para la paridad. El código que usamos en este experimento es el mismo del primer experimento.

Resultados

El resultado se muestra en la Figura 25.

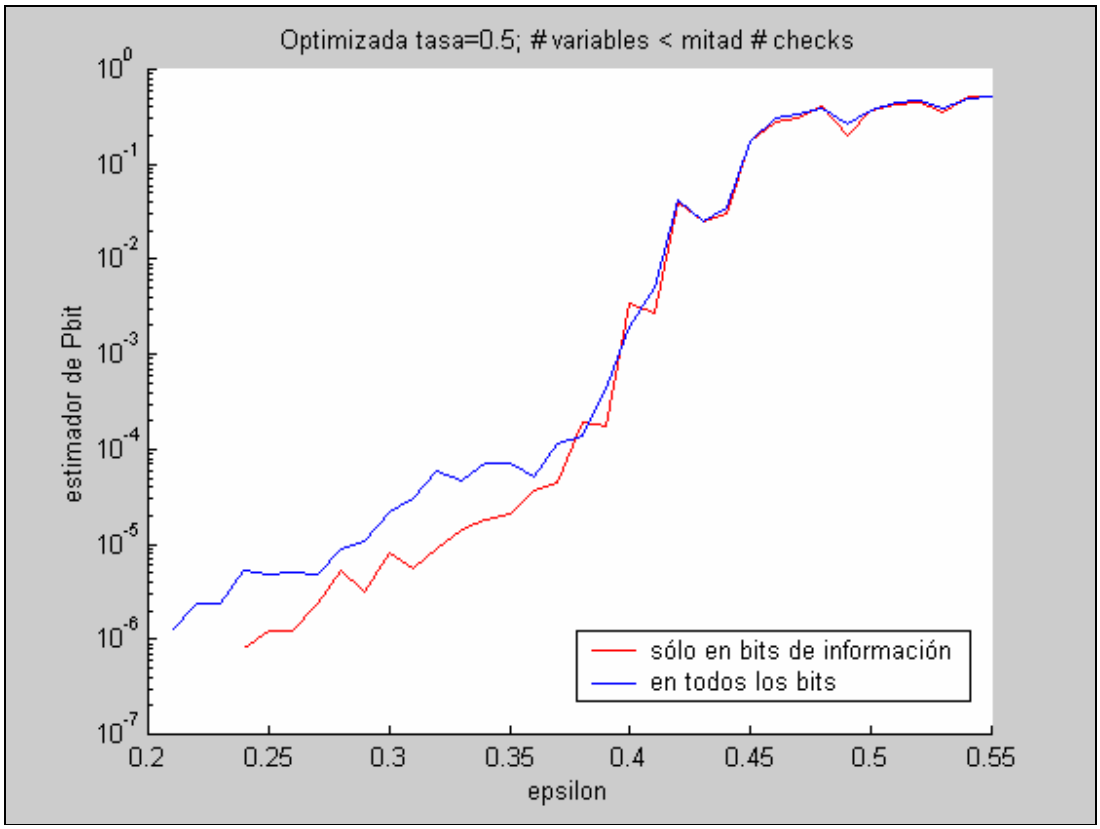

Figura 25

Observamos que la probabilidad de error disminuye, en general, al considerar únicamente los bits de información y, en el rango de ε's entre 0 y 0.37, el efecto es más notorio. Por lo tanto, lo anterior significa que los errores que permanecen al final están más concentrados en la parte de la paridad y, por lo tanto, la técnica parece ser eficaz.

[32] Con la modificación mencionada en la sección 3.10

La ganancia que se obtiene no es mucha (sobre todo si miramos la región cercana a la capacidad), sin embargo, este experimento muestra que teniendo cuidado en la construcción de los grafos es posible obtener mejoras.

5 TRABAJOS FUTUROS

En este capítulo, describimos tres propuestas de proyectos para realizar en el futuro.

5.1 Corrección de errores para distribución de datos en redes de computadoras

En este proyecto se pretende estudiar modelos de corrección de errores aplicables a problemas relacionados con Internet, entender diversos códigos específicos definidos para este propósito e implementar varios de ellos estudiando su comportamiento y comparando los resultados frente al uso del protocolo TCP/IP. Algunos trabajos de referencia para este proyecto serían [3], [28] y [32].
Este proyecto tendría un carácter más práctico y aplicado que el actual proyecto y daría la posibilidad de generar nuevos trabajos interdisciplinarios con el área de Sistemas Operativos y Redes.

Este proyecto ya ha sido presentado para el año actual como proyecto de grado bajo la dirección del Dr. A. Viola.

5.2 Análisis de largo finito y construcciones determinísticas para otros canales

Parece haber una tendencia actual (ver, por ejemplo, [37] y [22]) hacia el estudio combinatorio de los códigos LDPC inspirado en los avances realizados inicialmente para el canal de borradura. También han surgido varios algoritmos que permiten construir códigos LDPC en forma determinística o "menos aleatorias" que las que vimos en este proyecto. Esos algoritmos toman como punto de partida el análisis asintótico basado en density evolution.

Los objetivos de este proyecto serían los siguientes:

- Estudiar la teoría existente relativa a análisis de largo finito para canales distintos del de borradura, focalizándose en principio en el canal binario simétrico.

- Estudiar construcciones determinísticas de códigos LDPC existentes para distintos canales.
- De ser posible, hacer algún aporte original a la teoría o crear algún algoritmo de generación de buenos códigos LDPC.

En este proyecto, creemos que puede resultar fructífera la cooperación con el área de matemática discreta del IMERL debido al carácter combinatorio del mismo.

5.3 Mejora del algoritmo Belief Propagation

Dada la suboptimalidad del algoritmo Belief Propagation cuando se aplica a grafos con ciclos, puede ser un camino interesante el de buscar posibilidades de reducir el efecto de los ciclos en el algoritmo, tal como lo sugieren MacKay y Neal en [17], para que resulte una mejor aproximación de la regla de máxima verosimilitud. Un poco en ese sentido, durante el actual proyecto, ideamos una modificación del algoritmo Belief Propagation que permitió lograr, en algunos casos experimentales sobre el canal binario simétrico, una leve mejora de la probabilidad de error y de la cantidad de iteraciones necesarias para decodificar. Por lo tanto, los objetivos serían:

- Realizar más experimentos con el algoritmo modificado. En particular, jugando con el valor de un parámetro adicional que toma el algoritmo.
- En base a la experimentación, intentar realizar algún tipo de análisis o algún otro tipo de mejora del algoritmo.
- Buscar qué tipos de estructuras se comportan mejor con este algoritmo

La modificación fue inspirada por el caso de Belief Propagation aplicado al canal de borradura. La idea propuesta es la siguiente. Cuando se recorren los checks para calcular los mensajes a enviar a las variables, si el valor de una variable que no estaba determinada queda determinado, entonces es posible actualizar el mensaje enviado por dicha variable a sus checks adyacentes antes de que empiece la próxima iteración para que las próximas ecuaciones a recorrer puedan aprovechar inmediatamente el valor determinado. En el caso del canal de borradura, esta modificación no cambia la probabilidad de error, simplemente

se puede ver como una implementación eficiente del algoritmo Belief Propagation. Sin embargo, en el caso de otros canales los mensajes no son de certidumbre total pero, si definimos un nivel de certidumbre adecuado (este es el parámetro antes mencionado), se puede aplicar también la idea. Entonces, si una variable queda determinada con alta certidumbre por el mensaje enviado por un check, la variable actualiza sus mensajes y los checks que quedan por evaluarse en la iteración aprovechan esos mensajes actualizados. Una idea similar se usa en el algoritmo de Gauss-Seidel ([38]) para la resolución numérica iterativa de sistemas de ecuaciones. Quizás esta modificación no sea de mucho interés por si misma pero puede dar nuevas ideas sobre maneras de mejorar el algoritmo Belief Propagation.

6 CONCLUSIONES

El objetivo principal de este proyecto era hacer un relevamiento de los códigos LDPC y evaluar su impacto para resolver problemas de importancia práctica. Hicimos un relevamiento de los trabajos que presentan los resultados más importantes, enfocándonos en el caso del canal de borradura debido a su simplicidad y a que la mayoría de los resultados se extienden para otros canales. En particular, vimos que son adecuados para resolver problemas de distribución de datos masivos desde un servidor hacia varios clientes, problemas de almacenamiento distribuido de información y problemas de almacenamiento en medios magnéticos. También vimos su aptitud para acercarse al límite de Shannon con probabilidad arbitrariamente baja y con muy baja complejidad de codificación y de decodificación. La comparación con los códigos Reed-Solomon que presentamos en la sección 3.11, nos permitió ver como, a cambio de una menor capacidad de corrección de errores, los códigos LDPC ofrecen una importante reducción en la complejidad de los algoritmos.

Los resultados que estudiamos fueron esencialmente probabilísticos y asintóticos. Sin embargo, nuestras implementaciones nos mostraron la utilidad práctica de dichos resultados. En particular, vimos que el umbral de una familia de códigos es un buen indicador del comportamiento de largo finito de los códigos de la familia. También estudiamos una técnica para mejorar el rendimiento de ciertos códigos que, a pesar, de ser muy específica nos sugiere que, quizás, con construcciones de grafos más cuidadosas, se pueda lograr mejoras sustanciales. Para esto último, pensamos que sería deseable que la teoría de largo finito se desarrolle un poco más, en particular, en relación con las propiedades combinatorias de los grafos que representan a los códigos.

El hecho de que gran parte de la teoría de los códigos LDPC se haya desarrollado en los últimos 8 años y actualmente se siga haciendo mucho trabajo, nos ha brindado una experiencia nueva de aprendizaje que difiere considerablemente de las experiencias que tuvimos, en general, a lo largo de la carrera de Ingeniería en Computación. En particular,

tuvimos que enfrentarnos al trabajo de buscar y juntar las piezas de un rompecabezas del cual aún faltan muchas piezas.

Tal como lo detallamos en el capítulo 5, este proyecto puede ser continuado de varias maneras. Las propuestas que damos surgieron de distintas inquietudes que tuvimos a lo largo del proyecto y ofrecen la posibilidad de seguir trabajando en un tema de gran importancia en la actualidad.

APÉNDICE A. PROPIEDADES DE LA ENTROPÍA

En [30], Shannon sugirió que la función entropía $H(p_1, p_2, ..., p_n)$, en caso de existir, debiera tener las propiedades siguientes:

1. H debiera ser continua en p_i.
2. Si todos los p_i son iguales, entonces H debiera ser una función monótona en n. En otras palabras, si consideramos eventos equiprobables, hay más incertidumbre cuando hay más eventos posibles.
3. Si es posible dividir el proceso que selecciona cada evento en una sucesión de decisiones (o de eventos) intermedias hasta llegar al evento del conjunto que consideramos, entonces H debiera poder expresarse como la suma ponderada de las medidas de las incertidumbres que hay frente a los eventos intermedios. Por ejemplo, supongamos que tenemos 3 eventos con probabilidades $p_1 = 1/2, p_2 = 1/3, p_3 = 1/6$ cuyo proceso de selección puede tener un evento intermedio como se muestra en la Figura 26. Entonces, se requiere que $H(1/2, 1/3, 1/6) = H(1/2, 1/2) + \frac{1}{2} H(2/3, 1/3)$.

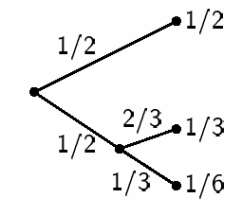

Figura 26

La función H tiene además las siguientes propiedades que la sustentan aún más como medida de la incertidumbre o de la cantidad de información:

1. $H = 0$ si y sólo si todos los p_i salvo uno son iguales a cero. O sea que la incertidumbre es cero cuando sólo puede ocurrir un evento con total certeza, lo cual es razonable. En todos los demás casos H es positiva.

2. Para una cantidad de eventos n dada, H es máxima e igual a $\log n$ cuando todos los p_i son iguales. Ésta es intuitivamente la situación de mayor incertidumbre.
3. Denotamos $H(x)$ y $H(y)$ a las entropías de los conjuntos de eventos x e y respectivamente. Podemos definir la **entropía de los eventos conjuntos** como $H(x,y) = -\sum_{i \in x, j \in y} \Pr(i,j) \log \Pr(i,j)$. Se demuestra fácilmente que la incertidumbre de un evento conjunto es menor o igual que la suma de las incertidumbres de los eventos individuales, es decir $H(x,y) \leq H(x) + H(y)$.
4. Cualquier cambio en las probabilidades de los eventos que tienda hacia la ecualización aumenta H. Por ejemplo, si $p_1 < p_2$ e incrementamos p_1 y decrementamos p_2 en la misma cantidad de manera tal de que p_1 y p_2 estén más cerca, entonces H crece.
5. Se demuestra fácilmente que la incertidumbre de un evento conjunto x, y es la incertidumbre de x más la incertidumbre de y cuando se conoce x, es decir $H(x,y) = H(x) + H(y \mid x)$.
6. De 3 y 5, se deduce que la entropía de y nunca crece por conocer x, es decir $H(y) \geq H(y \mid x)$.

APÉNDICE B. CANALES GAUSSIANO Y BINARIO SIMÉTRICO

El **canal de ruido blanco Gaussiano aditivo** (Additive White Gaussian Noise, AWGN) es de gran importancia práctica porque permite modelar varios medios físicos incluyendo enlaces de radio y satelitales.

En el caso general, tanto la salida como la entrada toman valores continuos y de tiempo discreto. El canal es sin memoria y la relación entre la salida Y_i y la entrada X_i en el tiempo i es la siguiente:

$Y_i = X_i + Z_i,\ Z_i \sim N\left(0, \sigma^2\right),$ i.i.d.

Si elegimos un conjunto discreto de valores, que puede tomar X_i, en el cual los valores pueden separarse arbitrariamente entonces la confiabilidad del canal puede disminuirse tanto como queramos. En la práctica, esto no es posible, debido a limitaciones físicas y una restricción típica, llamada **restricción de potencia promedio**, es:

$$\frac{1}{n}\sum_{i=1}^{n} x_i^2 \leq P$$

donde x_i es el valor que toma, en la posición i, la palabra de código enviada.

En [4], los autores dicen que el hecho de que el canal Gaussiano sea un buen modelo para muchos canales prácticos se debe esencialmente a que, a pesar de que el ruido que agregan los canales físicos puede ser variado y por causas muy distintas, el efecto acumulado de una cantidad grande de pequeños efectos aleatorios, por el teorema del límite central, es aproximadamente Gaussiano.

En [4], se demuestra que la capacidad de este canal es $\frac{1}{2}\log_2\left(1+\frac{P}{\sigma^2}\right)$ bits por uso del canal.

Si limitamos la entrada de un canal AWGN a dos valores posibles, por ejemplo $\{\pm 1\}$, entonces se dice que el canal es Gaussiano de entrada binaria (Binary Input Additive White Gaussian Noise, BIAWGN). La capacidad de este último canal es un poco inferior a la del canal AWGN (por más información, consultar [4]).

El **canal binario simétrico** (Binary Symmetric Channel, BSC), se puede ver como un canal BIAWGN cuantizado, es decir que si el valor a la salida es mayor que 0 entonces la salida se considera 0 y sino se considera 1. Por lo tanto, contrariamente al canal de borradura, el canal BSC corrompe la entrada y puede convertir un 0 en un 1 y vice-versa. El modelo se ilustra en la Figura 27.

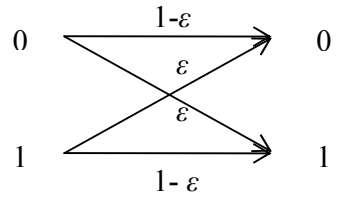

Figura 27

En [30], Shannon demuestra que la capacidad del BSC es: $1-h(\varepsilon)$ bits por dígito binario, donde h representa a la función entropía binaria, $h(\varepsilon)=\varepsilon\log\frac{1}{\varepsilon}+(1-\varepsilon)\log\frac{1}{1-\varepsilon}$.

APÉNDICE C. COMPRESIÓN DE INFORMACIÓN Y CODIFICACIÓN SEPARADA

C.1 Codificación separada

El teorema de la **codificación conjunta de fuente y canal** (también llamado de la separación fuente-canal) fue enunciado primero por Shannon en [1] para fuentes y canales estacionarios sin memoria y luego extendido para una variedad de fuentes y canales por distintos investigadores. El teorema tiene como consecuencia la posibilidad de codificar por separado la fuente y el canal, es decir que el codificador de fuente (respectivamente de canal) no toma en cuenta las estadísticas del canal (respectivamente de la fuente), sin perder nada, asintóticamente, comparado con una codificación conjunta que toma en cuenta tanto la estadística de la fuente como la del canal.

El primer teorema de Shannon (ver la sección C.2) dice que como mínimo es necesario un símbolo $r-$ario por unidad $r-$aria de información. Por lo tanto, si tomamos la salida de un codificador de fuente como entrada de un codificador de canal, a la salida del mismo, tendremos una entropía H, en unidades $r-$arias por símbolo $r-$ario, menor o igual que la tasa R del código.

Por lo tanto, al codificar en forma separada estamos exigiendo que $H \leq R$ y $R < C$ (por el teorema de codificación de canal) lo cual implica $H < C$.

El **teorema de la codificación separada de fuente y canal** dice que es posible transmitir confiablemente la información generada por la fuente si y sólo si $H < C$. Implícitamente, en la parte directa del teorema, está el hecho de que es posible transmitir confiablemente información codificando por separado la fuente y el canal (dado que $H < R$ y $R < C$ implica $H < C$). La parte inversa del teorema implica que se cumple una sola de las siguientes afirmaciones: es posible transmitir confiablemente información codificando por

separado o no es posible hacerlo de ninguna manera (dado que R se puede hacer arbitrariamente cercana a H y a C). Por lo tanto, asintóticamente, no se pierde nada por el hecho de codificar por separado.

C.2 Codificación de fuente

Para lograr comprimir la información es necesario encontrar descripciones de los símbolos de la fuente cuyo largo esperado sea mínimo. La idea básica para lograrlo consiste en asignar descripciones cortas para los símbolos que se emiten por la fuente con más frecuencia y, necesariamente, descripciones más largas para los símbolos menos frecuentes.

Un **código de fuente** C para una variable aleatoria X se define mediante una función biyectiva desde A, el rango de X, hacia D^*, el conjunto de las tiras de largo finito de símbolos pertenecientes a un alfabeto $D-$ario. La codificación se realiza aplicando la función anterior al valor tomado por la variable aleatoria y la decodificación consiste en aplicar la función inversa a una tira resultante de la codificación. Las tiras que devuelve la función de codificación se llaman **palabras de código**. El **largo esperado** del código es el valor esperado del largo de las palabras de código considerando la función de probabilidad de X.

Un código de fuente en un alfabeto binario para el ejemplo del clima puede ser el siguiente:
$\text{Soleado} \leftrightarrow 00, \text{Nublado} \leftrightarrow 01, \text{Lluvia} \leftrightarrow 10, \text{Niebla} \leftrightarrow 11$
En este caso, el largo esperado es igual a 2 dígitos binarios por símbolo. Pero, como veremos más adelante es posible lograr una representación más compacta si la función de probabilidad de los símbolos no es uniforme.

En [30], Shannon estableció el primer resultado fundamental para la teoría de la información que enunciamos en la forma presentada en [10].
Primer Teorema de Shannon: la cantidad promedio de símbolos $r-$arios necesarios para codificar los símbolos de la fuente puede hacerse tan cercana, pero no menor, a la entropía por símbolo de la fuente en unidades de información r-arias.

Dado que la entropía es la mínima cantidad promedio de símbolos necesarios para codificar la fuente, si usamos más símbolos decimos que el código tiene redundancia. Más precisamente definimos la **redundancia** como $\mathrm{P} = L - H$ donde L es el largo promedio del código y H es la entropía por símbolo de la fuente.

Si tomamos el ejemplo del estado del tiempo con las probabilidades dadas en la sección 2.2, entonces la redundancia del código sugerido es $\mathrm{P} \approx 2 - 1.76 = 0.24$ dígitos binarios por palabra.

Una manera de mejorar la codificación de esa fuente sería usando el siguiente código:
$\text{Soleado} \leftrightarrow 0, \text{Nublado} \leftrightarrow 10, \text{Lluvia} \leftrightarrow 110, \text{Niebla} \leftrightarrow 111$
Que tiene un largo esperado de 1.8 dígitos binarios por palabra y por lo tanto una redundancia mucho menor: $\mathrm{P}' \approx 1.76 - 1.8 = 0.04$.
En realidad, no es posible lograr una mejor compresión que ésta, tomando los símbolos de la fuente individualmente. Para acercarse arbitrariamente al límite que establece el teorema es necesario agrupar los símbolos de la fuente y considerar como nuevo alfabeto de la fuente los símbolos originales agrupados en pares, ternas, etc. Por más información, consultar [4] o [10].

Es necesario notar que, dado que el codificador de fuente emite vectores de largo variable y el codificador de canal toma vectores de largo fijo, puede ser necesario algún procesamiento intermedio como, por ejemplo, concatenar los vectores de largo variable para luego dividirlos en vectores de largo fijo.

APÉNDICE D. REGLAS DE DECODIFICACIÓN MAP

Cuando se dispone de las probabilidades a priori de la fuente puede ser conveniente tomarlas en cuenta, para lograr una probabilidad de error más baja que si se asume una distribución uniforme. La regla de **máxima probabilidad a posteriori (MAP) por bloque** minimiza la probabilidad de error por bloque (consultar, por ejemplo, [26]) y se define de la forma siguiente:

$$\begin{aligned}\hat{x}^{MAP}(y) &:= \arg\max_{x\in C} \Pr(x \mid y) \\ &= \arg\max_{x\in C} \Pr(y \mid x)\Pr(x)\end{aligned}$$

La segunda expresión muestra que esta regla es equivalente a la regla de máxima verosimilitud cuando la probabilidad de los vectores de información es uniforme.

La regla de **máxima probabilidad a posteriori por símbolo** minimiza la probabilidad de error por símbolo (consultar, por ejemplo, [11]) y se define de la forma siguiente:

$$\begin{aligned}\hat{x}_i^{MAP}(y) &:= \arg\max_{\alpha\in\mathcal{X}} \Pr(X_i = \alpha \mid Y = y) \\ &= \arg\max_{\alpha\in\mathcal{X}} \sum_{x_i=\alpha} \Pr(y \mid x)\Pr(x)\end{aligned}$$

La segunda expresión muestra que esta regla es equivalente a la regla de máxima verosimilitud por símbolo cuando la probabilidad de los símbolos de información es uniforme.

Al igual que para las reglas de máxima verosimilitud, es necesario definir una política para resolver los empates.

APÉNDICE E. CÓDIGOS LINEALES

Un código C con parámetros (n, M) sobre un cuerpo finito $F = \mathrm{GF}(q)$ es **lineal** si es un subespacio de F^n, o sea que para cualesquiera c_1 y $c_2 \in C$ y $a_1, a_2 \in F$ tenemos que $a_1 c_1 + a_2 c_2 \in C$. El subespacio tiene dimensión $k = \log_q M$. Un código lineal de largo n, dimensión k y distancia mínima d se denota $[n, k, d]$.

Una matriz **generatriz**, denotada, por lo general, G, de un código lineal $[n, k, d]$ sobre un cuerpo finito F es una matriz $k \times n$ cuyas filas constituyen una base del subespacio que constituye el código. Por lo tanto, una manera de codificar es considerando a los vectores de información como las coordenadas de las palabras de código en la base definida por una matriz generatriz. La codificación se define mediante la función $u \to uG$.

Es posible demostrar (ver por ejemplo [26], pág. 36 ej. 1.4) que si una matriz generatriz de un código lineal no tiene columnas de ceros y los vectores de información tienen probabilidad uniforme, entonces la distribución por símbolo es uniforme.

APÉNDICE F. CODIGOS MDS Y COTA DE SINGLETON

La **cota de Singleton** ([33]) determina la máxima distancia mínima que un código puede tener para un largo n, una cardinalidad M y un alfabeto de tamaño q:

$d \leq n - (\log_q M) + 1$

Los códigos que alcanzan la cota con igualdad se dicen separables por máxima distancia (Maximum Distance Separable, **MDS**).

Cuando se introducen k_1 borraduras, al pasar la palabra una palabra de código por un canal de borradura, la tarea del decodificador es determinar la palabra de código a partir de las $n - k_1$ posiciones no borradas. En general, si $k_1 < d$ entonces es posible identificar la palabra enviada dado que cualquier par de palabras difiere en por lo menos d posiciones. Por lo tanto, los códigos MDS son óptimos en el sentido de que permiten recuperar cualquier patrón de a lo sumo $d - 1 = n - k$ borraduras.

La cota de Singleton tiene como consecuencia la siguiente cota ([18]) para la probabilidad de error por bloque para códigos de largo n y dimensión k usando un canal $BEC(\varepsilon)$:

$$P_B(n,k,\varepsilon) \geq \sum_{i=n-k+1}^{n} \binom{n}{i} p^i (1-p)^{n-i}$$

La cota anterior es alcanzada con igualdad por los códigos MDS.

Existe una cota análoga para los casos del canal binario simétrico y del canal gaussiano. Por más información, consultar [18].

APÉNDICE G. COMPLEJIDAD EN LA DECODIFICACIÓN

G.1 Decodificación ML por bloque para el canal BSC

Si intentamos aplicar la regla ML a los códigos lineales sobre un canal $BSC(\varepsilon)$ tenemos que:

$$\begin{aligned}\hat{x}^{ML}(y) &= \arg\max_{x:Hx^T=0} p(y \mid x) \\ &= \arg\max_{x:Hx^T=0} \varepsilon^{d(x,y)}(1-\varepsilon)^{n-d(x,y)} \\ &= \arg\min_{x:Hx^T=0} d(x,y) \\ &= \arg\min_{x:Hx^T=0} w(x+y) \\ &= \arg\min_{e+y:He^T=Hy^T} w(e) \\ &= \arg\min_{e:He^T=s^T} w(e)\end{aligned}$$

donde $w(e)$ denota el peso de Hamming, es decir, $w(e) = d(e,0)$ y en el último paso se usó la definición de **síndrome** $s^T := Hy^T$.

Ahora, consideremos el problema de decisión siguiente:

Instancia: Una matriz binaria $H_{(n-k)\times n}$, un vector $s \in \{0,1\}^{n-k}$ y un entero $w > 0$.

Pregunta: ¿Existe un vector $e \in \{0,1\}^n$ de peso a lo sumo w tal que $He^T = s^T$?

Claramente, este problema queda resuelto una vez que se resuelve el problema de la decodificación ML, por lo tanto el problema de la decodificación es por lo menos tan difícil como este. Este problema de decisión está demostrado ser NP-completo (ver, por ejemplo, [1]), por lo tanto, el problema de la decodificación ML en un canal BSC también lo es.

G.2 Decodificación ML para el canal BEC

Consideremos ahora el problema de la decodificación ML por bloque o por símbolo para el canal de borradura. Sea E el conjunto de las posiciones borradas y $\overline{E}$ el de las posiciones no borradas.

Definimos al conjunto de las palabras de código que coinciden en la parte que no está borrada de la palabra recibida y cumplen $Hx^T = 0$ como

$$X^{ML}(y) \coloneqq \left\{ x \in C : H_E x_E^T = H_{\overline{E}} y_{\overline{E}}^T ; x_{\overline{E}} = y_{\overline{E}} \right\}$$

Es decir que las posiciones borradas son incógnitas de un sistema lineal en el cual el lado derecho se calcula en base a las posiciones no borradas.

Se demuestra fácilmente (ver [26], pág. 74) que el decodificador ML por bloque, que devuelve un error en caso de haber empate, es equivalente, en este caso, a:

$$\hat{x}_B^{ML}(y) = \begin{cases} x \in X^{ML}(y), \text{si } \left| X^{ML}(y) \right| = 1 \\ \text{error, sino} \end{cases}$$

También se demuestra que el decodificador ML por símbolo, que devuelve un error en caso de haber empate, es equivalente a:

$$\hat{x}_i^{ML}(y) = \begin{cases} \alpha, \text{ si } \forall x \in X^{ML}(y), x_i = \alpha \\ \text{error, sino} \end{cases}$$

Hacemos dos observaciones en el caso del canal de borradura:

- para una misma palabra recibida, si el decodificador ML por bloque devuelve algo distinto de “error”, entonces ML por bit devuelve lo mismo. Cuando el decodificador ML por bloque devuelve “error”, el decodificador ML por bit devuelve los bits que pudo recuperar y los demás (que hacen que ML por bloque falle) en error.
- los bits devueltos por ambos algoritmos si no están en error entonces son exactamente lo que fue enviado.

Por lo tanto, suponiendo uniformidad en los vectores de información (respectivamente en los símbolos de información), se puede hacer una decodificación óptima en el sentido de bloque (respectivamente de símbolo) para el canal de borradura resolviendo un sistema lineal de ecuaciones, por lo tanto con complejidad a lo sumo $O\left(n^3\right)$ (por eliminación gaussiana, por ejemplo). Además estos resultados proveen una caracterización de los errores de decodificación en función de condiciones de rango adecuadas.

G.3 Decodificación ML por símbolo para códigos sin ciclos

Existe una subclase de códigos para la cual es posible efectuar una decodificación ML por símbolo con un algoritmo de complejidad lineal en el largo del código.

Si partimos de la regla de decodificación ML por símbolo tenemos:

$$\begin{aligned}\hat{x}_i^{ML}(y) &= \arg\max_{\alpha\in\{0,1\}} \sum_{x\in C|x_i=\alpha} \Pr(y\mid x) \\ &= \arg\max_{x_i\in\{0,1\}} \sum_{\sim x_i} \Pr(y\mid x)\chi_{\{x\in C\}} \\ &= \arg\max_{x_i\in\{\pm 1\}} \sum_{\sim x_i} \left(\prod_{i=1}^{n} p_{Y_i|X_i}(y_i\mid x_i)\right)\chi_{\{x\in C\}} \\ &= \arg\max_{x_i\in\{\pm 1\}} \sum_{\sim x_i} \left(\prod_{i=1}^{n} p_{Y_i|X_i}(y_i\mid x_i)\right)\left(\prod_{l=1}^{n-k}\chi_{\{\sum_{j=1}^{n} H_{lj}x_j=0\}}\right)\end{aligned}$$

Donde χ representa a la función indicatriz y $\sim x_i$ significa que se varían todas las componentes del vector x salvo la $i-$ésima.

Por lo tanto, el problema es equivalente a calcular los marginales de una función factorizada.

El algoritmo Sum-Product, descrito en el Apéndice H, permite calcular en forma eficiente marginales de funciones factorizadas cuya representación en forma de grafo no tiene ciclos.

APÉNDICE H. EL ALGORITMO SUM-PRODUCT

El algoritmo Sum-Product permite calcular, en forma eficiente, marginales[33] de funciones factorizadas cuya representación en forma de grafo no tiene ciclos. Los grafos de Tanner (ver sección 3.2) son un caso particular de representación en forma de grafo de una función factorizada.

La eficiencia que permite lograr este algoritmo está basada en la **ley distributiva generalizada** y consiste en la siguiente igualdad:

$$\sum_{i,j} a_i b_j = \left(\sum_i a_i \right) \left(\sum_j b_j \right) \text{ donde } a_i, b_j \text{ pertenecen a un cuerpo}^{34}$$

H.1 Representación gráfica de funciones factorizadas

La representación gráfica de funciones factorizadas se realiza mediante grafos bipartitos denominados **grafos de factores**. La primera clase de nodos consiste en los denominados **nodos de variable** que corresponden a las variables involucradas en la función. La segunda clase de nodos consiste en los denominados **nodos de factores** que corresponden a los factores de la función. Existe una arista entre un nodo de variable y un nodo de factor si y sólo si la variable participa en el factor.

---

[33] El marginal respecto a x de una función $f(X)$, siendo X un conjunto de variables que contiene a x, es la función $\sum_{X \setminus \{x\}} f(X)$

[34] Hacemos notar que la expresión de la izquierda involucra $|i||j|$ productos y $|i||j|-1$ sumas, mientras que la de la derecha involucra $|i|+|j|-2$ sumas y 1 producto, siendo $|i|$ la cardinalidad del conjunto en el cual toma sus valores el índice i y análogamente para $|j|$.

Por ejemplo, la Figura 28 muestra el grafo de factores correspondiente a la función siguiente:

$$f(x_1,x_2,x_3,x_4,x_5,x_6) = f_1(x_1,x_2,x_3)f_2(x_1,x_4,x_6)f_3(x_4)f_4(x_4,x_5)$$

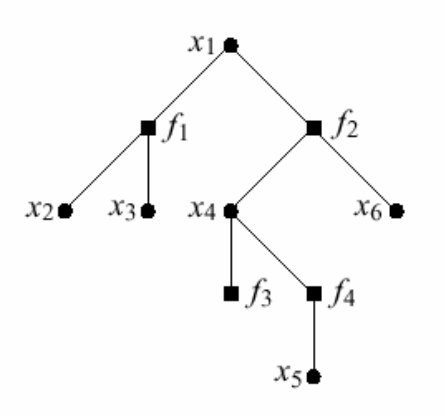

Figura 28

Como veremos a continuación, el grafo de factores es el marco adecuado para aprovechar sistemáticamente el ahorro que brinda la ley distributiva.

H.2 Calculo recursivo de los marginales

Para lograr cálculos exactos para los marginales es necesario que el grafo de factores sea un árbol. Asumiendo que el grafo es un árbol, existe una factorización de la forma siguiente:

$g(z,X_1,...,X_K) = \prod_{k=1}^{K}[g_k(z,X_k)]$, para algún entero K.

Debido a que el grafo es un árbol, se cumple una propiedad crucial para el algoritmo: $z,X_1,...,X_K$ es una partición del conjunto de las variables de g.

La Figura 29 ilustra esta factorización en forma genérica (a la izquierda) y con el ejemplo anterior (a la derecha):

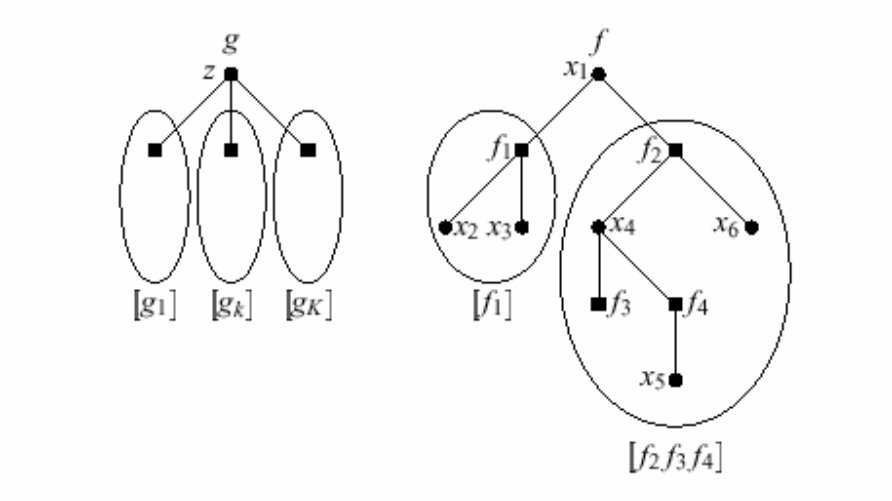

Figura 29

Si aplicamos la ley distributiva al cálculo del marginal respecto a la variable z obtenemos:

$$\sum_{\sim z} g(z,...) = \sum_{\sim z} \prod_{k=1}^{K} [g_k(z,...)] = \prod_{k=1}^{K} \left[\sum_{\sim z} g_k(z,...) \right]$$

Esta transformación ya permite hacer un ahorro en la cantidad de operaciones necesarias pero se puede obtener más aplicando esta idea recursivamente. Dado que se trata de un árbol bipartito, cada g_k debe tener la factorización siguiente:

$$g_k(z, X_k) = h(z, z_1, ..., z_j) \prod_{j=1}^{J} \left[h_j(z_j, X_{kj}) \right]$$

En este caso tenemos la siguiente propiedad: z aparece sólo en la función **kernel** $h(z, z_1, ..., z_j)$ y cada una de las variables z_j aparece a lo sumo dos veces, una vez en el kernel y otra en un factor h_j. Todas las otras variables aparecen solamente en un factor, es decir que $\{z_1, ..., z_j\}, X_{k1}, ..., X_{kj}$ es una partición de X_k.

Cuando una variable z_j aparece solamente en la función kernel, se puede considerar una función h_j ficticia, constante e igual a 1[35].

La Figura 30 ilustra esta factorización en forma genérica y con el ejemplo.

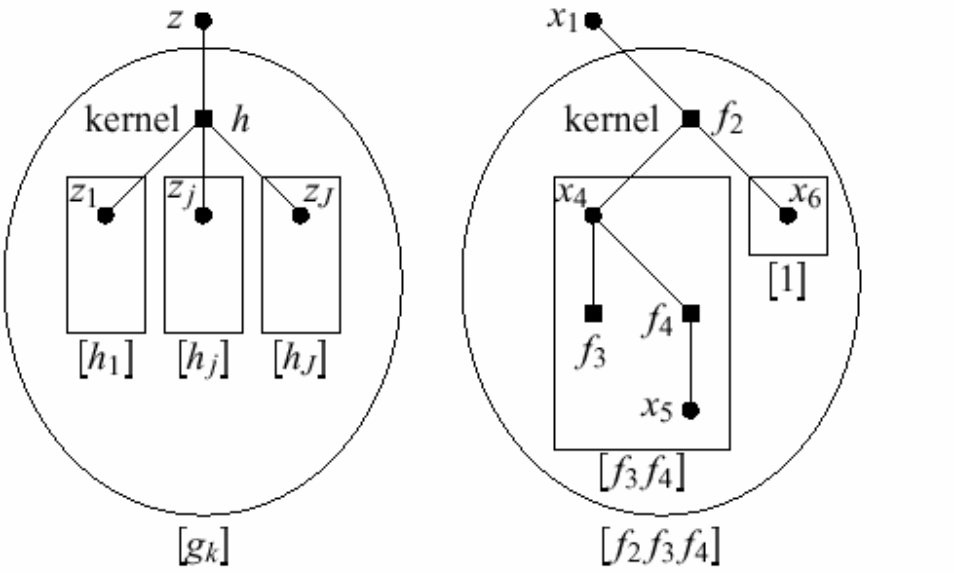

Figura 30

Si aplicamos la ley distributiva al cálculo del marginal respecto a z obtenemos:

$$\sum_{\sim z} g_k(z, X_k) = \sum_{\sim z}\left[h(z, z_1, ..z_j)\prod_{j=1}^{J} h_j(z_j, X_{kj})\right]$$

$$= \sum_{\{z_1,...,z_j\}}\left[h(z, z_1, ..z_j)\prod_{j=1}^{J}\sum_{\sim z_j} h_j(z_j X_{kj})\right]$$

$$= \sum_{\sim z}\left(h(z, z_1, ..., z_j)\prod_{j=1}^{J}\sum_{\sim z_j} h_j(z_j, X_{kj})\right)$$

[35] Esta idea es útil para entender el algoritmo Sum-Product

Cada función h_j tiene la misma forma que la función original $g(z,...)$, por lo tanto, podemos seguir la recursión. Cuando llegamos a una hoja función el cálculo del marginal es trivial:

$$\sum_{\sim z} g_k(z) = g_k(z)$$

Si seguimos la recursión en funciones ficticias $h_j = 1$, entonces el cálculo del marginal queda $\sum_{\sim z_j} 1 = 1$.

H.3 Marginalización eficiente mediante envío de mensajes

Si asumimos que todas las variables toman valores en un cierto alfabeto $\mathcal{X}$, entonces los mensajes son funciones en $\mathcal{X}$, o equivalentemente, si el conjunto $\mathcal{X}$ es finito, los mensajes son vectores de largo $|\mathcal{X}|$[36].

La idea es realizar el cálculo recursivo de marginal, mediante un **algoritmo de envío de mensajes** entre los nodos del grafo de factores que describimos a continuación:

Entrada: Una función $g(...)$ cuya representación en forma de grafo de factores no tiene ciclos y una variable z respecto a la cual se desea marginalizar la función

Salida: $\sum_{\sim z} g(z,...)$

- El pasaje de mensajes se origina en las hojas. Si la hoja es una variable, entonces ella envía como mensaje la función constante igual a 1 correspondiente al marginal de una función ficticia. Si la hoja es una función, se envía a sí misma como mensaje.
- Cuando un nodo, recibe todos los mensajes de sus hijos, calcula el mensaje resultante y lo envía a su padre. Sea $n(v)$ el conjunto de los vecinos de un nodo v. El mensaje que se envía de una variable x a una función f es $\mu_{x\to f}(x) = \prod_{h\in n(x)\setminus\{f\}} \mu_{h\to x}(x)$.

  El mensaje que se envía de la función f a la variable x es

[36] Para cada valor posible de la variable hay que decir que valor toma la función para poder especificarla

$\mu_{f\to x}(x)=\sum_{\sim\{x\}}\left(f(X)\prod_{y\in n(f)\setminus\{x\}}\mu_{y\to f}(x)\right)$ donde $X=n(f)$ es el conjunto de los los argumentos de f.

- Cuando la variable respecto a la cual se desea calcular el marginal, recibe todos los mensajes provenientes de sus hijos, el marginal es $\sum_{\sim z} g(z,...)=\prod_{h\in n(z)}\mu_{h\to z}(z)$.

Por lo tanto, hay una correspondencia directa entre las fórmulas recursivas presentadas en la sección anterior y este algoritmo.

En muchas circunstancias (en particular en el problema de decodificación), se necesita calcular el marginal de la función para más de una variable. Esto podría realizarse aplicando el algoritmo anterior varias veces, una vez para cada variable respecto a la cual se desea calcular el marginal, pero este procedimiento resulta muy ineficiente dado que muchos cálculos se repiten. La idea es "solapar" las diferentes instancias del algoritmo para reutilizar los cálculos. Ningún nodo en particular se considera como padre, por lo tanto no hay una relación padre/hijo fija. En cambio, cada vecino w de un nodo v se considera en algún momento el padre de v y en ese momento el mensaje que se pasa de v a w se calcula como en el algoritmo anterior.

Como en el algoritmo anterior, el pasaje de mensajes se inicia en las hojas. Cada nodo v se mantiene "ocioso" hasta que lleguen mensajes en todas menos una de las aristas de v. En ese momento, v calcula, a partir de los mensajes llegados, el mensaje a enviar por la arista restante (que se considera en forma temporaria como su padre). Esto se repite, considerando cada unos de sus vecinos como el padre. El algoritmo termina cuando dos mensajes pasaron por cada arista, uno en cada sentido. Finalmente, se puede calcular el marginal respecto a cada una de las variables que participan en la función general, multiplicando todos los mensajes llegados en cada variable (igual que en el algoritmo anterior).

Este algoritmo en el cual se calculan todos los marginales a la vez se denomina **Sum-Product**.

Este método es de complejidad lineal en la cantidad de aristas del árbol y por lo tanto es lineal en la cantidad de nodos del árbol. Es importante notar que este método es adecuado para una implementación paralela.

El algoritmo Sum-Product también es aplicable si el grafo es un bosque: hay que aplicarlo por separado a cada una de las componentes conexas.

H.4 Aplicación del algoritmo al problema de la decodificación ML por bit: Belief Propagation

En la sección G.3, vimos que la regla ML es equivalente a calcular el marginal de una función factorizada y, por lo tanto, podemos utilizar el algoritmo Sum-Product en forma exacta cuando el grafo de Tanner correspondiente no tiene ciclos.

A continuación veremos qué forma pueden tomar los mensajes para esta aplicación particular del algoritmo Sum-Product.

La función de densidad de probabilidad para una variable aleatoria binaria se puede representar por un vector (p_0, p_1). De acuerdo a las reglas del Sum-Product, cuando los mensajes (p_0, p_1) y (q_0, q_1) llegan a un nodo de variable de grado 3, el mensaje de salida es:

$VAR((p_0, p_1),(q_0, q_1)) = (p_0 q_0, p_1 q_1)$ (simplemente se multiplican los mensajes)

Cuando dos mensajes (p_0, p_1) y (q_0, q_1) provenientes de x e y respectivamente llegan a un check de grado 3 (que representa a la función $f(x, y, z) = [x \oplus y \oplus z = 0]$), el mensaje de salida hacia z es

$$
\begin{aligned}
CHK((p_0,p_1),(q_0,q_1)) &= \sum_{\sim z}(x \oplus y \oplus z = 0)(p_0,p_1)(q_0,q_1) \\
&= (0 \oplus 0 \oplus z = 0)p_0q_0 + (0 \oplus 1 \oplus z = 0)p_0q_1 \\
&+(1 \oplus 0 \oplus z = 0)p_1q_0 + (1 \oplus 1 \oplus z = 0)p_1q_1 \\
&= (z = 0)p_0q_0 + (1 \oplus z = 0)p_0q_1 \\
&+(1 \oplus z = 0)p_1q_0 + (z = 0)p_1q_1 \\
&= (p_0q_0 + p_1q_1, p_0q_1 + p_1q_0)
\end{aligned}
$$

Dado que $p_0 + p_1 = 0$, podemos representar a las funciones de densidad por un solo valor. Una posibilidad es mediante cocientes de verosimilitud (**likelihood ratios)**, cuya definición es la siguiente:

$$\lambda(p_0,p_1) = p_0/p_1$$

En este caso las reglas se transforman en:

$$VAR(\lambda_1,\lambda_2) = \lambda_1\lambda_2$$

$$CHK(\lambda_1,\lambda_2) = \frac{1+\lambda_1\lambda_2}{\lambda_1+\lambda_2}$$

Otra manera de representar a las funciones de probabilidad es mediante el logaritmo de los cocientes de verosimilitud (**log-likelihood ratios**):

$$\Lambda(p_0,p_1) = \ln(p_0/p_1)$$

En este caso las reglas se transforman en:

$$VAR(\Lambda_1,\Lambda_2) = \Lambda_1 + \Lambda_2$$

$$CHK(\Lambda_1,\Lambda_2) = 2\tanh^{-1}(\tanh(\Lambda_1/2)\tanh(\Lambda_2/2))$$

Es fácil extender estas reglas, para grados mayores que 3. En particular, se pueden extender las reglas mediante las relaciones:

$$VAR(x_1,x_2,...,x_n) = VAR(x_1,VAR(x_2,...,x_n))$$

$$CHK(x_1,x_2,...,x_n) = CHK(x_1,CHK(x_2,...,x_n))$$

APÉNDICE I. PROMEDIO DE LA PROBABILIDAD DE ERROR PARA CONJUNTOS DE LARGO FINITO

Sea $B(v)$ el conjunto de constelaciones inducidas por un conjunto fijo de variables v que contienen stopping sets no triviales, sea $B(v)$ la cardinalidad de ese conjunto. Sea $B(s_{\min}, s_{\max}, s, v) \subseteq B(v)$ el conjunto de constelaciones en un conjunto fijo v de variables cuyo máximo stopping set es de tamaño s y que contiene stopping sets de tamaño perteneciente únicamente al rango $(s_{\min}, s_{\max})$. Para $\chi \in \{0,1\}$, definimos:

$$B(\chi, s_{\min}, s_{\max}, s, v) = \sum_{s} B(s_{\min}, s_{\max}, s, v)\left(\frac{s}{n}\right)^{\chi}$$

También definimos:

$$P(\chi, s_{\min}, s_{\max}, \varepsilon) := \sum_{v=0}^{n}\binom{n}{v}\varepsilon^{v}(1-\varepsilon)^{n-v}\frac{B(\chi, s_{\min}, s_{\max}, v)}{T(v)}$$

donde $T(v)$ es la cantidad de constelaciones en v.

Consideremos primero, el caso en el que $\chi = 0$, tal que $B(\chi = 0, s_{\min} = 1, s_{\max} = n, v) = B(v)$, entonces:

$$P(\chi = 0, s_{\min} = 1, s_{\max} = n, \varepsilon) = \mathrm{E}_{LDPC}\left[P_B^{IT}(G, \varepsilon)\right]$$

Por otro lado, si hacemos $\chi = 1$, entonces $B(\chi = 1, s_{\min} = 1, s_{\max} = n, v)$ cuenta la cantidad de constelaciones pero ponderando con su tamaño relativo entonces tenemos:

$$P(\chi = 1, s_{\min} = 1, s_{\max} = n, \varepsilon) = \mathrm{E}_{LDPC}\left[P_b^{IT}(G, \varepsilon)\right]$$

La determinación de la cantidad $B(...)$, en general, resulta en fórmulas recursivas bastante complejas de evaluar (ver [24], [6] y [26]).

Más generalmente, $P(\chi, s_{\min}, s_{\max}, \varepsilon)$ cuenta la contribución a la probabilidad de error por bit/bloque debida a stopping sets en el rango $(s_{\min}, s_{\max})$. Una aplicación importante de esta generalización es la fabricación de conjuntos expurgados eliminando stopping sets de tamaño pequeño ([24]).

APÉNDICE J. CÁLCULO EFICIENTE DE LA PARIDAD

Sea $x=(s,p_1,p_2)$ una palabra de código tal que s es la parte sistemática y (p_1,p_2) es la redundancia. Por lo tanto, la ecuación $Hx^T=0$ se divide en las ecuaciones:

$$As^T+Bp_1^T+Tp_2^T=0$$
$$(-ET^{-1}A+C)s^T+(-ET^{-1}B+D)p_1^T=0$$

Definimos $\phi:=-ET^{-1}B+D$ y asumimos, por el momento, que es no singular (después veremos que hacer en el caso que sea singular). Entonces tenemos que $p_1^T=-\phi^{-1}(-ET^{-1}A+C)s^T$

Una manera eficiente de calcular p_1 es:

Operación	Comentario	Complejidad
As^T	Multiplicación por una matriz de baja densidad	$O(n)$
$T^{-1}\left[As^T\right]$	$T^{-1}\left[As^T\right]=y^T \Leftrightarrow \left[As^T\right]=Ty^T$ Resolución de un sistema de ecuaciones (baja densidad)	$O(n)$
$-E\left[T^{-1}As^T\right]$	Multiplicación por una matriz de baja densidad	$O(n)$
Cs^T	Multiplicación por una matriz de baja densidad	$O(n)$
$\left[-ET^{-1}As^T\right]+\left[Cs^T\right]$	Suma	$O(n)$
$-\phi^{-1}\left[-ET^{-1}As^T+Cs^T\right]$	Multiplicación por una matriz densa $g\times g$	$O(g^2)$

Una manera eficiente de calcular p_2 es:

Operación	Comentario	Complejidad
As^T	Multiplicación por una matriz de baja densidad	$O(n)$
Bp_1^T	Multiplicación por una matriz de baja densidad	$O(n)$
$\left[As^T\right]+\left[Bp_1^T\right]$	Suma	$O(n)$
$-T^{-1}\left[As^T+Bp_1^T\right]$	$-T^{-1}\left[As^T+Bp_1^T\right]=y^T \Leftrightarrow -\left[As^T+Bp_1^T\right]=Ty^T$	$O(n)$

Estas 2 tablas resumen el algoritmo de codificación que, por lo tanto, puede hacerse con complejidad lineal en n si g es de orden menor o igual que $\sqrt{n}$.

En cuanto al preprocesamiento, para ver si ϕ es singular, en vez de multiplicar por la matriz $\begin{pmatrix} I & 0 \\ -ET^{-1} & I \end{pmatrix}$ se puede hacer eliminación gaussiana para dejar en 0 la parte correspondiente a E. Si ϕ resulta singular es necesario permutar columnas en la matriz para que quede no singular. Esto es posible siempre y cuando H no sea singular.

APÉNDICE K. DESCRIPCIÓN DEL SISTEMA IMPLEMENTADO

En este apéndice, describimos los programas y funciones desarrollados más importantes. El código fuente de estos programas y funciones se encuentra en el Apéndice L, junto con el código fuente de otras funciones de menor importancia.

K.1 Generadores de grafos aleatorios

Generador de grafos aleatorios

Se trata de un programa escrito en C++ que, a partir de un par de distribuciones del punto de vista de los nodos (sin normalizar y con números enteros exclusivamente) especificadas en un par de archivos de texto, genera un conjunto de grafos aleatorios.

El método para generar los grafos es el que se describe en [14]. Este programa puede generar grafos con multiaristas que se pueden quitar, si se desea, importando el grafo a Matlab y tomando los valores de la matriz correspondiente modulo 2 (es conveniente usar la función rem). En el caso de quitar las multiaristas de esta forma, la distribución original se ve afectada; si la cantidad de multiaristas es pequeña comparada con el total de aristas el efecto va a ser despreciable en la distribución.

El programa guarda en archivos los grafos generados, usando una representación de baja densidad. También desarrollamos funciones que permiten hacer la conversión entre el formato de matriz de baja densidad de Matlab y el formato de los grafos almacenados.

Generador de grafos aleatorios sin ciclos en las variables de grado 2

Se trata de una función implementada en Matlab que permite generar grafos aleatorios a partir de distribuciones del punto de vista de los nodos, pero con el requisito de que la cantidad de variables de grado 2 sea inferior o igual a la cantidad de checks para lograr que el subgrafo inducido por las variables de grado 2 sea un árbol.

Además intenta evitar multiaristas, volviendo a sortear cuando una ocurre; esto lo intenta una cierta cantidad de veces que se especifica como parámetro de la función

K.2 Generación y manipulación de distribuciones

Implementamos una función que permite calcular la tasa de diseño de un par de distribuciones normalizadas del punto de vista de las aristas, así como funciones que permiten hacer la conversión entre distribuciones normalizadas del punto de vista de las aristas y distribuciones no normalizadas del punto de vista de los nodos.
Además, implementamos el método numérico explicado en la 3.6, para calcular el umbral de un par de distribuciones.

Para obtener elementos de la secuencia Heavy-Tail Poisson, implementamos una función que, dada una tasa, grafica, en función de N, el grado promedio derecho. Así, es posible ver cual es el N que permite acercarse lo más posible al grado promedio deseado. A partir de ese N y de la tasa, se calcula el parámetro α. Finalmente, a partir de N, α y el grado en el cual se trunca la serie de Taylor para la distribución del grado derecho, implementamos una función que devuelve el par de distribuciones deseado.

En el caso de la distribución Right-Regular, se calcula α a partir del grado derecho deseado y luego se grafica la tasa en función de α y de un rango de N para determinar gráficamente el N que da la tasa más cercana a la deseada.

Las distribuciones optimizadas se obtuvieron a partir de un programa disponible a través de una interfaz web en http://lthcwww.epfl.ch/research/ldpcopt que optimiza distribuciones sujetas a ciertas restricciones que pone el usuario y además guarda en una base de datos resultados de optimizaciones anteriores. Suponemos que el método es similar al que describimos en la sección 3.8.

Varias funciones auxiliares fueron implementadas en Matlab para realizar operaciones sobre distribuciones como dividir las funciones generatrices, integrarlas de 0 a x, integrarlas de 0 a 1, derivarlas, calcular el grado promedio, evaluarlas en un cierto x, etc.

Al pasar un par de distribuciones normalizadas del punto de vista de las aristas a distribuciones del punto de vista de los nodos no normalizadas, los coeficientes de los polinomios por lo general no quedan enteros y redondeando, en general, no se soluciona el problema ya que pueden quedar más aristas de un lado que del otro. No encontramos una solución a este problema en la literatura de los códigos LDPC. La solución que diseñamos y con la cual se obtuvieron buenos resultados en la práctica consiste en plantear el siguiente programa lineal entero:

$$\min \sum_{i \in G_{izq}} \left(e_{izq}^{+}(i) + e_{izq}^{-}(i) \right) + \sum_{i \in G_{der}} \left(e_{der}^{+}(i) + e_{der}^{-}(i) \right)$$

s.a.

$$e_{izq}^{+}(i) - e_{izq}^{-}(i) = \frac{L_i - x_{izq}(i)}{n} \quad i \in G_{izq}$$

$$e_{der}^{+}(i) - e_{der}^{-}(i) = \frac{R_i - x_{der}(i)}{m} \quad i \in G_{der}$$

$$\sum_{i \in G_{izq}} x_{izq}(i).i = \sum_{i \in G_{der}} x_{der}(i).i$$

$$\sum_{i \in G_{izq}} x_{izq}(i) = n$$

$$\sum_{i \in G_{der}} x_{der}(i) = m$$

donde

L_i es la cantidad (con posible parte fraccionaria) de nodos izquierdos de grado i

R_i es la cantidad (con posible parte fraccionaria) de nodos derechos de grado i

$x_{izq}(i)$ son las variables de decisión que representan la cantidad (entera) de nodos izquierdos de grado i

$x_{der}(i)$ son las variables de decisión que representan la cantidad (entera) de nodos derechos de grado i

n es el largo de código que se desea obtener (la cantidad de nodos izquierdos)

m es la cantidad de checks que se desea obtener

G_{izq} es el conjunto de los grados izquierdos

G_{der} es el conjunto de los grados derechos

El programa minimiza la suma de las diferencias normalizadas entre las cantidades "originales" (con parte fraccionaria) de nodos y las cantidades enteras de nodos de cada grado.

El programa fue implementado en What'sBest7.0 para MS Excel.

Para evaluar el resultado se calcula el umbral del par de distribuciones ajustado: debería ser similar al umbral del par original.

K.3 Simulación de canal de borradura

Implementamos, en C++, un simulador de canal $BEC(\varepsilon)$ que introduce errores en un archivo codificado que se le especifica o introduce errores en la palabra de código compuesta únicamente de ceros una cantidad especificada de veces. Es posible especificarle un rango de ε's para que genere varios archivos con distintos niveles de ruido.

K.4 Decodificación

Implementamos, en C++, 3 versiones del decodificador cada una más cuidadosa que la anterior para permitir reducir el tiempo de ejecución. Las tres reciben como parámetros el grafo que representa al código y los archivos codificados con errores y devuelven los archivos decodificados.

K.5 Codificación

Esta parte fue implementada en Matlab y aprovechamos las funcionalidades de Matlab que permiten aprovechar la baja densidad de las matrices.

Fue necesario implementar ciertas funciones auxiliares para trabajar eficientemente con matrices sobre GF(2) tales como sustitución hacia atrás, eliminación gaussiana, inversión y multiplicación de matrices.

Preprocesamiento

Implementamos el primer algoritmo de la serie propuesta en [23] para llevar una matriz H a la forma triangular aproximada. Implementamos este algoritmo más que nada para lograr un mejor entendimiento de las ideas básicas. Luego, implementamos el algoritmo que, en [23], se sugiere usar en la práctica (es el algoritmo presentado en la sección 3.10) y efectivamente los resultados fueron ampliamente mejores obteniendo, en general, gaps entre 1 y 3 tal como lo observaron Richardson y Urbanke.

También implementamos la función que calcula la inversa de ϕ, definida en el Apéndice J.

Codificador

A partir de una matriz que contiene las palabras de información, de la matriz de paridad en forma triangular (sin multiaristas, o sea, binaria) y de la inversa de ϕ, devuelve una matriz con las palabras codificadas correspondientes.

K.6 Comparación de archivos y estadísticas

Se trata de una función escrita en Matlab que permite hacer estadísticas de error, en forma similar a lo que hace el simulador descrito en la siguiente sección pero tomando dos archivos que representan la información codificada sin errores y el resultado de la decodificación.

K.7 Simulación

Se trata de un programa escrito en C++ que a partir de una matriz de paridad, genera una cantidad, que se le especifica, de palabras de código compuestas únicamente por ceros, simula el canal BEC introduciéndole errores a las palabras de código, decodifica y luego estima la probabilidad de error por bit y por bloque del código para distintos parámetros de ruido. Es posible indicarle que solo tome en cuenta los bits sistemáticos al estimar la probabilidad de error.

APÉNDICE L. CÓDIGO FUENTE

En este apéndice, presentamos el código fuente usado para la experimentación descrita en el capítulo 4. En algunos lugares, se referencia el archivo "random.h"; este último contiene los cabezales de funciones que permiten generar números seudo-aleatorios, obtenidas en http://www.math.keio.ac.jp/matumoto/MT2002/emt19937ar.html . La planilla Excel usada para ajustar las cantidades de nodos a números enteros se encuentra en el CD adjunto al informe, junto con todo el código fuente.

L.1 Programas y funciones implementadas en Visual C++ 6.0

Auxiliar.h

Este archivo contiene las definiciones de las estructuras que permiten representar grafos y pares de distribuciones, los cabezales de las funciones de decodificación y de la función de simulación del canal de borradura, entre otras cosas.

```
#include <stdlib.h>
#include <math.h>
#define null 0

struct par{
  int grado;
  int coef;//coef desnormalizado
  long double coefNorm;
};

class dist{
public:
  par* polinomio;
  int largo; // cantidad de coefs distintos de 0
  int cantNodos; //suma de los coeficientes
  int cantAristas; //suma grado*coef


  dist(char * archivo, char tipo_dist);

  void desnormalizar(long double n);

  long double integral0a1();

  void multNorm(long double x);

  static long double aPerspectivaNodos(dist * izq, dist * der);

  void integral0aX();

  void printNorm();
```

```
  void printDesNorm();

};


class lista_sock{
public:

  struct arista{
    int c2v_msg;
    int v2c_msg;
    double c2v_msg_real;
    double v2c_msg_real;

    int check;//check de la arista (posición en el array checks)
    int var;//var de la arista (idem)
    int borrada;//para el algoritmo que borra aristas
  };


  struct socket{ //"nodo" en la lista encadenada de sockets
    arista * arista_ptr;
    socket * siguiente;
  };


  socket * sockPtr;

  void insertar(arista * aPtr){
    socket * aux=sockPtr;

    sockPtr= new socket;
    sockPtr->arista_ptr=aPtr;
    sockPtr->siguiente=aux;
  }

  lista_sock(){
    sockPtr=null;
  }

  void borrar(){
    socket * sig;
    for(socket * aux=sockPtr;aux!=null;aux=sig){
      sig=aux->siguiente;
      delete aux;
    }
    sockPtr=null;
  }
};


class grafo{
public:
  struct nodo{//nodo del grafo
    lista_sock sockets;
    int grado;
  };

  nodo * vars;
  nodo * checks;
```

```
  int cant_vars;
  int cant_checks;
  int cant_aristas;


  grafo(int cantIzq, int cantDer);
  ~grafo();

  void generarMatchingAleat();


  void matchingAGrafo(grafo * m);
};


// operacion XOR
int xor(int a, int b);

int round(long double x);

int iguales(int x[],int y[],int n);

int decod (int y[], int x2[], grafo * gPtr );

int decod2 (int y[], int x2[], grafo * gPtr );

int decod3 (int y[],int x2[], grafo * gPtr, double densityEvol[],int tamDensEvol
);

void canalBorradura (int x[], int y[],int n, double eps_canal);

grafo * init_grafo(char * arch);
grafo * old_init_grafo(char * arch);

void save_grafo(grafo *gPtr, char* arch);
```

Auxiliar.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "auxiliar.h"
#include "random.h"

#define MAXITER 1000

dist::dist(char * archivo, char tipo_dist){
  polinomio= new par[1000];//lo hago bastante largo

  int i=0;
  int grado;


  FILE * stream=fopen(archivo,"r");

  if(tipo_dist=='n'){//norm punto vista aristas
    long double coef;
    while(fscanf(stream,"%i%lf",&grado,&coef)==2){
      polinomio[i].grado=grado;
      polinomio[i].coefNorm=coef;
```

```
      i++;
    }
    largo=i;
  }else if(tipo_dist=='d'){
    int coef;
    int accNodos=0;
    int accAristas=0;
    while(fscanf(stream,"%i%i",&grado,&coef)==2){
      polinomio[i].grado=grado;
      polinomio[i].coef=coef;
      accNodos+=coef;
      accAristas+=grado*coef;
      i++;
    }
    largo=i;
    cantAristas =accAristas;
    cantNodos = accNodos;
  }
}

void dist::desnormalizar(long double n){
  cantNodos=0;
  cantAristas=0;
  for(int i=0;i<largo;i++){
    polinomio[i].coef=round(polinomio[i].coefNorm*n);
    cantNodos+=polinomio[i].coef;
    cantAristas+=polinomio[i].grado*polinomio[i].coef;
  }
}
//calcula la integral entre 0 y 1 de la dist normalizada
long double dist::integral0a1(){
  long double acc=0;
  for(int i=0;i<largo;i++)
    acc+=polinomio[i].coefNorm/(polinomio[i].grado+1);
  return acc;
}

void dist::integral0aX(){
  for(int i=0;i<largo;i++){
    polinomio[i].grado++;
    polinomio[i].coefNorm/=polinomio[i].grado;
  }
}


void dist::multNorm(long double x){//multiplica los coefs normalizados por x
  for(int i=0;i<largo;i++)
    polinomio[i].coefNorm*=x;
}


long double dist::aPerspectivaNodos(dist * izq, dist * der){
  //calculo integrales
  long double integIzq=izq->integral0a1();
  long double integDer=der->integral0a1();

  izq->integral0aX();//integro izq
  izq->multNorm(1/integIzq);

  der->integral0aX();//integro der
  der->multNorm(1/integDer);
```

```
  //devuelvo la tasa
  return 1-integDer/integIzq;

}

void dist::printNorm(){
  for(int i=0;i<largo;i++)
    printf("%lf x**%i + ", polinomio[i].coefNorm,polinomio[i].grado);
  printf("\n");
}

void dist::printDesNorm(){
  for(int i=0;i<largo;i++)
    printf("%i x**%i + ", polinomio[i].coef,polinomio[i].grado);
  printf("\n");
}


// esto es para los grafos

grafo::grafo(int cantIzq, int cantDer){
  cant_vars=cantIzq;
  cant_checks=cantDer;
  cant_aristas=0;
  vars=new nodo[cant_vars];
  checks=new nodo[cant_checks];
  for(int i=0;i<cant_vars;i++)
    vars[i].sockets.sockPtr=0;
  for(int j=0;j<cant_checks;j++)
    checks[j].sockets.sockPtr=0;

}

grafo::~grafo(){
  for(int i=0;i<cant_vars;i++)
    vars[i].sockets.borrar();
  for(int j=0;j<cant_checks;j++)
    checks[j].sockets.borrar();

  delete vars;
  delete checks;
}

void grafo::generarMatchingAleat(){
  //acá no son realmente variables y checks, son nodos izq y nodos der

  int * izqAsig= new int[cant_vars]; // indica para cada nodo izq si fue asignado
o no
  for(int i=0;i<cant_vars;i++)//init
    izqAsig[i]=0;
  int * derAsig= new int[cant_checks];
  for( i=0;i<cant_checks;i++)//init
    derAsig[i]=0;

  for(int cantNoAsig=cant_vars;cantNoAsig>0 ;cantNoAsig--){

    lista_sock::arista * aristaPtr= new lista_sock::arista; //genero arista

    //sorteo uniformemente entre 1 e cantNoAsig  incluidos
```

```
    int pos=genrand_int32()%cantNoAsig + 1;


    int noAsigEnc=0; //cantidad de nodos asignados encontrados hasta el momento
    int i;
    for(i=0;noAsigEnc<pos;i++){
      if(!izqAsig[i])
        noAsigEnc++;
    }

    vars[i-1].sockets.insertar(aristaPtr);
    izqAsig[i-1]=1;


    //lo mismo para el lado derecho

    pos=genrand_int32()%cantNoAsig + 1;

    noAsigEnc=0; //cantidad de nodos asignados encontrados hasta el momento
    for(i=0;noAsigEnc<pos;i++){   //normalmente no debería irse del array...
      if(!derAsig[i])
        noAsigEnc++;
    }

    checks[i-1].sockets.insertar(aristaPtr);
    derAsig[i-1]=1;

  }
}


void grafo::matchingAGrafo(grafo * m){
  int k=0;
  for(int i=0;i<cant_vars;i++){
    for(int j=0;j<vars[i].grado;j++){
      vars[i].sockets.insertar(m->vars[k].sockets.sockPtr->arista_ptr );
      m->vars[k].sockets.sockPtr->arista_ptr->var=i;
      k++;
    }
  }
  k=0;
  for(i=0;i<cant_checks;i++){
    for(int j=0;j<checks[i].grado;j++){
      checks[i].sockets.insertar(m->checks[k].sockets.sockPtr->arista_ptr);
      m->checks[k].sockets.sockPtr->arista_ptr->check=i;
      k++;
    }
  }
}

int xor(int a, int b){
  if((a&&!b)||(!a&&b))
    return 1;
  else
    return 0;
}

int round(long double x){
  return (int)floor(x+0.5);
}
```

```
int iguales(int x[],int y[],int n){
  int igual=1;
  for(int i=0;(i<n)&&(igual);i++)
    if(x[i]!=y[i]){
      igual=0;
      if((y[i]!=-1)&&(x[i]!=-1))//hay un error si entra acá
        n=n;
    }
    return igual;
}

void canalBorradura (int x[], int y[],int n, double eps_canal){

  for(int i=0; i<n;i++){
    if(genrand_real1()>eps_canal)
      y[i]=x[i];
    else{
      y[i]=-1;//borradura
    }
  }

}

int decod2 (int y[],int x2[], grafo * gPtr ){
  int i,j;
  lista_sock::socket  * aux, * recuerdo;

  int contadorIter=0;

  int * XORacc= new int[gPtr->cant_checks ];
  for(j=0;j<gPtr->cant_checks ;j++)
    XORacc[j]=0;

  //hago una copia de los grados para despues modificarla
  int * gradoChecks=new int[gPtr->cant_checks ];
  for(j=0;j<gPtr->cant_checks ;j++)
    gradoChecks[j]=gPtr->checks[j].grado ;


  int * nuevoValor=new int[gPtr->cant_vars];//para mayor eficiencia
  for(i=0;i<gPtr->cant_vars;i++)
    nuevoValor[i]=-2;

  for(j=0;j<gPtr->cant_vars ;j++)
    x2[j]=y[j]; //pongo como value incial el received

  //inicializar todos los chk2var en -1;
  for(j=0;j<gPtr->cant_checks ;j++){
    aux=gPtr->checks[j].sockets.sockPtr ;
    while (aux!=null){//manda su valor de recepción a cada check adyacente
      aux->arista_ptr->c2v_msg =-1;
      aux->arista_ptr->borrada=0; //aprovecho para marcar las aristas como no
borradas
      aux=aux->siguiente;
    }
  }


  //vars mandan valores a checks;//aunque sea -1
  for(j=0;j<gPtr->cant_vars;j++){//cada variable
    aux=gPtr->vars[j].sockets.sockPtr ;
```

```
    while (aux!=null){//manda su valor de recepción a cada check adyacente
      aux->arista_ptr->v2c_msg =y[j];
      aux=aux->siguiente;
    }
  }

  int cambios=1;
  while(cambios){
    contadorIter++;
    cambios=0;
    //cada check lee los valores recibidos
    // y hace xor con acumulador (init en 0) y luego borra arista;
    for(j=0;j<gPtr->cant_checks ;j++){
      aux=gPtr->checks[j].sockets.sockPtr ;
      while (aux!=null){
        if((aux->arista_ptr->v2c_msg!=-1)&&(!aux->arista_ptr->borrada)){
          XORacc[j]=xor(XORacc[j],aux->arista_ptr->v2c_msg);
          //ahora borro arista
          aux->arista_ptr->borrada=1;
          gradoChecks[j]--;
        }else if (!aux->arista_ptr->borrada) //o sea que es -1 y no está borrada
          recuerdo=aux;//recuerdo aux por si despues queda grado 1

        aux=aux->siguiente;
      }
      //el que queda con grado 1 después de borrar aristas, manda su XORacc
      //chk2var está inicializado en -1
      if(gradoChecks[j]==1){

        //puedo hacer todo acá, ya sé que esta se va a cambiar
        cambios=1;
        //esto es para evitar buscar el mensaje cambiador en las aristas
        nuevoValor[recuerdo->arista_ptr->var]=XORacc[j];

        //borro la arista
        recuerdo->arista_ptr->borrada=1;
        gradoChecks[j]=0;
      }
    }

    //cada var cambiada graba su valor y lo manda y se borran las aristas por las
cuales recibió valor
    if(cambios)
      //vars mandan valores a checks
      for(j=0;j<gPtr->cant_vars;j++)//cada variable
        if(nuevoValor[j]>=0){//hubo un cambio en esta variable
          x2[j]=nuevoValor[j];
          nuevoValor[j]=-2;//lo borro
          aux=gPtr->vars[j].sockets.sockPtr ;
          while (aux!=null){//manda su nuevo valor a cada check adyacente
            //da lo mismo considerar si está borrada o no
            aux->arista_ptr->v2c_msg =x2[j];
            aux=aux->siguiente;
          }
        }


  }

  delete (XORacc);
  delete(nuevoValor);
```

```
  delete(gradoChecks);

  return contadorIter;
}




int decod (int y[], int x2[], grafo * gPtr ){
  lista_sock::socket  * aux;
  lista_sock::socket * socket_actual;
  int i,j;

  int contadorIter=0;

  int msg;//con 2 bits me alcanza en realidad

  for(j=0;j<gPtr->cant_vars ;j++)
    x2[j]=y[j]; //pongo como value incial el received

  bool hay_cambios=true;

  for(j=0;j<gPtr->cant_vars;j++){//cada variable
    aux=gPtr->vars[j].sockets.sockPtr ;
    while (aux!=null){//manda su valor de recepción a cada check adyacente
      aux->arista_ptr->v2c_msg =y[j];
      aux=aux->siguiente;
    }
  }

  while (hay_cambios){
    contadorIter++;
    hay_cambios=false;
    for(i=0;i<gPtr->cant_checks;i++){//para cada check
      socket_actual=gPtr->checks[i].sockets.sockPtr;
      //lo que sigue se puede hacer más eficientemente calculando de una cuantas
borraduras
      //hay y el xor del resto, luego es facil eliminar a la arista siendo
considerada
      while(socket_actual!=null){//recorro las aristas del check
        aux=gPtr->checks[i].sockets.sockPtr ;
        msg=0;

        while((aux!=null) && (msg!=-1)){//recorro las aristas del check salteando
la arista_actual
          if(aux!=socket_actual){
            if(aux->arista_ptr->v2c_msg==-1)//si alguno me mandó una borradura ya
está
              msg=-1;
            else
              msg=xor(msg, aux->arista_ptr->v2c_msg);
          }
          aux=aux->siguiente  ;
        }
        socket_actual->arista_ptr->c2v_msg = msg;
        //printf("Check  %i  a  Var  %i  msg:  %i\n",arista_actual->arista_ptr-
>check,arista_actual->arista_ptr->var,msg) ;

        if(msg!=-1){
```

```
          if(msg!=x2[socket_actual->arista_ptr->var]){
            hay_cambios=true;
            x2[socket_actual->arista_ptr->var]=msg;
          }
        }
        socket_actual=socket_actual->siguiente ;
      }

    }

    //esto que viene es inutil hacerlo si no hubieron cambios en el for anterior
    for(j=0;j<gPtr->cant_vars;j++){//para cada variable
      socket_actual=gPtr->vars[j].sockets.sockPtr;
      while(socket_actual!=null){//recorro las aristas de la variable
        aux=gPtr->vars[j].sockets.sockPtr ;

        if(y[j]==-1)
          msg=-1;
        else
          msg=0;


        while((aux!=null)&&(msg==-1)){
          if(aux!=socket_actual)
            if(aux->arista_ptr->c2v_msg !=-1)
              msg=0;
            aux=aux->siguiente ;
        }
        socket_actual->arista_ptr->v2c_msg = (msg==-1?-1: x2[j]);

        //printf("Var  %i  a  Check  %i  msg:  %i\n",arista_actual->arista_ptr-
>var,arista_actual->arista_ptr->check,(msg==-1?-1: decoded[j])) ;


        socket_actual=socket_actual->siguiente ;//ya terminé con esta, sigo
      }

    }
  }

  return contadorIter;
}

grafo * old_init_grafo(char * arch){
  int m,n;

  FILE * stream=fopen(arch,"r");
  //al principio del archivo se deben indicar las dimensiones de la matriz
  fread(&m,sizeof(int),1,stream);
  fread(&n,sizeof(int),1,stream);

  int ** H=new int*[m];
  for(int i=0;i<m;i++)
    H[i]=new int[n];

  for(i=0;i<m;i++)
    fread(H[i],sizeof(int),n,stream);//lee de a filas

  grafo * gPtr=new grafo(n,m);

  for(i=0;i<m;i++){
```

```
    gPtr->checks[i].grado=0;
  }
  for(int j=0;j<n;j++){
    gPtr->vars[j].grado=0;
  }

  for(i=0;i<m;i++){
    for(j=0;j<n ;j++){
      for(int k=0;k<H[i][j];k++){

        lista_sock::arista * aristaPtr= new lista_sock::arista; //genero arista

        aristaPtr->check=i;
        aristaPtr->var=j;

        gPtr->vars[j].sockets.insertar(aristaPtr);
        gPtr->checks[i].sockets.insertar(aristaPtr);

        gPtr->vars[j].grado++;
        gPtr->checks[i].grado++;
        gPtr->cant_aristas++;
      }
    }
  }

  //escribirH(gPtr);//esto es para debug
  return gPtr;
}

void save_grafo(grafo *gPtr, char* arch){
  FILE * stream=fopen(arch,"w");
  fprintf(stream,"%i %i 0\n",gPtr->cant_checks ,gPtr->cant_vars );//dimensiones,
al final hay un 0 para que entienda matlab
  for (int k=0;k<gPtr->cant_vars;k++){
    for(lista_sock::socket          *          skPtr=gPtr->vars[k].sockets.sockPtr
;skPtr!=null;skPtr=skPtr->siguiente){
      fprintf(stream,"%i %i 1\n",skPtr->arista_ptr->check,k);
      //si hay una multiarista al cargar el grafo hay tomarlo en cuenta (en
matlab funciona con spconvert)
    }
  }
  fclose(stream);
}

grafo * init_grafo(char * arch){
  int m,n,v,c,val;

  FILE * stream=fopen(arch,"r");

  int inutil;
  //leo las dimensiones
  fscanf(stream,"%i %i %i",&m,&n,&inutil);//al final hay un 0, para que entienda
matlab

  grafo * gPtr=new grafo(n,m);

  for(int i=0;i<m;i++){
    gPtr->checks[i].grado=0;
  }
  for(int j=0;j<n;j++){
    gPtr->vars[j].grado=0;
```

```
  }

  while(!feof(stream)){
    if(fscanf(stream,"%i %i %i",&c,&v,&val)==3){//fila columna valor. val debería
de ser 1 tanto en el caso en el que proviene de save_grafo como en el caso en el
que proviene de triangulizacion de matlab

      for(int i=0;i<val;i++){//por las dudas si está indicada como multiarista,
no deberia suceder
        lista_sock::arista * aristaPtr= new lista_sock::arista; //genero arista

        aristaPtr->check=c;
        aristaPtr->var=v;

        gPtr->vars[v].sockets.insertar(aristaPtr);
        gPtr->checks[c].sockets.insertar(aristaPtr);

        gPtr->vars[v].grado++;
        gPtr->checks[c].grado++;
        gPtr->cant_aristas++;
      }
    }
  }
  return gPtr;
}

int decod3 (int y[],int x2[], grafo * gPtr,double densityEvol[],int tamDensEvol
){//en este no se usan explicitamente los mensajes
  int j;
  lista_sock::socket  * aux;



  int contadorIter=0;

  int * XORacc= new int[gPtr->cant_checks ];
  for(j=0;j<gPtr->cant_checks ;j++)
    XORacc[j]=0;

  int * gradoChecks=new int[gPtr->cant_checks ];//hago una copia de los grados
para despues modificarla
  for(j=0;j<gPtr->cant_checks ;j++)
    gradoChecks[j]=gPtr->checks[j].grado ;



  int * chksRecordados=new int[gPtr->cant_checks+1];//lista de los checks que
reciben valores <>-1
  //-1 marca el fin de la lista (por eso pongo +1 en el tamaño)
  //inicializo
  int * varsRecordadas=new int[gPtr->cant_vars+1];//lista de vars que reciben
valores <>-1

  int k=0;
  for(j=0;j<gPtr->cant_vars;j++){
    x2[j]=y[j]; //pongo como valor inicial el recibido

    if(y[j]!=-1){
      varsRecordadas[k]=j;
      k++;
    }
```

```
    for(aux=gPtr->vars[j].sockets.sockPtr;aux!=null;aux=aux->siguiente)//manda su
valor de recepción a cada check adyacente
      aux->arista_ptr->borrada=0; //aprovecho para marcar las aristas como no
borradas

  }
  varsRecordadas[k]=-1;

  int cambios=1;
  while(cambios){

    if(densityEvol!=0){//viene en null si no quiero hacer esto
      int cantMensErr=0;
      for(j=0;j<gPtr->cant_vars  ;j++){//me fijo cuales van a mandar -1 en BP
        if(x2[j]==-1)
          cantMensErr+=gPtr->vars[j].grado;//en belief prop se manda por todas
las aristas
      }
      if(contadorIter<tamDensEvol)//para no irme del array
        densityEvol[contadorIter]=(double)cantMensErr/(double)gPtr->cant_aristas;
    }

    cambios=0;
    int c=0;
    for(k=0;varsRecordadas[k]!=-1 ;k++){//sólo proceso las que recibieron valores
      j=varsRecordadas[k];
      for(aux=gPtr->vars[j].sockets.sockPtr;aux!=null;aux=aux->siguiente){
        if(!aux->arista_ptr->borrada){
          XORacc[aux->arista_ptr->check]=xor(XORacc[aux->arista_ptr-
>check],x2[j]);
          //ahora borro arista
          aux->arista_ptr->borrada=1;
          gradoChecks[aux->arista_ptr->check]--;
          if(gradoChecks[aux->arista_ptr->check]==1){
            chksRecordados[c]=aux->arista_ptr->check; //después puede quedar en
cero: verificar después (igual no va a andar mal)
            c++;
          }
        }
      }
    }
    chksRecordados[c]=-1;

    contadorIter++;


    int v=0;
    for(k=0;chksRecordados[k]!=-1 ;k++){//sólo proceso los que recibieron valores
      j=chksRecordados[k];
      if(gradoChecks[j]==1){//si después de ser recordado pasó a cero no hago
        int encontre=0;
        for(aux=gPtr->checks[j].sockets.sockPtr;(aux!=null)&&(!encontre);aux=aux-
>siguiente)//busco arista de grado 1
          if(!aux->arista_ptr->borrada){
            //el que queda con grado 1 después de borrar aristas, manda su XORacc
//chk2var está inicializado en -1
            cambios=1;
            encontre=1;
            //le pongo el valor
            x2[aux->arista_ptr->var]=XORacc[j];
```

```
            varsRecordadas[v]=aux->arista_ptr->var;
            v++;
            aux->arista_ptr->borrada=1;
            gradoChecks[j]=0;
          }
      }
    }
    varsRecordadas[v]=-1;
  }


  delete (XORacc);
  delete(gradoChecks);
  delete(chksRecordados);
  delete(varsRecordadas);

  return contadorIter;
}
```

gengrafo.cpp

Este archivo contiene la implementación del programa que permite generar grafos aleatorios.

```
#include "..\auxiliar.h"
#include "..\random.h"
#include <stdio.h>
#include <string.h>
#include <time.h>


grafo * generar_codigo_aleatorio(dist izq,dist der){//izq y der son
distribuciones no normalizadas punto de vista de los nodos

  grafo * g= new grafo(izq.cantNodos,der.cantNodos);


  //asigno grado a los nodos (acá no hay nada aleatorio)
  int j=0;//con esta recuerdo en donde había quedado
  for(int i=0;i<izq.largo;i++){//recorro el polinomio
    for(int k=0;k<izq.polinomio[i].coef; k++){
      g->vars[j].grado=izq.polinomio[i].grado;
      j++;
    }
  }

  j=0;
  for(i=0;i<der.largo;i++){
    for(int k=0;k<der.polinomio[i].coef; k++){
      g->checks[j].grado=der.polinomio[i].grado;
      j++;
    }
  }

  //metodo de Luby
  grafo * m= new grafo(izq.cantAristas,izq.cantAristas);

  m->generarMatchingAleat();
```

```
  g->matchingAGrafo(m);

  delete m;

  return g;

}

int main(int argc, char* argv[]){

  if(argc!=6){
    printf("Uso : genGrafo archIzq archDer cant_codigos semilla prefijo\n");
    printf("Pone los códigos generados en <prefijo>1.cod,<prefijo>2.cod,... con
formato binario (enteros)\n");
    exit(1);
  }

  long semilla=atol(argv[4]);
  int cant_codigos=atoi(argv[3]);
  char prefijo[100];
  strcpy(prefijo,argv[5]);

  dist *izq = new dist(argv[1], 'd');
  dist *der = new dist(argv[2], 'd');

  if(izq->cantAristas!=der->cantAristas ){
    printf("El par de distribuciones es incorrecto\n");
    exit(1);
  }

  init_genrand(semilla);

  for(int i=0;i<cant_codigos;i++){
    printf("Generando código %i\n",i);
    grafo * gPtr;
    gPtr=generar_codigo_aleatorio(*izq,*der);

    char buffer[100];
    char nombre[100];
    nombre[0]=0;
    strcat(nombre,prefijo);
    strcat(strcat(nombre,itoa(i,buffer,10)),".cod");

    save_grafo(gPtr,nombre);

    delete gPtr;

  }

}
```

canalBEC.cpp

Este archivo contiene la implementación del programa que simula el canal de borradura

```
#include <stdlib.h>
#include <stdio.h>
#include "..\random.h"
#include <time.h>
#include <string.h>
#include "..\Auxiliar.h"
```

```
int main(int argc, char* argv[]){
  //   clock_t c0=clock();


  if(argc<7){
    printf("Uso para generar errores en palabras 0: canalBEC 0 cantPals n semilla
eps_desde eps_hast eps_delta\n");
    printf("Uso para usar archivo: canalBEC 1 archCodificado semilla eps_desde
eps_hast eps_delta\n");
    exit(1);
  }

  FILE * stream_in,* stream_out;


  int todo0=!atoi(argv[1]);
  int cantPals;
  int shift=0;
  int n;
  char * archCodif;
  if(todo0){
    cantPals=atoi(argv[2]);
    n=atoi(argv[3]);
    shift=1;
  }else
    archCodif=argv[2];


  long semilla=atol(argv[3+shift]);
  double eps_desde=atof(argv[4+shift]);
  double eps_hasta=atof(argv[5+shift]);
  double eps_delta=atof(argv[6+shift]);

  init_genrand(semilla);

  if(!todo0){
    stream_in=fopen(archCodif,"r");
    fread(&cantPals,sizeof(int),1,stream_in);
    fread(&n,sizeof(int),1,stream_in);
  }

  int * info=new int[cantPals*n];
  int * infoConErrs=new int[cantPals*n];

  if(todo0){
    for(int i=0;i<cantPals*n;i++)
      info[i]=0;
  }else{
    fread(info,sizeof(int),cantPals*n,stream_in);
    fclose(stream_in);
  }

  for(double eps=eps_desde;eps<=eps_hasta;eps+=eps_delta){
    char buffer[100];
    char buffer2[50];
    if(!todo0)
      strcpy(buffer,archCodif);
    else
      strcpy(buffer,"ceros");
```

```
    strcat(buffer,".eps_");
    sprintf(buffer2,"%f",eps);
    strcat(buffer,buffer2);

    stream_out=fopen(buffer,"w");

    canalBorradura(info,infoConErrs,cantPals*n,eps);

    fwrite(&cantPals,sizeof(int),1,stream_out);
    fwrite(&n,sizeof(int),1,stream_out);
    fwrite(infoConErrs,sizeof(int),cantPals*n,stream_out);
    fclose(stream_out);
  }


  //   clock_t c1=clock();
  //printf("Cant ciclos de reloj %li\n",c1-c0);
}
```

Bec3.cpp

Este archivo contiene la implementación del programa que decodifica archivos con borraduras

```
#include <stdio.h>
#include <string.h>
#include "auxiliar.h"

#define MAXITER 5000

int main(int argc, char* argv[]){
  //   clock_t c0=clock();

  double densEvol[MAXITER];//acá guardo density evol
  //x(l) es fracción de mensajes de error pasados en la iteración l left-to-right


  if(argc<4){
    printf("Uso : decod matHext algor archCodificado1 archCodificado2 ...\n");
    exit(1);
  }

  char* matHext=argv[1];
  int algor=atoi(argv[2]);

  char * archCodConErrs[100];

  int i;
  for(i=3;i<argc;i++){
    archCodConErrs[i-3]=argv[i];
  }
  int cantArchs=i-3;


  grafo * gPtr=init_grafo(matHext);//a partir de H


  //para cada archivo
```

```
  //leo al principio cuantas palabras tiene
  //voy leyendo de a palabra, decodifico y voy escribiendo resultado en archivo
de salida


  for(int j=0;j<cantArchs;j++){
    FILE * stream_in=fopen(archCodConErrs[j],"r");
    char buffer[100];
    buffer[0]=0;
    strcat(buffer,archCodConErrs[j]);
    strcat(buffer,".decod");
    FILE * stream_out=fopen(buffer,"w");

    int cantPals,n;
    fread(&cantPals,sizeof(int),1,stream_in);//leo cantidad de palabras que hay
en archivo
    fread(&n,sizeof(int),1,stream_in);//leo largo de las palabras que hay en
archivo
    if(n!=gPtr->cant_vars){
      printf("el largo de las palabras no coincide con el grafo\n");
      exit(1);
    }

    fwrite(&cantPals,sizeof(int),1,stream_out);
    fwrite(&n,sizeof(int),1,stream_out);

    double iterAcc=0;//cuento cuantas iteraciones se hicieron en total para el
archivo
    int * y=new int[gPtr->cant_vars];//acá pongo palabras a decodificar
    int * x2=new int[gPtr->cant_vars];//acá pongo palabras decodificadas

    FILE * stream=fopen("densEvol.txt","w");//si hay varios códigos se
sobreescribe
    fprintf(stream,"M=[");


    for(int k=0; k<cantPals; k++){
      fread(y,sizeof(int),n,stream_in);//leo proxima palabra a decodificar

      int it;

      if(algor==1)
        iterAcc+=decod(y,x2,gPtr);
      else if(algor==2)
        iterAcc+=decod2(y,x2,gPtr);
      else{
        it=decod3(y,x2,gPtr,densEvol,MAXITER);
        iterAcc+=it;
        //print result
        for(int l=0;l<it;l++){
          fprintf(stream,"%lf ",densEvol[l]);
        }
        fprintf(stream,";\n");
      }



      fwrite(x2,sizeof(int),n,stream_out);

    }
```

```
    fprintf(stream,"]\n");

    delete(x2);
    delete(y);

    printf("Promedio de iteraciones para archivo %s:
%lf\n",archCodConErrs[j],iterAcc/cantPals);

    fclose(stream_in);
    fclose(stream_out);
    fclose(stream);

  }
  //   clock_t c1=clock();

  //printf("Cant ciclos de reloj %li\n",c1-c0);
}
```

simulación.cpp

Este archivo contiene el programa que hace la simulación del sistema de codificación.

```
#include <stdlib.h>
#include <stdio.h>
#include "auxiliar.h"
#include "random.h"
#include "Comun\BeliefProp.h"
#include <time.h>
#include <string.h>


#define MAXITER 500

//esto es para hacer el dibujito del grafo con plotgraph, hace mat de adyacencia
nodos nodos
void matlab_grafo(grafo * gPtr){
  lista_sock::socket * skPtr;
  FILE * stream=fopen("matlab.txt","w");

  int ** mat= new int* [gPtr->cant_vars+gPtr->cant_checks];
  for(int i=0;i<gPtr->cant_vars+gPtr->cant_checks;i++)
    mat[i]=new int [gPtr->cant_vars+gPtr->cant_checks];

  //inicializo matriz
  for(i=0;i<gPtr->cant_vars+gPtr->cant_checks;i++){
    for(int k=0;k<gPtr->cant_vars+gPtr->cant_checks;k++)
      mat[i][k]=0;
  }


  for (i=0;i<gPtr->cant_vars;i++){
    for(skPtr=gPtr->vars[i].sockets.sockPtr ;skPtr!=null;skPtr=skPtr->siguiente)
      mat[i][gPtr->cant_vars+skPtr->arista_ptr->check]=1;
  }

  for (i=0;i<gPtr->cant_checks;i++){//esto es para verificar
    for(skPtr=gPtr->checks[i].sockets.sockPtr           ;skPtr!=null;skPtr=skPtr-
>siguiente)
      mat[gPtr->cant_vars+i][skPtr->arista_ptr->var]=1;
  }
```

```
  //matriz de adyacencia (nodos-nodos)
  fprintf(stream,"mat=[\n");
  for (i=0;i<gPtr->cant_vars+gPtr->cant_checks;i++){
    for (int j=0;j<gPtr->cant_vars+gPtr->cant_checks;j++){
      fprintf(stream,"%i " ,mat[i][j]);
    }
    fprintf(stream,";\n");
  }
  fprintf(stream,"]\n");

  //labels
  fprintf(stream,"lab={");
  for (i=0;i<gPtr->cant_vars;i++)
    fprintf(stream,"'%i' ",i);
  for (i=0;i<gPtr->cant_checks;i++)
    fprintf(stream,"'%i' ",i);
  fprintf(stream,"}\n");

  //circulos-cuadrados
  fprintf(stream,"boxcirc=[ ");
  for (i=0;i<gPtr->cant_vars;i++)
    fprintf(stream,"0 ");
  for (i=0;i<gPtr->cant_checks;i++)
    fprintf(stream,"1 ");
  fprintf(stream,"]\n");

  //coordenadas x
  fprintf(stream,"x=[ ");
  for (i=0;i<gPtr->cant_vars;i++)
    fprintf(stream,"0.1 ");
  for (i=0;i<gPtr->cant_checks;i++)
    fprintf(stream,"0.8 ");
  fprintf(stream,"]\n");

  //coordenadas y
  fprintf(stream,"y=[ ");
  float intervV=0.8/gPtr->cant_vars;
  float intervC=0.8/gPtr->cant_checks;

  for (i=0;i<gPtr->cant_vars;i++)
    fprintf(stream,"%g ",0.1+intervV*i);

  for (i=0;i<gPtr->cant_checks;i++)
    fprintf(stream,"%g ",0.1+intervC*i);

  fprintf(stream,"]\n");
}


int distHamming(int x[],int x2[],int n){
  int errs=0;
  for(int i=0;i<n;i++){
    if(x[i]!=x2[i])
      errs++;
  }
  return errs;

}

void confianzaBinomialProm2(int errBit, int cantPals, double & Pbit, double &
low, double & upp, int n){
```

```
  int cantMuestras=cantPals*n;

  //Spielman
  Pbit=(double)errBit/cantMuestras;
  low=(errBit>0?Pbit*exp(-2*sqrt((1-Pbit)/(Pbit*cantMuestras))):0);
  upp=(errBit>0?Pbit*exp(2*sqrt((1-Pbit)/(Pbit*cantMuestras))):1-exp((double)-
2/cantMuestras));


}


void acumularErrores(int * errBit,int * x,int * x2,int n){
  for(int i=0;i<n;i++){
    if(x[i]!=x2[i])
      errBit[i]++;
  }
}


int simulacion_1_codigo(int & sigo, grafo * gPtr, double eps_canal, long
cant_pals, int algor,double & Pbit, double & confBitLow,double & confBitUpp, int
sist){
  //double errores=0;//puse double para que al final no me haga división de
enteros

  int * x=new int[gPtr->cant_vars];
  int * y=new int[gPtr->cant_vars];
  int * x2=new int[gPtr->cant_vars];

  int errBit=0;
  int errs;

  int MIN_ERR=100;

  //la performance del algoritmo es indep de la palabra
  for(int i=0;i<gPtr->cant_vars;i++){
    x[i]=0;
  }

  double iterAcc=0;

  for(i=0;((i<cant_pals)||sigo)&&(errBit<MIN_ERR);i++){

    if(algor<5)
      canalBorradura(x,y,gPtr->cant_vars ,eps_canal);
    else if(algor>=5)
      BSC(x,y,gPtr->cant_vars,eps_canal);

    if(algor==1)
      iterAcc+=decod(y,x2,gPtr);
    else if(algor==2)
      iterAcc+=decod2(y,x2,gPtr);
    else if(algor==3)
      iterAcc+=decod3(y,x2,gPtr,0,0);
    else if(algor==4)
      iterAcc+=beliefPropBEC(y,x2,gPtr,100);
    else if(algor==5){
      if(i==0)
        iterAcc+=beliefPropBSC(0,y,x2,gPtr,eps_canal,100);
```

```
      else
        iterAcc+=beliefPropBSC(0,y,x2,gPtr,eps_canal,100);
    }else if(algor==6)
      iterAcc+=beliefPropBSCModif(6, y,x2,gPtr,eps_canal, 100);

    if(!sist)
      errs=distHamming(x,x2,gPtr->cant_vars );
    else
      errs=distHamming(x,x2,gPtr->cant_vars    -    gPtr->cant_checks);//considero
primera parte nada mas (largo n-m)

    if(errs>0){
      //errBlock++;
      errBit+=errs;
    }

  }

  int cantPalsEfectivas=i;

  confianzaBinomialProm2(errBit,cantPalsEfectivas,Pbit,confBitLow,
confBitUpp,(!sist?gPtr->cant_vars :gPtr->cant_vars - gPtr->cant_checks)   );


  if(Pbit>0)
    sigo=1; //en el proximo eps no me puede dar 0

  delete(x);
  delete(y);
  delete(x2);

  printf("\t\t\t\t\t\t\tPromedio de iteraciones %lf\n",iterAcc/cant_pals);

  return cantPalsEfectivas;
}



int main(int argc, char* argv[]){
  clock_t c0=clock();

  if(argc!=9){
    printf("Uso : sim archivoMatrizH eps_desde eps_hast eps_delta cant_palabras
algor semilla sistematico\n");
    printf("Si es sistematico se toma en cuenta solamente los errores ocurridos
en la parte sistematica\n");
    exit(1);
  }

  char * archMatrizH=argv[1];
  double eps_desde=atof(argv[2]);
  double eps_hasta=atof(argv[3]);
  double eps_delta=atof(argv[4]);
  int cant_pals=atoi(argv[5]);
  int algor=atoi(argv[6]);
  long semilla=atol(argv[7]);
  int sist=atoi(argv[8]);

  init_genrand(semilla);

  grafo * gPtr;
```

```
  gPtr=init_grafo(archMatrizH);//a partir de H
  char buffer[100];
  strcpy(buffer,archMatrizH);
  strcat(buffer,".res.txt");

  FILE * stream=fopen(buffer,"w");

  for(int i=1;i<argc;i++)
    fprintf(stream,"%s ",argv[i]);
  fprintf(stream,"\n");


  fprintf(stream,"M=[");


  double Pb=1; //para que entre en el for
  for(double eps=eps_hasta;(eps>=eps_desde)&&(Pb>0);eps-=eps_delta){

    printf("Simulando para eps=%lf\n",eps);

    double low, upp;
    int sigo=0;

    int  cantPalsEf=simulacion_1_codigo(sigo,  gPtr,  eps,cant_pals,algor,Pb,low,
upp,sist);

    fprintf(stream,"%lf %le %le %le \n",eps,Pb, low, upp);

    printf("Pbit=%le cant_pals=%i\n\n",Pb,cantPalsEf);

  }

  fprintf(stream,"]\n");

  fclose(stream);

  clock_t c1=clock();
  printf("Cant ciclos de reloj %li\n",c1-c0);
}
```

L.2 Funciones implementadas en Matlab

Funciones para manejar distribuciones

```
function alpha=alphaRightReg(avgRightDeg)
% alpha=alphaRightReg(avgRightDeg)
% A partir del grado derecho promedio deseado devuelve el parámetro alpha
% que se debe usar para generar un par de distribuciones right-regular
% con ese grado derecho promedio

alpha=1/(avgRightDeg-1);
```

```
function alpha=alphaTornado(N,R)
% alpha=alphaTornado(N,R)
% A partir del indice N de la secuencia Heavy-Tail Poisson y de la tasa deseada
```

```
% devuelve el parámetro alpha que se debe usar para generar un par de
distribuciones
% Heavy-Tail Poisson con esos parámetros

aux=N*H(N-1)/(N-1);

eq=strcat('=1-((1-exp(-x))/x)*',num2str(aux));
eq=strcat(num2str(R),eq);
alpha=eval(solve(eq));
alpha=alpha(1);
```

```
function ar=AvgRightDegTornado(R,N)
% ar=AvgRightDegTornado(R,N)
% a partir de la tasa R y el indice N en la secuencia Heavy-Tail Poisson
% devuelve el grado derecho promedio del par de distribuciones correspondiente

ar=N*H(N-1)/((N-1)*(1-R));
```

```
function cant=cantAristas(dist)
% cant=cantAristas(dist)
% devuelve la cantidad de aristas a partir de la distribución desnormalizada
especificada

cant=sum(dist(:,1).*dist(:,2))
```

```
function d=derivada(dist)
% d=derivada(dist)
% deriva el polinomio correspondiente a la distribución especificada

dist(:,2)=dist(:,2).*dist(:,1);
dist(:,1)=dist(:,1)-1;

d=dist;
```

```
function [L,R]=distribuciones(H)
% [L,R]=distribuciones(H)
% a partir de la matriz H devuelve las distribuciones L,R desnormalizadas
correspondientes

%izquierda
gradoI=sum(H);
gradoD=sum(H');

cantAristas=sum(gradoI);
distI=zeros(1,cantAristas);
distD=zeros(1,cantAristas);

for i=1:size(H,2)
   distI(gradoI(i))=distI(gradoI(i))+1;
end

for i=1:size(H,1)
   distD(gradoD(i))=distD(gradoD(i))+1;
end

L=[];
R=[];
```

```
for i=1:length(distI)
   if distI(i)>0
      L=[L ; i distI(i)];
   end
end

for i=1:length(distD)
   if distD(i)>0
      R=[R ; i distD(i)];
   end
end
```

```
function res=dividir(dist,num)
% res=dividir(dist,num)

res(:,1)=dist(:,1);
res(:,2)=dist(:,2)/num;
```

```
function g=gradoPromedio(dist)
% g=gradoPromedio(dist)
% calcula el grado promedio de la distribucion normalizada del punto de vista de
las aristas

g=1/integral0a1(dist);
```

```
function res=integral0a1(dist)
% calcula integral de 0 a 1 de dist

aux=integral(dist);
res=sum(aux(:,2));
```

```
function res=integral(dist)
% calcula la integral de la distribucion de 0 a x

res(:,1)=dist(:,1)+1;
res(:,2)=dist(:,2)./res(:,1);
```

```
function [lambda,rho]=pasarAPersAristas(L,R)
% pasa las distribuciones desnormalizadas del punto de vista de los nodos a
distribuciones
% desnormalizadas del punto de vista de las aristas

derivL=derivada(L);
derivR=derivada(R);

lambda=dividir(derivL,polyvaldist(derivL,1));
rho=dividir(derivR,polyvaldist(derivR,1));
```

```
function [L,R]=pasarAPersNodo(lambda,rho,n)
% [L,R]=pasarAPersNodo(lambda,rho,n)
% a partir de distribuciones normalizadas del punto de vista de las aristas
% y el largo deseado, devuelve las distribuciones desnormalizadas del punto
% vista de los nodos
```

```
rate=tasa(lambda,rho);

L = dividir(dividir(integral(lambda),integral0a1(lambda)),1/n);
R = dividir(dividir(integral(rho),integral0a1(rho)),1/(n*(1-rate)));
```

```
function plotAvgRightDegTornado(R,Nhasta)
% dada una tasa R, grafica el grado derecho promedio de la secuencia
% Heavy-Tail Poisson en función de N

for N=2:Nhasta
   ar(N-1)=N*H(N-1)/((N-1)*(1-R));

end

plot(2:Nhasta,ar);
```

```
function plotRRightReg(alpha,Nhasta)
% a partir del parametro alpha grafica la tasa de cada elemento de la secuencia
% right regular

for N=2:Nhasta
   r(N-1)=(N/alpha)*nchoosekR(alpha,N)*(-1)^(N-1)*(1-1/N)/(1-
(1/N)*(N/alpha)*nchoosekR(alpha,N)*(-1)^(N-1));

end

plot(2:Nhasta,r);
```

```
function [lambda, rho]=rightRegular(N, alpha)
% devuelve el par de distribuciones de la secuencia Right Regular con parametro N
y alpha

for i=1:N-1 lambda(i,:)=[i nchoosekR(alpha,i)*((-1)^(i+1))/(1-
(N/alpha)*nchoosekR(alpha,N)*((-1)^(N-1)))]; end;

rho=[1/alpha 1];
```

```
function r=tasa(lambda,rho)
% calcula la tasa del par de distribuciones

r=1-integral0a1(rho)/integral0a1(lambda);
```

```
function t=thresholdBEC2(lambda,rho,epsInicial,precision,shannon,tol)
% t=thresholdBEC2(lambda,rho,epsInicial,precision,shannon,tol)
% dado el par de distribuciones calcula el umbral correspondiente

t=epsInicial;

negativa=1;
while negativa & t<=shannon
   %verifico que la curva sea negativa
    t=t+precision;

   x=precision;
```

```
   while negativa & (x<=1)

      if t*polyvaldist(lambda,1-polyvaldist(rho, 1-x))-x>tol %errores de redondeo
         negativa=0;
      else
         x=x+precision;
      end
   end

end

if ~negativa
   t=t-precision;
end
```

```
function [lambda, rho]=tornado(N, trunco, alpha)
% devuelve el par de dists de la secuencia Heavy-Tail Poisson con parametros N y
alpha
% truncando la serie correspondiente la lado derecho en el sumando trunco

for i=1:N-1 lambda(i,:)=[i 1/(H(N-1)*i)]; end;

for i=0:trunco rho(i+1,:)=[i exp(-alpha)*(alpha^i)/factorial(i)]; end;

rho(:,2)=rho(:,2)/sum(rho(:,2));
```

Generador de grafos sin ciclos

```
function H=grafoSinCiclosDeg2NoMultiEff(L,R,seed,maxIter)
% H=grafoSinCiclosDeg2NoMultiEff(L,R,seed,maxIter)
% genera grafo sin ciclos en variables de grado 2
%este es el bueno, usa MERSENNE
%->tiene que haber por lo menos el doble de checks que de vars de grado 2
%NO PUEDEN HABER CHECKS DE GRADO 1, sino se corta

%randint(0,0,[],seed);
mersenne([-1,seed]);

m=sum(R(:,2));
i=find(L(:,1)==2);
n2=L(i,2);%cantidad de nodos de grado 2

L2=[L(1:i-1,:) ; L(i+1:size(L,1),:)];

H = sparse(m,n2);%todos ceros

checksNoAsignados=1:m;
grado=zeros(m,1);%acá pongo los grados de los checks

c=1;
for i=1:size(R,1)
   for j=1:R(i,2)%le asigno a los checks el i-esimo grado
      grado(c)=R(i,1);
      c=c+1;
   end
end
```

```
primeraDeSerie=1;%i es la variable actual
varsAAsignar=1;%por ahora tengo que asignar la primera

primeravez=1;

v2=2;%esta es la primera variable disponible del resto

while varsAAsignar>0 %si quedan variables a asignar %no puede quedar en 0 antes
de haber llegado a todas las vars

   for v=primeraDeSerie:primeraDeSerie+varsAAsignar-1
      v
      %sorteo los dos hijos de v
      %indiceC=randint(1,1,[1 length(checksNoAsignados)]);
      indiceC=mersenne(length(checksNoAsignados));
      c1=checksNoAsignados(indiceC);
      checksNoAsignados=[checksNoAsignados(1:indiceC-
1),checksNoAsignados(indiceC+1:length(checksNoAsignados))];%lo borro
      H(c1,v)=1;
      grado(c1)=grado(c1)-1;

      if primeravez
         %indiceC=randint(1,1,[1 length(checksNoAsignados)]);
         indiceC=mersenne(length(checksNoAsignados));

         c2=checksNoAsignados(indiceC);
         checksNoAsignados=[checksNoAsignados(1:indiceC-
1),checksNoAsignados(indiceC+1:length(checksNoAsignados))];%lo borro
         H(c2,v)=1;
         grado(c2)=grado(c2)-1;
      end


      %g=1;
      while (v2<=n2)&(grado(c1)>0) %mientras hayan vars disponibles y mientras
tenga sockets el check
         H(c1,v2)=1;
         grado(c1)=grado(c1)-1;
         v2=v2+1;
       %  g=g+1;
      end


      if primeravez
         %g=1;
         while (v2<=n2)&(grado(c2)>0) %mientras hayan vars disponibles y mientras
tenga sockets el check
            H(c2,v2)=1;
            grado(c2)=grado(c2)-1;
            v2=v2+1;
            %g=g+1;
         end
         primeravez=0;
      end

   end
   primeraDeSerie=primeraDeSerie+varsAAsignar;
   varsAAsignar=v2-primeraDeSerie;
end

%hago matriz para el resto
```

```
n3=sum(L2(:,2));
H2=sparse(m,n3);%todos ceros

%asigno grado a las vars
gradoIzq=zeros(n3,1);%acá pongo los grados de los checks

v=1;
for i=1:size(L2,1)
   for j=1:L2(i,2)%le asigno a las vars el i-esimo grado, L2 no tiene las de
grado 2
      gradoIzq(v)=L2(i,1);
      v=v+1;
   end
end

%hago una vector todos los sockets disponibles si el check 56 tiene 2 disponibles
pongo 56 56
%socketsDisp=[];
%for i=1:length(grado)
%   for j=1:grado(i)
%      socketsDisp=[socketsDisp i];
%   end
%end

%warning off;
cantSockets=sum(grado);

for v=1:n3
   v
   aristasActuales=[];
   for g=1:gradoIzq(v)
      %socketSorteado=randint(1,1,[1 cantSockets]);
      socketSorteado=mersenne(cantSockets);

      %busco el check sorteado
      i=0;
      s=0;%con esta recorro sockets
      while socketSorteado>s
         i=i+1;
         s=s+grado(i);
      end
      checkSorteado=i;

      iter=0;
      if ~isempty(aristasActuales)
         while ~isempty(find(aristasActuales==checkSorteado)) & (iter<maxIter)
%ya está, vuelvo a sortear
            %socketSorteado=randint(1,1,[1 cantSockets]);
            socketSorteado=mersenne(cantSockets);


            %busco el check sorteado
            i=0;
             s=0;%con esta recorro sockets
          while socketSorteado>s
             i=i+1;
             s=s+grado(i);
          end
          checkSorteado=i;

            iter=iter+1;
```

```
          end
       if iter==maxIter
          warning('Multiarista!!');
       end
      end

      grado(checkSorteado)=grado(checkSorteado)-1;
      cantSockets=cantSockets-1;
      aristasActuales=[aristasActuales checkSorteado];
   H2(checkSorteado,v)=H2(checkSorteado,v)+1;
   end
end

%warning on;
H=[H H2];
```

Generador de grafos sin multi-aristas

```
function H=grafoNoMulti(L,R,seed,maxIter)
% H=grafoNoMulti(L,R,seed,maxIter)
% a partir de las distribuciones desnormalizadas, genera un grafo
% intentando evitar multiaristas maxIter veces


%randint(0,0,[],seed);
mersenne([-1,seed]);

n=sum(L(:,2));
m=sum(R(:,2));

H = sparse(m,n);%todos ceros

grado=zeros(m,1);%acá pongo los grados de los checks

c=1;
for i=1:size(R,1)
   for j=1:R(i,2)%le asigno a los checks el i-esimo grado
      grado(c)=R(i,1);
      c=c+1;
   end
end

%asigno grado a las vars
gradoIzq=zeros(n,1);%acá pongo los grados de los checks

v=1;
for i=1:size(L,1)
   for j=1:L(i,2)%le asigno a las vars el i-esimo grado, L2 no tiene las de grado
2
      gradoIzq(v)=L(i,1);
      v=v+1;
   end
end

cantSockets=sum(grado);

for v=1:n
   v
```

```
   aristasActuales=[];
   for g=1:gradoIzq(v)
      %socketSorteado=randint(1,1,[1 cantSockets]);
      socketSorteado=mersenne(cantSockets);

      %busco el check sorteado
      i=0;
      s=0;%con esta recorro sockets
      while socketSorteado>s
         i=i+1;
         s=s+grado(i);
      end
      checkSorteado=i;

      iter=0;
      if ~isempty(aristasActuales)
         while ~isempty(find(aristasActuales==checkSorteado)) & (iter<maxIter)
%ya está, vuelvo a sortear
           %socketSorteado=randint(1,1,[1 cantSockets]);
           socketSorteado=mersenne(cantSockets);


           %busco el check sorteado
           i=0;
            s=0;%con esta recorro sockets
          while socketSorteado>s
             i=i+1;
             s=s+grado(i);
          end
          checkSorteado=i;

            iter=iter+1;
          end
       if iter==maxIter
          warning('Multiarista!!');
       end
      end

      grado(checkSorteado)=grado(checkSorteado)-1;
      cantSockets=cantSockets-1;
      aristasActuales=[aristasActuales checkSorteado];
   H(checkSorteado,v)=H(checkSorteado,v)+1;
   end
end
```

Lectura y escritura de grafos y matrices

```
function escribirMatrizSparse(mat)
% escribirMatrizSparse(mat)
% guarda la matriz en un formato similar al usado por sim.exe
% siempre guarda en codigo.cod
% para que quede usable por sim.exe, pegar el contenido de codigo.cod
% en excel, delimitar columnas y poner formato numero, luego pegar en archivo de
texto

[I, J]=find(mat>0);

salvame=zeros(length(I)+1,3);
```

```
salvame(1,1)=size(mat,1);
salvame(1,2)=size(mat,2);

for i=1:length(I)
   salvame(i+1,1)=I(i)-1; %los indices en el archivo tienen que ser a partir de 0
   salvame(i+1,2)=J(i)-1;
   salvame(i+1,3)=mat(I(i),J(i));
end

%ojo que siempre salva en codigo.cod, después hay que pasarlo a excel, delimitar
las columnas y poner formato numero
%sin decimales, después pegar en archivo .cod
save codigo.cod salvame  -ASCII;
```

```
function M=leerMatrizSparse(arch)
% guarda en M, el grafo leido a partir de arch (formato de gengrafo.exe)

mat=load(arch);
mat(2:size(mat,1),1:2)=mat(2:size(mat,1),1:2)+1;

M=spconvert(mat);
```

```
function escribirMatriz(archivo,mat)
% escribirMatriz(archivo,mat)

fid=fopen(archivo,'w');

fwrite(fid,size(mat),'integer*4');%leo M y N

fwrite(fid,mat','integer*4');%mi programa lee por filas: al revés de como escribe
matlab

fclose(fid);
```

```
function mat=leerMatriz(archivo)
%mat=leerMatriz(archivo)

fid=fopen(archivo,'r');

[dim,COUNT] = fread(fid,2,'integer*4');%leo M y N

[mat,COUNT] = fread(fid,fliplr(dim'),'integer*4');%mi programa graba por filas:
al revés de como lee matlab
mat=sparse(mat);
mat=mat';

fclose(fid);
```

Preprocesamiento y codificación

```
function [A,B,C,D,E,T,res,g]=practAproxTriang(Agnoqui)
%[A,B,C,D,E,T,g]=practAproxTriang(Agnoqui,alpha)
%AGnoqui no debe tener multiaristas
```

```
%practical greedy algorithm
%no hay ni filas ni columnas de grado 0

%para verificar que la matriz hst haya quedado aprox triang inferior con rowGap=g
%largoDiag=size(hst,1)-g;
%NumDiag=size(hst,2)-largoDiag+1;
%I=triu(hst,NumDiag);I=I(1:largoDiag,NumDiag+1:size(I,2));
%find(sum(I)~=1)

%para verificar que quedó con la misma dist que Agnoqui
%hallar grado de cada var y grado de cada check , en definitiva rearmar dist
%para eso usar find(histc(full(sum(hs)), [0:250])~=histc(full(sum(hst)),
[0:250])) para las vars
%y find(histc(full(sum(hs')), [0:250])~=histc(full(sum(hst')), [0:250])) para los
checks


largoDiag=0;
temp=[];
parteDerecha=[];

Agnoqui=Agnoqui'; %voy a trabajar con la transpuesta
cantFilas =size(Agnoqui,1);
cantColumnas=size(Agnoqui,2);

%comentario: Agnoqui en ningún momento puede tener solo ceros, porque arriba de
Agnoqui hay siempre ceros o la matriz vacía y eso significaría que hubiera checks
de grado 0

while ~isempty(Agnoqui)
   %busco filas de grado 1,2,3...
   %known(1,i)=1 si la columna i está conectada a una fila de grado 1 y
known(2,i) dice cual es esa fila

   %extiendo Agnoqui para saber grado de las vars

   %HT=[temp [zeros(size(temp,1)-size(Agnoqui,1),size(Agnoqui,2));Agnoqui]
parteDerecha];%asi va quedando
   if ~isempty(temp)
      AgExt=[temp(size(temp,1)-size(Agnoqui,1)+1:size(temp,1),:) Agnoqui
parteDerecha(size(parteDerecha,1)-size(Agnoqui,1)+1:size(parteDerecha,1),:)];
   else
      AgExt=Agnoqui;
   end

   i=find(sum([AgExt';zeros(1,size(AgExt,1))])==2);%busco filas de grado 2 en la
totalidad de la matriz
   Agnoqui2=Agnoqui(i,:);

   known=sparse(zeros(2,size(Agnoqui,2)));
   pos=0;

   if ~isempty(Agnoqui2) %intento buscar en las de grado 2 primero
      for i=find(sum([Agnoqui2';zeros(1,size(Agnoqui2,1))])==1);%este find busca
las filas de grado 1, le agrego zeros para que funcione sum en el caso de que
Agnoqui sea un vector columna
         pos=find(Agnoqui2(i,:)==1);
         known(1,pos)=1;%la columna pos esta conectada
         known(2,pos)=i;%a la fila i de grado 1
      end
   end
```

```
   if pos==0 %no encontré en las de grado 2, busco por todos lados
      for i=find(sum([Agnoqui';zeros(1,size(Agnoqui,1))])==1);%este find busca
las filas de grado 1, le agrego zeros para que funcione sum en el caso de que
Agnoqui sea un vector columna
         pos=find(Agnoqui(i,:)==1);
         known(1,pos)=1;%la columna pos esta conectada
         known(2,pos)=i;%a la fila i de grado 1
      end
   end

   if pos==0 %quiere decir que no encontre filas de grado 1
      grado=2;

      while pos==0 & grado<=size(Agnoqui,2) %busco grados mayores (busco una
sola). Tiene que encontrar algo. no pueden haber de grado 0.
         i=find(sum(Agnoqui')==grado); %acá no hace falta  hacer el truquito de
los ceros como cuando busco nodos de grado 1, en este caso no puedo tener un
vector columna (si es vector columna encuentra vars de grado 1 asumiendo que no
hay multiaristas)

         if ~isempty(i)
            %busco  en qué columnas están los unos
            pos=find(Agnoqui(i(1),:)==1);

            % los primeros grado-1 unos los tiro para la derecha: swapeo
columnas, no tengo que swapear en el pedazo de arriba porque hay ceros
            cont=0;
            for k=1:length(pos)-1;
               cont=cont+1;
               j=pos(k);
               aux=Agnoqui(:,size(Agnoqui,2)-cont+1);
               Agnoqui(:,size(Agnoqui,2)-cont+1)=Agnoqui(:,j);
               Agnoqui(:,j)=aux;


               %tengo que cambiar pos también si alguno estaba ahí!!
               i2=find(pos==(size(Agnoqui,2)-cont+1));%si i2 es <=k no calienta,
ya lo pasé
               pos(i2)=j;
            end

            if (cantFilas-size(Agnoqui,1))>0 %sino devuelve una matriz 0xcont, no
devuelve [], no sé porque
               ceros=zeros(cantFilas-size(Agnoqui,1),cont);
            else
               ceros=[];
            end

            parteDerecha=[[ceros;Agnoqui(:,size(Agnoqui,2)-
cont+1:size(Agnoqui,2))] ;parteDerecha];

            %pelo Agnoqui
            Agnoqui=Agnoqui(:,1:size(Agnoqui,2)-cont);

            %me quedo con el último uno para poner en la diagonal
            known=sparse(zeros(2,size(Agnoqui,2)));

            known(1,pos(length(pos)))=1;
             known(2,pos(length(pos)))=i(1);
       else
```

```
            grado=grado+1;
          end


       end

   end

   %ahora puedo aplicar diagonal extension aunque sea a un elemento

   %diagonal extension

   j=1;

   columnasRestantes=find(known(1,:));
   %al estar trabajando con la transpuesta puedo encontrar más filas de grado 1
que columnas en AGnoqui
   if length(columnasRestantes)>size(Agnoqui,2)
      columnasRestantes=columnasRestantes(1:size(Agnoqui,2));
   end


   while ~isempty(columnasRestantes)

      i=columnasRestantes(1);

      %swap row OJO ACA hay que swapear en toda la matriz, sino queda cualquier
cosa
      aux=Agnoqui(known(2,i),:);
      Agnoqui(known(2,i),:)=Agnoqui(j,:);
      Agnoqui(j,:)=aux;

      %ojo en temp las coordenadas son diferentes, hay que sumar largoDiag para
obtener la fila
      if ~isempty(temp) %si todavia no le puse nada a temp no tengo que hacer
nada
       aux=temp(known(2,i)+largoDiag,:);
       temp(known(2,i)+largoDiag,:)=temp(j+largoDiag,:);
       temp(j+largoDiag,:)=aux;
      end

      %cambio parte derecha , tambien son distintas las coordenadas
      if ~isempty(parteDerecha)
       aux=parteDerecha(known(2,i)+largoDiag,:);
       parteDerecha(known(2,i)+largoDiag,:)=parteDerecha(j+largoDiag,:);
       parteDerecha(j+largoDiag,:)=aux;
       end

      %hay que permutar en known, porque known(2,:) se refiere a filas y las
estoy cambiando !!!
      known(2,find(known(2,:)==j))=known(2,i);

      %swap col, hay que permutar known!!
      aux=Agnoqui(:,i);
      Agnoqui(:,i)=Agnoqui(:,j);
      Agnoqui(:,j)=aux;

      aux=known(:,i);
      known(:,i)=known(:,j);
      known(:,j)=aux;
      known(1,j)=0;%marco a la columna como procesada.
```

```
      columnasRestantes=find(known(1,:));

      if sum(sum(Agnoqui(1:j,j+1:size(Agnoqui,2))))~=0
         j=j;
      end

      j=j+1;
   end



   j=j-1;%asi me da el tamaño de la identidad tiene que ser igual
size(find(known(2,:)),2)

   %la diagonal creció en j lugares
   largoDiag=largoDiag+j;

   %para saber donde va
    largoDiag


   %le quito las columnas known
   temp=[temp, [zeros(size(temp,1)-size(Agnoqui,1),j); Agnoqui(:,1:j)]];
   Agnoqui=Agnoqui(j+1:size(Agnoqui,1),j+1:size(Agnoqui,2));
end

%Ahora tengo que juntar temp con la parte derecha
%y dar vuelta las cosas
res=fliplr(rot90([temp,parteDerecha]));
g=size(res,1)-largoDiag;

%llevo el pedacito para arriba (abajo, no?)
res=[res(g+1:size(res,1),:) ; res(1:g,:)];

m=size(res,1);
n=size(res,2);

A=res(1:m-g,1:n-m);
B=res(1:m-g,n-m+1:n-m+g);
C=res(m-g+1:m,1:n-m);
D=res(m-g+1:m,n-m+1:n-m+g);
T=res(1:m-g,n-m+g+1:n);
E=res(m-g+1:m,n-m+g+1:n);
```

```
function invPHI=preproc2(A,B,C,D,E,T)
% segunda parte del preprocesamiento

%acá se podría verificar que H no sea singular

[PHI toda]=eliminacionGF2(A,B,C,D,E,T);

%PHI queda sparse, no sirrve
PHI=full(PHI);

%PHI es g*g
g=size(PHI,1);
%gfrank2 es gfrank pero con el bug corregido
```

```
if gfrank2(PHI)~=g %problema de rango, hay que permutar columnas (si H no es
singular)
   invPHI=-1;
   return;
end

invPHI=gflineq2(PHI,eye(size(PHI)));%esta es la funcion que modifique
if sum(sum(mod(invPHI*PHI,2)-eye(size(PHI))))~=0 %no confio mucho en ella
   invPHI=-1;
   return;
end
```

```
function matCod=codificar(matInf,A,B,C,E,T,invPHI)
% matCod=codificar(matInf,A,B,C,E,T,invPHI)
% devuelve las palabras de código correspondientes a la matriz
% de información matInf especificada
%las filas de las matrices matInf y matCod son las palabras
%todas las matrices son binarias
%T,A,B,C y E son sparse

matCod=zeros(size(matInf,1),size(A,2)+size(B,2)+size(T,2));

for i=1:size(matInf,1)
   AST=mod(A*matInf(i,:)',2);
   y=backSubst(T,AST);
   %y no debería ser sparse

   p1=(-invPHI*(-E*y +C*matInf(i,:)'))';
   p1=mod(p1,2);

   aux=mod(-(AST + B*p1'),2);
   p2=backSubst(T,aux)';

   matCod(i,:)=[matInf(i,:) p1 p2];
   i
end
```

Comparación de archivos y estadísticas

```
function [Pbit, Pblock, PbitSyst,
PblockSyst]=estadistica(matDecod,matCod,cantChecks)
% en base a un archivo codificado y al archivo decodificado correspondiente
% calcula el estimador de la prob de error por bit y por bloque considerando
unicamente
% los bits de información o todos los bits

sist=size(matCod,2)-cantChecks; %n-m

errores=matDecod~=matCod;

Pblock=sum(sum(errores')>1)/size(matCod,1);

Pbit=sum(sum(errores))/(size(matCod,1)*size(matCod,2));

PblockSyst=sum(sum(errores(:,1:sist)')>1)/size(matCod,1);

PbitSyst=sum(sum(errores(:,1:sist)))/(size(matCod,1)*size(matCod,2));
```

```
sum( errores)
```

INDICE

REFERENCIAS

[1] Berlekamp E. R., McEliece R. J. y Van Tilborg H. C. A., "On the inherent intractability of certain coding problems", IEEE Trans. Inform. Theory, 2424, pp. 384-386, 1978.

[2] Berrou C., Glavieux A., y Thitimajshima P., "Near Shannon limit error-correcting coding and decoding", en Proceedings of ICC'93, Geneva, Switzerland, May 1993, pp. 1064-1070.

[3] Byers J., Luby M., Mitzenmacher M. y Rege A., "A digital fountain approach to reliable distribution of bulk data" en Proceedings of ACM SIGCOMM '98, 1998.

[4] Cover T. M. y Thomas J. A., Elements of Information Theory, John Wiley & Sons, ISBN 0-471-06259-6, 1991.

[5] Davey M. C. y MacKay D. J. C., Low density parity check codes over GF(q), IEEE Communications Letters, 2 (1998).

[6] Di C., Proietti D., Teletar E., Richardson T. y Urbanke R., "Finite-length analysis of low-density codes on the binary erasure channel", IEEE Trans. Inform. Theory, vol. 48, pp. 1570-1579, 2002.

[7] Elias P., "Coding for two noisy channels", in Information Theory, Third London Symposium, pp. 61-76, 1955.

[8] Etzion T., Trachtenberg A. y Vardy A., "Which codes have cycle-free Tanner graphs?", IEEE Trans. Inform. Theory, vol. 45, pp. 2173-2181, 1999.

[9] Gallager R.G., "Low Density Parity-Check Codes".MIT Press, Cambridge, MA, 1963.

[10] Hamming R., Coding and Information Theory, Prentice-Hall, ISBN 0-13-139139-9, 1980.

[11] Hartmann C., Rudolph L., "An optimum symbol-by-symbol decoding rule for linear codes", IEEE Trans. Inform. Theory, vol. IT-22, No. 5, pp. 514-517, 1976.

[12] Kiely A., Coffey J., Bell M., "Optimal information bit decoding of linear block codes", IEEE Trans. Inform. Theory, vol. 41, pp. 130-140, 1995.

[13] Kschischang F. R., Frey B. J. y Loeliger H., "Factor graphs and the sum-product algorithm", IEEE Trans. Inform. Theory, vol. 47, pp. 498-519, 2002.

[14] Luby M., Mitzenmacher M., Shokrollahi A. y Spielman D., "Efficient erasure correcting codes", IEEE Trans. Inform. Theory, vol. 47, pp. 569-584, 2001.

[15] Luby M., Mitzenmacher M., Shokrollahi A. y Spielman D., "Improved low-density parity-check codes using irregular graphs", IEEE Trans. Inform. Theory, vol. 47, pp. 585-598, 2002.

[16] Luby M., Mitzenmacher M., Shokrollahi A., Spielman D. y Stemman V., "Practical loss-resilient codes", Proc. 29th Annu. ACM Symp. Theory of Computing, 1997, pp. 150–159.

[17] MacKay D. J. C. y Neal R. M., Good codes based on very sparse matrices, en Cryptography and Coding. 5th IMA Conference, C. Boyd, ed., no. 1025 in Lecture Notes in Computer Science, Springer, Berlin, 1995, pp. 100-111.

[18] MacMullan S. J. y Collins O. M., "A comparison of known codes, random codes, and the best codes", IEEE Trans. Inform. Theory, vol. 44, pp. 3009-3022, 1998.

[19] McEliece R. J., "Achieving the Shannon limit: A progress report", Thirty-Eighth Allerton Conference, 2000.

[20] Oswald P. y Shokrollahi A., "Capacity-achieving sequences for the erasure channel", disponible en http://shokrollahi.com/amin/pub.html, 2000.

[21] Plank J.S. y Thomason M. G., "On the practical use of LDPC erasure codes for distributed storage applications", disponible en http://www.cs.utk.edu/˜plank/plank/papers/CS-03-510.html .

[22] Ramamoorthy A. y Wesel R., "Construction of short block length irregular low-density parity-check codes", disponible en http://www.ee.ucla.edu/~adityar/documents/adiicc2004.pdf .

[23] Richardson T. y Urbanke R., "Efficient Encoding of low-density parity-check codes", IEEE Trans. Inform. Theory, vol. 47, pp. 638-656, 2001.

[24] Richardson T. y Urbanke R., "Finite-length density evolution and the distribution of the number of iterations for the binary erasure channel", disponible en http://lthcwww.epfl.ch/publications/index.php

[25] Richardson T. y Urbanke R., "The capacity of low-density parity check codes under message-passing decoding", IEEE Trans. Inform. Theory, vol. 47, pp. 585-598, 2001.

[26] Richardson T. y Urbanke R., Modern Coding Theory, en borrador, versión de Noviembre 2003, la última versión está disponible en http://lthcwww.epfl.ch/papers/ics.ps

[27] Richardson T., Shokrollahi A. y Urbanke R., "Design of capacity-approaching irregular low-density parity-check codes", IEEE Trans. Inform. Theory, vol. 47, pp. 619-637, 2001.

[28] Rizzo L., "Effective erasure codes for reliable computer communication protocols", disponible en http://citeseer.nj.nec.com/rizzo97effective.html

[29] Sason I. y Urbanke R., "Parity-check density versus performance of binary linear block codes over memoryless symmetric channels", IEEE Trans. Inform. Theory, vol. 49, pp. 1611-1635, 2003.

[30] Shannon C.E., "A Mathematical Theory of Communication", The Bell System Technical Journal,Vol. 27, pp. 379-423, 1948.

[31] Shokrollahi A., "LDPC Codes: An Introduction", disponible en http://shokrollahi.com/amin/pub.html , 2003.

[32] Shokrollahi A., "Raptor Codes", disponible en http://www.inference.phy.cam.ac.uk/mackay/dfountain/RaptorPaper.pdf .

[33] Singleton R. C., "Maximum distance q-nary codes", IEEE Trans. Information Theory IT-10 , pp. 116 –118, 1964.

[34] Song H., Todd R. M. y Cruz J. R., "Low density parity check codes for magnetic recording channels", IEEE Trans. on Magnetics, Vol. 36, No. 5, Setiembre 2000.

[35] Spielman D., "Lecture 5", disponible en http://www-math.mit.edu/~spielman/ECC/lect5.pdf .

[36] Tanner R. M., "A recursive approach to low complexity codes", IEEE Trans. Inform. Theory, 27 (1981), pp. 533-547.

[37] Tian T., Jones C., Villasenor J. D. y Wesel R. D., “Construction of irregular LDPC codes with low error floors”, disponible en http://www.ee.ucla.edu/faculty/papers/villa_icc03_vol.5.pdf .

[38] Weisstein Eric W. et al. "Gauss-Seidel Method." disponible en http://mathworld.wolfram.com/Gauss-SeidelMethod.html .