UNIVERSIDAD
DE GRANADA

Facultad de Ciencias

Grado en Matemáticas

TRABAJO DE FIN DE GRADO

El problema P=NP. Aplicaciones a la criptografía

Presentado por:
Miguel Pinel Martínez

Curso académico 2023-2024

El problema P=NP. Aplicaciones a la criptografía

Miguel Pinel Martínez

Miguel Pinel Martínez *El problema P=NP. Aplicaciones a la criptografía*.
Trabajo de fin de Grado. Curso académico 2023-2024.

Responsable de tutorización

Gabriel Navarro Garulo
Ciencias de la computación e IA

Grado en Matemáticas
Facultad de Ciencias
Universidad de Granada

Declaración de originalidad

D. Miguel Pinel Martínez

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2023-2024, es original, entendido esto en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 21 de agosto de 2024

Fdo: Miguel Pinel Martínez

Si en otras ciencias quisieramos obtener la verdad sin errores y la certeza sin dudas nos correspondería basar nuestro conocimiento en las matemáticas.

Roger Bacon, Matemáticas, Sid Meier's Civilization IV

Índice general

Summary

The main focus of this project will be to prove the NP-completeness of some well-known problems. Them we will show how is it possible to create cryptosystem based on this problems, basing the security of the system on their NP-completeness. Lastly, we will attack this cryptosystem with an heuristic approach. We will use genetic and memetic algorithm. The results will basically show the sturdiness of this systems against a direct approach to attack them. We will also illustrate the reason why directs attacks, specifically genetic algorithms, are not the best suited to break the security of this systems.

In this project we will study the Turing machine as a mathematical model of an abstract machine and will used them to evaluate the complexity of problems. It will be defined the bases of the complexity theory as well as the Turing machine and their main modifications. We will use the Turing machine to define the complexity of a problem and will present the complexity classes of decisions problems and their relationships. We will focus on the NP-complete problems and how costly it is to solve them and will relate that to the P versus NP open problem. The main NP-complete problems discussed will be: the subset sum problem, a particular case of the knapsack problem, and the decoding of a general linear code. It will also include the SAT problem, Boolean Satisfiability Problem, and the discussion of the Cook's theorem, to utilize a Turing reduction of this problem to the subset sum and the decoding of a linear code problems to prove their NP-completeness.

The project will include a brief introduction to cryptography, explaining its importance and the difference between symmetric and asymmetric cryptography. The main cryptosystems discussed in this project will be asymmetric and they are: RSA, the Merkle-Hellman cryptosystem and the McEliece one. RSA is the most used cryptosystem and one of the simplest examples of using a NP-complete problem to ensure the security of the system. The Merkle-Hellman cryptosystem is based on the subset sum problem. Even though it has already been broken, it is the first system design to use the subset sum problem and there for is a good benchmark for any future cryptosystem based on this same problem. The McEliece system find its bases on code theory and uses the decoding of a general linear code problem for its security. This system has not yet been broken and its one of the possible candidates to use it as a post-quantum secure cryptosystem. This system uses the Goppa code matrixes and their capability to correct errors to decode messages. We will explain the basic principles of the code theory and will explain the Goppa code matrixes. Also we will use the Goppa code matrixes in our implementation of the McEliece system when we test the security of this system.

Finally, this project will include an experimentation chapter in which we will discuss different heuristic algorithms and show some of the results of testing those algorithms. The focus of the experiments will be on genetic algorithms and memetic ones as an extension of them. We will explain the main components of a genetic algorithm and the general way they works. We will explain a possible way of implementing genetic algorithm to solve the NP-complete

problems already explains and we will detail the algorithms that we will evaluate. Finally, we will compare the performance of this algorithms with direct exhaustive search and present the results. The results will show how the exponential scaling of the complexity of the problems makes this algorithms unable to solve cases of the problem when we increase the size of the problem to slightly realistic cases.

All the code use for the results shown in this project have been done with Python and using the servers provide for free by Google through its Google Colab service. The code is public and can be access through a link at the end of the last chapter.

Introducción

En 1928, David Hilbert propuso tres preguntas abiertas sobre los fundamentos de las matemáticas, la tercera de las cuales preguntaba si se podía crear un algoritmo capaz de decidir la veracidad de cualquier afirmación lógica. En 1936, Alonzo Church y Alan Turing demostraron cada uno independientemente que no puede existir tal algoritmo. En el proceso, Alan Turing inventó la conocida como máquina de Turing, un modelo matemático para abstraer el concepto fundamental de computadora, convirtiendole así, según algunos, en el padre de la informática. La máquina de Turing por su simplicidad conceptual y capacidad para simular otras máquinas de cálculo se ha convertido en uno de los pilares fundamentales de la teoría de la complejidad.

Los objetivos principales que se intentarán abordar en este proyecto son los siguientes. Estudiar los modelos matemáticos de abstracción de la computación, centrandose en la máquina de Turing como principal herramienta para la definición y el estudio de la complejidad de los problemas. Desarrollar las clases de complejidad de los problemas, principalmente la relación entre las clases P y NP. Detallar el problema de P contra NP, mostrando su importancia actual y sus implicaciones. Conocer algunos problemas NP-completos que se han utilizado en sistemas criptográficos, incluyendo el estudio del funcionamiento de dichos sistemas. Finalmente, estudiar los algoritmos genéticos, sus principales componentes y su implementación. Además de utilizar estos algoritmos genéticos para atacar los sistemas criptográficos basados en problemas NP-completos.

Los principales problemas NP-completos que se exponen en el trabajo han sido el problema de la suma de subconjuntos y el problema de decodificación de un código lineal. Se han presentado las demostraciones de la NP-completitud de los mismos siguiendo las demostraciones de autores anteriores. Se asocian estos dos problemas con los criptosistemas de Merkle-Hellman y McEliece respectivamente, referenciando los trabajos originales donde se presentaron estos. Incluyendo además referencias a los ataques de baja densidad que ya han conseguido romper el criptosistema de Merkle-Hellman. Siendo la versión del criptosistema de McEliece que se presenta, basada en matrices Goppa, todavía vigente y sin haberse conseguido vencer su seguridad.

Para este trabajo se ha creado una implementación original de algoritmos genéticos y meméticos para atacar los criptosistemas de Merkle-Hellman y McEliece. Los resultados obtenidos por estos algoritmos se muestran claramente inferiores a los resultados que ya se habían presentan por otros autores al atacar estos sistemas. Siendo los métodos que mejores resultados habían mostrado, órdenes de magnitud más eficientes y obteniendo peores resultados que otros estudios también basados en algoritmos genéticos.

1 Complejidad algorítmica

En este capítulo estudiaremos el concepto de algoritmo desde el punto de vista de las máquinas de Turing para poder evaluar su eficiencia. Definiremos lo que es una máquina de Turing y cuándo un problema es decidible o no. Usaremos las máquinas de Turing para definir la eficiencia algorítmica en base al número de pasos que utilice su máquina de Turing. Estudiaremos las distintas clases de complejidad de los algoritmos y nos centraremos en la diferenciación entre las clases P y NP. Por último estudiaremos la clase de problemas NP-completos y cómo se pueden reducir unos problemas a otros para acotar su complejidad.

1.1. Máquinas de Turing

En esta sección nos centraremos en el concepto de máquina de Turing. La máquina de Turing es una idea creada por Alan Turing en 1936 cómo una herramienta matemática para demostrar la falsedad de la conjetura de la decibilidad de las matemáticas de Hilbert. La máquina de Turing nos permite crear un modelo matemático, en la denominada Tesis de Church-Turing, en que consideramos funciones calculables aquellas que se pueden calcular mediante una máquina de Turing. Así cualquier otro dispositivo de cálculo que pueda simular una máquina de Turing podrá calcular cualquier función calculable. A pesar de que crear una máquina de Turing física es imposible por requerir de una cinta infinita y además poseer una terrible ineficiencia de cálculo en comparación con los ordenadores actuales es una gran herramienta para el estudio teórico de problemas de computación, pues nos permite evaluarlos todos desde la misma base.

Definición 1.1. Alfabeto: Un alfabeto A es un conjunto finito de elementos que denominaremos símbolos o letras.
Ejemplos: A={0,1} es el alfabeto binario y B = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,ñ,o,p,q,r,s,t,u,v,w,x,y,z} es el alfabeto castellano.

Definición 1.2. Palabra: Una palabra sobre un alfabeto A es una sucesión finita de elementos de dicho alfabeto.
Ejemplo: 01110 es una palabra en el alfabeto A={0,1}.

Una máquina de Turing es una máquina compuesta por tres partes: una cinta infinita en que cada posición contiene un simbolo, un cabezal que puede leer un simbolo de una posición y reemplazarlo por otro y una lista de instrucciones que debe seguir la máquina, que le indican a la máquina dado un determinado símbolo y el estado de la máquina como debe proceder. Cualquier máquina de Turing queda univocamente representada de la siguiente forma:

Definición 1.3. Máquina de Turing: Una máquina de Turing se define como una septupla MT = (Q, A, B, qo, #, F, δ) donde se tiene que:

- Q es el conjunto finito de estados en que puede estar la máquina.
- A es el alfabeto de entrada, que es el conjunto finito de símbolos que acepta la máquina.

- B es el alfabeto de la cinta, que es el conjunto finito de símbolos que pueden estar en la cinta tal que $A \subseteq B$.
- qo $\in$ Q es el estado inicial de la máquina.
- # $\in$ B es el símbolo en blanco de la máquina, el único que puede repetirse infinitas veces en la cinta.
- F es el conjunto de los estados terminales o finales de aceptación la máquina.
- $\delta : Q \times B \longrightarrow Q \times B \times \{L, R\}$ es la función de transición, que dado una posición y el estado actual de la máquina devuelve el simbolo a escribir en dicha posición, el nuevo estado al que pasará la máquina y L (left) para mover el cabeza a la izquierda o R (right) para mover el cabezal a la derecha.

El funcionamiento de una máquina de Turing será el siguiente: el cabezal se encontrará sobre una posición de la cinta con simbolo b y la máquina estará en un estado q, si dicho estado no es terminal sea δ(q,b)=(q',b',M) se escribirá en dicha posición el simbolo q', se cambiará el estado de la máquina a q' y el cabezal se moverá siguiendo la orden de movimiento M, a la derecha si M=R o a la izquierda si M=L. A realizar una vez este proceso se le denominará un paso de cálculo de la máquina de Turing.

Una máquina de Turing recibirá como entrada una palabra que pertenezca a su alfabeto de entrada. Comenzará siempre con el cabezal en la primera posición de dicha entrada y en su estado qo. A partir de ese punto seguirá las reglas marcadas por su función de transición cambiando símbolos y moviendo su cabezal hasta alcanzar un estado terminal. Una máquina de Turing se parará si alcanza un estado terminal o si en algún paso está en un estado y encuentra un símbolo que no tengan imagen por la función de transición. Si esto no llega a ocurrir la máquina no parará en ningún momento y continuará funcionando indefinidamente. Una máquina de Turing se considera que ha aceptado una entrada si en algún momento alcanza un estado de aceptación.

Definición 1.4. Lenguaje: Un lenguaje es un subconjunto del conjunto de todas palabras pertenecientes a un alfabeto. Un lenguaje se considera aceptado por una máquina de Turing si el alfabeto del lenguaje está contenido en el alfabeto de entrada de la máquina de Turing y la máquina de Turing acepta todas las palabras de dicho lenguaje y ninguna otra.

Definición 1.5. Problema de computación: Un problema de computación queda determinado por una tripleta P=(L, Q, δ), donde L es un lenguaje que contiene las palabras que se obtienen de codificar todas las instancias del problema, Q es el conjunto de respuestas posibles o estados de aceptación y δ es una aplicación de L a Q que le asigna a cada instancia del problema una única solución. Una máquina de Turing se dice que resuelve un problema si para toda palabra m del lenguaje L devuelve δ(m).

Definición 1.6. Problema de decisión: Un problema de decisión es aquel en que toda entrada admite únicamente dos posibles resultados "Verdadero" o "Falso". Podemos representar un problema de decisión como una función $P : L \longrightarrow \{0, 1\}$, donde L contiene todos los posibles casos del problema y a cada palabra de dicho lenguaje le asocia un valor binario. Se considerará el lenguaje asociado al problema a aquellas palabras a las que se les asocie el valor 1 y una máquina de Turing resolverá el problema si acepta este lenguaje.

Nos centraremos en los problemas de decisión, pues todo problema de computación se puede convertir en problema de decisión, tal que las soluciones del problema de decisión se puedan convertir en soluciones del problema original en un tiempo lineal. Sin embargo por la generalidad con que definimos un problema la mayoría de ellos no se podrán resolver. Esto se debe a que la cantidad de problemas que existen es no numerable, pero las máquinas de Turing que podemos llegar a crear son numerables. Por ello nos centraremos en los problemas que se pueden llegar a resolver, los decidibles.

Definición 1.7. Problema decidible: Un problema es decidible si existe una máquina de Turing que acepta el lenguaje y se detiene para todas las palabras construidas con su alfabeto de entrada. Aquellos problemas para los cuales no existe ninguna máquina de Turing que se detenga en todas las entradas y que acepte el lenguaje asociado se denominan indecidibles.

Que un problema sea indecidible implica que nunca podremos diseñar un algoritmo que sea capaz de resolver cualquier caso, ni siquiera en tiempos arbitrariamente grandes, siempre existirán casos del problema en que no se obtendrá nunca la solución. Imaginar un problema que carezca de algoritmo puede ser difícil, para ello discutiremos el ejemplo de problema indecidible más conocido, el problema de la parada.

Definición 1.8. Problema de la parada: Se conoce como problema de la parada al problema de dada una máquina de Turing y una palabra saber si la máquina de Turing se parará en algún momento.

Teorema 1.1. ***El problema de la parada es un problema indecidible***
Demostración: *Supongamos existe una máquina de Turing M(Q, A, B, qo, #, F, δ) que tenga dos estados finales Verdadero y Falso y que al recibir como entrada cualquier máquina de Turing y cualquier palabra alcance el estado Verdadero si la máquina se parará con dicha palabra y alcance el estado Falso si la máquina no se parará. Ahora crearemos una máquina de la cual no pueda determinar si termina para crear una contradicción. Para esto crearemos una máquina que dada como entrada una máquina de Turing se detenga sólo si dicha máquina no se detendría al tener como entrada sí misma. Crearemos esta nueva máquina a partir de M, llamemos M1 a esta nueva máquina de Turing que primero dada cualquier entrada la duplicará y después continuará actuando igual que M, de tal forma que actuará como si a M le preguntamos si una máquina se detendrá al recibirse a sí misma como entrada, pero además cambiamos los estados finales de M. El estado Falso será ahora el único estado final de M1, el estado Verdadero dejará de ser un estado final y se le añadirá a la función de transición δ de M que para todo símbolo del alfabeto de la cinta y el estado Verdadero devuelva el estado Verdadero y mueva el cabezal a la derecha. De esta manera si M devuelve Falso M1 se detiene, pero si M devuelve Verdadero M1 comenzará a mover el cabezal hacia la derecha infinitamente. Ahora si a la máquina M1 le damos como entrada la máquina M1 tendremos que M devolverá Verdadero si M1 se detine al recibir M1 como entrada y Falso si M1 no se detiene al recibir M1 como entrada. Si evaluamos los dos casos por separados tenemos:*
Primero: supongamos que M devuelve Verdadero, en tal caso M1 comenzará a moverse indefinidamente a la derecha y, por tanto, M1 nunca se parará, de tal manera que M debería haber devuelto Falso.
Segundo: supongamos que M devuelve Falso, en tal caso M1 se detendrá y, por tanto, M debería haber devuelto Verdadero.
De esta manera ninguno de los dos casos pueden ocurrir y alcanzamos una contradición que nos demuestra que tal máquina M no puede existir.

De ser este un problema decidible facilitaría enormemente el estudio de los problemas indecidibles, pues nos permitiría saber a priori qué máquinas y con qué entradas no se van

a detener y así no tener que ejecutarlas. Al ser este un problema indecidible eso implica que de una máquina genérica sólo podremos asegurar que se detiene al haber computado su funcionamiento hasta su detención y que sólo podremos asegurar que no se para hasta la cantidad de pasos que hayamos computado. Para poder afirmar que una máquina no se detiene necesitaremos un estudio específico de cada caso.

Vamos ahora a presentar distintas modificaciones o extensiones de la máquina de Turing original. La máquina de Turing tal y cómo la hemos definido hasta ahora sigue la idea original de Alan Turing, pero para el estudio detenido de algunos problemas nos interesará usar estas extensiones que nos simplificarán los cálculos. Existen distintas equivalencias funcionales entre todas estas extensiones, es decir, para los lenguajes que se pueden aceptar con un modelo se pueden crear versiones de los otros modelos que también los aceptan. Sin embargo, en nuestro estudio posterior de la eficiencia sí existirán diferencias fundamentales entre ellas.

Definición 1.9. Máquina de Turing no determinista: Una máquina de Turing no determinista es una variación de la máquina de Turing ya definida, que a partir de ahora se denominará máquina de Turing determinista, en que se cambia la función de transición $\delta(q, b)$. En este caso para cada pareja $(q, b) \in Q\text{x}B$ tendremos que la función de transición puede tener una lista finita de posibles transiciones que ejecutar. A la hora de ejecutar pasos de cálculo se escogerá cualquiera de las posibles transiciones de δ y se procederá de igual manera que la máquina de Turing determinista. Se denomina que una máquina de Turing no determinista acepta una palabra cuando existe alguna sucesión de elecciones de transiciones tal que la máquina llega a un estado de aceptación al comenzar con dicha palabra.

Definición 1.10. Máquina de Turing con varias pistas: Una máquina de Turing con n pistas tendrá en su cinta infinita n pistas en las cuales podrán haber diferentes símbolos del alfabeto de la máquina y la función de transición ahora recibirá una n-tupla de símbolos del alfabeto de la cinta y devolverá también una n-tupla de símbolos. Tal que en cada paso de cálculo la máquina podrá reescribir los símbolos de todas las pistas de una misma posición.

Definición 1.11. Máquina de Turing con varias cintas: Una máquina de Turing con n cintas tendrá n cintas independientes cada una con su propio cabezal que se moverá independientemente. En este caso la función de transición recibirá una n-tupla de símbolos del alfabeto cada uno de ellos correspondiente al símbolo sobre el que se encuentre cada uno de los cabezales y devolverá una n-tupla de símbolos cada uno de ellos correspodiendo al símbolo que debe reescribir cada cabezal, además de devolver una n-tupla del conjunto $\{L, R\}$ correspondiendo al movimiento de cada cabezal a izquierda o derecha. Tal que en cada paso de cálculo la máquina podrá reescribir los símbolos de todas las cintas de la posición de cada cabezal.

Por conveniencia siempre que se omita se supondrá que hablamos de máquinas deterministas. Mas adelante veremos cómo una máquina determinista puede simular una no determinista. Por otro lado, es bastante sencillo el cómo una máquina de una sóla cinta puede simular una de varias. Para ello sólo necesitamos pensar en el caso de simular una máquina de dos cintas y por indución finita podremos simular una máquina de n cintas. Si tenemos una máquina M(Q, A, B, qo, #, F, δ) de dos cintas, podemos crear la máquina M' que tenga como alfabeto BxB y que la función de transición en lugar de leer los símbolos de las dos cintas coja el simbolo de la cinta y evalue el producto cartesiano por separado. De esta forma nuestra máquina tendrá en cada casilla un simbolo de BxB que representará una

posición de la máquina M y funcionará de exactamente la misma manera.

En base a estas modificaciones surge de forma natural la cuestión de que otras posibles máquinas podemos diseñar que tengan la misma capacidad de cálculo que una máquina de Turing y así tiene sentido definir la idea de Turing completo

Definición 1.12. Turing completo: Se denomina Turing completo a aquel sistema que tenga un poder computacional equivalente a una máquina de Turing, es decir, que pueda resolver cualquier problema computacional que pueda resolver una máquina de Turing. Otra forma de interpretarlo es que el sistema y la máquina de Turing se puedan simular entre sí.

Podemos tomar como ejemplo los ordenadores que utilizamos, ¿son Turing completos? No es dificil comprobar que cogiendo un lenguaje de programación como C++ o Python se puede programar un algoritmo que simule una máquina de Turing y, por tanto, son Turing completos. De esta forma se comienza a entender la importancia del estudio de las máquinas de Turing, pues cualquier problema que demostremos que una máquina de Turing puede resolver lo podremos resolver con un ordenador, pues recordemos que no se puede crear una máquina de Turing física ya que siempre se presupone con una cinta infinita. Sin embargo, su verdadera importancia queda patente en la Tesis de Church-Turing.

Tesis de Church-Turing: *Toda función calculable se puede calcular mediante una máquina de Turing. Equivalentemente, todo procedimiento de cálculo que se pueda realizar mediante un proceso mecánico bien definido se puede realizar mediante una máquina de Turing.*

Lo que significa esto es que cualquier algoritmo que diseñemos una máquina de Turing lo podrá realizar también y que con sólo estudiar las máquinas de Turing podemos estudiar los límites de todos los sistemas de cálculo posibles. Además esto nos da una forma sencilla para conocer la potencia de cálculo de cualquier sistema, si es capaz de simular todas las máquina de Turing podrá realizar cualquier cálculo posible. Fijémonos en cómo hemos perdido formalismo en estos últimos apartados. Esto se debe principalmente a que no podemos definir de forma cerrada que es un sistema lógico o sistema de cálculo. No podemos conocer cómo funcionarán los ordenadores en el futuro y, de igual manera, probablemente Alan Turing tampoco imaginaba como serían nuestros sistemas actuales. Por esto terminamos en una especie de definición cíclica en que se considerará a algo como sistema de cálculo apropiado si es Turing completo. También es importante remarcar que a pesar de ser esta tesis casi universalmente aceptada como cierta se trata de una proposición lógica formalmente indemostrable. No podemos demostrar que no existen sistemas lógicos que una máquina de Turing no pueda replicar, sobre todo si ni siquiera podemos definir un sistema de cálculo. Por otro lado, si se encontrase algún algoritmo que una máquina de Turing no pudiese computar la tesis se refutaría. En cualquier caso la mayoría de la comunidad científica acepta la tesis como válida y toda la teoría de la computación se basa en que con sólo estudiar las máquinas de Turing cubrimos también cualquier otro dispositivo que se pueda crear en cualquier momento del futuro.

1.2. Complejidad algoritmica

En esta sección estudiaremos la dificultad de un problema midiendola según el tiempo o la memoria que requiera una máquina de Turing para resolver dicho problema.
En esta sección siempre estudiaremos la complejidad de un problema en base al número de caracteres que se requiera para escribir la entrada. A la hora de trabajar con problemas numéricos esto nos hace plantearnos si la forma de representar los números afecta a la eficiencia de nuestros algoritmos. Tenemos que dado un número x si llamamos n = |x| al número de caracteres necesarios para representar dicho número tenemos que es del orden n = log(x), con la diferencia de que si dicho número está en binario la base será 2 y si dicho número está en decimal será 10. Desde el punto de vista teórico esta diferencia no nos importará, pues más adelante veremos que sólo nos interesarán las diferencias entre distintos órdenes de crecimiento, tal que al ser siempre de orden logaritmico trabajaremos en la base que más nos interese para cada problema. Sin embargo, esto en la práctica indica que si creamos una máquina de Turing con una base mayor esta tardará menos pasos en resolver el problema. Por esto es importante que siempre midamos la eficiencia en base al número de caracteres de la entrada, pues si una máquina trabaja en binario para un mismo número tardará más pasos que una que trabaje en decimal, pero a la vez la entrada que habrá recibido será de mayor longitud, pues será un número escrito en binario que siempre tendrá más caracteres que en decimal, compensando la diferencia en pasos de cálculo y permitiendonos comparar las eficiencias de ambas máquinas.

Definición 1.13. Complejidad de una máquina de Turing determinista: Una máquina de Turing se dice que es de complejidad $f(n)$ si para toda entrada x tal que |x|=n la máquina se para en menos unidades que $f(n)$. Estas unidades podrán ser pasos de cálculo y entonces diremos que estamos midiendo la complejidad en el tiempo o podrán ser el número de casillas en la cinta que la máquina ha usado y entonces diremos que estamos midiendo la complejidad en el espacio.

Definición 1.14. Complejidad de un lenguaje: La complejidad de un lenguaje se dirá $f(n)$ si existe alguna máquina de Turing tal que acepta el lenguaje y es de complejidad $f(n)$. De igual manera, se dirá que la complejidad de un problema es igual a la complejidad del lenguaje que contiene todos sus casos positivos.

Definición 1.15. Ordenes de crecimiento: Se dirá que una función $f(n)$ es de orden $O(g(n))$ si existe un valor c y un valor N tal que para todo $n > N$ se tiene que $f(n) \leq cg(n)$, es decir, que f crece a una velocidad que podemos acotar por g.

Podemos por ejemplo pensar en la complejidad del algoritmo de la suma. Si una máquina recibe como entrada dos números y devuelve la suma de ambos deberá como mínimo haber pasado el cabezal una vez por cada dígito de los dos números que debe sumar y además deberá escribir el resultado de la suma. Sin embargo, en un caso óptimo esto último lo podría hacer a la vez que lee los números. Por tanto, en el mejor de los casos un algoritmo de suma podría ser de orden lineal con respecto a la longitud de la entrada, teniendo una complejidad $f(n) = an + b$ para algunos valores apropiados de a y b. Si pensamos en la multiplicación podemos pensar en un algoritmo que calculase el producto de cada dígito de un factor por el otro factor y después los sumase. Teniendo que recorrer cada factor al menos una vez por cada dígito del otro, teniendo una eficiencia del orden de n^2.

Veamos ahora un resultado importante que será útil más tarde. Supongamos que tenemos un algoritmo que tiene una eficiencia de tiempo polinómico y otro algoritmo que utiliza el primero. Si asumir que ejecutar el primer algoritmo tarda únicamente una unidad de tiempo nos permite calcular la complejidad del segundo como de tiempo polinómico, entonces este segundo algoritmo también tiene una complejidad polinómica. Esto es equivalente a que la composición de dos polinomios es otro polinomio, pues en el peor de los casos tendríamos que ejecutar el primer algoritmo en cada paso de ejecución y, por tanto, tardaríamos como máximo el producto de la eficiencia del primer algoritmo por el segundo. Esto nos muestra que ejecutar varios algoritmos de complejidad polinomial en serie también tiene una complejidad de orden polinomial. También nos muestra que si un algoritmo necesita realizar operaciones elementales como sumas y productos una cantidad polinomial de veces ese algoritmo también será de orden polinomial. De esta manera, cuando estudiemos si la complejidad temporal de un algoritmo es de orden polinómica podremos presuponer que toda operación elemental sólo tarda un paso de cálculo.

Definición 1.16. Complejidad no determinista: Una máquina de Turing no determinista se dice que es de complejidad $f(n)$ si para toda entrada x tal que |x|=n todas las posibles opciones de cálculo terminan en menos unidades que $f(n)$.

Definición 1.17. Clases de complejidad: Se denominará a la clase de complejidad de f(n) a todos los lenguajes para los cuales exista alguna máquina de Turing que los acepte en menos de f(n) unidades. Distinguiendo casos tenemos que usaremos la siguiente notación:

- TIEMPO(f): lenguajes aceptados por máquinas deterministas en tiempo $O(f(n))$
- ESPACIO(f): lenguajes aceptados por máquinas deterministas en espacio $O(f(n))$
- NTIEMPO(f): lenguajes aceptados por máquinas no deterministas en tiempo $O(f(n))$
- NESPACIO(f): lenguajes aceptados por máquinas no deterministas en tiempo $O(f(n))$

Nos interesa saber las relaciones entre estas clases. Primero, como toda máquina de turing determinista puede considerarse como una máquina no determinista que tiene en su función de transición siempre una sóla posible transición, es obvio que las clases de TIEMPO(f) y ESPACIO(f) están contenidas en las clases de NTIEMPO(f) y NESPACIO(f) respectivamente. Por otro lado, en general nos interesa estudiar principalmente la complejidad en el tiempo, pues para la mayoría de problemas a que nos enfrentamos su complejidad espacial es asumible y el gran limitante para su resolución en la práctica es el tiempo de ejecución del algoritmo. Sin embargo, veremos ahora relaciones entre la complejidad espacial y temporal que nos muestra que estas no son totalmente independientes.

Teorema 1.2. ***NTIEMPO***$(f(n)) \subseteq$ ***ESPACIO***$(f(n))$ ***y además NTIEMPO***$(f(n)) \subseteq$ ***TIEMPO(***$k^{f(n)}$***) para algún k.***

Demostración: *Para demostrar esto necesitaremos ver cómo se puede simular una máquina no determinista por una determinista. Para simplificar la explicación usaremos una máquina de 3 cintas para simularla. La idea consitirá en simular para cada paso de la máquina no determinista todas las posibles elecciones. Digamos que queremos simular hasta el paso L. En la primera cinta será la de entrada y no haremos nada, en la segunda copiaremos la entrada y simularemos la máquina no determinista funcionando de la misma forma que esta, cuando en una transición halla más de una opción pondremos en la tercera cinta todas las posibles elecciones que se pueden hacer y así nos aseguraremos de probar todas las opciones, si el máximo número de opciones de la máquina no determinista es k, entonces*

pondremos palabras de hasta k de longitud por cada paso de ejcución que hagamos. La máquina primero simulará todas las transiciones posibles en el primer paso de cálculo, después todas las del segundo paso de cálculo y así hasta el paso L. Si en alguna de las secuencias que simula acepta, la máquina acepta y se detiene, sino la máquina seguirá simulando todas las posibles secuencias del siguiente paso.

Veamos ahora la demostración, dada una máquina M no determinista que acepte un lenguaje en tiempo $O(f(n))$, podemos crear la máquina determinista M' que simula dicha máquina siguiendo la anterior descripción. En lo que se refiere a espacio la máquina M' usará en su segunda cinta en cada simulación cómo máximo $f(n)$ celdas de la cinta, pues cómo máximo ha tenido $f(n)$ pasos de cálculo que han podido ser todos a distintas celdas. En la tercera cinta cómo máximo puede tener $f(n)$ sucesiones de longitud k, siendo k el número máximo de opciones de transición de M, por tanto, el espacio será como máximo $k \times f(n)$ que es de orden $O(f(n))$. En lo que se refiere al tiempo para el primer paso tendrá que comprobar k posibles opciones, en el peor caso para el segundo tendrá que comprobar k^2 opciones y así para el paso $f(n)$ tendrá que simular $k^{f(n)}$ pasos y, por tanto, será de orden $O(k^{f(n)})$ en el tiempo

Para el siguiente teorema nos interesa antes estudiar la complejidad de un problema que usaremos en su demostración.

Proposición 1.1. ***El problema de la existencia de un camino en un grafo dirigido es de orden $O(v^3)$ donde v representa el número de vértices del grafo.***
Demostración:*Dado un grafo dirigido de v vértices y dos vértices v1 y v2 queremos saber si existe un camino que vaya de v1 a v2. Para resolver este problema usaremos una máquina de Turing de 3 cintas. Existen muchas formas distintas de codificar un grafo para darlo como entrada a una máquina de Turing, supongamos que como entrada en este caso recibimos los dos nodos a buscar y la matriz de adyacencia del grafo que tendrá longitud del orden $O(v^2)$, específicamente tendrá $v^2 + v - 1$ si la codificamos añadiendo separadores en cada fila. Nuestra máquina de Turing tendrá en su primera cinta la matriz de adyacencia, en su segunda entrada pondremos todos los nodos que nos faltan por analizar, al comienzo sólo el nodo del que partimos y en la tercera cinta pondremos todos los nodos que ya hemos analizado, comenzando vacía. El algoritmo que seguirá esta máquina será coger un nodo de la segunda cinta y buscar todas sus conexiones, si alguna es v2 la máquina termina y acepta, todo nodo con el que tenga conexión y que no esté ya en la tercera cinta se añade a la segunda cinta y después se borra el nodo actual de la segunda cinta y se pasa a la tercera. Si en algún momento la segunda cinta está vacía tendremos que no existía camino posible y la máquina termina. Para estudiar la complejidad temporal de la máquina podemos estudiar la complejidad de cada cinta. La segunda y tercera cinta tienen una longitud máxima de v y se deben recorrer cómo mucho una vez por vértice, pero la primera cinta que contiene la entrada es de longitud del orden $O(v^2)$ y se debe recorrer por cada vértice para buscar todas sus posibles conexiones. Por tanto, la complejidad temporal de la máquina será de $O(v^3)$ con respecto a su número de vértices.*

Teorema 1.3. ***NESPACIO$(f(n)) \subseteq$ TIEMPO$(k^{f(n)})$ para algún k.***
Demostración: *Dada una máquina de Turing no determinista de eficiencia $O(f(n))$, vamos a considerar el espacio de todas las posibles configuraciones en que puede estar dicha máquina. Para esto crearemos un grafo en que cada vértice representa una configuración posible de la máquina y los vértices sean transiciones posibles entre configuraciones. Una configuración la podemos ver como un tupla (q,a) donde q es el estado de la máquina y a es la palabra que está en la cinta cuando se le eliminan las infinitas colas de símbolos en blanco de casillas que no se han visitado. Sea Q el conjunto de estados posibles de la máquina y B su alfabeto de la cinta tenemos que a no puede tener longitud mayor de f(n) pues no se ocupan más celdas de la cinta. Por tanto, tenemos que el número máximo de*

configuraciones de la máquina de Turing y de vértices de nuestro grafo deberá ser $|Q||B|^{f(n)}$. Dada una palabra inicial, saber si esta será aceptada por nuestra máquina es lo mismo que buscar si existe un camino en el grafo de todas las configuraciones posibles desde la configuración inicial con esa palabra a un estado de aceptación. Como hemos visto antes que el buscar un camino en un grafo es de orden $O(v^3)$ en tiempo con v el número de vertices, en nuestro caso la máquina será de orden $(|Q||B|^{f(n)})^3$ en tiempo, que es lo mismo que $((|Q||B|)^3)^{f(n)} = K^{f(n)}$ para un K que no depende del tamaño de la entrada.

Veamos ahora algunas de las clases más importantes con las que vamos a tratar.

Definición 1.18. Clases de complejidad extensión:

- Clase P: es la unión de todas las clases TIEMPO$(n^j)\ \forall\, j > 0$.
- Clase NP: es la unión de todas las clases NTIEMPO$(n^j)\ \forall\, j > 0$.
- Clase PESPACIO: es la unión de todas las clases ESPACIO$(n^j)\ \forall\, j > 0$.
- Clase NESPACIO: es la unión de todas las clases NESPACIO$(n^j)\ \forall\, j > 0$.
- Clase EXP: es la unión de todas las clases TIEMPO$(2^{n^j})\ \forall\, j > 0$.
- Clase L: es la unión de todas las clases ESPACIO$(log(n))$.
- Clase NL: es la unión de todas las clases NESPACIO$(log(n))$.

Es interesante remarcar la importancia que se le da a la clase polinomial. Sabemos que una función de crecimiento exponencial terminará siempre siendo mayor que una que crezca de forma polinómica. En la resolución de problemas, independientemente de cómo crezca la complejidad con respecto al tamaño de la entrada, si poseemos un algoritmo que pueda resolver el problema podremos escoger una instancia del problema suficientemente pequeña tal que poseamos suficiente espacio y tiempo para resolverlo. Sin embargo, en la práctica en la mayoría de problemas de computación siempre nos centramos en la escalabilidad del problema, es decir, de que podamos enfrentarnos a entradas del problema muy grandes y seguir teniendo la capacidad de resolver el problema. Es aquí donde la clase exponencial genera problemas, pues los valores pueden explotar rápidamente conforme escala la entrada. Es por esto que aquellos problemas que tengan una complejidad exponencial serán considerados como intratables y, por contra, los problemas de complejidad polinomial serán considerados tratables, pues su crecimiento no es tan rápido al aumentar la entrada. Esta terminología puede sonar vaga, pues un problema particular polinomial puede ser del orden de un polinomio de muy alto grado y, por tanto, requerir de muchísimo esfuerzo para resolverlo, pero con esta terminología para una entrada suficientemente grande un problema tratable siempre se podrá resolver con menos recursos que uno intratable.
Utilizando los teoremas anteriores sobre inclusiones de clases podemos deducir la siguiente cadena de implicaciones

Teorema 1.4. ***Inclusiones de clases: Tenemos la siguiente cadena de inclusiones entre las clases de complejidad*** $L \subseteq NL \subseteq P \subseteq NP \subseteq PESPACIO$.
Demostración: *Como toda clase determinista está contenida en su clase no determinista tenemos que $L \subseteq NL$ y que $P \subseteq NP$. Para ver que $NL \subseteq P$ usamos que NESPACIO$(f(n)) \subseteq$ TIEMPO$(k^{f(n)})$,*

para cualquier máquina de Turing que acepte un lenguaje en espacio $O(log(n))$, tenemos que existirá una máquina determinista equivalente que resuelva el problema en tiempo $O(k^{log(n)})$, para algún k.

$$k^{log(n)} = exp\{log(k^{log(n)}) = exp\{log(k)log(n)\} = exp\{log(n^{log(k)})\} = n^{log(k)}$$

Por tanto, tendrá una complejidad de $O(n^{log(k)})$ tal que pertenece a P. Por último, para ver que $NP \subseteq PESPACIO$ usaremos que $NTIEMPO(f(n)) \subseteq ESPACIO(f(n))$ y tenemos claramente que se cumple.

Es importante remarcar que estas inclusiones no son necesariamente estrictas. Se conoce que $NL \subsetneq PESPACIO$ es una inclusión estricta, pero no se conoce todavía si el resto de clases son diferentes entre sí.

Hemos visto con estos resultados relaciones que podemos encontrar entre la complejidad temporal y la complejidad espacial y además entre las clases no deterministas y las deterministas, recordando siempre que toda clase determinista siempre está incluida en la no determinista. Hemos visto además que para complejidad temporal tenemos siempre limites superiores muy superiores a la espacial, pues dado que conocemos el orden de la complejidad temporal de un problema sabemos que como máximo dicho problema podrá tener una complejidad espacial de igual orden, mientras que dada la complejidad espacial de un problema sólo podemos acotar su complejidad temporal por una de orden exponencial con respecto a la espacial. Esto son sólo las cotas superiores que sabemos que deben cumplir siempre y para la mayoría de problemas se podrán encontrar soluciones más eficientes, pero ya esto nos muestra que la complejidad temporal de un problema siempre será igual o superior a su espacial y, por tanto bastante más determinante a la hora de resolver el problema. Llevando esto a nuestros ordenadores la complejidad espacial nos determina la cantidad de memoria que se deberá usar para resolver dicho problema, mientras que la complejidad temporal determina el tiempo necesario para resolver dicho problema. Más allá de nuestros resultados teóricos, la experiencia empírica nos enseña que la mayoría de algoritmos que intentan resolver problemas difíciles llegan a requerir tiempos que pueden superar los cientos de años cuando se hace crecer los valores de entrada, además añadir memoria a un ordenador se puede hacer de forma sencilla y no existe límite de cuanta puede tener, pero disminuir el tiempo en que tarda un ordenador en cada paso de cálculo es fundamentalmente más difícil.

Nos centraremos ahora principalmente en la complejidad temporal. Comencemos por uno de los resultados más importantes, que es una extensión de la tesis de Church-Turing, y que a su vez se ha convertido en bastante polémico en los últimos años.

Proposición 1.2. ***Tesis de Church-Turing Fuerte:*** *Todo procedimiento de cálculo, que se pueda realizar por un procedimiento mecánico, se puede realizar mediante una máquina de Turing con un incremento polinómico en el número de pasos necesarios. Es decir, dado un procedimiento mecánico que use $f(n)$ pasos la máquina de Turing necesitará $f(n)^k$ pasos para algún valor de k que dependerá del problema. [Yao(2002)]*

La implicación de este resultado es simple, querría decir que cualquier cálculo que algún dispositivo pueda hacer en tiempo polinomial también una máquina de Turing lo podría hacer en tiempo polinomial. Estudiaremos con más detenimiento los problemas en tiempo polinomial más adelante, pero en general se considera que estos problemas son aquellos que son fáciles de resolver. Dicho de otra manera, este resultado nos aseguraría que si

algún problema es fácil de resolver para algún sistema es fácil de resolver para todos. Sin embargo, al igual que la Tesis de Church-Turing este resultado no está probado y nunca se podrá probar cierto, por otro lado si existe un sistema que no se pueda replicar en tiempo polinomial por una máquina de Turing se demostraría falso. En la actualidad con el desarrollo de ordenadores cuánticos ya no existe un consenso como en la anterior de su veracidad, se considera por muchos que un ordenador cuántico funciona exponecialmente más rápido que una máquina de Turing y, por tanto, no sería cierta esta tesis. Por otro lado, habrá que demostrar que no existe algoritmo posible que simule un ordenador cuántico en tiempo polinomial,pues el sólo hecho de que todavía no se halla encontrado dicho algoritmo no es prueba suficiente.

1.3. NP- completos

En esta sección estudiaremos lo que es un problema NP- completo y NP-duro. Además definiremos lo que son las reducciones polinómicas de un problema a otro y las reducciones Turing y estudiaremos las clases de equivalencia que se pueden crear mediante la reducibilidad de los problemas.

Comenzaremos por dar unas definiciones alternativas para los problemas NP.

Definición 1.19. Relación binaria: Dado un alfabeto B, denotamos por B* a todas las palabras posibles de dicho alfabeto. Una relación binaria es un subconjunto $R \subseteq$ B* $\times$ B*. También se puede considerar R cómo una aplicación R : B* $\times$ B* $\longrightarrow \{0,1\}$, donde $R(x,y) = 1 \Longleftrightarrow (x,y) \in R$. Llamaremos a esta aplicación la función característica de R y por brevedad se escribirá xRy para representar que $(x,y) \in R$. Una relación binaria se denomina calculable polinómicamente si existe una máquina de Turing M que calcula la función característica de R en tiempo polinómico, es decir, que dado una pareja (x,y) calcula si xRy en tiempo polinómico.

Definición 1.20. Lenguajes NP por relaciones binarias: Un lenguaje $L \subseteq B^*$ es NP si y sólo si existe una relación binaria R y un polinomio p(n) tal que $L = \{x \in B^* : \exists y \in B^* \, con \, |y| \leq p(|x|)\,,\, xRy\}$ y R es calculable polinómicamente.

Esta definición es equivalente a la que teníamos mediante clases de complejidad, pero en esta se ve reflejada una idea intuitiva muy útil sobre los problemas NP. Un problema NP es aquel que se puede comprobar una solución fácilmente, o lo que es lo mismo, la relación binaria de dicho problema se puede calcular en tiempo polinomial. Al algoritmo que calcula la función característica se le denomina verificador y esto significa que dada una entrada x del problema y una posible solución y del problema, podemos calcular rápidamente si xRy y saber si hemos hallado la solución. Podemos entonces entender un problema NP como aquel en que dado un x hallar la solución específica y es difícil, pero comprobar si una solución específica es correcta es fácil. El siguiente ejemplo ejemplifica esta idea.

Proposición 1.3. ***Saber si un número es compuesto es un problema NP.***
Demostración: *Presupuesto que sabemos que una división se puede calcular en tiempo polinomial, lo cual se puede verificar rápidamente imaginando una máquina de Turing que replique el mismo algoritmo que cuando realizamos una división a mano. Creamos una máquina de Turing no determinista, que dado un número p como entrada nos diga si es compuesto. Para esto la máquina creará otro número aleatorio de igual número de dígitos y probará a dividir p por ese número. Como la máquina es no*

determinista puede tener en su función de transición todas las posibles cifras del sistema de numeración que estemos usando y recorrer la cinta escogiendo cualquier cifra de las posibles, devolviendonos un número aleatorio. Si la división es entera, entonces el número p no era primo. Si el número era compuesto al menos uno de los posibles números dará una división entera y la máquina aceptará.

La idea fundamental que nos muestra este ejemplo es la ventaja fundamental que poseen las máquinas no deterministas de que sólo necesitan que en una de las posibilidades la máquina acepte. Cualquier problema para el cual sepamos que su verificador es de orden polinomial será NP y además crear la máquina de Turing no determinista para dicho problema será muy sencillo. La máquina escogerá una solución del problema, con el único requisito de que tenga la posibilidad de escoger absolutamente todas las posibles soluciones del problema, y después aplicará el verificador para ver si la solución es correcta. Como la solución tiene una longitud de orden polinomial respecto a la entrada se puede escoger una solución en tiempo polinomial y como el verificador es también polinomial tenemos que la máquina tendrá una complejidad temporal de orden polinomial. De hecho, del ejemplo anterior se demostró en 2002 algo más fuerte y es que saber si un número es primo es un problema de clase P, pero para ello se utiliza el algoritmo AKS que es notablemente más difícil que la idea que hemos aplicado antes, mostrandonos de nuevo la ventaja de la clase no determinista a la hora de demostrar que un problema pertenece a esta.

Definición 1.21. Reducción de un lenguaje: Un lenguaje L1 se dice reducible a un lenguaje L2, denotado $L1 \propto L2$. Si existe una máquina de Turing que calcule en tiempo polinomial una función R del alfabeto de L1 al alfabeto de L2 y tal que $x \in L1 \iff R(x) \in L2$.

Definición 1.22. Reducción de un problema: Dados dos problemas P1 y P2 con L1 el lenguaje de P1 y Q1 el conjunto de soluciones de P1 y, respectivamente, L2 y Q2 de P2. Se dice que P1 es reducible a P2, denotado $P1 \propto P2$, si L1 es reducible a L2 por una función R y existe una máquina de Turing que calcule en tiempo polinomial una función S de Q2 a Q1 tal que si y es solución de R(x) en P2 se tenga que $S(y)$ sea solución de x en P1 $\forall x \in L1$.

Aplicado esto a un problema de decisión P1(x) en que consideramos el problema como una función que nos devuelve un booleano tendremos que P1(x) será reducible a otro problema de decisión P2(y) si existe una máquina de Turing que calcule en tiempo polinomial una función R del lenguaje de P1 al lenguaje de P2 tal que $P1(x) = P2(R(x))$.

Si tenemos un problema P1 reducible a un problema P2 y tenemos una máquina de Turing que resuelva el problema P2, entonces podremos resolver también P1. En otras palabras, el problema P1 será como mucho tan difícil como P2. Si además nuestra máquina de Turing lo resuelve en tiempo polinomial, tenemos que también P1 tendrá una complejidad temporal de orden polinomial. Por tanto, un problema reducible a otro de clase P tendrá una complejidad como máximo de P y un problema reducible a otro de clase NP tendrá una complejidad de cómo mucho NP. Siguiendo esta idea tenemos el siguiente tipo de problema.

Definición 1.23. Lenguaje NP-completo: Un lenguaje NP es NP-completo si cualquier otro lenguaje NP se reduce a él. De igual manera un problema será NP-completo si todo otro problema NP es reducible a él.

Los problemas NP-completos serán los problemas más difíciles que haya en la clase NP, pues cualquier otro problema NP se podrá reducir a él. Si encontramos un algoritmo para

resolver cualquier problema NP-completo podremos usar dicho algoritmo para resolver cualquier otro problema NP con una sobrecarga de orden polinomial en la complejidad. Esto nos crea una cota máxima de la complejidad que puede tener un problema NP, pues dado que consigamos encontrar un algoritmo que resuelva un problema NP-completo no podrá haber otro problema NP con una complejidad de orden superior a este problema. Esto es siempre presuponiendo que trabajamos con máquinas deterministas, pues las reducciones de problemas siempre se presuponen en tiempo polinomial para máquinas deterministas.

Definición 1.24. Problemas equivalentes: Dos problemas de computación P1 y P2 se denominan equivalentes si $P1 \propto P2$ y $P2 \propto P1$. La equivalencia de problemas es una relación de equivalencia y nos da una partición del conjunto de problemas de computación.

Por la definición que hemos dado de reducción de un problema es sencillo probar que es una relación de equivalencia. La única propiedad interesante es la transitividad, dados tres problemas P1, P2 y P3 tal que $P1 \propto P2$ y $P2 \propto P3$ tendremos que podemos crear una máquina de Turing que calcule la primera reducción y secuencialmente la segunda. Como el orden de cada reducción es polinomial tendremos que el orden de nuestra máquina de Turing será a lo más la suma de los órdenes de ambas reducciones que es otro polinomio. De esta forma podemos dividir nuestros problemas según esta relación de equivalencia, tal que dentro de los problemas NP tendremos la clase de los problemas NP-completos que son todos equivalentes entre sí. De igual manera podríamos definir los problemas P-completos dentro de la clase P y su clase de equivalencia.

Una duda razonable que nos podría surgir es si existen clases de equivalencia entre los problemas P-completos y los NP-completos, es decir, si podemos encontrar un problema NP que no pertenezca a P y tampoco sea NP-completo. La realidad es que hasta día de hoy esta pregunta está sin respuesta y, de hecho, ni siquiera se puede afirmar que las clases P y NP sean distintas entre sí, lo cual es el tema de discusión de la siguiente sección. Un ejemplo de esto podría ser el problema de isomorfismo de grafos, dado dos grafos finitos saber si son isomorfos, este problema no se ha demostrado todavía que sea NP-completo, ni tampoco se ha hallado una forma de resolverlo en tiempo polinomial. A este tipo de problemas se les puede denominar de clase NP-intermedia, pero de nuevo, se desconoce si esta es una clase propia o si en algún momento se demostrará que sea de clase P, NP-completo o ambas.

Definiremos ahora la reducibilidad Turing que es una forma menos restrictiva de reducibilidad que nos da otra de las formas en que se pueden clasificar los problemas según su complejidad.

Definición 1.25. Reducción Turing: Un problema P1 se reduce Turing a otro P2, denotado $P1 \propto_T P2$, si existe una función que resuelve P2, posiblemente una máquina de Turing, y existe una máquina de Turing que resuelve el problema P1 en tiempo polinómico usando llamadas a dicha función y contando dichas llamadas como un único paso de cálculo.

Definición 1.26. NP-duro: Un problema se considerará NP-duro si cualquier problema NP se puede reducir Turing a este.

La principal diferencia entre los NP-duros y los NP-completos es que los problemas NP-duros no tienen que pertenecer a la clase NP. Por la definición que hemos dado de reducibilidad Turing no importa lo alta que pueda llegar a ser la complejidad de un problema

NP-duro, pues al calcular su reducibilidad Turing ese cálculo se contará siempre como una sóla unidad de tiempo. Por esta razón, de igual manera que los NP-completos son los más difíciles de la clase NP, los NP-duros se pueden entender como aquellos problemas que son al menos tan difíciles como un problema NP.

Cuando sólo evaluamos el orden de complejidad de un problema será equivalente resolver el problema directamente o reducirlo a otro problema y resolver aquel. Sin embargo a la hora de afrontar un problema particular será más eficiente buscar una solución particular del problema para evitar la sobrecarga de tiempo de aplicar reducciones. Debemos tener en cuenta que si por ejemplo tenemos un problema de orden exponencial y aplicamos una reducción de orden polinomial no cambiamos el orden de complejidad de nuestro problema, pero aumentamos las constantes determinadas de nuestro problema aumentando su coste de resolución. Desde el punto de vista teórico encadenar finitas reducciones no cambia la complejidad de los problemas que trataremos, pero esto nos puede dar ordenes polinomiales con coeficientes demasiado grandes como para ser computados con nuestros recursos usuales.

1.4. Problema P vs NP

En esta sección nos centraremos en el estudio del problema del milenio P vs NP. Discutiremos sobre la conjetura de Cook y sus posibles implicaciones para la computación.

Parece importante remarcar que las clases P y NP se diferencian en que en la primera usamos máquinas deterministas y en la segunda no deterministas. Sin embargo, si pensamos en cualquiera de las máquinas reales que usamos, cualquier ordenador, calculadora u otro dispositivo que utilizemos para el cálculo es inherentemente determinista, esto es dado un algoritmo siempre lo realizará de exactamente la misma manera. Si dejamos por un momento los ordenadores quánticos fuera de esta discusión, pues todavía sólo se están desarrollando y no existe ningún modelo accesible para el público general, en la computación no existe en realidad la aleatoriedad. Es cierto que podemos pedirle a nuestro ordenador números aleatorios, pero nuestro ordenador no puede mágicamente elegir números al azar. El procedimiento general que se usa es crear números pseudoaleatorios a partir de una semilla, que a falta de mejor candidatos puede ser el tiempo en el instante que se ejecuta el código, pero siempre de forma perfectamente determinista. Por todo esto los ordenadores que utilizamos, si bien no son una copia exacta, tienen un funcionamiento mucho más cercano al de una máquina de Tuirng determinista.

Por esto nos interesa siempre más la complejidad de los problemas resueltos con una máquina de Turing determinista y no es tan relevante lo que nos permitan las no deterministas. Si escogemos un problema NP sabemos que en el peor de los casos se podrá resolver con un tiempo de orden exponecial, pero por otro lado como P está contenido en NP puede que se resuelva en un tiempo polinomial y, a priori, no tenemos herramientas para asegurar ni una ni otra. Por una cuestión de notación y de economía del lenguaje cuando hablamos de un problema NP presuponemos que NP es su menor clase de complejidad, pues aunque L y P están contenidas en NP daría una información más precisa del problema decir que es de clase P, por ejemplo, antes que NP, pues es una afirmación más restrictiva. Como la clase de complejidad se puede ver como una cota al tiempo tiene sentido preferir las cotas más

bajas siempre. Esto que en otras áreas de las matemáticas es un convenio muy usual aquí se vuelve una presuposición inherentemente errónea debido a la naturaleza de los problemas de computación. Esto es para casi todos los problemas de computación es muy sencillo crear algoritmos que resuelvan el problema a costa de ser muy ineficientes, generalmente mediante una busquedad exahustiva o fuerza bruta, pero encontrar el algoritmo óptimo y demostrar que no existe ninguno mejor es una tarea casi impracticable para la mayoría de problemas. Por tanto, para la mayoría de problemas conocemos una clase de complejidad a la que pertenecen, pero sin poder asegurar que no tengan una complejidad menor que esa. En el caso de los problemas NP esto es todavía más evidente, pues con probar que la solución de un problema se puede comprobar fácilmente ya tenemos que es NP, pero no por esto encontrar dicha solución debe ser fácil.

LLegados a este punto tiene sentido preguntarse si son las clases P y NP iguales o, como se suele escribir, si es P=NP.

Proposición 1.4. ***Conjetura de Cook:*** *La clase de complejidad P y la clase NP son distintas.*

El matemático Stephen Arthur Cook formuló esta conjetura en 1971, como lo que parecía ser un problema interesante, pero sin demasiada importancia. Si lo pensamos obviando los conocimientos actuales que poseemos del problema parece bastante claro que ambas clases son distintas, pues una máquina no determinista es mucho más potente que una determinista y por tanto parece natural pensar que podrán resolver más problemas. Por esto en su origen parecía un problema que aunque difícil de resolver no tendría mayor importancia.

Tras su no resolución en los consecuentes años, no tardaron mucho en percatarse de las posibles implicaciones que podría llegar a tener este problema, pues si resultase que P=NP significa que si es fácil comprobar la solución de un problema también lo debe de ser resolverlo. El problema fue ganando en importancia hasta que en el año 2000 se anuncio como uno de los problemas del milenio del Clay Mathematics Institute. Los problemas del milenio son una colección de 7 problemas premiados con un millón de dólares que se consideran los más relevantes problemas abiertos de las matemáticas actualmente. De estos sólo se ha resuelto hasta ahora la Conjetura de Poincaré y el problema de P vs NP sigue abierto a día de hoy.

Si nos centramos ahora en las posibles consecuencias de este problema, tenemos tres posibles casos. Primero puede ser que P no sea igual a NP, lo cual es lo más intuitivo y es lo que se presuponía al principio, esto querría decir que tenemos problemas muy difíciles de resolver, pero que poseen un verificador muy fácil. El segundo caso es que P sea igual a NP que implicaría un cambio de paradigma en la computación, pues implicaría que muchos problemas que considerabamos difíciles simplemente es que los estábamos resolviendo con algoritmos ineficientes y obligaría a toda una oleada de revisionismo de los algoritmos para estos problemas para crear algoritmos mejores. Por último el tercer caso es que encontremos un problema NP que se resuelve en tiempo polinomial y, por tanto sea P=NP, pero que su complejidad sea un polinomio de tan alto grado que igualmente sea inabarcable computacionalmente, esto significaría que para la mayoría de problemas seguiría siendo preferible los algoritmos de eficiencia exponencial antes que uno polinomial de orden extremadamente alto y reduciría el problema a sólo una curiosidad matemática sin relevancia en la computación actual.

Uno de los principales campos donde este problema podría tener un gran impacto y, además, que es más relevante para nuestro estudio es en la criptografía. En las comunicaciones

en Internet entre personas que nunca se han conocido ni han intercambiado nunca información los sistemas criptográficos que se usan para encriptar las comunicaciones deben ser públicos para que todos los usuarios puedan usarlos. Esto permite a cualquiera que quiera descifrar un mensaje saber exactamente cómo se ha codificado este. Para sobrepasar el hecho de que todo el mundo conozca que algoritmo se usa para la codificación en criptografía siempre se ha dependido de funciones o algoritmos que sean muy fáciles de realizar en un sentido, pero muy difíciles de deshacer. De esta manera dado un mensaje y una clave se pueda rápidamente crear un mensaje codificado con dicha clave, pero aún cuando posible optener el mensaje inicial y la clave sólo del mensaje encriptado sea una tarea muy difícil y demasiado costosa en tiempo para ser practicable. Aunque existen muchos sistemas criptográficos distintos, uno de los más usuales se centra en la descomposición factorial en números primos, que como sabemos es un problema NP. Si resultase que P=NP, este problema se podría resolver en un tiempo polinomial y el tiempo para romper este criptosistema se reduciría a un periodo razonable como para decodificar mensajes sistemáticamente. Como este existen muchos otros problemas NP que se usan para sistemas criptográficos, como por ejemplo el problema de la suma, tal que la falsedad de la conjetura de Cook podría poner en duda la seguridad de la mayoría de sistemas.

Por último, recordemos que al principio de esta sección decíamos que era mejor no centrarse en los ordenadores cuánticos para poder entender más fácilmente toda la importancia que nos presenta el problema de P vs NP. Discutamos ahora brevemente sobre este tipo de computadores. Hemos visto que la descomposición factorial de números primos es un problema NP y, sin embargo, existe un algoritmo para descomponer en números primos en tiempo polinomial por un ordenador cuántico, el algoritmo de Shor. ¿Implica esto que la factorización en números primos es un problema P? Pues en realidad no, primero de todo porque ese algoritmo no se puede replicar en una máquina de Turing, que es la que usamos para la definición de las clases de complejidad y, por otro lado, porque un ordenador cuático se parece muy notablemente más a una máquina no determinista. Antes decíamos que un ordenador usual no puede generar números aleatorios, pero un ordenador cuántico mediante el uso de las propiedades de la mecánica quantica si puede devolver valores aleatorios. Si bien un ordenador cuántico tampoco se puede considerar equivalente a una máquina de Turing no determinista, sino que se podría equiparar a una máquina de Turing quántica, modificación que no hemos explicado en este documento. En general, el estudio de la complejidad con ordenadores cuánticos se considera siempre un caso a parte que no influye en las conclusiones sobre la complejidad de los problemas clásicos que nos atañen. Además debemos recordar que el problema de factorización es NP, pero no NP-completo. Por tanto, todavía no se ha resuelto ningún problema de clase NP-completo en tiempo polinomial y tampoco se espera conseguirlo pronto. Esto se debe a que hasta ahora se conoce el algoritmo de Glover para la aceleración cuántica de búsqueda en bases de datos [Grover(1996)]. Sin embargo, este sólo proporciona una aceleración cuadrática respecto a la computación clásica que no exponencial. Tal que problemas NP-completos, como los que vamos a ver en la siguiente sección, todavía no han sido resueltos por ordenadores cuánticos.

2 Ejemplos de problemas NP-completos

En este capítulo estudiaremos algunos ejemplos de problemas NP-completos y sus correspondientes demostraciones de que son NP-completos. Comenzaremos por el problema SAT por ser este el más conocido y el que más fácilmente podremos usar para demostrar que otros problemas son NP-completos al reducir este a otros problemas. Después estudiaremos algunos problemas que se utilizan en la criptografía para la creación de sistemas criptográficos. En esta sección sólo estudiaremos los distintos problemas de decisión y en los siguientes capítulos profundizaremos en como se pueden aplicar dichos problemas a la criptografía.

2.1. Teorema de Cook

Esta sección se centrará en el Teorema de Cook[Cook(1971)], que nos dá el primer ejemplo de problema NP-completo. Primero definamos el problema computacional que vamos a tratar.

Definición 2.1. Problema SAT: El problema SAT o problema de consistencia en lógica proposicional es en su versión de problema de decisión dado un conjunto X de símbolos proposicionales y una colección C de cláusulas sobre estos símbolos, decidir si existe o no alguna asignación de valores de verdadero o falso a cada símbolo de X tal que todas las cláusulas de C se satisfagan simultáneamente.

Ejemplo 2.1. Sea $X = \{x1, x2\}$ y $C = \{x1 \vee \neg x2, \neg x1 \vee x2\}$. Tenemos que se pueden cumplir todas las cláusulas asignando verdadero tanto a $x1$ y a $x2$.

Como se puede inferir rápidamente, el problema SAT es un problema extremadamente general, pues trata cláusulas lógicas cualesquiera e implica que cualquier problema que podamos reescribir con notación lógica se puede ver como un problema SAT, de hecho usaremos este hecho para demostrar el teorema de Cook. Por otro lado, es fácil comprobar que es un problema NP, pues dado una posible solución sólo deberemos recorrer cada cláusula para comprobar si se satisface.
Es interesante remarcar que el tener dos cláusulas distintas c_1 y c_2 en C es equivalente a establecer una sóla cláusula que sea $c_1 \wedge c_2$, por tanto, cada cláusula en la que aparezca una conjunción lógica se puede dividir por esta en varias cláusulas equivalentes a la primera. Si unimos todas las cláusulas de C en una sóla podemos reescribir esta en la forma normal conjuntiva, esto es, escribirla como conjunción de subcláusulas en donde sólo aparece el operador lógico "o", la disyunción lógica. En el ejemplo anterior $(x1 \vee \neg x2) \wedge (\neg x1 \vee x2)$ está escrito en forma normal conjuntiva, pues son dos subcláusulas unidas por el operador "y" que en su interior sólo contienen el operador "o". Llamaremos la forma disyuntiva a estas cláusulas que sólo contienen el operador "o" y como cualquier proposición lógica se puede reescribir en forma normal conjuntiva en nuestros problemas SAT siempre podremos presuponer que el conjunto de cláusulas C sólo contiene cláusulas en forma disyuntiva.

Teorema 2.1. ***Teorema de Cook-Levin:*** *El problema SAT es un problema NP-completo.*

Para demostrar este teorema debemos demostrar que cualquier problema NP se puede reducir a un problema SAT en tiempo polinomial. Para esto simularemos una máquina no determinista mediante proposiciones lógicas y que el problema sea aceptado por una máquina será equivalente a que se acepte el problema SAT asociado. Supongamos tenemos un problema NP cualquiera y, por tanto, existe una máquina de Turing no determinista M que decide este problema en tiempo polinomial que denotaremos p(n) y que tendrá un estado de aceptación q_a y un estado de rechazo q_r. Denotemos $Q = \{q_0, ..., q_r\}$ los estados de la máquina, con $q_a = q_1$ y $A = \{a_0, ..., a_v\}$ el alfabeto de la máquina, siendo a_0 el símbolo en blanco. Como es una máquina no deteminista supondremos que en cada paso tiene como máximo I opciones distintas que elegir.

En nuestro problema SAT definiremos los siguientes símbolos proposicionales:

1. $Q[i,k], 0 \leq i \leq p(n), 0 \leq k \leq r$, representa que la máquina está en el estado q_k en el paso i.
2. $H[i,j], 0 \leq i \leq p(n), 0 \leq j \leq p(n)$, representa que la máquina está en la casilla j en el paso i.
3. $S[i,j,k], 0 \leq i \leq p(n), 0 \leq j \leq p(n), 0 \leq k \leq v$, representa que en la casilla j está el símbolo s_k en el paso i.
4. $O[i,k], 0 \leq i \leq p(n), 0 \leq k \leq I$, representa que la máquina escoje la opción k en el paso i.

Ahora definiremos las cláusulas de nuestro problema

1. $Q[i,0] \vee ... \vee Q[i,r], 0 \leq i \leq p(n)$, representa que la máquina debe estar en algún estado en cada paso.
2. $\neg Q[i,k] \vee \neg Q[i,k'], 0 \leq i \leq p(n), 0, \leq k < k' \leq r$, representa que la máquina no puede estar en dos estados en el mismo paso.
3. $H[i,0] \vee ... \vee H[i,p(n)], 0 \leq i \leq p(n)$, representa que el cabezal debe estar en alguna casilla en cada paso.
4. $\neg H[i,j] \vee \neg H[i,j'], 0 \leq i \leq p(n), 0, \leq j < j' \leq p(n)$, representa que el cabezal no puede estar en dos casillas en el mismo paso.
5. $S[i,j,0] \vee ... \vee H[i,j,v], 0 \leq i \leq p(n), 0 \leq j \leq p(n)$ representa que cada casilla debe tener algún símbolo en cada paso.
6. $\neg S[i,j,k] \vee \neg S[i,j,k'], 0 \leq i \leq p(n), 0, \leq j \leq p(n), 0 \leq k < k' \leq v$ representa que cada casilla no puede tener dos símbolos en el mismo paso.
7. $O[i,0] \vee ... \vee O[i,r], 0 \leq i \leq p(n) - 1$, representa que la máquina toma alguna decisión en cada paso.
8. $\neg O[i,j] \vee \neg O[i,j'], 0 \leq i \leq p(n), 0, \leq j < j' \leq I$, representa que la máquina no toma dos decisiones en cada paso.
9. Configuración inicial:
   a) $Q[0,0]$ comienza en el estado inicial.

b) $H[0,0]$ comienza en la primera casilla.

c) Si la entrada es $a_{k_1}...a_{k_n}$ añadimos: $S[0,0,k_1],...,S[0,n-1,k_n]$ representando la entrada de la máquina y $S[0,n,0],...,S[0,p(n),0]$ para dejar el resto de casillas en blanco.

10. Si $\delta(q_k,s_d)=\{(q_{k_1},s_{d_1},m_1),...,(q_{k_I},s_{d_I},m_I)\}$ añadimos:
$\neg H[i,j]\vee\neg Q[i,k]\vee\neg S[i,j,d]\vee\neg O[i,e]\vee(Q[i+1,k_e]\wedge S[i+1,j,d_e]\wedge H[i+1,j+m_e])$
$0\leq i\leq p(n)-1, 0\leq j\leq p(n), 0\leq k\leq r, 0\leq d\leq v, 1\leq e\leq I$
Representando el funcionamiento de la máquina en cada paso. Si estamos en un estado, casilla y símbolo determinado escojemos alguna opción de la función de transición Consideramos que para los estados finales la máquina tiene en su función de transición I opciones iguales que dejan el estado, la casilla y el cabezal igual.

11. $\neg H[i,j]\vee\neg S[i,j',d]\vee S[i+1,j',d], 0\leq i\leq p(n)-1, 0, \leq j<j'\leq p(n), 0\leq d\leq v$, representa que los símbolos de la cinta no pueden cambiar si no está el cabezal sobre ellos.

12. $Q[p(n),1]$ Restringimos a que la máquina en el último paso acepte.

Tal y como hemos definido nuestras cláusulas estas simulan todo el funcionamiento de la máquina no determinista. La veracidad de la mayoría de cláusulas está determinada desde un inicio, pero la libertad en elegir cada opción le permite seguir el funcionamiento no determinista de la máquina. Al haber añadido la condición de que llegue a un estado de aceptación, tenemos que si el problema SAT se puede satisfacer, entonces la máquina acepta. Por tanto, el problema genérico NP se ha podido reducir al problema SAT. Por último, veamos que se ha hecho en tiempo polinomial. Como la máquina resolvía en tiempo polinomial $p(n)$, tenemos que en toda la transformación hemos trabajado con índices entre o y $p(n)$ o con valores constantes. Los valores constantes no afectan a la complejidad y el producto de polinomios es polinómico, por tanto, el orden de complejidad de la redución es polinomial. Con esto queda demostrado el Teorema de Cook, teniendo ahora el primer ejemplo de problema NP-completo. Para demostrar que otros problemas también son NP-completos ahora podremos usar que SAT lo es y con lograr una reducción de SAT a ese problema habremos demostrado que también es NP-completo.

2.2. Problema de la suma de subconjuntos

En esta sección estudiaremos el problema de la suma de subconjuntos que es un caso particular del problema de la mochila. Además estudiaremos su complejidad y crearemos una reducción del problema SAT a este problema.

Definición 2.2. Problema de la suma de subconjuntos: El problema de la suma de subconjuntos es un problema de decisión en que dado un conjunto V de enteros positivos se debe determinar si existe alguna suma de elementos de V que den un valor W determinado.

Ejemplo 2.2. Sea $V=\{1,2,4,8,16\}$ y $W=13$ tenemos que sí existe un subconjunto de V que sumen W, siendo $13=1+4+8$.

Primero debemos ver que este problema es de clase NP, esto no presenta gran dificultad, pues dado una posible solución del problema sólo tendremos que sumar todos los valores de dicha solución para comprobar si valen W.

Teorema 2.2. *[Xanda Schoefield(2018)] El problema de la suma de subconjuntos es un problema NP-completo.*

Para demostrar este teorema vamos a suponer un caso genérico de problema SAT y crearemos una reducción de dicho problema a uno de suma de subconjuntos. Como ya hemos demostrado la NP completitud de SAT esto concluirá la demostración de que la suma de subconjuntos es NP-completo.
Supongamos un problema SAT en que tenemos n variables $x_1, ..., x_n$ y m cláusulas $c_1, ..., c_m$. Podemos presuponer además que las clausulas están escritas en forma disyuntiva, es decir, cada clausula está compuesta únicamente por disyunciones lógicas, por ejemplo: $x_1 \vee x_2$. Además denominaremos k_i al número de variables que aparece en la cláusula c_i.
Para crear nuestro problema de suma de subconjuntos trabajaremos en una base B suficientemente grande tal que nunca haya suma con llevada, que estableceremos al final. Todos los números de nuestro conjunto V se podrán escribir como $\sum_{i=0}^{n+m} a_i B^i$. Añadiremos un número v_i a nuestro problema por cada variable x_i y otro v_{n+i} por cada variable negada $\bar{x}_i$. Los primeros n dígitos servirán para asegurar que para cada variable siempre se escoje x_i o $\bar{x}_i$ como verdadero, para esto los números v_i y v_{n+i} ambos tendrán un 1 en el dígito i-ésimo y nuestro valor objetivo W tendrá valor 1 en ese dígito, tal que sólo una de las dos se pueda escojer cada vez. Para asegurar que se satisfacen todas las cláusulas pondremos en el dígito (n+i)-ésimo de W el valor k_i y pondremos un 1 en ese mismo dígito en todos los v_i que aparezcan en dicha cláusula. Al estar escritas estas de forma normal disyuntiva, sólo necesitaremos que al menos una de las variables que aparezcan en ellas sean verdaderas, por este motivo por cada cláusula c_i añadiremos $k_i - 1$ números que sólo tengan 1 en el dígito (n+i)-ésimo. Ahora si sumamos todos los números que hemos creado el mayor resultado posible es en los dígitos correspondientes a las cláusulas donde sumamos k_i números correspondientes a variables y otros $k_i - 1$ que hemos añadido, por esto podemos seleccionar $B = 2 \max_i(k_i)$ tal que nunca se producirá una suma con llevadas. Así nos terminará quedando que para $0 < i \leq n$, $v_i = B^i + \sum_{j|x_j \in c_j} B^{n+j}$ y $v_{n+i} = B^i + \sum_{j|\bar{x}_j \in c_j} B^{n+j}$, además de que por cada cláusula c_j tendremos $k_j - 1$ valores iguales a B^j y nuestro valor objetivo será $W = \sum_i^n B^i + \sum_j^m k_j B^{n+j}$.

	Dígitos desde n+1 hasta n+m	Dígitos desde 1 hasta n
W	$k_m \quad \dots \quad k_j \quad \dots \quad k_1$	$1 \quad \dots \quad 1 \quad \dots \quad 1$
W_i	$0 \quad \dots \quad 0 \quad 1 \quad 0 \quad \dots \quad 0$ 1 en la posición n+j si $x_i \in c_j$	$0 \quad \dots \quad 1 \quad \dots \quad 0$ 1 en la posición i
W_{i+n}	$0 \quad \dots \quad 0 \quad 1 \quad 0 \quad \dots \quad 0$ 1 en la posición n+j si $\bar{x}_i \in c_j$	$0 \quad \dots \quad 1 \quad \dots \quad 0$ 1 en la posición i

Figura 2.1: Ejemplo elementos de nuestro problema de suma de subconjuntos

Ahora que hemos creado la reducción veamos que esta es correcta. Si tenemos una solución del problema SAT, entonces tendremos una asignación de valores de veracidad que cumplen todas las cláusulas. Si realizamos la suma de todas las variables que son ciertas, tendremos que los primeros n dígitos sumarán 1, pues para cada variable x_i era verdadera o si era falsa sería verdadera $\bar{x}_i$, en cualquier caso siempre se escoje uno de los dos sumando 1 en el i-ésimo dígito. Ahora como esta asignación satisfacía todas las cláusulas tenemos que el los dígitos desde n+1 hasta n+m tenemos siempre suman al menos 1, pues alguna variable debía cumplirla, y como para cada cláusula c_i sólo hay k_i variables que pueden aparecer tenemos que la suma siempre será menor o igual a k_i en el dígito (n+i)-ésimo. Por tanto la suma en cada dígito (n+i)-ésimo estará entre 1 y k_i y podremos siempre completarla

hasta k_i sumandole algunos de los $(k_i - 1)$ valores B^{n+i}, teniendo así que hemos resuelto el problema de suma de subconjuntos. Por otro lado, supongamos ahora que el problema de suma de subconjuntos tiene solución y veamos que lo tenía el problema original. Por los n primeros dígitos, tenemos que si tenemos un subconjunto que suma W, entonces debemos haber escogido para cada dígito i-ésimo de los n primeros uno sólo de los dos valores que tienen 1 en ese dígito, así si se ha escogido el número v_i, entonces x_i es verdadero y si se ha escogido v_{n+i}, entonces x_i será falso. Con esto ya tenemos la asignación de veracidad de todas las variables de nuestro problema SAT. Por otro lado, para $0 < i \leq m$ tenemos que en el dígito n+i suman k_i y sólo tenemos $k_i - 1$ valores B^{n+i}, así que concluimos que alguna de las variables que aparecen en c_i debe ser verdadera, de nuevo porque nunca hay llevada al sumar. Como esto se cumple para todas las cláusulas deducimos que todas ellas se deben cumplir con nuestra asignación de veracidad.

Hemos creado la reducción desde SAT al problema de suma de subconjuntos y demostrado que las soluciones en uno de los problemas equivalen a uno en el otro. Por último sólo falta comprobar que esta reducción se puede hacer en tiempo polinomial. Para calcular nuestra W objetivo sólo necesitamos recorrer la entrada una vez para calcular cada k_i y para nuestra base B sólo debemos hallar el k_i máximo, ambas cosas se pueden hacer en tiempo lineal. Después por cada variable x_i deberemos crear dos números W_i y W_{i+n}, para los cuales deberemos recorrer cada cláusula para comprobar cuales los contienen. Por tanto, para las n variables deberemos recorrer 2n veces la entrada para crear sus correspodientes números asociados, lo cual sigue siendo de orden polinomial. Finalmente, hay que añadir $k_i - 1$ números por cada cláusula, que sólo requiere que se recorra la entrada una vez. Como además todos los números que hemos añadido son de longitud n+m, tenemos que en conjunto toda la transformación se puede hacer en tiempo polinomial. Con esto queda demostrado que el problema de suma de subconjuntos es NP-completo, pues SAT lo era.

2.3. Búsqueda del vector de mínimo peso de Hamming

En esta sección vamos a centrarnos en un problema de teoría de códigos. Para esto debemos primero definir algunos de los elementos claves de nuestro siguiente ejemplo de problema NP-completo.

Definición 2.3. Distancia de Hamming: Dadas dos palabras $c_1...c_n$ y $d_1...d_n$ que tengan la misma longitud se denomina distancia de Hamming al número de posiciones en que los símbolos c_i y d_i son distintos.

En teoría de códigos se presupone que nos referimos a la distancia entre el mensaje mandado y el mensaje recibido, por esto a cada posición diferente se la suele denominar error y la distancia de Hamming es el número de errores que contiene un mensaje respecto al otro.

Definición 2.4. Peso de Hamming: El peso de Hamming de una palabra $c_1...c_n$ es el número de símbolos distintos del símbolo en blanco del alfabeto de dicha palabra, equivalentemente es la distancia de Hamming a la palabra de igual longitud que sólo contiene el símbolo en blanco del alfabeto.

En general, si nuestras palabras representan números el símbolo en blanco es el o y, por tanto, el peso de Hamming es el número de cifras distintas de cero en cada número.

Definición 2.5. Problema de búsqueda del vector de mínimo peso de Hamming: Dada una matriz binaria A, un entero positivo w y un vector y, el problema consiste en determinar si existe un vector x de peso de Hamming menor o igual a w tal que $xA = y$.

Este problema es claramente NP, pues podemos escoger un vector aleatorio de peso Hamming menor que w y comprobar en tiempo polinomial, multiplicando por A, si es solución del problema.

Teorema 2.3. *[Vardy(1997)] El problema de búsqueda del vector de mínimo peso de Hamming es NP-completo.*

Para demostrar que este problema es NP-completo requeriremos usar un problema intermedio hasta llegar a SAT, primero demostraremos que el problema de acoplamiento de tripletas es reducible a este y después demostraremos que ese problema es NP-completo reduciendo el problema SAT a él.

Definición 2.6. Problema de acoplamiento de tripletas: Este problema también denominado problema de emparejamiento tridimensional consiste en dado tres conjuntos W, X e Y, y un subconjunto $M \subset W \times X \times Y$ determinar si existe algún subconjunto $M' \subset M$ con q elementos tal que cada par de elementos de M' (w_1, x_1, y_1) y (w_2, x_2, y_2) cumplan que $w_1 \neq w_2$, $x_1 \neq x_2$ y $y_1 \neq y_2$.

Primero es evidente que se trata de un problema NP, pues dado un conjunto M' como posible solución podemos recorrerlo comprobando que no contiene dos elementos que tengan mismos elementos en la misma posición y esto se puede calcular en tiempo polinomial.
Veamos que podemos realizar algunas presuposiciones previas en este problema sin perdida de generalidad. Primero podemos presuponer que $q = |W|$, obviamente q no puede ser mayor, porque todos los elementos de M' deben tener un elemento distinto de W y, por tanto, nunca podría existir un M' con más elementos que los que tiene W. Si q fuera menor que $|W|$, entonces sea k la diferencia entre ambos, podemos crear un nuevo problema equivalente añadiendo a X y a Y k nuevos elementos y añadiendo a M todas las combinaciones de esos k nuevos elementos con todos los elementos de W. Si en este nuevo problema encontramos un M' con $|W|$ elementos, podemos eliminar todos aquellos que contengan alguno de nuestros nuevos k elementos y nos quedará un M' con al menos q elementos que sólo contiene elementos del M de nuestro problema original.
Otra presuposición que podemos realizar es asumir que $|M| = |X| = |Y|$ e incluso que los tres son iguales. Lo último lo podemos presuponer porque los elementos particulares de los conjuntos no influyen en el problema, por tanto, si tienen el mismo número de elementos podemos presuponer que esos elementos son los mismos simplemente renombrando los elementos. Si asumimos que escogemos M como el conjunto de menos elementos, por el proceso anterior podemos suponer que q = $|M|$. Podemos hacer que $|X| = |Y|$, añadiendo al que tenga menos elementos hasta que tengan los mismos, al no incluir estos nuevos elementos en ningún elemento de M no alteramos el problema original. Ahora para conseguir que M tenga tantos elementos como X e Y, simplementele añadimos tantos elementos como le falten para tener la misma cardinalidad y para mantener que q =$|M|$, añadimos a M todas las combinaciones de estos nuevos elementos con todos los elementos de X e Y. De esta manera si hallamos un M' que resuelva este nuevo problema y le eliminamos todos los elementos que contengan alguno de los elementos que hemos añadido, obtendremos un M' que resuelve el problema original.

Lema 2.1. *El problema de acoplamiento de tripletas se puede reducir en tiempo polinomial al problema de búsqueda del vector de mínimo peso de Hamming.*

La demostración de este lema aprovecha las presuposiciones que hemos especificado antes. Así sin perdida de generalidad un problema de acoplamiento de tripletas se puede definir como sea W un conjunto finito y M un subconjunto de $W \times W \times W$ queremos buscar un conjunto M' de $|W|$ elementos. Veamos que podemos transforma este problema en un caso del problema de búsqueda del vector de mínimo peso de Hamming. Creemos una matriz de incidencia binaria de tamaño $|M| \times 3|W|$ que tenga por cada elemento de M una fila en la cual se encoden los elementos de esa tripleta. Las columnas estarán divididas en tres bloques de $|W|$ columnas y cada uno de esos bloques representarán que elemento de W contiene cada elemento de M. Así cada fila tendrá un 1 en el primer bloque de columnas, correspondiendo al elemento de W que tiene en su primera posición, tendrá otro 1 en el segundo y tercer bloque de columnas correspondiendo a los elementos de su segunda y tercera posición. De esta manera encontrar el conjunto M' es equivalente a encontrar $|W|$ filas tal que su suma en binario sea 11...11. Sea ahora esta matriz de incidencia nuestra matriz A y sea $y = 11...11$, es equivalente a encontrar el vector x que cumple $xA = y$ con peso de Hamming $|W|$. Como el tamaño de la matriz A que creamos es polinómico respecto a la entrada esta transformación es polinómica y, por tanto, hemos demostrado que el problema de acoplamiento de tripletas es reducible al problema de búsqueda del vector de mínimo peso de Hamming en tiempo polinomial. Sea $m_a \in M$ tal que $m_a = (m_a[1], m_a[2], m_a[3])$ su fila en la matriz sería:

	$w_1 \ \dots \ w_i \ \dots \ w_q$	$w_1 \ \dots \ w_j \ \dots \ w_q$	$w_1 \ \dots \ w_k \ \dots \ w_q$
m_a	$0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 0$	$0 \ \dots \ 1 \ \dots \ 0$	$0 \ \dots \ 1 \ \dots \ 0$
	1 en la posición i si $m_a[1] = w_i$	si $m_a[2] = w_j$	si $m_a[3] = w_k$

Figura 2.2: Ejemplo fila de nuestra matriz de incidencia

Ahora demostraremos que el problema de acoplamiento de tripletas es NP-completo reduciendo el problema SAT a él.

Lema 2.2. *El problema SAT se puede reducir al problema de acoplamiento de tripletas en tiempo polinomial.*

Supongamos tenemos un conjunto $U = u_1, ...u_n$ de variables lógicas y un conjunto $C = c_1, ..., c_m$ de cláusulas lógicas de los elementos de U. Ahora crearemos un caso del problema de acoplamiento de tripletas que sea equivalente a este. Nuestro problema tendrá los conjuntos W, X, Y y el subconjunto de tripletas $M \subset W \times X \times Y$. Por cada variable u_i en U incluiremos en W los elementos $u_i[j]$ y $\bar{u}_i[j]$ con $1 \leq j \leq m$ y en X e Y los elementos $a_i[j]$ con $1 \leq j \leq m$. Además añadiremos a M las tripletas $(u_i[j], a_i[j], a_i[j])$ con $1 \leq j \leq m$, las tripletas $(\bar{u}_i[j], a_i[j+1], a_i[j])$ con $1 \leq j < m$ y $(\bar{u}_i[m], a_i[1], a_i[m])$. Con esto nos aseguraremos que se escogen todas las tripletas con u_i o todas las tripletas con $\bar{u}_i$, pero que si se escogen de ambas a la vez no puedes nunca llegar a escoger m tripletas, sino que siempre terminas con menos para que no se repitan posiciones. Por cada cláusula c_j añadiremos un elemento $s[j]$ a X e Y, y incluiremos en M las tripletas $(u_i[j], s[j], s[j])$ para cada variable u_i que aparezca en c_j y $(\bar{u}_i[j], s[j], s[j])$ para cada variable u_i que aparezca negada en c_j. La idea es que si u_i es falsa escogeremos todas las tripletas que contienen a $u_i[j]$ y quedarán libres todas las que contienen $\bar{u}_i[j]$ tal que se puedan escojer todas las tripletas de cláusulas que se cumplan cuando u_i es falsa. Por último, escogeremos que el tamaño de M' que buscamos q sea $q = |W|$

para tener siempre el mismo tamaño de W. Tenemos que W contiene $2nm$ elementos y hasta ahora hemos escogido primero nm tripletas al seleccionar la veracidad de nuestras variables y después otras m cada una correspondiendo a una cláusula, por tanto, faltan $(n-1)m$ tripletas por elegir para completar nuestros $2nm$ elementos de M'. Por ello simplemente añadiremos $(n-1)m$ elementos a X e Y permitiendo que se combinen con todos los posibles elementos de W. Sea $1 \leq k \leq (n-1)m$ añadiremos $r[k]$ a X e Y, y añadiremos $(u_i[j], r[k], r[k])$ y $(\bar{u}_i[j], r[k], r[k])$ a M para $1 \leq i \leq n$ y $1 \leq j \leq m$.

Ahora veamos que ambos problemas son equivalentes. Si tenemos una solución del problema de acoplamiento de tripletas, quiere decir que hemos escogido mn tripletas de la forma $(u_i[j], a_i[j], a_i[j])$ o $(\bar{u}_i[j], a_i[j+1], a_i[j])$, tal que para cada variable queden libres m tripletas de forma $(u_i[j], a_i[j], a_i[j])$ significando que es falsa o de forma $(\bar{u}_i[j], a_i[j+1], a_i[j])$ significando que es verdadera. También habremos escogido una tripleta por cada cláusula, significando que con nuestra asignación de valores de verdad se cumplen todas las cláusulas. Esto se debe a que las cláusulas que sobran siempre sumarán únicamente hasta $(n-1)m$ tripletas, necesitando que se satisfagan todas las cláusulas para que M' sea de tamaño $2mn$. De forma inversa, si tenemos una solución del problema SAT, escogemos las nm tripletas correspondiendo a la asignación de valores de veracidad de cada variable y como sabemos que esta asignación cumple todas las cláusulas siempre se podrá escoger al menos una tripleta por cada cláusula, estas junto con las cláusulas extran sumarán hasta $2nm$.
Por último, en el peor caso cada cláusula puede contener hasta n variables, por tanto podemos estimar que nuestro problema tendrá como máximo $2mn + nm + 2m^2n(n-1) = 3mn + 2m^2n(n-1)$ tripletas, tal que esta reducción se puede realizar en tiempo polinomial.

Con esto hemos terminado de demostrar que el problema de búsqueda del vector de mínimo peso de Hamming es un problema NP-completo, pues hemos visto que el problema SAT se puede reducir polinomialmente al problema de acoplamiento de tripletas y este a su vez se reduce a nuestro problema. Como ya demostramos que SAT era NP-completo esto termina la demostración.

3 Criptografía

En este capítulo introduciremos conceptos básicos de la criptografía y explicaremos los objetivos generales de la misma. Posteriormente nos centraremos en determinados sistemas de criptografía asimétrica basados en algunos de los problemas NP-completos que ya hemos visto.

3.1. Introducción a la criptografía

En esta sección daremos una breve introducción a la criptografía en general, explicando la idea principal que intenta lograr cualquier criptosistema y además explicando la diferencia entre sistemas de criptografía simétrica y asimétrica.

La criptografía es una amplia rama de estudio que se centra en estudiar como se pueden codificar mensajes tal que receptores no autorizados del mismo no puedan conocer su contenido, pero sin que el mensaje pierda información y se pueda recuperar en su totalidad su contenido. El escenario básico en el que trabaja la criptografía es el siguiente, tenemos un emisor del mensaje y un receptor determinado del mismo, pero no tenemos una seguridad de que sólo el receptor vaya a recibir dicho mensaje o algún otro también vaya a leer el contenido de este mensaje. Lo interesante de este planteamiento tan general es que es aplicable a casi cualquier tipo de comunicación, no únicamente a la comunicación entre ordenadores. Cuando tenemos una simple conversación no tenemos la seguridad de que no haya alguien escuchando lo que decimos y queda fuera de nuestro conocimiento lo que le ocurra a cualquier carta que enviemos hasta alcanzar a su destinatario. La idea general de cualquier sistema criptográfico será la siguiente, el emisor aplicará una función de codificación al mensaje que dependerá de una clave y el emisor aplicará una función de decodificación al mensaje codificado que dependerá de otra clave. Para que esto tenga sentido, la función de decodificación debe invertir el efecto de la de codificación. Ilustremos esto con uno de los ejemplos más sencillos.

Ejemplo 3.1. Código Cesar: El código Cesar consiste en dado un mensaje escrito en un determinado alfabeto, que consideramos enumerado, cambiamos cada símbolo por el símbolo 3 posiciones más adelante y decodificamos sustituyendo cada símbolo por aquel 3 posiciones antes. Si usamos el alfabeto castellano la frase "Hola mundo"sería "Krñd oxpgr". El código Cesar es el cifrado de sustitución más antiguo conocido.

En el ejemplo anterior la función de cifrado sería $f_k(m) = m + k$ y la de decodificación $g_k(m) = m - k$, donde k sería la clave para ambos, en nuestro caso 3.

Para poder plantearnos cuales pueden ser buenas funciones de codificación, decodificación y claves debemos primero pensar en que queremos que cumplan. Nuestro objetivo final es que si alguien intenta atacar nuestro criptosistema, es decir, conocer el contenido del mensaje, esto le resulte lo más dificil posible. Primero de todo aclaremos que cuando nos referimos a funciones de codificación o decodificación, nos referimos a una familia de funciones y la clave es la que determina una única función dentro de esa familia. Si la función de codificación

y decodificación son desconocidas para el atacante posiblemente le sea mucho más dificil romper el sistema. Sin embargo, existen ciertos beneficios por dejar pública estas funciones, por ejemplo cuando dos ordenadores se conectan entre sí se intercambian las instrucciones del criptosistema que van a utilizar para su comunicaión, si la seguridad de ese criptosistema dependiese del desconocimiento de su funcionamiento interno esto no sería posible. Aunque se utilizan algunos sistemas que refuerzan su seguridad no permitiendo a nadie conocer su funcionamiento interno no es el tipo de sistema en que nos centraremos. Por tanto, la seguridad debe depender de la clave que utilicemos. En la mayoría de sistemas estas claves serán simplemente números y es importante para la seguridad de estos sistemas que dicha clave se escoja siempre de forma aleatoria y sólo se utilice una única vez. Esto se debe a que un proceso determinista para elegir la clave siempre se podría repetir por un atacante que conociese el funcionamiento del criptosistema y si repetimos el uso de claves damos información extra a un posible atacante para determinar dicha clave. Por tanto, todo sistema criptografico terminará siendo un proceso pseudoaleatorio. De esta forma las funciones que queremos son aquellas de las cuales no se pueda deducir la clave a partir del texto codificado, ni que se puedan invertir, o al menos que estas tareas sean computacionalmente muy costosas.

Dependiendo de como se usen las claves podemos distinguir dos tipos de criptografía. La más clásica e intuitiva es la criptografía simétrica, que utiliza una misma clave para encriptar que para desencriptar. El código Cesar es un ejemplo de este tipo. Los métodos de este tipo son los más rápidos y son los más utilizados para transmitir información, pero presentan un gran defecto. Para poder utilizar un sistema de criptografía simétrica ambos receptor y emisor deben conocer de antemano la clave. Si quieren comunicarse dos individuos que no habían interactuado antes, necesitarán alguna otra forma de comunicarse la clave antes de poder usar ningún sistema simétrico. Ante esto es lógico plantearse por qué entonces nadie usaría la criptografía simétrica, si ya poseemos otro método para enviarnos la clave de forma segura, podemos utilizar ese otro método para toda la comunicación. De nuevo esto es por una cuestión de eficiencia, la criptografía simétrica es muy rápida y supera notablemente en tiempo a la asimétrica. De esta manera, actualmente la comunicación siempre se realiza enviando primero una clave por algún método asimétrico y después comunicando todo lo demás mediante criptografía simétrica. Como contraposición a esta tenemos la criptografía asimétrica o también conocida como de clave pública y clave privada. La diferencia no es sólo que la clave de codificación sea distinta a la de decodificación, sino que la clave para codificar se dejará pública para que cualquiera que quiera la conozca y únicamente la clave para decodificar será privada y sólo conocida por el receptor del mensaje. Esto soluciona el problema anterior de cómo dos personas que nunca hubieran interactuado se pudiesen comunicar de forma segura, pero plantea un nuevo problema. Si cualquier persona conoce la clave pública y la función de codificación y además esta función deberá ser inyectiva, pues no queremos que dos mensajes tengan la misma codificaión y se puedan confundir, entonces un atacante podría invertir esta función y obtener el mensaje original. La forma de solucionar este problema será encontrado las funciones adecuadas.

Definición 3.1. Funciones con puerta trasera Se denominan funciones unidireccionales a aquellas funciones que sean fáciles de evaluar, es decir, se puede hacer en tiempo polinomial, pero sean difíciles de invertir, no se pueda hacer en tiempo polinomial. Se denominarán funciones con puerta trasera o funciones trampa si además existe un valor deteminado o clave que permite que se inviertan en tiempo polinomial.

Para crear sistemas de criptografía asimétrica necesitamos encontrar funciones con puerta

trasera que nos permitirán codificar rápidamente y, conociendo la clave privada, decodificar rápidamente, pero sean computacionalmente costosas de atacar. Es interesante remarcar que estas funciones son invertibles y que un atacante con suficiente tiempo y potencia computacional podría romperlas, siendo así la seguridad radicará en que esos tiempos sean muy superiores a los que queramos que nuestros mensajes sigan siendo seguros. Si calculamos que se tardarán al menos un siglo en decodificar nuestro mensaje probablemente nos parezca seguro, pues la información del mismo no será relevante para entonces. Precisamente por esto buscamos funciones que no se resuelvan en tiempo polinomial y es que el crecimiento terriblemente rápido de las funciones exponenciales nos asegurarán esos margenes temporales que buscamos. Llegados a este punto es lógico concluir que podemos utilizar los problemas NP-completos para esta tarea, pues de ellos se puede comprobar si una solución es correcta en tiempo polinomial, pero no se puede encontrar dicha solución en un tiempo polinomial. Nos resta ahora explicar cómo podemos aprovechar dichos problemas para elaborar sistemas criptográficos. Nos centraremos ahora en diferentes sistemas de criptografía asimétrica, comenzando por RSA, por ser el más utilizado actualmente.

3.2. Criptosistema RSA

En esta sección explicaremos el criptosistema RSA, Rivest-Shamir-Adleman [Rivest, Shamir, and Adleman(1978)], que es el sistema de criptografía asimétrica más conocido y utilizado hasta ahora. Este criptosistema utiliza ideas sencillas de la aritmética modular para crear una encriptación fácil de realizar, pero difícil de invertir, mediante el uso de módulos respecto a números suficientemente grandes. Al haberse utilizado tan extensamente este sistema se han desarrollado muchas formas de atacarlo, principalmente centrandose en conseguir factorizar un número con factores primos muy grandes. Actualmente como la computación cuántica ha mostrado una forma rápida de factorizar números se están buscando alternativas más seguras a este sistema, aún cuando todavía se sigue usando en la mayoría de sistemas.

La idea principal de este sistema es que si escogemos un número n muy grande podemos encriptar nuestro mensaje m como $c = m^e \, mod \, n$ para algún exponente e, si alguien quisiese deshacer este proceso tendrían que hallar la raiz e-ésima de c, que es computacionalmente muy costoso. Sin embargo, para que el receptor pueda desencriptar este mensaje de forma eficiente nos aprovecharemos de las propiedades de la función de Euler. Escojamos dos números primos p y q, consideraremos $n = p \times q$, ahora podemos calcular $\phi(n)$, aprovechando que como n sólo tiene dos factores primos podemos usar las propiedades de la función de Euler, tal que $\phi(n) = (p-1)(q-1)$. Escojamos un e que sea primo relativo con $\phi(n)$ y, por tanto, poseerá un inverso módulo $\phi(n)$. Denominemos d a ese inverso tal que $ed = 1 \, mod \, \phi(n)$. Aplicando el teorema chino del resto podemos deducir que para cualquier entero $x < n$ tenemos que $x = x^{ed} \, mod \, n$, es decir, que calcular la raiz e-ésima módulo n es equivalente a elevar a d, lo que es computacionalmente más fácil.

Si un ususario quisiese utilizar este sistema para comunicarse tendría que hacer el siguiente proceso. Primero escogería dos números primos p y q muy grandes y con ellos calcularía n y $\phi(n)$. Después escogería un exponente e, como este será público no aporta ninguna seguridad utilizar uno aleatorio, por tanto, casi siempre se usan primos de Fermat, principalmente el 65537. El uso de este exponente en particualr se debe a que para calcular potencias se puede obtimizar el cálculo con el algoritmo de cuadrados iterados, que consiste en elevar

al cuadrado reiteradas veces un número para obtener sus potencias cuyo exponente tiene base 2. Como $65537 = 2^{16} + 1$, las potencias con este exponente son muy eficientes, esto no merma la seguridad del sistema, pues el valor de n sí cambia cada vez y su inverso d también. Por último, se debe calcular d a partir de e y $\phi(n)$. El valor n y e serán las claves públicas, mientras que el valor de d será la clave privada. Si cualquiera quiere enviarle un mensaje a este usuario, tomarán ese mensaje y lo convertirán en un entero m menor que n. Después calcularán $c = m^e \, mod \, n$ y enviarán c. El receptor del mensaje conociendo la clave privada calculará $m = c^d \, mod \, n$ y podrá conocer el contenido del mensaje. Cualquier otro que quiera conocer el mensaje sin conocer el valor de d deberá calcular la raiz e-ésima que es computacionalmente muy costoso para valores de e grandes.

Una de las formas más directas de intentar atacar este sistema es intentar calcular el valor de d. Quien calcula su clave privada encuentra este valor fácilmente, pues puede computar $\phi(n)$ rápidamente usando p y q. De esta manera si logramos hallar p y q podemos rehacer exactamente el mismo proceso que para crear la clave privada y así poder desencriptar cualquier mensaje codificado con ese valor n. Por tanto, la seguridad de todo este sistema radica en la dificultad de encontrar p y q a partir de n. Esto puede resultar altamente contraintuitivo, pues conceptualmente factorizar un número en sus factores primos es una tarea muy sencilla y es algo que se enseña a realizar muy pronto en el aprendizaje de conocimientos matemáticos. De hecho, tampoco parece una tarea costosa para un ordenador, pues son capaces generalmente de factorizar muy rápidamente la mayoría de números que se les pida. Esto se debe a una cuestión estadística. Si escogemos un número aleatorio de un determinado número de cifras, tendremos que la mitad de ellos son divisibles por 2, un tercio serán divisibles por 3 y así. De esta manera es muy probable que el número escogido tenga factores primos muy pequeños que permitan su rápida factorización. Sin embargo, en términos generales es un problema que tiene una complejidad no polinomial y, en la práctica son los casos con factores primos muy elevados los que vuelven intratable este problema. Esta es la razón por la que escojemos un n que posea sólo dos factores primos lo más grandes posibles. De nuevo remarcar que en específico este problema sí se puede resolver en tiempo polinomial por un ordenador cuántico. Lo cual implica que este sistema ya se prevee que se vaya a quedar obsoleto conforme se mejore la tecnología de los ordenadores cuánticos. Por este motivo, aunque resulta un sistema muy ilustrativo y probablemente es el más sencillo sistema de criptografía asimétrica, no vamos a experimentar en como romper la seguridad de este sistema.

3.3. Criptosistema basado en el problema de la suma de subconjuntos

En esta sección explicaremos una de las posibles formas en que se puede crear un criptosistema utilizando el problema de la suma de subconjuntos, que ya demostramos previamente que era de clase NP-completo. En específico, explicaremos el primer criptosistema que se inventó basado en este problema, el de Merkle-Hellman[Merkle and Hellman(1978)]. Mostraremos como podemos crear versiones fáciles de resolver del problema que enmascararemos como versiones generales del mismo. Además detallaremos como se puede crear un sistema de criptografía asimétrica presentando cuales deberán ser las claves públicas y privadas. Por último evaluaremos la posible seguridad que presenta este criptosistema frente a ataques.

Hemos demostrado que el problema general de la suma de subconjuntos es un problema de clase NP-completo y, por tanto, que no existe un algoritmo que pueda resolver de forma eficiente cualquier caso de este problema. Por supuesto, esto no implica que no se puedan crear casos específicos del problema que se puedan resolver de formas muy rápidas. Para ilustrar esto tenemos el siguiente ejemplo:

Ejemplo 3.2. Sea $V = \{1, 2, 4, 8, 16, 32, 64, 128\}$ y $W = 77$, tendremos que $W = 1 + 4 + 8 + 64$, que es equivalente a encontrar la representación en binario de 77, es decir, 1001101.

En general sea $V = \{1, 2, 4, ..., 2^n\}$ y sea $W = x$ tal que $x < 2^{n+1}$, tendremos que podemos encontrar la solución al problema reescribiendo x en binario, pues invirtiendo el orden a dicha representación nos dará una cadena binaria en que cada uno representa la posición del valor de V que debemos añadir. De hecho, en general cualquier solución al problema de la suma de subconjuntos se puede expresar como un vector binario de misma longitud que V tal que la suma de los v_i que tengan un uno en su posición correspondiente sea W. Podemos denominar $S = \{s_1, ..., s_n\}$ a ese vector solución en binario de igual longitud que V tal que $s_i = 1$ si y sólo si la solución incluye a v_i.

Podemos ampliar este caso específico a muchos otros mediante la siguiente idea. Sea $V = \{v_1, ..., v_n\}$, tal que $v_i > \sum_{j=1}^{i-1} v_j, \forall i \leq n$, entonces dado $W = x$, tendremos que cualquier posible solución deberá incluir a v_n, equivalentemente $s_n = 1$, si y sólo si $x \geq v_n$, puesto que sumando todos los demás valores de V nunca se llega a sumar hasta v_n. Siguiendo la misma lógica tendremos que $s_{n-1} = 1$ si y sólo si $x - v_n * s_n \geq v_{n-1}$ y, en general, $s_i = 1$ si y sólo si $x - \sum_{j=i+1}^{n} v_j * s_j \geq v_i$. Cualquier instancia del problema que tenga estas características se puede resolver de foma muy rápida siguiendo los cálculos anteriores, por ello, denominaremos a este tipo de casos como las versiones fáciles del problema.

Veamos ahora como podemos pasar de uno de estos casos sencillos a uno más complicado de resolver. Presupongamos que poseemos V y W una versión fácil del problema como descrita anteriormente, de la cual podemos hallar la solución S fácilmente, y elijamos dos números m y p tal que sean primos relativos y $m > \sum v_i$. Podemos crear una nueva versión V' y W' del problema mediante la siguiente relacción:

$$v'_i = p * v_i \, mod \, m$$

$$W' = p * W \, mod \, m$$

Analicemos ahora este problema de forma aislada. La distribución general que nos dará este procedimiento es de un caso del problema de suma de subconjuntos que no cumple los requisitos anteriores de las versiones fáciles y, por tanto, a priori no podemos aplicar ningún algoritmo sencillo para resolver este problema, si no tenemos conocimiento previo de los valores V, W, m o p. Por otro lado conocemos que este problema es NP-completo, es decir, si creasemos un caso totalmente genérico del problema, escogiendo todos los valores de V aleatoriamente y creando un vector solución S también aleatoriamente, entonces ese caso no se podría resolver de forma eficiente. Este proceso por el que hemos creado este problema sigue una distribución pseudo-aleatoria, que aunque en apariencia los valores no parezcan seguir ningún patrón que facilite resolver el problema fácilmente, tampoco podemos asegurar que este método nos pueda devuelva cualquier caso del problema y no un subconjunto particular de casos del problema. Si ocurriese esto último puede ser que el conjunto específico de problemas que se obtienen mediante este método sí admitan un algoritmo eficiente que

los resuelva. En cualquier caso, hasta ahora no existen evidencias de que se pueda explotar el hecho de que un problema de suma de subconjuntos se halla generado de esta forma para resolverlo en tiempo polinomial. El hecho de carecer de esta certeza no es un fenómeno aislado en la criptografía, de nuevo este método es seguro mientras nadie halla encontrado la forma de resolver estos problemas particulares. Por otro lado, más adelante explicaremos como se podría volver más seguro este método mediante el uso iterado de este método.

Para poder utilizar este problema en un criptosistema nos interesa poder usarlo como una función con puerta trasera, ya hemos creado una versión aparentemente difícil del problema, por tanto, nos queda mostrar como podemos resolverlo fácilmente usando la información de cómo se ha creado dicho problema. Supongamos conocemos p y m, entonces podemos volver al problema original V y W con la siguiente relación:

$$v_i = p^{-1} * v'_i \, mod \, m$$

$$W = p^{-1} * W' \, mod \, m$$

Donde utilizamos que mcd(p,m)=1. Además podemos usar que $m > \sum v_i$ tal que:

$$W = p^{-1} * W' \, mod \, m = p^{-1} * \sum v'_i * s_i \, mod \, m$$

$$W = p^{-1} * \sum v_i * p * s_i \, mod \, m = \sum v_i * s_i \, mod \, m = \sum v_i * s_i$$

Tal que W sería igual a $\sum v_i * s_i$ en la suma usual como en módulo m. Ahora podríamos resolver el caso fácil de V y W y utilizar que el vector solución S es el mismo para ambos problemas.

Centremonos ahora en los detalles de como se utilizaría todo este método para construir un sistema de criptografía asimétrica. Un usuario que quisiese poder recibir mensajes que sólo él pudiera leer tendría que primero crear una versión fácil del problema V, con las condiciones que hemos visto, que pudiera resolverse fácilmente para cualquier W. Una forma en que se podría crear este conjunto V podría ser escoger aleatoriamente v_1 en un rango $[1, 2^{100}]$ y los siguientes v_i en rangos $[(2^{i-1} - 1) * 2^{100} + 1, 2^{100+i-1}]$. De esta forma se obtiene una versión fácil del problema, para transformarla en una versión difícil escogerá valores m y p. Como queremos que m sea mayor que la suma de todos los v_i podemos escogerlo de los mismos rangos de antes para $i = n + 1$. El valor de p se puede escoger tomando un número aleatorio en $[2, m - 2]$ y dividiendo por el mínimo común multiplo de este con m. Con estos valores escogidos el usuario crearía el conjunto V' que sería la versión difícil del problema a partir de V, m y p. El conjunto V no sería necesario almacenarlo, pues se puede obtener de V', m y p, por tanto, la clave privada serían los valores m y p. La clave pública sería el propio conjunto V' que se dejaría accesible para que cualquier otro usuario que quisiese contactar con este pudiera conocerla.

Ahora supongamos que alguien quisiese enviarle un mensaje a este usuario. Ese mensaje se reescribiría en binario en un vector de longitud menor o igual a la del conjunto V'. Recordemos que generalmente no se usa la criptografía asimétrica para intercambiar grandes textos, sino que principalmente se usa para enviar claves que permitan continuar después utilizando un sistema de criptografía simétrica más rápido, por tanto, no requerimos que el tamaño de este mensaje sea excesivamente grande. Obtenemos el vector binario S que

contiene el mensaje y lo vamos a considerar como la solución al problema de la suma de subconjuntos de V'. Por tanto, calculamos la W' correspondiente que tendría S como solución, lo cual es muy rápido, pues $W' = \sum_{i=1}^{n} v'_i * s_i$. Ese valor W' es el que se enviaría al usuario. El receptor del mensaje como posee la clave privada puede volver al problema original V y encontrar aquí el vector solución S, desencriptando el mensaje y pudiendo leer su contenido. Cualquier otro que recibiese el mensaje tendría el valor W' y conocería V' por ser público. Con lo cual si quisiese descifrar el mensaje S original tendría que resolver el problema de la suma de subconjuntos que al ser NP-completo no se conoce una forma rápida de poder hacer y, si se escoge un V de tamaño n suficientemente grande, se tardarían tiempos inasumibles para cualquier posible uso práctico. En el peor de los casos existen 2^n combinaciones posibles de subconjuntos que se tendrían que comprobar.

Por último volvamos a la cuestión de la seguridad. Es cierto que no tenemos ninguna certeza sobre el conjunto de problemas que se obtienen a partir de nuestro enmascaramiento de problemas fáciles. Estos problemas aparentan ser una colección aleatoria de valores sin ninguna distribución específica, pero por la forma en que los obtenemos sabemos que no es así y existe la duda razonable de si estos problemas fáciles enmascarados no guardarán todavía una forma fácil para ser resueltos. De hecho, pocos años después de que fuera publicado por primera vez este sistema ya se encontró una forma de romper su seguridad [Shamir(1984)]. Sin embargo, poseemos una forma de volver más robusta la manera de enmascarar nuestros problemas. Hemos visto que si conocemos los valores m y p podemos pasar de V y W a V' y W' y viceversa eficientemente. Por tanto, nada nos impide considerar V' y W' como nuestro problema fácil y escoger con exactamente el mismo método valores m' y p' que nos permitieran crear V'' y W''. Conociendo m, m', p y p' podemos ir desde V'' y W'' hasta V y W, por tanto, seguimos sabiendo como resolver V'' y W''. No existe ninguna limitación a la cantidad de veces que podemos iterar este método. Otros sistemas criptográficos, por ejemplo un cifrado por sustitución, no se benefician de aplicarse reiteradas veces, pues la composición de dos sustituciones nos da otra sustitución distinta. Comprobemos con un ejemplo que este no es el caso para nuestro sistema.
Sea $V = \{5, 10, 20\}$, $m = 47$ y $p = 17$, $m' = 89$, $p' = 3$, entonces será $V' = \{38, 29, 11\}$ y $V'' = \{25, 87, 33\}$. Supongamos existen $\bar{m}$ y $\bar{p}$ que nos llevan V en V''. Como $v_1 = 25$ y $v''_1 = 5$, tenemos:

$$25 = \bar{p} * 5 \, mod \, \bar{m} \Longrightarrow 50 = \bar{p} * 10 \, mod \, \bar{m}$$

Por otro lado $v_2 = 87$ y $v''_2 = 10$

$$87 = \bar{p} * 10 \, mod \, \bar{m} \Longrightarrow 87 = 50 \, mod \, \bar{m} \Longrightarrow 37 = 0 \, mod \, \bar{m}$$

Por tanto, $\bar{m} = 37$ y además $\bar{p} = 5$ por la siguiente ecuación

$$25 = \bar{p} * 5 \, mod \, 37$$

Sin embargo, esto entra en contradicción con que $v_1 = 20$ y $v''_1 = 33$

$$33 \neq 5 * 20 \, mod \, 37 = 26 \, mod \, 37$$

En definitiva, el iterar este método sí aporta seguridad extra frente a su aplicación única. Esto nos proporciona una forma de incrementar la seguridad de este criptosistema a costa de un incremento de la complejidad. Por desgracia, este sistema en particular, incluyendo

la iteración de la ocultación del problema inicial, ya ha sido roto mediante ataques de baja densidad[Odlyzko(1998)]. Esto no quita que puedan existir otros problemas basados en este mismo problema NP-completo y sigue siendo un ejemplo ilustrativo del diseño de criptosistemas apoyados en problemas NP-completos.

3.4. Criptosistema basado en teoría de códigos

En esta sección explicaremos como podemos utilizar el problema de la búsqueda del vector de mínimo peso de Hamming para crear un sistema criptográfico. En este caso expondremos el criptosistema de McEliece que fue uno de los primeros en incluir aleatoriedad en el proceso de encriptación y es un criptosistema que todavía no se ha roto[McEliece(1978)]. Codificaremos mensajes considerandolos como vectores, multiplicandolos por matrices y añadiendoles errores aleatorios. Al igual que en el caso anterior, partiremos de la idea de que existen casos particulares en que podemos deshacer esta transformación fácilmente. Aplicando teoría de códigos conocemos casos particulares de matrices que nos permiten codificar mensajes con la capacidad de que si al recibir dicho mensaje conteniese errores seríamos capaces de corregirlos. En general intentar resolver el problema de corregir errores en un mensaje codificado con una matriz genérica es de complejidad equivalente a calcular un vector de mínimo peso Hamming, lo cual como ya demostramos es un problema de clase NP-completo.

Para ilustrar mejor la idea detrás de este criptosistema centremonos en una idea básica de la teoría de códigos. Digamos que queremos enviar un mensaje m, codificado en binario, pero tememos que este mensaje se pueda ver alterado antes de llegar al receptor y terminará recibiendo $m + e$, donde e es un vector de errores. Para poder solventar este problema podemos codificar nuestro mensaje en un vector de mayor tamaño que posea información redundante que nos permita corregir posibles errores al recibirlo, presupuesto que el número de errores sea pequeño, es decir, que el peso de Hamming del vector de errores sea pequeño. La forma en que nos va a interesar hacer esto es, dado que queramos enviar un mensaje de tamaño n, utilizar una matriz C de dimensión $n \times k$, para $k > n$, tal que enviemos el mensaje mC que es de longitud k. Supuesto que recibamos un mensaje $mC + e$ y que no exista ningún m' tal que $m'C = mC + e$, entonces podremos calcular el vector $\hat{m}$ que al multiplicarlo por C tenga menor distancia de Hamming a $mC + e$. Este $\hat{m}$ es el vector que menor número de alteraciones hubiera necesitado para convertirse en lo que recibimos, por tanto, es el vector que más probablemente fue el que se envió en primer lugar. Denominemos t al número mínimo de errores que queremos ser siempre capaces de corregir, es fácil imaginar que dado una dimensión k suficientemente grande una matriz aleatoria de máximo rango podría permitirnos hacer este proceso, pero si escogemos de forma estratégica dicha matriz podemos asegurarnos que este cálculo se pueda realizar siempre de forma eficiente.

Existen muchas posibles elecciones para estas matrices. Sin embargo, criptosistemas basados en algunas de las posibles elecciones ya se han conseguido romper recuperando dicha matriz y, así, corrigiendo el error de forma eficiente. En nuestro caso destacaremos las matrices Goppa, pues se han mostrado hasta ahora resistentes a la mayoría de posibles ataques, no habiendose roto este criptosistema cuando se usan matrices Goppa, y además siendo estas en las que se basaba el sistema original de McEliece .

Definición 3.2. Códigos Goppa: Un código Goppa se define mediante un polinomio g(x) de grado t sobre un cuerpo de Galois F_{p^m} de p^m elementos que no contenga raices múltiples.

En nuestro caso sólo consideraremos las matrices binarias tal que $p = 2$. Además de con un conjunto $\{A_1, ..., A_n\}$ de n elementos de dicho cuerpo que no anulen dicho polinomio. Dado un elemento $c = (c_1, ..., c_n)$ con $c_i \in Z_p$, podemos definir la siguiente función:

$$R_c(x) = \sum_{i=1}^{n} \frac{c_i}{x - A_i}$$

Si consideramos el conjunto T de elementos c tal que $R_c(x) \equiv 0 \ mod \ (g(x))$, tendremos que tiene una dimensión de al menos $n - mt$ y que la distancia de sus elementos es de al menos $2t + 1$. Tal que podemos considerar este espacio como el conjunto de mensages que podemos codificar que tendrá al menos dimensión $n - mt$ y que nos permitirá corregir al menos t errores de los mensages codificados con este código.

Además se puede definir una matriz de paridad para este código Goppa de la forma $H = VD$, donde V es una matriz que definida por columnas tiene en la columna i las potencias desde 0 a $t - 1$ de los elementos A_i y D es una matriz diagonal donde el i-ésimo elemento de su diagonal es $\frac{1}{g(A_i)}$. La matriz de paridad cumple que para todo elemento $c \in T$ se cumplirá $Hc^T = 0$. De esta manera podemos buscar cualquier matriz G que cumpla $GH^T = 0$, tal que esta matriz generará todo el espacio T y es la que denominaremos matriz Goppa de este código Goppa.

Las matrices Goppa fueron inventadas por Valery Denisovich Goppa y son muy utilizadas en teoría de códigos, porque su capacidad de correción de errores tiende al límite de Gilbert-Varshamov[Varshamov(1957)] sobre el tamaño de un código. Restaría únicamente mostrar el algoritmo para corregir errores de forma eficiente, pero eso queda fuera del alcance de este proyecto y no influye en el funcionamiento del criptosistema de McEliece. Por tanto, remitimos al artículo original sobre códigos Goppa para su explicación [Goppa(1970)].

Definición 3.3. Problema de decodificación de un códico lineal general: Dada una matriz C binaria de dimensión $n \times k$ y rango n, un vector m' y un valor entero positivo t. Se denomina problema de decodificación de un código lineal general, en su versión de problema de decisión, a determinar si existe algún vector e de peso de Hamming menor o igual a t, tal que $m' = mC + e$ para algún vector m.

Veamos que el problema de búsqueda del vector de mínimo peso de Hamming se puede reducir al problema de decodificación de un código lineal general. Definimos el problema de busqueda del vector de mínimo peso de Hamming multiplicando por la derecha por una matriz, es decir, $xA = y$, el problema es equivalente si ahora multiplicamos por la izquierda, $Ax^T = y^T$. Sea A una matriz binaria de dimensión $p \times k$ y rango p, un valor w y un vector y. Sea $n = k - p$, podemos determinar una matriz C de dimensión $n \times k$ tal que $AC' = 0$ en tiempo polinomial haciendo una eliminación de Gauss. Entonces podemos encontrar un vector m' que cumpla que $A(m')^T = y^T$ resolviendo un sistema lineal, lo cual también se puede realizar en tiempo polinomial. Si existe un vector x de peso de Hamming menor o igual a w que cumpla que $Ax^T = y^T$, entonces ocurrirá que $A(m' - x)^T = 0$. Por tanto, existirá un m tal que $m' = mC + x$ y x será de peso Hamming menor o igual que w. Esto implica que el problema de decodificación de un códico lineal general es un problema NP-completo.

Supongamos que conocemos una matriz C para la cual conocemos un algoritmo rápido para corregir hasta t errores, aplicando transformaciones lineales podemos transformar esta

matriz en una que parezca genérica y que no presente a priori una forma rápida de resolver el problema de decodificación de un códico lineal con ella. Sea la matriz C de tamaño $n \times k$, donde n es el tamaño de nuestro mensaje, escogeremos dos matrices P y S. La matriz P será una matriz de permutación de dimensión $k \times k$ y la matriz S será una matriz cualquiera no singular de dimensión $n \times n$. Con estas podremos computar la matriz $\hat{C} = SCP$, que tendrá la misma dimensión que C, pero que presenta una distribución aleatoria, pues P y S son matrices aleatorias.
Ahora veamos que conocidas las matrices P y S la matriz $\hat{C}$ todavía permite corregir t errores. Supongamos se envía el mensaje $c = m\hat{C} + e$, donde e es un vector de errores de peso de Hamming menor o igual a t y queremos conocer m. Primero calculamos la inversa de P y calculamos $\hat{c} = cP^{-1} = m\hat{C}P^{-1} + eP^{-1} = mSC + eP^{-1}$, donde es relevante que P sea una permutación pues nos asegura que eP^{-1} sigue teniendo el mismo peso de Hamming que e. Ahora como conocíamos un algoritmo para corregir errores de la matriz C, lo podemos usar con $\hat{c}$ y calculamos mS. Por último, como S no es singular calculamos su inversa y obtenemos m. De esta manera conociendo P y S se puede de manera rápida corregir errores con $\hat{C}$. Presupongamos que no conocemos dichas matrices, entonces estaríamos ante una matriz aparentemente aleatoria y nos enfrentaríamos al problema de decodificación de un códico lineal general que es de clase NP-completo, tal que no tendríamos ninguna forma rápida de corregir errores. Es interesante remarcar que la familia de matrices C que se usen puede afectar a la seguridad del sistema. Algunas elecciones de dichas familias permiten a alguien que desconozca P y S, pero sepa la familia de C, reconstruir C y así conseguir una forma rápida de corregir errores, rompiendo la seguridad de este sistema. Las matrices binarias Goppa hasta el momento han resistido este tipo de ataque y, por ello, son las más comunes para este tipo de criptosistemas.

Finalmente, el funcionamiento completo del criptosistema sería el siguiente. Un usuario que desease poder recibir mensajes encriptados escogería una matriz C de la que conociera un algoritmo para corregir hasta t errores y dos matrices P y S aleatorias. Las matrices P, S y C o el algoritmo para corregir errores, serían la clave privada del usuario. Calcularía la matriz $\hat{C} = SCP$ que sería la clave pública, junto con el valor t. Esta clave se daría a conocer a cualquiera que quisiese comunicarse con el usuario. Cualquiera que quisiese enviarle un mensaje m encriptado leería $\hat{C}$ y t y enviaría $m\hat{C} + e$, donde e fuera un vector aleatorio de peso de Hamming t. De la forma que hemos visto anteriormente, el usuario que conce la clave privada es capaz de desencriptar el mensaje y conocer m en un tiempo eficiente. Cualquier otro que interceptase dicho mensaje tendría que ser capaz de corregir t errores conociendo que ha sido encriptada con la matriz $\hat{C}$, lo cual es un problema NP-completo tal que para una matriz suficientemente grande dicha tarea sería inasumible.

4 Experimentación

En este capítulo nos planteamos posibles formas de atacar los criptosistemas que hemos estudiado hasta ahora. Los criptosistemas que hemos presentado estaban todos basados directamente en la resolución de un problema NP-completo y utilizabamos que con información de la clave privada podíamos encontrar una manera rápida de resolver los casos específicos de nuestros criptosistemas. Intentar crear algoritmos generales para resolver estos problemas NP-completos es una tarea notablemente difícil, pues implicaría resolver el problema abierto de si P =NP. Por otro lado, podríamos intentar aplicar nuestro conocimiento sobre como se genera el criptosistema para determinar si los posibles problemas que originan son un subconjunto particular del problema general que sí admite una solución eficiente. Sin embargo, se han diseñado estos criptosistemas con la idea principal de enmascarar estos problemas para que parezcan lo más generales posibles.

El enfoque en que nos centraremos será en intentar resolver estos problemas con algoritmos estocásticos, principalmente los algoritmos genéticos. Sabemos que encontrar un algoritmo que resuelva cualquier problema en un tiempo razonable es muy complicado, pero si conseguimos un algoritmo que algunas veces sea capaz de encontrar la solución habremos roto la seguridad del algoritmo. Por ejemplo, un criptosistema que se pueda romper en un 1 % de los casos, es decir, que se puedan descifrar el 1 % de los mensajes codificados con el mismo, es un criptosistema que no nos aporta suficiente seguridad para ser útil. Para intentar que al menos en algunas ocasiones sea capaz de resolver el problema es importante que el algoritmo contenga un componente aleatorio.

4.1. Algortimos genéticos

En esta sección explicaremos la estructura básica de un algoritmo genético y posteriormente presentaremos resultados de los mismos frente a problemas NP-completos que hemos visto. Un algoritmo genético es una metaheurística con el objetivo de maximizar una función adaptando ideas originarias de la biología. Como la mayoría de las metaheurísticas es un procedimiento estocástico que debería converger al máximo que buscamos, pero del que no podemos asegurar a priori su orden de convergencia más allá de que gracias a la aleatoriedad de las mutaciones en tiempo infinito la probabilidad de que encuentre dicho máximo tiende a uno. Las metaheurísticas inspiradan en la naturaleza como esta no suelen tener una base teórica robusta que las soporten, sino que intentan imitar estrategias que ya funcionan en la naturaleza, en este caso el funcionamiento de la evolución.

Un algoritmo genético se basa en imitar ideas de la genética siguiendo la lógica de la evolución darwinista de la supervivencia del más fuerte. Se centra en crear una población en que cada individuo posee una serie de genes que determinarán como de aptos son mediante una función de adaptabilidad. En base a la adaptabilidad de cada individuo se escogerán que individuos sobreviven y estos se reproducirán generando individuos con genes similares. Siguiendo la lógica evolutiva esto llevará a que en la población terminen predominando

los individuos con genes que maximizan la función de adaptabilidad. De esta manera la función de adaptabilidad será nuestra función objetivo que determinará el problema que intentamos resolver, los genes serán las variables que podemos variar y el algoritmo genético nos permitirá maximizar o minimizar dicha función, según convenga.

Un algoritmo genético tendrá 5 procesos principales: inicialización, selección, reproducción, mutación y reemplazamiento. Las diferencias entre los diferentes algoritmos radicarán en la forma en que realizan estos procesos. La inicialización es la elección de la población inicial, la forma más directa es escogerla totalmente aleatoria, pero si poseemos algún método rápido de generar una población que ya sea relativamente buena para nuestra función de adaptabilidad podemos acelerar notablemente el tiempo que tarda el método en converger. La selección consiste en escoger de la población a individuos que se vayan a convertir en progenitores mediante la función de adaptabilidad, tal que los que tengan mayor adaptabilidad tengan más probabilidades de ser escogidos. La reproducción consiste en dados los progenitores escogidos en la selección crear nuevos individuos que vayan a constituir la nueva población, de tal manera que retengan los genes de sus progenitores. La mutación consiste en seleccionar algunos individuos y mutarlos, es decir, cambiar algunos de sus genes por otros totalmente aleatorios, para el correcto funcionamiento de un algoritmo genético es imprescindible este elemento de aleatoriedad, pues es el único componente que permitirá al método salir de máximos locales hacia el máximo global. Finalmente el reemplazamiento consiste en sustituir parte de nuestra población original por los individuos de la nueva población.

Algorithm 1: **Algoritmo Genético General**

```
1: Población <- Inicializar población
2: while no se cumpla el criterio de parada do
3:    Adaptabilidad <- Función Objetivo(Población)
4:    Progenitores <- Selección(Población,Adaptabilidad)
5:    NuevaPoblación <- Reproducción(Progenitores)
6:    NuevaPoblación <- Mutación(NuevaPoblación)
7:    Poblacion <- Reemplazamiento(Poblacion,NuevaPoblación)
8: end while
```

Ahora detallaremos los métodos particulares que se han utilizado en los algoritmos. La inicialización de la población ha consistido en la selección totalmente aleatoria de individuos del conjunto de todas las posibles combinaciones de genes. La mutación se ha realizado estableciendo para cada algoritmo un valor fijo de probabilidad de mutación por gen por individuo en cada generación. El reemplazamiento se ha realizado escogiendo a un número T de los mejores individuos de la generación anterior e incorporando estos a la nueva población. Para optimizar la velocidad del algoritmo siempre se calcula una nueva generación con exactamente T individuos menos que la anterior, tal que al incorporarle los mejores de la generación anterior vuelva a haber la misma cantidad de individuos que en la anterior generación. De esta forma todas las generaciones tienen el mismo número de individuos.

La selección se ha hecho mediante una selección por torneo binario. Un torneo binario consiste en dados dos individuos escoger aquel con la mayor adaptabilidad. Si tenemos una población de N individuos y queremos escoger K de ellos, entonces escogemos aleatoriamente dos de ellos y nos quedamos con el ganador de el torneo entre ambos, si repetimos este proceso K veces obtendremos nuestro conjunto de progenitores.

Algorithm 2: **Selección por torneo binario(N,K)**

for $i \in [0, .., K]$ **do**
  i_1, i_2 <- GenerarIndicesAleatorios$(0, N-1)$
  if $Adaptabilidad[i_1] > Adaptabilidad[i_2]$ **then**
    Añadir Población$[i_1]$ a Progenitores
  else
    Añadir Población$[i_2]$ a Progenitores
  end if
end for

Por último la reproducción se ha realizado mediante un cruce uniforme. Se realiza una partición de los genes en dos mitades tal que cada parte contenga el mismo número de genes. Después se escogen todos los genes de una mitad de uno de los progenitores y el resto de genes se escogen del otro progenitor.

Algorithm 3: **Reproducción por cruce uniforme(N)**

MitadIndices <- SubconjuntoIndices$(0, N)$
for $i \in [0, .., N]$ **do**
  if $i \in$ MitadIndices **then**
    GenesNuevoIndividuo$[i]$ <- GenesProgenitor1$[i]$
  else
    GenesNuevoIndividuo$[i]$ <- GenesProgenitor2$[i]$
  end if
end for

4.2. Algortimos meméticos

En esta sección estudiaremos los algoritmos meméticos tomando como partida los algoritmos genéticos, viendo las mejoras que estos presentan frente a los anteriores al igual que sus posibles desventajas. Los algoritmos meméticos toman su nombre del concepto de meme acuñado por el biólogo Richard Dawkins en su novela El gen egoísta como un concepto equivalente al gen en un ámbito cultural en lugar de biológico. De la misma forma que los genes serían la unidad evolutiva fundamental, los memes serían la unidad cultural fundamental que serían ideas que se pasasen de unos individuos a otros. La diferencia radicaría en que mientras los genes sólo estan expuestos a un proceso de selección evolutiva mediante la supervivencia del más fuerte, un meme también estaría sometido a la evaluación consciente del individuo que lo conoce, es decir, que un individuo puede refinar una idea que tenga pensando en ella. Por establecer un paralelismo evolutivo esto sería como una evolución lamarckista donde los individuos pueden cambiar los genes que les pasan a sus hijos.

Los algoritmos meméticos se diferencian de los genéticos en que antes de añadir un nuevo individuo a la población se le aplica un algoritmo de búsqueda local y el individuo con mejor adaptabilidad que obtenga ese algoritmo será el que se añada a la nueva generación. Por motivos de eficiencia no siempre se aplica la búsqueda local a todos los nuevos individuos sino sólo a un porcentaje de ellos. Otra alternativa común es aplicar la búsqueda local a

toda la población cada varias generaciones, permitiendo así dedicar más tiempo a la parte genética que a la optimización local.

La principal motivación detrás de los algoritmos meméticos radica en aumentar la velocidad con la que el algoritmo genético es capaz de alcanzar mínimos locales, permitiendo que si un individuo se encuentra cerca de un máximo la búsqueda local proporcione dicho máximo. Esto es algo que un algoritmo genético no es capaz de hacer y si deseamos que alcance exactamente el máximo por muy cerca que esté puede tardar muchas generaciones en encontrar la combinación exacta de genes que dan el máximo. Recordemos por otro lado que estos son algoritmos estocásticos y que existen muchos posibles casos de uso donde no se requiera la solución exacta sino que se pueda tolerar un margen de error. En los problemas NP-completos que hemos estudiado no es el caso, pues sólo la solución correcta nos permite decodificar los mensages.

El algoritmo de búsqueda local que se use puede depender del espacio de soluciones que se consideren e incluso se puede sustituir por cualquier otra técnica de optimización local que se considere apropiada. En nuestro caso aplicaremos un algoritmo de búsqueda local simple que consistirá en cambiar un número L de genes y comprobar si esto aumenta la adaptabilidad del individuo. Para simplificar el algoritmo este proceso lo haremos de forma determinista, es decir, probaremos todas las posibles combinaciones de L genes para determinar el mejor individuo. Este proceso se podría hacer de forma estocástica realizando cambios a genes hasta que algún cambio obtuviese a un individuo mejor o se alcanzase un número máximo de intentos. El motivo por el que no es necesariamente tan costoso el buscar todas las posibles combinaciones de cambios es que en las codificaciones de los problemas todos los genes terminan siendo variables binarias, por tanto, a cada gen sólo se le puede aplicar un cambio que es sumarle una unidad.

Algorithm 4: **Búsqueda local simple**

```
1: MejorIndividuo <- IndividuoActual
2: Combinaciones <- Todas las combinaciones de L elementos de un total de N
3: for combinación ∈ Combinaciones do
4:    NuevoIndividuo <- IndividuoActual
5:    for i ∈ combinación do
6:       GenesNuevoIndividuo[i] <- Cambiar(GenesNuevoIndividuo[i])
7:    end for
8:    if Adaptabilidad(NuevoIndividuo) >Adaptabilidad(MejorIndividuo) then
9:       MejorIndividuo <- NuevoIndividuo
10:   end if
11: end for
```

4.3. Aplicaciones al problema de la suma de subconjuntos

En esta sección nos centraremos en como se pueden aplicar los algoritmos genéticos al problema de la suma de subconjuntos y presentaremos los resultados que hemos obtenido.

Supongamos que tenemos un criptosistema basado en la suma de subconjuntos como de-

tallado anteriormente. Como vimos dado que conozcamos un mensaje cifrado y la clave pública del receptor podremos conocer el contenido de dicho mensaje si somos capaces de resolver un problema de la suma de subconjuntos. Para ello usaremos algoritmos genéticos, pero primero debemos de identificar los componentes del mismo.

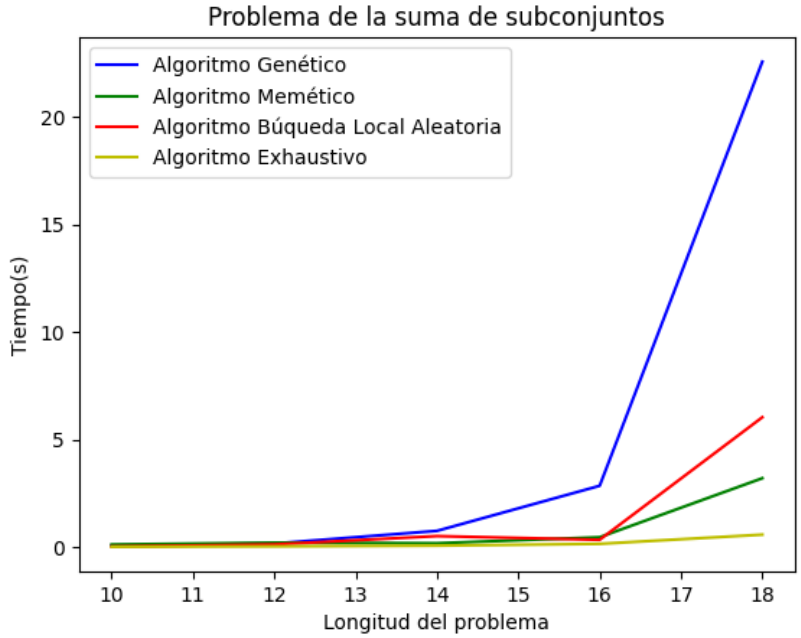
Sea V el vector de enteros positivos que es la clave pública del criptosistema y que define nuestro problema de suma de subconjuntos. Sea W la suma objetivo a obtener por valores de V y sea S el vector binario de igual longitud de V que es la solución a este problema. Tenemos que cada individuo de nuestra población serán vectores binarios de igual longitud que V y que intentarán alcanzar el valor de S. Cada posición de esos vectores se considerará como un gen y serán los valores que se iran cruzando y mutando. De esta manera el número de genes queda determinado por el tamaño de V y el gen n-ésimo representa si se incluye o no en la suma al elemento n-ésimo de V, haciendo así que todos los genes sean valores binarios. Además, así podemos referirnos al tamaño del problema como la longitud que tenga V, pues será esto lo que determine principalmente su dificultad de resolución.

Por último nuestra función objetivo será la distancia de W al producto escalar de un individuo con V. Es fácil comprobar que la solución que buscamos S es el mínimo de esta función, pues es una distancia y para S vale 0, por tanto, queremos minimizar dicha función. En los algoritmos que se han mostrado antes se interpretaba como que se quería maximizar la adaptabilidad, así que podemos considerar nuestra función de adaptabilidad como menos la distacia que nos proporciona nuestra función objetivo. En general cuando se suele referir a una función objetivo que se desea minimizar se la denomina función de perdida, la cual en este caso sería dicha distancia.

Para comprobar las capacidades de los algoritmos genéticos en este problema primero hemos hecho una selección de hiperparámetros para escoger los valores que obtienen mejores resultados. Hemos concluido que una población de 1000 individuos y una probabilidad de mutación de 0.1 daban resultados mejores que otros valores cercanos, con lo cual son estos los valores que se han usado en las pruebas.

Además de los algoritmos genéticos y meméticos hemos utilizado otros algoritmos para permitirnos establecer una comparación de los resultados que deberíamos esperar. El primero de ellos es una búsqueda local aleatoria que genera aleatoriamente una posible solución y le aplica el algoritmo de búsqueda local que usa el algoritmo memético y sino halla la solución genera otra solución aleatoria y repite el proceso.

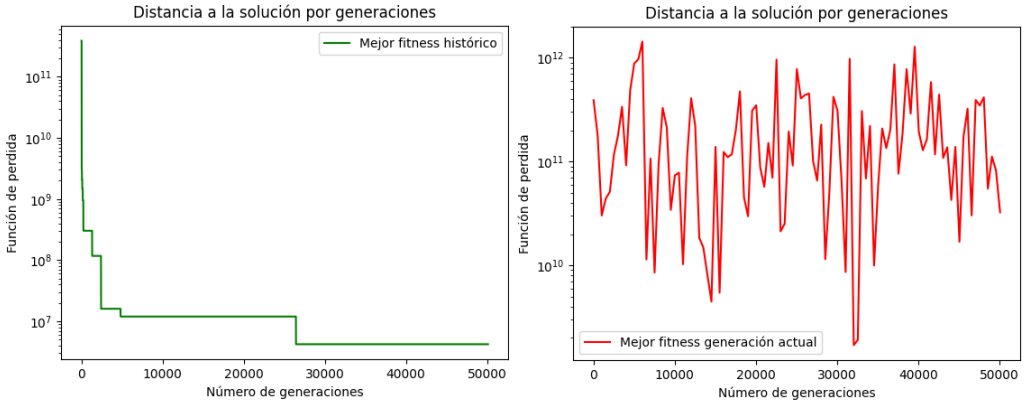
El segundo de estos algoritmos es una búsqueda exhaustiva que va comprobando todas las soluciones en orden y descarta los conjuntos de soluciones que no pueden ser correctas en base a que si una solución supera la suma objetivo ninguna solución que se obtenga aumentando el número de unos de esta pueden ser correctas. Presentamos ahora una gráfica que compara el tiempo en segundos que tarda cada algoritmo en resolver de media un problema según la longitud que tenga el vector V.



Observando los datos vemos que el algoritmo memético consigue ser considerablemente mejor que el algoritmo genético sólamente, lo cual no es necesariamente algo que siempre vaya a ocurrir, pero en este caso le debe de estar ayudando la capacidad de la búsqueda local para alcanzar los mínimos locales directamente sin tener que oscilar hacia ellos durante varias generaciones como en el algoritmo genético. Vemos también que supera al algoritmo de búsqueda aleatoria, lo cual es esperable, pues significa que su búsqueda guiada por el algoritmo genético mejora su eficiencia frente a la búsqueda aleatoria. Por otro lado, la búsqueda exhaustiva parece ser el que menos tarda en estos casos, pero eso se debe a que la longitud del problema es muy pequeña.

Podemos comprobar que los algoritmos genéticos obtienen los peores resultados de todos los métodos. Uno de los motivo para esto puede ser que les estamos exigiendo que encuentren exactamente la solución correcta. Si limitamos la tarea a obtener una solución cercana admitiendo un pequeño margen de error puede que obtuviésemos en pocas generaciones una buena aproximación, alcanzando mínimos locales que se encuentren suficientemente cerca del valor objetivo. Sin embargo, para este criptosistema determinado las soluciones cercanas no nos proporcionan ninguna información útil a la hora de intentar decodificar un mensage.

Para comprobar esta idea, centrémonos ahora sólo en los algoritmos genéticos y veamos cuanto son capaces de acercarse a la solución de un problema de mayor tamaño. En este caso utilizaremos un problema de longitud 50 y mostraremos como disminuye la distancia a la solución correcta. Mostraremos respecto a la mejor solución que ha encontrado el algoritmo hasta ese punto y también respecto a la mejor solución de cada generación. Al ser un problema de tamaño tan grande que no esperamos que sea capaz de resolverlo, estableceremos un máximo de generaciones, en este caso 50000.

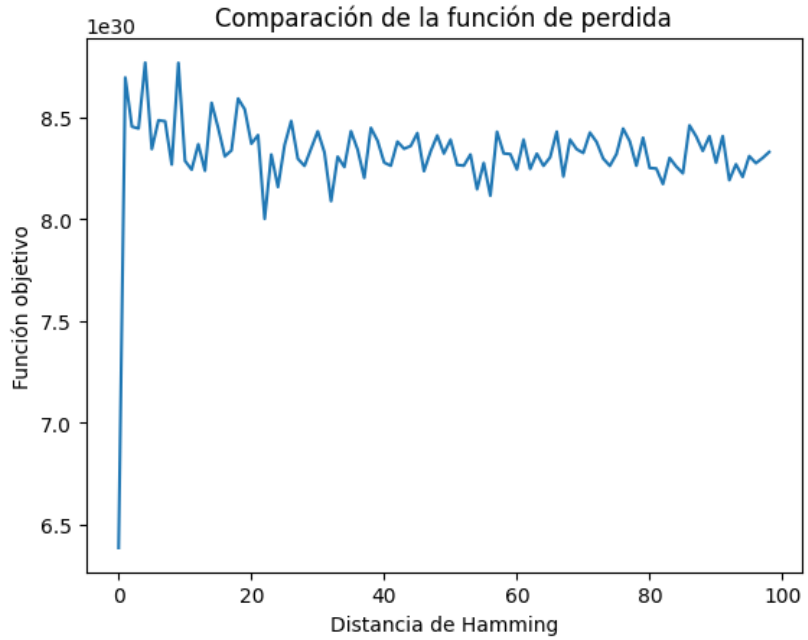


Test algoritmo genético con problema de longitud 50

Podemos comprobar que si bien es capaz de reducir notablemente mucho la distancia en las primeras generaciones, después tiende a costarle encontrar nuevos mínimos. Es más, si nos fijamos en la gráfica de mejor fitness por generación, observaremos que durante muchas generaciones la mejor solución era notablemente peor que el máximo histórico del algoritmo. Alrededor de las últimas 15000 generaciones, la mejor solución de cada generación es notablemente peor que la mejor que ha encontrado el algoritmo. Incluso parece que tiende a aumentar la distancia a la solución alcanzando en una de esas generaciones uno de los máximos de todo el proceso. Es cierto que no necesariamente deberíamos esperar que cada generación fuera mejor que la anterior. Sin embargo, esperaríamos una cierta tendencia a la baja conforme el algoritmo busca crea mejores individuos.

En conjunto, vemos que el algoritmo exhaustivo es el que mejor resultados obtiene lo cual es notablemente desesperanzador, pues es un algoritmo que busca todas las posibles soluciones y, por tanto, no será en ningún caso capaz de mantenerse útil conforme aumentemos el tamaño del problema. Como sabemos el número de soluciones crece exponencialmente y en un problema de longitud 50 la búsqueda exhaustiva tendría que buscar las 2^{50} posibles configuraciones. Esto implica que el resto de algoritmos nos están dando resultados peores que buscar todas las posibles soluciones, siendo entonces totalmente inadecuados para esta tarea.

Uno de los motivos plausibles para los resultados de estos algoritmos podría residir en nuestra función objetivo en cuestión. La función de perdida que estamos considerando, la distancia del producto escalar de nuestras soluciones al valor objetivo W, es una función que sabemos que es mínima cuando alcanzamos la solución real, pero que le ocurre con las demás soluciones. Intuitivamente deberíamos esperar que cuanto más cerca esté una solución de la solución real, o lo que es lo mismo pues son vectores binarios, cuanto menor sea la distancia de Hamming de una solución a la correcta menor debería de ser el valor de su función de perdida. Para que nuestra función de perdida nos sea útil para algoritmos genéticos debería existir una correlación lineal entre la distancia de Hamming y la función de perdida, pues de esta forma cuando el algoritmo genético intente minimizar la función de perdida tenderá al valor correcto de la solución y no a algún mínimo local. Por este motivo presentamos una gráfica comparativa de la media del valor de la función de perdida de un conjunto aleatorio de soluciones a una determinada distancia Hamming de la solución real.



Se puede observar claramente que aunque la función de perdida se anula cuando la distancia es cero en el resto de casos tienden a ser practicamente incorrelada con respecto a la distancia de Hamming, oscilando alrededor de un valor con un coeficiente de correlación lineal cercano a 0. Esto nos muestra que el problema está muy mal acondicionado para estos métodos y para poder utilizar de forma efectiva algoritmos genéticos necesitaríamos una función de perdida mejor condicionada. Sin embargo, la simplicidad del problema no permite de forma fácil plantear ninguna otra fucnión de perdida, pues el valor de la suma objetivo es toda la información que disponemos para conocer como de cerca nos encontramos frente a la solución.

Planteémonos ahora a que se debe el comportamiento de esta función de perdida. La forma en que se creaban los valores de V se basaba en elevar todos los valores a un número y después aplicar módulo respecto a otro, este tipo de operciiones tienen una ventaja de la que nos aprovechamos para este criptosistema y es que generan un conjunto de valores bastante homogéneo, en el sentido de la mayoría de valores de V tendrán un número similar de dígitos, además al ser el número de dígitos notablemente alto sumar varios de ellos nos devolverá valores mayores, pero relativamente bastante cercanos a los valores originales. La homogeneidad de estos valores explica que los valores de la función de perdida parezcan oscilar alrededor de un valor del orden de 10^{30}. Por otro lado aparece un patrón debido a la paridad de la distancia de Hamming que hace que la función objetivo oscile de forma muy rápida. Cada vez que la distancia de Hamming aumenta en uno hemos añadido o eliminado un valor de V de la suma, pero cuando aumenta la distancia en dos puede darse el caso de que hayamos añadido un valor y quitado otro, si la diferencia entre ambos no es muy significativa el efecto en el valor de la función de perdida será menor que cuando sólo se añade o quita uno. De esta manera la función de perdida es ligeramente menor cuando la distancia de Hamming es par que cuando es impar.

En conclusión podemos entender la baja correlación de nuestra función de perdida con la distancia de Hamming a la solución real como una prueba de la homogeneidad de los valores de V que se obtienen utilizando este criptosistema. Para poder atacar este criptosistema se debe resolver el problema de la suma de subconjuntos de forma directa, que es NP-completo, o intentar deshacer la transformación que se le aplica a este problema para enmascararlo y recuperar el problema fácil original, sin conocer la clave privada. Sin embargo, la homogeneidad de los valores de V nos permiten suponer que dicha tarea es a su vez notablemente difícil de lograr, aún cuando no poseamos una prueba teórica de ello.

4.4. Aplicaciones al problema de decodificación de un código lineal

En esta sección nos centraremos en el problema de decodificación lineal, aplicaremos algoritmos genéticos a este problema y presentaremos los resultados.

En el criptosistema basado en teoría de códigos creábamos una matriz lineal C que se convertía en la clave pública y el mensaje codificado correspondía a un vector m' que estaba a una distancia t, siendo este el número de errores que se le añadían, del vector mC donde m era el mensaje original que deseamos recuperar. De nuevo ser capaces de obtener el mensaje original sin conocer la clave privada se reduce a resolver un problema de decodificación de un código lineal que es NP-completo.

Dados entonces C la matriz lineal, t el número de errores y m' veamos como podemos diseñar un algoritmo genético para este problema. La forma directa de enfrentar este problema es dado que la matriz sea de tamaño $n \times k$ considerar el conjunto de vectores binarios de tamaño n como el espacio de soluciones. Para cada posible solución s podemos calcular sC y calcular su distancia de Hamming a m', como sabemos que la solución correcta es la que menor distancia se encuentra de m' y además que esa distancia es t, podemos utilizar esta distancia como función objetivo. Sabemos entonces que nuestra función objetivo tiene un mínimo global únicamente para m que es la solución que estamos buscando, así que esta será la función de perdida que intentaremos minimizar con nuestros algoritmos genéticos. Como estamos considerando el espacio de soluciones los vectores binarios de tamaño n, consideraremos cada posición de ese vector un gen y cada gen será un valor binario que indicará si esa posición es 0 o 1. Este es el método directo de enfrentar el problema, pero vamos a plantear una segunda forma de aproximar el problema.

La matriz C es de rango n tal que se puede calcular una inversa suya por la derecha D tal que D sea matriz $k \times n$ y CD = I donde I representa la identidad de orden n. Esto nos permite que dado un mensaje s codificado por C como c = sC, podemos obtener s multiplicando por D tal que cD = sCD = s. Es importante remarcar que D no es una inversa de C por la derecha tal que si escogemos un vector binario aleatorio c de orden k no siempre es cierto que c = cDC, de hecho esto sólo será cierto si c pertenece al subespacio de dimensión n generado por C. Lo importante es que si encontramos el vector mC, con m el mensaje que buscamos, podremos obtener m aplicando D. Como sabemos que m' está a distancia de Hamming t de mC, sólo tendremos que buscar las posibles soluciones en el espacio de vectores de dimensión k a distancia de Hamming t de m'. Suponemos siempre que $k > n$, por tanto, el espacio de soluciones en que buscamos ahora será de tamaño $\binom{k}{t}$ posibilidades, mientras que antes buscábamos en todos los vectores binarios de dimensión n, es decir, 2^n

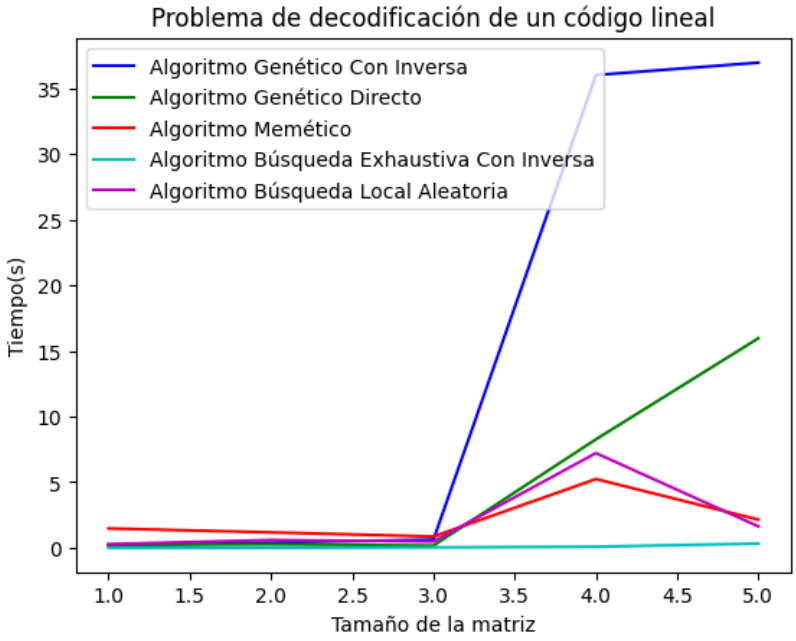
posibilidades.

A priori esto sólo reduce la dimensión del problema si t es un valor pequeño, pero en general no podemos saber cual de los dos espacios de posibilidades es menor. Sin embargo, la matriz D es de orden n, por tanto sólo tiene n filas significativas y el resto serán combinaciones lineales de las anteriores. Esto implica que existe una matriz D inversa por la derecha de C tal que tenga únicamente n filas no nulas. De esta manera al mutiplicar cualquier vector de tamaño k por D sólo importarán las n posiciones del vector correspondientes a las filas no nulas de D. Ahora bien, antes cuando buscabamos los posibles vectores a distancia t de m' esto se puede realizar sumandole a m' todos los posibles vectores de peso de Hamming exactamente t, que son $\binom{k}{t}$ vectores. Ahora podemos sólo tener en cuenta las posiciones de esos vectores que correspondan con las filas no nulas de D y considerar nulas todas las demás, pero esto implica que ya no podemos asegurar que su peso de Hamming sea de t, sino que sólo sabemos que su peso de Hamming será menor o igual que t. De esta manera el espacio de soluciones sería de tamaño $\sum_{i=0}^{t}\binom{n}{i}$ que sí podemos asegurar que es siempre menor o igual que 2^n, pues $2^n = \sum_{i=0}^{n}\binom{n}{i}$ si entendemos todos los posibles vectores de tamaño n como la suma de todos los posibles vectores de tamaño n con peso de Hamming igual a i, llendo i desde 0 hasta n. Este segundo método lo denominaremos como método con la inversa y para cualquier posible solución lo multiplicaremos por DC, tal que si el resultado está a distancia t de m' habrémos encontrado la solución correcta, esta función es la que usaremos como función de perdida, sabiendo que su único mínimo es t para la solución real.

Para valores realistas de este criptosistema es cierto que el valor de t será considerablemente inferior que el valor de n, siendo n=1000 y t=50 posibles valores para un uso real. Esto implica que el método con la inversa trabaja en un espacio de soluciones notablemente menor que el directo, siendo este aún así un espacio que crece más rápidamente que un crecimiento polinomial al aumentar n, pues sino rompería el problema P=NP. Por otro lado, el método de la inversa presenta un notable problema frente al método directo que es que la forma en que representamos las soluciones, como vectores a distancia de Hamming t de m', dificulta aún más evaluar el efecto que está teniendo nuestra función de perdida sobre las soluciones. Ya veíamos en el problema anterior como el hecho de que la función de perdida no estuviera bien acondicionada para nuestro enfoque resultaba en que los algoritmos genéticos no dieran los mejores resultados. En este caso la complicada forma en que manipulamos el espacio de soluciones para reducirlo dimensionalmente obscurece el efecto de la función de perdida. Lo cual mostraremos después al estudiar la función de perdida de ambos métodos.

Para estudiar la eficiencia de los distintos algoritmos hemos escogido una serie de matrices goppa de tamaños ascendentes y hemos creado sus correspondientes criptosistemas con valores de t también ascendentes, remarcar que las matrices aparecen sólo enumeradas, pues los tamaños de las matrices Goppa crecen con potencias de 2 y no permitirían un eje lineal. En este caso como tenemos dos métodos distintos, realizaremos un algoritmo genético y otro memético junto con un algoritmo de búsqueda local aleatoria con el método directo y crearemos un algoritmo genético y un algoritmo de búsqueda exhaustiva con el método con la inversa. No usaremos algoritmos meméticos con el método de la inversa, pues la búsqueda local no se adapta bien a la forma que adquiere el espacio de soluciones en ese caso y no proporciona buenos resultados. Tras probar con distintos valores de los hiperparámetros he-

mos decidido realizar los test utilizando poblaciones de 1000 individuos y una probabilidad de mutación de 0.1.

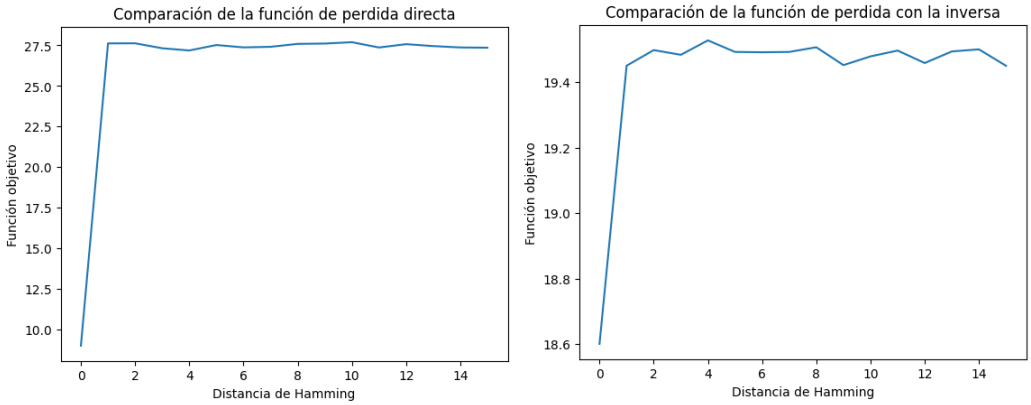


Primero es interesante remarcar que como pequeña optimización a los métodos directos se ha inicializado la población de los mismos como vectores de tamaño k a distancia de Hamming t de m' y después se les ha aplicado la inversa para que fueran vectores de tamaño n. Esta pequeña mejora en la selección de la población inicial mejora significativamente la velocidad de convergencia de los mismos. Si comparamos los algoritmos directos vemos que el algoritmo memético consigue en este caso una mejora significativa respecto al genético aunque no aporta una mejora muy significativa respecto a directamente la búsqueda local aleatoria, lo cual nos hace pensar que la parte genética no está ayudando mucho.

Por otro lado, si observamos los algoritmos que usan la inversa vemos que el algoritmo exhaustivo es el que mejor lo hace, lo cual en este caso se debe simplemente a que es el que trabaja en un espacio de soluciones notablemente menor, pero para matrices todavía más grandes no será capaz de obtener resultados pues tiene que recorrer todo el espacio de soluciones. Sin embargo, el algoritmo genético que está utilizando la inversa termina siendo el que más tiempo tarda en resolver el problema lo cual contrasta fuertemente con el algoritmo exhaustivo. Además el algoritmo genético con la inversa trabaja en un espacio de soluciones mucho menor que el genético directo, pero obtiene resultados notablemente peores. El principal motivo al que podemos achacar estos resultados es que la forma en que representamos los datos para poder aprovechar la inversa no es adecuada para funcionar con algoritmos genéticos y es notablemente peor que el enfoque directo.

Estudiemos ahora como se comportan nuestras funciones de perdida. Al igual que en el problema anterior compararemos la distancia de Hamming de un vector con respecto a la solución correcta con su valor de la función de perdida. En este caso como estamos utilizando dos métodos distintos, mostraremos ambas funciones de perdida, teniendo en cuenta que en la inversa presentamos vectores de tamaño k a los cuales vamos aumentando su distancia de

Hamming con el vector m' y la función de perdida en ese caso será primero multiplicarlos por DC y después calcular su distancia de Hamming con m'.



Los resultados en ambos casos son muy similares, donde al igual que en el problema anterior, vemos que parece existir en ambos casos una gran incorrelación entre la distancia de Hamming y el valor de nuestra función objetivo. Como las dos funciones se calculan de forma distinta no nos aporta ninguna información los valores exactos que alcanza cada función de perdida, lo relevante es la tendencia de cada una. Parece que mediante el uso de la inversa existe una mayor variación en los valores que toma la función, pero sigue sin tener una correlación positiva con la distancia de Hamming que es lo que nos interesaría. Ambas funciones de perdida acaban calculando la distancia respecto a m', por tanto, el hecho de que ambas terminen oscilando alrededor de un mismo valor independientemente de la distancia inicial a la solución correcta nos da a entender que la matriz C distribuye de forma muy homogénea los vectores en el espacio de dimensión k y que es difícil predecir a partir de la misma la distancia de Hamming respecto al vector que buscamos sin realizar el producto matricial. De forma muy similar a la del problema anterior podemos interpretar estos datos como una muestra de que la matriz C que creamos partiendo de una matriz Goppa es una matriz que aparenta ser totalmente genérica y homogénea si se desconoce la clave privada.

Como conclusión añadir que todos los experimentos que se han realizado para la realización de este trabajo han sido con problemas con valores relativamente pequeños para sus parámetros para que se pudieran realizar todos los experimentos de forma más rápida. Los valores que se recomiendan utilizar en criptosistemas que utilicen estos métodos son muy superiores a los usados y además estos criptosistemas se consideran todavía seguros, por tanto, no se cree que para esos valores se pueda encontrar solución a los problemas en tiempos humanamente razonables. Finalmente todos los experimentos han sido programandos todo en python y calculados usando la herramienta gratuita de Google Colab para su cálculo en máquinas virtuales en servidores de Google, asegurando esta que todos se han realizado en una máquina con iguales especifícaciones. Todo el código se deja público para que lo pueda revisar cualquier persona en este Link.

Bibliografía

[Cook(1971)] Cook, S.A. 1971. "The complexity of theorem-proving procedures." In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*.

[Goppa(1970)] Goppa, V.D. 1970. "A New Class of Linear Correcting Codes." In *Problemy Peredachi Informatsii*.

[Grover(1996)] Grover, L.K. 1996. "A fast quantum mechanical algorithm for database search." In *Symposium on the Theory of Computing*.

[McEliece(1978)] McEliece, R.J. 1978. "A Public-Key Cryptosystem Based On Algebraic Coding Theory." In *Deep Space Network Progress Report*.

[Merkle and Hellman(1978)] Merkle, R., and M. Hellman. 1978. "Hiding information and signatures in trapdoor knapsacks." In *IEEE Transactions on Information Theory*.

[Odlyzko(1998)] Odlyzko, A.M. 1998. "The Rise and Fall of Knapsack Cryptosystems."

[Rivest, Shamir, and Adleman(1978)] Rivest, R.L., A. Shamir, and L. Adleman. 1978. "A method for obtaining digital signatures and public-key cryptosystems." In *Association for Computing Machinery*. New York, NY, USA.

[Shamir(1984)] Shamir, A. 1984. "A polynomial-time algorithm for breaking the basic Merkle - Hellman cryptosystem." In *IEEE Transactions on Information Theory*.

[Vardy(1997)] Vardy, A. 1997. "The intractability of computing the minimum distance of a code." In *IEEE Transactions on Information Theory*.

[Varshamov(1957)] Varshamov, R. 1957. "Estimate of the number of signals in error correcting codes." In *Dokl. Akad. Nauk SSSR*.

[Xanda Schoefield(2018)] Xanda Schoefield, E.T. 2018. *Subset Sum is NP-complete*, introduction to algorithms ed. Cornell University, Lecture Notes.

[Yao(2002)] Yao, A. 2002. "Classical Physics and the Church-Turing Thesis." In *Electronic Colloquium on Computational Complexity (ECCC)*.