UNIVERSIDAD DE GRANADA

E.T.S. de Ingenierías Informática y de Telecomunicación y Facultad de Ciencias

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Teoría de Grafos y Criptografía

Presentado por:
Carlota Valdivia Manzano

Tutores:
Jesús García Miranda
Departamento de Álgebra, E.T.S. de Ingenierías Informática y de Telecomunicación

Pedro A. García Sánchez
Departamento de Álgebra, Facultad de Ciencias

Curso académico 2022-2023

Teoría de Grafos y Criptografía

Carlota Valdivia Manzano

Carlota Valdivia Manzano *Teoría de Grafos y Criptografía*.
Trabajo de fin de Grado. Curso académico 2022-2023.

Responsable de tutorización

Jesús García Miranda
Departamento de Álgebra, E.T.S. de Ingenierías Informática y de Telecomunicación

Pedro A. García Sánchez
Departamento de Álgebra, Facultad de Ciencias

Doble Grado en Ingeniería Informática y Matemáticas

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación y Facultad de Ciencias

Universidad de Granada

Declaración de originalidad

Dña. Carlota Valdivia Manzano

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2022-2023, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 21 de noviembre de 2022

Fdo: Carlota Valdivia Manzano

Índice general

Breve resumen

Este trabajo presenta una introducción a la Teoría de Grafos y tres distintas aplicaciones en Criptografía. Cada una de estas aplicaciones hace uso de diferentes nociones de grafos.

En primer lugar, se formaliza el concepto de grafo y se explican sus propiedades y elementos más importantes. Toda esta teoría servirá de base para presentar las diversas aplicaciones de grafos en la criptografía.

En particular, el desarrollo de esta teoría nos ha permitido estudiar e implementar un algoritmo de clave simétrica para cifrar y descifrar datos de forma segura. Los principales conceptos que han sido empleados son los de grafo completo, ciclo y árbol generador minimal.

Del mismo modo, el estudio de grafos nos ha permitido presentar dos construcciones de funciones hash criptográficas distintas. Ambas construcciones se realizan a partir de dos tipos de grafos especiales: los grafos de expansión y los grafos de Cayley. Estos nos permiten relacionar las propiedades más importantes de las funciones hash criptográficas con propiedades de los grafos.

Por último, esta misma teoría nos permite examinar un protocolo de conocimiento cero basado en el problema de isomorfismo de grafos, debatiendo si este problema es adecuado para este tipo de protocolos.

Palabras clave: teoría de grafos, criptografía, protocolos criptográficos, funciones hash criptográficas, grafos de expansión, grafos de Cayley, protocolo de conocimiento cero, problema de isomorfismo de grafos.

Extended summary

This document presents an introduction to Graph Theory and three different applications in Cryptography. Each application uses different graph notions to obtain different cryptographic objectives.

Therefore, the two main areas addressed in the work are Graph Theory and Cryptography. The first chapter has a more mathematical and theoretical point of view, since it explains an introduction to graphs, while the other three remaining chapters present how Graph Theory can be used to develop cryptographic protocols. In particular, Chapter 2 studies an encryption algorithm based on graphs, and provides an implementation of the algorithm. In chapter 3, we proceed to explain two constructions of hash functions from expansion and Cayley graphs. Finally, Chapter 4 presents a zero knowledge protocol based on the graph isomorphism problem.

Next, we will explain in more detail the development of each chapter.

Chapter 1: This chapter presents an introduction to Graph Theory. A graph is a structure composed of a set of vertices, a set of edges and an incidence function that relates each edge to a pair of vertices.

Graphs have an order and a size. These represent the number of vertices and edges they have, respectively. On the other hand, each vertex has a degree, which corresponds to the number of edges that are attached to it.

Walks are a succession of vertices and edges. A walk is said to be closed if its ending vertices coincide, otherwise it is said to be open. If a walk has no repeated edges, it is called a trail, and if there are no repeated vertices, except for the starting and ending ones, it is called a path. A path that is closed is a circuit, and a closed simple path is called a cycle.

Graphs can be represented in many different ways, in particular they can be represented as matrices. The adjacency matrix of a graph is a matrix whose entries are identified by the number of edges that join the two vertices whose indices correspond to the row and column of the matrix. The incidence matrix is a matrix whose entries are one if the edge whose index corresponds to the column number is attached to the vertex whose index corresponds to the row number. Otherwise, the entry value is zero.

The same graph can be represented in many ways, because its vertices and edges can be renamed with different labels. This gives rise to the problem of isomorphism of graphs. An isomorphism is a morphism whose pair of functions are bijective. An automorphism of a graph is an isomorphism of the graph itself. The graph isomorphism problem is a problem that, in the worst case, no polynomial-time algorithms are known to solve it. This problem consists, in its general version, in proving that given two graphs there is an isomorphism between them.

There are many different types of graphs depending on the properties they verify. For example, complete graphs are those whose vertices are connected to all other vertices; connected

graphs those where each pair of vertices is connected by a walk; and weighted graphs have a function that assigns a weight to each of their edges.

Throughout the first chapter, some more special graphs are studied in depth, such as trees, expander graphs and Cayley graphs.

Trees are connected graphs that do not have cycles. Given a connected graph, a subgraph of it that is a tree and whose vertices sets coincide is called a spanning tree. Based on these concepts, we focus on the study of minimun spanning trees, which are spanning trees where the sum of their edges' weight is the smallest possible. There are different algorithms to find such trees. Among the best known are Kruskal's and Prim's algorithms, which are explained in detail in the chapter.

Next, we study the expansion properties of the graphs that give rise to the definition of a family of expander graphs. Given a graph, there is an expanding constant that represents the smallest possible ratio between the number of edges coming out of a subset of vertices and the size of that subset. From it, a family of connected graphs is said to be a family of expander graphs if they verify that the expanding constant is strictly greater than zero, the maximum degree of the vertices of each graph is bounded; and the number of vertices of that family is increasing.

Finally, Cayley graphs are those that arise from the idea of representing the group structure visually. They are associated with a group and a subset of elements of it, usually generators of the group.

Chapter 2: This chapter describes a symmetric-key encryption and decryption algorithm based on Graph Theory. This algorithm draws upon several graph notions such as complete graphs, cycles and minimum spanning trees.

Since it is a symmetric-key algorithm, both parts of the communication must own a shared secret key. This key is established by a random matrix, used to encrypt and decrypt the messages; an initial special character, and a codificacion table to be able to relate characters to numerical values.

The algorithm starts from a secret message that will be transformed into a graph. To encrypt the message, their characters need to be represented as graph vertices, and two adjoining characters portray an edge between their relative vertices. The edges' weight will be assigned using the codification table.

The idea is that once the graph is obtained from the secret message, the graph's adjancency matrix is used to find a minimum spanning tree. Then the minimum spanning tree's adjancency matrix will be multiplied by the key matrix. The result of this product concatenated with the graph's adjacency matrix will be the ciphered message that the transmitter will send. In order to decipher the message, the receiver will calculate the inverse of the key matrix and the graph's adjacency matrix obtaining the minimum spanning tree's adjacency matrix and decode the information from it.

Once the encryption and decryption algorithms are explained in detail, we illustrate its implementation with an example.

A discussion about the algorithm is made. In particular, it is debated whether the algorithm is considered to be efficient and secure, and their main advantages and downsides are stated.

Chapter 3: Cryptographic hash functions are functions that transform a bit message of an arbitrary length into a bit string of fixed length. This output is called a hash value.

Hash functions may or may not depend on a key, but they must meet the following properties: resistance to preimage computation, weak resistance to collisions and strong resistance to collisions. A collision occurs when two different inputs to a hash function produce the same hash value. The last two concepts can be distinguished through the birthday paradox, which states the probability that in a group of people two of them share the same birthday.

Hash functions can be constructed iteratively. The Merkle-Damgard construction is one of the most widely used methods in Cryptography for this purpose.

However, the main constructions proposed in the chapter are from expansion graphs and Cayley graphs. In both cases it is necessary to start from a graph, an initial edge or vertex, and a sorting function to construct the corresponding cryptographic hash function. These functions can relate graph properties to hash functions properties.

Specific attacks on expansion or Cayley hash functions are also presented, such as the automorphism attack, subgroup attack or trapdoor attack.

Chapter 4: This chapter approaches zero knowledge proofs. A zero knowledge proof is a cryptographic protocol that allows demonstrating that certain information is known without having to expose it. Two parties are involved in the process: the prover and the verifier. The prover is the part who knows the secret, and the verifier is the one who is in charge of testing the truthfulness of the prover.

These protocols fulfill three main properties: completeness, soundness and zero knowledge. Completeness means that if both the prover and the verifier are honest and follow the protocol correctly, then the prover must be able to convince the verifier that the statement is true. Soundness is obtained by minimizing the probability of the prover deceiving the verifier. Finally, zero knowledge means that if the statement is true, no misleading verifier can know more than this fact.

Zero knowledge protocols can be classified as interactive or non-interactive. In interactive ones, it is necessary for both the prover and the verifier to be present during the execution of the protocol. In non-interactive ones, it is not necessary for them both to be present during the protocol execution, but the prover can leave a document of the protocol for the verifier to test at any time.

Then, a zero knowledge protocol based on the graph isomorphism problem is presented. This protocol starts from two isomorphic graphs. The prover knows an isomorphism defined between the two graphs, and wants to convince the verifier that he knows it without revealing it.

Keywords: graph theory, cryptography, cryptographic protocols, cryptographic hash functions, expansion graphs, Cayley graphs, zero knowledge proof, graph isomorphism problem.

Introducción

El notable incremento de la tecnología a lo largo de los últimos años nos ha llevado a un continuo intercambio de información a través de redes. La criptografía ha tomado un papel clave en el mantenimiento y desarrollo de protocolos que garanticen la seguridad de dicha información.

Por tanto, es necesario encontrar protocolos y algoritmos criptográficos eficientes que nos aseguren la confidencialidad, integridad y autenticidad de los datos transmitidos. En este trabajo, estudiaremos nuevas alternativas y técnicas basadas en Teoría de Grafos con el propósito aprovecharnos de las propiedades de esta teoría para explicar propiedades criptográficas.

En particular, con el fin de encontrar nuevos algoritmos de cifrado más seguros, estudiaremos el algoritmo propuesto por [Eta14] basado en grafos completos, ciclos y árboles generadores minimales.

Del mismo modo, las vulnerabilidades que se van encontrando en funciones hash criptográficas como en SHA-1, nos hacen buscar nuevas alternativas para construir este tipo de funciones. Los grafos de expansión y los grafos de Cayley ofrecen un diseño simple para dicha construcción y la posibilidad de interpretar las propiedades de las funciones hash criptográficas más importantes como propiedades de grafos.

Por otro lado, ante los problemas de confidencialidad que se pueden presentar en el intercambio de información, se pueden usar protocolos de conocimiento cero. En ellos, un probador puede mostrar que conoce cierta información sin revelársela al verificador. De este modo, al no revelar la información secreta, no podría ser suplantado por el propio verificador ni por otra tercera persona.

Además del estudio del algoritmo de cifrado basado en grafos, se ha realizado una implementación del mismo haciendo uso del lenguaje de programación *python* y de las diversas librerías matemáticas que ofrece. Dicha implementación se ha realizado en un cuaderno de *jupyter*, que es un entorno interactivo que permite ejecutar bloques de código y añadir áreas de texto para documentar el propio código, favoreciendo su comprensión. En dicho cuaderno se encuentra la implementación del algoritmo, varios ejemplos de cifrado y una serie de ejecuciones del algoritmo para evaluar sus tiempos medios.

Todo el código implementado en la elaboración de trabajo se encuentra en el siguiente repositorio de GitHub:

`https://github.com/lmd-ugr/grafos-criptografia`.

Objetivos iniciales y resultados alcanzados

Los objetivos planteados a desarrollar en el trabajo fueron los siguientes.

- Realizar un estudio sobre la Teoría de Grafos, incluyendo las nociones necesarias para realizar las tres aplicaciones distintas en el ámbito criptográfico.

- Analizar un protocolo de cifrado de clave simétrica presentado por [Eta14] basado en Teoría de Grafos y realizar una implementación del mismo.
- Estudiar las funciones hash criptográficas junto a dos posibles construcciones de las mismas a partir de grafos de expansión y grafos de Cayley.
- Comprender los protocolos de conocimiento cero y, en particular, estudiar y analizar el protocolo para el problema de isomorfismo de grafos.

El primer objetivo ha sido realizado en el primer capítulo. Dicho capítulo ha sido más extenso debido a la profundización en distintos tipos de grafos, como los árboles generadores minimales, los grafos de expansión o los de Cayley, y en la noción de isomorfismo de grafos.

El segundo objetivo se ha llevado a cabo en el respectivo segundo capítulo. En él se ha explicado el algoritmo propuesto por [Eta14] y se ha realizado un ejemplo ilustrativo. Posteriomente, se ha descrito la implementación elaborada y se ha realizado un análisis del mismo exponiendo los inconvenientes y limitaciones que presenta.

En el tercer capítulo se ha alcanzado el tercer objetivo. En primer lugar, se han explicado las funciones hash criptográficas, al igual que las propiedades que deben verificar, y se han presentado dos construcciones basadas en grafos además de la de Merkle-Damgård. Dichas construcciones han partido de grafos de expansión y grafos de Cayley, dando a lugar funciones hash de expansión y funciones hash de Cayley. Se han presentado sus propiedades específicas al igual que sus posibles ataques.

Por último, el cuarto objetivo se ha realizado en el cuarto capítulo. En él se han estudiado los protocolos de conocimiento cero, y se ha particularizado en el protocolo para el problema de isomorfismo de grafos.

Las principales fuentes consultadas en el trabajo han sido [GM17, Eta14, Pet09, Kow19, Dur16, Aye09].

1 Introducción a la Teoría de Grafos

1.1. Introducción

La Teoría de Grafos es una rama de la Matemática Discreta que se ha desarrollado a partir de la necesidad de buscar solución a problemas asociados a un número de puntos determinado y a las líneas que los conectan. En otras palabras, la Teoría de Grafos se encarga de estudiar la conectividad entre los elementos de un conjunto.

Las aplicaciones de la Teoría de Grafos se pueden encontrar no solo en otras ramas de las matemáticas, sino también en otras disciplinas científicas como la informática, la física, la química o la biología. Debido al desarrollo de la computación, la Teoría de Grafos se ha convertido en una herramienta de gran importancia en las últimas décadas. Sus aplicaciones varían desde el análisis de redes de sociales, la detección de fraude bancario o el modelo de conexiones de redes de tráfico, hasta el estudio de moléculas, la solución a problemas de genética o la difusión de enfermedades infecciosas.

En el siguiente capítulo se presenta una introducción a la Teoría de Grafos. Por un lado, se pondrá en contexto el problema que dio lugar al origen de dicha teoría, y posteriormente, se introducirán los conceptos fundamentales de la Teoría de Grafos y algunas de sus aplicaciones.

En la elaboración de este capítulo se ha utilizado [GM17, Capítulo 6], [Cha77], [ACMC], [Kow19] y [CLRS22].

1.2. Problema de los Siete Puentes

La Teoría de Grafos surgió a partir de un problema de ámbito turístico que fue resuelto por el matemático Leonhard Euler en 1736. Dicho problema es conocido como el «*Problema de los Siete Puentes*» o el «*Problema de los Puentes de Königsberg*».

Este problema residía en la necesidad de encontrar una ruta a pie por la ciudad de Königsberg, famosa por sus siete puentes que atravesaban el río Pergel, de forma que saliendo desde un punto concreto, cruzase los siete puentes una sola vez cada uno antes de regresar al punto de origen.

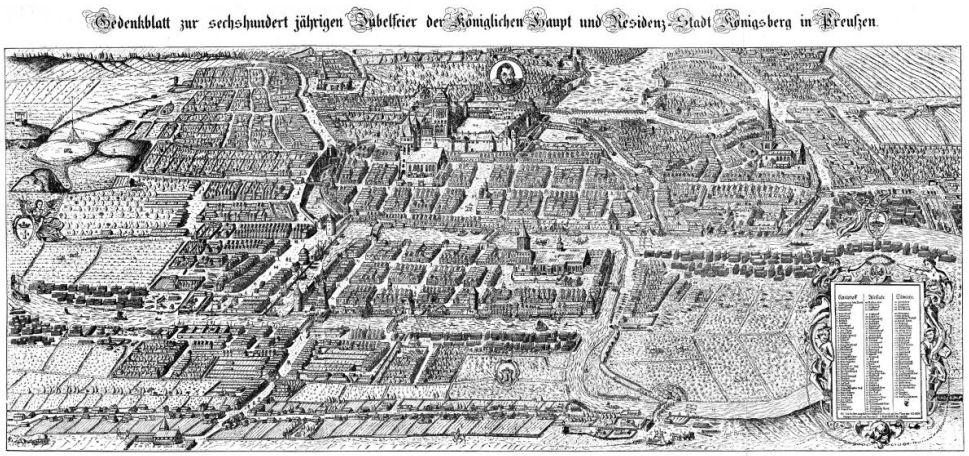


Figura 1.1: Grabado de la ciudad de Königsberg de Joachim Bering en 1613

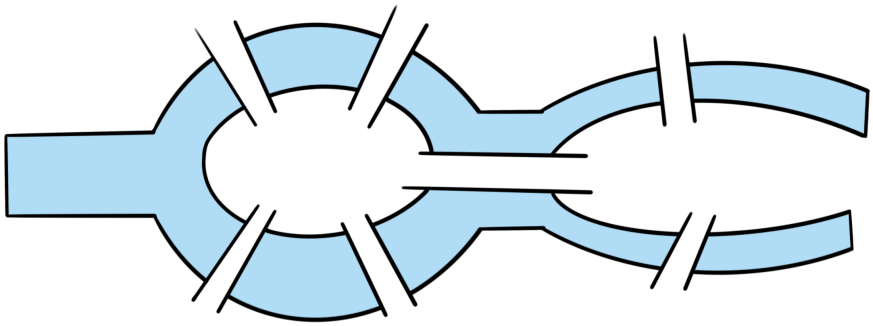


Figura 1.2: Esquema del Problema de los Siete Puentes

Para resolver dicho problema, Euler representa de forma abstracta las cuatro regiones terrestres como cuatro puntos, y los puentes como lados que unen dichos puntos.

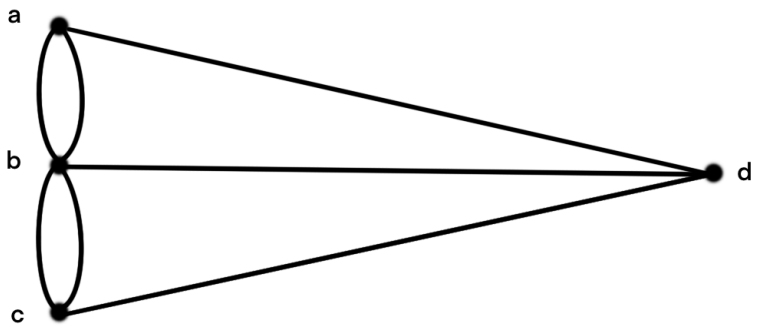


Figura 1.3: Grafo del Problema de los Siete Puentes

Los puntos (a, b, c, d) son los que denominaremos vértices, y los lados que conectan dichos vértices serán las aristas.

Estos dos conceptos son los que dan lugar a la definición de grafo.

1.3. Nociones básicas

Comenzaremos esta sección con definiciones básicas para la construcción de grafos. Después se estudiaran diversas propiedades que verifican, y se presentarán distintos tipos de grafos.

Definición 1.1. Un grafo G es un par (V, E), donde $V \neq \emptyset$ es un conjunto al que llamaremos *conjunto de vértices o nodos* y E es un conjunto al que llamaremos *conjunto de lados o aristas*, junto con una aplicación

$$\gamma_G : E \longrightarrow \{\{u, v\} : u, v \in V\},$$

denominada *aplicación de incidencia*, la cual lleva cada arista del conjunto E al subconjunto de vértices de V en los que incide.

Ejemplo 1.1. En el caso del Problema de los Siete Puentes, el grafo de la Figura 1.3 es el par (V, E) donde el conjunto de vértices es $V = \{a, b, c, d\}$, el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ y la aplicación de incidencia viene definida de la siguiente forma:

$$\gamma_G(e_1) = \gamma_G(e_2) = \{a, b\}, \quad \gamma_G(e_3) = \gamma_G(e_4) = \{b, c\}, \quad \gamma_G(e_5) = \{a, d\},$$

$$\gamma_G(e_6) = \{b, d\}, \quad \gamma_G(e_7) = \{c, d\}.$$

Definición 1.2. Diremos que dos vértices son *adyacentes* si están unidos por una misma arista. También podemos decir que son vértices vecinos o que son *indicentes* a la arista que los une. Por otro lado, diremos que un vértice es *aislado* si no hay ninguna arista que incida sobre él mismo.

Definición 1.3. Diremos que dos aristas son *paralelas* si inciden sobre los mismos vértices. Por otro lado, si tenemos una arista que solamente incide en un único vértice, diremos que dicha arista es un *lazo*.

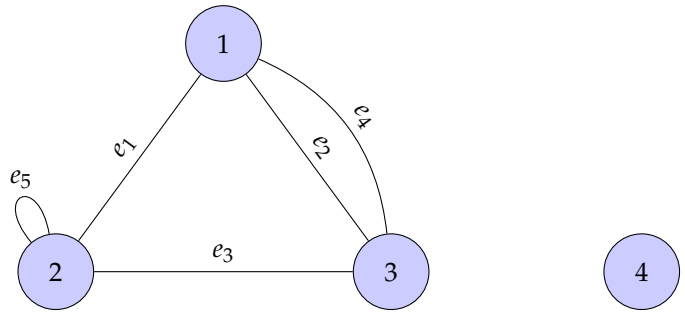


Figura 1.4: Grafo con nodos aislados y adyacentes

Ejemplo 1.2. En la Figura 1.4, el grafo es el par (V, E) donde el conjunto de vértices es $V = \{1, 2, 3, 4\}$, el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4, e_5\}$ y la aplicación de incidencia se define como:

$$\gamma_G(e_1) = \{1, 2\}, \quad \gamma_G(e_2) = \gamma_G(e_4) = \{1, 3\}, \quad \gamma_G(e_3) = \{2, 3\}, \quad \gamma_G(e_5) = \{2\}.$$

Podemos remarcar algunas características referentes a este grafo.

- Por un lado, podemos observar que la arista e_1 une los vértices 1 y 2, por lo que éstos son adyacentes.
- Por otro lado, podemos apreciar que sobre el vértice 4 no incide ninguna arista, por lo que es aislado.
- Asimismo, tenemos dos aristas del grafo, e_1 y e_2, que inciden sobre los mismos vértices, es decir, que $\gamma_G(e_1) = \gamma_G(e_2) = \{a, b\}$. Por tanto, ambas aristas son *paralelas*.
- Además, como tenemos $\gamma_G(e_5) = \{2\}$, es decir, que e_5 solamente incide sobre el vértice 2, por tanto dicha arista es un *lazo*.

Cabe añadir que algunos autores denominan a este tipo de grafos, *multigrafos*, refiriéndose a los *grafos* como aquellos que no poseen ni aristas paralelas ni lazos. No obstante, en este documento se hará referencia a los grafos y multigrafos como *grafos simples* y *grafos*, respectivamente.

Podemos apreciar que al indicar las aristas que inciden sobre vértices no siempre mencionamos cuál es su origen y su fin. Esto se debe a que consideramos, salvo indicación, que las aristas pueden recorrerse en ambos sentidos, es decir, no tienen dirección. De esta forma, podemos clasificar los grafos según su dirección como: *grafos dirigidos u orientados* y *no dirigidos*.

Definición 1.4. Un *grafo dirigido u orientado* es un par (V, E), donde V es el conjunto de vértices y E el conjunto de aristas, junto con dos aplicaciones

$$s, t : E \longrightarrow V$$

denominadas aplicaciones dominio y codominio (en inglés, *source* y *target*). La *aplicación dominio* devuelve el nodo origen de la arista evaluada, mientras que la *aplicación codominio* devuelve el nodo final de la misma.

Diremos que un grafo es orientado o dirigido si sus aristas tienen dirección.

Ejemplo 1.3. Sea G el grafo que se muestra en la Figura 1.5, definido como el par $G = (V, E)$ donde el conjunto de vértices es $V = \{1, 2, 3\}$ y el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4\}$. Las aplicaciones dominio y codominio vendrían definidas:

$$s(e_1) = \{1\}, \quad s(e_2) = \{1\}, \quad s(e_3) = \{2\}, \quad s(e_4) = \{3\},$$

$$t(e_1) = \{2\}, \quad t(e_2) = \{3\}, \quad t(e_3) = \{3\}, \quad t(e_4) = \{1\}.$$

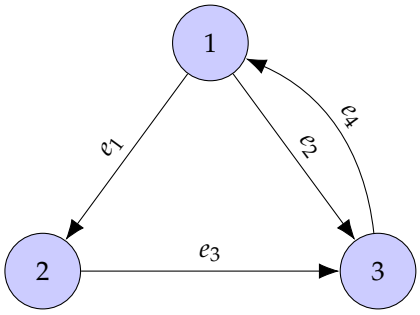


Figura 1.5: Grafo dirigido

Definición 1.5. Sea $G = (V, E)$ un grafo con aplicación de incidencia γ_G. Decimos que $G' = (V', E')$ es un *subgrafo* de G si $V' \subseteq V$, $E' \subseteq E$ y se verifica que $\gamma_{G'}(e) = \gamma_G(e)$ para todo $e \in E'$.

Se dice que $G' = (V', E')$ es un *subgrafo completo* de $G = (V, E)$ si este es un subgrafo suyo y dado $e \in E$ tal que $\gamma_G(e) \subseteq V'$ se cumple que $e \in E'$. En otras palabras, G' es subgrafo completo de G, si es subgrafo suyo y si tiene todos las aristas de G que unen vértices de V'.

Nota 1.1. *Un subgrafo completo está determinado por el conjunto de vértices.*

Definición 1.6. Dado un grafo $G = (V, E)$ con aplicación de incidencia γ_G, consideramos un subconjunto $S \subseteq V$. Se define el *subgrafo generado* por S en G como el grafo $G[S] = (S, E')$ de forma que $E' \subseteq E$ y dado $e \in E$ tal que $\gamma_G(e) \subseteq S$ se verifica que $e \in E'$. El subgrafo generado por S en G es un subgrafo completo.

1.3.1. Propiedades y elementos de los Grafos

A continuación, introduciremos parte de la terminología básica de grafos. Posteriormente, comentaremos algunas propiedades acerca de ellos.

Definición 1.7. El número de vértices que tiene un grafo $G = (V, E)$ se denomina *orden del grafo* y se puede denotar como $|V|$.

Por otro lado, el número de lados que tiene un grafo $G = (V, E)$ se denomina *tamaño del grafo* y se puede denotar como $|E|$.

En la Figura 1.4 el orden del grafo es $|V| = 4$ y el tamaño es $|E| = 5$. En la Figura 1.5 el orden es $|V| = 3$ y el tamaño $|E| = 4$.

Definición 1.8. Sea $G = (V, E)$ un grafo y $v \in v$. Llamaremos *grado de un vértice* v al número de aristas que inciden sobre él, y lo denotaremos por $\mathrm{gr}(v)$.

En grafos dirigidos, podemos diferenciar entre *grado de entrada*, gr^+, y *grado de salida*, gr^-, para hacer alusión al número de aristas que llegan al vértice o salen de él, respectivamente.

En el Ejemplo 1.2 los grados de los distintos vértices son:

$$\mathrm{gr}(1) = \{3\}, \quad \mathrm{gr}(2) = \{4\}, \quad \mathrm{gr}(3) = \{3\}, \quad \mathrm{gr}(4) = \{0\}.$$

En el caso de tener un lazo, su grado aumenta en dos en vez de en uno. Así pues, para el vértice 2 tenemos que su grado es 4.

Definición 1.9. Sea $G = (V, E)$ un grafo donde V es su conjunto de vértices y E su conjunto de aristas. Denotaremos por $D_k(G)$ al número de vértices de V que tienen grado igual a k, con $k \in \mathbb{N}$. De esta forma, podemos definir la *sucesión de grados* de un grafo como la sucesión:

$$D_0(G), D_1(G), \ldots, D_k(G), \ldots$$

En el Ejemplo 1.2 la sucesión de grados del grafo es:

$$1, 0, 0, 2, 1, \ldots$$

Teorema 1.1. *Sea $G = (V, E)$ un grafo. La suma de los grados de los vértices del grafo es el doble del número de aristas del mismo, es decir,*

$$\sum_{i=1}^{n} gr(v_i) = 2|E|,$$

donde $V = \{v_1, \ldots, v_n\}$.

Demostración. Puesto que cada arista incide sobre dos vértices, al sumar el grado de cada vértice, cada arista se ha contado por cada uno de los vértices sobre los que incide, es decir, dos veces. Por tanto, tenemos que la suma de todos los grados de los vértices es igual al doble de número de aristas del grafo. □

Teorema 1.2. *Todo grafo $G = (V, E)$ contiene un número par de vértices de grado impar.*

Demostración. Si suponemos que G no tiene vértices de grado impar, entonces obtenemos directamente que contiene un número par de vértices que lo cumple, en concreto, cero vértices. Por otro lado, supongamos que G contiene k vértices de grado impar, llamémoslos $v_1, v_2, \cdots, v_k$.

Si suponemos que G contiene a su vez vértices de grado par, a los que denotaremos $u_1, u_2, \cdots, u_m$. Por el Teorema 1.1,

$$\sum_{i=1}^{k} gr(v_i) + \sum_{i=1}^{m} gr(u_i) = 2|E|.$$

Como los grados de los vértices $u_1, u_2, \cdots, u_m$ son pares, entonces $\sum_{i=1}^{m} gr(u_i)$ es par también. Por tanto tenemos que,

$$\sum_{i=1}^{k} gr(v_i) = |E| - \sum_{i=1}^{m} gr(u_i),$$

es par. Sin embargo, sabemos por hipótesis que los grados de los vértices $v_1, v_2, \cdots, v_k$ son impares. Por lo tanto, k debe de ser un número par y obtenemos así que G contiene un número par de vértices de grado impar.

Si suponemos que G no contiene vértices de grado par obtenemos,

$$\sum_{i=1}^{k} \mathrm{gr}(\mathrm{v_i}) = 2|E|,$$

por lo que de nuevo k debe ser un número par y hemos concluido la demostración. □

En el Ejemplo 1.2, la suma de los grados de sus vértices es $\sum_{i=1}^{4} \mathrm{gr}(\mathrm{v_i}) = 10$, que es igual al doble del número de aristas que posee el grafo, $2 \cdot |E| = 2 \cdot 5 = 10$. Asimismo, contiene dos vértices de grado impar, los vértices v_1 y v_3 con grados $\mathrm{gr}(\mathrm{v_1}) = \mathrm{gr}(\mathrm{v_3}) = 3$.

Proposición 1.1. *Sea $G = (V, E)$ un grafo y $G' = (V', E')$ un subgrafo suyo, se cumple que*

$$\mathrm{gr}_{G'}(v) \leq \mathrm{gr}_G(v) \quad \textit{para todo } v \in V'.$$

Definición 1.10. Se denomina *camino de longitud n* en un grafo $G = (V, E)$ a una sucesión de vértices $v_1 v_2 \cdots v_n$ y una sucesión de aristas $e_1 e_2 \cdots e_{n-1}$, de forma que $\gamma_G(e_i) = \{v_i, v_{i+1}\}$.

Al camino $e_1 e_2 \cdots e_{n-1}$ lo llamaremos camino del vértice v_1 al vértice v_n.

A los vértices v_1 y v_n los denominaremos *extremos* del camino. En concreto, v_1 y v_n serán el vértice inicial y final del camino, respectivamente.

Ejemplo 1.4. En la Figura 1.3, si tomamos la sucesión de vértices *abc* y la sucesión de aristas $e_1 e_3$, tenemos que $e_1 e_3$ es un camino del vértice a al vértice c.

Diremos que un camino de v a v es de longitud cero a aquel cuya sucesión de vértices es v y cuya sucesión de aristas es vacía.

Si $e_1 e_2 \cdots e_n$ es un camino de u a v, entonces $e_n e_{n-1} \cdots e_2 e_1$, es un camino de v a u.

Definición 1.11. Definimos la *distancia* entre dos vértices como la longitud del menor camino entre ellos. Si no existe un camino uniéndolos, diremos que la distancia es infinita.

En el Ejemplo 1.2 la distancia entre los vértices 1, 2 y 3 dos a dos es 1, puesto que son adyacentes. Por otro lado, la distancia entre el vértice 1 y el 4 es infinita, ya que no hay ningún camino que los una.

Definición 1.12. El *diámetro* de un grafo es la mayor distancia entre cualesquiera dos vértices del grafo.

En el Ejemplo 1.1, el diámetro del grafo G es 2, puesto que todos los vértices b y d son adyacentes al resto, por lo que sus distancias al resto de vértices es 1, y la distancia entre los vértices a y c es 2. Por otro lado, en el Ejemplo 1.2, el diámetro es ∞ porque tenemos un vértice aislado.

Definición 1.13. Diremos que un camino es *cerrado* si sus extremos coinciden, es decir, si comienza y acaba en el mismo vértice. En caso contrario, diremos que un camino es *abierto*.

Ejemplo 1.5. En la Figura 1.3, si tomamos la sucesión de vértices *aba* y la sucesión de aristas e_1e_2 tal que $\gamma_G(e_1) = \{a, b\}$ y $\gamma_G(e_2) = \{a, b\}$, entonces e_1e_2 es un camino cerrado del vértice a al vértice a.

Por otro lado, si tomamos la sucesión de vértices *abc* y la sucesión de aristas e_1e_3 tal que $\gamma_G(e_1) = \{a, b\}$ y $\gamma_G(e_3) = \{b, c\}$, el camino e_1e_3 es un camino abierto del vértice a al c.

Definición 1.14. Se denomina *recorrido* a un camino en el que no se encuentran aristas repetidas.

En el Ejemplo 1.4, hemos tomado el camino de a a c, e_1e_3, el cual no tiene aristas repetidas, por lo que se trata de un recorrido.

Definición 1.15. Llamaremos *camino simple* a aquel recorrido en el no hay vértices repetidos, excepto posiblemente el primero y el último.

En el Ejemplo 1.4, también tenemos un camino simple ya que no se encuentran vértices repetidos.

Definición 1.16. Denominamos *circuito* a todo recorrido que a su vez es un camino cerrado, es decir, un camino en cuál el vértice inicial y final coinciden pero no se repite ninguna arista.

En el Ejemplo 1.5, tenemos a su vez un circuito, puesto que el camino del vértice a a sí mismo es un camino cerrado en el cual no se repiten aristas.

Definición 1.17. Un *ciclo* es un camino simple que a su vez es un circuito. Es decir, es un camino en el que no hay aristas repetidas ni vértices repetidos, a excepción del primero y el último que coinciden.

En el Ejemplo 1.5, también tenemos un ciclo, ya que no se repiten ni vértices ni aristas a excepción del vértice a.

Proposición 1.2. *Sea $G = (V, E)$ un grafo y $u, v \in V$. Si suponemos que existe un camino de u a v, entonces existe un camino simple de u a v.*

Demostración. Supongamos que existe un camino de u a v, dado por la sucesión de vértices $u_1u_2 \dots u_n$.

Si el camino $u = u_1u_2 \dots u_n = v$ no es simple, entonces existirán al menos $i, j \in \{1, \dots, n\}$ con $i < j$, tal que $u_i = u_j$.

Por tanto, tomaremos como camino $u_1 \dots u_iu_{j+1} \dots u_n$. Volvemos a comprobar si es un camino simple o no y, en caso de no serlo, repetimos el proceso hasta obtener un camino simple. □

Proposición 1.3. *Sea $G = (V, E)$ un grafo y $u, v \in V$ con $u \neq v$. Si tenemos dos caminos simples distintos de u a v, entonces existe un ciclo en el grafo.*

Demostración. Supongamos que existen dos caminos simples de u a v, definidos por las sucesiones de vértices $u = u_1u_2 \dots u_n = v$ y $u = v_1v_2 \dots v_m = v$. Por la definición de camino simple, no hay vértices repetidos en cada uno de ellos.

Si tomamos el camino definido por la sucesión de vértices $u_1u_2 \dots u_n v_{m-1} \dots v_2v_1$, tenemos un camino cerrado.

Si el camino obtenido no es simple, entonces existirán un $i \in \{1, \dots, n\}$ y un $j \in \{1, \dots, m\}$, tal que $u_i = v_j$.

Tomando el camino $u_1u_2 \dots u_i v_{j-1} \dots v_2v_1$, volvemos a comprobar si es un camino simple, y en caso de no serlo, repetimos el proceso anterior hasta obtenerlo.

Una vez obtenido tendremos un ciclo, puesto que tendremos un camino simple que a su vez es un camino cerrado. □

Definición 1.18. La *circunferencia* de un grafo no dirigido es la longitud de su ciclo más largo. Si el grafo no contiene ningún ciclo entonces diremos que su valor es infinito.

Definición 1.19. La *cintura* de un grafo no dirigido es la longitud de su ciclo más corto. Si el grafo no contiene ningún ciclo su cintura será infinita.

En el Ejemplo 1.2 la circunferencia del grafo es 3 y la cintura del mismo es 1.

1.3.2. Tipos de Grafos

En este apartado presentaremos distintos tipos de grafos seguidos de sus respectivos ejemplos.

Definición 1.20. Un grafo $G = (V, E)$ se dice que es *vacío o nulo* si $E = \emptyset$, es decir, si G no contiene ninguna arista.

Ejemplo 1.6. Sea G el grafo que se muestra en la Figura 1.6. Como no contiene ninguna arista, se trata de un grafo vacío.

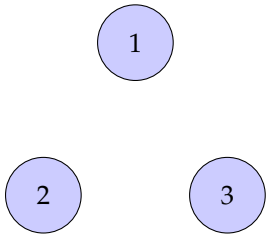


Figura 1.6: Grafo vacío

Definición 1.21. Un grafo $G = (V, E)$ se dice que es *trivial* si $V = \{v\}$, es decir, si el conjunto de vértices está formado por un solo elemento.

Ejemplo 1.7. Sea $G = (V, E)$ el grafo que se muestra en la Figura 1.7. Como está formado por un único vértice, es decir, $V = \{1\}$, entonces se trata de un grafo trivial.

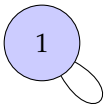


Figura 1.7: Grafo Trivial

Definición 1.22. Un grafo simple $G = (V, E)$ de orden n se dice que es completo si cada vértice está conectado con todos los demás, es decir, si todos los vértices son adyacentes entre sí. Dicho grafo se suele denotar como K_n.

En un grafo completo de orden n se tiene que el grado de cada uno de sus vértices es $n - 1$.

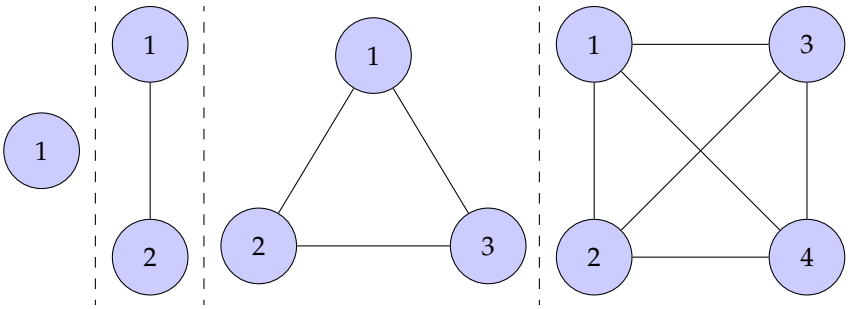


Figura 1.8: Grafos Completos de órdenes 1 hasta 4

Ejemplo 1.8. Sea G uno de los grafos que se muestra en la Figura 1.8. Como cada vértice está conectado con todos los demás, se trata de un grafo completo.

Definición 1.23. Diremos que un grafo $G = (V, E)$ es *regular* de orden n si todos sus vértices tienen el mismo grado, es decir, $\mathrm{gr}(v) = n$ para todo $v \in V$.

Ejemplo 1.9. El grafo K_4 de la Figura 1.8 es regular de orden 3, puesto que $\mathrm{gr}(1) = \mathrm{gr}(2) = \mathrm{gr}(3) = \mathrm{gr}(4) = 3$.

Definición 1.24. Dado un grafo $G = (V, E)$. Diremos que G es *conexo*, si dados $u, v \in V$ existe al menos un camino de u a v, es decir, si todas las posibles parejas de vértices del grafo están conectadas por al menos un camino.

Si $G = (V, E)$ es un grafo, podemos definir una relación sobre los vértices del mismo:

$$uRv \text{ si existe un camino de u a v.}$$

Dicha relación es de equivalencia, ya que verifica las siguientes propiedades.

- Es reflexiva, puesto que todo vértice está unido con el mismo por un camino de longitud 0.
- Es simétrica, ya que si $e_1 e_2 \dots e_n$ es un camino de u a v entonces $e_n \dots e_2 e_1$ es un camino de v a u.

- Es transitiva, puesto que si $e_1e_2 \dots e_n$ es un camino de u a v y $a_1a_2 \dots a_m$ es un camino de v a w, entonces $e_1e_2 \dots e_na_2 \dots a_m$ es un camino de u a w.

Por tanto, un grafo es *conexo* si el conjunto cociente de V por la relación previamente definida está formado por un solo elemento.

En caso contrario, diremos que el grafo es *no conexo* o *disconexo* y definiremos por *componente conexa* a cada subgrafo completo de G generado por los vértices de cada clase de equivalencia.

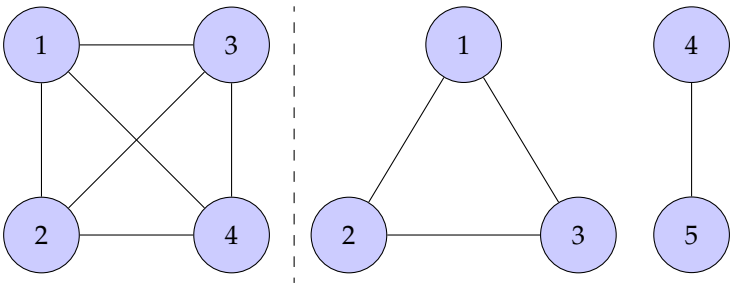


Figura 1.9: Grafos con una y dos componentes conexas respectivamente

Ejemplo 1.10. Sean G_1 y G_2 los grafos que se muestran en la Figura 1.9.

El grafo G_1 es conexo puesto que siempre existe un camino para cualquier par de vértices. Dicho grafo tiene por tanto una sola componente conexa.

El grafo G_2 no es conexo, al no existir un camino que una el vértice 1 y el 4. Sin embargo, los conjuntos de nodos $\{1, 2, 3\}$ y $\{4, 5\}$ forman dos componentes conexas.

Definición 1.25. Diremos que un grafo $G = (V, E)$ es *ponderado* si existe una función $w : E \longrightarrow \mathbb{R}$ que le asigna un peso a cada una de sus aristas.

Ejemplo 1.11. El grafo de la Figura 1.10 es ponderado con $w : \{e_1, e_2, e_3\} \longrightarrow \mathbb{R}$:

$$w(e_1) = \{5\}, \quad w(e_2) = \{8\}, \quad w(e_3) = \{2\}.$$

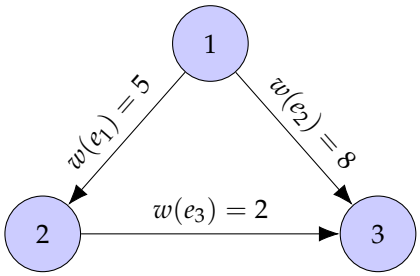


Figura 1.10: Grafo ponderado

Definición 1.26. Diremos que un grafo $G = (V, E)$ es *etiquetado* si sus vértices tienen etiquetas. Se puede distinguir entre etiquetado de vértices y aristas. El etiquetado de vértices se define mediante una función desde el conjunto de vértices hacia un conjunto simbólico

(normalmente de letras o números). Si el grafo presenta etiquetado de aristas a un conjunto ordenado éste puede ser un grafo ponderado.

Definición 1.27. Un grafo $G = (V, E)$ se dice que es *aleatorio* cuando sus aristas y vértices siguen una distribución aleatoria. Por tanto, dichos grafos no pueden ser diseñados siguiendo un criterio o patrón concreto.

Definición 1.28. Sea $G = (V, E)$ un grafo simple. El *complementario* de G es el grafo simple $\overline{G} = (V, \overline{E})$ que tiene el mismo conjunto de vértices, y tiene aquellas aristas que no aparecen en E.

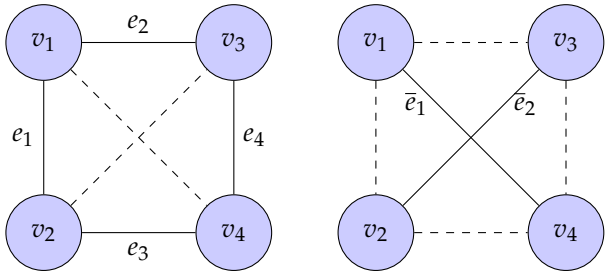


Figura 1.11: Grafos complementarios

Ejemplo 1.12. Sean G y $\overline{G}$ los grafos de la Figura 1.11. Sea $G = (V, E)$, donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4\}$ y el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4\}$, dadas por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_1, v_3\}, \quad \gamma_G(e_3) = \{v_2, v_4\}, \quad \gamma_G(e_4) = \{v_3, v_4\}.$$

Sea $\overline{G} = (\overline{V}, \overline{E})$, donde el conjunto de vértices es $\overline{V} = V$ y el de aristas es $\overline{E} = \{\overline{e_1}, \overline{e_2}\}$, y cuya aplicación de incidencia se define como sigue:

$$\gamma_G(e_1) = \{v_1, v_4\}, \quad \gamma_G(e_2) = \{v_2, v_3\}.$$

Podemos apreciar que ambos grafos son complementarios, al verificarse $\overline{V} = V$ y al ser $\overline{E}$ el conjunto de aristas que no aparecen en E.

1.4. Matrices asociadas a grafos

Los grafos pueden ser representados de muchas formas distintas. En esta sección veremos que los grafos también pueden representarse como matrices, las cuales pueden resultar ventajosas para extraer ciertas propiedades e información sobre el grafo.

Existen diversos tipos de matrices asociadas a un grafo. Nos centraremos en estudiar la *matriz de adyacencia* y la *matriz de incidencia*. A continuación, definiremos ambas matrices y algunas de sus propiedades sobre los grafos que caracterizan.

Definición 1.29. Sea $G = (V, E)$ un grafo donde el conjunto de vértices es $V = \{v_1, \ldots, v_n\}$. Definimos su matriz de *adyacencia* como la matriz $A \in \mathcal{M}_n(\mathbb{N})$ cuya entrada (i, j) con $i, j \in \{1, \ldots, n\}$ se identifica con el número de aristas que unen los vértices v_i y v_j, es decir, que cumplen $\gamma_G(e) = \{v_i, v_j\}$.

Si G es un grafo ponderado, la entrada (i, j) con $i, j \in \{1, \cdots, n\}$ se identifica con el peso de la arista que une el vértice v_i con el vértice v_j.

A continuación, estudiaremos algunas de las propiedades de este tipo de matrices.

- La matriz de adyacencia de un grafo no dirigido es una matriz simétrica, ya que si una arista une un vértice v_i con uno v_j, también une v_j con v_i.
- Si tomamos una ordenación diferente de los vértices, la matriz de adyacencia sería distinta. Por ello, un mismo grafo puede tener diversas matrices de adyacencia. Si A y B son matrices de adyacencia de un grafo, entonces existe una matriz de permutación P que cumple $P^{-1}BP = A$. Además, la matriz P verifica que en cada fila y columna tiene un elemento que vale 1 y el resto valen 0.
- Si existen aristas paralelas en un grafo, entonces existirán elementos de la matriz que tomen valores mayores que 1.
- Si existen lazos en un grafo, entonces existirán elementos de la diagonal de la matriz que tomen valores distintos de 0.
- Si el grafo es no dirigido y la fila i-ésima es nula, entonces el vértice i es un vértice aislado.

Ejemplo 1.13. La matriz de adyacencia del grafo de la Figura 1.4 es:

$$\begin{pmatrix} 0 & 1 & 2 & 0 \\ 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Podemos apreciar que al ser el grafo no dirigido tenemos una matriz simétrica. Además, como el vértice 4 está aislado, la cuarta fila es nula. Asimismo, por la diagonal podemos observar que el vértice 2 tiene un lazo.

Por otro lado, estudiaremos algunas particularidades de la matriz de adyacencia asociada a distintos tipos de grafos.

- Si el grafo es dirigido, podemos asignar a cada entrada (i, j) de la matriz, con $i, j \in \{1, \ldots, n\}$, el número de aristas que cumplen $s(e) = v_i$ y $t(e) = v_j$. En estos grafos, la matriz no tiene que ser simétrica.
- Un grafo puede venir determinado por su matriz de adyacencia. Por ello, podemos definir un grafo como una matriz cuadrada con coeficientes en $\mathbb{N}$.
- Si el grafo es completo y suponemos que tiene orden n, entonces todos los elementos de la diagonal principal de la matriz de adyacencia serían cero, y el resto de elementos serían $n - 1$.
- Si G es un grafo de orden n y todos los elementos de la matriz de adyacencia que no pertenecen a la diagonal principal son no nulos, entonces G es conexo.

Ejemplo 1.14. La matriz de adyacencia del grafo dirigido de la Figura 1.5 es:

$$\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Podemos observar que en este caso no tenemos una matriz simétrica.

Ejemplo 1.15. La matriz de adyacencia del grafo completo y conexo de orden 4 de la Figura 1.8 es:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Podemos ver que en este caso todos los elementos de la matriz menos los de la diagonal principal son 1, y por tanto, no nulos.

Proposición 1.4. *Sea $G = (V, E)$ un grafo donde el conjunto de vértices es $V = \{v_1, \ldots, v_n\}$ y sea A su matriz de adyacencia. Entonces la entrada (i, j) de la matriz A^n es igual al número de caminos de longitud n que conectan v_i con v_j.*

Demostración. Vamos a realizar la demostración por inducción. Para $n = 1$, tenemos la propia matriz de adyacencia A, y por su propia definición verifica que a_{ij} es el número de caminos de longitud 1, es decir, es el número de aristas que unen v_i con v_j.

Supongamos ahora que el resultado se verifica para $n - 1$ y demostremos que es cierto para n.

Definimos las matrices $B = A^{n-1}$ y $C = A^n$. Lo que queremos demostrar es que c_{ij} equivale al número de caminos de longitud n que conectan los vértices v_i y v_j con $i, j \in \{1, \ldots, n\}$. Sabemos que $c_{ij} = \sum_{k=1}^{n} b_{ik} a_{kj}$. Por tanto, todos los caminos de longitud n entre los vértices v_i y v_j podemos obtenerlos añadiendo el vértice v_j a un camino de longitud $n - 1$ entre v_i y v, siendo v un vértice cualquiera. Sin embargo, esto solo podremos hacerlo cuando tengamos una arista que incida en los vértices v y v_j. Es por ello, que para poder obtener el número de caminos de longitud n entre v_i y v_j es necesario contar el número de caminos de longitud $n - 1$ entre v_i y v_k con $k \in \{1, \ldots, n\}$, y para cada uno de estos, contar el número de aristas que unen v_k y v_j.

Así pues, el número de caminos de longitud n entre v_i y v_j es, por hipótesis de inducción:

$$c_{ij} = b_{i1}a_{1j} + b_{i2}a_{2j} + \cdots + b_{in}a_{nj}. \qquad \square$$

Definición 1.30. Sea $G = (V, E)$, donde V es el conjunto de vértices $V = \{v_1, \ldots, v_n\}$ y E es el conjunto de aristas $E = \{e_1, \ldots, e_m\}$. Definimos la matriz de *incidencia* del grafo como la matriz de dimensión $n \times m$ tal que:

$$a_{ij} = \begin{cases} 1 \text{ si} \quad v_i \in \gamma_G(e_j), \\ 0 \text{ en otro caso.} \end{cases}$$

Si tenemos un grafo dirigido, la entrada (i, j) de la matriz con $i, j \in \{1, \ldots, n\}$, puede tomar el valor -1. En concreto, le asignamos a los vértices inicial el valor -1 y a los vértices final 1. Sin embargo, no podemos definir la matriz de incidencia para grafos dirigidos con lazos.

A continuación, enumeramos algunas de las propiedades de este tipo de matrices.

- La matriz de incidencia no tiene porqué ser simétrica.
- La matriz de incidencia no tiene porqué ser cuadrada. Tendrá tantas filas como vértices tenga el grafo, y tantas columnas como aristas posea.
- Dado un grafo G, si tomamos una ordenación diferente de los vértices o aristas o los renombramos podemos obtener una matriz de incidencia diferente, pero podemos llegar a la anterior a través de operaciones de intercambio de sus filas o columnas.
- Si existen aristas paralelas en un grafo, entonces la matriz de incidencia tendrá dos columnas iguales.
- Si existen lazos en el grafo, entonces existirán columnas con un único elemento igual a 1.
- Si la fila i-ésima es nula, entonces el vértice v_i es aislado.

Ejemplo 1.16. La matriz de incidencia del grafo de la Figura 1.4 es:

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Por un lado, como las aristas e_2 y e_4 son paralelas, las columnas 2 y 4 son iguales. Por otro lado, como e_5 es un lazo, la quinta columna solo tiene un elemento no nulo.

En particular, si tenemos un grafo completo de orden n, entonces la matriz de incidencia estará compuesta por n filas y $n(n-1)/2$ columnas, y en cada fila tiene $n-1$ elementos que son 1 y el resto 0.

Ejemplo 1.17. Sea $G = (V, E)$ el grafo completo de orden 4, donde $V = \{v_1, v_2, v_3, v_4\}$ es el conjunto de vértices, $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ el conjunto de aristas, y la aplicación de incidencia se define como:

$$\gamma_G(e_1) = \{1, 2\}, \quad \gamma_G(e_2) = \{1, 3\}, \quad \gamma_G(e_3) = \{1, 4\},$$

$$\gamma_G(e_4) = \{2, 3\}, \quad \gamma_G(e_5) = \{2, 4\}, \quad \gamma_G(e_6) = \{3, 4\}.$$

Su matriz de incidencia asociada sería:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Podemos apreciar que la matriz tiene 4 filas y $4 \cdot (4-1)/2 = 6$ columnas, y en cada fila hay $4 - 1 = 3$ elementos que son 1 y el resto 0.

1.5. Isomorfismo de grafos

En esta sección comenzaremos una pequeña discusión motivados por la siguiente pregunta: ¿cuándo son dos grafos distintos?

Consideramos los grafos G_1 y G_2 de la Figura 1.12. Si observamos ambos grafos, podemos apreciar que parecen diferentes. No obstante, analizando sus características podemos apreciar que existen muchas similitudes entre ambos.

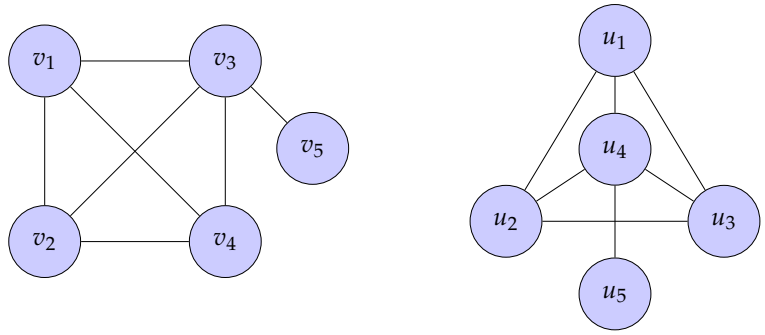


Figura 1.12: Grafos isomorfos G_1 y G_2

Por un lado, si denotamos ambos grafos por $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, podemos observar que tienen el mismo orden $|V_1| = |V_2| = 5$ y el mismo tamaño $|E_1| = |E_2| = 7$. Asimismo, en los dos grafos existe un vértice que está conectado al resto de los vértices del grafo, en el caso del grafo G_1 tenemos el vértice v_3 y en el de G_2 el vértice u_4.

Si renombramos los vértices del segundo grafo como

$$u_1 \mapsto v'_1, \quad u_2 \mapsto v'_2, \quad u_3 \mapsto v'_4, \quad u_4 \mapsto v'_3, \quad u_5 \mapsto v'_5,$$

entonces tenemos que por cada arista que conecta los vértices v_i y v_j en el primer grafo, tenemos una arista que conecta los vértices v'_i y v'_j en el segundo.

De esta forma, podemos ver que ambos grafos se pueden considerar *iguales*, con la distinción residiendo en el nombre de los vértices y las aristas, y la forma de representarlos. Así pues, podemos decir que las propiedades que se verifiquen para el primer grafo, se van a verificar para el segundo también.

A continuación, vamos a nombrar las aristas como sigue en la Figura 1.13.

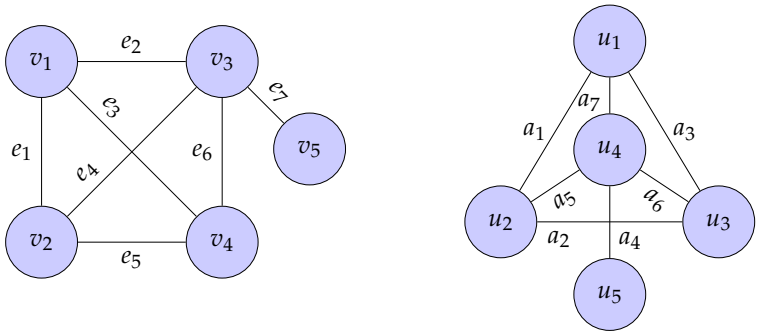


Figura 1.13: Grafos isomorfos G_1 y G_2

De este modo, podemos definir dos biyecciones $h_V : V_G \mapsto V_{G'}$ y $h_E : E_G \mapsto E_{G'}$:

$$h_V(v_1) = u_1, \quad h_V(v_2) = u_2, \quad h_V(v_3) = u_4, \quad h_V(v_4) = u_3, \quad h_V(v_5) = u_5,$$

$$h_E(e_1) = a_1, \quad h_E(e_2) = a_7, \quad h_E(e_3) = a_3, \quad h_E(e_4) = a_5,$$

$$h_E(e_5) = a_2, \quad h_E(e_6) = a_6, \quad h_E(e_7) = a_4,$$

cumpliendo que si $\gamma_G(e) = \{v_i, v_j\}$ entonces $\gamma_{G'}(h_E(e)) = \{h_V(v_i), h_V(v_j)\}$.

A partir de esta motivación, podemos pasar a definir los conceptos de morfismo e isomorfismo de grafos.

Definición 1.31. Sean dos grafos $G = (V, E)$ y $G' = (V', E')$ con aplicaciones de incidencia γ_G y $\gamma_{G'}$ respectivamente. Diremos que un *morfismo* entre ambos grafos es un par de aplicaciones $\phi_V : V \longrightarrow V'$ y $\phi_E : E \longrightarrow E'$ de forma que para toda arista $e \in E$ con $\gamma_G(e) = \{u, v\}$, se cumple que $\gamma_{G'}(\phi_E(e)) = \{\phi_V(u), \phi_V(v)\}$.

Definición 1.32. Sean dos grafos $G = (V, E)$ y $G' = (V', E')$ con aplicaciones de incidencia γ_G y $\gamma_{G'}$ respectivamente. Diremos que un *isormorfismo* entre ambos grafos es un morfismo cuyo par de aplicaciones son biyectivas. Por lo que, ambos grafos serán *isomorfos* si existen dos biyecciones $h_V : V \longrightarrow V'$ y $h_E : E \longrightarrow E'$ tales que para toda arista $e \in E$ con $\gamma_G(e) = \{u, v\}$, se cumple que $\gamma_{G'}(h_E(e)) = \{h_V(u), h_V(v)\}$. En ese caso, diremos que las aplicaciones h_V y h_E forman un isomorfismo de G a G'.

En los grafos sin lados paralelos, el morfismo y el isomorfismo pueden venir determinados únicamente por la aplicación ϕ_V y h_V, respectivamente.

Proposición 1.5. *La relación de isomorfía es una relación de equivalencia.*

La demostración de esta proposición se sigue de forma sencilla usando la definición.

Proposición 1.6. *Dos grafos son isomorfos si, y sólo si, sus complementarios también lo son.*

No obstante, en la práctica no es sencillo determinar si dos grafos son isomorfos. Una condición necesaria muy útil para determinar si dos grafos son isomorfos es la siguiente.

Proposición 1.7. *Si los grafos G_1 y G_2 son isomorfos, entonces el sucesión de vértices de G_1 coincide con la de G_2.*

Demostración. ([Cha77], Capítulo 2). Supongamos que los grafos G_1 y G_2 son isomorfos. Por tanto, existe una biyección $h_V : V \longrightarrow V'$.

Sea $u \in V_1$ y supongamos que $\mathrm{gr(u)} = n$ y $h_V(u) = v$. Probemos entonces que $\mathrm{gr(v)} = n$.

Puesto que $\mathrm{gr(u)} = n$, sabemos que el grafo G_1 tiene n vértices, $u_1 \cdots u_n$, adyacentes a u. Sean $h_V(u_i) = v_i$ con $i = 1, 2, \cdots, n$. Al ser h_V biyectivo, v es adyacente a todos los vértices $v_1, \cdots, v_n$, por lo que $\mathrm{gr(v)} \geq n$.

Asimismo, como u es adyacente a un vértice $w \in V$ si, y sólo si, v es adyacente a $h_V(w) \in V'$. Entonces, $\mathrm{gr(v)} = n$. □

Cabe remarcar, que la condición anterior es necesaria pero no suficiente para saber si existe esta relación entre ambos grafos. Lo mismo sucede con el número de vértices y aristas, como ocurre en el ejemplo anterior.

En el siguiente ejemplo, tenemos dos grafos que coinciden en orden, tamaño y sucesión de grados. Sin embargo, como hemos comentado, estas propiedades no nos garantizan que ambos grafos sean isomorfos. De hecho, no podemos definir una biyección entre los conjuntos de vértices y aristas de ambos grafos, puesto que en el primer grafo los vértices de grado 2 no son adyacentes a diferencia del segundo.

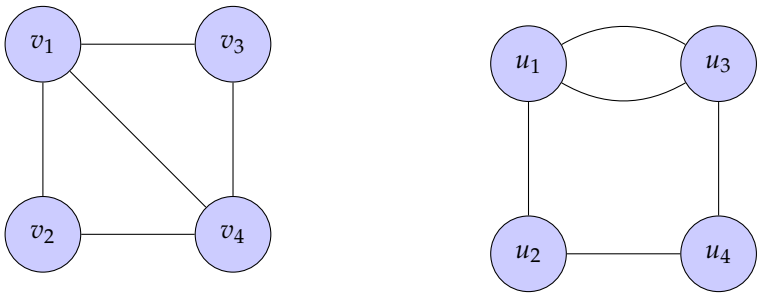


Figura 1.14: Grafos no isomorfos

A continuación, veremos que hay ciertas propiedades que no van a variar entre grafos isomorfos.

Definición 1.33. Diremos que una propiedad es *invariante por isomorfismo* si dados dos grafos isomorfos G y G', uno satisface la propiedad si, y sólo si, el otro también la satisface.

Las propiedades de tamaño y de ser de un determinado orden de un grafo son invariantes por isomorfismos.

Definición 1.34. Un *automorfismo* de un grafo $G = (V, E)$ es un isomorfismo del grafo G en sí mismo.

Para grafos simples un automorfismo de un grafo $G = (V, E)$ es una permutación $h_V : V \longrightarrow V$ del conjunto de vértices V, de forma que $\{u, v\} \in E$ si, y sólo si, $\{h_V(u), h_V(v)\} \in E$.

El conjunto de todos los automorfismos de un grafo G forma un grupo. Recordemos que un grupo es una estructura algebraica compuesta por un conjunto de elementos y una operación interna que verifica las propiedades de existencia de elemento neutro, inverso y asociatividad. En este caso la operación interna es la composición, ya que el elemento neutro es la aplicación identidad, que es un automorfismo; el elemento inverso es la propia inversa de un automorfismo, que también es un automorfismo; y la composición de dos automorfismos es asociativa y es un automorfismo. Denotaremos a este grupo por $Aut(G)$ y lo denominaremos *grupo de automorfismos de G*.

Nota 1.2. *Cuando dos grafos $G = (V, E)$ y $G' = (V', E')$ son isomorfos, entonces el número de isomorfismos de G a G′ es igual al orden del grupo $Aut(G)$.*

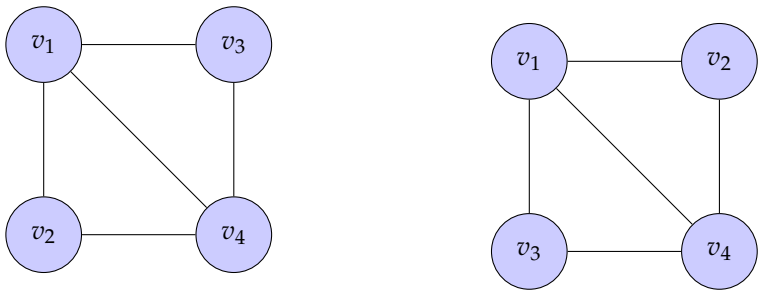


Figura 1.15: Automorfismo de grafos

Ejemplo 1.18. Sea G el grafo simple de las representaciones de la Figura. 1.15. Sea $G = (V, E)$ donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4\}$ y el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4, e_5\}$.

Podemos definir la biyección $h_V : V_G \mapsto V_G$:

$$h_V(v_1) = v_1, \quad h_V(v_2) = v_3, \quad h_V(v_3) = v_2, \quad h_V(v_4) = v_4,$$

verificando que si $\{v_i, v_j\} \in E_G$ entonces $\{h_V(v_i), h_V(v_j)\} \in E_G$.

A partir de la noción de automorfismo, podemos definir el siguiente tipo de grafo.

Definición 1.35. Un grafo $G = (V, E)$ se dice *transitivo respecto a sus vértices* si para cualesquiera dos vértices $u, v \in V$ existe un automorfismo $h_V : V \longrightarrow V$ tal que $h_V(u) = v$.

1.6. Grafos de Euler

En este apartado presentaremos un tipo de grafo que le debe su nombre al matemático Leonard Euler. Después, estudiaremos algunas de sus propiedades y caracterizaciones.

Definición 1.36. Sea $G = (V, E)$ un grafo conexo. Diremos que G posee un *camino de Euler* si existe un recorrido en el que aparecen todas las aristas del grafo. Diremos que G posee un *circuito de Euler* si tiene un recorrido cerrado en el que aparecen todas las aristas de G. Definiremos un *grafo de Euler* como un grafo que posee un circuito de Euler.

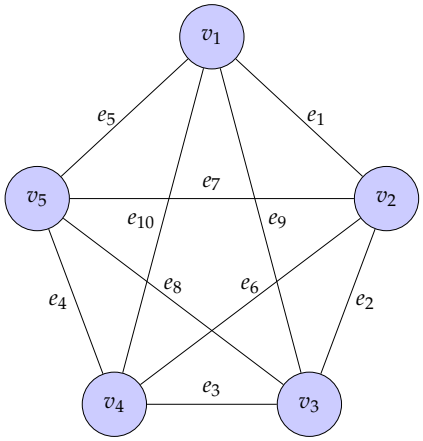


Figura 1.16: Grafo de Euler

Ejemplo 1.19. Sean G el grafo conexo de la Figura. 1.16, donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4, v_5\}$ y el conjunto de aristas es $E_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$, dadas por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_2, v_3\}, \quad \gamma_G(e_3) = \{v_3, v_4\}, \quad \gamma_G(e_4) = \{v_4, v_5\},$$
$$\gamma_G(e_5) = \{v_5, v_1\}, \quad \gamma_G(e_6) = \{v_2, v_4\}, \quad \gamma_G(e_7) = \{v_2, v_5\}, \quad \gamma_G(e_8) = \{v_3, v_5\},$$
$$\gamma_G(e_9) = \{v_1, v_3\}, \quad \gamma_G(e_{10}) = \{v_1, v_4\}.$$

Si tomamos la sucesión de aristas $e_1e_2e_3e_4e_5e_9e_8e_7e_6e_{10}$ tenemos un circuito de Euler, por lo que el grafo G es un grafo de Euler.

Ejemplo 1.20. Sean G el grafo conexo de la Figura. 1.17, donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4, v_5\}$ y el conjunto de aristas es $E_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$, dadas por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_2, v_3\}, \quad \gamma_G(e_3) = \{v_3, v_4\}, \quad \gamma_G(e_4) = \{v_4, v_5\},$$
$$\gamma_G(e_5) = \{v_5, v_1\}, \quad \gamma_G(e_6) = \{v_2, v_4\}, \quad \gamma_G(e_7) = \{v_2, v_5\}, \quad \gamma_G(e_8) = \{v_3, v_5\}.$$

Tomando la sucesión de aristas $e_2e_1e_5e_7e_6e_3e_8e_4$ tenemos un camino de Euler.

Proposición 1.8. *Sea $G = (V, E)$ un grafo. Si G tiene un circuito de Euler, entonces el grado de cada vértice es par, mientras que si G tiene un camino de Euler, G tiene dos vértices de grado impar, en concreto los extremos del camino.*

Demostración. Supongamos que G tiene un circuito de Euler. Dado un vértice cualquiera $v \in V$, veamos cual es su grado. Recorremos el circuito de Euler empezando en un vértice

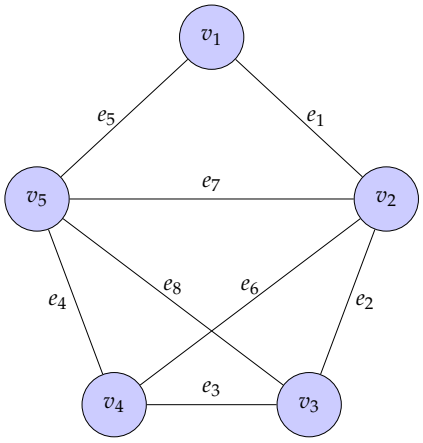


Figura 1.17: Grafo con un camino de Euler

distinto de $u \in V \backslash \{v\}$ y vamos contando el número de aristas que son incidentes en v. Para ello tenemos en cuenta que cada vez que pasamos por el vértice v encontramos dos aristas incidentes sobre él, una entrante y una saliente, por lo que el grado de v será el doble que el número de veces que el circuito pase por v. Por tanto, el grado de cada vértice será par.

Supongamos ahora que G tiene un camino de Euler, que comienza en u y v. Vamos a añadir al grafo G una arista que una ambos vértices, obteniendo así un circuito de Euler, por lo que para ese nuevo grafo el grado de cada vértice es par.

Si eliminamos la nueva arista, el grado de todos los vértices será el mismo excepto el de los extremos que disminuirá en uno, por lo que el grafo tendrá exactamente dos vértices de grado impar. □

En el Ejemplo. 1.19 el grafo G tiene un circuito de Euler y $\mathrm{gr}(v) = 4$ es par para todo $v \in V$. En el Ejemplo. 1.20 el grafo G tiene un camino de Euler cuyos extremos son v_3 y v_4 y, $\mathrm{gr}(v_5) = \mathrm{gr}(v_2) = 4$, $\mathrm{gr}(v_1) = 2$ y $\mathrm{gr}(v_3) = \mathrm{gr}(v_4)$. Por lo que, vemos que tiene dos vértices de grado impar que son los extremos.

Lema 1.1. *Sea $G = (V, E)$ un grafo en el que cada vértice tiene grado mayor que 1. Entonces G contiene un circuito, y por tanto, un ciclo.*

Demostración. Tomamos un vértice $v_1 \in V$. Por hipótesis $\mathrm{gr}(v_1) > 1$. Tomamos ahora una arista que incida en dicho vértice, e_1, y otro vértice que sobre el que incide e_1, v_2.

Si $v_1 = v_2$, entonces ya tendríamos un recorrido. Si por el contario, $v_1 \neq v_2$, como $\mathrm{gr}(v_2) > 1$, existe otra arista incidente con v_2 y con otro vértice v_3. De esta forma tenemos el camino formado por la sucesión de vértices $v_0 v_1 v_2$.

Si seguimos el proceso hasta encontrar algún vértices repetido sin haber repetido ninguna arista, ya habremos encontrado el circuito buscado. □

Teorema 1.3. *Sea $G = (V, E)$ un grafo conexo. Entonces G es un grafo de Euler si, y sólo si, cada vértice tiene grado par.*

Demostración. Supongamos que $G = (V, E)$ es un grafo de Euler conexo y, por tanto, tiene un circuito de Euler. Por la Proposición 1.8 tenemos que cada vértice de V tiene grado par.

Supongamos ahora que G es un grafo conexo y que cada vértice tiene grado par. Veamos que G es un grafo de Euler.

Realizaremos la demostración de esta implicación mediante inducción sobre el número de aristas n.

Para $n = 0$ tenemos que, al ser G conexo, solamente tiene un vértice, que es de grado 0 y, por tanto, par. Para $n = 1$, si G es conexo y cada vértice debe tener grado par, entonces G solamente puede tener un vértice con un lazo, que sería el propio circuito de Euler.

Supongamos que el resultado se verifica para un grafo $G = (V, E)$ conexo con $|E| < n$ y donde $\mathrm{gr(v)}$ es par para todo $v \in V$, y probemos que se cumple para $|E| = n$.

Al ser G un grafo conexo por hipótesis, entonces G no tiene vértices aislados. Por lo que aplicando el Lema. 1.1 sabemos que existe un circuito en G. Denotemos dicho circuito por C. Si eliminamos C de G se mantiene que el grado de todos los vértices es par puesto que para cada vértice se ha quitado la arista entrante y saliente desde él. Aunque el grafo obtenido no tiene porqué se conexo, para cada componente conexa del mismo que hay al menos una arista, habrá un circuito de Euler, ya que hemos supuesto que el resultado se verifica para todo grafo de tamaño menor que n.

Denotamos por $c_1, \ldots, c_m$ a los circuitos de las respectivas componente conexas. Para cada uno de los circuitos c_i, existe un vértice v_i que estará también en C.

Al recorrer el circuito C, cuando lleguemos a algún vértice v_i añadimos el circuito c_i y seguimos con el circuito c. Así pues, al cerrar el circuito C habremos recorrido todas las arista del grafo G solamente un vez, y tendremos un circuito de Euler. □

Corolario 1.1. *Sea $G = (V, E)$ un grafo conexo. Entonces G tiene un camino de Euler si, y sólo si, G tiene solamente dos vértices de grado impar.*

1.7. Grafos de Hamilton

En esta sección, estudiaremos otro tipo de grafos de gran importancia: los grafos de Hamilton. Estos grafos deben su nombre al matemático irlandés William Rowan Hamilton.

Definición 1.37. Sea $G = (V, E)$ un grafo. Diremos que G posee un *camino de Hamilton* si existe un camino que recorre todos los vértices solo una vez. Diremos que G posee un *circuito de Hamilton* si tiene un camino cerrado que recorre todos los vértices una sola vez. Definiremos un *grafo de Hamilton* o *grafo hamiltonioano* como un grafo que posea un circuito de Hamilton.

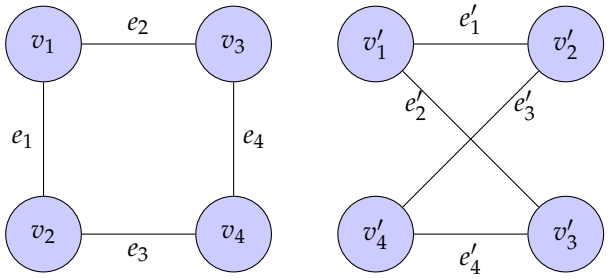


Figura 1.18: Grafos de Hamilton

Ejemplo 1.21. Sean G_1 y G_2 los grafos de la Figura. 1.18. Sea $G_1 = (V_1, E_1)$ donde el conjunto de vértices es $V_1 = \{v_1, v_2, v_3, v_4\}$ y el conjunto de aristas es $E_1 = \{e_1, e_2, e_3, e_4\}$, dadas por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_1, v_3\}, \quad \gamma_G(e_3) = \{v_2, v_4\}, \quad \gamma_G(e_4) = \{v_3, v_4\}.$$

Si tomamos la sucesión de vértices $v_1v_2v_4v_3v_1$ tenemos un circuito de Hamilton, por lo que el grafo G_1 es un grafo de Hamilton.

Sea $G_2 = (V_2, E_2)$ donde el conjunto de vértices es $V_2 = \{v'_1, v'_2, v'_3, v'_4\}$ y el de aristas es $E_1 = \{e'_1, e'_2, e'_3, e'_4\}$, y cuya aplicación de incidencia se define como sigue:

$$\gamma_G(e'_1) = \{v'_1, v'_2\}, \quad \gamma_G(e'_2) = \{v'_1, v'_3\}, \quad \gamma_G(e'_3) = \{v'_2, v'_4\}, \quad \gamma_G(e'_4) = \{v'_3, v'_4\}.$$

Tomando la sucesión de vértices $v'_1v'_2v'_4v'_3v'_1$ obtenemos un circuito de Hamilton. Por tanto, G_2 es un grafo de Hamilton.

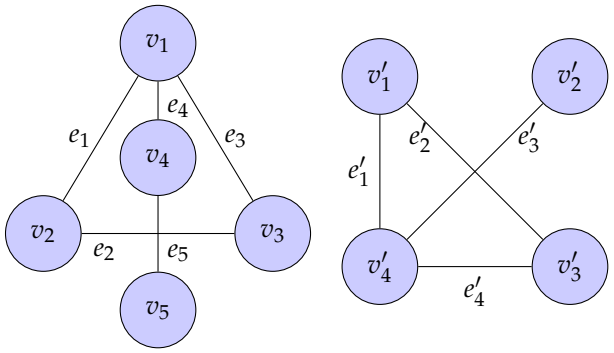


Figura 1.19: Grafos con un camino de Hamilton.

Ejemplo 1.22. Sean G_1 y G_2 los grafos de la Figura. 1.19. Sea $G_1 = (V_1, E_1)$ donde el conjunto de vértices es $V_1 = \{v_1, v_2, v_3, v_4, v_5\}$ y el conjunto de aristas es $E_1 = \{e_1, e_2, e_3, e_4\}$, dadas por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_2, v_3\}, \quad \gamma_G(e_3) = \{v_1, v_3\},$$
$$\gamma_G(e_4) = \{v_1, v_4\}, \quad \gamma_G(e_5) = \{v_4, v_5\}.$$

Podemos tomar el camino de Hamilton $v_5v_4v_1v_2v_3$. Sin embargo, no podemos encontrar ningún circuito de Hamilton.

Sea $G_2 = (V_2, E_2)$ donde el conjunto de vértices es $V_2 = \{v'_1, v'_2, v'_3, v'_4\}$ y el de aristas es $E_1 = \{e'_1, e'_2, e'_3, e'_4\}$, y cuya aplicación de incidencia se define como sigue:

$$\gamma_G(e'_1) = \{v'_1, v'_2\}, \quad \gamma_G(e'_2) = \{v'_1, v'_3\}, \quad \gamma_G(e'_3) = \{v'_2, v'_4\}, \quad \gamma_G(e'_4) = \{v'_3, v'_4\}.$$

Tomando la sucesión de vértices $v'_2v'_4v'_3v'_1$ obtenemos un camino de Hamilton, pero no podemos encontrar ningún circuito de Hamilton.

Podemos apreciar que ningún grafo es de Hamilton, puesto que en un circuito de Hamilton, cada vértice tiene al menos dos aristas adyacentes, una para llegar al vértice y otra para partir desde él. En el caso del grafo G_1, el vértice v_5 solamente tiene una arista adyacente, del mismo modo en el grafo G_2, el vértice v'_2 también tiene grado 1.

Cabe notar, que si no podemos pasar dos veces por el mismo vértices en un camino o en un circuito de Hamilton, entonces carece de sentido considerar grafos con lazos o aristas paralelas, por lo que suponemos que trabajamos con grafos simples.

Teorema 1.4 (Teorema de Dirac). *Sea $G = (V, E)$ un grafo simple con orden $n \geq 3$. Si para cada vértice $v \in V$ se cumple $\mathrm{gr(v)} \geq n/2$, entonces G es un grafo de Hamilton.*

Demostración. Supongamos que $G = (V, E)$ verifica las hipótesis del teorema. Entonces, G tiene que se conexo. De no serlo tendríamos que el grado de cualquier vértice de la menor componente conexa C de G tendría $|C| - 1 < n/2$, ya que como mucho la menor componente conexa puede tener $n/2$ vértices. Por lo que llegaríamos a una contradicción, ya que $\mathrm{gr(v)} \geq \mathrm{n}/2$ para todo vértice $v \in V$.

Tomemos ahora el camino de mayor longitud de G, denotémoslo $c = v_0 \dots v_k$. Puesto que dicho camino no se puede extender al ser el de mayor longitud, entonces todos los vértices adyacentes de los extremos v_0 y v_k se encuentran en el camino.

Como por lo menos $n/2$ vértices adyacentes de v_0 se encuentran en el camino c, y por lo menos otros $n/2$ vértices adyacentes de v_k también. Entonces, va a existir algún vértice $v_i \in \{v_0, \dots, v_{k-1}\}$ tal que $(v_0, v_{i+1}), (v_i, v_k) \in E$.

De esta forma si tomamos el circuito,

$$L = v_0v_{i+1}v_{i+2} \dots v_{k-1}v_kv_iv_{i-1} \dots v_1v_0$$

tenemos que es un circuito hamiltoniano de G. Por tanto, tenemos que G es un grafo de Hamilton. □

En el Ejemplo. 1.22, para el grafo $G_1 = (V_1, E_1)$ de orden $n = 5$, podemos ver que al no ser un grafo hamiltoniano entonces no se verifica $\mathrm{gr(v)} \geq n/2$ para todo $v \in V_1$. Concretamente, si tomamos v_5 tenemos que $\mathrm{gr(v_5)} = 1 < |V_1|/2 = 5/2$.

Teorema 1.5. *Sea $G = (V, E)$ un grafo simple con orden n. Si $n \geq 3$ y para cada par de vértices no adyacentes $u, v \in V$ se cumple que $\mathrm{gr(u)} + \mathrm{gr(v)} \geq n$, entonces G es un grafo de Hamilton.*

Demostración. Queremos probar que si G no es un grafo de Hamilton, entonces hay al menos dos vértices $u, v \in V$ no adyacentes de forma que $\mathrm{gr(u)} + \mathrm{gr(v)} \leq n$.

Supongamos que G no es de Hamilton. A continuación, le añadimos una arista al grafo. Si el grafo sigue sin ser de Hamilton, seguimos añadiendo aristas. Si por el contrario hemos obtenido un grafo hamiltoniano, denotamos por $e = (u, v)$ a la última añadida. En el peor de los casos se añadirán tantas aristas hasta obtener un grafo completo. En este último caso, como $n \geq 3$ por hipótesis y $\mathrm{gr(v)} = n - 1$ por definición de grafo completo, tenemos que $\mathrm{gr(v)} \geq n/2$. Por tanto, aplicando el Teorema 1.4 obtenemos que G es un grafo de Hamilton.

El grafo de Hamilton obtenido debe contener un circuito de Hamilton con la arista e, $uvv_3 \cdots v_n u$. Denotemos por H al grafo obtenido antes de añadir la última arista e. Para cada $i \in \{3, n\}$, veamos que no pueden estar las arista uv_{i-1} y vv_i a la vez en el grafo H.

En el caso de $i = 3$, tenemos que $uv_{i-1} = uv_2 = uv$, que no está en H. En el caso $i \geq 4$ y que estuvieran las aristas uv_{i-1} y vv_i, podemos tomar el circuito de Hamilton: $vv_iv_{i+1} \cdots v_n uv_{i-1} \cdots v_3 v$, que no contiene la arista uv. Lo cual nos lleva a una contradicción, ya que el grafo H no es de Hamilton.

Por tanto, H verifica que $\mathrm{gr(u)} + \mathrm{gr(v)} \leq n$, y al ser G subgrafo de H se en G la misma propiedad.

De esta forma, hemos probado que existen dos vértices no adyacentes de forma que la suma de sus grados es menor que n, como queríamos probar. □

En el Ejemplo. 1.22, tomando el grafo $G_1 = (V_1, E_1)$ de orden $n = 5$, al no ser un grafo hamiltoniano no se cumple $\mathrm{gr(v)} + \mathrm{gr(u)} \geq n$ para todo $v, u \in V_1$. En particular, para $v_4, v_5 \in V_1$ tenemos que $\mathrm{gr(v_4)} + \mathrm{gr(v_5)} = 3 < 5$.

Teorema 1.6. *Sea $G = (V, E)$ un grafo con orden n. Si el tamaño del grafo es menor o igual que $\frac{1}{2}(n-1)(n-2) + 2$, entonces el grafo es de Hamilton.*

Demostración. Sean $u, v \in V$ dos vértices no adyacentes. Probemos que $\mathrm{gr(u)} + \mathrm{gr(v)} \geq n$, y apliquemos el Teorema 1.5.

Sea $G' = (V', E')$ un subgrafo completo suyo, donde $V' = V \backslash \{u, v\}$. En particular, el grafo G' es subgrafo a su vez del grafo completo de orden $n - 2$, K_{n-2}. Por ello, $|E'| \leq \frac{(n-2)(n-3)}{2}$.

Asimismo, como la arista $uv \notin E$ tenemos que $|E| = |E'| + \mathrm{gr(u)} + \mathrm{gr(v)}$.

Partiendo de las desigualdades obtenidas y de la hipótesis $\frac{1}{2}(n-1)(n-2) + 2 \leq |E|$, tenemos:

$$\begin{aligned} \frac{1}{2}(n-1)(n-2) + 2 &\leq |E| \\ &= |E'| + \mathrm{gr(u)} + \mathrm{gr(v)} \leq \frac{(n-2)(n-3}{2} + \mathrm{gr(u)} + \mathrm{gr(v)}\,. \end{aligned}$$

De esta forma obtenemos lo que queríamos probar,

$$\begin{aligned} \mathrm{gr}(u)+\mathrm{gr}(v) &\geq \frac{1}{2}(n-1)(n-2)+2-\frac{(n-2)(n-3)}{2} \\ &= \frac{(n-1)(n-2)+4-(n-2)(n-3)}{2} \\ &= \frac{(n-2)[(n-1)-(n-3)]+4}{2} \\ &= \frac{(n-2)(2)+4}{2} \\ &= \frac{2n-4+4}{2} \\ &= \frac{2n}{2}=n. \quad \square \end{aligned}$$

En el Ejemplo. 1.22, tomando el grafo G_1, podemos ver que al no ser un grafo hamiltoniano entonces se verifica que $|E_1| = 5 < \frac{1}{2}(5-1)(5-2)+2 = 8$.

1.8. Grafos bipartidos

En esta sección presentaremos los grafos bipartidos.

Definición 1.38. Sea $G = (V, E)$ un grafo. Diremos que G es *bipartido* si el conjunto de vértices V se puede descomponer en dos subconjuntos disjuntos V_1 y V_2 de forma que toda arista incide en un vértice de V_1 y en otro de V_2.

Denominaremos grafo *bipartido completo* a un grafo $G = (V, E)$ que es bipartido, y para cada par $(v_1, v_2) \in V_1 \times V_2$ existe una única arista $e \in E$ de forma que $\gamma_G(e) = \{v_1, v_2\}$.

Cabe destacar que los grafos bipartidos completos están totalmente determinados por el cardinal de la partición $\{V_1, V_2\}$. Así pues, si $G = (V, E)$ es un grafo bipartido completo con $V = V_1 \cup V_2$ y, m y n los respectivos cardinales de V_1 y V_2, entonces denotaremos a dicho grafo como $K_{m,n}$.

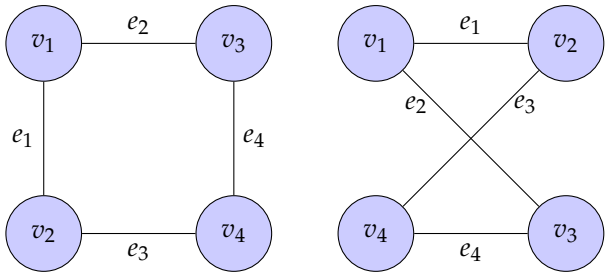


Figura 1.20: Grafo bipartido completo $K_{2,2}$

Ejemplo 1.23. Sea $G = (V, E)$ el grafo de la Figura 1.20 donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4\}$, el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4\}$, y la aplicación de incidencia

viene definida por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_1, v_3\}, \quad \gamma_G(e_3) = \{v_2, v_4\}, \quad \gamma_G(e_4) = \{v_3, v_4\}.$$

Veamos que G es un grafo bipartido completo. Podemos tomar la partición del conjunto de vértices V como $\{V_1, V_2\}$, donde $V_1 = \{v_1, v_4\}$ y $V_2 = \{v_2, v_3\}$. De este modo, observamos que no hay ninguna arista que incida sobre dos vértices distintos de V_1 o V_2, por lo que G es bipartido. Además, podemos comprobar que para cualquier par de vértices $u \in V_1$ y $w \in V_2$ existe una única arista $e \in E$ de forma que $\gamma_G(e) = \{u, w\}$.

Por tanto, G es un grafo bipartido completo. Como el cardinal de V_1 y de V_2 es 2 en ambos casos, podemos denotar a G como $K_{2,2}$.

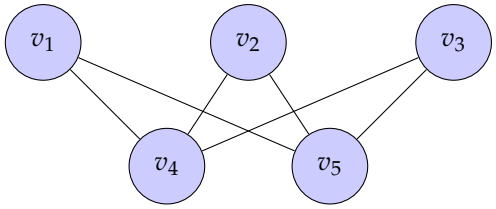


Figura 1.21: Grafo bipartido completo $K_{2,3}$

Podemos apreciar que un grafo bipartido completo tiene $|V| = m + n$ y $|E| = mn$.

En el Ejemplo de la Figura 1.21 el grafo $K_{2,3}$ verifica que tiene $m + n = 2 + 3 = 5$ vértices y $m \cdot n = 2 \cdot 3 = 6$ aristas.

Lema 1.2. *Sea G un grafo bipartido donde $\{V_1, V_2\}$ es una partición del conjunto de vértices V. Supongamos que $v_1 v_2 \cdots v_n$ es un camino en G y $v_1 \in V_1$, entonces $\{v_1, v_3, v_5, \cdots\} \subseteq V_1$ y $\{v_2, v_4, v_6, \cdots\} \subseteq V_2$.*

Lema 1.3. *Sea $G = (V, E)$ un grafo. Entonces G es bipartido si, y sólo si, sus componentes conexas son bipartidas.*

A continuación, enunciaremos un teorema que caracteriza los grafos bipartidos.

Teorema 1.7. *Sea $G = (V, E)$ un grafo. Entonces G es bipartido si, y sólo si, G no contiene ciclos de longitud impar.*

Demostración. [CZ13, Teorema 1.12] Supongamos que $G = (V, E)$ es un grafo bipartido cuya partición es $\{V_1, V_2\}$. Sea $v_1 v_2 \cdots v_n v_1$ un ciclo. Como empieza y acaba por el mismo vértice $v_1 \in V_1$ su longitud tiene que ser par, puesto que en caso contrario el último nodo del camino pertenecería a V_2 y no sería un ciclo. Por tanto, hemos probado que de haber ciclos tienen que ser de longitud par.

Supongamos que $G = (V, E)$ es un grafo que no contiene ciclos de longitud impar. Probemos que G es bipartido. Por el lema anterior tenemos que un grafo es bipartido si, y sólo si, cada una de sus componentes conexas es bipartida, por lo que podemos suponer que G es conexo. Sea $u \in V$. Definimos el conjunto V_1 como aquel compuesto por los vértices $v \in V$ de forma

que la distancia entre u y v sea par, y el conjunto V_2 formado por aquellos cuya distancia con u sea impar.

Así pues, tenemos que $V = V_1 \cup V_2$ y $V_1 \cap V_2 = \emptyset$ por lo que V_1 y V_2 forman una partición de V. Veamos que cualquier arista de G une un vértice de V_1 con uno de V_2.

Supongamos que existen dos vértices v, w adyacentes en V_1 o en V_2. Denotamos $d(u,v) = r$ y $d(u,v) = s$.

- Si $r = s + 1$, entonces una de las distancias es par y otra impar. Por lo que, un vértice pertenece a V_1 y el otro a V_2.
- Si $s = r + 1$, sucede lo mismo que en el caso anterior.
- Si $r = s$, tomamos dos caminos simples de longitud r, $uv_1 \cdots v_r$ y $uw_1 \cdots w_r$, donde $v_r = v$ y $w_r = w$.
  - Si la intersección $\{v_1 \cdots v_r\} \cap \{w_1 \cdots w_r\} = \emptyset$ podemos tomar el ciclo $uv_1 \cdots v_r$ $w_r \cdots w_1 u$ con longitud $2r + 1$, que es impar.
  - Si la intersección de ambos conjuntos de nodos no es vacía tendríamos que eliminar los vértices repetidos. Si $v_i \in \{v_1, \cdots, v_r\} \cap \{w_1, \cdots, w_r\}$, entonces $v_i = w_i$. Esto se verifica puesto que si $v_i = w_j$ con $i \neq j$, entonces $i < j$ ó $j < i$. Si se diese $i < j$, entonces $uv_1 \cdots v_i w_{j+1} \cdots w_r$ es un camino que une u con $w_r = w$ cuya longitud es menor que r, lo cual no posible porque $d(u,w) = r$ es la menor longitud entre u y w. Lo mismo ocurre en el caso de $j < i$. Si tomamos el mayor índice i de forma que $v_i \in \{v_1, \cdots, v_r\} \cap \{w_1, \cdots, w_r\}$, entonces podemos tomar el ciclo $v_i v_{i+1} \cdots v_r w_r w_{r-1} \cdots w_i = v_i$, con lo que obtenemos un ciclo de longitud $2(r - i) + 1$, es decir, un ciclo de longitud impar.

  Como G no contiene ciclos de longitud impar por hipótesis, entonces r y s no pueden coincidir, por lo que obtenemos que cualquier arista de G une un vértice de V_1 con uno de V_2.

De este modo, hemos probado que G es bipartido. □

En la Figura 1.20 el grafo G no contiene ciclos de longitud impar, pues solamente contiene los ciclos: $v_1v_2v_4v_3v_1$, $v_2v_4v_3v_1v_2$, $v_4v_3v_1v_2v_4$ y $v_3v_1v_2v_4v_3$ cuya longitud en todos ellos es 4, que es par. Por tanto, el grafo G por el teorema anterior es bipartido. Del mismo modo, al ser el grafo G bipartido por el Ejemplo 1.23, entonces dicho grafo no puede tener ciclos de longitud impar por el teorema anterior.

1.9. Grafos planos

A continuación, estudiaremos aquellos grafos que se pueden representar en el plano sin que sus aristas se corten.

Definición 1.39. Sea $G = (V, E)$ un grafo. Diremos que G es un grafo *plano* si admite una representación plana, es decir, una representación donde los vértices y las aristas están contenidas en un plano verificando que cada par de aristas distintas no se cortan.

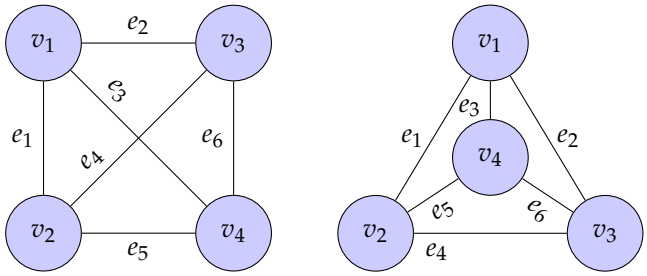


Figura 1.22: Representaciones distintas de un mismo grafo

Ejemplo 1.24. Sea G el grafo definido como el par $G = (V, E)$ donde el conjunto de vértices es $V = \{v_1, v_2, v_3, v_4\}$, el conjunto de aristas es $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, y la aplicación de incidencia viene definida por:

$$\gamma_G(e_1) = \{v_1, v_2\}, \quad \gamma_G(e_2) = \{v_1, v_3\}, \quad \gamma_G(e_3) = \{v_1, v_4\},$$

$$\gamma_G(e_4) = \{v_2, v_3\}, \quad \gamma_G(e_5) = \{v_2, v_4\}, \quad \gamma_G(e_6) = \{v_3, v_4\}.$$

Admite, entre otras, las representaciones que se muestran en la Figura 1.22. La primera representación no es plana puesto que se cortan dos aristas entre sí. Sin embargo, la segunda si lo es. Por tanto, al admitir el grafo G una representación plana podemos concluir que dicho grafo también lo es.

Definición 1.40. Decimos que una *cara* es una región del plano en la que se divide la representación plana de un grafo. El número de caras en las que se divide una representación plana de un grafo se denota por $\alpha(G)$ o α. Podemos denotar cada cara como $C_1, C_2, \ldots, C_\alpha$, y concretamente, llamaremos a la cara C_α como *cara exterior*. Dicha cara es infinita.

En el Ejemplo 1.24, el grafo K_4 en su representación plana está formado por $\alpha(G) = 4$ caras distintas. Dichas caras son las delimitadas por las siguientes aristas:

$$C_1 = \{e_1 - e_3 - e_5\}, \quad C_2 = \{e_2 - e_3 - e_6\}, \quad C_3 = \{e_4 - e_5 - e_6\}, \quad C_4 = \{e_1 - e_4 - e_2\}.$$

C_4 es la cara exterior.

Definición 1.41. Sea $G = (V, E)$ un grafo plano. Diremos que el *grado de una cara* es el número de aristas que delimitan dicha cara. Denotaremos dicho grado por $\mathrm{gr}(C)$.

En el Ejemplo 1.24, las caras de la representación plana del grafo K_4 tienen los grados $\mathrm{gr}(C_i) = 3$, $i \in \{1, 2, 3, 4\}$.

Proposición 1.9. *Sea $G = (V, E)$ un grafo plano conexo tal que $\mathrm{gr}(v) > 1$ para todo $v \in V$. La suma de los grados de las caras del grafo es el doble del número de aristas del mismo, es decir,*

$$\sum_{i=1}^{\alpha} \mathrm{gr}(C_i) = 2|E|.$$

Demostración. Supongamos que para cada cara de la representación plana del grafo, hay $|E|$ aristas que la delimitan, es decir, $\mathrm{gr}(C_i) = |E|$ con $i \in \{1, 2, \ldots, \alpha\}$. Sin embargo, cada arista delimita a su vez a otra cara distinta. Por tanto, cada arista será contada dos veces, una por cada una de las dos caras que bordea. De esta forma, obtenemos que al sumar los grados de las caras, tenemos el doble del número de aristas, es decir, $2 \cdot |E|$. □

En el Ejemplo 1.24, la suma de los grados de las caras del grafo es $\sum_{i=1}^{4} \mathrm{gr}(C_i) = 12$, que es igual al doble del número de aristas que posee el mismo, $2 \cdot |E| = 2 \cdot 6 = 12$.

Lema 1.4. *Sea $G = (V, E)$ un grafo conexo que contiene un ciclo. Entonces si eliminamos una de las aristas que componen el ciclo, entonces el grafo sigue siendo conexo.*

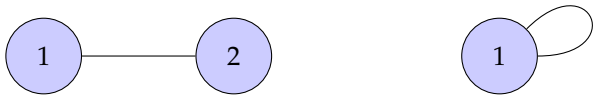
Demostración. Si suponemos que G es conexo y existe un ciclo, podemos suponer que existen dos caminos entre dos vértices distintos, y por tanto si eliminamos una arista de uno de esos dos caminos, el grafo no dejaría de ser conexo puesto que contamos con otro camino que une ambos vértices. □

En 1750, el matemático Leonhard Euler escribió un teorema para poliedros en el que consiguió relacionar el número de vértices, aristas y caras de un grafo plano conexo. Este teorema dio lugar a lo que conocemos actualmente como «*Característica de Euler*».

Teorema 1.8 (Característica de Euler). *Sea $G = (V, E)$ un grafo plano y conexo. Denotamos por v al número de vértices, a al número de aristas y α al número de caras de una representación plana del mismo. Entonces, $v - a + \alpha = 2$. En general, si G es un grafo plano, y χ es el número de componentes conexas, se verifica $v - a + \alpha = 1 + \chi$.*

Demostración. [GM17, Capítulo 6] Realizaremos la demostración por inducción sobre el número de aristas.

Para grafos con una sola arista se verifica el resultado, puesto que solamente se pueden dar las siguientes dos posibilidades.



En el primer caso, $v = 2$, $a = 1$ y $\alpha = 1$ verifican $v - a + \alpha = 2 - 1 + 1 = 2$. En el segundo caso, $v = 1$, $a = 1$ y $\alpha = 2$ también cumplen $v - a + \alpha = 1 - 1 + 2 = 2$.

Supongamos que el resulto se verifica para todos los grafos planos conexos de n aristas, y veamos si se cumple para los aquellos con una arista más.

Sea $G = (V, E)$ un grafo plano conexo con $|E| = n + 1$. Clasificaremos el grafo dependiendo de si tiene al menos un ciclo o no.

- Si G tiene algún ciclo. Denotemos G' al grafo que obtenemos de eliminar una arista de G que formaba parte de un ciclo en cuestión. De esta forma, tenemos que G' sigue siendo un grafo conexo. Denotemos ahora por v', a' y α' al número vértices, aristas y caras respectivamente de este grafo. Puesto que no hemos eliminado ningún vértice, se cumple $v = v'$. Puesto que hemos eliminado una sola arista, se verifica $a = a' + 1$. Y

puesto que hemos quitado una arista de un ciclo, las dos caras que separaba dicha arista se convierten en una sola, por lo que $\alpha = \alpha' + 1$. Así pues, como por hipótesis sabemos que $v - a + \alpha = 2$, entonces $2 = v - a + \alpha = v' - (a' + 1) + (\alpha' + 1) = v' - a' + \alpha'$.

Por lo que hemos obtenido que para $n + 1$ aristas también se verifica.

- Si G no tiene ningún ciclo. Por el Lema 1.1 el grafo G debe tener algún vértice $v \in V$ tal que $\mathrm{gr(v)} = 1$. Denotemos G' al grafo obtenido al quitar a G el vértice v y la correspondiente arista que incide sobre él. Puesto que solamente hemos eliminado un vértice, se verifica $v = v' + 1$. Del mismo modo, como solo hemos eliminado una arista, se cumple $a = a' + 1$. Y puesto que la arista eliminado no separaba ninguna región, se tiene $\alpha = \alpha'$. De esta forma, como por hipótesis sabemos que $v - a + \alpha = 2$, entonces $2 = v - a + \alpha = (v' + 1) - (a' + 1) + \alpha' = v' - a' + \alpha'$.

  Por lo que para $n + 1$ aristas se también cumple. □

En el Ejemplo 1.24, el grafo plano y conexo G tiene 4 vértices, 6 aristas y 4 caras. Por tanto, se verifica $4 - 6 + 4 = 2$.

Nota 1.3. *Un poliedro siempre admite una representación plana.*

En el Ejemplo 1.24 se representa un tetraedro.

Corolario 1.2. *En un poliedro, si denotamos v al número de vértices, a al número de aristas y α al número de caras, entonces $v - a + \alpha = 2$.*

De nuevo, en el caso del tetraedro, al tener 4 vértices, 6 aristas y 4 caras se verifica $4 - 6 + 4 = 2$.

Corolario 1.3. *Sea $G = (V, E)$ un grafo simple, plano y conexo. Entonces, $3\alpha \leq 2a$ y $a \leq 3v - 6$.*

Demostración. Sea G un grafo simple, plano y conexo con α caras. Es claro que al ser G un grafo simple no tiene lazos ni aristas paralelas. Por tanto, el grado de cualquier cara es mayor o igual que 3, es decir, $\mathrm{gr(C_i)} \geq 3$ para cualquier $i \in \{1, 2, \cdots, \alpha\}$.

De esta forma,

$$\sum_{i=1}^{\alpha} \mathrm{gr(C_i)} \geq \sum_{i=1}^{\alpha} 3 = 3\alpha.$$

Por la proposición anterior tenemos lo que queríamos probar, $3\alpha \leq \sum_{i=1}^{\alpha} \mathrm{gr(C_i)} = 2a$.

Por otro lado, obtenemos como consecuencia del Teorema 1.8 la segunda desigualdad,

$$\begin{aligned} 2 = v - a + \alpha &\leq v - a + \frac{2a}{3} = v - \frac{a}{3} \\ 6 &\leq 3v - a \\ a &\leq 3v - 6. \quad \square \end{aligned}$$

Como hemos comprobado en la Figura 1.22 el grafo completo K_4 es plano, puesto que admite una representación plana. Asimismo, podemos observar en la Figura 1.8 que los grafos completos de órdenes 1 hasta 3 también lo son. Sin embargo, esto no ocurre con los grafos completos de todos los órdenes. En concreto, probaremos que los grafos K_5 y $K_{3,3}$ no admiten representación plana.

Proposición 1.10. *El grafo completo K_5 no es plano.*

Demostración. Supongamos que el grafo K_5 es plano. De ser así, por la proposición anterior tendríamos que verificaría $a \leq 3v - 6$, es decir, $10 \leq 3 \cdot 5 - 6 = 9$. Por lo que hemos llegado a una contradicción, y como consecuencia, K_5 no puede ser plano. □

Proposición 1.11. *El grafo bipartido $K_{3,3}$ no es plano.*

Demostración. Al ser el grafo $K_{3,3}$ bipartido, por el Teorema 1.8 tenemos que no existen ciclos de longitud impar y por tanto, no pueden existir caras delimitadas por tres aristas, por lo que como mínimo estarán delimitadas por cuatro aristas.

Partiendo del corolario anterior y suponiendo que toda cara tiene al menos cuatro aristas que son frontera, tenemos $(4-2)a \leq 4(v-2)\ 2a \leq 4(6-2)\ a \leq 8$. Pero esto no puede verificarse puesto que partimos de que $K_{3,3}$ tiene 9 aristas. Por lo tanto, hemos llegado a una contradicción y obtenemos que $K_{3,3}$ no es plano. □

Teorema 1.9 (Kuratowski). *Sea $G = (V, E)$ un grafo. Entonces G no es plano si, y sólo si, existe un subgrafo suyo que se puede contraer a K_5 o a $K_{3,3}$.*

Demostración. Puede encontrarse en [Ede68]. □

1.10. Árboles

En esta sección estudiaremos un tipo especial de grafos, denominados árboles.

El concepto de árbol surgió del estudio del físico alemán Gustav Kirchhoff en 1847, quien empleó la Teoría de Grafos en el cálculo de corrientes en redes eléctricas. No obstante, a día de hoy los árboles han adquirido una gran importancia en muchos campos distintos, desde su uso en problemas de optimización hasta en el estudio de estructuras de datos.

Comenzaremos introduciendo algunos conceptos y definiciones. Posteriormente, nos centraremos en explicar algunos tipos de árboles y algoritmos para encontrarlos.

Definición 1.42. Diremos que un grafo G es un *árbol* si G es conexo y no tiene ciclos.

Definición 1.43. Se denomina *bosque* a un grafo G que no tiene ciclos.

Nota 1.4. *Cada componente conexa de un bosque es un árbol.*

Definición 1.44. Dado un grafo $G = (V, E)$ conexo, y $H = (V', E')$ un subgrafo suyo. Diremos que es H es un *árbol generador* si $V = V'$ y es un árbol.

Nota 1.5. *Un árbol no puede tener lazos ni aristas paralelas.*

Una forma alternativa de definir un árbol es la siguiente.

Proposición 1.12. *Un grafo $G = (V, E)$ es un árbol si, y sólo si, todo par $u, v \in V$ está unido por un solo camino simple.*

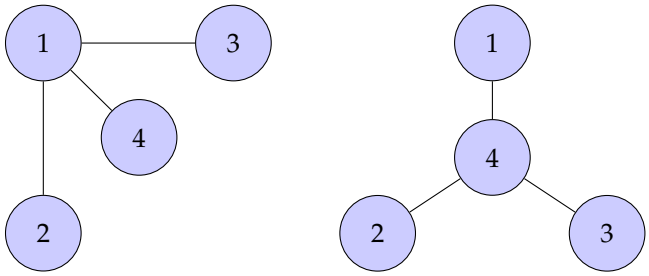


Figura 1.23: Bosque formado por dos árboles

Demostración. Supongamos que $G = (V, E)$ es un árbol. Por la definición de árbol sabemos que G es conexo y no tiene ciclos, por tanto existe un camino entre cualquier par vértices u y v. Supongamos ahora que existen dos caminos distintos entre que ambos vértices, entonces podríamos unir ambos caminos y obtendríamos un ciclo (Proposición 1.3), con lo que llegamos a una contradicción. Por ello, existe únicamente un camino entre cualquier par de vértices de V.

Recíprocamente, si para cualquier par de vértices u y v de V existe un solo camino entre ambos obtenemos que todos sus vértices están conectados y, como consecuencia, el grafo G es conexo. Asimismo, como el camino es único no es posible la existencia de ciclos, por lo que al tener un grafo conexo y sin ciclos concluimos que G es un árbol. □

Sin embargo, ambas definiciones de árbol no presentan una de las características más importante de los mismos. Esta es que este tipo de grafos son los grafos conexos con el menor número de aristas posible. A continuación, se enunciarán otras caracterizaciones que presentan dicha característica.

Proposición 1.13. *Un grafo $G = (V, E)$ es un árbol si, y sólo si, no tiene ciclos y, si añadimos una arista cualquiera, se forma un ciclo.*

Demostración. Supongamos que $G = (V, E)$ es un árbol. Por la definición de árbol sabemos que G es conexo y no tiene ciclos. Veamos ahora que si añadimos una arista cualquiera, se forma un ciclo. Para ello tomamos dos vértices cualesquiera $u, v \in V$ tal que $(u, v) \notin E$. Al ser G conexo, existe un camino c que los conecta en G. Por lo que al añadir la arista (u, v) al grafo se formará un ciclo.

De forma recíproca, supongamos que el grafo G no contiene ciclos y que al añadir una arista cualquiera se forma uno. Supongamos que no es conexo. Entonces tendríamos que existe al menos un par de vértices u y v de forma que no hay un camino entre ambos. Si le añadimos una arista que conecte ambos vértices no obtendríamos ningún ciclo, por lo que llegamos a una contradicción. Por lo que obtenemos que G es conexo y no tiene ciclos, y por tanto, G es un árbol. □

Proposición 1.14. *Un grafo $G = (V, E)$ es un árbol si, y sólo si, es conexo y verifica que, al eliminar una arista deja de serlo.*

Demostración. Vamos a suponer que el grafo $G = (V, E)$ es un árbol. Por su definición sabemos que es conexo y no tiene ciclos. Si eliminamos una arista cualquiera del mismo tenemos que deja de ser conexo puesto que si no ocurriera así, tendríamos dos vértices unidos por dos caminos distintos, y por la Proposición 1.12 tendríamos que G no es un árbol llegando a una contradicción.

Por otro lado, supongamos que G es conexo, y que eliminar una arista implica dejar de ser conexo. Si suponemos que existen ciclos, obtenemos por el Lema 1.4 que el grafo sigue siendo conexo al eliminar una de las aristas del ciclo, por lo que llegamos a una contradicción. Por tanto, obtenemos como consecuencia que no pueden existir ciclos. Al ser G conexo y sin ciclos, tenemos que G es un árbol. □

El siguiente resultado es consecuencia inmediata del lema anterior.

Proposición 1.15. *Dado $G = (V, E)$ un grafo conexo. Entonces G tiene un árbol generador.*

Demostración. Sea G un grafo conexo, si contiene algún ciclo, eliminamos una arista del mismo. Repetimos el proceso hasta obtener un grafo sin ciclos. Dicho grafo sigue siendo conexo puesto que no hemos eliminado aristas que aumenten el número de componentes conexas del grafo. Por lo tanto, tenemos un subgrafo $H = (V', E')$, que al ser conexo y sin ciclos es un árbol. Como se verifica que $V = V'$ y H es un subgrafo de G que a su vez es un árbol, entonces H es un árbol generador de G. □

Proposición 1.16. *Todo árbol es un grafo plano.*

Demostración. Sea $G = (V, E)$ un árbol. Vamos a probar el resultado por inducción sobre el número de aristas, a.

Para el caso $a = 0$, el árbol no tiene aristas, por lo que no pueden cruzarse y por tanto, tenemos que G es plano.

Supongamos ahora que es cierto para a y probaremos que se cumple para $a + 1$.

Partimos de un grafo G conexo, sin ciclos y con $a + 1$ aristas. Si eliminamos una arista nos quedaría un grafo con tamaño a y sin ciclos, que sabemos que admite una representación plana por hipótesis de inducción. Al no tener ciclos, no divide el plano en regiones, de forma que dos vértices distintos pueden unirse por una arista sin cortar ninguna otra. Así pues, cualquier arista que le añadamos no va a cortar a ninguna de las anteriores, dando lugar a un grafo con $a + 1$ aristas que no se cortan entre sí, es decir, dando lugar a un grafo plano. □

Corolario 1.4. *Sea G un grafo conexo de orden n. Entonces dicho grafo es un árbol si, y sólo si, tiene tamaño $n - 1$.*

Demostración. Por un lado, vamos a suponer que G es un árbol. Por la Proposición 1.16 tenemos que dicho árbol es un grafo plano, y el número de regiones en las que se divide el plano es 1. Por el Teorema 1.8 tenemos que $n - l + 1 = 2$, por lo que $l = n - 1$.

Por otro lado, suponemos que tenemos un grafo conexo de orden n y de tamaño $n - 1$. En caso de no ser un árbol, podríamos obtener un árbol generador eliminando aristas (Proposición 1.15), lo que nos daría un árbol de orden n y de tamaño menor a $n - 1$, pero esto no es posible. □

Proposición 1.17. *Sea $G = (V, E)$ un árbol de orden $n \geq 2$. Entonces G tiene al menos dos vértices $u, v \in V$ tal que $\mathrm{gr}(u) = \mathrm{gr}(v) = 1$.*

Demostración. Sea G un árbol de orden $n \geq 2$. Entonces dicho grafo tiene tamaño $n - 1$. Supongamos que no hay vértices de grado 1, es decir, $\mathrm{gr}(\mathrm{v}) \geq 2$ para todo $v \in V$.

Por tanto, por el Teorema 1.1,

$$\begin{aligned} 2a = \sum_{i=1}^{n} \mathrm{gr}(\mathrm{v_i}) &\geq \sum_{i=1}^{n} 2 = 2n, \\ 2(n-1) = \sum_{i=1}^{n} \mathrm{gr}(\mathrm{v_i}) &\geq \sum_{i=1}^{n} 2 = 2n, 2(n-1) = \sum_{i=1}^{n} \mathrm{gr}(\mathrm{v_i}) \geq \sum_{i=1}^{n} 2 = 2n, \end{aligned}$$

donde a es el número de aristas que contiene el grafo.

Sin embargo, hemos llegado a una contradicción, $2n - 2 \geq 2n$.

Por otro lado, si suponemos que hay solamente un vértice de grado 1 tendríamos,

$$2(n-1) = 2a = \sum_{i=1}^{n} \mathrm{gr}(\mathrm{v_i}) = \sum_{\substack{i=1 \\ i \neq j}}^{n} \mathrm{gr}(\mathrm{v_i}) + \mathrm{gr}(\mathrm{v_j}) = 2(n-2) + 1 = 2n - 3.$$

Lo cual no lleva a otra contradicción, por lo que deben de haber al menos dos vértices de grado 1. □

Definición 1.45. Diremos que un árbol es un *árbol con raíz* cuando existe un vértice que se ha designado como raíz. En estos casos, las aristas tienen una orientación desde o hacia la raíz.

Nota 1.6. *En un árbol $G = (V, E)$ con raíz r, el padre de un vértice $v \in V$ es un vértice conectado a v en su camino hacia la raíz. Un hijo de un vértice $v \in V$ es un vértice del cual v es su padre. Un ascendiente de un vértice $v \in V$ es un vértice que es el padre de v o es, de forma recursiva, ascendiente del padre de v. Un descendiente de un vértice v es un vértice que es el hijo de v o es, de forma recursiva, descendiente del hijo de v.*

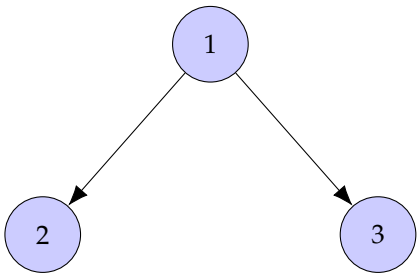


Figura 1.24: Árbol con raíz

En la Figura anterior podemos ver un árbol con raíz, donde vértice designado como raíz es el 1, y presenta dos hijos, los vértices 2 y 3. Respectivamente, notar que el vértice padre de los nodos 2 y 3 es el nodo raíz 1.

1.10.1. Árboles generadores minimales

A continuación, nos vamos a centrar en el estudio de los árboles generadores minimales y dos algoritmos que resuelven el problema de encontrar el árbol generador minimal: el algoritmo de Prim y el de Kruskal.

Definición 1.46. Sea G un grafo conexo ponderado. Un *árbol generador minimal* de G es un árbol generador de forma que la suma de los pesos de sus aristas es la menor posible.

Teorema 1.10. *Todo grafo ponderado contiene algún árbol generador minimal.*

Demostración. Sea $G = (V, E)$ un grafo finito, el número de subgrafos suyos es finito, y más concretamente, el número de árboles generadores es finito. Luego, existe un valor mínimo para la suma de los pesos de sus aristas. □

Una vez definido el concepto de árbol generador minimal nos surge la necesidad de buscar un método para encontrar dicho árbol. Dicho problema es comúmente conocido como «*Minimum Spanning Tree Problem (MST)*» o «*Problema del Árbol Generador Minimal*». Para resolverlo existen numerosos algoritmos, pero nos centraremos en examinar un algoritmo genérico y los algoritmos voraces de Prim y de Kruskal, que se basan en el anterior.

Los algoritmos voraces, también conocidos como «*greedy*», son algoritmos empleados en problemas de optimización. En dichos algoritmos se realiza una secuencia de pasos con diferentes opciones, y en cada paso se selecciona la aparente mejor elección del momento, es decir, una elección localmente óptima, con esperanza de que dicha elección nos lleve a una solución globalmente óptima. Esto no siempre nos garantiza una solución óptima, sin embargo, para el problema del *Árbol Generador Minimal* podemos probar que algunas estrategias voraces sí nos garantizan un árbol generador minimal.

1.10.1.1. Algoritmo genérico

Partimos de un grafo $G = (V, E)$ conexo ponderado con una función peso $w : E \longrightarrow \mathbb{R}$, y queremos encontrar un árbol generador minimal para G.

Este algoritmo genérico sirve como base para ambos algoritmos voraces que estudiaremos a continuación. Una de sus principales diferencias reside en la forma de aplicar el enfoque de este procedimiento.

Dicho método va aumentado el tamaño del árbol en cada paso añadiéndole una arista. Para ello, partimos de un conjunto de aristas A y se tiene que verificar la condición de que, antes de cada iteración, A es un subconjunto de algún árbol generador minimal. A la arista (u, v) que cumple que esta condición, es decir, que al añadirla al conjunto A el conjunto resultante $A \cup \{(u, v)\}$ también es un subconjunto de un árbol generador minimal la denominamos «*arista segura*» o «*safe edge*».

Cuando no encontremos ninguna arista segura que añadir al conjunto A habremos encontrado el árbol generador minimal.

A continuación, se muestra la implementación del algoritmo genérico.

Algorithm 1: MST-GENERICO

```
Data: (G, w)
begin
    A ← ∅
    while A doesn't form a spanning tree do
        find safe edge (u,v) for A
        A ← A ∪{(u, v)}
    end
    return A
end
```

En la primera línea del algoritmo expuesto, el conjunto A satisface la condición previamente mencionada. En el bucle *while* se sigue cumpliendo la condición puesto que solamente añadimos aristas seguras. Por último, todas las aristas que hemos añadido al conjunto A están en un árbol generador minimal, por lo que el conjunto A que devolvemos al final del algoritmo debe ser un árbol generador minimal.

Lo más complicado del proceso es ir encontrando las aristas seguras, pero podemos utilizar dos algoritmos para encontrarlas de forma eficiente.

Para ello debemos definir previamente algunos conceptos.

Definición 1.47. Un *corte* de un grafo $G = (V, E)$ es un partición de V en dos conjuntos. Podemos denotarlo como $(S, V \setminus S)$.

Definición 1.48. Dado un grafo $G = (V, E)$. Diremos que una arista $a \in E$ *cruza* el corte $(S, V \setminus S)$ si uno de sus extremos pertenece a S y el otro a $V \setminus S$. Por otro lado, diremos que un corte *respeta* un conjunto A de aristas si ninguna arista del mismo cruza el corte.

Definición 1.49. Una arista es una «*arista ligera*» o «*light edge*» que cruza un corte, si su peso es menor o igual que el de cualquier otra arista que cruce el corte.

De forma más general, podemos decir que una arista es una arista ligera que satisface una propiedad dada, si su peso es menor o igual que el de cualquier otra arista que cumpla esa propiedad.

Para conocer las aristas seguras usamos el siguiente teorema.

Teorema 1.11. *Sea $G = (V, E)$ un grafo conexo ponderado con una función peso $w : E \longrightarrow \mathbb{R}$. Sea A un subconjunto de aristas de E que está incluido en algún árbol generador minimal para G, $(S, V \setminus S)$ un corte cualquiera del grafo G respetando el conjunto A y, a una arista ligera que cruza $(S, V \setminus S)$. Entonces, la arista a es una arista segura en A.*

Demostración. ([CLRS22], Capítulo 23) Sea T un árbol generador minimal que contiene al conjunto A. Supongamos que la arista (u, v) también está contenida en el árbol T. En ese caso, la demostración se daría por terminada puesto que si añadimos dicha arista al conjunto A, como A ya era un subconjunto de aristas incluido en el árbol T, al añadirle esta última perteneciente al propio T, el conjunto A seguiría contenido en el árbol T. Así pues, (u, v) sería una arista segura en A.

Supongamos ahora que la arista (u, v) no está contenida en el árbol T. Queremos buscar ahora otro árbol generador minimal T' que contenga al conjunto A y a la arista (u, v), mostrando así que (u, v) es una arista segura para A.

Tomamos un camino simple c de u a v, formado por una sucesión de aristas en T. Como los extremos u y v están en los conjuntos S y $V \setminus S$ respectivamente, tiene que haber otro par de vértices del camino escogido en T de forma que la arista que los una también cruce el corte. Denotemos a este par de vértices x y y. Como el corte respeta al conjunto A, entonces la arista (x, y) no pertenece al conjunto A. Asimismo, como (x, y) pertenece al único camino simple de u a v en T, si quitamos dicha arista partimos T en dos partes. Si ahora añadimos la arista (u, v) lo vuelve a unir creando un nuevo árbol generador $T' = (T \cup \{(u, v)\}) \setminus \{(x, y)\}$.

Probemos ahora que T' es un árbol generador minimal. Puesto que la arista (u, v) es una arista ligera que cruza el corte, y (x, y) también lo cruza, evaluando en la función peso tenemos, $w(u, v) \leq w(x, y)$. De esta forma tenemos,

$$w(T') = w(T) - w(x, y) + w(u, v) \leq w(T).$$

Pero al ser T un árbol generador minimal se verifica la otra desigualdad $w(T) \leq w(T')$. Por lo que T' es un árbol generador minimal a su vez.

Por último, vamos a probar que la arista (u, v) es una arista segura en A. Como tenemos que $A \subseteq T$ y $(x, y) \notin A$, entonces $A \subseteq T'$, por lo que $A \cup \{(u, v)\} \subseteq T'$. De esta forma, como T' es un árbol generador minimal, la arista (u, v) es una arista segura en A. □

Corolario 1.5. *Sea $G = (V, E)$ un grafo conexo ponderado con una función peso $w : E \longrightarrow \mathbb{R}$. Sea A un subconjunto de aristas de E que está incluido en algún árbol generador minimal para G. Sea C una componente conexa en el bosque $G_A = (V, A)$. Entonces, si (u, v) es una arista ligera que conecta C a otra componente en G_A, entonces (u, v) es una arista segura en A.*

Los algoritmos de Prim y de Kruskal son casos particulares de este algoritmo genérico. Difieren principalmente en la forma de encontrar una arista segura.

1.10.1.2. Algoritmo de Prim

El Algoritmo de Prim consiste en incrementar el tamaño de un árbol, comenzando por un vértice inicial al cual le vamos añadiendo de forma sucesiva vértices de forma que su distancia a los anteriores sea mínima.

Este funciona de forma similar al algoritmo de Dijkstra [CLRS22, Capítulo 24] para encontrar el camino más corto en un grafo. Sin embargo, la particularidad de este algoritmo reside en la propiedad de que todas las aristas en el conjunto de aristas A del que partimos siempre forman un árbol.

Sea $G = (V, E)$ un grafo conexo ponderado con una función peso $w : E \longrightarrow \mathbb{R}$ y sea A un subconjunto de aristas de E. Si partimos de un árbol con raíz r arbitraria y lo expandimos hasta que se generen todos los vértices de V, entonces, en cada paso añadimos al árbol una arista ligera que conecta el conjunto A a un vértice aislado. Por el corolario anterior, esta regla solamente añade aristas que son seguras en A. Por ello, cuando el algoritmo acaba, las aristas en A forman un árbol generador minimal.

Este algoritmo lo consideramos voraz, como hemos comentado anteriormente, puesto que en cada paso al añadir una arista al árbol lo hace de forma que le añada el menor peso posible al árbol.

En cuanto a la implementación del algoritmo, al ser un método voraz queremos encontrar una forma rápida de seleccionar una nueva arista en cada paso.

Algorithm 2: MST-PRIM

```
Data: (G, w, r)
begin
    for each v ∈ V do
        v.weight ← ∞
        v.parent ← null
    end
    r.weight ← 0
    Q ← V
    while Q ≠ ∅ do
        v ← ExtractMin(Q)
        for each u ∈ G.Adj[v] do
            if u ∈ Q and w(v, u) < u.weigth then
                u.parent ← v
                u.weight ← w(v, u)
            end
        end
    end
end
```

En el algoritmo planteado tenemos como entrada el grafo conexo ponderado G, la función peso w y la raíz r del árbol generador minimal que vamos a generar.

Los vértices que no están en el árbol se encuentran en una cola de prioridad Q. Dichos vértices están etiquetados con un peso y el nombre de su nodo padre. En concreto, dicho peso corresponde al menor peso de una arista que conecte dicho vértice a un vértice del árbol. Por defecto, pondremos que su peso es infinito si no existe ninguna arista con la propiedad anterior.

Al comenzar el algoritmo, ponemos todos los vértices por defecto con peso ∞ y con padre nulo, a excepción del vértice raíz, que tendrá peso 0 para ser el primer vértice que se visita.

Inicializamos la cola de prioridad ordenada de forma ascendente con todos los vértices del grafo G.

A continuación, mientras la cola de prioridad contenga vértices, tomamos el menor de ellos y actualizamos sus etiquetas con los valores reales haciendo uso de la matriz de adyacencia para calcularlos.

Al finalizar el algoritmo, la cola de prioridad queda vaciada y el conjunto A resulta de la forma $A = \{(v, v.father) : v \in V \backslash \{r\}\}$. Las aristas resultantes del conjunto A forman un árbol generador minimal, ya que durante el procedimiento sólo se han añadido aristas seguras en A.

1.10.1.3. Algoritmo de Kruskal

El Algoritmo de Kruskal, al igual que el de Prim, tiene como objetivo encontrar un árbol generador minimal pero en este caso parte de la arista de menor peso, en vez de un nodo raíz.

Además, en este algoritmo el conjunto A es un bosque cuyos vértices son todos aquellos dados por el grafo, y la arista segura que añadimos al conjunto A es siempre la arista de menor peso en el grafo que conecta dos componentes conexas distintas, es decir, dos árboles distintos.

Este algoritmo lo consideramos voraz puesto que, en cada iteración, añadimos al bosque la arista de menor peso posible.

Algorithm 3: MST-KRUSKAL

```
Data: (G, w)
begin
    A ← ∅
    for each v ∈ V do
        Create set(v)
    end
    sort E edges by nondecreasingly weight
    for each edge (u, v) ∈ E by nondecreasingly weight do
        if Find − Set(u) ≠ Find − Set(v) then
            A ← A ∪ {(u, v)}
            Union(u,v)
        end
    end
    return A
end
```

En el algoritmo expuesto, tenemos como entrada el grafo conexo ponderado $G = (V, E)$ y la función peso $w : E \longrightarrow \mathbb{R}$.

Al comienzo del algoritmo, inicializamos el conjunto A como vacío y creamos tantos árboles como vértices hay, cada uno de ellos conteniendo cada vértice. Después, ordenamos de forma no decreciente las aristas del conjunto E por su peso.

A continuación, vamos tomando arista por arista en el orden fijado anteriormente y comprobamos por cada arista (u, v) si sus extremos pertenecen al mismo árbol. Este procedimiento lo podemos realizar a través de la función *«Find-Set(u)»* que se encarga de devolver un elemento representativo del conjunto que contiene al vértice u.

- Si pertenecen al mismo árbol, entonces la arista dada no puede añadirse al bosque sin formar un ciclo. Por lo que descartamos la arista.
- Si no pertenecen al mismo árbol, entonces añadimos la arista al conjunto A y unimos los vértices en los dos árboles para formar una única componente conexa a través de la función *«Union(u,v)»*.

Al acabar el algoritmo, obtenemos el conjunto A compuesto por todas las aristas seguras.

1.11. Grafos de expansión

Los grafos de expansión se han convertido en una herramienta fundamental tanto en matemáticas aplicadas como en informática. Sus aplicaciones son muy variadas, desde su uso en códigos correctores de errores hasta en el estudio de la convergencia de algoritmos de Monte Carlo.

En esta sección, estudiaremos las propiedades de expansión de los grafos que darán lugar a la definición de una familia de grafos de expansión. Para ello, queremos encontrar una invariante cuantitativa que sirva para medir el nivel de conectividad de un grafo y detectar que el grafo no se puede desconectar fácilmente.

Cabe remarcar que la teoría de grafos de expansión ha sido definida fundamentalmente para grafos regulares no dirigidos, es decir, grafos cuyos vértices tienen el mismo grado cuyas aristas tienen dirección. Esto es debido a la simetría que presenta la matriz de adyacencia de este tipo de grafos y al hecho de que todos los valores propios de la matriz sean reales, ya que conlleva un análisis más sencillo. Sin embargo, también vamos a necesitar nociones para grafos dirigidos, por lo que intentaremos dar definiciones generales y extender los resultados a estos últimos.

Definición 1.50. Sea S un subconjunto de vértices del grafo $G = (V, E)$, es decir, $S \subset V$. Sea $\overline{S}$ el complementario de S, es decir, $\overline{S} = V \setminus S$. Dado otro subconjunto de vértices T, denotamos por $E(S, T)$ al conjunto de aristas que comienzan con un vértice de S y acaban con uno de T. Denominamos *límite de borde de* S al conjunto $\delta(S) \coloneqq E(S, \overline{S})$.

A continuación, vamos a definir las constantes de expansión de grafos dirigidos, hacia delante y hacia atrás, dependiendo de si los vértices entran o salen del conjunto S.

Definición 1.51. La *constante de expansión hacia adelante* h^+ y la *constante de expansión hacia atrás* h^- de un grafo $G = (V, E)$ se definen como

$$h^+(G) = \min_{S \subset V, 1 \leq |S| \leq \frac{|V|}{2}} \frac{|\delta(S)|}{|S|}, \quad h^-(G) = \min_{S \subset V, 1 \leq |S| \leq \frac{|V|}{2}} \frac{|\delta(\overline{S})|}{|S|}.$$

Si el grafo es no dirigido, o si el grafo es dirigido verificando $\mathrm{gr}^+(\mathrm{v}) = \mathrm{gr}^-(\mathrm{v})$ para cualquier vértice $v \in V$, entonces $h^+(G) = h^-(G)$. Denotamos por h a dicho valor, y lo denominamos *constante de expansión*. En ocasiones también se puede denominar *constante de Cheeger*. Diremos que la constante es ∞ si G como mucho tiene un vértice. Dicha constante representa el menor radio posible entre el número de aristas que salen de S y el tamaño de S. De esta forma, cuanto más grande sea $h(G)$ más difícil será desconectar un subconjunto grande de V del grafo.

Esto se puede expresar de la siguiente forma.

Proposición 1.18. *Sea $G = (V, E)$ un grafo con $|V| \geq 2$ tal que $h(G) < +\infty$. Entonces $h(G) > 0$ si, y sólo si, G es conexo. Asimismo, si $S \subset V$ verifica $|S| = \alpha|V|$, con $0 < \alpha \leq \frac{1}{2}$, se deben eliminar como mínimo $\alpha h(G)|V|$ aristas de G para desconectar S del resto del grafo.*

Demostración. Supongamos que el grafo G tiene constante de expansión h que verifica $h(G) > 0$. Probemos que entonces G es conexo por contradicción.

Si G no es conexo, entonces existen al menos dos componentes conexas en el grafo G. Por lo que el conjunto de vértices de una de ellas, llamémoslo $S \subset V$, tiene que tener tamaño $|S| < |V|/2$. Puesto que $S \neq \emptyset$ y $\delta(S) = \emptyset$, entonces $h(G) \leq \frac{|\delta(S)|}{|S|} = 0$. Por lo que hemos llegado a una contradicción, ya que por hipótesis $h(G) > 0$.

Supongamos ahora que el grafo G es conexo. Veamos ahora por contradicción que su constante de expansión es mayor que 0.

Si $h(G) = 0$, entonces existe un subconjunto de vértices $\emptyset \neq S \subset V$ de forma que $|U| \leq |S|/2$ y $\delta(S) = \emptyset$. Como $|V \setminus S| \geq 1$, si tomamos $s \in S$ y $v \notin S$, no hay un camino en G entre ambos vértices, ya que un camino tendría que cruzar de S a $V \setminus S$. Por tanto, hemos obtenido que G no es conexo y hemos llegado a una contradicción.

Probemos ahora que si tenemos un subconjunto $S \subset V$ con $|S| = \alpha|V|$ y $0 < \alpha \leq \frac{1}{2}$, entonces se deben quitar al menos $\alpha h(G)|V|$ lados de G para desconectar S del resto del grafo.

Si eliminamos un conjunto de aristas $U \subset V$, entonces S quedará desconectado de $V \setminus S$ si $\delta(S) \subset U$.

Usando la propia definición obtenemos, $|\delta(S)| \geq h(G)|S| = \alpha h(G)|V|$. Por lo que hemos probado lo que buscábamos. □

A continuación, podemos escribir una de las definiciones del concepto de grafo de expansión.

Definición 1.52. Una familia $\{G_n\}_{n\in\mathbb{N}}$ de grafos conexos $G_n = (V_n, E_n)$ es una *familia de grafos expansores*, si existen dos constante $\epsilon > 0$ y $c \geq 1$ independientes de n, de forma que se verifique:

1. El número de vértices $|V_n|$ va aumentando con n.
2. Para cada $n \in \mathbb{N}$, la constante de expansión del grafo G_n cumple $h(G_n) \geq \epsilon > 0$.
3. Para cada $n \in \mathbb{N}$, $\max_{v\in V_n} \mathrm{gr(v)} \leq c$, es decir, el máximo grado de los vértices de V_n está acotado.

En particular, para una familia de grafos regulares de orden k, $\{G_n\}_{n\in\mathbb{N}}$, bastaría probar las dos primeras condiciones.

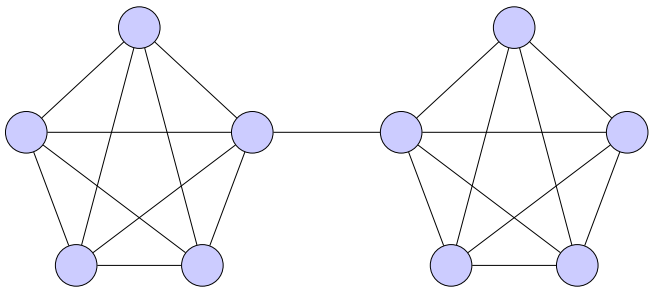


Figura 1.25: Grafo de Expansión

Ejemplo 1.25. Sea $G = (V, E)$ el grafo de la Figura 1.25. Dicho grafo está formado por dos subgrafos completos de orden 5, K_5 y K_5', unidos por una arista e.

Al ser G un grafo no dirigido, podemos calcular la constante de expansión de G de la forma:

$$h(G) = \min_{S \subset V, 1 \leq |S| \leq \frac{|V|}{2}} \frac{|\delta(S)|}{|S|}.$$

Como $1 \leq |S| \leq \frac{|V|}{2} = \frac{10}{2} = 5$, el cociente $\frac{|\delta(S)|}{|S|}$ será mínimo si tomamos como subconjunto de vértices S el conjunto de vértices del subgrafo K_5. Entonces $|S| = 5$ y el límite de borde de S es $\delta(S) = \{e\}$, obteniendo así que $h(G) = \frac{1}{5} \geq \epsilon > 0$.

Este mismo procedimiento se verifica para todo grafo $G_n = (V_n, E_n)$, con $n \in \mathbb{N}$, formado por dos subgrafos completos de orden n, K_n y K'_n, unidos por una arista e_n. Por lo que la constante de expansión de todos ellos viene de acotada por $h(G_n) = \frac{1}{n} \geq \epsilon > 0$.

Por otro lado, para cada $n \in \mathbb{N}$, el tamaño del grafo G_n va aumentando de dos en dos, y se cumple que $max_{v \in V_n} \mathrm{gr(v)} = n$, es decir, que el máximo grado de los vértices de V_n está acotado.

Por tanto, como $\{G_n\}_{n \in \mathbb{N}}$ es una familia de grafos conexos de manera que el número de vértices $|V_n|$ va aumentado con n y, para cada $n \in \mathbb{N}$, verifica que la constante de expansión del grafo G_n es estrictamente mayor que cero y el máximo grado de los vértices de V_n está acotado, entonces es una familia de grafos de expansión.

Cabe mencionar que el *espectro* de un grafo regular de orden k es un conjunto formado por los autovalores de su matriz de adyacencia. A dichos autovalores los denotaremos por $\lambda_1 = k \geq \lambda_2 \geq \cdots \geq \lambda_{|V|}$, donde $\lambda_1 = k$ es el mayor autovalor. El espectro de un grafo contiene mucha información acerca del mismo. Por ejemplo, podemos saber si un grafo es bipartido o conexo a partir de sus valores propios. Por otro lado, la *brecha espectral* es la diferencia entre el sus dos mayores valores propios en valor absoluto, es decir, $\lambda_1 - \lambda_2$. Éste también proporciona información sobre el propio grafo, en particular, proporciona una estimación de su expansión.

Teorema 1.12. *Sea $G = (V, E)$ un grafo no dirigido regular de orden k con espectro $\lambda_1 = k \geq \lambda_2 \geq \cdots \geq \lambda_{|V|}$. Entonces,*

$$\frac{k - \lambda_2}{2} \leq h(G) \leq \sqrt{2k(k - \lambda_2)}.$$

Las desigualdades anteriores se conocen como las *desigualdades de Cheeger*. Este teorema surgió a partir de Cheeger [Che15] y Buser [Bus82]. Sin embargo, fue probado por Dodzniuk [Dod84], Alon-Milman [AM85] y Alon [Alo84] independientemente. A partir de este teorema podemos definir el concepto de expansión de otra forma. Esta definición se conoce como *expansión espectral*.

Dado $G = (V, E)$ un grafo regular de orden k cuyos autovalores son $\lambda_1 = k \geq \lambda_2 \geq \cdots \geq \lambda_{|V|}$, denotemos por $\lambda(G) := \max_{i \neq 1} |\lambda_i|$ al segundo mayor autovalor en valor absoluto de G.

Definición 1.53. Sea $\{G_n\}_{n \in \mathbb{N}}$ una secuencia de grafos regulares de orden k con tamaño creciente. Si existe un $\epsilon > 0$ tal que $k - \lambda(G_n) \geq \epsilon$ para todo n, entonces G_n es una familia de grafos expansores.

1.11.1. Caminatas aleatorias en grafos de expansión

En esta sección estudiaremos a los grafos de expansión desde una perspectiva más estadística, para ello haremos uso del concepto *caminatas aleatorias*.

Recordemos que un camino en un grafo $G = (V, E)$ es una secuencia de vértices $v_0, \cdots, v_n \in V$ de forma que cada par de vértices v_i, v_{i+1} son adyacentes. Cuando se va seleccionando v_{i+1} de forma aleatoria a partir de todos los vecinos de v_i, entonces el camino que obtenemos se denomina *camino aleatorio*. En otras palabras, una *caminata* o *camino aleatorio* (en inglés *Random Walk*) es una trayectoria resultante de realizar pasos aleatorios de forma sucesiva. Dicho concepto fue introducido por Karl Pearson en 1905 [Pea05].

Los grafos de expansión tienen muchas propiedades en común con los grafos aleatorios, y en particular, es muy probable que los grafos aleatorios sean de expansión. De hecho, el conjunto de vértices obtenidos durante una caminata aleatoria en un grafo de expansión es similar al conjunto de vértices obtenidos de forma aleatoria. Asimismo, la distribución de los vértices obtenidos por la caminata aleatoria converge a la distribución uniforme rápidamente.

A continuación presentaremos la relación que hay entre los grafos aleatorios y los grafos de expansión, pero primero, explicaremos de forma más detallada la construcción de las caminatas aleatorias en este tipo de grafos.

El proceso de obtener una caminata aleatoria de longitud α comienza a partir de un grafo de expansión ponderado $G = (V, E)$ con función peso $w : E \longrightarrow [0, 1]$, y una distribución inicial π_0 en V. De ellas, se obtendrá una secuencia de distribuciones de probabilidad π_i en V de forma que:

$$\mathcal{P}[v_i = v] = \pi_i(v),$$

para todo $v_i, v \in V$, $i \in \{0, \ldots, \alpha\}$.

A partir de la función peso $w : E \longrightarrow [0, 1]$, obtenemos una matriz de adyacencia estocástica $A \in [0, 1]^{|V| \times |V|}$. Estas matrices son utilizadas con frecuencia para describir las transiciones de una cadena de Markov [BR97, Capítulo 1] y cumplen la condición de que cada fila es un vector de probabilidad.

Ahora denotaremos por *paso aleatorio* en el grafo, al proceso de multiplicar la matriz de adyacencia A por la distribución inicial, dando lugar a otra distribución

$$\pi_1 := A\pi_0 = A \begin{pmatrix} \pi_0(v_0) \\ \vdots \\ \pi_0(v_{|V|}) \end{pmatrix},$$

con $\{v_0, \ldots, v_{|V|}\}$ los vértices de G. Si realizamos α pasos aleatorios obtendremos una caminata aleatoria de longitud α, $\pi_\alpha = A^\alpha \pi_0$.

Una vez descrita la construcción, presentaremos la conexión entre grafos aleatorios y grafos de expansión. Dicha conexión se conoce como el *Lema de Mezcla de Expansores*.

Lema 1.5. *Sea $G = (V, E)$ un grafo no dirigido regular de orden k. Sea $\lambda = \lambda(G) = \max(|\lambda_2|, |\lambda_{|V|}|)$. Entonces para todos los subconjuntos $S, T \subset V$, se verifica*

$$\left| |E(S, T)| - \frac{k|S||T|}{|V|} \right| \leq \lambda \sqrt{|S||T|}.$$

Como se había definido previamente, el conjunto $E(S,T)$ está compuesto por aquellas aristas que parten del conjunto S y acaban en el conjunto T. Por otro lado, $\frac{k|S||T|}{|V|}$ es el número de aristas esperado entre dos conjuntos de vértices aleatorios. Por tanto, lo que nos muestra el lema es, que si el grafo tiene un espectro de gran tamaño entonces ambos términos no se diferencian demasiado.

1.12. Grafos de Cayley

En este apartado estudiaremos un tipo de grafos que le deben su nombre al matemático británico Arthur Cayley. En 1878 los introdujo con la idea de representar la estructura de un grupo de forma visual.

Generalmente, los grafos de Cayley son grafos dirigidos. Por otro lado, los grafos de Cayley parten de un grupo para su construcción, que como recordamos anteriormente, es una estructura algebraica formada por un conjunto de elementos y una operación interna que cumple las propiedades de existencia de elemento neutro, inverso y asociatividad.

Definición 1.54. Sea G un grupo y S un subconjunto de elementos de dicho grupo. Diremos que $C_{G,S} = (V,E)$ es un grafo de *Cayley* si, V contiene un vértice v_g asociado a cada elemento $g \in G$ y dados $u, v \in G$ el conjunto E contiene una arista de u a v si, y sólo si, $us = v$ para algún $s \in S$.

Para un conjunto S de tamaño k, el grafo de Cayley $C_{G,S}$ es un grafo regular de orden k. Si el subconjunto S es simétrico, es decir, $s \in S$ si, y sólo si, $s^{-1} \in S$, entonces el grafo es no dirigido. Los elementos de S los denominaremos *generadores de grafos*.

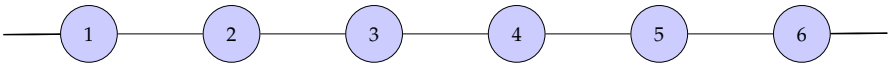


Figura 1.26: Grafo de Cayley no dirigido con $G = \mathbb{Z}, S = \{-1, 1\}$

Ejemplo 1.26. En la Figura. 1.26 se muestra un grafo de Cayley $C_{(G,S)} = (V,E)$ no dirigido, donde el grupo G corresponde con los números enteros con la suma y el subconjunto $S \subset G$ con $S = \{-1, 1\}1$. Si tomamos un vértice $v_i \in V$ asociado al número $i \in \mathbb{Z}$, veamos que dicho vértice va a ser adyacente a v_{i-1} y a v_{i+1}. Para ello, probemos primero que existe algún $s \in S$ de forma que $v_i s = v_{i+1}$. Si tomamos $s = 1$ y tomando la operación interna de la suma del grupo $\mathbb{Z}$, entonces $v_i + 1 = v_{i+1}$. Del mismo modo, tomando $s = -1$, $v_i + (-1) = v_{i-1}$. Por tanto, hemos probado que todo vértice v_i es adyacente al anterior y al siguiente.

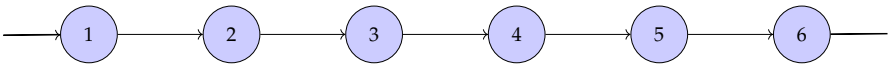


Figura 1.27: Grafo de Cayley dirigido con $G = \mathbb{Z}, S = \{1\}$

Ejemplo 1.27. En la Figura. 1.27 podemos observar un grafo de Cayley $C_{(G,S)} = (V,E)$ dirigido, donde el grupo G es el de los números enteros con la suma, y el subconjunto $S \subset G$ está formado por el elemento 1. Tomando vértice cualquiera $v_i \in V$, asociado al número

$i \in \mathbb{Z}$, queremos probar que dicho vértices es adyacente a v_{i+1}. Para ello, tomamos $s = 1$ y tomando la operación interna de la suma del grupo $\mathbb{Z}$, y obtenemos que $v_i + 1 = v_{i+1}$. Por tanto, hemos probado que un vértice va a ser siempre adyacente al siguiente.

Proposición 1.19. *Sea G un grupo y S un subconjunto simétrico de G. El grafo de Cayley $C_{G,S} = (V, E)$ es conexo si, y sólo si, S es un conjunto generador de G.*

Demostración. Supongamos que el grafo de Cayley $C_{G,S}$ es conexo. Probemos que S genera el grupo G, para ello queremos probar que podemos escribir cada elemento de G como producto de elementos de S.

Como el grafo $C_{G,S}$ es conexo, entonces existe al menos un camino entre cada par de vértices $u, v \in V$. Sea $g \in G$ y sea $1 \in G$ el elemento neutro del grupo G, entonces existe un camino entre 1 y g. Como una arista de u a v del grafo de Cayley verifican $us = v$ para algún $s \in S$, entonces del camino de 1 a g obtenemos que $1s_1 \cdots s_n = g$, con $s_i \in S$ para $i \in \{1, \ldots, n\}$. Por lo que g se puede expresar como producto de elementos de S, $g = s_1 \cdots s_n$, y por tanto, hemos probado que S es un conjunto generador de G.

Supongamos ahora que el grafo de Cayley $C_{G,S}$ cumple que el conjunto S genera al grupo G. Veamos que entonces $C_{G,S}$ es conexo, es decir, que dados dos vértices cualesquiera $u, v \in V$ existe un camino entre ambos.

Por las propiedades de grupo $u^{-1}v \in G$ por lo que al ser S un generador de G, podemos expresar $u^{-1}v$ como producto de elementos de S, $u^{-1}v = s_1 \cdots s_n$. Si tomamos $u(s_1 \cdots s_n) = u(u^{-1}v) = v$, por lo que hay un camino entre ambos, y por tanto, $C_{G,S}$ es conexo. □

El diámetro de los grafos de Cayley de grupos finitos no abelianos es usualmente pequeño. Para grupos finitos simples, presentaremos la siguiente conjetura.

Conjetura 1.1. *(Conjetura de Babai) Existe una constante c de forma que para cualquier grupo finito simple no abeliano G, y cualquier subconjunto simétrico $S \subset G$ que genera G,*

$$\mathrm{d}(C_{G,S}) < (\log |G|)^c.$$

Cabe mencionar que una familia infinita de grupos G_n puede convertirse en una familia de expansores si existe una constante k y un subconjunto generador S_n de tamaño k para cada G_n, de forma que la familia $C_{G,S}$ es una familia de expansores.

2 Análisis de protocolos criptográficos basados en Teoría de Grafos

En este capítulo presentaremos un algoritmo para cifrar y descifrar datos de forma segura basado en Teoría de Grafos [Eta14]. Se trata de un algoritmo de cifrado simétrico que hace uso de los conceptos de grafo completo, ciclo y árbol generador minimal para poder generar mensajes cifrados complejos usando una clave compartida.

La teoría desarrollada hasta ahora nos va a permitir presentar este algoritmo de cifrado cómodamente. Después, realizaremos un ejemplo para ver su funcionamiento y mostraremos una sencilla implementación.

Consideraremos conveniente recordar en primer lugar el concepto de criptografía de clave simétrica de forma breve.

2.1. Criptografía de clave simétrica

La criptografía permite el intercambio seguro de información entre dos partes distintas a fin de evitar que su contenido pueda ser interceptado y leído por una tercera persona no autorizada. Para ello, el emisor se encarga de cifrar o codificar dichos mensajes antes de enviarlos, y el receptor por su parte debe poseer una clave que le permita descifrar la información recibida y poder leer el mensaje original. En otras palabras, la criptografía tiene como objetivo principal la elaboración de códigos y algoritmos de cifrado a fin de proteger la información durante un intercambio.

Entre los primeros métodos de cifrado empleados a lo largo de la historia se encuentra la *criptografía simétrica*, también conocida como *criptografía de clave simétrica*. Su nombre se debe a que este método usa la misma clave tanto para cifrar como para descifrar un mensaje. Por lo que dicha clave debe haber sido compartida antes entre ambas partes de la comunicación de forma segura. Esto puede suponer un problema si deseas realizar un intercambio de información pero no dispones de un medio seguro con anterioridad para realizar el intercambio de clave.

La criptografía de clave simétrica, como hemos mencionado, ha sido empleada a lo largo de los años, por lo que ha ido sufriendo grandes avances a partir de nuevos sistemas de cifrado. Entre ellos destacan la conocida *máquina Enigma* y la *rueda de cifrado* de Jefferson.

Sin embargo, no fue hasta la década de los 70 cuando se propuso un nuevo esquema criptográfico que solventaba el problema del previo intercambio de clave seguro. Diffie y Hellman [Dif76] propusieron un protocolo de intercambio de claves a través de un canal inseguro, que dio lugar a lo que hoy conocemos como *criptografía de clave pública o asimétrica*. En esta, el emisor y receptor usan dos claves diferentes, una privada y una pública.

Frente al claro inconveniente presentado anteriormente sobre los cifrados de clave simétrica, sus principales ventajas residen en su gran rapidez, y su fácil uso, ya que sólo requiere de

una única clave.

Por último, señalaremos brevemente los algoritmos de cifrado simétrico más conocidos y utilizados.

- *Data Encryption Standard (DES).* Fue el primer método de cifrado informático desarrollado por la compañía IBM en 1976 y está basado en el *algoritmo Lucifer* de Horst Feistel. Dicho algoritmo cifra un bloque de 64 bits de texto en un bloque de 64 bits de texto cifrado. Para ello, hace uso de una clave simétrica de 56 bits de longitud y realiza 16 iteraciones de cifrado. Sin embargo, no se considera seguro por el tamaño de la clave de 56 bits ya que es susceptibles a ataques de fuerza bruta.
- *Triple Data Encryption Standard (3DES).* Es el mismo que el algoritmo DES, pero se aplica 3 veces, es decir, se cifran tres veces consecutivas los datos de entrada del bloque usando tres claves distintas de 56 bits. Esta mejora supuso un avance en cuanto a la seguridad del algoritmo, no obstante, existen alternativas mejores que hizo que su popularidad disminuyera con el tiempo.
- *Advanced Encryption Standard (AES).* A fin de encontrar una mejor alternativa a los algoritmos DES y 3-DES, en 1997 se realizó un concurso llamado *Advanced Encryption Standard* donde ganó el algoritmo de *Rijndael*. Es un algoritmo de cifrado basado en bloques de 128 bits, los cuales se organizan en una matriz de orden cuatro por cuatro, donde cada entrada se corresponde con cada byte. A dicha matriz se le aplican una serie de rondas de cifrado según la longitud de sus claves. Este algoritmo permite escoger claves de hasta 256 bits de seguridad, por lo que lo hace más seguro y difícil de romper.

2.2. Descripción del algoritmo

A continuación, se presentará el algoritmo de cifrado propuesto por [Eta14] basado en propiedades de los grafos. Después, se realizará un ejemplo en que se ilustre el funcionamiento de dicho algoritmo y se mostrará una implementación realizada con el lenguaje *python*.

Al ser un algoritmo de cifrado de clave simétrica, es necesario que ambas partes de la comunicación cuenten con una clave compartida. Ésta está compuesta por:

- Una matriz aleatoria K. Dicha matriz debe ser invertible en $\mathbb{Z}$ y debe tener determinante 1 ó -1. La dimensión de dicha matriz se corresponde con la longitud del mensaje a cifrar incrementado en uno. Sin embargo, se podría tomar una dimensión cualquiera y expandirla duplicándola en caso de ser necesario.
- Un carácter inicial especial. Dicho carácter señalará por donde comienza el mensaje.
- Una tabla de codificación, para poder asignarle un código a los distintos caracteres y trabajar con ellos para cifrar y descifrar.

Para cifrar un mensaje, debemos representar los datos de dicho mensaje como vértices de un grafo. Cada carácter se representa por un vértice, y aquellos caracteres que son contiguos se representarán por una arista que una ambos vértices, por lo que serán vértices adyacentes en el grafo. Uniremos también el primer y último carácter para formar un ciclo. Cada arista del grafo tiene su propio peso, que representa la distancia entre los dos caracteres que une en la tabla de codificación.

Después, todos los vértices del grafo se unen con aristas para obtener un grafo completo, y añadimos un carácter adicional como vértice adyacente al primer vértice que señalará por donde comienza el mensaje a cifrar. A partir del vértice añadido, calculamos el peso de la arista que lo une con el anterior, y construimos una matriz de adyacencia M del grafo. A dicho grafo calculamos un árbol generador minimal, y lo representamos como matriz de adyacencia que contiene el orden de los caracteres de datos en su diagonal. Multiplicamos ambas matrices de adyacencia, y multiplicamos la matriz resultante por la matriz de la clave compartida. De esta forma, hemos obtenido una matriz cifrada y el mensaje a enviar será una única línea formada por las filas consecutivas de la matriz cifrada y la matriz de adyacencia M.

Para descifrar el mensaje, operaremos con las dos matrices enviadas y con la matriz K de la clave compartida hasta obtener la matriz de adyacencia del árbol generador minimal, a partir de cuya representación y de la tabla de codificación podremos obtener el mensaje decodificado.

A continuación, presentaremos los pasos de forma más esquematizada.

2.2.1. Algoritmo de cifrado

El algoritmo de cifrado propuesto se compone de los siguientes pasos. Usaremos la misma notación para los grafos que para sus respectivas matrices de adyacencia.

- Añadimos un vértice al grafo por cada carácter del texto a cifrar.
- Unimos los vértices, de aquellos caracteres que son secuenciales en el texto, mediante aristas hasta formar un ciclo. En concreto, uniremos el último carácter con el primero.
- Le asignamos un peso a cada arista usando una tabla de codificación. De esta forma, el peso de cada arista se corresponde con la distancia entre los dos caracteres en la tabla de codificación.
- Añadimos más aristas hasta obtener un grafo completo M. Cada nueva arista que añadimos tiene un peso secuencial partiendo del peso máximo en la tabla de codificación.
- Añadimos un vértice por el carácter especial para indicar el carácter con el que empezaremos.
- Unimos el primer vértice con el vértice asociado al carácter especial y calculamos su peso.
- Calculamos el árbol generador minimal T de M.
- Almacenamos en la diagonal de la matriz T el orden de los vértices. Dicho orden viene dado por la posición que ocupa el carácter asociado a cada vértice en la clave. El carácter especial se corresponderá con la posición 0.
- Multiplicamos la matriz M por T para obtener P.
- Multiplicamos la matriz P por la clave compartida K para obtener C.
- El texto cifrado contiene la matriz C y M fila por fila en una misma línea.

2.2.2. Algoritmo de descifrado

A su vez, el algoritmo de descifrado está formado por los pasos descritos a continuación.

- Calculamos la matriz P multiplicando la matriz C por la inversa de la llave compartida K.
- Calculamos la matriz T multiplicando la matriz inversa de M por la matriz P.
- Obtenemos el texto original decodificando la matriz T usando la tabla de codificación.

2.2.3. Ejemplo

Supongamos el escenario en el que Alicia y Bruno quieren realizar un intercambio de información de manera confidencial. En concreto, Alicia quiere enviarle un mensaje a Bruno de forma que nadie más pueda leerlo. El mensaje que Alicia quiere enviarle a Bruno es el siguiente: «*CLAVE*».

Para ello, Alicia y Bruno cuentan con una clave compartida que sólo ellos conocen. Dicha clave está formada por tres elementos: una matriz aleatoria invertible K, un carácter inicial especial y una tabla de codificación.

Por un lado, denotemos a la matriz de dicha clave por K, y definámosla teniendo en cuenta que al ser la clave de longitud cinco, la matriz debe ser de dimensión 6×6.

$$K = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Por otro lado, supongamos que el carácter inicial especial es la S y la tabla de codificación es la Tabla 2.1:

A	B	C	D	E	F	...	M	N	...	X	Y	Z
1	2	3	4	5	6		13	14		24	25	26

Tabla 2.1: Tabla de codificación

El primer paso que debe realizar Alicia es convertir el mensaje a cifrar en vértices de un grafo. Denotemos dicho grafo por $G = (V, E)$, donde V es el conjunto de vértices y E el conjunto de aristas. El conjunto de vértices va a estar formado por las cinco letras del mensaje «*CLAVE*» (Figura 2.1).

A continuación, por cada par de caracteres secuenciales en el texto a cifrar, se añadirá una arista entre ambos vértices. Además, se unirá el primer y último vértice para formar un ciclo (Figura 2.2).

Para poder asignarle un peso a cada arista, es necesario hacer uso de una tabla de codificación. Alicia hará uso de la tabla compartida con Bruno (Tabla 2.1).

El peso de cada arista corresponde con la distancia entre los caracteres que representan los vértices en la tabla de codificación. En este caso, el peso de la arista que une los vértices C y L se corresponde con la distancia entre los valores C y L en la tabla de codificación.

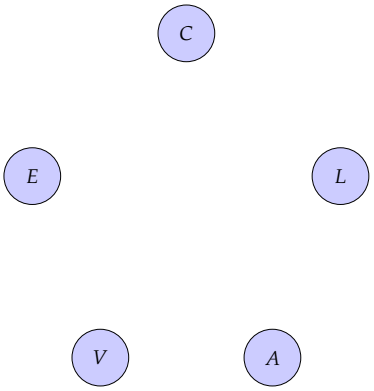


Figura 2.1: Grafo formado por los caracteres

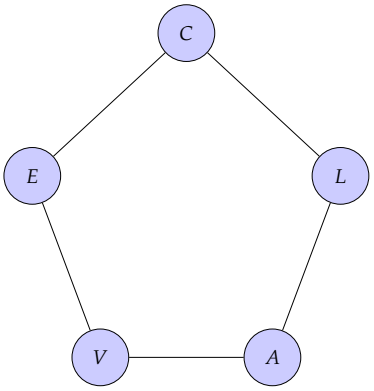


Figura 2.2: Ciclo formado por los caracteres

De este modo, tenemos:

- $w(C, L) = code(L) - code(C) = 12 - 3 = 9$.
- $w(L, A) = 1 - 12 = -11$.
- $w(A, V) = 22 - 1 = 21$.
- $w(V, E) = 5 - 22 = -17$.
- $w(E, C) = 3 - 5 = -2$.

Sustituyendo los pesos calculados en el grafo se obtiene un grafo ponderado (Figura 2.3).

Después se continuará añadiendo aristas hasta conseguir un grafo completo, al que se le asignará su correspondiente peso. En este caso cada uno de las aristas irá tomando un valor secuencial a partir del máximo peso de la tabla de codificación. Obteniendo así un grafo completo de orden 5 (Figura 2.4).

A continuación, será necesario seleccionar un carácter especial antes del primer carácter para señalar por donde comenzar. Tomemos el carácter S (Figura 2.5).

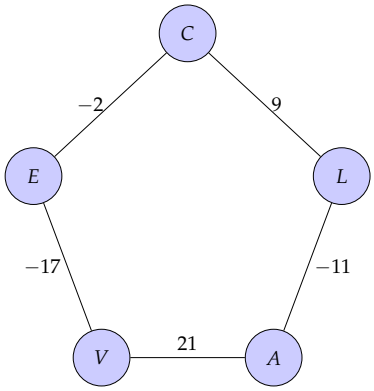


Figura 2.3: Grafo ponderado

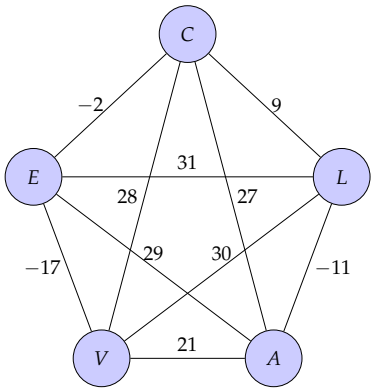


Figura 2.4: Grafo ponderado completo

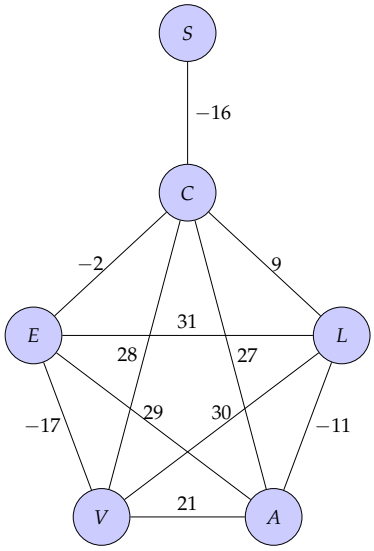


Figura 2.5: Grafo completo con carácter especial

Representamos el grafo obtenido a partir de su matriz de adyacencia M, definida de la forma:

$$M = \begin{pmatrix} 0 & -16 & 0 & 0 & 0 & 0 \\ -16 & 0 & 9 & 27 & 28 & -2 \\ 0 & 9 & 0 & -11 & 29 & 30 \\ 0 & 27 & -11 & 0 & 21 & 31 \\ 0 & 28 & 29 & 21 & 0 & -17 \\ 0 & -2 & 30 & 31 & -17 & 0 \end{pmatrix}.$$

Ahora se debe encontrar el árbol generador minimal (Figura 2.6).

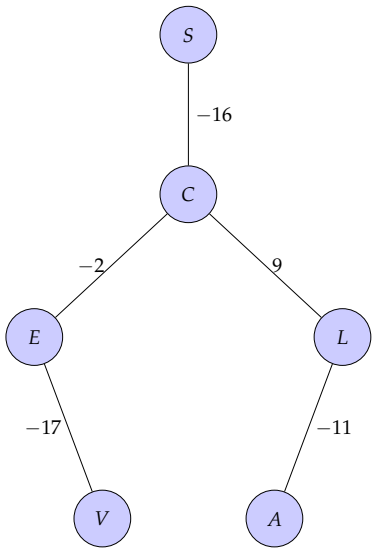


Figura 2.6: Árbol generador minimal

La matriz de adyacencia del Árbol Generador Minimal obtenido, 2.6 es:

$$T = \begin{pmatrix} 0 & -16 & 0 & 0 & 0 & 0 \\ -16 & 0 & 9 & 0 & 0 & -2 \\ 0 & 9 & 0 & -11 & 0 & 0 \\ 0 & 0 & -11 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -17 \\ 0 & -2 & 0 & 0 & -17 & 0 \end{pmatrix}.$$

A continuación, se sustituyen los ceros de la diagonal principal por el orden de los caracteres.

Carácter	S	C	L	A	V	E
Orden	0	1	2	3	4	5

Tabla 2.2: Orden de los caracteres

$$T = \begin{pmatrix} 0 & -16 & 0 & 0 & 0 & 0 \\ -16 & 1 & 9 & 0 & 0 & -2 \\ 0 & 9 & 2 & -11 & 0 & 0 \\ 0 & 0 & -11 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & -17 \\ 0 & -2 & 0 & 0 & -17 & 5 \end{pmatrix}.$$

Una vez obtenidas las matrices M y T, Alicia deberá operar con ellas para obtener la matriz P.

$$P = M \times T = \begin{pmatrix} 256 & -16 & -144 & 0 & 0 & 32 \\ 0 & 341 & -279 & -18 & 146 & -486 \\ -144 & -51 & 202 & -33 & -394 & -361 \\ -432 & -134 & 221 & 121 & -443 & -256 \\ -448 & 323 & 79 & -256 & 289 & -141 \\ 32 & 268 & -299 & -237 & -68 & 293 \end{pmatrix}.$$

A partir de la matriz P y la clave compartida K, se obtiene la matriz cifrada C.

$$C = P \times K = \begin{pmatrix} 256 & 240 & 112 & 256 & 256 & 128 \\ 0 & 341 & -279 & -18 & 146 & -296 \\ -144 & -195 & 58 & -177 & -538 & -781 \\ -432 & -566 & -211 & -311 & -875 & -923 \\ -448 & -125 & -369 & -256 & 289 & -141 \\ 32 & 300 & -267 & -205 & -36 & -11 \end{pmatrix}.$$

Una vez obtenida la matriz C, Alicia deberá enviar las matrices C y M de forma lineal, fila por fila, a Bruno.

Bruno recibe el mensaje cifrado, y debe pasarlo a forma matricial, obteniendo así las matrices C y M.

A partir de la clave compartida K, Bruno puede obtener la matriz P. Para ello es necesario que en primer lugar calcule la inversa de K.

$$K^{-1} = \begin{pmatrix} 1 & -1 & -1 & -1 & -1 & 3 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Una vez calculada la inversa de K, puede obtener la matriz P.

$$P = C \times K^{-1} = \begin{pmatrix} 256 & -16 & -144 & 0 & 0 & 32 \\ 0 & 341 & -279 & -18 & 146 & -486 \\ -144 & -51 & 202 & -33 & -394 & -361 \\ -432 & -134 & 221 & 121 & -443 & -256 \\ -448 & 323 & 79 & -256 & 289 & -141 \\ 32 & 268 & -299 & -237 & -68 & 293 \end{pmatrix}.$$

Suponiendo que la matriz M es invertible. A partir de ésta y de la matriz P, Bruno puede obtener la matriz del árbol generador minimal T.

$$T = M^{-1} \times P = \begin{pmatrix} 0 & -16 & 0 & 0 & 0 & 0 \\ -16 & 1 & 9 & 0 & 0 & -2 \\ 0 & 9 & 2 & -11 & 0 & 0 \\ 0 & 0 & -11 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & -17 \\ 0 & -2 & 0 & 0 & -17 & 5 \end{pmatrix}.$$

Representando ahora el grafo asociado a la matriz de adyacencia T, obviando la diagonal principal (Figura 2.7), Bruno puede descifrar el mensaje original haciendo uso de la tabla de codificación (Tabla 2.1).

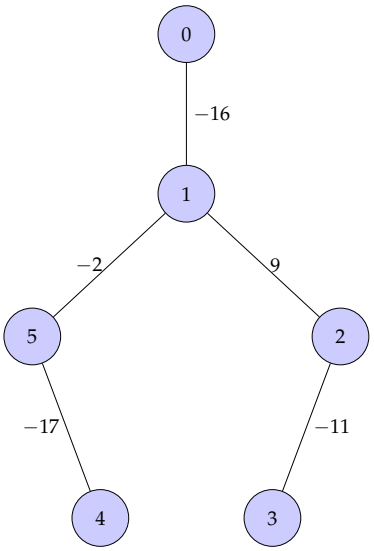


Figura 2.7: Grafo final

Partiendo de que el nodo inicial 0 corresponde con el carácter inicial especial S, y haciendo uso de la tabla de codificación obtiene:

$$nodo1 = code(S) + (-16) = 3,$$

por lo que el nodo 1 representa el carácter C.

Siguiendo el mismo procedimiento, tenemos:

- $nodo2 = code(C) + 9 = 12$. El nodo 2 representa el carácter L.
- $nodo3 = code(L) + (-11) = 1$. El nodo 3 representa el carácter A.
- $nodo5 = code(C) - (-2) = 5$. El nodo 5 representa el carácter E.
- $nodo4 = code(E) - (-17) = 22$. El nodo 4 representa el carácter V.

Por tanto, Bruno ha descifrado el mensaje original *«CLAVE»*.

2.3. Implementación del algoritmo

En esta sección explicaremos las consideraciones que se han tenido en cuenta a la hora de implementar el algoritmo de cifrado y descifrado. Esto incluye, las herramientas que se han utilizado para su implementación, la exposición de las representaciones de los datos y soluciones escogidas, la descripción de las funciones y métodos implementados para el desarrollo del algoritmo, y los resultados experimentales obtenidos.

Antes de comenzar la implementación se ha realizado un análisis del problema y se han determinado las herramientas necesarias para llevarlo a cabo. Después, se ha partido de la entidad principal en el algoritmo: el grafo, y se ha implementado una clase incluyendo la funcionalidad necesaria para las operaciones que necesitamos realizar con él, como el cálculo de árboles generadores minimales. Posteriormente, se ha decidido la representación de los mensajes a cifrar y la clave compartida. Una vez seleccionadas las distintas representaciones de los datos se han procedido a implementar todas las funciones auxiliares necesarias para el algoritmo y por último, se ha implementado el propio algoritmo.

Dicha implementación, incluyendo un tutorial de uso, se encuentra en el siguiente repositorio de GitHub:

`https://github.com/lmd-ugr/grafos-criptografia`.

2.3.1. Herramientas empleadas

Por un lado, el lenguaje de programación empleado ha sido *python*, debido a su gran versatilidad y para poder hacer uso de distintas librerías matemáticas que ofrecen.

Entre las librerías más importantes que se han utilizado se encuentran:

- **math**: para hacer uso de operaciones matemáticas básicas.
- **numpy**: para realizar operaciones sobre matrices, como multiplicaciones, determinantes o inversas.
- **random**: para poder generar de forma aleatoria mensajes que cifrar.
- **time**: para calcular los tiempos de ejecución del algoritmo.

Por otro lado, se ha hecho uso de un cuaderno de *jupyter*, que es un formato que permite presentar el código de manera didáctica favoreciendo la compresión de la implementación del algoritmo.

En dicho cuaderno se presenta la implementación y se muestran varios ejemplos de cifrado y descifrado haciendo uso del mismo. Posteriormente se prueba el algoritmo implementado mediante una serie de ejecuciones para evaluar sus tiempos medios.

2.3.2. Representación de los datos

A continuación, nos centraremos en explicar la representación escogida de los datos. En particular, explicaremos como hemos representado los grafos, la tabla de codificación y los mensajes a cifrar.

2.3.2.1. Representación de los grafos

Para la representación de los grafos se ha creado una sencilla clase llamada *Graph* con los atributos y métodos básicos de grafos para la implementación del algoritmo de cifrado.

La clase *Graph* está compuesta por dos atributos, un constructor y tres métodos.

Los atributos se utilizan para representar al grafo. No se ha considerado necesario representar el grafo mediante su conjunto de vértices y aristas, puesto que para el algoritmo lo que necesitaremos es utilizar las matrices de adyacencia de los grafos para poder operar entre ellas. Sin embargo, sí se ha almacenado el conjunto de vértices para poder obtener un árbol generador minimal mediante el algoritmo de Prim.

Los atributos de la clase *Graph* son:

- **V**: conjunto de vértices.
- **Matrix**: matriz de adyacencia del grafo.

Por tanto, un grafo vendrá representado por su matriz de adyacencia. Se ha elegido esta decisión debido a que haremos uso de dichas matrices para el desarrollo del algoritmo, ya que es necesario realizar operaciones con éstas.

Los métodos implementados se han centrado principalmente en la capacidad para poder obtener un árbol generador minimal a partir del grafo dado. Para obtenerlo se ha implementado el algoritmo de Prim. La elección de dicho algoritmo frente al de Kruskal se debe principalmente a la forma en la que se representan los grafos en la clase implementada. Se ha considerado más sencillo almacenar los vértices del grafo que sus aristas. Por lo tanto, como algoritmo de Kruskal parte de las aristas de menor peso en vez de un nodo raíz, se ha estimado más sencillo utilizar el algoritmo de Prim.

Los métodos desarrollados han sido:

- **Constructor** con parámetros.
- **primMST**: Método para calcular un árbol generador minimal (MST) usando el algoritmo de Prim.
- **minWeight**: Método para encontrar el vértice con menor distancia a cualquier otro vértice no procesado en el MST.
- **printMST**: Método para imprimir el MST.

El método *primMST* ha seguido la estructura del pseudocódigo explicado en el Capítulo 1.

2.3.2.2. Representación de la tabla de codificación

La tabla de codificación es un diccionario donde cada carácter, en este caso letra del abecedario, está asociado a un código numérico. Dicho código es necesario para poder calcular el peso de las aristas.

La elección del diccionario está ligada a la idea de que la tabla de codificación forma parte de la clave compartida por las dos partes de una comunicación. Dicha tabla puede partir de los caracteres que ambas partes consideren y sus códigos pueden tomar los valores que ellos mismos concuerden. Por tanto, dicha tabla puede ser variable a lo largo del tiempo y a través de esta estructura de datos es muy sencillo actualizarlo sin repercutir en el propio algoritmo.

2.3.2.3. Representación de los mensajes

Los mensajes a introducir en el algoritmo de cifrado son cadenas de caracteres. A partir de diversos métodos se transforman dichos caracteres en vértices del grafo.

Por otro lado, los mensajes cifrados son cadenas de caracteres que al introducirlo en el algoritmo de descifrado se someterán a una transformación para poder obtener dos matrices a partir de ellos. Dichos mensajes están formados por las filas de la matriz cifrada C obtenida, concatenadas con las filas de la matriz de adyacencia del grafo inicial M.

2.3.3. Descripción de las funciones implementadas

En este apartado se explicarán los algoritmos de cifrado y descifrado implementados. Además, se detallarán las funciones más importantes que se han usado para su desarrollo.

2.3.3.1. Función para obtener el peso de una arista según la codificación

La función *codificarLetras* calcula el peso de una arista haciendo uso de la tabla de codificación.

En particular, la arista se representa como los caracteres asociados a los vértices que une, por tanto, los datos de entrada son dos cadenas de caracteres formadas por cada uno de ellos.

Como salida se devuelve un entero que se corresponde con el peso de la arista. Dicho peso se ha obtenido como la diferencia entre los valores asociados a los caracteres en la tabla de codificación.

2.3.3.2. Función para obtener el carácter correspondiente a un valor numérico

La función *obtenerLetra* obtiene, dado un valor numérico, el carácter perteneciente al diccionario de la tabla de codificación cuyo código es dicho valor.

Por tanto, como entrada se admite un valor numérico y como salida se obtiene su carácter.

2.3.3.3. Función para decodificar un carácter a partir del peso de una arista

La función *decodificarLetra* obtiene, dado un peso y un extremo de una arista, el carácter correspondiente al otro extremo de la arista.

Se introduce el peso de la arista y el carácter asociado un extremo de la arista, y se obtiene el carácter asociado al otro extremo.

2.3.3.4. Función para obtener un grafo a partir de un mensaje

La función *grafoMensaje* crea un grafo a partir de un mensaje dado y un carácter inicial especial. Para ello almacena el número de vértices y crea un objeto de la clase Graph de dicho tamaño. Después se crea una lista con el peso de las aristas a partir de la función *codificarLetras*. Por último, añade los pesos a la matriz de adyacencia y devuelve el objeto creado.

Por ello, la entrada de la función son dos cadenas de caracteres formadas por el mensaje y el carácter inicial, y como salida se devuelve el grafo asociado a dicho mensaje.

2.3.3.5. Función para sustituir la diagonal principal por el orden de los caracteres

La función *sustituirDiag* sustitye la diagonal principal de una matriz dada por el orden de los caracteres, que en este caso, siempre corresponderá con el número de fila y columna.

La entrada siempre será una matriz dada y la salida la misma matriz modificada.

2.3.3.6. Función para convertir las matrices cifradas en formato lineal

La función *matrizLineal* transforma una matriz dada en una lista formada por las entradas de dicha matriz ordenadas por filas.

2.3.3.7. Función para convertir formato lineal en matrices

La función *linealMatriz* transforma una cadena de caracteres dada en una matriz cuadrada cuyos elementos se corresponde con las entradas de la lista ordenados por filas.

2.3.3.8. Función para descifrar un mensaje de un MST

La función *descifrarMatriz* que dada la matriz de adyacencia de un árbol generador minimal y un carácter inicial obtiene el mensaje descifrado haciendo uso de una tabla de codificación.

Para ello se parte del carácter inicial y se hace uso de la función *decodificarLetra* para obtener el primer carácter del mensaje descifrado. Recorriendo las entradas de la matriz se obtienen del mismo modo el resto de caracteres hasta obtener el mensaje descifrado completo.

2.3.3.9. Algoritmo de cifrado

La función *cifrar* transforma una cadena de caracteres con el mensaje inicial en una lista con el mensaje cifrado. Dicha función realiza todos los pasos del algoritmo de cifrado haciendo uso de las funciones auxiliares explicadas anteriormente.

Como entrada se introduce una cadena de caracteres con el mensaje a cifrar y dos componentes de la clave compartida: un carácter inicial especial y una matriz. Como salida obtenemos una lista con el mensaje cifrado formada por las entradas de la matriz cifrada C y la matriz de adyacencia M, ordenadas por filas.

A continuación, se muestra dicha función explicando cada uno de los pasos que se han seguido y las funciones que ha empleado.

```
# Algoritmo de cifrado dado un mensaje, un caracter inicial y
# una matriz perteneciente a la clave compartida
def cifrar(mensaje, carac, K):
    """
        'mensaje' texto plano al que convertir en grafo
        'carac' caracter especial para indicar el caracter inicial del mensaje
        'K' matriz de clave compartida
    """

```

```
    # 1. Construimos un grafo g cuyos vertices son los caracteres del mensaje
    # y el caracter especial
    g = grafoMensaje(mensaje, carac)
    M = g.matrix

    # 2. Comprobamos si la matriz M es invertible
    if np.linalg.det(M) != 0:

        # 3. Encontramos la matriz de adyacencia del arbol generador minimal
        # (MST) del grafo g usando el Algoritmo de Prim
        T = g.primMST()

        # 4. Sustituimos los ceros de la diagonal principal por el orden de los
        # caracteres
        sustituirDiag(T)

        # 5. Multiplicamos la matriz de adyacencia del grafo, M, y la del arbol
        # generador minimal, T, para obtener la matriz P
        P = np.dot(M,T).tolist()

        # 6. Usamos la clave K para cifrar P
        C = np.dot(P,K).tolist()

        # 7. Pasamos las matrices C y M a formato lineal
        lista1 = matrizLineal(C)
        lista2 = matrizLineal(M)
        mensaje_cifrado = lista1 + lista2

        return mensaje_cifrado
```

2.3.3.10. Algoritmo de descifrado

La función *descifrar* transforma una lista con el mensaje cifrado en una cadena de caracteres con el mensaje original. Dicha función realiza todos los pasos del algoritmo de descifrado haciendo uso de las funciones auxiliares comentadas.

Como entrada se introduce una lista con el mensaje cifrado y dos componentes de la clave compartida: un carácter inicial especial y una matriz. Como salida se obtiene el mensaje original sin cifrar.

A continuación, se muestra dicha función explicando en detalle los pasos seguidos y las funciones que han sido llamadas.

```
# Algoritmo de descifrado dado un mensaje, un caracter inicial
# y una matriz perteneciente a la clave compartida
def descifrar(mensaje_cifrado, carac, K):
    """
        'mensaje' texto plano al que convertir en grafo
        'carac' caracter especial para indicar el caracter inicial del mensaje
        'K' matriz de clave compartida
    """
    # 1. Dividimos el mensaje cifrado en dos listas y lo transformamos en dos
    # matrices, C_desc y M_desc respectivamente.
    lista1_desc = mensaje_cifrado[0:int(len(mensaje_cifrado)/2)]
    lista2_desc = mensaje_cifrado[int(len(mensaje_cifrado)/2):len(mensaje_cifrado)]

    C_desc = linealMatriz(lista1_desc)
    M_desc = linealMatriz(lista2_desc)
```

```
16
17      # 2. Calculamos la inversa de la clave K
18      K_i = np.linalg.inv(K).tolist()
19
20      # 3. Obtenemos la matriz P a partir de la matriz C descifrada y la
21      # inversa de la clave K
22      P_desc = np.dot(C_desc,K_i).tolist()
23
24      # 4. Calculamos la inversa de la matriz M
25      M_i = np.linalg.inv(M_desc).tolist()
26
27      # 5. Calculamos la matriz T a partir de la inversa de M
28      T_desc = np.round(np.dot(M_i,P_desc)).tolist()
29
30      # 6. Representando el arbol generador minimal T, y usando
31      # la tabla de codificacion obtenemos el mensaje cifrado.
32      mensaje_desc = descifrarMatriz(T_desc,carac)
33
34      return mensaje_desc
```

2.3.4. Resultados experimentales

Al final del cuaderno de jupyter se han realizado unas ejecuciones para probar los algoritmos de cifrado y descifrado implementados.

Como hemos comentado anteriormente, ha sido necesario hacer uso de las librerías *random*, *time*. También se ha usado la librería *string* para generar mensajes aleatorios.

En esta sección se han implementado dos nuevas funciones para generar mensajes y claves de forma aleatoria:

- **generarK**: Función que, dado un tamaño n, genera una matriz de dimensiones $n \times n$ que formará parte de la clave compartida. En particular, se generan matrices triangulares superiores con sus elementos no nulos igual 1. Se han elegido este tipo de matrices por simplicidad para las ejecuciones, ya que sabemos que son invertibles en $\mathbb{Z}$ y tienen determinante 1 ó -1.
- **generarMensaje**: Función que, dado un tamaño n, genera una cadena de caracteres aleatoria de dicha longitud.

Después, se generan 55 mensajes aleatorios distintos a cifrar, y se ejecutan los algoritmos 30 veces por cada mensaje. En dichas ejecuciones se obtiene los tiempos de cifrado y descifrado, y se realizan las medias por cada mensaje.

Los 55 mensajes aleatorios generados tienen longitud creciente, es decir, el primer mensaje tiene tamaño 3 y el último 57.

Los tiempos de cifrado obtenidos no son eficientes para un algoritmo de cifrado realista. Gran parte de estos tiempos elevados se deben a las operaciones de matrices de gran tamaño, ya que se deben realizar muchas multiplicaciones, determinantes e inversas de matrices.

A continuación, se expone una tabla comparando los resultados obtenidos con los del artículo propuesto.

Tamaño del mensaje a cifrar	Tiempo (ms)			
	Algoritmo de cifrado	Algoritmo de descifrado	Ambos algoritmos	Algoritmo propuesto
4	0.38	0.54	0.92	15
6	0.68	0.95	1.63	18
12	1.83	2.36	4.19	41
19	4.59	6.05	10.64	105
28	13.26	21.17	34.43	199
56	83.26	170.4	252.66	947

Los experimentos de la segunda y tercera columna han sido realizados en un mismo ordenador portátil con las siguientes características: Sistema operativo Ubuntu 20.04.5, arquitectura *x*86_64, procesador Intel(R) Core(TM) *i*7-7500U CPU @ 2.70GHz y con una memoria RAM de 16GB de tipo DDR4.

Podemos observar en la tabla que los resultados obtenidos a partir de la implementación realizada han sido mejores que los obtenidos en la implementación de [Eta14]. Sin embargo, en el artículo no hay más referencias a dicha implementación que las de los tiempos obtenidos y las características del ordenador en el que se ejecutó, por lo que no podemos realizar un estudio exhaustivo de las posibles diferencias que han llevado a esta mejora más allá del incremento de CPU y memoria RAM de mi ordenador respecto al suyo.

2.4. Conclusiones y comentarios sobre el algoritmo

El algoritmo propuesto por [Eta14] es un algoritmo teóricamente interesante debido a que usa propiedades de la teoría de grafos para cifrar. Sin embargo, presenta diversas limitaciones y flaquezas en distintos ámbitos. A continuación, se comentarán todos los inconvenientes y restricciones que se han encontrado a lo largo del estudio e implementación del algoritmo.

- En primer lugar, el algoritmo propuesto no ha contemplado muchos aspectos matemáticos a tener en cuenta antes de aplicar el algoritmo. Pese a haber comentado en la descripción del algoritmo que la matriz K de la clave compartida debe de ser invertible en $\mathbb{Z}$ y con determinante 1 ó -1, esto no se ha comentado en el algoritmo original. Es necesario que la matriz aleatoria K cumpla dichas propiedades para poder descifrar un mensaje, ya que se tiene que calcular la inversa de dicha matriz en el proceso.
- Asimismo, debemos tener en cuenta que la matriz K de la clave compartida tiene que tener un tamaño que depende de la longitud del mensaje que quieres enviar. Por tanto, una misma matriz no nos sirve como clave para enviar mensajes de distinto tamaño. No es realista tener que disponer de una matriz por cada tamaño o tener que hacer llegar dicha matriz cada vez que se quiera cifrar un mensaje usando este algoritmo. En [Eta14] se propone como solución tomar una dimensión cualquiera y expandirla duplicándola en caso de ser necesario. Sin embargo, esto supondría un mayor tiempo de ejecución al tener que realizar las comprobaciones y transformaciones adicionales en ambos procesos, a no ser que trabajes por bloques con un tamaño fijo.
- Entre sus principales inconvenientes, reside el hecho de que a partir de un mensaje concreto no podemos asegurarnos de que al crear el grafo mediante el algoritmo obtengamos una matriz de adyacencia M invertible. Hay muchos casos en los que no

se obtiene una matriz invertible. Por ejemplo, en el caso de los mensajes compuesto por dos letras su matriz siempre va a ser singular. En casos de mensajes con longitud mayor a dos hay también muchos ejemplos, como la palabra *leer*. En otros casos, la matriz es invertible pero no en $\mathbb{Z}$, por lo que tenemos que redondear su producto con la matriz P para obtener la matriz T, lo cual conlleva un mayor tiempo de ejecución, o trabajar con aritmética de múltiple precisión racional.

- Otro problema que se ha encontrado ha sido el de notación. Al obtener un grafo ponderado, si queremos calcular su matriz de adyacencia debemos tener el cuenta el valor a tomar cuando no existe una arista entre un par de vértices. El número tomado en el artículo es el 0, no obstante, esto presenta un problema, puesto que no podemos diferenciar cuándo hay una arista que une dos vértices que representan a una misma letra, como es el caso de la palabra *coordenada*. Si calculamos el peso entre el segundo vértice O y el tercer vértice O, obtenemos que su valor es 0 independientemente de la tabla de codificación empleada. Por tanto, esto supone un problema en mensajes donde hay letras repetidas y contiguas.
- Cabe notar que la tabla de codificación propuesta solo contempla letras mayúsculas. Si se quisiera enviar un texto completo con espacios, signos de puntuación, letras mayúsculas y minúsculas, números y otros caracteres especiales, debería de hacerse uso de una tabla de mayor tamaño más completa.
- No es un algoritmo de gran eficiencia debido a sus largos tiempos de ejecución. Puesto que estamos trabajando con matrices que pueden llegar a tomar un gran tamaño, el tiempo empleado en operaciones como multiplicación de matrices es muy elevado, haciendo imposible el uso de este algoritmo para procesos de transmisión en directo.
- También es interesante remarcar que la clave compartida está formada por tres componentes que pueden presentar gran tamaño, por lo que es necesario que, previo a la comunicación, se haya producido el intercambio seguro de clave.

En cuanto a los aspectos positivos del algoritmo se encuentra principalmente la seguridad. Al ser un algoritmo que cuenta con una clave compartida formada por tres componentes distintas, entre ellas cuenta con una matriz aleatoria, si se toma una matriz de un tamaño considerablemente grande es computacionalmente difícil poder descifrar un mensaje sin conocerla.

Por tanto, a modo de conclusión podemos decir que el algoritmo propuesto no es un algoritmo efectivo ni eficiente ya que no nos garantiza poder cifrar cualquier mensaje y presenta un largo tiempo de ejecución. Sin embargo, desde un punto teórico es un algoritmo de interés al hacer uso de distintos conceptos matemáticos en el ámbito de la teoría de grafos y sus propiedades.

3 Funciones hash criptográficas basadas en grafos

El término *función resumen*, también conocido como *función hash* (del inglés, "picar y mezclar"), ha sido empleado en el área de la informática a lo largo de los años, y hace referencia a una función que comprime una cadena de entrada arbitraria en una cadena de longitud fija. Sin embargo, si dicha función cumple ciertas propiedades adicionales, que pueden ser de gran utilidad en aplicaciones criptográficas para dotarlas de seguridad, se conocen como *funciones hash criptográficas*.

En este capítulo presentaremos este tipo de funciones junto a las propiedades que deben verificar y a sus principales aplicaciones. Asimismo, haciendo uso de la teoría de grafos previamente desarrollada, se estudiarán las *funciones hash de expansión*, que consisten en definir una función hash criptográfica a partir de un grafo de expansión dado. De esta forma, podremos interpretar importantes propiedades de las funciones hash como propiedades de los grafos. Es conveniente comentar también algunos conceptos de seguridad en el ámbito de la criptografía.

Por cuestiones de notación, se podrá hacer referencia a las funciones hash criptográficas simplemente como funciones hash o resumen.

En la elaboración de este capítulo se ha hecho uso de [Kow19] y [SG12], siguiendo principalmente la estructura de [Pet09].

3.1. Funciones hash criptográficas

En esta sección estudiaremos el concepto de este tipo de funciones y su principal diferencia con las funciones hash estándar. Asimismo, se estudiarán algunas definiciones de seguridad en la criptografía, métodos para construir funciones hash, ataques populares a este tipo de funciones, y sus aplicaciones principales.

Definición 3.1. Una *función hash criptográfica* es una función, $H : \{0,1\}^* \longrightarrow \{0,1\}^\lambda$, que toma cadenas de bits de longitud arbitraria como entrada y produce una cadena de bits de longitud fija como salida. La salida se suele denominar *resumen*, *código hash* o *valor hash*.

Las funciones hash pueden depender o no de una clave.

- **Funciones hash sin clave**. Las funciones hash sin clave también conocidas como *Manipulation Detection Code (MDC)*, producen como salida un valor hash que solamente depende de la cadena de entrada.
- **Funciones hash con clave**. Las funciones hash con clave conocidas como *Message Authentication Code (MAC)*, producen como salida un valor hash que depende tanto de la cadena de entrada como de una clave secreta.

En general, el término función hash se refiere a funciones hash sin clave [SG12].

3.1.1. Seguridad

A la hora de definir requerimientos de seguridad de funciones hash, como la *resistencia a colisiones*, y poder hacer uso de expresiones o definiciones de seguridad formalmente, como *computacionalmente difícil*, es necesario definir una familia de funciones $\{H_n\}_{n\geq 0}$ que se encuentren *parametrizadas por un parámetro de seguridad* n:

$$H_n : \{0,1\}^* \longrightarrow \{0,1\}^{\lambda(n)},$$

para alguna función $\lambda(n)$.

En teoría, una función hash se define como una familia de funciones hash con una clave criptográfica secreta,

$$H_n : \{0,1\}^* \times \{0,1\}^{\mu(n)} \longrightarrow \{0,1\}^{\lambda(n)},$$

para algunas funciones $\lambda(n)$ y $\mu(n)$. Aunque no siempre es necesaria una clave para establecer pruebas de seguridad en protocolos, son necesarias para definir formalmente nociones como la resistencia a colisiones. Sin embargo, pese a que a la mayoría de definiciones se aplican a funciones resumen con clave, actualmente la mayor parte de los algoritmos prescinden de ella y se definen simplemente para varios valores de un parámetro de seguridad.

Cabe notar que existen dos enfoques principales para la seguridad computacional: el enfoque concreto y el enfoque asintótico.

- **El enfoque concreto.** Las definiciones de seguridad están parametrizadas por números concretos. Algunos protocolos serán (e, t, m)-seguros si cualquier algoritmo que se ejecuta en un tiempo menor que t y usa menos memoria que m, satisface una tarea con una probabilidad menor que e.
- **El enfoque asintótico.** Las definiciones de seguridad se aplican a una familia de funciones hash $\{H_n\}$ indexada por un parámetro de seguridad n en vez de a una función simple H. La familia $\{H_n\}$ es segura si cada H_n es $(e(n), t(n), m(n))$-segura. Las funciones $t(n)$ y $m(n)$ no crecen demasiado rápido, mientras que $e(n)$ decrece lo suficientemente rápido con n.

Una vez definidas las familias de funciones parametrizadas por un parámetro de seguridad, podemos definir formalmente expresiones como *computacionalmente difícil*, *algoritmo eficiente* y *probabilidad despreciable*. Dichas definiciones se han extraído de [KL20] y [Pet09].

Definición 3.2. Un algoritmo A se dice *eficiente* o de *tiempo probabilístico polinomial (PPT)* si existe un polinomio p de forma que para cada entrada $x \in \{0,1\}^*$ la ejecución de $A(x)$ termina como mucho en $p(|x|)$ pasos, donde $|x|$ es la longitud de x.

Definición 3.3. Una función f se dice que es *despreciable* si para todo polinomio p existe un m de forma que para todos los enteros $n \geq m$, se cumple que $f(n) < \frac{1}{p(n)}$.

Definición 3.4. Diremos que una función f es *perceptible* si existe un polinomio p tal que para todos los enteros n suficientemente grandes, se cumple que $f(n) > \frac{1}{p(n)}$.

Definición 3.5. Un cálculo se dice que es *computacionalmente difícil* de resolver si no puede ser resuelto de forma eficiente, es decir, en tiempo polinomial.

Definición 3.6. Un cálculo se dice que es *computacionalmente inviable* cuando a pesar de ser computable, requiere demasiados recursos para calcularse realmente.

Definición 3.7. Un sistema criptográfico se dice que es *computacionalmente seguro* si el mejor algoritmo para descifrarlo requiere n operaciones, donde n es un número a partir del cual es computacionalmente inviable realizar tantas operaciones.

En la actualidad podemos asumir que 2^{128} es un número computacionalmente inviable de operaciones, por lo que un valor n mayor implicaría que el sistema es computacionalmente seguro. Sin embargo, nunca puede saberse con certeza si un sistema es computacionalmente seguro ya que no podemos saber si existe un algoritmo mejor que el conocido. Por lo que en la práctica, usamos el mejor algoritmo conocido hasta el momento.

Es de notable interés el tamaño de las claves en cuanto a la seguridad. Si el tamaño de clave es demasiado pequeño, es posible que se posean los suficientes recursos computacionales para romper el sistema a través de fuerza bruta.

No obstante, con el paso del tiempo es probable encontrar algoritmos mejores que irán dejando atrás a los mejores conocidos previamente. Del mismo modo, con el avance computacional, las capacidades de los dispositivos serán cada vez mayores pudiendo variar el umbral del número de operaciones a partir del cual es inviable realizarlas.

El esfuerzo computacional que requiere resolver un problema está altamente relacionado con la complejidad de dicho problema. Por lo que presentaremos brevemente las dos clases de complejidad más destacables: la clase $\mathcal{P}$ y la clase $\mathcal{NP}$. La clase $\mathcal{P}$ es la clase de problemas para los que existe un algoritmo eficiente, es decir, que los resuelve en un número acotado de pasos respecto a la longitud de entrada. Por otro lado, los problemas $\mathcal{NP}$ son aquellos para los que se puede comprobar en tiempo polinomial si una solución candidata es realmente una solución.

La clase de problemas $\mathcal{P}$ está contenida en $\mathcal{NP}$, es decir, $\mathcal{P} \subseteq \mathcal{NP}$. Sin embargo, saber si dicha inclusión es estricta o si se da la igualdad es uno de los problemas más importantes sin resolver de la informática. En concreto, existe otra clase de complejidad denominada clase $\mathcal{NP}$-completo que está compuesta por todos aquellos problemas $\mathcal{NP}$ tal que cualquier otro problema de $\mathcal{NP}$ se puede reducir a ellos. Estos problemas son aquellos problemas de $\mathcal{NP}$ que no han conseguido probar que son de clase $\mathcal{P}$. Si se demostrase que un problema $\mathcal{NP}$-completo se puede resolver en tiempo polinomial, todos los problemas de $\mathcal{NP}$ podrían también resolverse en tiempo polinomial. Si por el contrario, se demostrase que un problema $\mathcal{NP}$-completo no se puede resolver en tiempo polinomial, entonces los demás problemas de $\mathcal{NP}$-completos tampoco podrían también resolverse en tiempo polinomial.

Una vez definidos los conceptos de seguridad necesarios, nos centraremos en explicar las propiedades que cumplen las funciones hash criptográficas y las que los diferencian de las estándar.

3.1.2. Propiedades de la funciones hash criptográficas

Las funciones hash criptográficas son un tipo de funciones hash. La principal diferencia entre ambas reside en que las funciones hash criptográficas deben cumplir las siguientes propiedades.

- **Resistencia a cálculo de preimágenes**. Diremos que una función hash H es resistente a preimágenes si dado un valor hash v es computacionalmente inviable encontrar una entrada u en el dominio de H, es decir, $H(u) = v$.
- **Resistencia débil a colisiones**. Diremos que una función hash H presenta resistencia débil a colisiones si dada una entrada u del dominio H es difícil encontrar otra entrada $v \neq u$ que de lugar a la misma salida, es decir, que $H(u) = H(v)$.
- **Resistencia fuerte a colisiones**. Diremos que una función hash H presenta resistencia fuerte a colisiones si es difícil encontrar dos entradas que den lugar a la misma salida, es decir, dos entradas u y v del dominio de H de forma que $H(u) = H(v)$.

La propiedad de resistencia a cálculo de preimágenes es más débil que las otras dos, y la propiedad de resistencia débil a colisiones preimágenes lo es respecto a la resistencia fuerte a colisiones. Sin embargo, puede parecer que las dos últimas son similares. Para diferenciarlas presentaremos la conocida *paradoja del cumpleaños*.

3.1.2.1. Paradoja del cumpleaños

La *paradoja del cumpleaños* establece que de un conjunto de 23 personas la probabilidad de que al menos dos personas compartan día de cumpleaños es mayor del 50 % (concretamente del 50.7 %). Si partimos de un conjunto de 58 personas o más esta probabilidad supera el 99 %.

Estrictamente, el problema anterior no es una paradoja, puesto que no es una contradicción lógica, sino que simplemente contradice a la intuición. Esto se debe a que cuando se pide una estimación sobre el tamaño del conjunto de personas de forma que, la probabilidad de que dos de ellas cumplan años el mismo día sea mayor o igual que $\frac{1}{2}$, la respuesta intuitiva es 183, es decir, la mitad de 365.

Para probar dicho problema demostraremos un caso más general.

Tenemos una función hash $H : \{0,1\}^* \to \{0,1\}^n$. Por lo tanto, hay 2^n posibles valores hash distintos. Denotaremos por m a la cardinalidad del codominio.

Asumimos que todos los elementos del codominio tienen aproximadamente la misma probabilidad de ser imágenes de alguna entrada l.

El problema se puede escribir como: ¿cuántos elementos hay que elegir para que la probabilidad de que haya una colisión sea mayor o igual que $\frac{1}{2}$?

Para resolver el problema vamos a calcular primero la probabilidad de que no haya una colisión.

Si elegimos un solo elemento, la probabilidad de que no haya una colisión es de 1.

Si elegimos dos elementos, la probabilidad es

$$p = \frac{m-2+1}{m} = 1 - \frac{1}{m}.$$

Si elegimos un tercer elemento, la probabilidad se multiplica por $1 - \frac{2}{m}$,

$$p = \left(1 - \frac{1}{m}\right)\left(1 - \frac{2}{m}\right).$$

Siguiendo el mismo procedimiento hasta haber elegido k elementos, tenemos que la probabilidad de que no haya colisiones es,

$$p = \left(1 - \frac{1}{m}\right)\left(1 - \frac{2}{m}\right)\cdots\left(1 - \frac{k-1}{m}\right).$$

Una vez calculada dicha probabilidad, lo que haremos es aproximar cada factor usando series de Taylor [Lan98, Capítulos 13 y 14].

De acuerdo a las series de Taylor, la función exponencial se puede escribir como,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Cuando el valor x es considerablemente pequeño, la función exponencial se puede aproximar a sus dos primeros términos de la serie,

$$e^x \approx 1 + x.$$

Por tanto, aplicando la aproximación de la función exponencial por series de Taylor,

$$e^{\frac{-1}{m}} \approx \left(1 - \frac{1}{m}\right).$$

Por lo que la probabilidad p resulta,

$$p \approx e^{\frac{-1}{m}} \dot{e}^{\frac{-2}{m}} \cdots e^{\frac{-k+1}{m}}.$$

Una vez calculada la probabilidad de que no haya una colisión, veamos cuánto tiene que valer k para que la probabilidad p sea menor o igual que $\frac{1}{2}$,

$$\begin{aligned} e^{\frac{-1}{m}} \dot{e}^{\frac{-2}{m}} \cdots e^{\frac{-k+1}{m}} &\leq e^{\frac{1}{2}} \\ e^{\frac{-1}{m} + \frac{-2}{m} + \cdots + \frac{-k+1}{m}} &\leq e^{\frac{1}{2}}. \end{aligned}$$

Tomando logaritmo neperiano obtenemos,

$$\frac{-1}{m} + \frac{-2}{m} + \cdots + \frac{-k+1}{m} \leq \ln\left(\frac{1}{2}\right) = -\ln(2).$$

Cambiamos de signo,

$$\begin{aligned} \frac{1}{m} + \frac{2}{m} + \cdots + \frac{k-1}{m} &\geq \ln(2) \\ \frac{(k-1)k}{2m} &\geq \ln(2) \\ k^2 - k &\geq 2m\ln(2) = m\ln(4). \end{aligned}$$

Como $\sqrt{m} = \sqrt{2^n}$, entonces tenemos,

$$k \geq \sqrt{m \ln(4)} = \sqrt{\ln(4)} \cdot 2^{n/2} \approx 1.18 \cdot 2^{n/2},$$

y por tanto, k tiene que ser aproximadamente mayor o igual que $1.18 \cdot 2^{n/2}$ para que exista una colisión.

Como hemos supuesto que la colisión puede ser entre dos elementos cualesquiera distintos de los k elegidos, entonces la probabilidad que hemos calculado se corresponde con la resistencia fuerte a colisiones.

En particular, para la paradoja del cumpleaños, consideremos como dominio al conjunto de personas, y como valores hash la fecha de su cumpleaños. Supongamos que nadie ha nacido el 29 de febrero, por lo que la cardinalidad del codominio es $m = 365$. Veamos cuántas personas deben haber para que la probabilidad de que dos de ellas cumplan años el mismo días sea mayor que $\frac{1}{2}$.

Sustituyendo en la aproximación obtenida, obtenemos

$$k \geq \sqrt{m \ln(4)} \approx 22.49,$$

por lo que, deben haber al menos $k = 23$ personas distintas.

Si quisiéramos calcular cuántas personas deben haber para que la probabilidad de que dos de ellas tengan el mismo cumpleaños sea mayor o igual que el 99 %, tendríamos entonces,

$$\begin{aligned}
\frac{-1}{m} + \frac{-2}{m} + \cdots + \frac{-k+1}{m} &\leq \ln\left(\frac{1}{100}\right) = -\ln(100) \\
\frac{1}{m} + \frac{2}{m} + \cdots + \frac{k-1}{m} &\geq \ln(100) \\
\frac{(k-1)k}{2m} &\geq \ln(100) \\
k^2 - k &\geq 2m \ln(100) \\
k &\geq \sqrt{2m \ln(100)}.
\end{aligned}$$

Por tanto, sustituyendo en la ecuación,

$$k \geq \sqrt{2m \ln(100)} \approx 57.98,$$

deben haber al menos $k = 58$ personas distintas.

Estudiemos ahora el siguiente problema: dado un elemento del codominio u con valor hash $H(u)$, ¿cuántos elementos distintos hay que elegir para que la probabilidad de que su valor hash coincida con el de u sea mayor o igual que $\frac{1}{2}$?

Al igual que para el problema anterior, vamos a calcular primer la probabilidad de que no haya colisión.

Si tomamos solo un elemento, la probabilidad de que no colisione con u es

$$p = \frac{m-1}{m}.$$

Si tomamos dos elementos, la probabilidad se multiplica por $1 - \frac{1}{m}$

$$p = \left(\frac{m-1}{m}\right)\left(\frac{m-1}{m}\right).$$

Si tomamos k elementos, la probabilidad que obtenemos es

$$p = \left(\frac{m-1}{m}\right)^k.$$

Una vez hemos calculado la probabilidad de que u no colisione con otra entrada, veamos cuánto debe valer k para que la probabilidad sea menor o igual que $\frac{1}{2}$,

$$p = \left(\frac{m-1}{m}\right)^k \leq \frac{1}{2}.$$

Tomamos logaritmo neperiano,

$$k \ln\left(\frac{m-1}{m}\right) \leq \ln\frac{1}{2} = -\ln 2.$$

Buscamos ahora una aproximación de

$$\ln\left(\frac{m-1}{m}\right) = \ln\left(1 - \frac{1}{m}\right),$$

usando series de Taylor.

La función $f(x) = \ln(1+x)$ se puede escribir como,

$$f(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} x^n = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots$$

Cuando el valor x es notablemente pequeño, la función f se puede aproximar al primer término de la serie,

$$f(x) \approx x.$$

Por lo que, si aplicamos la aproximación por series,

$$k\frac{-1}{m} \leq -\ln 2.$$

Cambiando de signo y despejando el valor k obtenemos,

$$k \geq \ln 2 \cdot m = \ln 2 \cdot 2^n \approx 0.69 \cdot 2^n,$$

y concluimos que k tiene que ser aproximadamente mayor o igual que $0.69 \cdot 2^n$ para que u colisione con otra entrada.

Como hemos supuesto que la colisión entre u y un elemento cualquiera distintos de los k

elegidos, entonces la probabilidad que hemos calculado se corresponde con la resistencia débil a colisiones.

Para la paradoja del cumpleaños, dada una persona, veamos cuántas personas distintas deben haber para que la probabilidad de que alguna de ellas coincida fecha de cumpleaños con la primera sea mayor o igual que $\frac{1}{2}$.

Para ello, sustituimos $m = 364$ en la aproximación obtenida,

$$k \geq 364 \ln 2 \approx 252.31,$$

y tenemos que deben haber al menos otras $k = 253$ personas distintas.

Si de nuevo quisiéramos calcular el número de personas necesario para que dicha probabilidad sea mayor o igual que el 99 %, tendríamos,

$$\begin{aligned} k \ln\left(1 - \frac{1}{m}\right) &\leq \ln \frac{1}{100} = -\ln 100 \\ k\frac{-1}{m} &\leq -\ln 100 \\ k &\geq \ln 100 \cdot m \approx 4.61 \cdot m. \end{aligned}$$

Sustituyendo en la última ecuación obtenemos,

$$k \geq \ln 100 \cdot 364 \approx 1676.28.$$

Por tanto, se necesitarían unas $k = 1677$ personas distintas.

De esta forma, haciendo uso de la paradoja del cumpleaños, hemos podido diferenciar las colisiones débiles de las fuertes.

3.1.3. Construcción Merkle-Damgård

La *construcción Merkle-Damgård* o la *función hash Merkle-Damgård* es un método en criptografía empleado para construir funciones hash criptográficas resistentes a colisiones a partir de funciones de compresión unidireccionales resistentes a colisiones, es decir, a partir de funciones que transforman dos entradas de longitud fija en una salida de longitud fija y que a su vez son resistentes a colisiones.

La construcción de Merkle-Damård fue propuesta por Ralph Merkle [Mer79] [Mer90] en 1979. Posteriormente, Ivan Damgård [Dam89] probó que presentaba una estructura sólida.

La función hash Merkle-Damgård aplica en primer lugar una función de relleno compatible con ella para crear una entrada de tamaño múltiplo de un número fijo. Se debe aplicar esta función de relleno puesto que las funciones de compresión no permiten manejar entradas de tamaño aleatorio. Después de haber creado dicha entrada, se divide el resultado en bloques de tamaño fijo y los procesa uno a uno con la función de compresión, combinando un bloque de la entrada con la salida de la iteración anterior.

Para garantizar la seguridad de la construcción, Merkle y Damgård propusieron el *fortalecimiento Merkle-Damgård*, que consiste en rellenar los mensajes con un relleno capaz de codificar la longitud del mensaje original.

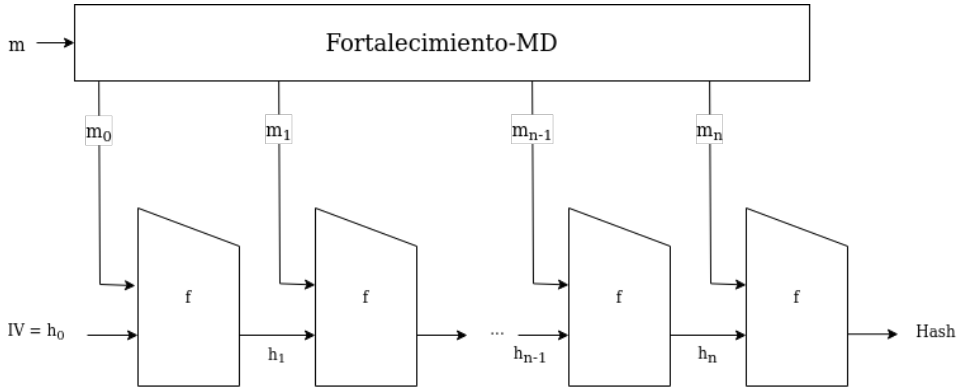


Figura 3.1: Esquema de la construcción de Merkle-Damgård

En el esquema anterior se denota por f a la función de compresión unidireccional, y se rellena y transforma el mensaje m a través del fortalecimiento de Merkle-Damgård para poder dividirlo en $n+1$ partes de igual tamaño. El algoritmo comienza con un valor inicial fijo denominado *vector de inicialización (IV)*. Para cada bloque m_i con $i \in \{0, \cdots, n\}$, la función de compresión f toma el resultado anterior h_i y lo combina con el bloque actual para obtener el resultado intermedio h_{i+1}. Cuando llegamos a la última iteración, el resultado obtenido al aplicar f será el propio hash. En algunas ocasiones, se puede aplicar la *función de finalización* al último resultado obtenido. Esto se aplica para comprimir en un tamaño aún menor el hash.

3.1.4. Funciones hash más conocidas

Hay que tener en cuenta que no existe una única función resumen, sino que hay muchas distintas. Entre las familias más conocidas se encuentran:

- **SHA (Secure Hash Algorithm).** La familia de funciones hash criptográficas SHA fue publicada por el Instituto Nacional de Estándares y Tecnología (NIST) como estándar federal de procesamiento de información (FIPS) de Estados Unidos [Sta95]. Esta familia está formada por seis funciones: *SHA-0*, *SHA-1*, *SHA-224*, *SHA-256*, *SHA-384* y *SHA-512* [Sta95] [Sta]. En las últimas cuatro funciones, cada uno de sus nombres representa la longitud en bits del código a obtener. Actualmente, por motivos de seguridad no se utiliza *SHA-0* ni *SHA-1*, sino las versiones más recientes. Esto se debe a que en 2017, un grupo de investigadores de Google y del CWI (Certifified Welding Inspector) de Ámsterdam realizaron el primer ataque por colisión a *SHA-1*. Dicha colisión supone la posibilidad de que puedan existir dos documentos cuyo valor hash es el mismo. Esto puede permitir a atacantes cambiar dos ficheros sin levantar sospechas.
- **MD (Message Digest)**. La familia MD fue diseñada por Ronald Rivest . Está compuesta por las funciones *MD2*, *MD4*, *MD5* y *MD6*. *MD5* [Riv92] ha sido una de las más populares, empleada para garantizar la integridad de los archivos transferidos. Sin embargo, en 2004 se encontraron colisiones, y a causa de un ataque reportado en tan solo una hora, *MD5* dejó de ser recomendado para uso.
- **RIPEMD (RACE Integrity Primitives Evaluation Message Digest)**. La familia de funciones RIPEMD fue diseñada a partir de los principios usados en *MD4* por Antoon

Bosselaers, Hans Dobbertin y Bart Preneel. Está compuesta por *RIPEMD*, *RIPEMD-128*, *RIPEMD-160*, *RIPEMD-256* y *RIPEMD-360* [DBP96].

3.1.5. Aplicaciones de las funciones hash criptográficas

Las funciones hash criptográficas son una de las herramientas más importantes en el campo de la criptografía y tienen numerosas aplicaciones para proporcionar seguridad, entre ellas destacan la autenticidad e integridad de mensajes, la gestión de contraseñas y las firmas digitales entre otras.

3.1.5.1. Integridad y autenticidad de mensajes

Verificar la integridad y autenticidad de la información es de gran importancia en redes y sistemas informáticos. Por ejemplo, si nos encontramos en el caso donde dos partes se tienen que comunicar a través de un canal que no es seguro, deben de requerir de un método por el cual el receptor pueda comprobar que la información enviada es auténtica o no ha sido modificada.

Las mejores alternativas para implementar la autenticación e integridad de mensajes son métodos basados en cifrado simétrico o funciones MAC [SG12].

3.1.5.2. Gestión de contraseñas

Las funciones hash proveen de protección para el almacenamiento de contraseñas. En vez de almacenar la contraseña en sí, almacenan su valor hash, por lo que un fichero de contraseñas consiste en una tabla formada por el par (usuario, H(contraseña)). De esta forma, una persona ajena solamente puede ver los hashes de las contraseñas, y por la propiedad de resistencia a cálculo de preimágenes no puede obtener la contraseña a partir del hash.

3.1.5.3. Firmas digitales

Una firma digital es un mecanismo criptográfico que pretende lograr los siguientes dos objetivos:

- Asegurar al receptor que, mediante los mensajes firmados digitalmente, el mensaje proviene del remitente señalado. Esto nos proporciona *autenticidad*.
- Asegurar al destinatario que el mensaje no se modificó de forma involuntaria o maliciosa durante la comunicación. Esto nos proporciona *integridad*.

El concepto de firma digital no surgió hasta la década de los 70, cuando Diffie y Hellman definieron dicho concepto a partir de la necesidad de una firma que dependiera del mensaje para evitar problemas entre emisores y receptores. Definieron la firma digital como un conjunto de datos que van asociados a un mensaje de forma que comprueban la autenticidad de quien firma y la integridad del mensaje.

Unos años después, Rivest, Shamir y Adleman, del Instituto Tecnológico de Massachusetts (MIT), dieron lugar al algoritmo *RSA* [RSA78], cuyo nombre corresponde a las iniciales de

sus desarrolladores. Dicho algoritmo estaba basado en clave pública y fue muy empleado para la firma digital.

Posteriormente, el algoritmo *DSA (Digital Signatura Algorithm)* [Nis92] se convirtió en el estándar para firma digitales en Estados Unidos. Hasta la actualidad han ido avanzando los algoritmos para firmas digitales precisando una mayor seguridad.

Las funciones hash de por sí no implementan el objetivo de seguridad que proporcionan las firmas digitales, pero se utilizan para poder optimizar el tamaño de la firma digital. Concretamente, el remitente solo firmará digitalmente el hash del mensaje que se quiera autenticar. Sin las funciones hash la firma digital sería del mismo tamaño que el mensaje.

3.2. Funciones hash de expansión

La idea de construir funciones hash haciendo uso de grafos de expansión surge de Zémor [Zém91] [Zém94] y Tillich [TZ93] [TZ94]. Aproximadamente unos diez años más tarde y de forma independiente, Charles, Lauter y Goren [CLG09] volvieron a estudiar dicha idea.

A raíz de algunas propiedades de grafos podemos interpretar propiedades de funciones hash. Por ejemplo, partiendo de la propiedad de expansión de los grafos podemos obtener que los valores hash de mensajes largos están casi uniformemente distribuidos, o que a partir de caminos y ciclos del grafo podemos obtener información acerca de la resistencia a cálculo de preimágenes y colisiones de la función hash.

En esta sección, haciendo uso de la teoría sobre grafos y funciones hash desarrollada previamente, se explicará la construcción de las funciones hash de expansión a partir de grafos de expansión y grafos de Cayley. Además, se estudiarán las propiedades e interpretaciones que derivan de dichas construcciones, al igual que la seguridad que poseen y los posibles ataques particulares que pueden sufrir dichas funciones.

3.2.1. Construcción a partir de grafos de expansión

Para construir una función hash de expansión es necesario partir de un grafo de expansión regular, una arista o un vértice inicial, y una función de ordenación.

Explicaremos dos algoritmos de construcción de funciones hash de expansión distintos diferenciando entre grafos dirigidos y no dirigidos.

En primer lugar, trataremos el caso en el que partimos de un grafo dirigido. Sea $G = (V, E)$ un grafo de expansión, regular de orden k y dirigido. Sea $v_0 \in V$ el vértice inicial y sea $\sigma : V \times \{0, \ldots, k-1\} \longrightarrow V$ una función de ordenación de forma que el conjunto $\{\sigma(v, i) : i \in \{0, \ldots, k-1\}\}$ contiene todos los vértices que son adyacentes a v.

Dado un mensaje m, supongamos que el algoritmo hash del que partimos transforma el mensaje dado en α dígitos entre 0 y $k-1$ de la forma $m = m_0 \ldots m_{\alpha-1}$, donde $m_i \in \{0, \cdots, k-1\}$. Después, se realizan de forma sucesiva $\alpha - 1$ pasos, desde $i = 0$ hasta $i = \alpha - 2$, donde se irá calculando $v_{i+1} := \sigma(v_i, m_i)$. En el último paso, obtenemos como resultado $v_{\alpha-1}$.

Los cálculos realizados dan lugar a un camino formado por los vértices $v_0, \cdots, v_{\alpha-1}$, donde el valor hash es el último obtenido, es decir, $v_{\alpha-1}$.

Ahora trataremos el caso en el que el grafo es no dirigido. Este caso es bastante similar al de los grafos dirigidos, diferenciándose esencialmente en la función de ordenación y en partir de una arista inicial en vez de un vértice. El motivo principal del cambio de función es evitar el backtracking. Nombremos ahora $l = k - 1$. Sea $G = (V, E)$ un grafo de expansión, regular de orden l y no dirigido. Sea $e_0 \in E$, con $e = (v_1, v_2)$, una arista inicial y sea $\sigma : E \times \{0, \ldots, l-1\} \longrightarrow V$ una función de ordenación tal que el conjunto $\{\sigma(e, i) : i \in \{0, \ldots, l-1\}\} \bigcup \{v_1\}$ está compuesto por todos los vértices que son adyacentes a v_2.

Al igual que para los grafos dirigidos, el algoritmo hash transforma un mensaje m en α dígitos comprendidos entre 0 y $l - 1$, de forma que $m = m_0 \cdots m_{\alpha-1}$, con $m_i \in \{0, \cdots, l-1\}$. A continuación, se realizarán de forma sucesiva $\alpha - 1$ pasos, desde $i = 0$ hasta $i = \alpha - 2$, donde calculará $v_{i+1} \coloneqq \sigma(e_i, m_i)$ y $e_{i+1} \coloneqq (v_i, v_{i+1})$. Por último, se obtiene como resultado $v_{\alpha-1}$.

Como en el caso anterior, a cualquier cálculo de un hash le corresponde un camino en el grafo formado por los vértices $v_0, \cdots, v_{\alpha-1}$, donde el último vértice del camino es el valor hash obtenido.

Cabe notar que en ambos casos los vértices deberían ser transformados posteriormente a cadenas de bits de forma inyectiva, pero por comodidad lo obviaremos en esta explicación.

Es evidente una clara relación entre la construcción de los algoritmos hash explicados en ambos casos y la construcción Merkle-Damgård. De hecho, podríamos tomar la función de compresión f como

$$f(v_i || m_i) \coloneqq \sigma(v_i, m_i)$$

en el caso del grafo dirigido, y

$$f((v_i, v_{i+1}) || m_{i+1}) \coloneqq (v_{i+1}, \sigma((v_i, v_{i+1}), m_{i+1}))$$

en el caso del grafo no dirigido, donde $||$ es el operador de concatenación.

Las funciones de compresión construidas son resistentes a colisiones en caso de que no haya aristas paralelas, puesto que éstas son biyectivas. Por otro lado, si el grado de todos los vértices del grafo es de tamaño pequeño, entonces las funciones de compresión pueden no ser resistente a preimágenes.

Puesto que una colisión para una función de ordenación puede ser fácilmente trasladada a otra distinta, supondremos que, al no aportar ninguna seguridad adicional, la función de ordenación es fija.

Por otra parte, el elegir de forma aleatoria un vértice o arista inicial no siempre puede aportar seguridad adicional. Por ejemplo, en el caso de las funciones hash de Cayley puede dar lugar a un posible ataque si algunos son más débiles que otros. Además, para este tipo de grafos en especial es recomendable elegir de forma aleatoria los parámetros del grafo.

3.2.2. Construcción a partir de grafos de Cayley

Para construir una función hash de expansión a partir de un grafo de Cayley (a la que llamaremos también *función hash de Cayley*) distinguiremos dos casos, dependiendo de si el grafo es dirigido y no dirigido.

Primero, estudiaremos el caso en el que el grafo es dirigido. Sea $C_{G,S}$ un grafo de Cayley dirigido, donde G es un grupo y S un subconjunto de G no simétrico, es decir, que para cualquier $s \in S$, $s^{-1} \notin S$. Ordenamos los elementos de S de la forma: $S = \{s_0, \ldots, s_{k-1}\}$. Sea $g_0 \in G$ el vértice inicial y sea

$$\begin{aligned} \sigma : G \times \{0, \ldots, k-1\} &\longrightarrow G, \\ (g, i) &\longmapsto g s_i, \end{aligned}$$

la función de ordenación.

Dado un mensaje m, suponemos que el algoritmo hash divide el mensaje en α dígitos de la forma $m = m_0 \cdots m_{\alpha-1}$, donde los dígitos están en base k. El valor hash del mensaje m es

$$H(m) = g_0 \prod_{i=0}^{\alpha-1} s_{m_i} \coloneqq g_0 s_{m_0} \cdots s_{m_{\alpha-1}}.$$

A continuación, estudiaremos el caso en el que el grafo es no dirigido. Sea $C_{G,S}$ un grafo de Cayley no dirigido, donde G es un grupo y S un subconjunto de G simétrico, es decir, que para cualquier $s \in S$, $s^{-1} \in S$. Los elementos de S se ordenan de la forma $S = \{s_0, \ldots, s_{k-1}\}$, donde $k = |S|$, y denotamos $l = k - 1$. Sea la arista inicial $(g_0 s_{-1}^{-1}, g_0)$ donde $g_0 \in G$ y $s_{-1} \in S$. Sea la función de ordenación $\sigma : S \times \{0, \ldots, l-1\} \longrightarrow S$ tal que el conjunto $\{\sigma(s, i) | 0 \leq i \leq l-1\} \bigcup \{s^{-1}\}$ coincide con el conjunto S para todo $s \in S$.

Dado un mensaje m, supongamos que el algoritmo hash regular del que partimos divide el mensaje en α dígitos entre 0 y $l-1$ de la forma $m = m_0 \cdots m_{\alpha-1}$, con $m_i \in \{0, \ldots, l-1\}$. Después, se realizan sucesivamente α pasos, desde $i = 0$ hasta $i = \alpha - 1$, donde se irá calculando $s_i = \sigma(s_{i-1}, m_i)$ y $v_i = v_{i-1} s_i$. El valor hash del mensaje m es

$$H(m) = g_0 \prod_{i=0}^{\alpha-1} s_i \coloneqq g_0 s_0 \cdots s_{\alpha-1}.$$

Para las funciones hash de Cayley también podemos encontrar una clara relación con las funciones iteradas. En particular, para el caso del grafo dirigido, podríamos tomar la función de compresión f como

$$f(g_i || m_i) \coloneqq \sigma(g_i, m_i),$$

donde notamos $g_{i+1} = \sigma(g_i, m_i) = g_i s_{m_i}$.

Para el caso del grafo no dirigido podríamos tomar,

$$f(s_{i-1} || m_i) \coloneqq \sigma(s_{i-1}, m_i)$$

en el caso del grafo no dirigido.

Ejemplo 3.1. Sea $C_{G,S} =$ un grafo de Cayley dirigido, donde $G = (\mathbb{Z}_6, +)$ y $S = \{2, 3\} = \{s_0, s_1\}$ un subconjunto generador de G no simétrico. Sea $g_0 = 0 \in G$ el vértice inicial y sea

$$\begin{aligned} \sigma : G \times \{0, 1\} &\longrightarrow G, \\ (g, i) &\longmapsto g + s_i, \end{aligned}$$

la función de ordenación.

Dado un mensaje m suponemos que el algoritmo divide el mensaje en 9 dígitos de la forma $m = m_0 \cdots m_8 = 010100101$, donde los dígitos están en base 2. El valor hash del mensaje sería:

$$\begin{aligned} H(m) &= g_0 + s_{m_0} + \cdots + s_{m_8} \\ &= g_0 + s_0 + s_1 + s_0 + s_1 + s_0 + s_0 + s_1 + s_0 + s_1 \\ &= 0 + 2 + 3 + 2 + 3 + 2 + 2 + 3 + 2 + 3 \\ &= 4. \end{aligned}$$

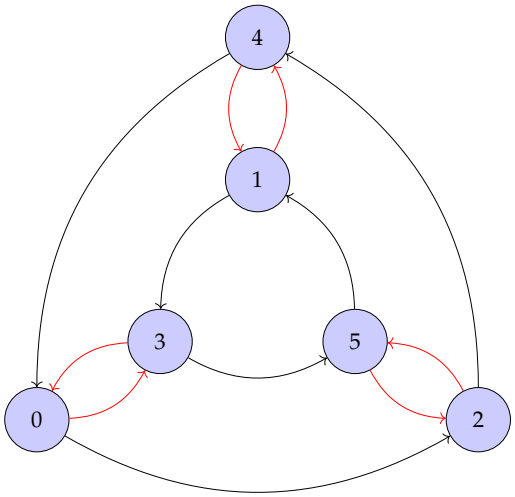


Figura 3.2: Grafo de Cayley

3.2.3. Resistencia a colisiones y a cálculo de preimágenes

La resistencia a colisiones y a cálculo de preimágenes pueden ser interpretadas de forma paralela mediante propiedades de los grafos. Por un lado, encontrar una colisión es equivalente a encontrar un ciclo o dos caminos que comiencen en el vértice inicial y terminen en un mismo vértice. Por otro lado, encontrar una preimagen es equivalente a encontrar un camino que comience en el origen y que termine en un vértice dado.

A continuación, se enunciarán una serie de problemas matemáticos de importancia asociados a la resistencia de colisiones y cálculo de preimágenes en funciones hash de expansión.

Problema 3.1. ***(Problema de los dos caminos restringidos)*** *Sea $G = (V, E)$ un grafo aleatorio. Dado un vértice inicial, elegido de forma aleatoria, $v_0 \in V$, encontrar dos caminos en G de longitud al menos l de forma que comiencen en v_0 y acaben en un mismo vértice.*

Problema 3.2. ***(Problema de los ciclos restringidos)*** *Sea $G = (V, E)$ un grafo aleatorio. Dado un vértice inicial, elegido aleatoriamente, $v_0 \in V$, encontrar un ciclo en G de longitud al menos l que pase por el vértice v_0.*

Problema 3.3. ***(Problema del camino)*** *Sea $G = (V, E)$ un grafo aleatorio. Dados dos vértices elegidos de forma aleatoria, v_0 y v, donde v_0 es el vértice inicial y v el vértice final, encontrar un camino en G de forma que su longitud sea al menos l, que empiece en v_0 y acabe en v.*

Problema 3.4. ***(Problema de los dos caminos)*** *Dado $G = (V, E)$ un grafo aleatorio, encontrar dos caminos en G de forma que su longitud sea al menos l y sus extremos coincidan.*

Problema 3.5. ***(Problema del ciclo)*** *Dado $G = (V, E)$ un grafo aleatorio, encontrar un ciclo en él con longitud al menos l.*

Los tres primeros problemas se dice que son *difíciles* cuando el grafo o el vértice inicial se eligen de manera aleatoria. Para que este término tenga sentido es necesario que el grafo no tenga un tamaño pequeño.

Otra propiedad de los grafos que nos da información sobre la función hash es la cintura del mismo, es decir, la longitud de su ciclo más corto. En concreto, nos dice la distancia mínima que hay entre dos mensajes cualesquiera que colisionan, ya que asociamos un ciclo con una colisión. Por ejemplo, si tenemos dos mensajes $m = m_1||m_2$ y $m' = m_1||m_2'$, de forma que $H(m) = H(m')$, entonces la longitud de m_2 o m_2' coincide con la cintura del grafo. Si la cintura del grafo no es muy grande la función hash tiende a ser menos segura, especialmente si el atacante elige un vértice inicial o si el grafo es de transitivo respecto a sus vértices, o en particular es de Cayley.

Del mismo modo, existen problemas de teoría de grupos más específicos a funciones hash de Cayley. Previamente daremos una definición que usaremos en dichos problemas.

Definición 3.8. Dado un grupo G, y $g_0, g_1, \ldots, g_{\alpha-1} \in G$. Diremos que un producto de elementos de G, $g_0 g_1 \cdots g_{\alpha-1}$, es *reducido* si $g_i g_{i+1} \neq 1$ para todo $i \in \{0, \ldots, \alpha - 2\}$.

Problema 3.6. ***(Problema de representación)*** *Dado un grupo G y un subconjunto suyo S, encontrar un producto reducido de elementos de S con longitud a lo sumo l, que es equivalente al elemento neutro del grupo, es decir, encontrar $s_0, \ldots, s_\alpha \in S$ tales que*

$$\prod_{0 \leq i \leq \alpha} s_i = 1,$$

donde $s_i \in S$, $s_i s_{i+1} \neq 1$ y $0 < \alpha \leq l$.

Problema 3.7. ***(Problema del balance)*** *Dado un grupo G y un subconjunto suyo S, encontrar dos productos reducidos de elementos de S de forma que su longitud sea como mucho l y que sean iguales, es decir, encontrar $s_0, \ldots, s_\alpha, s_0', \ldots, s_\alpha' \in S$ tales que*

$$\prod_{0 \leq i \leq \alpha} s_i = \prod_{0 \leq i \leq \alpha'} s_i',$$

donde $s_i, s_i' \in S$, $s_i s_{i+1}, s_i' s_{i+1}' \neq 1$ y $0 < \alpha, \alpha' \leq l$.

Problema 3.8. ***(Problema de la factorización)*** *Dado un grupo G, un subconjunto suyo S, y un elemento $g \in G$, encontrar un producto reducido de elementos de S con longitud como mucho l, que es equivalente a g, es decir, encontrar $s_0, \ldots, s_\alpha \in S$ tales que*

$$\prod_{0 \leq i \leq \alpha} s_i = g$$

donde $s_i \in S$, $s_i s_{i+1} \neq 1$ *y* $0 < \alpha \leq l$.

Estos últimos problemas son *difíciles* si l no es demasiado grande. Concretamente el Problema 3.6 es más difícil que el 3.7 generalmente pero coinciden si S es simétrico, como en el caso de las funciones hash de Cayley. En el caso del Problema 3.8, podemos apreciar que es un caso general del Problema 3.6.

3.2.4. Posibles ataques

En este apartado estudiaremos algunos específicos ataques a funciones hash de expansión y de Cayley.

3.2.4.1. Ataque del automorfismo

Se trata de un método de ataque hacia la resistencia a colisiones de funciones hash de expansión propuesto por Charles, Lauter y Goren [CLG09].

Sea $G = (V, E)$ un grafo regular de orden k y c una caminata entre v_0 y v_α. En este ataque existe un automorfismo del grafo, distinto de la identidad, que se puede calcular de manera eficiente. Supongamos que el atacante conoce el automorfismo f y sabe que la imagen $f(v)$ para cualquier vértice $v \in V$ se puede calcular de manera eficiente. Si la distancia media de un vértice v a su imagen $f(v)$ es pequeña, entonces es más fácil para el atacante encontrar a través de fuerza bruta dos caminatas $c_{v_0,f(v_0)}$ y $c_{v_\alpha,f(v_\alpha)}$ entre los vértices v_0,$f(v_0)$, y v_α,$f(v_\alpha)$ respectivamente. Como caminatas $(c_{v_0,f(v_0)}||f(c))$ y $(c||c_{v_\alpha,f(v_\alpha)})$ tienen los mismos extremos y la misma longitud, el atacante ha encontrado dos entradas distintas que devuelven el mismo valor hash. Por otro lado, c y $(c_{v_0,f(v_0)}||f(c)||f(c_{v_\alpha,f(v_\alpha)}))$, también son dos entradas que tienen los mismos extremos pero no tienen porqué tener la misma longitud, que devuelven el mismo valor hash.

3.2.4.2. Ataque por encuentro a medio camino

El *ataque por encuentro a medio camino*, también conocido por *meet-in-the-middle attack*, es un ataque para funciones hash iteradas cuya función de compresión f sea invertible.

El atacante puede construir mensajes correspondiente a un determinado resumen. Para ello debe aplicar la función de compresión a $2^{n/2}$ mensajes aleatorios y aplicarlo hacia atrás a otros $2^{n/2}$ mensajes aleatorios. Por la paradoja del cumpleaños sabemos que hay una gran probabilidad de que el atacante encuentre un valor en común en el medio.

Ahora nos centraremos en las particularidades que surgen al aplicar este ataque a funciones hash de expansión, en concreto, las de Cayley.

Sea $G = (V, E)$ un grafo de Cayley, H una función hash de Cayley, f la función de compresión y m_1, m_1', m_2, m_2' mensajes aleatorios. Si $H(m_1) = H(m_1')$ y $H(m_2) = H(m_2')$ entonces se cumple tanto $H(m_1||m_2) = H(m_1||m_2') = H(m_1'||m_2) = H(m_1'||m_2')$ como $H(m_2||m_1) = H(m_2||m_1') = H(m_2'||m_1) = H(m_2'||m_1')$. Además, como f es invertible, el ataque calcula preimágenes en un tiempo equivalente a la raíz cuadrada del tamaño de la salida.

3.2.4.3. Ataques a subgrupos

Se trata de un método de ataque hacia la resistencia a colisiones de funciones hash de Cayley.

Supongamos que existe una torre de subgrupos $G = G_0 \supset G_1 \supset \cdots \supset G_\alpha = \{1\}$ donde $|G_{i-1}|/|G_i| \leq B$ para todo i y para un límite B. Dado $S = \{s_1, \ldots, s_k\}$, la idea es ir de G_i a G_{i-1} de forma sucesiva hasta alcanzar la identidad.

El proceso comienza con la generación de productos de S hasta obtener un elemento $s_i^{(1)} \in G_1$. Calculamos del mismo modo más elementos hasta obtener $S_1^{(1)} = \{s_1^{(1)}, \ldots, s_{k'}^{(1)}\}$, que genera todos los elementos de G_2. Repetimos el proceso para $S_1^{(1)}$ y así sucesivamente.

3.2.4.4. Ataque trapdoor

Si la cintura de un grafo es pequeña, no significa que la función hash construida a partir de él no verifique la resistencia a colisiones. Esto es debido a que es difícil de encontrar ciclos que empiecen por un punto inicial v_0. Sin embargo, si se da este caso y es posible encontrar un vértice inicial que de lugar a colisiones, hay mayor riesgo de sufrir un ataque *trapdoor*.

Supongamos que existen dos mensajes m y m' de forma que parten del mismo vértice inicial es v y tienen el mismo valor hash. Si la función de compresión es invertible, entonces el atacante puede elegir el vértice inicial v y producir colisiones de la forma $(m_1||m||m_2, m_1||m'||m_2)$. Estas colisiones se pueden producir a través de los cálculos hacia atrás la función hash partiendo de v y teniendo en cuenta los dígitos de m_1. Para así después elegir el último vértice como vértice inicial para la función hash.

3.2.5. Maleabilidad

La maleabilidad es una propiedad de las funciones hash. Diremos que una función hash es maleable si modificaciones en un mensaje de entrada se puede relacionar con modificaciones en el valor hash obtenido. De esta forma, para un mensaje de entrada m con valor hash $H(m)$ es posible generar otro valor cifrado, sin conocer necesariamente el valor m, de forma que al descifrar ambos existe una relación entre ellos. Esto puede suponer en ocasiones problemas de seguridad, ya que puede permitir que un atacante altere el contenido de los mensajes.

Veamos cuales son las particularidades de la maleabilidad en grafos de expansión y grafos de Cayley.

Por un lado, si tenemos H una función hash de expansión construida a partir de un grafo $G = (V, E)$ regular de orden k, y tomamos un mensaje m. Al descomponer m en α dígitos entre 0 y $k-1$, o 0 y k, dependiendo de si es dirigido o no. Entonces, los valores hash obtenidos a partir de $m_0 \cdots m_{\alpha-2}$ y $m_0 \cdots m_{\alpha-1}$ son vértices adyacentes en el grafo. Por lo que un valor hash puede ser calculado a partir de otro.

En el caso de las funciones hash de Cayley, si partimos de un grafo de Cayley $C_{G,S} = (V, E)$, una función de Cayley H, y tomamos como vértice inicial el elemento neutro del grupo G. Dados dos mensajes $m = m_0 \cdots m_{\alpha-1}$ y $m' = m'_0 \cdots m'_{\alpha-1}$, entonces $H(m||m') = H(m) \cdot H(m')$, donde $\cdot$ es la operación interna del grupo.

Si tenemos el caso en el que el valor hash del mensaje m es $H(m) = 1$, entonces $H(m_1||m||m_2) = H((m_1||m)||m_2) = H(m_1||m) \cdot H(m_2) = H(m_1) \cdot H(m) \cdot H(m_2) = H(m_1) \cdot H(m_2)$ para

cualesquiera m_1 y m_2, es decir, existe una colisión entre $m_1||m_2$ y $m_1||m||m_2$. Por otro lado si tomamos como vértice inicial un elemento cualquiera del grupo $g \in G$, entonces $H(m||m') = H(m) \cdot g^{-1} \cdot H(m')$.

3.3. Conclusiones

En este capítulo se ha estudiado un diseño simple de funciones hash criptográficas construidas a partir de grafos de expansión y grafos de Cayley.

Estas funciones hash merecen especial interés debido a que algunas de sus propiedades pueden ser estudiadas desde el punto de vista de la teoría de grafos, haciendo uso de nociones como la de cintura de un grafo, ciclo o constantes de expansión. Entre estas propiedades se encuentran algunas de las más significativas para funciones hash criptográficas, como la resistencia a colisiones y a preimágenes. En particular, para las funciones hash de Cayley estas dos propiedades pueden admitir otra interpretación en términos de problemas de teoría de grupos. Dicho problemas, a pesar de no ser problemas clásicos en la criptografía, han sido capaz de resistir hasta intentos de ataque por más de 15 años.

Otra de las ventajas que presentan este tipo de funciones frente a las tradicionales es que no necesitan realizar una transformación previa para obtener un determinado tamaño ya que son capaces de procesar mensajes de distintos tamaños aleatorios.

Asimismo, las funciones hash de Cayley se pueden calcular en paralelo, lo cual es de gran interés debido a la eficiencia que pueden presentar en computadoras con arquitecturas con paralelismo.

También es de interés remarcar qué sucede si se usa una familia de grafos de expansión G_n. En este caso, la función hash construida es escalable, por lo que puede alcanzar cualquier nivel de seguridad con el parámetro n apropiado.

Cabe notar que este tipo de funciones al tener sus propias particularidades, se han desarrollado ataques específicos para ellas. Aunque posiblemente, la principal desventaja que presentan es la maleabilidad, que viene inducida por su estructura matemática. Esta propiedad se considera en general indeseable para la integridad de un mensaje, al poder ser alterados de forma maliciosa los mensajes por un atacante en el camino.

Por tanto, para concluir podemos decir que las ventajas que aportan los grafos de expansión y de Cayley son de mayor peso que el principal inconveniente que presentan. Esta construcción nos proporciona una mayor variedad de herramientas para poder probar propiedades de funciones hash de gran importancia, como lo son la resistencia a colisiones o al cálculo de preimágenes, y nos ofrecen otras mejoras en cuanto a eficiencia, como el paralelismo en grafos de Cayley o el procesamiento de mensajes de tamaños arbitrarios.

4 Protocolo de conocimiento cero basado en el problema de isomorfismo de grafos

A lo largo de los años se han buscado protocolos y métodos que garantizasen la seguridad y el anonimato de sus sistemas. A partir de dicha búsqueda y como resultado del trabajo de muchos criptógrafos, nació el *protocolo de conocimiento cero*.

En esta sección explicaremos en qué consiste dicho protocolo y cómo surgió. También se explicarán sus características y aplicaciones. Después, se presentará un protocolo de conocimiento cero basado en el problema de isomorfismo de grafos. Para ello, haremos uso de la teoría de grafos estudiada con anterioridad.

4.1. Protocolos de conocimiento cero

El *protocolo de conocimiento cero*, también conocido como *Zero Knowledge Proof (ZKP)*, es un método criptográfico que permite limitar y minimizar la accesibilidad de los datos, dando lugar a sistemas distribuidos anónimos y seguros.

Lo que caracteriza a este protocolo es que permite demostrar que se dispone de una determinada información sin tener que exponerla. En el proceso participan dos partes: el *probador* y el *verificador*.

El probador es la parte que conoce el secreto o información confidencial, y el verificador es la parte que se encarga de probar que la veracidad del probador y que efectivamente dispone de dicha información. Este proceso se puede realizar sin la intervención de una tercera parte.

4.1.1. Contexto histórico

El desarrollo de este protocolo ha sido posible gracias al trabajo de muchos criptógrafos y nuevos conceptos que han ido surgiendo a lo largo de los últimos años.

La criptografía siempre pretendía encontrar sistemas que nos proporcionasen privacidad y seguridad al mismo tiempo. A medida que pasaron los años se fueron encontrando sistemas de cifrados más seguros, especialmente con la aparición de los ordenadores que proporcionaron una potente herramienta para poder realizar cálculos complejos. No obstante, no fue hasta los años 70 cuando Diffie y Hellman [Dif76] desarrollaron un protocolo de intercambio de claves a través de un canal inseguro que dio lugar a la *criptografía asimétrica*. En la actualidad, es una de las herramientas de cifrado criptográficas más usadas debido a su alta seguridad.

La criptografía de clave asimétrica está compuesta de dos claves: una privada y una pública. La clave privada es de uso personal, y sirve para cifrar y descifrar mensajes de forma segura. Por otro lado, la clave pública va estar al alcance del resto de usuarios que participen en la comunicación o intercambio de información. En particular, el creador de ambas claves distribuirá su clave pública al resto, pero nunca debe compartir su clave privada. La clave

pública se crea a partir de la clave privada, pero la clave privada no puede obtenerse a partir de la pública. De esta forma, si una parte desea enviarle a otra información de forma confidencial, el receptor debe proporcionarle al emisor su clave pública de forma que el emisor cifre el mensaje con dicha clave y solo pueda ser descifrada con la clave privada del receptor.

Pocos años después, David Chaum diseñó las *Firmas Ciegas* [Cha83]. Éstas están basadas en esquemas de firma digital de clave pública y, permiten a un usuario enviarle unos datos a un firmante y obtenerlos firmados sin que el firmante conozca todo el contenido de los mismos.

Este concepto surgió de toda la información que puede llegar a un base de datos cuando compramos un producto o servicio electrónicamente o realizamos algún pago, violando nuestro derecho a privacidad.

De forma intuitiva, David Chaum relacionó el concepto que introdujo con el siguiente sistema de voto. En dicho sistema, un votante debe introducir una papeleta de forma anónima en un sobre con las credenciales del votante en el exterior y forrado con papel carbón. Después, un funcionario debe firmar dicho sobre impregnando su firma a la papeleta a través del material del sobre. Posteriormente, el sobre se devuelve al votante y transfiere dicha papeleta a un nuevo sobre sin sus datos. De esta forma, el firmante ha firmado la papeleta sin haberla visto y una tercera persona puede comprobar que la papeleta es válida al verificar la firma.

No obstante, David Chaum también creó las *Firmas Ciegas Grupales*, permitiendo a un firmante firmar un mensaje proveniente de un grupo de personas, verificar que dicho mensaje proviene de ese grupo pero sin conocer quién lo había firmado. Por estos trabajos, Chaum es reconocido como uno de los precursores de los protocolos de conocimiento cero.

No obstante, no fue hasta 1985 cuando Shafi Goldwasser, Silvio Micali y Charles Rackoff publicaron el artículo *La complejidad del conocimiento de los sistemas de prueba interactivos* [GMR19] en el que definieron la base del protocolo de conocimiento cero.

Finalmente, David Chaum publicó en 1987, junto a Gilles Brassard y Claude Crépeau, el artículo *Pruebas de revelación mínima de conocimiento* [BCC88], donde se definiría formalmente el protocolo.

4.1.2. La Cueva de Alí Babá

En 1992, Jean-Jacques Quisquater y Louis Guillou escribieron un artículo llamado *¿Cómo explicar a tus hijos los protocolos de conocimiento cero?* [QQQ+89]. En él explicaron de forma sencilla el protocolo a través de una historia denominada *La cueva de Alí Babá*.

Dicha historia narra los hechos de un hombre llamado Alí Babá que frecuentaba un bazar en la ciudad de Baghdad. Un día Alí Babá fue atracado por un ladrón y lo persiguió hasta llegar a la entrada de una cueva. Al adentrarse en la cueva llegó hasta un punto en el que se separaba en dos caminos distintos. Tomó uno de ellos en busca del ladrón pero se encontró con un camino sin salida. Regresó a la bifurcación y optó por ir por el otro camino sin esperar que éste tampoco tuviese salida.

Durante los siguiente 39 días, distintos ladrones volvieron a robarle y a huir a la cueva, y en todos esos días Alí Babá no encontró al ladrón en ninguno de los pasajes. Confundido por la inverosímil suerte de los 40 ladrones, al siguiente día Alí Babá decidió esconderse en la cueva en uno de los pasajes. Tras un tiempo de espera, llegó un ladrón que huía de su víctima y susurró las palabras "Ábrete sésamo". Acto seguido se abrió un pasadizo por donde escapó

el ladrón y unos instantes después llegó la víctima del robo decepcionado por no haber dado con el ladrón. Alí Babá, por el contrario, había descubierto el secreto de la cueva. Al decir las palabras mágicas descubrió que se abría un pasadizo que conectaba ambos pasajes, y tras muchos intentos consiguió cambiar las palabras secretas que lo abrían. De esta forma, Alí Babá pudo pillar el siguiente día al ladrón y decidió escribir la historia en un manuscrito.

Siglos después la cueva fue encontrada por diversos arqueólogos y fue foco de muchos investigadores y cadenas de televisión. Uno de los investigadores, Mick Ali, un posible descendiente de Alí Babá, decía saber las palabras secretas de la cueva y quería probarlo sin revelarlas.

Para ello, contactó con una cadena de televisión que grabó de forma detallada el interior de la cueva. Una vez la cueva estaba vacía, Mick Alí entró solo y decidió pasar por uno de los pasajes. Después, un reportero y un cámara llegaron hasta la bifurcación y lanzando una moneda al aire decidían si Mick Alí debía salir por un pasaje o por otro. Desde aquel lugar le gritaron a Mick Alí por donde debía salir y éste lo hizo.

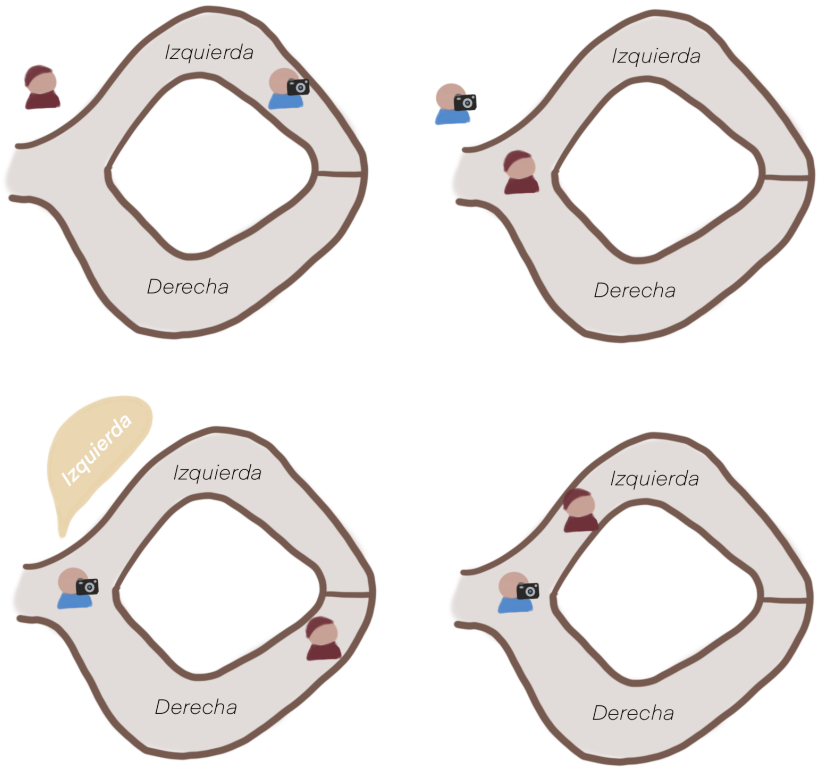


Figura 4.1: Cueva de Alí Babá

En honor a los cuarenta ladrones, repitieron este suceso en cuarenta ocasiones y, en todas ellas, Mick Alí salió por donde se le indicó. Si Mick Alí no conociera las palabras mágicas no podría abrir el pasadizo por lo que tendría un 50 % de probabilidades de haber elegido

el pasaje correcto. Al repetir este proceso 40 veces, la probabilidad de que haya escogido el camino correcto se va disminuyendo en un 50 % en cada experimento, por lo que al final sería del $(0.5^{40} \cdot 100)$ %.

De esta forma, nadie supo el secreto de la cueva, pero todos quedaron convencidos de que la única forma en la que Mick Alí pudo salir en las cuarenta ocasiones distintas por el camino adecuado no pudo ser otra que conocer dicho secreto.

4.1.3. Características

Un protocolo de conocimiento cero debe cumplir tres propiedades.

- Integridad. Suponiendo que el probador y el verificador son honestos y siguen el protocolo correctamente, el protocolo hará que el probador convenza al verificador de que la declaración es verdadera.
- Solidez. Se debe reducir al máximo la probabilidad de que el probador engañe al verificador.
- Conocimiento cero. Si la declaración es verdadera, ningún verificador engañoso puede saber más que este hecho.

Las dos primeras propiedades son fundamentales para decir que un protocolo es de conocimiento, pero la última de ellas es esencial para probar el anonimato y decir que es un protocolo de conocimiento cero.

Además, se debe proveer una fuente de aleatoriedad fiable. En el ejemplo de la *Cueva de Alí Babá*, el reportero elige el camino de forma aleatoria usando una moneda y lanzándola al aire.

Por otro lado, estas pruebas no son pruebas en el sentido estricto ya que siempre existe una probabilidad, por pequeña que sea, de que un probador tramposo convenza al verificador de una declaración falsa. No obstante, existen técnicas para disminuir el error de la solidez a valores prácticamente despreciables.

4.1.4. Clasificación

Los protocolos de conocimiento cero pueden ser interactivos o no interactivos.

4.1.4.1. Protocolo de conocimiento cero interactivo

En este tipo de protocolo de conocimiento cero será necesario que tanto el probador como el verificador estén presentes durante la ejecución del protocolo.

La comunicación entre ambas partes suele seguir la siguiente estructura. En primer lugar, el probador genera un mensaje indicando que es conocedor de un secreto. El verificador le presenta un desafío, normalmente aleatorio, al probador. Este último le responde al verificador con la respuesta. Para calcular dicha respuesta se necesita saber el secreto. Se repite este proceso varias veces para garantizar una mayor seguridad.

En este caso, el probador le está revelando que posee el conocimiento al verificador, pero si una tercera personas los estuviera observando, no podría verificar su conocimiento.

Este protocolo provee una mayor privacidad, pero si es necesario probarlo a más de una persona requiere de gran esfuerzo ya que habría que repetir el mismo proceso para cada personas un número determinado de veces.

Algunos protocolos de este tipo son el *Protocolo de Chaum-Pedersen*, el *Protocolo de Cramar-Damgård-Schoenmakers* y el *Algoritmo de identificación de Schnorr*.

4.1.4.2. Protocolo de conocimiento cero no interactivo

En este tipo de protocolo no es necesario que el probador y el verificador estén presentes en la ejecución del protocolo, sino que el probador puede dejar un escrito del protocolo para que el verificador lo pueda verificar en cualquier momento.

Es frecuente utilizar la *heurística de Fiat-Shamir* en estos casos.

Veamos el siguiente ejemplo de un protocolo de conocimiento cero no interactivo.

Sudoku con cartas El sudoku es un juego simple en cual todas las filas, columnas y sectores deben tener los números del 1 al 9 una única vez. A pesar de sus sencillas reglas es un juego que se considera difícil, ya que puede tardar días en ser resuelto por ordenadores.

Supongamos que sabes la solución de este juego, y quieres vender la solución. Para que el comprador sepa que no le estás engañando tienes que convencerle de que realmente conoces la solución sin desvelársela. Este es el desafío del sudoku con cartas.

Para ello, necesitarías 27 cartas de cada número del 1 al 9, por lo que tendrías en total 243 cartas. Después, tendrás que poner tres cartas en una casilla con la solución. Por lo que si el número correcto en esa casilla es el 5, colocas tres cartas del 5 sobre ella.

En las casillas donde están las respuestas visibles se colocarán las cartas boca arriba, mientras que en las demás boca abajo.

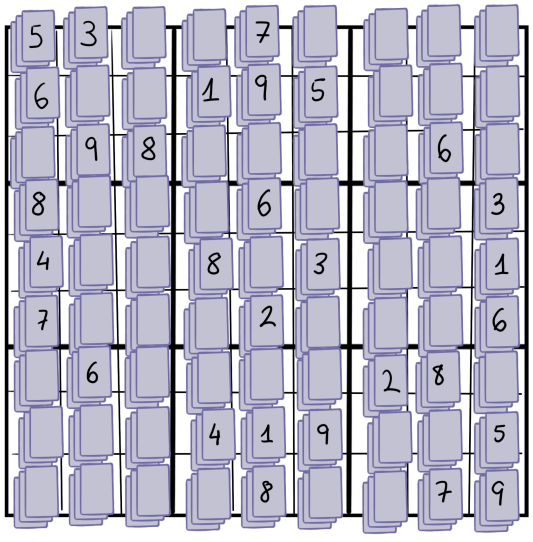


Figura 4.2: Tablero sudoku con 243 cartas

Para probar que has colocado las cartas en el sitio correcto sin revelarlas, debes coger una carta superior de cada columna hasta que tengas nueve pilas. Otra por filas y otra por sectores. Después barajas cada mazo de cartas y los volteas para comprobar los números.

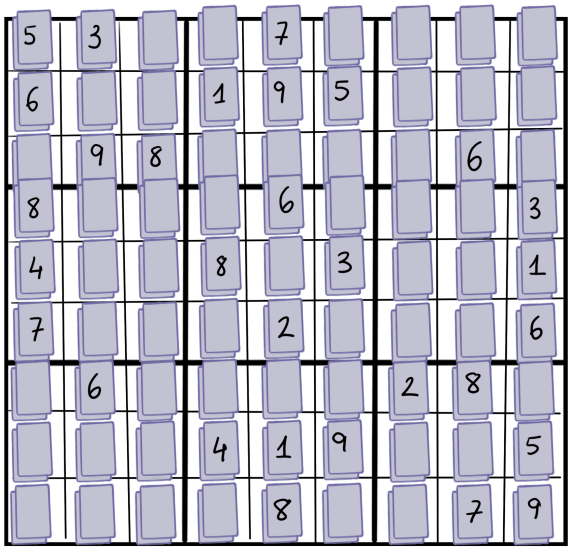


Figura 4.3: Tablero sudoku con 162 cartas

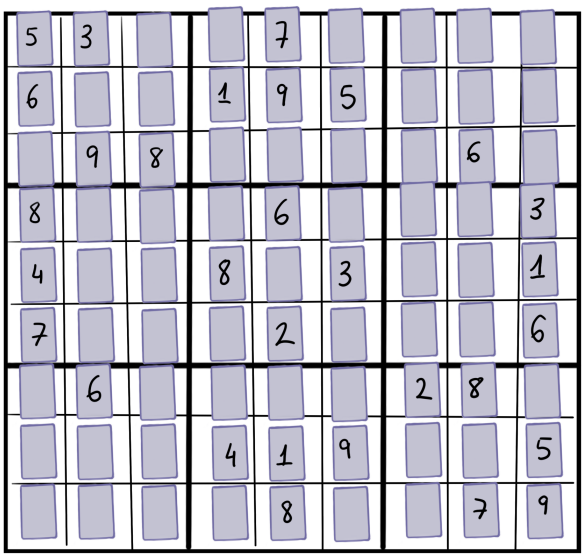


Figura 4.4: Tablero sudoku con 81 cartas

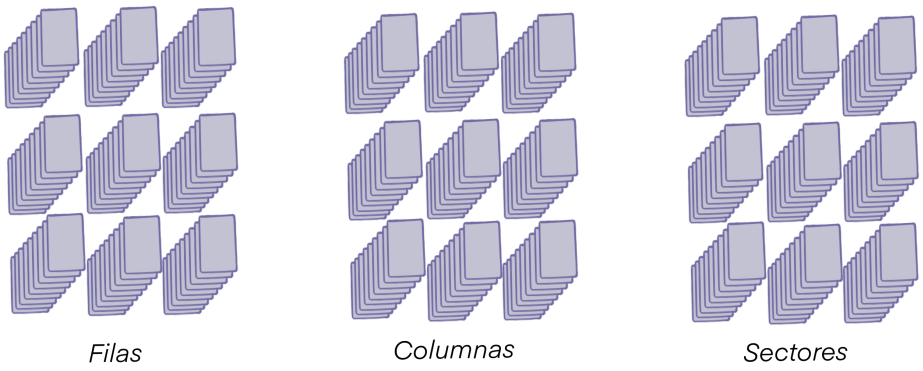


Figura 4.5: Mazos de cartas barajadas

Por tanto, si aparecen en cada pila los números del 1 al 9 sin repetir entonces el verificador sabrá que efectivamente conoces la solución.

Figura 4.6: Comprobación de cada mazo

Cabe notar que no ha sido necesario que el probador y el verificador interactúen, sino que el probador dejó el tablero de sudoku resuelto de forma que si el verificador posteriormente, en cualquier momento, sigue el protocolo dado podrá saber que el probador conocía la solución.

Este tipo de protocolo es más útil cuando quieres demostrar tu conocimiento a un gran número de personas de forma sencilla y reduciendo costos.

4.1.5. Aplicaciones

Los protocolos de conocimiento cero permiten alcanzar altos niveles de privacidad, seguridad y anonimato, por lo que distintos sectores emplean esta tecnología.

Desde el ámbito sanitario para el uso de historiales médicos, hasta el ámbito militar para asegurar comunicaciones, los protocolos de conocimiento cero tienen gran importancia en la actualidad.

Veamos de forma más detallada algunas de sus aplicaciones hoy en día.

- **Autenticación.** Los protocolos de conocimiento cero ayudan a evitar fuga de datos. Esto se debe a su capacidad de crear un canal seguro para que el usuario pueda hacer uso de su información de autenticación sin tener que revelarla.

- **Mensajería**. El cifrado de extremo a extremo es necesario a la hora de enviar y recibir mensajes, para que nadie pueda leer tus mensajes privados sin permiso. Haciendo uso de protocolos de conocimiento cero se garantiza esa seguridad de extremo a extremo sin poner en riesgo la privacidad.
- **Compartir datos.** Siempre existen riesgos a la hora de compartir información a través de la red, por lo que los protocolos de conocimiento cero pueden presentar ayuda en estos casos.
- **Seguridad para información importante**. Es imprescindible la seguridad y confidencialidad en información relativa a la cuentas bancarias o tarjetas de crédito. Los protocolos de conocimiento puede ayudar a que tu historial de tarjetas obtenga la suficiente seguridad. Bancos como ING hacen uso de estos protocolos.

Sin embargo, cabe destacar las aplicaciones de estos protocolos en las criptomonedas. Éstos son especialmente utilizados para asegurar transacciones de forma privada entre usuarios. Este es el caso de Zcash y Monero.

Zcash es una plataforma blockchain de código abierto que usa un sistema de pruebas denominado *zk-SNARKs* basado en el funcionamientos de los protocolos de conocimiento cero no interactivos.

Por otro lado, es frecuente el uso de *transacciones confidenciales* en las redes de algunas criptomonedas. Las transacciones confidenciales son un protocolo criptográfico del tipo conocimiento cero que se emplea para hacer transacciones de criptomonedas en una blockchain de forma anónima y privada. Monero también hace uso de esta tecnología.

4.2. Protocolo de conocimiento cero basado en el problema de isomorfismo de grafos

En esta sección se presentará un protocolo de conocimiento cero basado en el problema de isomorfismo de grafos [Aye09, Dur16]. Dicho protocolo combina pruebas de conocimiento cero con métodos de intercambio de clave para obtener una comunicación segura.

4.2.1. Problema de isomorfismo de grafos

En el Capítulo 1 se explican los conceptos de isomorfismo y automorfismo de grafos. Estas nociones dan lugar a diversos problemas en la teoría de grafos.

Algunos de ellos se considera que pueden resolverse en tiempo polinomial, es decir, que son problemas de clase $\mathcal{P}$. Sin embargo, otros se sabe que pertenecen a la clase de complejidad $\mathcal{NP}$. En concreto, para el problema general de isomorfismo de grafos, en el peor de los casos, no se conocen algoritmos de tiempo polinomial para resolverlo sino que tienen una complejidad de tiempo exponencial.

El problema general de isomorfismo de grafos es el siguiente.

Problema 4.1 (Problema general de isomorfismo de grafos). *Dados dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ determinar si existe un isomorfismo entre ellos, es decir, determinar si ambos grafos son isomorfos.*

Diremos que dos problemas que pertenecen a la clase de complejidad $\mathcal{NP}$ son equivalentes en tiempo polinomial si la existencia de un algoritmo de tiempo polinomial para uno de ellos implica la existencia de otro para el otro problema.

A continuación, enunciaremos algunos problemas sobre isomorfismo y automorfismo de grafos que son equivalentes en tiempo polinomial con el Problema 4.1 [Hof82].

Problema 4.2 (Existencia de un isomorfismo de grafos). *Dados dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, con $|V_1| = |V_2| = n$, probar que son isomorfos.*

Problema 4.3 (Existencia de un isomorfismo de grafos etiquetados). *Dados dos grafos etiquetados $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, probar que son isomorfos.*

Problema 4.4 (Automorfismo de grafos). *Dado un grafo $G = (V, E)$, determinar un conjunto generador para su grupo de automorfismos.*

Problema 4.5 (Orden del grupo de automorfismos). *Dado un grafo $G = (V, E)$, determinar el orden de su grupo de automorfismos.*

Problema 4.6 (Número de isomorfismos). *Dados dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, determinar el número de isomorfismos que existen del uno al otro.*

También existen algunas generalizaciones de problemas de isomorfismos de grafos. En concreto, presentaremos generalizaciones del problema de isomorfismo de subgrafos.

Problema 4.7 (El problema del subgrafo). *Dados dos grafos, $G = (V, E)$ y $G' = (V', E')$, determinar si uno es subgrafo del otro.*

Problema 4.8 (El problema del subgrafo común). *Dados dos grafos, $G = (V, E)$ y $G' = (V', E')$, determinar el subgrafo común a ambos grafos que tenga el mayor orden posible.*

4.2.2. Descripción del protocolo

A continuación, se presentará un protocolo de conocimiento cero propuesto en [Ayeo9, Dur16] basado en el problema de isomorfismo de grafos.

Partimos de dos grafos isomorfos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ que verifican $h(G_1) = G_2$, donde h es un isomorfismo entre ambos grafos.

El probador sabe que G_1 y G_2 son isomorfos puesto que conoce un isomorfismo definido entre ambos. Por ello, el probador quiere convencer al verificador que conoce dicho isomorfismo sin revelárselo.

Los pasos a seguir en el protocolos son los siguientes.

- En primer lugar, el probador elige de forma aleatoria un número $a \in \{1, 2\}$ para elegir de que grafo partir, si G_1 o G_2. Por ejemplo, tirando una moneda y obtener cara o cruz.
- Después, el probador elige de aleatoriamente una permutación p y obtiene un nuevo grafo, $G_3 = p(G_a)$, permutando los vértices del grafo elegido G_a.
- El probador le envía la matriz de adyacencia del grafo G_3 al verificador.

- El verificador recibe la matriz de adyacencia del grafo G_3 y elige de forma aleatoria un índice $b \in \{1, 2\}$.
- El verificador envía al probador el índice b elegido y le presenta un desafío: obtener una función λ tal que $\lambda(G_3) = G_b$.
- Dependiendo de los valores de a y b el probador le mandará una función λ distinta.
  - Si $a = b$, entonces el probador le debe enviar $\lambda = p^{-1}$ al verificador.
  - Si $a = 1$ y $b = 2$, entonces el probador le envía $\lambda = h \circ p^{-1}$ al verificador.
  - Si $a = 2$ y $b = 1$, el probador debe enviar $\lambda = h^{-1} \circ p^{-1}$ al verificador.
- El verificador comprueba si $\lambda(G_3) = G_b$.

Es necesario aplicar el protocolo en varias rondas para que el probador convenza al verificador, ya que el probador puede haber tenido suerte y haber adivinado el índice b elegido por el verificador.

La probabilidad de que en n rondas el probador escoja con anterioridad el índice que elegirá el verificador es de $\frac{1}{2^n}$. Por lo que si n es lo suficientemente grande la probabilidad de que el probador engañe al verificador es prácticamente despreciable.

4.3. Conclusiones y comentarios sobre el protocolo

El problema de isomorfismo de grafos es comúnmente considerado adecuado para el protocolo de conocimiento cero, sin embargo, comentaremos en esta sección algunas de las cuestiones que se deben abordar y de los inconvenientes que se plantean a la hora de implementar este protocolo.

Una cuestión importante que abordar es el tipo de grafo que utilizar a la hora de aplicar el protocolo presentado. A pesar de que la versión original del problema de isomorfismo de grafos no se conoce que se pueda resolver en tiempo polinomial, existen algunos tipos de grafos para los cuales sí es posible. Hay algunos tipos de grafos para los cuales es comúnmente sencillo encontrar sus isomorfos. Por ejemplo, los árboles, los grafos aleatorios o los grafos planos. Por lo que no es recomendado su uso para la práctica del protocolo.

Por otro lado, debemos tener en cuenta los problemas que pueden surgir en el procedimiento del protocolo. Uno de ellos es que el probador o el verificador mientan o hagan trampas.

Como los grafos G_1 y G_2 están al alcance de todas las partes involucradas en la interacción, si suponemos que el verificador (u otra parte) posee un algoritmo capaz de obtener el isomorfismo h en un corto periodo de tiempo, entonces el verificador (u otra parte) será capaz de hacerse pasar por el probador ante el verificador o ante un tercero.

Otro de los inconvenientes de este protocolo es la gran cantidad de exposición de algoritmos disponibles para resolver el problema. Estos algoritmos prácticos son capaces de resolver el problema general de isomorfismo de grafos en tiempo polinomial, y han sido el centro de numerosos estudios en las últimas décadas.

Por un lado, Brendan McKay desarrolló un paquete con el algoritmo *Nauty* [M+81]. Fue considerado uno de los algoritmos más rápidos hasta que aparecieron otros mejores. Entre ellos se encontraban *Bliss* de Tommi Juntilla y Petteri Kashi [JK07], *Saucy* de Martin Kutz

y Pascal Schweitzer [KS07], *Traces* de Adolfo Piperno [Pip08] y *Vsep* de Stoicho Stoichev [Sto10].

Otro de los problemas que presenta el protocolo es qué pasaría si se intercepta la matriz de adyacencia del grafo G_3 por una tercera parte. En este caso, si una tercera parte intercepta la matriz de adyacencia de dicho grafo podría hacerse pasar por el probador si posee un algoritmo potente y rápido como en el problema anterior, o si el tipo de grafo empleado no es muy complejo para el problema.

Por tanto, podríamos decir que para obtener una mayor seguridad en el protocolo de conocimiento cero para el problema de isomorfismo de grafos la principal solución es encontrar grafos complejos para el problema y para los algoritmos que lo resuelven.

Sin embargo, son inevitables los inconvenientes que surgen a partir del avance de ciertos algoritmos de resolución del problema en tiempo polinomial en algunos casos, por lo que podríamos decir que el problema de isomorfismo de grafos no es perfectamente adecuado para ser implementado como protocolo de conocimiento cero.

A modo de conclusión, podríamos decir que es de notable interés tanto encontrar algoritmos eficientes que permitan resolver el problema de forma práctica, como encontrar grafos complejos para el problema. El primero es especialmente útil para poder probar isomorfismos en tiempo polinomial, mientras que el segundo lo es para la seguridad del protocolo de conocimiento cero basado en el problema de isomorfismo de grafos.

5 Conclusiones y futuros trabajos

La Teoría de Grafos presenta una gran variedad de aplicaciones en la Criptografía. Dichas aplicaciones nos permiten aprovechar las propiedades de los grafos para explicar propiedades más complejas en la Criptografía. En este trabajo se ha realizado de forma efectiva un estudio de la Teoría de Grafos y de dichas aplicaciones, completando así los objetivos propuestos inicialmente. No obstante, a través del desarrollo dichos objetivos, hemos podido comprobar que las aplicaciones estudiadas en el trabajo también presentan desventajas inducidas de la estructura matemática de los grafos.

En particular, después de haber analizado el algoritmo de cifrado propuesto podemos concluir que no es un algoritmo eficiente ni práctico, ya que no es capaz de cifrar cualquier mensajes aleatorio y los tiempos de ejecución son muy extensos debido a las operaciones realizadas con matrices. No obstante, teóricamente presenta interés al usar distintas nociones de grafos.

Por otro lado, una vez estudiadas las funciones hash criptográficas y sus construcciones a partir de grafos de expansión y de Cayley, podemos decir que dichas construcciones sí nos han aportado una gran cantidad de ventajas frente a su principal inconveniente, que es la maleabilidad. Por tanto, se considera de interés una profundización de estas funciones hash.

A partir del estudio del protocolo de conocimiento cero para el problema de isomorfismo de grafos, hemos podido apreciar las dos áreas de interés relacionadas con el tema: el encontrar algoritmos eficientes para resolver el problema y encontrar grafos difíciles para el mismo. El avance del primero resulta un inconveniente para el segundo y viceversa.

En cuanto a los posibles problemas abiertos o futuros trabajos, nos centraremos de forma individual en cada una de las aplicaciones de Teoría de Grafos en Criptografía.

En primer lugar, se pueden realizar muchas mejoras al algoritmo de cifrado propuesto por [Eta14]. En cuanto a mejoras de eficiencia se puede reducir el tiempo de ejecución de mensajes de gran tamaño dividiendo dicho mensajes en bloques más pequeños para cifrarlos por separado, y después, concatenarlos. También sería conveniente solucionar los problemas que presenta respecto a las palabras que no generan una matriz invertible, en especial aquellas que tienen dos o más letras iguales consecutivas.

Para las funciones hash criptográficas de expansión, en particular para las de Cayley, hay muchos problemas de representación abiertos. Sabemos que hay problemas de representación que son fáciles de resolver, mientras otros han conseguido superar todos los intentos de ataque. Además, no sabemos la complejidad o dificultad exacta que poseen este tipo de problemas, por lo que sería interesante estudiarla diviéndolos en clases de problemas fáciles, problemas que se pueden reducir a problemas conocidos, y problemas que no se pueden reducir a otros conocidos pero que se consideran difíciles.

Por otro lado, como futuro trabajo se pueden encontrar nuevos algoritmos específicos para cada tipo de grafo difícil que resuelvan el problema de isomorfismo de grafos en tiempo polinomial. Sin embargo, esto supondría un problema para el protocolo de conocimiento cero descrito. Por lo que para mejorar su seguridad se podría desarrollar un protocolo de

conocimiento cero que dependa de tiempo, es decir, que el probador deba responder al desafío del verificador en un tiempo limitado.

En definitiva, podemos concluir que en la actualidad tanto la Teoría de Grafos como la Criptografía son de notable interés, y hemos podido comprobar que quedan muchas nuevas alternativas y vías de trabajo que fusionan ambas áreas.

6 Apéndices

6.1. Código

Todo el código implementado en la elaboración de trabajo se encuentra en el siguiente repositorio de GitHub:

https://github.com/lmd-ugr/grafos-criptografia.

6.2. Agradecimientos

En primer lugar quiero agradecer a mi familia su apoyo incondicional a lo largo de la realización de este trabajo.

Quiero agradecerles a mis tutores Jesús García Miranda y Pedro A. García Sánchez el gran seguimiento que han realizado de mi trabajo, su disponibilidad a la hora de resolver todas mis dudas, y sus rigurosas correcciones que han hecho posible la realización del mismo.

Por último, dar las gracias a mis amigos por haberme animado y aconsejado durante todo el proceso.

Bibliografía

Las referencias se listan por orden alfabético. Aquellas referencias con más de un autor están ordenadas de acuerdo con el primer autor.

[ACMC] Biron Enrique Acosta Carvajal and Luisa María Montoya Conde. Elementos de la teoría de grafos y la conjetura de evasividad. [Citado en pág. 17]

[Alo84] Noga Alon. Eigenvalues and expanders, combinatorica 6 (1986), 83–96. *Theory of computing (Singer Island, Fla., 1984)*, 1984. [Citado en pág. 59]

[AM85] Noga Alon and Vitali D Milman. $\lambda 1$, isoperimetric inequalities for graphs, and superconcentrators. *Journal of Combinatorial Theory, Series B*, 38(1):73–88, 1985. [Citado en pág. 59]

[Aye09] Eric Ayeh. *An Investigation Into Graph Isomorphism Based Zero-knowledge Proofs*. PhD thesis, Citeseer, 2009. [Citado en págs. 16, 106, and 107]

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of computer and system sciences*, 37(2):156–189, 1988. [Citado en pág. 100]

[BR97] Albert T Bharucha-Reid. *Elements of the Theory of Markov Processes and their Applications*. Courier Corporation, 1997. [Citado en pág. 60]

[Bus82] Peter Buser. A note on the isoperimetric constant. In *Annales scientifiques de l'École normale supérieure*, volume 15, pages 213–230, 1982. [Citado en pág. 59]

[Cha77] Gary Chartrand. *Introductory graph theory*. Courier Corporation, 1977. [Citado en págs. 17 and 34]

[Cha83] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983. [Citado en pág. 100]

[Che15] Jeff Cheeger. A lower bound for the smallest eigenvalue of the laplacian. In *Problems in analysis*, pages 195–200. Princeton University Press, 2015. [Citado en pág. 59]

[CLG09] Denis X Charles, Kristin E Lauter, and Eyal Z Goren. Cryptographic hash functions from expander graphs. *Journal of CRYPTOLOGY*, 22(1):93–113, 2009. [Citado en págs. 91 and 96]

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022. [Citado en págs. 17, 53, and 54]

[CZ13] Gary Chartrand and Ping Zhang. *A first course in graph theory*. Courier Corporation, 2013. [Citado en pág. 43]

[Dam89] Ivan Bjerre Damgård. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*, pages 416–427. Springer, 1989. [Citado en pág. 88]

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. Ripemd-160: A strengthened version of ripemd. In *International Workshop on Fast Software Encryption*, pages 71–82. Springer, 1996. [Citado en pág. 90]

[Dif76] Whitfield Diffie. New direction in cryptography. *IEEE Trans. Inform. Theory*, 22:472–492, 1976. [Citado en págs. 63 and 99]

[Dod84] Jozef Dodziuk. Difference equations, isoperimetric inequality and transience of certain random walks. *Transactions of the American Mathematical Society*, 284(2):787–794, 1984. [Citado en pág. 59]

[Dur16] Mariana Durcheva. Zero knowledge proof protocol based ongraph isomorphism problem. *Journal of Multidisciplinary Engineering Science and Technology*, October 2016. [Citado en págs. 16, 106, and 107]

[Ede68] Murray Edelberg. *Introduction to combinatorial mathematics*. Mcgraw-Hill College, 1968. [Citado en pág. 48]

[Eta14] Wael Etaiwi. Encryption algorithm using graph theory. *Journal of Scientific Research and Reports*, (JSRR.2014.19.004), January 2014. [Citado en págs. 15, 16, 63, 64, 78, and 111]

[GM17] Jesús García Miranda. Lógica para informáticos (y otras herramientas matemáticas). 2017. [Citado en págs. 16, 17, and 46]

[GMR19] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 203–225. 2019. [Citado en pág. 100]

[Hof82] Christoph M Hoffmann. *Group-theoretic algorithms and graph isomorphism*. Springer, 1982. [Citado en pág. 107]

[JK07] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 135–149. SIAM, 2007. [Citado en pág. 108]

[KL20] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020. [Citado en pág. 82]

[Kow19] Emmanuel Kowalski. *An introduction to expander graphs*. Société mathématique de France, 2019. [Citado en págs. 16, 17, and 81]

[KS07] Martin Kutz and Pascal Schweitzer. Screwbox: a randomized certifying graph-non-isomorphism algorithm. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 150–157. SIAM, 2007. [Citado en pág. 109]

[Lan98] Serge Lang. *A first course in calculus*. Springer Science & Business Media, 1998. [Citado en pág. 85]

[M+81] Brendan D McKay et al. Practical graph isomorphism. 1981. [Citado en pág. 108]

[Mer79] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979. [Citado en pág. 88]

[Mer90] Ralph Charles Merkle. Advances in cryptology—crypto 89 proceedings. *Lecture Notes in Computer Science*, 435:218–238, 1990. [Citado en pág. 88]

[Nis92] Corporate Nist. The digital signature standard. *Communications of the ACM*, 35(7):36–40, 1992. [Citado en pág. 91]

[Pea05] Karl Pearson. The problem of the random walk. *Nature*, 72(1865):294–294, 1905. [Citado en pág. 60]

[Pet09] Christophe Petit. On graph-based cryptographic hash functions, May 2009. [Citado en págs. 16, 81, and 82]

[Pip08] Adolfo Piperno. Search space contraction in canonical labeling of graphs. *arXiv preprint arXiv:0804.4881*, 2008. [Citado en pág. 109]

[QQQ+89] Jean-Jacques Quisquater, Myriam Quisquater, Muriel Quisquater, Michaël Quisquater, Louis Guillou, Marie Annick Guillou, Gaïd Guillou, Anna Guillou, Gwenolé Guillou, and Soazig Guillou. How to explain zero-knowledge protocols to your children. In *Conference on the Theory and Application of Cryptology*, pages 628–631. Springer, 1989. [Citado en pág. 100]

[Riv92] Ronald Rivest. The md5 message-digest algorithm. Technical report, 1992. [Citado en pág. 89]

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. [Citado en pág. 90]

[SG12] Rajeev Sobti and Ganesan Geetha. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*, 9(2):461, 2012. [Citado en págs. 81 and 90]

[Sta] Secure Hash Standard. National institute of standards and technology (nist), fips publication 180-2, aug 2002. [Citado en pág. 89]

[Sta95] Secure Hash Standard. Fips pub 180-1. *National Institute of Standards and Technology*, 17(180):15, 1995. [Citado en pág. 89]

[Sto10] Stoicho D Stoichev. Vsep-new heuristic and exact algorithms for graph automorphism group computation. *arXiv preprint arXiv:1007.1726*, 2010. [Citado en pág. 109]

[TZ93] Jean-Pierre Tillich and Gilles Zémor. Group-theoretic hash functions. In *Workshop on Algebraic Coding*, pages 90–110. Springer, 1993. [Citado en pág. 91]

[TZ94] Jean-Pierre Tillich and Gilles Zémor. Hashing with sl 2. In *Annual International Cryptology Conference*, pages 40–49. Springer, 1994. [Citado en pág. 91]

[Zém91] Gilles Zémor. Hash functions and graphs with large girths. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 508–511. Springer, 1991. [Citado en pág. 91]

[Zém94] Gilles Zémor. Hash functions and cayley graphs. *Designs, Codes and Cryptography*, 4(3):381–394, 1994. [Citado en pág. 91]