Universidad de Valladolid

Facultad de Ciencias

Trabajo Fin de Grado

Grado en Matemáticas

El problema del viajante con grafos

Autor: Ismael Crehuet Lucas

Tutor/es: Diego Ruano Benito

Índice

1. Introducción

Este Trabajo Fin de Grado en Matemáticas aborda el problema del viajante, más conocido por su denominación en inglés, traveling salesman problem (TSP) que responde a la siguiente pregunta: dada una lista de posibles ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible para visitar cada ciudad exactamente una vez y al finalizar regresar a la ciudad origen?

El origen del problema del viajante parece encontrarse en un manual para vendedores ambulantes de 1832 que menciona el problema e incluye viajes de ejemplo por Alemania y Suiza. Fue formulado matemáticamente en el siglo XIX por William Rowan Hamilton y Thomas Kirkman y desde entonces, este problema ha sido estudiado por diversos expertos en la materia.

El problema es modelizado mediante grafos, cuya teoría se desarrollará a lo largo del trabajo. El problema del viajante es muy difícil de resolver en la práctica de forma efectiva y en un tiempo razonable, a pesar de la apariencia del mismo. Sin embargo, existen algoritmos aproximados que dan una solución muy cercana a la solución exacta. Resolverlo teóricamente es sencillo, mediante la fuerza bruta, que es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

Este es un problema clásico de optimización combinatoria, que es un campo de las matemáticas aplicadas fuertemente relacionado con la investigación operativa, la teoría algorítmica y la teoría de la complejidad computacional.

El objetivo es el desarrollo de soluciones algorítmicas a problemas que requieren optimizar una función objetivo.

La importancia del problema del viajante reside en que múltiples problemas existentes en el mundo real que, aparentemente no tienen una relación clara entre sí, son formulados como un caso particular del mismo. Entre ellos, tiene aplicación en problemas de logística, de planificación o de robótica.

La logística es generalmente la organización e implementación detallada de una operación compleja, por ejemplo, un caso sobre esto son los problemas de gestión de la cadena de suministro donde se planifica, ejecuta y controla el flujo eficiente y efectivo y el almacenamiento de bienes, servicios e información relacionada entre el punto de origen y el punto de consumo.

La planificación es el proceso de toma de decisiones para alcanzar un futuro deseado, teniendo en cuenta la situación actual y los factores internos y externos que pueden influir en el logro de los objetivos, como elegir los destinos de un vendedor, que debe considerar el costo y los posibles beneficios en el lugar de venta.

La robótica se ocupa del diseño, construcción, operación, estructura, manufactura y aplicación de los robots. Pueden formularse problemas de circuitos electrónicos como un caso particular del TSP.

El trabajo está organizado en tres partes:
En una primera parte, que involucra los dos primeros capítulos del trabajo, se enuncian los principales conceptos teóricos relacionados con la teoría de grafos. Un grafo es una estructura matemática (composición de un conjunto de objetos que se relacionan con otros a través de conexiones) que permite representar problemas de manera gráfica.

En el primer capítulo, se desarrollan los diferentes tipos de grafos existentes, como son los grafos simples o los multigrafos, que pueden ser tanto dirigidos como no dirigidos, los subgrafos y los grafos ponderados entre otros. Se enuncia un tipo de grafos especial, los grafos completos, que serán los grafos que describan el problema del viajante. Además, se verá la relación existente entre los grafos y las matrices e isomorfismos. También, se explicará la conexión en un grafo, esta última propiedad permite la definición de camino y de circuito, en particular, se describen los caminos y circuitos de Euler, y de Hamilton, cuya definición responde a la pregunta en cuestión.

En el segundo capítulo del trabajo, se aborda el concepto de árbol, que es un tipo especial de grafo y también un cocepto derivado de este, los árboles de expansión. Concretamente veremos qué son los árboles de expansión mínima, cuya definición es fundamental para el capítulo posterior y la resolución del problema. Expondremos dos algoritmos para encontrarlos, el algoritmo de Prim y el de Kruskal.

En la segunda parte de este trabajo, el tercer capítulo desarrolla la teoría de la complejidad computacional, veremos una breve introducción para recordar algunos conceptos, cómo el de algoritmo, donde hay que tener el cuenta dos fundamentos. El primero que dicho algoritmo sea general, es decir, que pueda resolver todos los problemas de una misma clase y el segundo que sea eficiente, es decir, que termine en un tiempo razonable. Para esto último, se hablará en este capítulo del concepto de tiempo de ejecución de un algoritmo.

Hablaremos de la clasificación de los tiempos de ejecución que pueden tener los algoritmos, así como de la diferencia entre los problemas de la clase P y la clase NP, se verá que el problema del viajante es un problema de clase NP-hard. Además, se ejemplifica algunos de los algoritmos de búsqueda y de ordenación más importantes y sencillos, que serán subrutinas para los algoritmos que resuelven el problema, para explicar el tiempo de ejecución, en el peor de los casos, de estos algoritmos. Se definen por último los algoritmos aproximados o heurísticos, que son algoritmos que encuentran soluciones casi óptimas en un tiempo razonable.

Finalmente, en la tercera parte de este trabajo, el último capítulo estudia el algoritmo de Christofides, uno de los algoritmos más conocidos para la resolución del TSP, es un algoritmo aproximado de aproximación polinomial y se basa en la teoría de grafos. También, se definen una serie de algoritmos, heurísticos y exactos, con los que se puede solucionar el problema del viajante.

Para finalizar, se proponen algunos ejemplos para visualizar y comprobar la diferencia y eficacia entre los algoritmos que se mencionan durante esta parte. Asimismo, se desarrolla un apéndice de programación, para ilustrar cómo y dónde se han realizado los distintos algoritmos empleados y los ejemplos que han sido expuestos en el proyecto.

Este Trabajo Fin de Grado trata sobre uno de los problemas más conocidos en el mundo de las matemáticas, el TSP, el cuál tiene muchas consecuencias en la vida real. Se desarrollan los prerrequisitos necesarios para comprenderlo de una manera adecuada y sencilla, desde la teoría de grafos hasta la teoría de la complejidad algorítmica, de forma autocontenida. Incluye una gran cantidad de ejemplos e ilustraciones que ayudan a comprender mejor tanto las definiciones, como los algoritmos que han sido explicado en los diferentes capítulos y contiene demostraciones teóricas, como por ejemplo el algoritmo de Prim, o el factor de aproximación del algoritmo de Christofides. Además, se han utilizado diferentes programas de sofware científico, y el código utilizado se encuentra disponible en el apéndice de programación. Lo cual nos ha permitido hacer una comparación, tanto del tiempo como de la exactitud de algunos de los diferentes métodos que han sido explicados para la resolución del problema.

2. Teoría de grafos

(Fuente principal: [13], fuentes secundarias: [3], [11]) Los grafos son estructuras discretas que consisten en vértices y aristas que conectan dichos vértices. Hay diferentes tipos de grafos, dependiendo de si las aristas tienen o no dirección, de si hay múltiples aristas que conectan el mismo par de vértices o de si se permiten lazos.

Existen problemas en casi todas las disciplinas que se pueden resolver utilizando la teoría de grafos. Por ejemplo, para representar la competencia de diferentes especies en un nicho ecológico, para representar quién influye a quién en una organización o para representar los resultados de los torneos. También, para modelar el conocimiento entre personas, la colaboración entre investigadores, las llamadas telefónicas entre números de teléfono o los enlaces entre sitios web. Así como para modelar hojas de ruta o la asignación de puestos de trabajo a los empleados de una organización. Con la teoría de grafos, se puede llegar a determinar cuando es posible caminar por todas las calles en una ciudad sin pasar dos veces por una, o también para encontrar el camino más corto que recorre varias ciudades.

2.1. Nociones básicas

Definición 1. *Un **grafo** (no dirigido) $G = (V, E)$ consiste en un conjunto no vacío V de vértices (también denominados nodos) y un conjunto E de pares (no ordenados) de vértices llamados aristas. Cada arista tiene asociado uno o dos vértices, llamados extremos (o puntos finales) de la arista.*

Observación 1. *Un grafo con un conjunto de vértices o el número de aristas infinito es llamado grafo infinito. En este trabajo, se utilizan únicamente los grafos finitos. En diferentes textos, se suele denominar al número de vértices, el orden del grafo, y al número de aristas, su tamaño.*

Ahora, por ejemplo, supongamos que una red está formada por centros de datos y enlaces de comunicación entre computadoras, podemos representar la ubicación de cada centro de datos por un punto y cada enlace de comunicaciones por un segmento de línea, como se muestra en la figura siguiente.

Figura 1: Fuente [13]

En general, se visualizan los grafos usando puntos que representan los vértices y segmentos de línea para representar las aristas.

Un grafo donde cada arista conecta dos vértices diferentes y donde no hay dos aristas que conecten el mismo par de vértices es llamado **grafo simple**. En un grafo simple, cada arista está asociada a un par no ordenado de vértices distintos, y ninguna otra arista está asociada a este mismo par. Por tanto, cuando se tiene una arista en un grafo simple asociada al par $\{u, v\}$, se puede decir sin ambigüedad, que $\{u, v\}$ es una arista del grafo.

Los grafos que tienen multiples aristas conectando los mismos vértices son llamados **multigrafos**. Cuando hay m aristas diferentes que conectan el mismo par no ordenado de vértices $\{u, v\}$, se dice que $\{u, v\}$ es una arista de multiplicidad m.

Una arista que une un vértice consigo mismo se llama bucle o **lazo**, los multigrafos pueden incluir tanto lazos, como posibles aristas múltiples conectando el mismo par de vértices.

Hasta ahora, los grafos que se han presentado son grafos no dirigidos. Sin embargo, se puede encontrar que sea necesario asignar direcciones a las aristas del grafo.

Definición 2. *Un **grafo dirigido** (o digrafo) $G = (V, E)$ consiste en un conjunto no vacío de vértices V y un conjunto E de aristas dirigidas (también denominadas arcos). Cada arista dirigida está asociada a un par ordenado de vértices. La arista dirigida asociada al par ordenado (u, v) se dice que empieza en u y termina en v.*

Cuando se representa un grafo dirigido con un dibujo lineal, se utiliza una flecha que apunta desde u hasta v para indicar la dirección de una arista que comienza en u y termina en v.

Cuando un grafo dirigido no tiene lazos ni aristas dirigidas múltiples, se dice grafo simple dirigido. Ya que este tiene como máximo una arista asociada a cada par ordenado de vértices, llamamos a (u, v) arista si existe una arista en el grafo que comienza en u y termina en v. Los grafos dirigidos que tienen aristas múltiples dirigidas desde un primer vértice a un segundo vértice (posiblemente el mismo) son llamados multigrafos dirigidos. Cuando hay m aristas dirigidas asociadas a un par de vértices (u, v), se dice que (u, v) es una arista de multiplicidad m. Un grafo con aristas dirigidas y no dirigidas es denominado grafo mixto.

Esta terminologia para los diferentes tipos de grafos se resume en la siguiente tabla.

Tipo	Arista	¿Múltiples aristas?	¿Lazos?
Grafo simple	No dirigida	No	No
Multigrafo	No dirigida	Si	Si
Grafo simple dirigido	Dirigida	No	No
Multigrafo dirigido	Dirigida	Si	Si
Grafo mixto	Dirigida y no dirigida	Si	Si

2.2. Modelos de grafos

Redes sociales: Los grafos son usados para modelar estructuras sociales basadas en diferentes tipos de relaciones. Los individuos u organizaciones son representados por vértices, y las relaciones entre estos por aristas. Como los grafos de relación y amistad, que sirven para representar cuando dos personas se conocen, una arista no dirigida es usada para conectar este tipo de grafos. No se utilizan aristas múltiples o lazos. Otro ejemplo son los grafos de influencia, en estudios de comportamiento grupal se observa que ciertas personas pueden influir en el pensamiento de otras, en ellos, se utilizan aristas dirigidas. También grafos de colaboración, donde dos personas se relacionan trabajando juntos de una manera particular, son grafos simples.

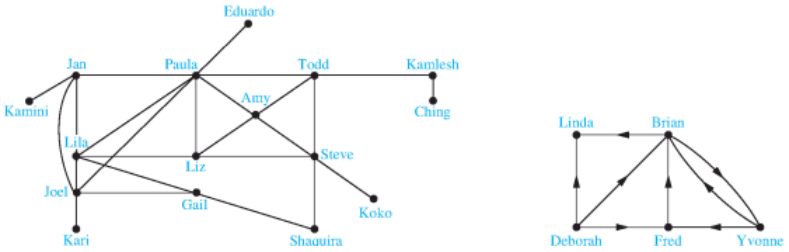


Figura 2: Fuente [13]

Redes de comunicación: se pueden modelar usando vértices para representar dispositivos y aristas para representar el tipo particular de enlace. Los grafos de llamadas pueden ser usados para modelar las llamadas telefónicas, un grafo múltiple dirigido puede ser usado para modelar llamadas donde cada número de teléfono es representado por un vértice y cada llamada telefónica por una arista dirigida.

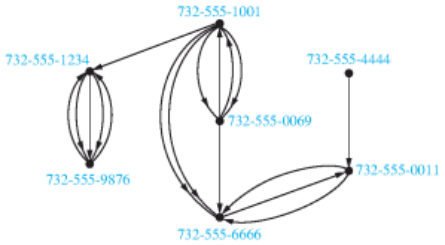


Figura 3: Fuente [13]

Redes de información: para modelar varias redes que vinculan tipos particulares de información o para modelar las citas en diferentes tipos de documentos. El grafo Web, puede ser modelado como un grafo dirigido donde cada página Web es representada por un vértice y donde una arista empieza en la página a y termina en la página b si hay un enlace en a que apunta a b. Grafos de citas pueden ser usados para representar citaciones en diferentes documentos, cada documento es representado por un vértice, y hay una arista desde un primero a un segundo documento si el primero cita al segundo en su lista de citaciones.

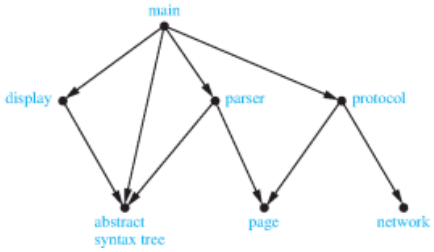


Figura 4: Fuente [13]

Diseño de software: una de las tareas más importantes es cómo estructurar un programa en diferentes módulos. Comprender cómo interactuan los diferentes módulos es esencial no solo para el diseño del programa, sino también para las pruebas y el mantenimiento del software resultante. Hay una arista dirigida desde un módulo (los vértices) hasta un segundo módulo si este depende del primero.

Redes de transporte: como rutas aéreas, se pueden modelar todos los vuelos de una areolínea usando aristas dirigidas entre los vértices, que representan el aeropuerto de salida y el de llegada. El grafo resultante es generalmente un multigrafo dirigido, ya que hay múltiples vuelos entre dichos aeropuertos para un mismo día. También se utilizan para redes viales.

Redes biológicas: como grafos de interacción de proteínas, donde, una interacción de proteínas en una célula viva sucede cuando dos o más proteínas se unen para realizar una función biológica en esta célula viva. Pueden ser modelados como grafos no dirigidos en los cuales una arista conectando los vértices, es decir, las proteinas, representa cada par de proteinas que interactúan entre sí.

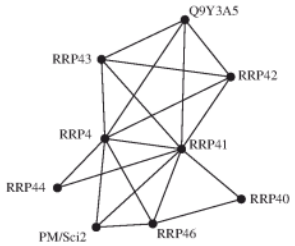


Figura 5: Fuente [13]

Torneos: para modelar torneos de todos contra todos (cada equipo juega contra todos los demás exactamente una vez) o torneos de eliminación.

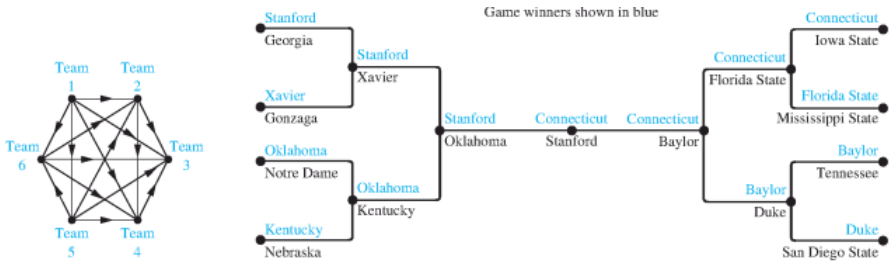


Figura 6: Fuente [13]

2.3. Terminología y tipos especiales de grafos

Definición 3. *Dos vértices u y v en un grafo no dirigido G son llamados **vértices adyacentes** (o vecinos) si u y v son extremos de una arista e en G. Cada arista e es llamada **arista incidente** con los vértices u, v. De otro modo, los vértices son independientes. Dos aristas son independientes si no tienen vértices en común.*

Para realizar un seguimiento de cuántas aristas son incidentes a un vértice, se tiene la siguiente definición.

Definición 4. *El **grado** de un vértice en un grafo no dirigido es el número de aristas incidentes con él (contando cada lazo dos veces). El grado de un vértice v se denota por* $\deg(v)$.

Cada arista contribuye dos a la suma de los grados de los vértices porque una arista es incidente con exactamente dos (posiblemente el mismo) vértices. Así pues, cuando sumamos los grados de todos los vértices de un grafo, obtenemos el siguiente teorema, denominado el teorema del apretón de manos (handshake theorem):

Teorema 1. *Sea $G = (V, E)$ un grafo no dirigido con m aristas. Entonces:*

$$2m = \sum_{v \in V} \deg(v).$$

Este teorema muestra que la suma de los grados de los vértices en un grafo no dirigido es par. Este hecho tiene muchas consecuencias, una de estas consecuencias es la siguiente.

Teorema 2. *Un grafo no dirigido tiene un número par de vértices de grado impar.*

Demostración. Sean V_1 y V_2 el conjunto de vértices de grado par y el conjunto de vértices de grado impar, respectivamente, en un grafo no dirigido $G = (V, E)$ con m aristas. Entonces,

$$2m = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v).$$

Ya que $\deg(v)$ es par para $v \in V_1$, el primer término en el lado derecho de la última igualdad es par. Además, la suma de los dos términos en el lado derecho de la última igualdad es par, porque esta suma es $2m$. Por eso, el segundo término es también par. Ya que todos los términos de esta última suma son impares, debe haber un número par de tales términos. Así, hay un número par de vértices de grado impar. □

La terminología de los grafos con aristas dirigidas refleja el hecho de que las aristas en los grafos dirigidos tienen dirección.

Definición 5. *Cuando (u, v) es una arista de un grafo dirigido G, u se dice que es adyacente a v y v adyacente desde u. El vértice u se llama vértice inicial de (u, v) y el vértice final o terminal de (u, v) es v.*

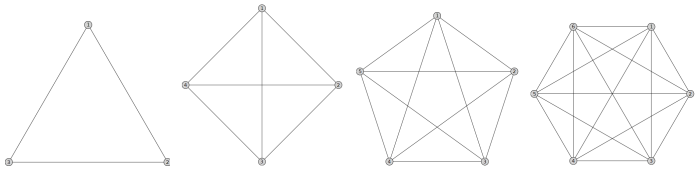
Como las aristas en los grafos con aristas dirigidas son pares ordenados, la definición del grado de un vértice puede ser refinada para reflejar el número de aristas con este vértice como el inicial y como el terminal.

Definición 6. *En un grafo con aristas dirigidas, el grado de entrada del vértice v, denotado por* $\deg^-(v)$*, es el número de aristas con v como vértice terminal; y el grado de salida de v,* $\deg^+(v)$*, el número con v como vértice inicial. (Nótese que un lazo contribuye uno en ambos).*

Como cada arista tiene un vértice inicial y terminal, la suma de los grados de entrada y de los grados de salida de todos los vértices es la misma, ambas son el número de aristas del grafo.

Definimos ahora algunos grafos simples especiales.

Definición 7. *Un **grafo completo** de n vértices, denotado por K_n, es un grafo simple que contiene exactamente una arista entre cada par de vértices distintos.*



(a) K_3 (b) K_4 (c) K_5 (d) K_6

Figura 7: Grafos completos

Definición 8. *Un ciclo C_n, $n \geq 3$, consiste en n vértices $v_1, ..., v_n$ y aristas $\{v_1, v_2\}$, $\{v_2, v_3\}$,..., $\{v_{n-1}, v_n\}$, $\{v_n, v_1\}$.*

A veces, un grafo tiene la propiedad de que sus vértices pueden dividirse en dos subconjuntos disjuntos tales que cada arista une un vértice de uno de estos conjuntos con un vértice del otro.

Definición 9. *Un grafo simple G es denominado bipartito si su conjunto de vértices V puede ser dividido en dos conjuntos disjuntos V_1 y V_2 tales que toda arista en el grafo une un vértice de V_1 y un vértice de V_2 (de modo que ninguna arista una dos vértices en V_1 o dos vértices en V_2). Así, (V_1, V_2) es una bipartición del conjunto de vértices V de G.*

Los grafos bipartitos se pueden usar para modelar muchos tipos de aplicaciones que implican hacer coincidir los elementos de un conjunto con elementos del otro.

Definición 10. *Un **emparejamiento** de un grafo $G = (V, E)$ es un subconjunto M de E de aristas independientes, ninguna de las cuales son lazos. Es decir, no hay dos aristas incidentes con el mismo vértice.*

Un emparejamiento máximo es un emparejamiento con el mayor número de aristas, y un emparejamiento es maximal si no está contenido en otro distinto. Un ***emparejamiento perfecto*** *en G es un subconjunto M de E, tal que todo vértice en V es adyacente a exactamente una arista en M, es decir, si todos los vértices de G son extremos de alguna arista de M. Un emparejamiento perfecto es máximo y maximal.*

En otras palabras, un emparejamiento es un subconjunto de aristas tal que si $\{s, t\}$ y $\{u, v\}$ son aristas distintas, entonces s, t, u y v son vértices distintos.

Ejemplo 1. *En este ejemplo, es posible visualizar algunos de los emparejamientos perfectos (no todos) que se pueden obtener del grafo ilustrado en la figura 8.*

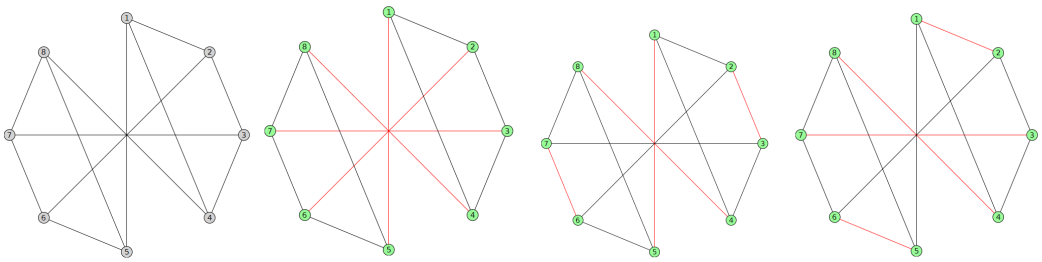
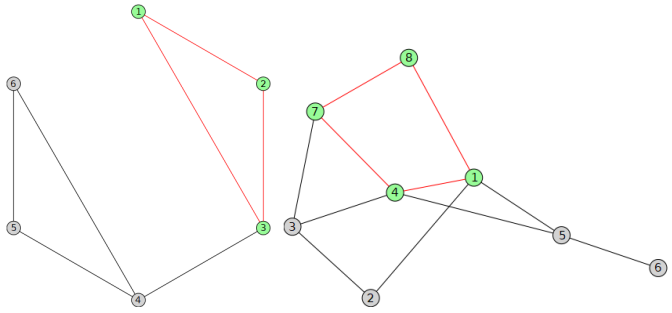


Figura 8: Emparejamientos perfectos del grafo $G = (V, E)$

A veces, se necesita solo una parte del grafo para resolver un problema. Para ello, se introducen las siguientes definiciones:

Definición 11. *Un* ***subgrafo*** *de un grafo $G = (V, E)$ es un grafo $H = (W, F)$, donde $W \subseteq V$ y $F \subseteq E$. Un subgrafo H de G es un subgrafo propio de G si $H \neq G$.*

Ejemplo 2. *La figura 9 muestra dos grafos, G y H, un subgrafo de G, y otro de H, cuyos vértices están en color verde y cuyas aristas están en color rojo.*



(a) Grafo G (b) Grafo H

Figura 9: Subgrafos

Dado un conjunto de vértices en un grafo, se puede formar un subgrafo con estos vértices y con las aristas del grafo que los unen.

Definición 12. *Sea $G = (V, E)$ un grafo simple. El* ***subgrafo inducido*** *por un subconjunto W del conjunto de vértices V es el grafo $H = (W, F)$, donde el conjunto de aristas F contiene una arista en E si y solo si ambos extremos de esta arista están en W. En otras palabras, si el subgrafo H contiene todas las aristas que unen en G dos vértices de W.*

Se puede generar un nuevo grafo añadiendo o quitando aristas, también quitando vértices, y las aristas incidentes con estos vértices.

2.4. Matrices e isomorfismos

2.4.1. Matrices adyacentes e incidentes

Dos tipos de matrices son utilizadas para representar grafos, una está basada en la adyacencia de los vértices, y la otra sobre la incidencia de vértices y aristas.

Definición 13. *Supongamos que $G = (V, E)$ es un grafo simple donde $|V| = n$. Supongamos también que los vértices de G se enumeran arbitrariamente como $v_1, v_2, ..., v_n$. La* ***matriz adyacente*** *A (o A_G) de G, con respecto a este listado de los vértices, es la matriz estándar de dimensión $n \times n$ con 1 como su (i, j)-ésima entrada cuando v_i y v_j son adyacentes, y con 0 sino son adyacentes. En otras palabras, si esta matriz adyacente es $A = [a_{ij}]$, entonces*

$$a_{ij} = \begin{cases} 1 & \text{si } \{v_i, v_j\} \text{ es una arista en } G \\ 0 & \text{en otro caso} \end{cases}$$

Una matriz adyacente esta basada en la ordenación de los vértices. Para un grafo simple, es simétrica, y como este no tiene lazos, cada entrada en la diagonal principal es 0.

Ejemplo 3. *Se tiene que la matriz adyacente que representa el grafo $G = (V, E)$ de la figura 10, es la matriz A estandar con 1 en su (i, j)-ésima entrada si los vértices v_i y v_j son adyacentes (definición 5) o 0 en otro caso, cuya dimensión es 5×5 (ya que $|V| = 5$),*

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

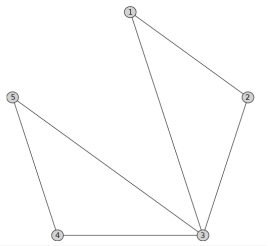


Figura 10: Grafo G

Las matrices adyacentes pueden ser usadas para representar grafos no dirigidos con lazos y con multiples aristas. Cuando multiples aristas conectan el mismo par de vértices v_i y v_j, o multiples lazos en el mismo vértice están presentes, la (i, j)-ésima entrada es

igual al número de aristas que estan asociadas a $\{v_i, v_j\}$. Todos los grafos no dirigidos tienen matrices adyacentes simetricas.

También se puede representar grafos dirigidos mediante matrices adyacentes, en este caso, dicha matriz no tiene porque ser simetrica. El número de unos en la fila i de la matriz es el grado de salida del vértice, $\deg^+(v)$, mientras que $\deg^-(v)$, el de entrada, corresponde al número de unos en la columna i, cuando v se corresponde con la fila/columna i.

Ejemplo 4. *Vemos por tanto, en este ejemplo, que la matriz que representa el grafo dirigido $G = (V, E)$ de la figura 11, que tiene 6 vértices, es la matriz A, no simétrica, ya que las aristas son dirigidas y puede haber una arista que vaya del vértice i al j, pero no del vértice j al i, de dimensión 6×6,*

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

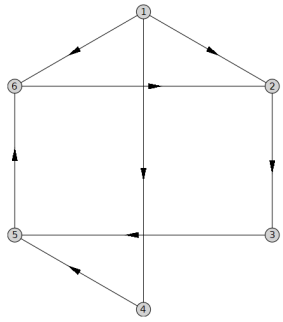


Figura 11: Grafo G

Otra manera común de representar grafos es mediante matrices de incidencia.

Definición 14. *Sea $G = (V, E)$ un grafo no dirigido. Suponemos que $v_1, v_2, ..., v_n$ son los vértices y $e_1, e_2, ..., e_m$ son las aristas de G. La* ***matriz de incidencia*** *con respecto al orden dado es la matriz de dimensión $n \times m$, $B = [b_{ij}]$, donde*

$$b_{ij} = \begin{cases} 1 & \text{cuando la arista } e_j \text{ es incidente con } v_i \\ 0 & \text{en otro caso} \end{cases}$$

La aristas multiples son representadas en la matriz de incidencia usando columnas con entradas idénticas porque estas aristas son incidentes con el mismo par de vértices. Los lazos son representados usando columnas con exactamente una entrada igual a uno, esta entrada corresponde al vértice que es incidente con el lazo.

Ejemplo 5. *Las columnas de la matriz de incidencia que representa al grafo $G = (V, E)$ de la figura 12 son las aristas $e_1 = \{1, 2\}$, $e_2 = \{1, 3\}$, $e_3 = \{1, 4\}$, $e_4 = \{2, 3\}$, $e_5 = \{2, 5\}$, $e_6 = \{2, 6\}$, $e_7 = \{3, 4\}$, $e_8 = \{4, 5\}$ y $e_9 = \{5, 6\}$, y las filas de la matriz son los vértices ordenados. Por tanto, la matriz de incidencia es la matriz B de*

dimensión 6×9 *siguiente,*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

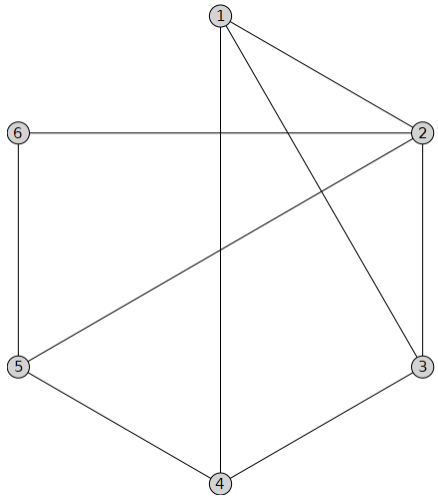


Figura 12: Grafo $G = (V, E)$

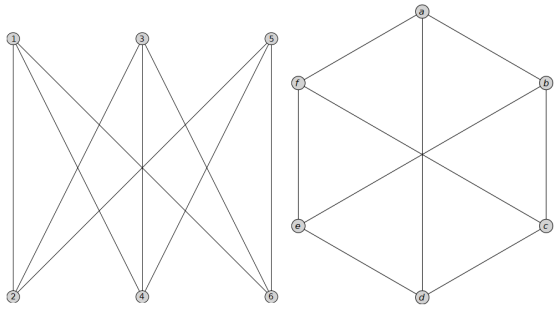
2.4.2. Isomorfismos de grafos

Hay una terminología útil para grafos con la misma estructura.

Definición 15. *Los grafos simples* $G_1 = (V_1, E_1)$ *y* $G_2 = (V_2, E_2)$ *son* ***grafos isomorfos*** *si existe una aplicación biyectiva* f *de* V_1 *a* V_2 *con la siguiente propiedad,* a *y* b *son adyacentes en* G_1 *si y solo si* $f(a)$ *y* $f(b)$ *son adyacentes en* G_2*, para todos* a *y* b *en* V_1*. Cada aplicación* f *es denominada isomorfismo.*

En otras palabras, si la aplicación $f : V_1 \to V_2$ preserva las adyacencias. Es decir, se tiene que $\{u, v\} \in E_1 \Leftrightarrow \{f(u), f(v)\} \in E_2$. La relación de isomorfismo entre grafos simples es una relación de equivalencia.

Ejemplo 6. *Sean* $G = (V, E)$ *y* $H = (W, F)$*. La aplicación* f *con* $f(1) = a$*,* $f(2) = b$*,...,* $f(6) = f$ *es un isoformismo entre* $V = \{1, 2, 3, 4, 5, 6\}$ *y* $W = \{a, b, c, d, e, f\}$ *que preserva la adyacencia. Por tanto, los grafos* G *y* H *de la figura 13 son isomorfos. Dos grafos isomorfos sólo se diferencian por la rotulación de los vértices (y, en general, por su representación gráfica).*



(a) Grafo G (b) Grafo H

Figura 13: Grafos isomorfos

A veces no es difícil mostrar que dos grafos no son isomorfos si podemos encontrar una propiedad que solo uno de los dos grafos tiene, pero esta se preserva por isomorfismo. Una propiedad preservada por isomorfismo se denomina parámetro invariante. Un invariante de un grafo G es un número asociado a G que toma el mismo valor para cada grafo isomorfo a G. Por ejemplo, los grafos simple isomorfos deben tener el mismo número de vértices, y el mismo número de aristas. Además, los grados de los vértices deben ser los mismos. Sin embargo, esto no necesariamente significa que los dos grafos sean isomorfos. Una forma útil de mostrar que la aplicación f es un isomorfismo es usando las matrices adyacentes, mostrando que ambas son iguales (salvo permutación).

2.5. Conectividad

Informalmente, se tiene que un camino es una sucesión de aristas que empieza en un vértice del grafo y recorre los vértices de este a través de las aristas del grafo.

Definición 16. *Sea n un entero no negativo y G un grafo no dirigido. Un* ***camino*** *de longitud n desde el vértice u hasta el vértice v en G es una sucesión de n aristas $e_1, ..., e_n$ para la cuál existe otra sucesión $x_0 = u, x_1, ...x_{n-1}, x_n = v$ de vértices tales que e_i tiene, para $i = 1, ..., n$, los extremos x_{i-1} y x_i.*

Un camino de longitud mayor que cero que comienza y termina en el mismo vértice se denomina ***circuito****. Un camino o circuito es simple si cada arista está una única vez en él. Cuando el grafo es simple, el camino (o el circuito) se denota por la sucesión de vértices ya que no hay ambigüedad.*

La notación por vértices $x_0, ..., x_n$ identifica un camino solo en cuanto a los vértices por los que pasa. En consecuencia, esto no especifica un único camino cuando hay más de uno que pasa a través de esta sucesión de vértices, lo que sucederá si y solo si hay múltiples aristas entre algunos vértices sucesivos en la lista.

Definición 17. *Sea n un entero no negativo y G un grafo dirigido. Un* ***camino*** *de longitud n desde el vértice u hasta el vértice v en G es una sucesión de aristas $e_1, ..., e_n$*

de G tales que e_1 está asociada con (x_0, x_1), e_2 con (x_1, x_2) y así sucesivamente, con e_n asociada a (x_{n-1}, x_n), donde $x_0 = u$ y $x_n = v$.

Cuando no hay multiples aristas, el camino puede ser denotado por su sucesión de vértices $x_0, ..., x_n$. Un camino de longitud mayor que cero que comienza y termina en el mismo vértice es denominado circuito. Un camino o circuito es simple si no contiene la misma arista más de una vez.

Definición 18. *La distancia, denotada por $d(u, v)$, entre dos vértices u y v en un grafo conexo (definición 19) se define como la longitud mínima de un camino entre estos vértices. La distancia en un grafo conexo es una métrica.*

2.5.1. Conectividad en grafos no dirigidos

Definición 19. *Un grafo no dirigido se dice **conexo** si hay un camino entre todos los pares distintos de vértices del grafo. Cuando esto no es así, se dice no conexo. Una desconexión de un grafo se produce cuando se eliminan vértices, aristas o ambos, para producir un subgrafo no conexo.*

Teorema 3. *Hay un camino simple entre todos los pares de vértices distintos en un grafo no dirigido conexo.*

Demostración. Sea u y v dos vértices distintos del grafo conexo no dirigido $G = (V, E)$. Como G es conexo, hay al menos un camino entre los vértices u y v. Si este camino contiene cada arista una única vez, entonces el camino elegido es simple. Si no, existe alguna arista que se repite en dicho camino, así pues, se eliminan las aristas recorridas entre esta arista repetida, y esto para cada arista repetida en el camino. Finalmente se obtiene un camino simple. Por tanto, para cada par de vértices distintos tenemos un camino simple. □

Una **componente conexa** de un grafo G es un subgrafo conexo de G que no es un subgrafo propio de otro subgrafo conexo de G. Esto es, una componente conexa de un grafo es el subgrafo conexo máximo de G. Un grafo G que no es conexo tiene dos o más componentes conexas que son disjuntas y tienen a G como su unión.

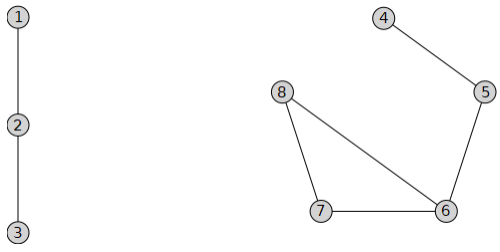
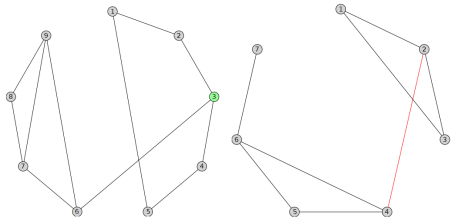


Figura 14: Componentes conexas del grafo $G = (V, E)$

Ejemplo 7. *En la figura 14, se representa un grafo $G = (V, E)$, con $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ que no es conexo, ya que no existe un camino entre todos los pares de vértices distintos de este grafo. El grafo tiene dos componentes conexas que tienen a G como su unión.*

A veces, la eliminación en un grafo de un vértice y todas sus aristas incidentes produce un subgrafo con más componentes conexas. Tales vértices son llamados vértices de corte o **articulaciones**. La eliminacíon de una articulación en un grafo conexo produce un subgrafo que no es conexo. Análogamente, una arista cuya eliminación produce un grafo con más componentes conexas que en el grafo original es llamada arista de corte o **puente**. No todos los grafos tienen articulaciones o puentes.

Ejemplo 8. *El vertice verde de la figura 15(a) es una articulación y la arista roja de la figura 15(b) es un puente.*



(a) Articulación (b) Puente

Figura 15: Ejemplo de articulación y de puente

2.5.2. Conectividad en grafos dirigidos

Hay dos nociones de conexión en grafos dirigidos, que dependen de si se consideran o no las direcciones de las aristas.

Definición 20. *Un grafo dirigido es fuertemente conexo si hay un camino desde el vértice a hasta el vértice b y viceversa, es decir, desde b hasta a.*

Para que un grafo dirigido sea fuertemente conexo debe haber una sucesión de aristas dirigidas desde cualquier vértice en el grafo a cualquier otro.

Definición 21. *Un grafo dirigido es debilmente conexo si hay un camino entre cada par de vértices en el grafo no dirigido subyacente.*

Esto es, un grafo dirigido es debilmente conexo si y solo si hay siempre un camino entre dos vértices cuando la dirección de las aristas es ignorada.

Los subgrafos de un grafo dirigido G que son fuertemente conexos pero no contenidos en un subgrafo fuertemente conexo más grande, esto es, el subgrafo máximo fuertemente conexo, son llamados las componentes fuertes de G. Nótese que si a y b son dos vértices en un grafo dirigido, sus componentes fuertes son, o bien la misma, o bien disjuntas.

2.5.3. Relación de los caminos con los isomorfismos y las matrices

Hay varias formas en que los caminos y los circuitos pueden ayudar a determinar si dos grafos son isomorfos. Por ejemplo, la existencia de un circuito simple de una longitud particular es un invariante útil que se puede usar para mostrar que dos grafos no son isomorfos. El número de caminos entre dos vértices en un grafo puede ser determinado usando su matriz adyacente.

Teorema 4. *Sea G un grafo con matriz adyacente A con respecto al ordenamiento $v_1, v_2, ..., v_n$ de los vértices del grafo (con aristas dirigidas o no dirigidas, con aristas multiples o lazos permitidos). El número de caminos diferentes de longitud r desde v_i hasta v_j, donde r es un entero positivo, es igual a la (i, j)-ésima entrada de A^r.*

Este teorema puede ser utilizado para encontrar la longtud del camino más corto entre dos vértices en el grafo, y para determinar si un grafo es conexo.

2.6. Caminos de Euler y de Hamilton

¿Se puede viajar a lo largo de las aristas de un grafo empezando en un vértice y volviendo a este atravesando cada arista exactamente una vez?, ¿Se puede viajar a lo largo de las aristas empezando en un vértice y volviendo a este mientras visitamos cada vértice una vez? A pesar de que estas preguntas parecen similares, la primera, que pregunta cuándo un grafo tiene un circuito de Euler, puede ser fácilmente contestada examinando los grados de los vértices. La segunda, que pregunta cuándo se tiene un circuito de Hamilton, es bastante difícil de resolver para la mayoría de grafos.

2.6.1. Caminos y circuitos de Euler

Definición 22. *Un **circuito de Euler** en un grafo G es un circuito simple que contiene cada arista de G. Un **camino de Euler** es un camino simple que contiene cada arista de G.*

Hay criterios sencillos para determinar cuando un multigrafo tiene un circuito o camino de Euler.

Teorema 5. *Un multigrafo conexo con al menos dos vértices tiene un circuito de Euler si y solo si sus vértices tienen grado par.*

Demostración. Para hacer esto, nótese que dicho circuito comienza con un vértice a y continua con un vértice incidente con a, digamos $\{a, b\}$. La arista contribuye uno al $deg(a)$. Cada vez que el circuito pasa por un vértice aporta dos al grado del vértice, porque el circuito entra por una arista incidente con este vértice y sale por otra. Finalmente el circuito termina donde empezó, contribuyendo uno al $deg(a)$. Por tanto, si un grafo conexo tiene un circuito de Euler, entonces cada vértice debe tener grado par. □

Ejemplo 9. *La ciudad de Königsberg está dividida en cuatro secciones separadas por el río. Hay siete puentes conectando las secciones. ¿Es posible empezar en alguna localización, cruzar todos los puentes una única vez, y volver al punto de partida? Esto*

no es posible ya que no hay un circuito de Euler, el grafo tiene cuatro vértices de grado impar. Este problema fue el comienzo de la teoría de grafos en 1736 cuando Leonard Euler lo resolvió.

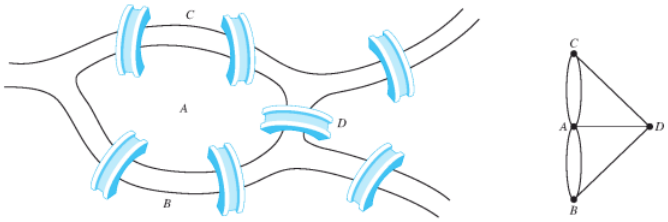
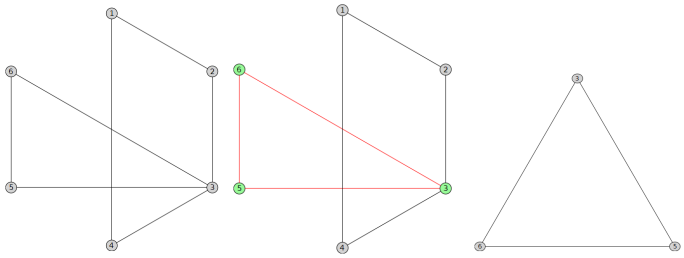


Figura 16: Fuente [13]: Representación de la ciudad de Königsberg

Veamos que dicha condicion es necesaria para la existencia de circuitos de Euler. Se supone que G es un multigrafo conexo en el cuál cada vértice tiene grado par. Se formará un circuito simple que empieza en un vértice arbitrareo a, sea $x_0 = a$, se elige una arista $\{x_0, x_1\}$ incidente con a, esto es posible ya que G es conexo. Se continua construyendo un camino simple $\{x_0, x_1\}, \{x_1, x_2\}, ..., \{x_{n-1}, x_n\}$, añadiendo sucesivamente aristas una a una al camino hasta que no se pueda añadir otra, esto ocurre cuando se alcanza un vértice para el cuál se tiene ya incluida todas las aristas incidentes.

El camino que se ha construido debe terminar porque el grafo tiene un número finito de aristas. El camino empieza en a con una arista de la forma $\{a, x\}$, y se muestra que debe terminar en a con una arista de la forma $\{y, a\}$. Para ver que debe terminar en a, nótese que cada vez que el camino pasa por un vértice con grado par, utiliza solo una arista para entrar a este vértice. Además, cada vez que se entra y se sale en un vértice de grado par, hay un número par (o ninguna) de aristas incidentes con este vértice que no se han usado todavía en nuestro camino. En consecuencia, mientras formamos el camino, cada vez que se entra en un vértice diferente a a, podemos salir de este. Esto significa que el camino solo puede acabar en a. Notar que el camino que se construye puede usar todas las aristas o puede que no si se vuelve a a por última vez antes antes de usar todas las aristas.

Se ha construido un circuito de Euler si se han utilizado todas las aristas. De lo contrario, se considera el subgrafo H obtenido de G al eliminar las aristas ya usadas y los vértices que no son incidentes con cualquier arista restante. Como G es conexo, H tiene al menos un vértice en común con el circuito que se ha eliminado. Sea w tal vértice. Cada vértice en H tiene grado par. Nótese que H puede ser no conexo. Empezando en w, al contruir un camino simple en H eligiendo aristas mientras sea posible, como se hizo en G, este camino debe terminar en w. Luego, se forma un circuito en G pegando el circuito en H con el circuito original en G a través de ese vértice w, esto es, sustituyendo en el circuito original el vértice w por el circuito obtenido en el subgrafo. Se continua el proceso hasta que todas las aristas sean usadas. Esto produce un circuito de Euler. Esta construcción es el algoritmo de Hierholzer.



(a) Grafo G (b) Subgrafo H de G (c) Subgrafo H

Figura 17: Caminos de Euler

Ejemplo 10. *Se considera, en el grafo G de la figura 17, el circuito $1, 2, 3, 4$. Tomamos el subgrafo H obtenido de G al eliminar las aristas ya usadas y los vértices que no son incidentes con cualquiera arista restante. El vértice en común es el vértice 3. Por tanto, se construye en H el circuito $3, 5, 6, 3$. Ahora se juntan ambos circuitos mediante ese vértice en común como sigue; $1, 2, 3, 5, 6, 3, 4, 1$ obteniendo un circuito de Euler en G.*

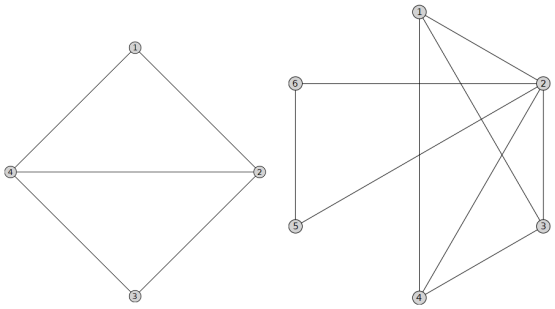
Los mejores algoritmos para encontrar un circuito de Euler en un grafo o para determinar que no existe dicho circuito tienen complejidad $O(m)$ donde m es el números de aristas del grafo. (ver sección 4.2). Como el algoritmo de Hierholzer, descrito anteriormente, cuya complejidad depende linealmente del número de aristas.

Teorema 6. *Un multigrafo conexo tiene un camino de Euler pero no un circuito de Euler si y solo si tiene exactamente dos vértices de grado impar.*

Demostración. Primero, se supone que un multigrafo conexo tiene un camino de Euler desde el vértice a hasta el vértice b, pero no un circuito de Euler. La primera arista contribuye uno al grado de a. Una contribución de dos al grado de a se hace cada vez que el camino pasa a través de a. La última arista contribuye uno al grado de b. Cada vez que el camino pasa a través de b hace una contribución de dos a su grado. Por lo tanto, a y b tienen grado impar. Cualquier otro vértice tiene grado par, porque el camino contribuye dos al grado de estos vértices cada vez que pasa a través de estos.

Recíprocamente, se supone que el grafo tiene exactamente dos vértices de grado impar, sean estos a y b. Se considera el grafo más grande formado por el grafo original con la adición de la arista $\{a, b\}$. Todo vértice de este grafo largo tiene grado par, luego tiene un circuito de Euler. La eliminación de la nueva arista produce un camino de Euler en el original. □

Ejemplo 11. *En el ejemplo de la ciudad, no puede existir un camino de Euler ya que los cuatro vértices tienen grado impar. Para la figura 18, G tiene exactamente dos vértices de grado impar, denotados por 2 y 4. Por tanto, hay un camino de Euler que debe tener a los vértices 2 y 4 como sus extremos. Uno de estos caminos de Euler es $4, 1, 2, 3, 4, 2$. Sin embargo, H no tiene un camino de Euler porque tiene cuatro de sus vértices (1, 2, 3, 4) de grado impar.*



(a) Grafo G (b) Grafo H

Figura 18: Caminos de Euler

Los caminos y circuitos de Euler pueden ser usados para resolver problemas como atravesar cada calle en un vecindario, cada carretera en una red de transporte o cada link en una red de comunicación exactamente una vez, entre muchos otros.

2.6.2. Caminos y circuitos de Hamilton

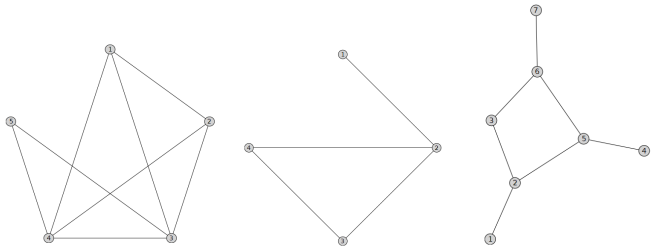
Se han desarrollado condiciones necesarias y suficientes para la existencia de caminos y circuitos que contienen cada arista de un multigrafo una vez exactamente. ¿Se puede hacer lo mismo para caminos y circuitos que contengan cada vértice del grafo exactamente una vez?

Definición 23. *Un camino simple en un grafo G que pasa a través de todos los vértices una sola vez es llamado* ***camino de Hamilton****, y un circuito con esta característica,* ***circuito de Hamilton****. Esto es, el camino simple $x_0, x_1, ..., x_n$ en el grafo $G = (V, E)$ es un camino de Hamilton si $V = \{x_0, x_1, ..., x_{n-1}, x_n\}$ y $x_i \neq x_j$ para $0 \leq i < j \leq n$, y el circuito simple $x_0, x_1, ..., x_n, x_0$ es un circuito de Hamilton si $x_0, x_1, ..., x_n$ es un camino de Hamilton.*

No se conocen criterios simples que den una condición necesaria y suficiente de la existencia de circuitos de Hamilton. Sin embargo, se conocen muchos teoremas que dan condiciones suficientes para la existencia de circuitos de Hamilton. Además de un algoritmo efectivo para carcularlo en caso de existencia. También, ciertas propiedades pueden ser usadas para mostrar que un grafo no tiene un circuito de Hamilton.

Por ejemplo, un grafo con un vértice de grado uno no puede tener un circuito de Hamilton, porque en dichos circuitos, cada vértice es incidente con dos aristas en el circuito. Además, si un vértice tiene grado dos, entonces ambas aristas que son incidentes con este vértice debe ser parte de cualquier circuito de Hamilton. También, nótese que cuando un circuito de Hamilton se está construyendo y este pasa a través de un vértice, entonces todas las aristas restantes incidentes con este vértice, aparte de las dos usadas en el circuito, puede ser eliminadas de la consideración. Además, un circuito de Hamilton no puede contener un circuito más pequeño dentro de él.

Ejemplo 12. *En la figura 19, se puede observar que G_1 tiene un circuito de Hamilton: $1, 2, 3, 5, 4, 1$. No hay un circuito de Hamilton en G_2, esto se puede ver observando que cualquier circuito que contenga todos los vértices debe contener la arista $\{1, 2\}$ dos veces, pero G_2 tiene un camino de Hamilton, $1, 2, 3, 4$. G_3 no tiene ni circuito ni camino de Hamilton, porque cualquier camino que contenga todos los vértices debe contener una de las aristas $\{1, 2\}$, $\{4, 5\}$ y $\{6, 7\}$ más de una vez.*



(a) Grafo G_1 (b) Grafo G_2 (c) Grafo G_3

Figura 19: Circuitos de Hamilton

Aunque no hay condiciones necesarias útiles para la existencia de circuitos de Hamilton, se han encontrado bastantes condiciones suficientes. Nótese que cuantas más aristas tiene el grafo, más probable es que tenga un circuito de Hamilton. En consecuencia, esperaríamos que hubiera condiciones suficientes para la existencia de circuitos de Hamilton que dependieran de que los grados de los vértices fueran lo suficientemente grandes.

Enunciamos dos de las condiciones suficientes más importantes, el teorema de Dirac y el teorema de Ore.

Teorema 7. *Si G es un grafo simple con $n \geq 3$ vértices tal que el grado de todos los vértices en G es al menos $n/2$, entonces G tiene un circuito de Hamilton.*

Teorema 8. *Si G es un grafo simple con n vértices, $n \geq 3$, tal que $deg(u) + deg(v) \geq n$ para todo par de vértices no adyacentes, entonces G tiene un circuito de Hamilton.*

Sin embargo, estos teoremas no proporcionan condiciones necesarias para la existencia de un circuito de Hamilton como se puede comprobar con el siguiente ejemplo.

Ejemplo 13. *El ciclo C_5 tiene 5 vértices, $n = 5$, y el grado de cualquiera de sus vértices es 2, es decir, tenemos que $deg(u) < \frac{n}{2}$ y $deg(u) + deg(v) = 4 < n$ para vértices u y v de C_5. Por tanto, no se cumplen las condiciones de los teoremas 7 y 8, respectivamente. Sin embargo, si existe un circuito de Hamilton para este ciclo, ya que se puede ver claramente que existe un circuito simple que pasa una sola vez a través de cada vértice del ciclo.*

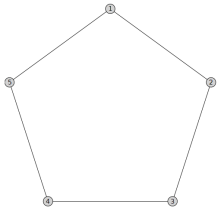


Figura 20: Ciclo C_5

Los mejores algoritmos para encontrar un circuito de Hamilton en un grafo o determinar que no existe dicho circuito tienen complejidad de tiempo exponencial (ver sección 4.2) en el peor de los casos (en el número de vértices del grafo).

Una de las aplicaciones de los circuitos de Hamilton, es el famoso problema del vendedor ambulante o problema del viajante (sección 5) del que trata este trabajo. Este problema pregunta cuál es la ruta más corta que debe tomar un viajante para visitar un determinado conjunto de ciudades. Este problema se reduce a encontrar un circuito de Hamilton en un grafo completo (definición 7), las ciudades (vértices) que tiene que visitar y las rutas (aristas) entre estas ciudades, tal que el peso total de sus aristas (ver definición 24) sea lo más pequeño posible, es decir, la suma total de los kilometros recorridos, donde se especifica en cada arista el número de kilometros entre dos ciudades, sea la más pequeña posible.

En este problema se presentan $n!$ rutas posibles, siendo n el número de ciudades, aunque se puede simplificar ya que dada una ruta, no importa el punto de partida y esto reduce el número de rutas a examinar en un factor n, quedando por tanto $(n-1)!$. Además, como no importa la dirección en que se desplace el viajante, el número de rutas a examinar se reduce de nuevo en un factor de 2. Por tanto, se consideran $\frac{(n-1)!}{2}$ rutas posibles. Se tiene una complejidad $O(n!)$ (ver sección 4.2).

2.7. Grafos ponderados

Muchos problemas pueden ser modelados usando grafos con pesos o distancias asignadas en sus aristas. Como por ejemplo el problema anteriormente expuesto. Para ello, presentamos dos definiciones.

Definición 24. *Un **grafo ponderado** o pesado es un grafo $G = (V, E)$ cuyas aristas tienen un número asignado, el peso de la arista.*

En el caso de que G sea un grafo ponderado, un emparejamiento perfecto (definición 10) de peso mínimo M de G es el emparejamiento perfecto cuyo peso total, es decir, la suma de los pesos de las aristas de M, es el menor de todos los emparejamientos perfectos de G.

Definición 25. *La longitud o distancia de un camino en un grafo ponderado es la suma de los pesos de las aristas de este camino.*

3. Árboles

(Fuentes: [3], [13]) Supongamos que se quieren comunicar n nodos utilizando una red de interconexión que tenga el menor número posible de enlaces. Para ello, se utilizan los árboles, que son un tipo particular de grafos. También se utilizan para construir algoritmos eficientes que encuentran elementos en una lista o en algoritmos que construyen códigos eficientes para ahorrar costos en la transmisión y en el almacenamiento de datos. Así como para modelar procedimientos llevados a cabo utilizando una secuencia de decisiones.

Estos tipos de árboles son llamados árboles de decisión, utilizados para el diseño de algoritmos y estructuras de datos. La construcción de estos modelos puede ayudar a determinar la complejidad computacional de los algoritmos basados en una secuencia de decisión. Por ejemplo, usando árboles ponderados podemos desarrollar algoritmos para construir redes que contengan el conjunto menos costoso, son los llamados árboles generadores de coste mínimo o de expansión mínima.

También, los arboles son usados en juegos para determinar una estrategia a seguir, para representar cuadros genealógicos, etc.

3.1. Nociones básicas

Definición 26. *Un **árbol** es un grafo no dirigido conexo sin circuitos simples.*

Un árbol no puede tener circuitos simples (definición 16), porque si tuviese y $\{u, v\}$ fuese una arista perteneciente a un circuito, entonces el grafo $G-\{u, v\}$ sería aún conexo y tendría menor número de aristas que G, además, un árbol no puede contener aristas múltiples o lazos.

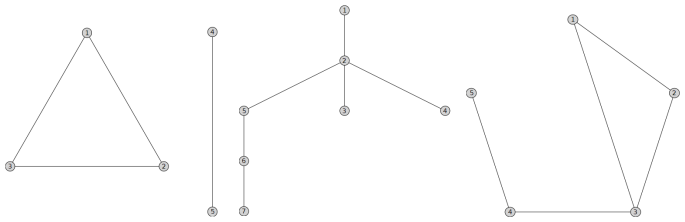
Un grafo conexo de orden n (observación 1) debe tener al menos $n-1$ aristas. En este sentido, los árboles son los grafos conexos de tamaño más pequeño, ya que un grafo conexo sin circuitos de longitud n tiene tamaño $n-1$. Además, cada arista de un árbol es un puente, sin embargo, un vértice de un árbol es una articulación si y solo si $\deg(v) > 1$.

Un grafo no conexo sin circuitos simples es un bosque y tienen la propiedad de que cada una de sus componentes conexas es un árbol.

Ejemplo 14. *Para la figura 21, G_1 no es un árbol porque no es conexo (no hay un camino para los vértices 3 y 5). G_2 es un árbol, porque es un grafo conexo sin circuitos simples. G_3 no es un árbol porque $1, 2, 3, 1$ es un circuito simple en el grafo.*

Teorema 9. *Un grafo no dirigido es un árbol si y solo si hay un único camino simple entre dos de sus vértices.*

Demostración. Se asume que T es un árbol. Entonces T es un grafo conexo sin circuitos simples. Sean x e y dos vértices de T. Como T es conexo, hay un camino simple entre x e y (teorema 3). Es más, este camino debe ser único, porque si hubiera un segundo camino así, el camino formado por la combinación del primero desde x hasta y seguido



(a) Grafo G_1 (b) Grafo G_2 (c) Grafo G_3

Figura 21: Ejemplo de árboles

del camino de y a x obtenido invirtiendo el orden en el primero, formaría un circuito. Esto implica que hay un circuito simple en T, por tanto, hay un único camino simple entre dos de los vértices del árbol.

Reciprocamente, se asume que hay un camino simple entre dos de los vértices del grafo T. Luego T es conexo, porque hay un camino entre dos vértices cualesquiera (teorema 3). Además, T no puede tener circuitos simples. Para ver esto, se supone que T tiene un circuito simple que contiene los vértices x e y. Entonces habría dos caminos simples entre x e y, ya que un circuito simple está formado por un camino simple desde x hasta y y por un segundo camino simple de y a x. Por eso, un grafo con un único camino simple entre dos vértices cualesquiera es un árbol. □

3.2. Árboles de expansión

Definición 27. *Sea G un grafo simple. Un **árbol de expansión** (o árbol generador) de G es un subgrafo de G que es un árbol que contiene cada vértice de G.*

Un grafo simple con un árbol de expansión debe ser conexo, ya que existe un camino en el árbol de expansión entre dos vértices cualesquiera. Lo recíproco también es cierto, es decir, para todo grafo simple conexo, existe un árbol de expansión en él.

Ejemplo 15. *El grafo $G = (V, E)$ es conexo, pero no es un árbol porque contiene circuitos simples en él. Suprimiendo la arista $\{a, e\}$, esto elimina un circuito simple, y el subgrafo resultante es aún conexo y todavía contiene todos los vértices de G. Ahora, suprimiendo la arista $\{e, f\}$, se elimina un segundo circuito simple, aún se conservan los vértices y la conexión. Finalmente, suprimiendo la arista $\{c, g\}$, se produce un grafo simple sin circuitos simples. Este subgrafo conexo es un árbol de expansión, porque es un árbol que contiene todos los vértices del grafo G, no es el único árbol de expansión de $G = (V, E)$.*

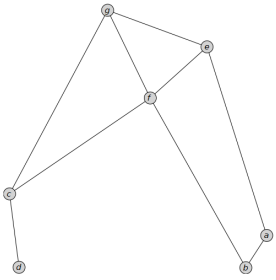
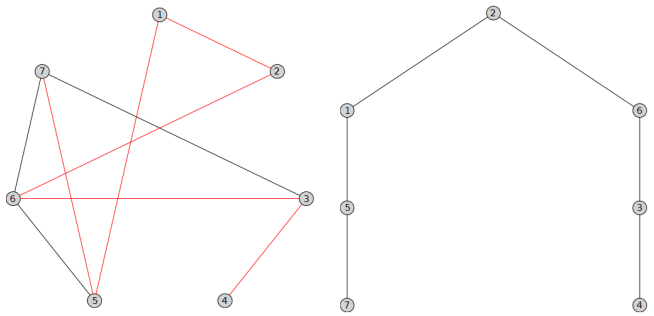


Figura 22: $G = (V, E)$

Otra representación del grafo G de la figura 22, se tiene en la figura 23, donde las aristas rojas forman un árbol de expansión (diferente al anterior).



(a) Grafo $G = (V, E)$ (b) Árbol de expansión

Figura 23: Árbol de expansión

Teorema 10. *Un grafo simple es conexo si y solo si tiene un árbol de expansión.*

Demostración. Primero, suponemos que un grafo simple tiene un árbol de expansión T, luego T contiene todos los vértices de G. Además, hay un camino en T entre dos vértices cualesquiera. Como T es un subgrafo de G que contiene todos los vértices de G, hay un camino en G entre dos vértices cualesquiera. Luego, G es conexo.

Reciprocamente, suponemos que G es conexo. Si G no es un árbol, debe contener un circuito simple. Se elimina una arista de uno de estos circuitos simples. El subgrafo resultante de eliminar dicha arista tiene una arista menos pero todavía contiene todos los vértices de G y es conexo porque cuando dos vértices están conectados por un camino que contiene la arista eliminada, están conectados también por un camino que no contiene esta arista. Se puede construir un camino de este tipo insertando en el camino original, en el punto donde estaba la arista eliminada, el circuito simple con esta arista eliminada. Si este subgrafo no es aún un árbol, tiene un camino simple, entonces como antes, se elimina una arista que está en un circuito simple. Se repite este proceso hasta que no queden circuitos simples. Esto es posible porque solo hay un número finito de aristas en el grafo. El proceso termina cuando no queden circuitos simples. Se produce un árbol ya que el grafo se mantiene conexo cuando las aristas son eliminadas. Este árbol es un árbol de expansión porque contiene todos los vértices de G. □

Ejemplo 16. *Los árboles de expansión son importantes en redes de datos. Por ejemplo, juegan un papel importante en la multidifusión a través de las redes de protocolo de Internet (IP). Con la multidifusión IP, un ordenador envia una copia simple de datos sobre la red, y a medida que los datos llegan a los routers intermedios, los datos se reenvían a uno o más routers para que, en última instancia, todos los ordenadores receptores en sus diversas subredes reciban los datos.*

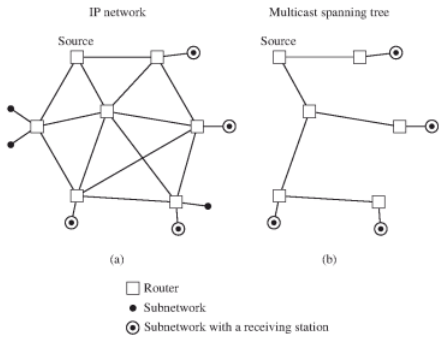


Figura 24: Fuente [13]

Para que los datos lleguen a los ordenadores receptores lo más rápido posible, no debe haber bucles (que en la terminología de grafos son cicuitos) en la ruta que toman los datos a través de la red. Esto es, una vez los datos han alcanzado un router particular, los datos no deben volver a este. Para evitar bucles, los routers de multidifusión usan algoritmos para construir un árbol de expansión, que tiene las fuentes de multidifusión, los routers, y las subredes que contienen ordenadores receptores como vértices, con aristas que representan los enlaces entre ordenadores y/o routers. Como se muestra en la figura 24.

3.3. Árboles de expansión mínima

Definición 28. *Un **árbol de expansión mínima** en un grafo ponderado conexo es un árbol de expansión que tiene la menor suma posible de pesos de sus aristas.*

Veamos dos algoritmos para construir árboles de expansión mínima. Ambos proceden añadiendo sucesivamente aristas de menor peso a partir de aquellas aristas con una propiedad específica que aún no se han utilizado. Son procedimientos que hacen una elección óptima en cada uno de sus pasos y producen soluciones finales óptimas, este tipo de algoritmos son denominados algoritmos egoistas.

El primero, el **algoritmo de Prim**, comienza eligiendo cualquier arista con el menor peso, colocándola en el árbol de expansión. Se añaden sucesivas aristas de peso mínimo que son incidentes a un vértice que esté ya en el árbol, nunca formando un circuito simple con estas aristas ya en el árbol. Se para cuando $n - 1$ aristas han sido añadidas.

Nótese que la elección de una arista para agregar en una etapa del algoritmo no está determinada cuando hay más de una arista con el mismo peso que satisface los criterios apropiados. Se necesita un orden en las aristas para hacer que las elecciones sean deterministas, el procedimiento se puede hacer determinista de una forma sencilla, como por ejemplo, elegir dichas aristas de mismo peso por orden alfabético. Nótese también que puede haber más de un árbol de expansión mínima para un grafo simple ponderado conexo dado.

procedure Prim(G: weighted connected undirected graph with n vertices)
T := a minimum-weight edge
for $i := 1 \to n-2$ **do**
    e := an edge of minimum weight incident to a vertex in T and not forming a simple circuit in T if added to T
    T := T with e added
end for
return T {T is a minimum spanning tree of G}

Teorema 11. *El algoritmo de Prim produce un árbol de expansión mínima en un grafo ponderado conexo.*

Demostración. Sea G un grafo ponderado conexo. Se supone que las sucesivas aristas elegidas en el algoritmo de Prim son e_1,e_2,...e_{n-1}. Sea S el árbol con e_1,e_2,...e_{n-1} como sus aristas, y sea S_k el árbol con e_1,e_2,...e_k como sus aristas. Sea T el árbol de expansión mínima de G conteniendo las aristas e_1,e_2,...e_k donde k es el mayor entero con la propiedad de que existe un árbol de expansión mínima que contiene las primeras k aristas elegidas en el algoritmo de Prim. El teorema finaliza si se demuestra que $S = T$.

Supongamos que $S \neq T$, así que $k < n-1$. Por tanto, T contiene e_1,e_2,...e_k pero no e_{k+1}. Consideremos el grafo formado por T junto con e_{k+1}. Ya que este grafo es conexo y tiene n aristas, demasiadas para ser un árbol, debe contener un circuito simple. Este circuito simple debe contener e_{k+1} porque no hay un circuito simple en T. Además, debe haber una arista en el circuito simple que no pertenece a S_{k+1} porque S_{k+1} es un árbol. Comenzando en un extremo de e_{k+1} que sea también un extremo de una de las aristas e_1,e_2,...e_k, y siguiendo el circuito hasta llegar a una arista que no esté en S_{k+1}, se puede encontrar una arista e que no esté en S_{k+1} que tenga un extremo que también sea extremo de una de las aristas e_1,e_2,...,e_k.

Eliminando la arista e de T y añadiendo e_{k+1}, obtenemos un árbol T' con $n-1$ aristas (es un árbol porque no tiene circuitos simples). Nótese que el árbol T' contiene las aristas e_1,e_2,...,e_k,e_{k+1}. Además, ya que e_{k+1} fue elegida en el algoritmo de Prim en el paso $(k+1)$, y e estaba también disponible en ese paso, el peso de e_{k+1} es menor o igual que el peso de e. De esta observación, se sigue que T' es tambien un árbol de expansión mínima, porque la suma de los pesos de sus aristas no excede la suma de los pesos de las aristas de T. Esto contradice la elección de k como el entero máximo para el que existe un árbol de expansión mínima que contiene e_1,e_2,...,e_k. Por tanto, $k = n-1$, y $S = T$. Así pues, el algoritmo de Prim produce un árbol de expansión mínima. □

El segundo, el **algoritmo de Kruskal**, elige una arista en el grafo con peso mínimo. Se añaden sucesivas aristas con peso mínimo que no formen un circuito simple con estas aristas ya elegidas. Se detiene después de que $n-1$ aristas hayan sido seleccionadas.

procedure Kruskal(G: weighted connected undirected graph with n vertices)
T := empty graph
for $i := 1 \to n - 1$ **do**
    e := any edge in G with smallest weight that does not form a simple circuit when added to T
    T := T with e added
end for
return T {T is a minimum spanning tree of G}

En el primer algoritmo, las aristas de peso mínimo que son incidentes con un vértice en el árbol, y no forman un circuito, son elegidas. Mientras que, en el segundo, las aristas de peso mínimo que no son necesariamente incidentes con un vértice que esté ya en el árbol, y que no formen un circuito son elegidas.

El tiempo de ejecución del algoritmo de Kruskal es $O(mlog\,m)$ siendo m el número de aristas y n el número de vértices. Encontrar todas las aristas del grafo tiene complejidad $O(m)$, porque es una búsqueda (ver sección 4.3). Ordenar las aristas por su peso tiene complejidad $O(mlog\,m)$ por ser una ordenación (ver sección 4.4). Además, se toma $O(mlog\,m)$ de tiempo en fusiones y búsquedas.

Por tanto, se tiene que el tiempo total para el algoritmo de Kruskal es $O(m(log\,m + log\,m))$, es decir, $O(mlog\,m)$ omitiendo constantes. El tiempo de ejecución del algoritmo de Prim es $O(mlog\,n)$, procediendo de manera similar al anterior.

Es preferible usar el algoritmo de Kruskal para grafos que son dispersos, es decir, donde m es muy pequeño en comparación con $n(n-1)/2$, que es el número de pares de vértices del grafo.

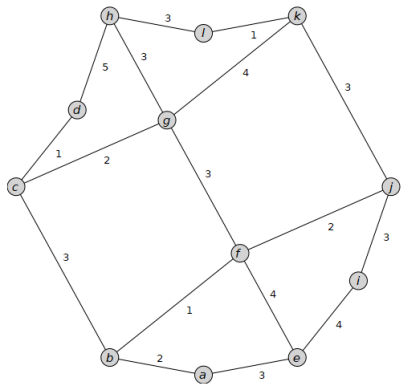
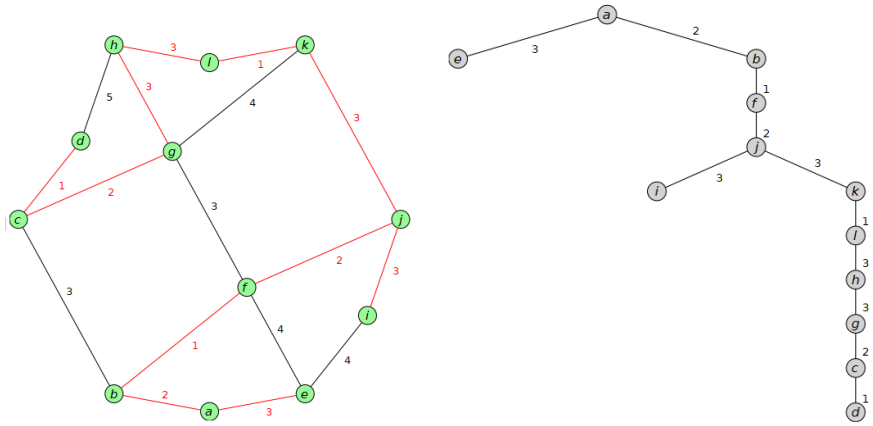


Figura 25: Grafo ponderado conexo $G = (V, E)$

Ejemplo 17. *En la figura 25 se muestra un grafo ponderado $G = (V, E)$. Usando el algoritmo de Prim y el algoritmo de Kruskal, encuentraremos un árbol de expansión mínima para dicho grafo (no son los únicos).*

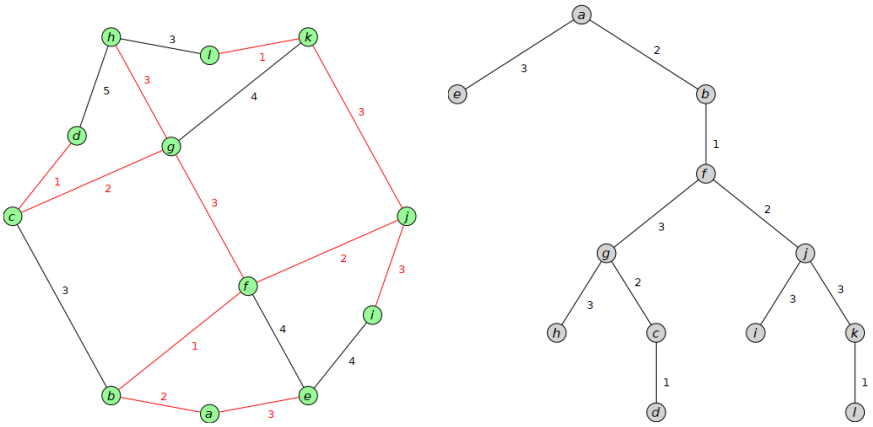
Procediendo mediante el algoritmo de Prim (figura 26), obtenemos el árbol de expansión mínima, $\{b, f\}$, $\{a, b\}$, $\{f, j\}$, $\{a, e\}$, $\{i, j\}$, $\{j, k\}$, $\{k, l\}$, $\{h, l\}$, $\{g, h\}$, $\{c, g\}$, $\{c, d\}$, *y la suma de los pesos de las aristas seleccinadas es 24.*



(a) Grafo $G = (V, E)$ (b) Árbol de expansión mínima

Figura 26: Algoritmo de Prim

Procediendo mediante el algoritmo de Kruskal (figura 27), un árbol de expansión mínima del grafo ponderado G *es:* $\{c, d\}$, $\{k, l\}$, $\{b, f\}$, $\{c, g\}$, $\{a, b\}$, $\{f, j\}$, $\{f, g\}$, $\{j, k\}$, $\{g, h\}$, $\{i, j\}$, $\{a, e\}$, *y de nuevo, la suma de los pesos de las aristas es 24.*



(a) Grafo $G = (V, E)$ (b) Árbol de expansión mínima

Figura 27: Algoritmo de Kruskal

4. Teoría de la complejidad computacional

4.1. Introducción

(Fuente: [3]) Un **algoritmo** es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. Es decir, un algortimo es, por tanto, una descripción de cómo se realiza una tarea, la resolución de un problema en concreto. Por procedimiento, se entiende que es una secuencia de instrucciones tales que hechas en el orden adecuado, llevan al resultado deseado. Un mismo algoritmo debe resolver todos los problemas de una misma clase, es decir, debe ser general. Además, un algoritmo es eficiente cuantos menos recursos en tiempo, espacio (de memoria) y procesadores consuma. Por lo general es difícil encontrar un algoritmo que reúna ambas por lo que se debe alcanzar un compromiso que satisfaga lo mejor posible los requisitos del problema.

Se debe considerar la identificación de los problemas que resolverá el algoritmo, se debe tratar problemas que puedan caracterizarse por su dimensión. Los algoritmos tienen una complejidad diferente, la cual afecta al tiempo de ejecución y a la ocupación de memoria, y posiblemente a una exactitud de los resultados diferente.

Un algoritmo puede ser expresado o formulado de maneras diferentes. Los algoritmos iterativos son aquellos que se encaminan hacia la solución paso a paso, un algoritmo es recursivo si se define a si mismo. En un algoritmo iterativo, una parte de él puede ejecutarse diversas veces, sin embargo, en un algoritmo recursivo, se realiza la reejecución del algoritmo completo desde el comienzo. Existen también los llamados procedimientos, que se pueden considerar algoritmos que son llamados desde dentro del principal.

Para resolver un problema determinado y una vez decidido el modelo para su tratamiento, pueden existir varios algoritmos. Una manera de comparar algoritmos que resuelven un mismo problema puede ser medir el tiempo que tardan en encontrar la solución. Como el tiempo que tarda el ordenador en encontrar la solución es proporcional al número de operaciones elementales que este debe efectuar, se suele llamar **tiempo de ejecución** a este número de operaciones. Otro parámetro es el espacio de memoria que se necesita para la ejecución.

Cuando se describe el número de operaciones $f(n)$, que efectúa un algoritmo, es preciso conocer un orden de magnitud para $f(n)$ válido para todo n salvo, quizás, un número finito de casos especiales. El análisis de un algoritmo, normalmente se hace analizando el caso más desfavorable, esto es, la rapidez que posee el algoritmo en el peor de los casos, es decir, para datos de entrada que hacen que el algoritmo tenga el peor rendimiento.

Existen varios tipos de problemas que se pueden plantear. Problemas de búsqueda, se trata de encontrar una cierta x en los datos que satisfaga la propiedad P. Problemas de transformación, se trata de transformar los datos de entrada para que satifagan

la propiedad P. Problemas de construcción, se trata de construir un conjunto c que satisfaga cierta propiedad P. Problemas de optimización, se trata de encontrar el mejor x que satisface la propiedad P. Problemas de decisión, se trata de decidir si los datos de entrada satisfacen la propiedad P. Para una discusión abstracta sobre la posibilidad de resolver el problema mediante algoritmos eficientes, resulta conveniente reformularlos en términos de problemas de decisión, es decir, buscar problemas cuya respuesta sea del tipo si/no.

4.2. Complejidad algorítmica

(Fuentes: [1], [5], [13], [16], [18]) La **complejidad** representa la cantidad de recursos (temporales) que necesita un algoritmo para resolver un problema, es decir, el tiempo de ejecución mínimo en el peor de los casos sobre todos los algoritmos de solución, en función del tamaño del problema, y por tanto, permite determinar la eficiencia de dicho algoritmo. Se estima comúnmente contando el número de operaciones elementales realizadas por el algoritmo, suponiendo que cada operación elemental requiere una cantidad fija de tiempo.

La medida del tiempo tiene que ser independiente de la máquina, del lenguaje de programación, del compilador y de cualquier otro elemento hardware o software que influya en el análisis. El coste depende del tamaño de los datos. Sin embargo, si el tamaño de los datos es grande lo que importa es el comportamiento asintótico de la eficiencia.

El tiempo requerido por una algoritmo es función del tamaño de los datos, por eso la complejidad temporal se expresa como $T(n)$. Otra característica importante es que no todos los datos poseen la misma importancia de cara a la complejidad algorítmica.

El orden de la función $T(n)$ expresa el comportamiento dominante para los datos de gran tamaño. Para poder determinar y expresar correctamente el comportamiento asintótico es conveniente disponer de una adecuada notación, donde una función g, es comparada con una función de referencia f.

Definición 29. *Sea la función $f : \mathbb{N} \to \mathbb{N}$, se dice que $f(n)$ es **$O(g(n))$** o que f es del orden de g si existen constantes positivas k y n_0 tal que, para $n \geq n_0$ se tiene que $f(n) \leq kg(n)$.*

Esto significa que f no crece más deprisa que g, donde n es el tamaño de entrada en unidades de bits necesarios para representarla. De esta forma acotamos superiormente el comportamiento asintótico de la función salvo constantes. Las complejidades algorítmicas se clasifican según el tipo de función que aparece en la notación O grande. Los **comportamientos asintóticos** de más frecuente aparición se pueden ordenar de menor a mayor crecimiento de la siguiente forma,

$$1 \ll log\, n \ll n \ll n\, log\, n \ll n^2 \ll n^3 \ll ..... \ll 2^n \ll n!$$

Ejemplo 18. *La definición del producto de dos matrices puede ser expresada como un algoritmo. Supongamos que $C = [c_{ij}]$ es la matriz $m \times n$, que es el producto de la matriz $A = [a_{ij}]$ de tamaño $m \times k$ y la matriz $B = [b_{ij}]$ de tamaño $k \times n$. Supongamos que $n > m$ y que $n > k$. El algoritmo basado en la definición del producto de dos matrices es expresado en el siguiente pseudocódigo.*

```
procedure matrix multiplication(A, B: matrices)
for i := 1 → m do
    for j := 1 → n do
        c_ij = 0
        for q := 1 → k do
            c_ij = c_ij + a_iq b_qj
        end for
    end for
end for
return C {C = [c_ij] is the product of A and B}
```

Hay mn entradas en el producto $A \times B$, para cada entrada tenemos un total de k multiplicaciones y $k-1$ sumas. Por tanto, un total de mnk multiplicaciones y $mn(k-1)$ sumas. Como hemos supuesto $n > m$ y $n > k$, tenemos que el algoritmo requiere $O(n^3)$ multiplicaciones y sumas. Sin embargo, existen otros algoritmos de menor tiempo de ejecución, como el algoritmo de Strassen, que requiere $O(n^{log_2 7})$ de tiempo.

Se dice que un algoritmo es de **tiempo constante** (también escrito como tiempo $O(1)$) si el valor de $T(n)$ está limitado por un valor que no depende del tamaño de la entrada. Por ejemplo, acceder a cualquier elemento en una matriz, encontrar el valor mínimo en una matriz ordenada. Sin embargo, encontrar el valor mínimo en un vector desordenado, es una operación de tiempo lineal, que toma tiempo $O(n)$, con n como longitud del vector. Sin embargo, si el número de elementos se conoce de antemano y no cambia aún se puede decir que dicho algoritmo se ejecuta en tiempo constante. A pesar del nombre, el tiempo de ejecución no tiene que ser independiente del tamaño del problema, pero un límite superior para el tiempo de ejecución tiene que estar limitado independientemente del tamaño.

Se dice que un algoritmo toma **tiempo logarítmico** cuando $T(n) = O(log\, n)$, es independiente de la base del logaritmo que aparece en la expresión, ya que en la fórmula de cambio de base se tiene $log_a\, x = \frac{log_b\, x}{log_b\, a}$, y $log_b\, a$ es una constante. Los algoritmos que toman tiempo logarítmico se encuentran comúnmente en operaciones con árboles binarios o cuando se usa la búsqueda binaria. Un algoritmo $O(log\, n)$ se considera altamente eficiente, ya que la relación entre el número de operaciones y el tamaño de la entrada disminuye y tiende a cero cuando aumenta n. Un ejemplo es la búsqueda dentro de un diccionario.

Se dice que un algoritmo toma **tiempo lineal** si su complejidad temporal es $O(n)$. Informalmente, esto significa que el tiempo de ejecución aumenta como máximo lineal-

mente con el tamaño de la entrada. Por ejemplo, un procedimiento que suma todos los elementos de una lista requiere un tiempo proporcional a la longitud de la lista. El tiempo lineal es la mejor complejidad de tiempo posible en situaciones en las que un algoritmo tiene que leer secuencialmente toda su entrada.

Se dice que un algoritmo se ejecuta en **tiempo subcuadrático** (también denominado tiempo log-lineal) si $T(n) = O(nlog^k n)$ para alguna constante positiva k. Por ejemplo, el ordenamiento de árboles binarios crea un árbol binario insertando uno por uno los elementos de la matriz de tamaño n. Dado que la operación de inserción en un árbol de búsqueda binaria toma tiempo $O(log\, n)$, todo el algoritmo toma tiempo $O(nlog\, n)$.

Se dice que un algoritmo es de **tiempo cuadrático** si $T(n) = O(n^2)$. Por ejemplo, los algoritmos de ordenamiento simples basados en la comparación son cuadráticos pero se pueden encontrar algoritmos más avanzados que son subcuadráticos. Ningún algoritmo de ordenamiento de propósito general se ejecuta en tiempo lineal pero la reducción de un tiempo cuadrático a uno subcuadrático es de gran importancia práctica.

Se dice que un algoritmo es de **tiempo polinómico** si su tiempo de ejecución está limitado por una expresión polinómica en el tamaño de la entrada para el algoritmo, es decir, $T(n) = O(n^k)$ para alguna constante positiva k. Los problemas para los que existe un algoritmo de tiempo polinómico pertenecen a la clase de complejidad P, que es central en el campo de la teoría de la complejidad computacional. El tiempo polinomial es sinónimo de manejable, factible, eficiente o rápido. Algunos ejemplos de algoritmos de tiempo polinomiales son el ordenamiento por selección, todas las operaciones aritméticas básicas o los emparejamientos en los grafos.

Se dice que un algoritmo es de **tiempo exponencial**, si $T(n)$ está limitado por $2^{poly(n)}$, donde $poly(n)$ es un polinomio en n, es decir, un algoritmo es de tiempo exponencial si $T(n)$ está limitado por $O(2^{n^k})$ para alguna constante k. También es de tiempo exponencial si $T(n)$ es del orden de $n!$. Los algoritmos de tiempo exponencial solo finalizan si la entrada es pequeña.

Definición 30. *Un problema es **tratable** si existe un programa que resuelve el problema y proporciona, en tiempo razonable, soluciones para entradas de tamaño grande. Se dice que un problema es **intratable** si ningún programa que resuelve el problema proporciona, en tiempo razonable, soluciones para entradas de tamaño grande.*

Otra forma de definirlo es que un problema es tratable si se puede resolver por un algoritmo en tiempo polinomial (la cota superior es polinomial) y un problema es intratable si no se puede resolver por un algoritmo en tiempo polinomial (la cota inferior es exponencial).

4.3. Algoritmos de búsqueda

(Fuente: [13]) El problema de ubicar un elemento en una lista ordenada ocurre en muchos contextos. Por ejemplo, un programa que verifica la ortografía de las palabras las busca en un diccionario, que es solo una lista ordenada de palabras. Los problemas de este tipo se denominan problemas de búsqueda.

El problema general de búsqueda se puede describir de la siguiente manera, ubicar un elemento x en una lista de elementos distintos $a_1, ..., a_n$, o determinar que dicho elemento no está en la lista. La solución a este problema de búsqueda es la ubicación del término en la lista que es igual a x y es 0 si x no está en la lista.

El **algoritmo de búsqueda líneal** comienza comparando x y a_1. Cuando $x = a_1$, la solución es la localización de a_1, es decir, 1. Cuando $x \neq a_1$, compara x con a_2. Si $x = a_2$, la solución es la ubicación de a_2, es decir, 2. Cuando $x \neq a_2$, compara a x con a_3. Continua este proceso, comparando x sucesivamente con cada término de la lista hasta encontrar una coincidencia, donde la solución es la localización de ese término, a menos que no ocurra ninguna coincidencia. Si se ha buscado en toda la lista sin encontrar x, la solución es 0. El pseudocódigo es el siguiente.

procedure linear search$(x, a_1, ..., a_n)$
$i := 1$
while $i \leq n$ and $x \neq a_i$ **do**
    $i := i + 1$
end while
if $i \leq n$ **then** location $:= i$
else location $:= 0$
end if
return location

El número de comparaciones usadas es la medida del tiempo de complejidad del problema. En cada paso del bucle en el algoritmo, se realizan dos comparaciones, una es $i \leq n$, para ver si se ha llegado al final de la lista, y otra es $x \neq a_i$, para comparar el elemento x con un término de la lista. Finalmente, se hace una comparación más fuera del ciclo. Por tanto, si $x = a_i$, se utilizan $2i + 1$ comparaciones. En el peor de los casos, se hacen $2n + 2$ comparaciones cuando el elemento no está en la lista, $2n$ comparaciones para determinar que x no es ningún a_i, para $i = 1, ..., n$, otra más para salir del bucle y una última fuera del bucle. Por tanto, una búsqueda lineal requiere $O(n)$ comparaciones en el peor de los casos.

El **algoritmo de búsqueda binaria** se puede usar cuando los términos aparecen en orden de tamaño creciente, procede comparando el elemento a ubicar con el término medio de la lista. Luego, la lista se divide en dos sublistas más pequeñas del mismo tamaño, o donde una de estas listas más pequeñas tiene un término menos que la otra. La búsqueda continúa restringiendo esta a la sublista apropiada basada en la comparación del elemento a localizar y el término medio.

Para buscar el entero x en la lista $a_1, ..., a_n$ donde $a_1 < a_2 < ... < a_n$, se comienza a comparar x con el término medio a_m de la lista, donde $m = \lfloor (n+1)/2 \rfloor$ ($\lfloor x \rfloor$ esta función suelo asigna a cada número el mayor número entero igual o menor que ese número x). Si $x > a_m$, la búsqueda de x se restringe a la segunda mitad de la lista, que es $a_{m+1}, ..., a_n$. Si x no es mayor que a_m, la búsqueda de x se restringe a la primera mitad de la lista, que es $a_1, ...a_m$. La búsqueda ahora se ha restringido a una lista con no más de $\lceil n/2 \rceil$ elementos ($\lceil x \rceil$ es la función techo, que a cada número asigna el número entero más próximo por exceso, es decir, el menor número entero igual o mayor que ese número x).

Usando el mismo procedimiento, se compara x con el término medio de la lista restringida. Luego se restringe la búsqueda a la primera o segunda mitad de la lista. Se repite este proceso hasta obtener una lista con un término. Luego se determina si este término es x. El pseudocódigo es el siguiente, para $a_1 < a_2 < ... < a_n$.

procedure binary search$(x, a_1, ..., a_n)$
$i := 1$
$i := n$
while $i < j$ **do**
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ **then** $i := m + 1$
    else $j := m$
    end if
end while
if $x = a_i$ **then** location $:= i$
else location $:= 0$
end if
return location

Este algoritmo procede estrechando sucesivamente la parte de la secuencia que se busca. En cualquier etapa dada, solo se están considerando los términos de a_i a a_j. Esto continúa hasta que solo queda un término. Cuando se hace esto, se hace una comparación para ver si este término es igual a x.

De nuevo, el número de comparaciones usadas es la medida del tiempo de complejidad del problema (ignorando el tiempo requerido para computar $m = \lfloor (i + j)/2 \rfloor$ en cada iteración del bucle). Para simplificarlo, se asume que hay $n = 2^k$ elementos en la lista donde k es un entero no negativo. Nótese que $k = log\, n$. (Si n no es una potencia de 2, esta lista puede ser considerada como parte de una lista más grande con 2^{k+1} elementos, donde 2^{k+1} es la potencia más cercana de 2 mayor que n.)

En cada paso del algoritmo, i y j, las localizaciones del primer y último término de la lista restringida, son comparadas para ver cuando esta lista restringida tiene más de un término. Si $i < j$, una comparación es realizada para determinar cuando x es más grande que el término medio.

En el primer paso, la busqueda está restringida a una lista con 2^{k-1} elementos. Hasta aquí, se han utilizado dos comparaciones. Este proceso continua, utilizando dos

comparaciones en cada paso para restringir la búsqueda en una lista con la mitad de los términos. En otras palabras, dos comparaciones son usadas cada vez que restringimos la lista hasta que la búsqueda se reduce a una lista con $2^1 = 2$ elementos. Finalmente, cuando queda un término en la lista, una comparación se usa para ver que no quedan términos adicionales, y una comparación más es usada para determinar si este término es x.

Por lo tanto, se requieren a lo sumo $2k + 2 = 2log\, n + 2$ comparaciones para realizar una búsqueda binaria cuando la lista que se busca tiene 2^k elementos (si n no es una potencia de 2, la lista original se expande a una lista con 2^{k+1} elementos, donde $k = \lfloor log\, n \rfloor$, y la busqueda requiere a lo sumo $2\lfloor log\, n \rfloor + 2$ comparaciones). Por lo tanto, en el peor de los casos, la búsqueda binaria requiere $O(log\, n)$ comparaciones. Por tanto el algoritmo de búsqueda binaria es más eficiente que el algoritmo de búsqueda lineal.

4.4. Algoritmos de ordenación

(Fuente: [13]) Ordenar elementos en una lista es un problema frecuente. Por ejemplo, para producir una guía telefónica, producir un directorio de canciones disponibles para descargar, para crear un diccionario o poner las direcciones en orden en una lista de correo electrónico para poder determinar si hay direcciones duplicadas. Existen muchos algoritmos de ordenación, y se desarrollan nuevos con mucha frecuencia. Se discutirán en esta sección tres algoritmos de ordenación, bubble sort, insertion sort y merge sort.

El **algoritmo bubble sort** es uno de los algoritmos más sencillos para ordenar elementos, pone una lista en orden creciente comparando sucesivamente elementos adyacentes, intercambiándolos si están en el orden incorrecto. Es expresado en el siguiente pseudocódigo, para una lista de $n \geq 2$ números reales $a_1, ..., a_n$.

```
procedure bubblesort(a_1, ..., a_n)
for i := 1 → n − 1 do
    for j := 1 → n − i do
        if a_j > a_{j+1} then
            interchange a_j and a_{j+1}
        end if
    end for
end for
{a_1, ..., a_n is in increasing order}.
```

Durante cada paso, el bubble sort compara sucesivamente los elementos adyacentes, intercambiándolos si es necesario. Cuando comienza el i-ésima paso, se garantiza que los $i - 1$ elementos más grandes estén en las posiciones correctas. Durante este paso, se utilizan $n - 1$ comparaciones. En consecuencia, el número total de comparaciones usadas en este algoritmo para ordenar una lista de n elementos es,

$$(n-1) + (n-2) + ... + 2 + 1 = \frac{(n-1)n}{2}$$

Nótese que este algoritmo siempre usa esta cantidad de comparaciones, porque continúa incluso si la lista se ordena por completo en algún paso intermedio. Por tanto, usa $(n-1)n/2$ comparaciones, esto tiene por tanto $O(n^2)$ de complejidad en el peor de los casos en términos del número de comparaciones utilizadas.

El **algoritmo insertion sort** es un algoritmo de ordenación simple, pero normalmente no es el más eficiente. Para ordenar una lista de n elementos, este algoritmo comienza en el segundo elemento, y compara este elemento con el primero y inserta este antes del primer elemento si no excede el primer elemento y después del primer elemento si excede el primer elemento. En este punto, los dos primeros elementos están en el orden correcto. El tercer elemento es comparado con el primero, y si este es más largo, es comparado con el segundo, este elemento es insertado en la posicion correcta entre los primeros tres elementos.

En general, en el paso j del algoritmo, el elemento j-ésimo de la lista es insertado en la posicion correcta de la lista de los $j-1$ elementos previamente ordenados. El elemento j-ésimo se compara sucesivamente con los $j-i$ elementos ya ordenados al principio de la lista hasta que se encuentra el primer elemento que no es menor que este elemento o hasta que se ha comparado con todos los $j-1$ elementos. El j-ésimo elemento se inserta en la posición correcta para que los primeros j elementos se ordenen. El algoritmo continua hasta que el último elemento este colocado en la posición correcta. Este algoritmo es expresado en el siguiente pseudocódigo, para una lista de $n \geq 2$ números reales $a_1, ..., a_n$.

```
procedure insertion sort(a_1, ..., a_n)
for j := 2 → n do
    i := 1
    while a_j > a_i do
        i := i + 1
        m := a_j
        for k := 0 → j − i − 1 do
            a_{j−k} = a_{j−k−1}
            a_i := m
        end for
    end while
end for
{a_1, ..., a_n is in increasing order}.
```

Este algoritmo inserta el j-ésimo elemento en la posición correcta entre los primeros $j-1$ elementos que ya están colocados en el orden correcto. En el peor de los casos, se requieren j comparaciones para insertar el j-ésimo elemento en la posición correcta. Por lo tanto, el número total de comparaciones usadas en este algoritmo para una lista de n elementos es,

$$2 + 3 + ... + n = \frac{n(n+1)}{2} - 1$$

Nótese que insertion sort puede usar muchas menos comparaciones si los elementos más

pequeños comenzaron al final de la lista. Sin embargo, tiene $O(n^2)$ de complejidad en el peor de los casos.

El algoritmo recursivo llamado **algoritmo merge sort** procesa iterativamente una lista en dos sublistas de igual longitud (o donde una sublista tiene un elemento más que la otra) hasta que cada sublista contenga un elemento. El procedimiento continúa juntando sucesivamente pares de listas, donde ambas listas están en orden creciente, en una lista más grande con elementos en orden creciente, hasta que la lista original quede en orden creciente.

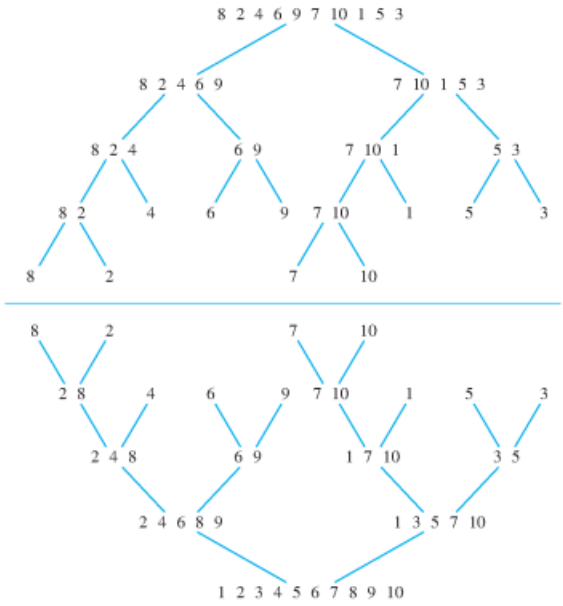


Figura 28: Fuente [13]

Ejemplo 19. *En la figura 28 se ilustra como funciona este algoritmo. Una ordenación por fusión comienza dividiendo sucesivamente la lista en dos hasta obtener elementos individuales. La ordenación se realiza fusionando sucesivamente pares de listas. En la primera etapa, los pares de elementos individuales se fusionan en listas de dos elementos en orden creciente. Luego se realizan fusiones sucesivas de pares de listas hasta obtener la lista completa, que se obtiene en orden creciente.*

También podemos describir recursivamente el merge sort. Para realizar una ordenación por fusión, dividimos una lista en dos sublistas de igual tamaño, ordenamos cada sublista usando el algoritmo de ordenación por fusión y luego fusionamos las dos listas. La versión recursiva del merge sort es, para una lista L de $n \geq 2$ elementos $a_1, ..., a_n$. Este algoritmo utiliza la subrutina *merge*, descrita porteriormente.

procedure mergesort($L = a_1, ..., a_n$)
if $n > 1$ **then**
  $m := \lfloor n/2 \rfloor$
  $L_1 := a_1, ..., a_m$
  $L_2 := a_{m+1}, ..., a_n$
  $L := merge(mergesort(L_1), mergesort(L_2))$
end if
{L is now sorted into elements in nondrecreasing order}.

Se necesita un algoritmo eficiente para fusionar dos listas ordenadas en una lista ordenada más grande para implementar el merge sort. Describimos este algoritmo. Se comienza con una lista vacía L. Se compara los elementos más pequeños de las dos listas. Se coloca el más pequeño de estos dos elementos en el extremo derecho de L y se elimina de la lista en la que estaba. A continuación, si una de L_1 y L_2 está vacía, se agrega la otra lista a L, lo que completa la fusión. Si ni L_1 ni L_2 están vacías, se repite este proceso. A continuación se muestra el pseudocódigo de este procedimiento.

procedure merge(L_1, L_2: sorted lists)
L:= empty list
while L_1 **and** L_2 are both nonempty **do**
  remove smaller of first elements of L_1 and L_2 from its list; put it at the right end of L
  if this removal makes one list empty **then**
    remove all elements from the other list and append them to L
  end if
end while
return L {L is the merged list with elements in increasing order}

El número de comparaciones usadas para fusionar dos listas ordenadas en el análisis del merge sort es como sigue, cada vez que la comparación de un elemento de L_1 y un elemento de L_2 se realiza, un elemento adicional es agregado a la lista L. Sin embargo, cuando L_1 o L_2 está vacía, no se necesitan más comparaciones. Por lo tanto, este algoritmo es menos eficiente cuando se realizan $m+n-2$ comparaciones, donde m y n son el número de elementos en L_1 y L_2, respectivamente, dejando un elemento en cada una de L_1 y L_2. La próxima comparación será la última necesaria, porque dejará una de estas listas vacía. Por lo tanto, este algoritmo no utiliza más de $m + n - 1$ comparaciones.

Lema 1. *Dos listas ordenadas con m elementos y n elementos se puede fusionar en una lista ordenada usando no más de $m + n - 1$ comparaciones.*

A veces, dos listas ordenadas de longitud m y n se pueden fusionar utilizando muchas menos comparaciones que $m+n-1$. Por ejemplo, cuando $m = 1$, se puede aplicar un procedimiento de búsqueda binaria para colocar el elemento de la primera lista en la segunda lista. Esto requiere solo $\lceil log\, n \rceil$ comparaciones, que es mucho menor que $m + n - 1 = n$. Por otro lado, para algunos valores de m y n, el lema anterior da la

mejor cota posible. Es decir, hay listas con m y n elementos que no se pueden fusionar con menos de $m+n-1$ comparaciones.

Ahora se puede analizar la complejidad del merge sort. En lugar de estudiar el problema general, se supone que n, el número de elementos en la lista, es una potencia de 2, digamos 2^m. Esto hará que el análisis sea menos complicado, pero cuando este no sea el caso, se pueden aplicar varias modificaciones que darán la misma estimación.

En la primera etapa del procedimiento de división, la lista se divide en dos sublistas, de 2^{m-1} elementos cada una. Este proceso continúa, dividiendo las dos sublistas con 2^{m-1} elementos en cuatro sublistas de 2^{m-2} elementos cada una. En general, hay 2^{k-1} listas en el paso $k-1$, cada una con 2^{m-k+1} elementos. Estas listas en el paso $k-1$ se dividen en 2^k listas en el paso k, cada una con 2^{m-k} elementos. Al final de este proceso, tenemos 2^m listas, cada una con un elemento en el paso m.

Comenzamos a fusionar combinando pares de las 2^m listas de un elemento en 2^{m-1} listas, cada una con dos elementos. Para hacer esto, se fusionan 2^{m-1} pares de listas con un elemento cada uno. La fusión de cada par requiere exactamente una comparación.

El procedimiento continúa, de modo que cuando se tienen 2^k listas, para $k = m, m-1, m-1, ..., 3, 2, 1$, cada una con 2^{m-k} elementos, se fusionan en 2^{k-1} listas, cada una con 2^{m-k+1} elementos. Para hacer esto, se necesitan un total de 2^{k-1} fusiones de dos listas, cada una con 2^{m-k} elementos. Pero, por el lema 1, cada una de estas fusiones puede llevarse a cabo utilizando como máximo $2^{m-k}+2^{m-k}-1 = 2^{m-k+1}-1$ comparaciones. Por lo tanto, pasar del paso k al $k-1$ se puede lograr utilizando como máximo $2^{k-1}(2^{m-k+1}-1)$ comparaciones. La suma de todas las estimaciones muestra que el número de comparaciones requeridas para el ordenamiento combinado es como máximo (porque $m = log\, n$ y $n = 2^m$),

$$\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1}-1) = \sum_{k=1}^{m} 2^m - \sum_{k=1}^{m} 2^{k-1} = m2^m - (2^m - 1) = nlog\, n - n + 1$$

Teorema 12. *El número de comparaciones necesario para una lista de n elementos en el merge sort es* $O(n \log n)$.

4.5. NP-completitud

(Fuente: [5]) El concepto de tiempo polinomial conduce a varias clases de complejidad en la teoría de la complejidad computacional. Algunas clases importantes definidas usando el tiempo polinomial son la **clase** P, que es la clase de complejidad de los problemas que se pueden resolver en tiempo $O(n^k)$ o la **clase** NP, esta clase consiste en aquellos problemas que son "verificables" en tiempo polinomial, lo que se quiere decir aquí es que si de alguna manera nos dieran un "certificado" de una solución, entonces podríamos verificar que el certificado es correcto en tiempo polinomial.

Como por ejemplo, en un problema de circuitos de Hamilton, dado un grafo dirigido $G = (V, E)$, un certificado podría ser una sucesión $v_1, v_2, ..., v_n$ de $n = |V|$ vértices, es

fácil ver en tiempo polinomial que $\{v_i, v_{i+1}\} \in E$ para $i = 1, ..., n-1$ y que $\{v_n, v_1\} \in E$ también.

Cualquier problema en P también está en NP, ya que si un problema está en P podemos resolverlo en tiempo polinomial sin siquiera darnos un certificado. Por tanto $P \subseteq NP$, sin embargo, aún no es posible responder a la pregunta de si son iguales. La importancia de esta última clase es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas, como en el problema a tratar en este trabajo. Dada su importancia, se han hecho muchos esfuerzos para encontrar algoritmos que decidan algún problema de NP en tiempo polinómico. Sin embargo, para algunos problemas de NP (los del conjunto NP-completo) no es posible encontrar un algoritmo mejor que simplemente realizar una búsqueda exhaustiva. Informalmente, se define un problema de **clase NP-completo** o NPC si es un problema NP y es tan "duro" como cualquier problema en NP.

Algunos ejemplos de problemas NPC son, el problema del camino más largo, que, dado un grafo, consiste encontrar un camino simple de longitud máxima. A diferencia del problema del camino más corto, que se puede solucionar en tiempo polinómico, este problema es NP-completo, lo que quiere decir que la solución óptima no se puede encontrar en tiempo polinómico o el problema de encontrar un circuito de Hamilton.

Uno de los conceptos claves al mostrar que un problema es NP-completo es diferenciar entre problemas de decisión y problemas de optimización, muchos de los problemas de interés son **problemas de optimización**, en el que cada solución factible tiene un valor asociado, y deseamos encontrar una solución factible con el mejor valor. Sin embargo, la NP-completitud no se aplica directamente a los problemas de optimización, sino a los **problemas de decisión**, en los que la respuesta es simplemente sí o no, o más formalmente, 1 o 0.

Aunque mostrar que un problema es NPC nos limita al ámbito de los problemas de decisión, existe una relación conveniente entre los problemas de optimización y los problemas de decisión. Por lo general, podemos convertir un problema de optimización dado en un problema de decisión relacionado imponiendo un límite al valor que se va a optimizar. Si se puede proporcionar evidencias de que un problema de decisión es difícil, también se tienen evidencias de que su problema de optimización relacionado es difícil.

La noción anterior de mostrar que un problema no es más difícil ni más fácil que otro se aplica incluso cuando ambos problemas son problemas de decisión. Consideremos un problema de decisión, digamos A, que nos gustaría resolver en tiempo polinómico. Llamamos a la entrada de un problema particular una **instancia** de ese problema. Supongamos ahora que hay un problema de decisión diferente, digamos B, que ya sabemos cómo resolver en tiempo polinomial. Finalmente, supongamos que tenemos un procedimiento que transforma cualquier instancia α de A en alguna instancia β de B con las siguientes características; la transformación toma tiempo polinomial; la respuesta

es la misma. A este procedimiento lo llamamos **algoritmo de reducción en tiempo polinomial**, otro de los conceptos claves, y nos proporciona una manera de resolver el problema A en tiempo polinomial.

Llevando la idea un paso más allá, se muestra cómo se pueden usar reducciones de tiempo polinomial para demostrar que no puede existir ningún algoritmo de tiempo polinomial para un problema particular B. Se supone que se tiene un problema de decisión A para el que ya sabemos que no puede existir ningún algoritmo de tiempo polinomial y que, además, se tiene una reducción de tiempo polinomial que transforma instancias de A en instancias de B. Usando contradicción se muestra que no puede existir ningún algoritmo de tiempo polinomial para B.

Para la NP-completitud, se prueba que el problema B es NP-completo suponiendo que el problema A lo es también. Como la técnica de reducción se basa en tener un problema que ya se sabe que es NPC para probar un problema diferente NPC, es necesario un "primer" problema NP-completo.

La clase de complejidad NP**-hard** o NP-difícil es el conjunto de los problemas de decisión que contiene los problemas H tales que todo problema L en NP puede ser transformado polinomialmente en H. Esta clase puede ser descrita como aquella que contiene a los problemas de decisión que son como mínimo tan difíciles como un problema de NP. Esta afirmación se justifica porque si podemos encontrar un algoritmo A que resuelve uno de los problemas H de NP-hard en tiempo polinómico, entonces es posible construir un algoritmo que trabaje en tiempo polinómico para cualquier problema de NP ejecutando primero la reducción de este problema en H y luego ejecutando el algoritmo A.

En la figura 29, se puede observar un diagrama de Euler de las familias P, NP, NP-completo y NP-hard, dependiendo si consideramos $P \neq NP$ o $P = NP$.

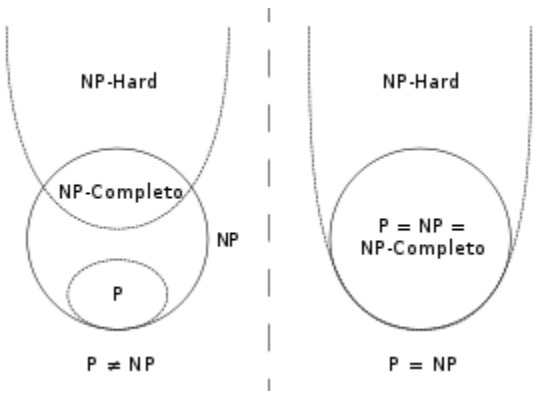


Figura 29: Fuente [19]

4.6. Algoritmos aproximados

(Fuente: [5]) Muchos problemas de importancia práctica son NP-completos, pero son demasiado importantes para abandonarlos simplemente porque la obtención de una solución óptima es intratable. Si un problema es NPC, es poco probable que se encuentre un algoritmo de tiempo polinomial para resolverlo exactamente, pero aún así, puede haber otras soluciones. Hay al menos tres enfoques para sortear la completitud de NP. Primero, si la entrada es pequeña, un algoritmo con tiempo de ejecución exponencial puede ser perfectamente satisfactorio. En segundo lugar, podemos aislar casos especiales importantes que se pueden resolver en tiempo polinomial. En tercer lugar, puede ser posible encontrar soluciones casi óptimas, lo que se denomina **algoritmo de aproximación**.

Dependiendo del problema, una solución óptima puede ser definida como una con el máximo coste posible o otra con el mínimo coste posible, es decir, el problema puede ser un problema de maximización o un problema de minimización. Se dice que un algoritmo tiene una **relación de aproximación** de $p(n)$ si, para cualquier entrada de tamaño n, el coste C de la solución producida por un algoritmo está dentro de un factor de $p(n)$ del costo C^* de una solución óptima, es decir, $max(\frac{C}{C^*}, \frac{C^*}{C}) \leq p(n)$. También se denomina a un algoritmo que logra una relación de aproximación de $p(n)$, un algoritmo de aproximación $p(n)$. La relación de aproximación nunca es inferior a 1.

Para muchos problemas, se han desarrollado algoritmos de aproximación en tiempo polinomial con relaciones de aproximación constantes pequeñas, mientras que para otros problemas, los algoritmos de aproximación en tiempo polinomial más conocidos tienen relaciones de aproximación que crecen en función del tamaño de entrada n. Algunos problemas NP-completos permiten algoritmos de aproximación en tiempo polinomial (ver ejemplo 20) que pueden lograr proporciones de aproximación cada vez más pequeñas utilizando más tiempo de cálculo. Es decir, existe un equilibrio entre el tiempo de cálculo y la calidad de la aproximación.

Ejemplo 20. *Recubrimiento por vértices (The vertex-cover problem).*

Dado un grafo no dirigido $G = (V, E)$, este problema consiste en hallar un subconjunto $V' \in V$ de cardinal mínimo tal que si la arista $(u, v) \in E$, entonces también $u \in V'$ o $v \in V'$ (o ambos). Tal subconjunto es un recubrimiento óptimo. Este problema es la versión de optimización de un problema de decisión NP-completo.

Aunque puede ser difícil encontrar un recubrimiento óptimo en un grafo G, no es demasiado difícil encontrar un subconjunto que sea casi óptimo. El siguiente algoritmo de aproximación toma como entrada un grafo no dirigido G, y devuelve un recubrimiento cuyo tamaño se garantiza que no será más del doble del tamaño del recubrimiento óptimo.

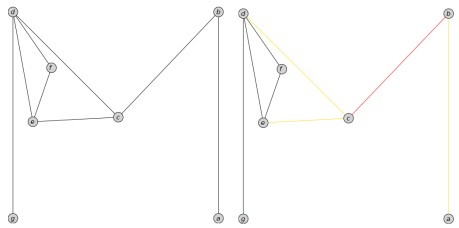
```
procedure AproxVertexCover(G: undirected graph)
C:= empty list
E':= E
while E' ≠ ∅ do
    let (u, v) be an arbitrary edge of E'
    C:= C ∪ {u, v}
    remove from E' every edge incident on either u or v
end while
return C
```

Se inicializa C como un conjunto vacío y E' como una copia de E, el bucle elige repetidamente una arista (u, v) de E', agrega sus extremos u y v a C y elimina todos las aristas en E' que están cubiertas por u o v, es decir, que u o v (o ambos) son sus extremos. El tiempo de ejecución de este algoritmo depende linealmente del número de aristas y vértices, esto es $O(n+m)$, donde n es el número de vétices de G y m el número de aristas. Este procedimiento es un algoritmo de aproximación $p(n) = 2$.

Para ver que este algoritmo devuelve un recubrimiento que es como máximo el doble del tamaño de un recubrimiento óptimo, sea A el conjunto de aristas seleccionadas en el algoritmo. Para cubrir las aristas en A, cualquier recubrimento, en particular, un recubrimiento óptimo C^*, debe incluir al menos un extremo de cada arista en A. No hay dos aristas en A que compartan un extremo, dado que una vez que se selecciona una arista, todas las demás que inciden en sus extremos se eliminan de E'. Por lo tanto, dos aristas en A no están cubiertas por el mismo vértice de C^*, y tenemos por tanto el límite inferior $|C^*| \geq |A|$ en el tamaño de un recubrimiento óptimo.

Cada vez se toma una arista para la cual ninguno de sus extremos está en C, produciendo un límite superior exacto, en el tamaño del recubrimiento devuelto por el algoritmo, $|C| = 2|A|$. Combinando ambos límites, tenemos que $|C| = 2|A| \leq 2|C^*|$.

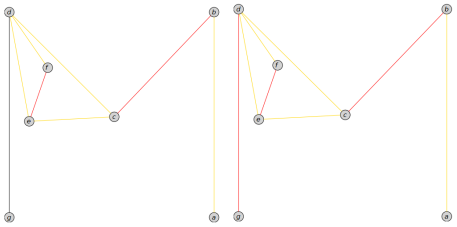
Veamos un ejemplo. Dado $G = (V, E)$, comenzamos tomando una arista arbitraria, sea esta $\{b, c\}$, estos vértices se añaden a C. Las aristas $\{a, b\}$, $\{c, e\}$ y $\{c, d\}$ son eliminadas por tener como extremo un vértice en C. En la figura 30, la arista roja es la arista cuyos vértices son añadidos a C, y las aristas doradas, las eliminadas.



(a) Grafo $G = (V, E)$ (b) Primer paso

Figura 30: Vertex-Cover

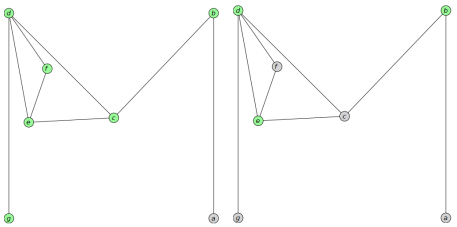
En el siguiente paso, se elige la arista $\{e, f\}$, añadiendo estos vértices a C y se eliminan las aristas $\{c, e\}$ y $\{d, e\}$, por tener como extremo un vértice en C. Finalmente, se añaden los vértices de la arista $\{d, g\}$. Estos pasos se muestran en la figura 31.



(a) Segundo paso (b) Paso final

Figura 31: Vertex-Cover

El conjunto C, recubrimento del algoritmo, contiene 6 vértices. Sin embargo, el recubrimiento óptimo contiene unicamente 3 vértices, como se muestra en la figura 32.



(a) Recubrimiento C (b) Óptimo

Figura 32: Vertex-Cover

5. El problema del viajante

(Fuentes: [2], [20]) El problema del viajante (*TSP-Traveling Salesman Problem*), ya introducido en la sección 2.6.2, consiste en determinar, dada una lista de ciudades y las distancias entre ellas, la ruta más corta que visita cada ciudad una sola vez y regresa a la ciudad de origen.

El TSP se puede modelar como un grafo ponderado no dirigido (definición 24), de modo que las ciudades sean los vértices, las rutas sean las aristas y la distancia sea el peso de estas. Así, más formalmente, el problema del viajante en un grafo ponderado G, consiste en encontrar un circuito de Hamilton (definición 23) en G de coste mínimo. Es un problema de minimización que comienza y termina en un vértice específico después de haber visitado cada vértice exactamente una vez. A menudo, el modelo es un grafo completo, es decir, contiene exactamente una arista entre cada par de vértices distintos. Si no existe un camino entre dos ciudades, se puede agregar una arista lo suficientemente larga para completar el grafo sin afectar el recorrido óptimo.

El problema del viajante es uno de los problemas de optimización más intensamente estudiados, tiene numerosas aplicaciones en logística e industria. Se utiliza como punto de referencia para muchos métodos de optimización. Se trata de un problema NP-hard (sección 4.5) y por tanto no tiene solución computacionalmente simple, sin embargo, existen numerosos algoritmos heurísticos que dan una solución aproximada a este problema de una forma efectiva.

Este trabajo enfoca dicho problema de optimización desde la teoría de grafos, explicada en las secciones 2 y 3. En particular, se va a estudiar el algoritmo de Christofides, cuyo desarrollo se encuentra en la sección 5.2.

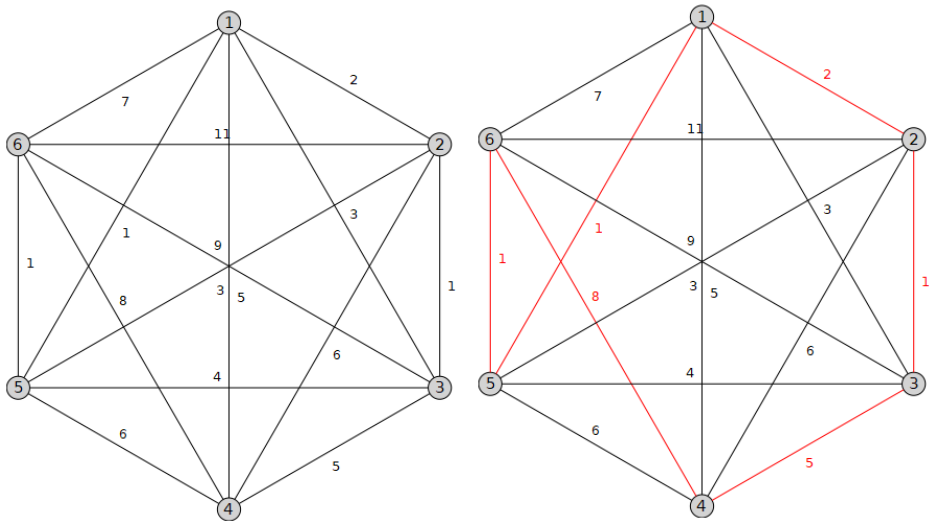
Hay muchas clasificaciones diferentes del TSP, una posible distinción (referencia [6]) que se puede hacer, son las dos variaciones siguientes, el problema del viajante simétrico ($sTSP$), el problema del viajante asimétrico ($aTSP$).

Definición 31. *El $sTPS$ es el problema que consiste en encontrar un circuito de mínimo coste que visite cada ciudad una única vez, a excepción de la primera ciudad que será la primera y la última en ser visitada, donde la distancia entre dos ciudades es la misma independientemente de la dirección tomada.*

En el $aTSP$, sin embargo, la distancia entre dos ciudades cualesquiera puede ser diferente dependiendo de la dirección en que se viaja.

Veamos un primer ejemplo de un TSP simétrico, figura 33.

Ejemplo 21. *Para el grafo ponderado conexo $G = (V, E)$, se tiene que la ruta a seguir por el viajante es el circuito de Hamilton $1, 2, 3, 4, 6, 5, 1$, que pasa a través de todas las ciudades (vértices) y cuyo coste es 18, la suma de los pesos asignados a las rutas (aristas) que sigue el viajante. La ruta de la ciudad 1 a la ciudad 2, $\{1, 2\}$ tiene peso 2, la ruta $\{2, 3\}$, peso 1. Para la ruta $\{3, 4\}$, 5 y para $\{4, 6\}$ se tiene como peso 8, y finalmente, 1 es el peso asignado para las rutas $\{6, 5\}$ y $\{5, 1\}$.*



(a) Grafo G ponderado (b) TSP

Figura 33: Ejemplo TSP simétrico

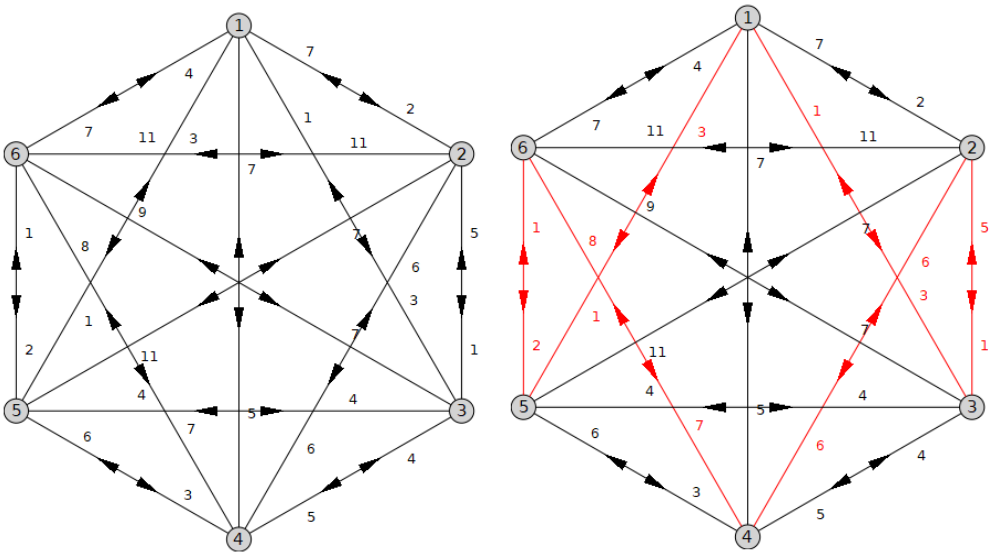
Veamos ahora, un ejemplo de un TSP no simétrico que se ilustra en la figura 34.

Ejemplo 22. *La ruta que sigue el viajante, en este caso, es viajar a través de las ciudades* $1, 5, 6, 4, 2, 3, 1$*, cuyo coste es 17, ya que la arista* $(1, 5)$ *tiene asignado un peso de 1, el mismo para la arista* $(5, 6)$*. La arista* $(6, 4)$ *tiene un peso asignado de 7, y la arista* $(4, 2)$ *de 6. Finalmente, las aristas* $(2, 3)$ *y* $(3, 1)$ *tienen asignado 1 como peso. En la siguiente tabla, se muestran las rutas (las aristas) y el costo (peso) de dichas rutas, la ruta* (i, j) *se refiere a, ir de la ciudad* i *a la ciudad* j*.*

Ruta	*Costo*	*Ruta*	*Costo*	*Ruta*	*Costo*	*Ruta*	*Costo*	*Ruta*	*Costo*
$(1, 2)$	*2*	$(2, 4)$	*6*	$(4, 5)$	*6*	$(5, 1)$	*3*	$(4, 3)$	*4*
$(1, 3)$	*3*	$(2, 5)$	*11*	$(4, 6)$	*8*	$(6, 1)$	*4*	$(5, 3)$	*4*
$(1, 4)$	*5*	$(2, 6)$	*11*	$(5, 6)$	*1*	$(3, 2)$	*5*	$(6, 3)$	*7*
$(1, 5)$	*1*	$(3, 4)$	*5*	$(2, 1)$	*7*	$(4, 2)$	*6*	$(5, 4)$	*3*
$(1, 6)$	*7*	$(3, 5)$	*4*	$(3, 1)$	*1*	$(5, 2)$	*7*	$(6, 4)$	*7*
$(2, 3)$	*1*	$(3, 6)$	*9*	$(4, 1)$	*7*	$(6, 2)$	*11*	$(6, 5)$	*2*

5.1. Historia

Los orígenes del problema del viajante no están claros. Un manual para vendedores ambulantes de 1832 menciona el problema e incluye viajes de ejemplo por Alemania y Suiza, pero no contiene ningún tratamiento matemático. El problema del viajante fue formulado matemáticamente en el siglo XIX por el matemático irlandés William Rowan Hamilton y por el matemático británico Thomas Kirkman. La forma general del TSP parece haber sido estudiada por primera vez durante la década de 1930 en Viena



(a) Grafo G ponderado (b) TSP

Figura 34: Ejemplo TSP no simétrico

y en Harvard, especialmente por Karl Menger, quien define el problema, considera el algoritmo de fuerza bruta y observa la no optimización.

Fue considerado matemáticamente por primera vez en la década de 1930 por Merrill M. Flood , quien buscaba resolver un problema de enrutamiento de autobuses escolares. La primera vez que se usó "problema del viajante de comercio" fue en el informe de RAND Corporation de 1949 de Julia Robinson.

En las décadas de 1950 y 1960, el problema se hizo cada vez más popular después de que RAND Corporation ofreciera premios por los pasos para resolver el problema. George Dantzig , Delbert Ray Fulkerson y Selmer M. Johnson, resolvieron una instancia con 49 ciudades de manera óptima mediante la construcción de un recorrido y demostrando que ningún otro podría ser más corto. Sin embargo, especularon que, dada una solución casi óptima, podemos encontrar la optimización o probarla agregando una pequeña cantidad de desigualdades adicionales. Estas ideas fueron indispensables para crear posteriormente métodos de solución exactos.

En 1959, Jillian Beardwood , JH Halton y John Hammersley publicaron un artículo titulado "El camino más corto a través de muchos puntos". El teorema de Beardwood-Halton-Hammersley proporciona una solución práctica al problema, derivaron una fórmula asintótica para determinar la longitud de la ruta más corta para un vendedor que comienza en una oficina y visita un número fijo de lugares antes de regresar al inicio.

En la década de 1960 se creó un nuevo enfoque que, en lugar de buscar soluciones óptimas, produciría una solución cuya longitud esté limitada por un múltiplo de la longitud óptima y, al hacerlo, crearía límites inferiores para el problema. Un método para

hacer esto fue crear un árbol de expansión mínima del grafo y luego duplicar todas sus aristas, lo que produce que la longitud de un recorrido óptimo es como máximo el doble del peso de un árbol de expansión mínimo.

En 1976, Christofides y Serdyukov, de forma independiente el uno del otro, hicieron un gran avance en esta dirección. El algoritmo Christofides-Serdyukov produce una solución que, en el peor de los casos, es como mucho $1,5$ veces más larga que la solución óptima. Como el algoritmo era simple y rápido, muchos esperaban que daría paso a un método de solución casi óptimo. Sin embargo, esta esperanza de mejora no se materializó hasta el siglo XXI.

Richard M. Karp mostró en 1972 que el problema del circuito hamiltoniano era NP-completo, lo que implica la dureza NP del TSP y por tanto una explicación para la aparente dificultad computacional. Se lograron grandes avances a finales de la década de 1970 y 1980, cuando se logró resolver instancias de forma exacta con hasta 2392 ciudades.

Gerhard Reinelt publicó el TSPLIB en 1991, una colección de instancias de referencia de diversa dificultad, que ha sido utilizada por muchos grupos de investigación para comparar resultados. En 2006, se calculó un recorrido óptimo a través de una instancia de 85900 ciudades, actualmente la instancia TSPLIB resuelta más grande. Para muchos otros casos con millones de ciudades, se pueden encontrar soluciones que garantizan estar dentro del 2-3 % de un recorrido óptimo.

5.2. Algoritmo de Christofides

El algoritmo de Christofides o algoritmo de Christofides-Serdyukov es un algoritmo aproximado para encontrar soluciones aproximadas al problema del viajante, en los casos en que las distancias forman un espacio métrico (son simétricas y obedecen a la desigualdad triangular). Es un algoritmo de aproximación polinomial que garantiza que sus soluciones estarán dentro de un factor de $3/2$ de la longitud óptima de la solución, y lleva el nombre de Nicos Christofides y Anatoliy I. Serdyukov, quienes lo descubrieron de forma independiente en 1976. Este algoritmo es considerado como uno de los mejores para el problema del viajante en una métrica arbitrarea.

Sin embargo, para la métrica euclidea, algunas instancias con decenas de miles de ciudades pueden resolverse completamente e incluso problemas con millones de ciudades pueden aproximarse dentro de una pequeña fracción del 1 %, como el algoritmo heurístico LKH. También algoritmos de ramificación y acotación. Algunos de estos algoritmos están explicados en la sección 5.4.

Dado un grafo completo $G = (V, E)$, donde $|V| = n$ y $|E| = m = n(n-1)/2$, se puede describir el algoritmo en pseudocódigo de la siguiente manera.

1: Crear un árbol de expansión mínima T de G.
2: Sea O el conjunto de vértices de grado impar en T. Por el lema del apretón de manos, O tiene un número par de vértices.
3: Encontrar un emparejamiento perfecto de peso mínimo M en el subgrafo inducido de G dado por los vértices de O.
4: Combinar las aristas de M y T para formar un multigrafo conexo H en el que cada vértice tiene un grado par.
5: Formar un circuito de Euler en H.
6: Convertir el circuito encontrado en el paso anterior en un circuito de Hamilton omitiendo vértices repetidos.

Construir un árbol de expansión mínima $T = (V, E_T)$ tiene complejidad $O(mlog\, n)$, ya que se utiliza alguno de los algoritmos explicados en la sección 3.3. Encontrar un emparejamiento perfecto de peso mínimo para el conjunto de todos los vértices de grado impar O de T, con $|O| = n_1$, tiene complejidad $O(n_1^3)$ ([7], [9]) y formar el circuito de Euler en $H = (V, F)$ y convertirlo en un circuito de Hamilton omitiendo los vértices repetidos, $O(n)$ de tiempo. Por tanto, en el peor de los casos, ejecutar este algoritmo tiene complejidad $O(n^3)$.

Veamos que efectivamente es un algoritmo de aproximación que garantiza que sus soluciones están dentro de un factor de 3/2 de la longitud óptima de la solución. Consideremos el TSP de n ciudades definido en el grafo completo $G = (V, E)$ donde V es el conjunto de vértices y E es el conjunto de aristas. Sea $C = [cij]$ la matriz de costos (la matriz que almacena en la entrada c_{ij} el costo de ir de la ciudad i a la ciudad j) que satisface la desigualdad triangular.

Sea $T = (V, E_T)$ un árbol de expansión mínima del grafo G, y sea $C(T)$ el costo de T. Sea $O = \{v_i \mid \deg_T(v_i) \text{ impar}\}$ donde $\deg_T(v_i)$ es el grado del vértice $v_i \in V$ con respecto a T, es decir, respecto a las atistas que están en el árbol de expansión mínima, y como ya sabemos (teorema 2), $|O|$ es par (O contiene todas las hojas de T y quizás también algunos vértices internos). Consideramos ahora el subgrafo inducido (definición 12) dado por los vértices de O, como este conjunto de vértices de grado impar tiene un número par de vértices, existe siempre un emparejamiento perfecto en el subgrafo inducido (el número de aristas en un emparejamiento perfecto, para $G = (V, E)$ debe ser $|V|/2$, por tanto existe un emparejamiento perfecto si el número de vértices es par). Sea $M = (O, E_M)$ un emparejamiento perfecto de peso mínimo en el subgrafo inducido y $C(M)$ su coste. Podemos ahora enunciar el siguiente teorema.

Teorema 13. *Un circuito de Hamilton ϕ de G, solución del algoritmo, puede ser encontrado con coste $C(\phi) \leq C(T) + C(M) < \frac{3}{2}C(\Phi)$ donde $C(\Phi)$ es el valor óptimo del circuito Φ, la solución óptima.*

En la demostración de este teorema, se utiliza el siguiente lema.

Lema 2. *En un TSP con n ciudades, donde n es par, se tiene que $C(M) \leq \frac{1}{2}C(\Phi)$ donde M es el emparejamiento perfecto de coste mínimo del grafo G que define el TSP y Φ es el circuito óptimo TSP, es decir, la solución óptima.*

Demostración. Consideramos $\Phi = (v_{i_1}, v_{i_2}, ..., v_{i_n})$. Partiendo del vértice v_{i_1} y recorriendo el circuito Φ, se asignan las aristas recorridas de manera alterna a dos conjuntos M_1 y M_2. Empezando por M_1, por ejemplo, $M_1 = \{(v_{i_1}, v_{i_2}), (v_{i_3}, v_{i_4}), ..., (v_{i_{n-1}}, v_{i_n})\}$ y $M_2 = \{(v_{i_2}, v_{i_3}), (v_{i_4}, v_{i_5}), ..., (v_{i_n}, v_{i_1})\}$. M_1 y M_2 son emparejamientos en G y se tiene que $C(M_1) + C(M_2) = C(\Phi)$. Dado que se definen arbitrariamente, se puede suponer $C(M_1) \leq C(M_2)$, sin pérdida de generalidad, por lo que,

$$C(M) \leq C(M_1) \leq \frac{1}{2}C(\Phi)$$

□

Veamos ahora la demostración del teorema.

Demostración. Para un grafo G, se tiene que,

$$C(T) \leq C(\Phi_p) < C(\Phi) \tag{1}$$

donde Φ_p es el camino de Hamilton más corto de G. Combinando las aristas de M y T, se forma un multigrafo conexo H en el que cada vértice tiene un grado par ($\deg_T(v_i)$ impar y $\deg_M(v_i) = 1$).

Ahora, se puede formar un circuito de Euler en H (teorema 5), sea este $\Phi_H = (v_{i_1}, v_{i_2}, ..., v_{i_k})$, Φ_H pasa a traves de todas las aristas de H y visita, al menos una vez, todos sus vértices. Sea $C(\Phi_H)$ el coste de este circuito, es decir,

$$C(\Phi_H) = C(T) + C(M) \tag{2}$$

Si Φ_O es la solución TSP al problema definido por el subgrafo inducido, entonces, por el lema 2, $C(M) \leq \frac{1}{2}C(\Phi_O)$ y como $C(\Phi_O) \leq C(\Phi)$, inmediatamente obtenemos que,

$$C(M) \leq \frac{1}{2}C(\Phi) \tag{3}$$

De las expresiones 1, 2 y 3 se sigue que,

$$C(\Phi_H) < \frac{3}{2}C(\Phi)$$

Consideremos el recorrido de Φ_H desde v_{i_1} hasta el punto en que se alcanza un vértice v_{i_r} que ha sido visitado previamente. Sea v_{i_s} el primer vértice que sigue a v_{i_r} en la sucesión de Φ_H que no ha sido visitado previamente y consideremos el circuito $\phi_1 = (v_{i_1}, ..., v_{i_{r-1}}, v_{i_s}, ...v_{i_k})$ derivado de Φ_H al remplazar el camino $P_{rs} = (v_{i_{r-1}}, v_{i_r}, ..., v_{i_{s-1}}, v_{i_s})$ por la arista $(v_{i_{r-1}}, v_{i_s})$.

Debido a la condición de triangularidad, tenemos $c_{i_{r-1}i_s} \leq \sum_{(v_i,v_j)\in P_{rs}} c_{ij}$ donde P_{rs} también se usa como un conjunto desordenado de aristas en el camino P_{rs}. Así, $C(\Phi_H) \geq C(\phi_1)$. De la misma manera, a partir de un recorrido de ϕ_1 se puede producir un circuito ϕ_2 con un camino de ϕ_1 reemplazado por una arista y $C(\phi_1) \geq C(\phi_2)$. Eventualmente, un circuito de Hamilton ϕ de G resultará cumpliendo que,

$$C(\phi) \leq ... \leq C(\phi_1) \leq C(\Phi_H) < \frac{3}{2}C(\Phi)$$

□

5.3. Ejemplo práctico del algoritmo de Christofides

Veamos un ejemplo para ilustrar el algoritmo de Christofides. Se han elegido 12 ciudades españolas y las distancias respectivas entre estas ciudades. Queremos una ruta que comience y termine en Madrid y visite todas las ciudades una vez.

1	Madrid	2	Barcelona	3	Bilbao	4	Sevilla
5	Valencia	6	Zaragoza	7	Málaga	8	Valladolid
9	A Coruña	10	Oviedo	11	Ciudad Real	12	Badajoz

Un resumen de las distancias se muestra en la siguiente tabla, donde por ejemplo, la ruta $\{1, 2\}$ indica que la distancia entre Madrid y Barcelona es 621, o la ruta $\{8, 10\}$ donde 252 es la distancia entre Valladolid y Oviedo.

Ruta	Distancia	Ruta	Distancia	Ruta	Distancia	Ruta	Distancia
$\{1, 2\}$	621	$\{2, 8\}$	663	$\{4, 7\}$	219	$\{6, 10\}$	604
$\{1, 3\}$	395	$\{2, 9\}$	1118	$\{4, 8\}$	589	$\{6, 11\}$	515
$\{1, 4\}$	538	$\{2, 10\}$	902	$\{4, 9\}$	947	$\{6, 12\}$	726
$\{1, 5\}$	352	$\{2, 11\}$	811	$\{4, 10\}$	789	$\{7, 8\}$	737
$\{1, 6\}$	325	$\{2, 12\}$	1022	$\{4, 11\}$	339	$\{7, 9\}$	1153
$\{1, 7\}$	544	$\{3, 4\}$	993	$\{4, 12\}$	217	$\{7, 10\}$	995
$\{1, 8\}$	193	$\{3, 5\}$	633	$\{5, 6\}$	326	$\{7, 11\}$	388
$\{1, 9\}$	609	$\{3, 6\}$	324	$\{5, 7\}$	648	$\{7, 12\}$	436
$\{1, 10\}$	451	$\{3, 7\}$	939	$\{5, 8\}$	545	$\{8, 9\}$	455
$\{1, 11\}$	190	$\{3, 8\}$	280	$\{5, 9\}$	961	$\{8, 10\}$	252
$\{1, 12\}$	401	$\{3, 9\}$	644	$\{5, 10\}$	803	$\{8, 11\}$	377
$\{2, 3\}$	620	$\{3, 10\}$	304	$\{5, 11\}$	398	$\{8, 12\}$	414
$\{2, 4\}$	1046	$\{3, 11\}$	585	$\{5, 12\}$	716	$\{9, 10\}$	340
$\{2, 5\}$	349	$\{3, 12\}$	694	$\{6, 7\}$	869	$\{9, 11\}$	799
$\{2, 6\}$	296	$\{4, 5\}$	697	$\{6, 8\}$	367	$\{9, 12\}$	772
$\{2, 7\}$	997	$\{4, 6\}$	863	$\{6, 9\}$	833	$\{10, 11\}$	641
$\{10, 12\}$	614	$\{11, 12\}$	318				

El grafo que representa las ciudades y las distancias es un grafo completo, $G = K_{12}$ que puede verse representado en la figura 37 junto con la solución dada por un algoritmo de ramificación y acotación (sección 5.4.1), donde los vértices representan las ciudades numeradas del 1 al 12 y las aristas ponderadas las distancias. El primer paso del algoritmo consiste en crear un árbol de expansión mínima T de G, representado en la figura 35.

El conjunto O de vértices de grado impar de T es $O = \{2, 5, 6, 7, 8, 9\}$, un número par de vértices de grado impar. Formamos el subgrafo inducido de K_{12} dado por los vértices de O cuyo emparejamiento perfecto de peso mínimo M es el formado por las rutas $\{2, 6\}$, $\{5, 7\}$ y $\{8, 9\}$. Ahora, formamos el multigrafo H combinando las aristas de M y de T, donde todos los vértices en H tienen grado par. Ambas representaciones se muestran el la figura 36.

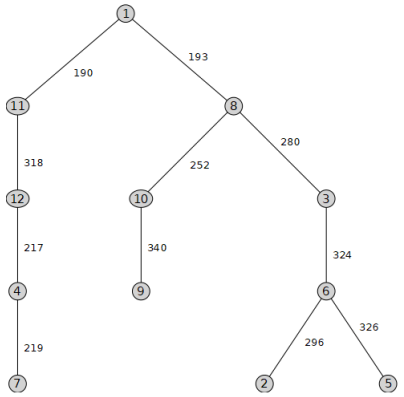
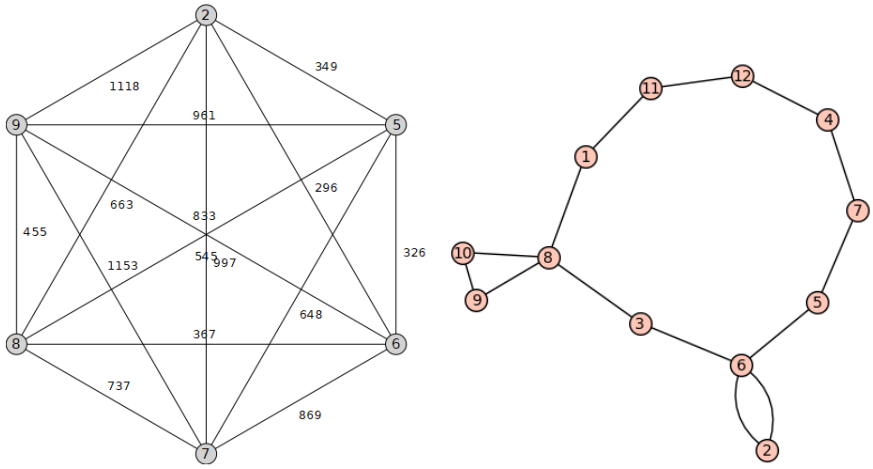


Figura 35: Árbol de expansión mínima T de G



(a) Subgrafo inducido (b) Multigrafo H

Figura 36: Representaciones de los grafos

Formamos un circuito de Euler en H, este circuito es como sigue,

$$1, 8, 10, 9, 8, 3, 6, 2, 6, 5, 7, 4, 12, 11, 1.$$

Eliminando los vértices repetidos, obtenemos el circuito de Hamilton buscado,

$$1, 8, 10, 9, 3, 6, 2, 5, 7, 4, 12, 11, 1.$$

La distancia total que habría que recorrer para pasar una vez por cada ciudad y volver a la ciudad de origen, es 3990. La ruta comienza en Madrid, y consiste en viajar a Valladolid, Oviedo, A Coruña, Bilbao, Zaragoza, Barcelona, Valencia, Málaga, Sevilla, Badajoz, Ciudad Real, y volver de nuevo a Madrid.

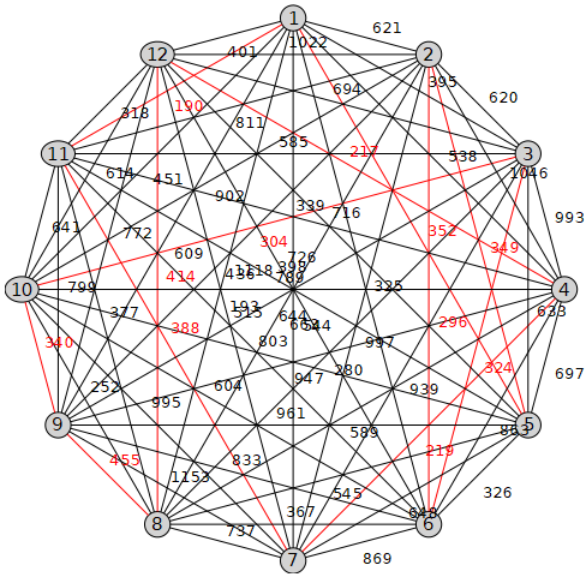


Figura 37: TSP

Sin embargo, el TSP para el grafo G usando un algoritmo de ramificación y acotación, da la ruta siguiente, ilustrada en la figura 37 cuya distancia es 3848.

$$1, 11, 7, 4, 12, 8, 9, 10, 3, 6, 2, 5, 1.$$

Procediento mediante el algoritmo de fuerza bruta, sección 5.4.1, cuyo código para la ejecución ha sido obtenido en el enlace [12], se obtiene la ruta de 3848 kilometros, el tiempo de ejecución de este algoritmo para un problema con 12 vértices ha sido de 72772 segundos, es decir, superior a 20 horas (ejecución con un ordenador de 8 GB RAM, 1000 GB disco duro, 2.5 GHz de velocidad del procesador Intel Core i5). La ruta es la misma que la anterior, ya que se puede recorrer en ambas direcciones.

$$1, 5, 2, 6, 3, 10, 9, 8, 12, 4, 7, 11, 1.$$

Se obtiene la misma ruta, evidentemente, para los dos algoritmos exactos que se han utilizado en este ejemplo, el algoritmo de ramificación y acotación y el algoritmo de fuerza bruta. Sin embargo, el tiempo de ejecución para el primero es instantáneo (para este caso con 12 vértices) y el del segundo supera las 20 horas.

5.4. Métodos de solución del TSP

(Fuentes: [4], [8], [10], [14], [15], [20]) Existen diferentes formas de tratar este problema, una de ellas es mediante algoritmos exactos, utilizados cuando las dimensiones del problema son relativamente pequeñas. Otra forma es mediante algoritmos heurísticos y aproximados, que proporcionan rápidamente buenas soluciones, aunque no se puede garantizar siempre que sean óptimas.

5.4.1. Algoritmos exactos

Los algoritmos exactos son aquellos que buscan obtener la solución óptima independientemente del coste computacional. Estos algoritmos obtienen la solución óptima del problema, lo que implica un tiempo de complejidad elevado.

La solución más directa de encontrar la solución exacta consiste en intentar todas las permutaciones y ver cuál de estas es la menor, lo que se suele denominar búsqueda por fuerza bruta. Esta posibilidad se convierta en impracticable a medida que aumenta el número de ciudades, ya que el tiempo de ejecución es del orden de $O(n!)$, siendo n el número de ciudades a visitar (con tan solo 20 ciudades, el tiempo de complejidad para este método es del orden de $2 \cdot 10^{18}$).

Un pseudocódigo para el **algoritmo de fuerza bruta**, [20], es el siguiente.

1: Se genera un primer circuito de Hamilton τ_0 desde una ciudad fijada. El circuito óptimo en este momento es $\tau = \tau_0$.
2: Se genera el siguiente circuito de Hamilton τ_1, si su coste es menor que τ, entonces el circuito óptimo pasa a ser $\tau = \tau_1$. Se genera de nuevo otro circuito de Hamilton τ_2, si $c(\tau_2) < c(\tau)$ entonces $\tau = \tau_2$ ... Se repite este proceso hasta que se hayan generado todos los posibles circuitos.
3: τ es el circuito óptimo para el TSP.

A continución, se muestra una gráfica para ilustrar que, efectivamente, el tiempo de ejecución sigue una función factorial al aumentar los vértices.

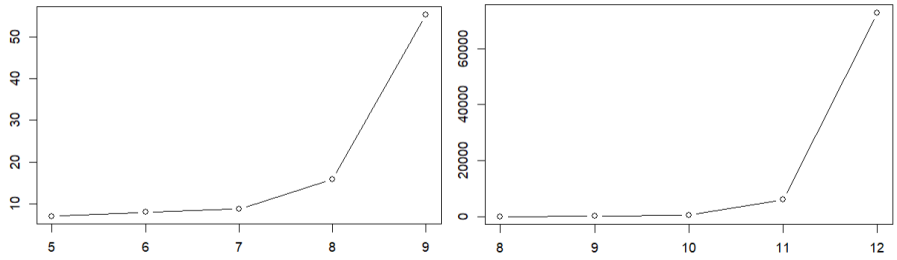


Figura 38: Gráfica de tiempos del algoritmo de fuerza bruta [12]

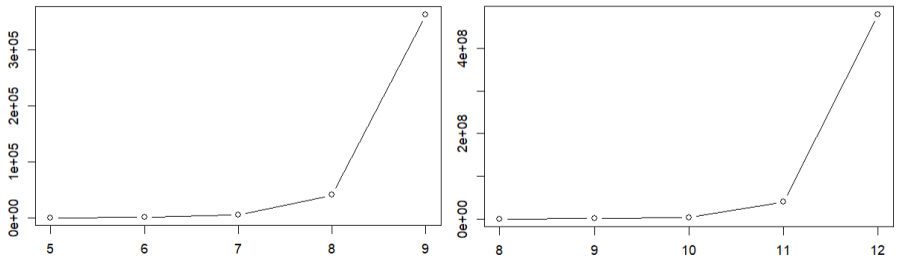


Figura 39: Gráfica de la función factorial

La **técnica branch and bound** (ramificación y acotación, [10]) se suele interpretar como un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la actual. La característica principal de este método es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para "podar" esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima. La complejidad de este tipo de algoritmos aumenta exponencialmente.

En la actualidad **Concorde**, [4], es uno de los algoritmos más destacados. Consiste en uno de los desarrollos más recientes del método de ramificación y acotación sobre árboles de búsqueda. Este algoritmo fue utilizado para obtener la solución óptima a instancias del problema de tamaño record hasta la fecha.

5.4.2. Algoritmos heurísticos o aproximados

Un método heurístico o aproximado, como se ha visto en la sección 4.6, es un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución. En contraposición a los métodos exactos que proporcionan una solución óptima del problema, los métodos heurísticos se limitan a proporcionar una buena solución no necesariamente óptima.

Uno de los heurísticos más conocidos es el **algoritmo del vecino más cercano** (nearest neighbor, [10]), ya que es el más sencillo e intuitivo para resolver el problema TSP. Este algoritmo trata de construir un circuito de Hamilton de bajo coste basándose en el vértice más próximo a un vértice dado.

Este procedimiento tiene un tiempo de ejecución del orden de $O(n^2)$. Dicho algoritmo comienza muy bien, seleccionando aristas de bajo coste en el grafo. Sin embargo, al final del proceso, probablemente quedarán vértices cuya conexión obligará a introducir aristas de coste elevado, ya que, en una iteración se escoge la mejor opción disponible sin tener en cuenta que esto puede obligar a realizar malas elecciones en iteraciones posteriores.

Como norma general, si los últimos pasos del recorrido son comparables en longitud al de los primeros pasos, el recorrido es razonable; si estos son mucho mayores, entonces es probable que existan caminos mucho mejores. Un pseudocódigo para el algoritmo del vecino más cercano es el siguiente.

1: Elección de un vértice arbitrario v en el grafo.
2: Selección de la arista de menor peso cuyos extremos sean el vértice actual v y un vértice no visitado v_i.
3: Convertir el vértice v_i en v y marcarlo como visitado.
4: Si todos los vértices estuvieran visitados, finaliza el algoritmo y la salida es la sucesión de vértices visitados. Si no, retomar de nuevo el paso 2.

Los algoritmos de inserción ([15], [8]) agregan nuevos puntos entre los puntos existentes en un circuito a medida que crece.

Una implementación de la **inserción más cercana** (nearest insertion) comienza con dos ciudades, formando un circuito entre estas. A continuación, selecciona la ciudad fuera del circuito más cercana a cualquiera de las ciudades que ya están en él, y la coloca entre las dos ciudades que harían que el circuito resultante fuera lo más corto posible. Se detiene cuando no quedan más inserciones. El tiempo de ejecución es del orden de $O(n^2)$.

A diferencia de la anterior, la técnica de la **inserción más lejana** (farthest insertion) comienza con una ciudad y la une con la ciudad que está más alejada de ella. A continuación, busca repetidamente la ciudad fuera del circuito que está más alejada de cualquier ciudad del circuito y la ubica entre las dos ciudades que harían que el circuito resultante fuera lo más corto posible. La complejidad computacional es $O(n^2)$.

La **inserción más barata** (cheapest insertion) también comienza con dos ciudades. Luego encuentra la ciudad que no está en el circuito y que, cuando se coloca entre dos ciudades, da como resultado el circuito más corto posible. Inserta la ciudad entre estas dos y repite el proceso hasta que no quedan más inserciones. El tiempo de ejecución de este método es $O(n^2 log(n))$.

El **algoritmo de Lin-Kernighan**, [14], se publicó en 1973 y sigue siendo uno de las más populares. Se utiliza como una subrutina de la mayoría de las heurísticas rápidas actuales. También es conocido como el algoritmo de búsqueda variable, ya que permite ejecutar varias tranformaciones consecutivas a la hora de mejorar la solución actual. Veamos ahora un par de conceptos que ayudarán a la hora de entenderlo mejor.

Definición 32. *Supongamos tener un circuito de Hamilton con dos aristas* $\{i, j\}$ *y* $\{k, l\}$ *incidentes en cuatro vértices distintos y tales que el recorrido pasa por estos vértices en el orden* i, j, k, l*. Entonces, el 2-intercambio asociado a las aristas anteriores consiste en reemplazarlas por las aristas* $\{i, k\}$ *y* $\{j, l\}$*.*

Hay que tener en cuenta que un 2-intercambio en un circuito de Hamilton siempre dará lugar a uno nuevo como se muestra en la figura 40, donde se produce un 2-intercambio entre las aristas $\{2, 4\}$ y $\{3, 5\}$ para obtener las aristas $\{2, 3\}$ y $\{4, 5\}$ (también cambia por tanto la arista $\{4, 3\}$ a $\{3, 4\}$) para obtener un nuevo circuito de Hamilton.

La diferencia de coste entre un circuito original y el nuevo se corresponderá con la diferencia de costes entre las aristas intercambiadas. Existe también un algoritmo llamado **2-opt** que considera todos los posibles intercambios de dos aristas, intercambiando dos de ellas cuando se obtiene un circuito mejorado. El método 2-opt tiene complejidad $O(n^2)$ por cada iteración.

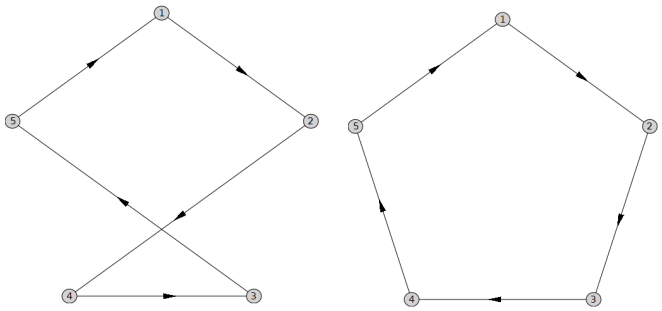


Figura 40: 2-intercambio

La definición anterior de 2-intercambio se puede modificar convenientemente para definir para definir k-intercambios. Los algoritmos (heurísticos) de búsqueda local basados en k-intecambios se conocen como k-opt. En la práctica casi siempre se trabaja con 2-intercambios y 3-intercambios ya que está demostrado que para valores más altos de k el tiempo extra de computación no compensa en relación con la mejora de los resultados.

La idea del algoritmo de Lin-Kernighan es sencilla, es posible que un algoritmo como 2-opt se pare en un máximo local τ porque cualquier 2-intercambio incrementa el coste del circuito. Sin embargo, es posible que si se realiza el menos malo de los 2-intercambios para pasar de τ a τ' y después se busque el mejor 2-intercambio desde τ' llegásemos a un circuito τ'' de coste más bajo que τ. Es decir, podríamos tener que,

$$c(\tau'') < c(\tau) < c(\tau')$$

El algoritmo de Lin-Kernighan no es más que una modificación de los algoritmos 2-opt y 3-opt que permita también una pequeña búsqueda en vecindarios de mayor profundidad (la aplicación sucesiva de 2-intercambios será equivalente a algún k-intercambio para k suficientemente grande). Este algoritmo permite hacer cambios inicialmente costosos pero que a medio plazo resultan beneficiosos. El pseudocódigo es el siguiente,

1: Se elige un circuito de Hamilton τ.
2: Se realiza el mejor 2-intercambio desde τ (aunque esto incremente la longitud del circuito) obteniendo el circuito τ_1. A continución, se realiza el mejor 2-intercambio desde τ_1 que no involucre a ninguna de las aristas añadidas en el paso anterior obteniendo el circuito τ_2... Se termina este paso cuando no haya más 2-intecambios posibles porque ya no queda ninguna arista del circuito original τ. Se define como τ' el circuito de coste más bajo de los circuitos τ_1, τ_2, ...
3: Si $c(\tau') < c(\tau)$, se reemplaza τ por τ' y se repite el paso anterior. Si $c(\tau') \geq c(\tau)$, el algoritmo finaliza ya que no se ha conseguido una mejora.

En el segundo paso del algoritmo, se exige que una arista que fue añadida en algún momento de este paso pueda ser cambiada de nuevo, esto es debido a que una de las motivaciones de este algoritmo es intentar evitar que este termine en óptimos locales.

Ya que si no, se podría dar el caso de que el mejor 2-intercambio que se hace primero sea también el mejor pero a la inversa, en la siguiente realización del 2-intercambio. Con lo cual estaríamos estancados en un mismo circuito.

El algoritmo de Lin-Kernigan es uno de los mejores algoritmos para la resolución del TSP y es ampliamente usado en la práctica. El algoritmo evita caer en un máximo locar pero no se tiene la seguridad completa de que no lo haga.

Ejemplo 23. *Para el ejemplo de la sección 5.3, se puede realizar un 2-intercambio que disminuye los kilometros a recorrer por el viajante.*

Dado el circuito de Hamilton resultante, tomamos las dos aristas incidentes en cuatro de los vértices por los cuales el circuito pasa en el orden 8, 10, 9, 3, *es decir, tomamos* $\{8, 10\}$ *y* $\{9, 3\}$. *El costo en la subruta anterior es 1236, realizamos el 2-intercambio para obtener* $\{8, 9\}$, $\{10, 3\}$ *y el coste en la subruta* 8, 9, 10, 3 *es ahora 1099. El nuevo circuito es* 1, 8, 9, 10, 3, 6, 2, 5, 7, 4, 12, 11, 1, *y la distancia asociada a este pasa a ser 3853, menor en 137 km al circuito producido por el algoritmo de Christofides.*

5.5. Ejemplo final del problema del viajante

La National Basketball Association, más conocida simplemente por sus siglas NBA, es una liga privada de baloncesto profesional que se disputa en Estados Unidos desde 1949. Está conformada por 30 equipos o franquicias divididos en dos conferencias, la conferencia Este, y la conferencia Oeste, las cuales a su vez se encuentran divididas en tres divisiones de cinco equipos cada una. En la temporada regular cada equipo disputa 82 partidos, divididos en partes iguales entre encuentros de local y visitante. El calendario no es el mismo para todos. Los equipos se enfrentan con los oponentes de su propia división en cuatro ocasiones, ante los de las otras dos divisiones de su conferencia, entre tres o cuatro veces; y contra los de la otra conferencia, dos veces al año.

Las divisiones se conforman por los siguientes equipos. Para la conferencia Oeste, tenemos la división Noroeste esta formada por Denver Nuggets (Denver, Colorado), Minnesota Timberwolves (Mineápolis, Minnesota), Oklahoma City Thunder (Oklahoma City, Oklahoma), Portland Trail Blazers (Portland, Oregón) y Utah Jazz (Salt Lake City, Utah). La división Suroeste está formada por, Dallas Mavericks (Dallas, Texas), Houston Rockets, (Houston, Texas), Memphis Grizzlies (Memphis, Tennessee), New Orleans Pelicans, (Nueva Orleans, Luisiana) y San Antonio Spurs (San Antonio, Texas). Por último, la división del Pacífico, integrada por Golden State Warriors (San Francisco, California), Loa Angeles Clippers y Los Angeles Lakers (Los Ángeles, California), Phoenix Suns (Phoenix, Arizona) y Sacramento Kings (Sacramento, California).

Para la conferencia Este, tenemos la división del Atlántico, formada por Boston Celtics (Boston, Massachusetts), Brooklyn Nets y New York Knicks (Nueva York, NY), Philadelphia 76ers (Filadelfia, Pensilvania) y Toronto Raptors (Toronto, Ontario, Canadá). La división Central, integrada por Chicago Bulls (Chicago, Illinois), Cleveland

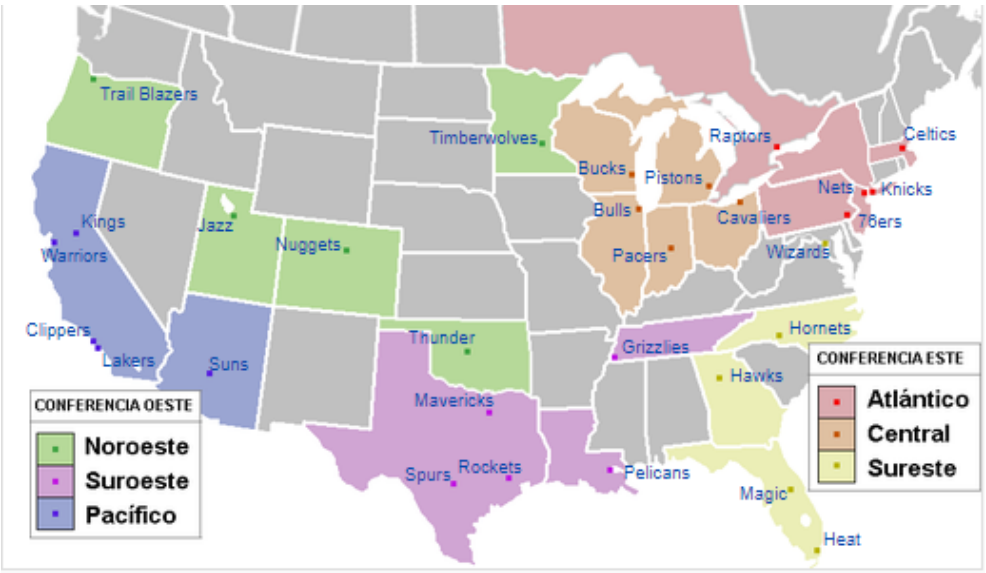


Figura 41: Fuente [21], mapa NBA

Cavaliers (Cleveland, Ohio), Detroit Pistons (Detroit, Michigan), Indiana Pacers (Indeanápolis, Indiana) y Milwaukee Bucks (Milwaukee, Wisconsin). Finalmente, la última división, la división del Sureste, formada por Atlanta Hawks (Atlanta, Georgia), Charlotte Hornets (Charlotte, Carolina del Norte), Miami Heat (Miami, Florida), Orlando Magic (Orlando, Florida) y Washington Wizards (Washington D.C, Distrito de Columbia). Vemos en la figura 41 un resumen de lo anterior. En el enlace [17], se han obtenido las respectivas distancias entre los diferentes pabellones de la NBA.

Supongamos que un aficionado a la NBA desea visitar cada uno de sus pabellones una vez. Dicho aficionado desea recorrer el mínimo número de millas para viajar entre las ciudades y ver a todos los equipos de la NBA competir una vez en su pabellón principal. Supongamos que este aficionado desea empezar visitando el Madison Square Garden, pabellón de los New York Knicks.

Veamos como ofrecerle a este aficionado del baloncesto estadounidense la ruta deseada, con los diferentes algoritmos explicados anteriormente, desarrollando por pasos el algoritmo principal del trabajo, el algoritmo de Christofides.

Enumeramos los equipos por orden alfabético, teniendo en cuenta que LA Lakers y LA Clippers juegan en el mismo pabellón, por lo tanto no supone un desplazamiento, y sin embargo Brooklyn y Knicks, equipos de Nueva York, tienen un pabellón diferente. Es decir, tenemos por tanto 29 enumeraciones.

1	Atlanta	2	Boston	3	Brooklyn	4	Charlotte
5	Chicago	6	Cleveland	7	Dallas	8	Denver
9	Detroit	10	Golden State	11	Houston	12	Indiana
13	LA	14	Memphis	15	Miami	16	Milwaukee
17	Minnesota	18	New Orleans	19	NY Knicks	20	Oklahoma
21	Orlando	22	Philadelphia	23	Phoenix	24	Portland
25	Sacramento	26	San Antonio	27	Toronto	28	Utah
29	Washington						

El grafo ponderado que representa el problema es $G = K_{29}$. El primer paso del algoritmo de Christofides, consiste en crear un árbol de expansión mínima T de G, figura 42.

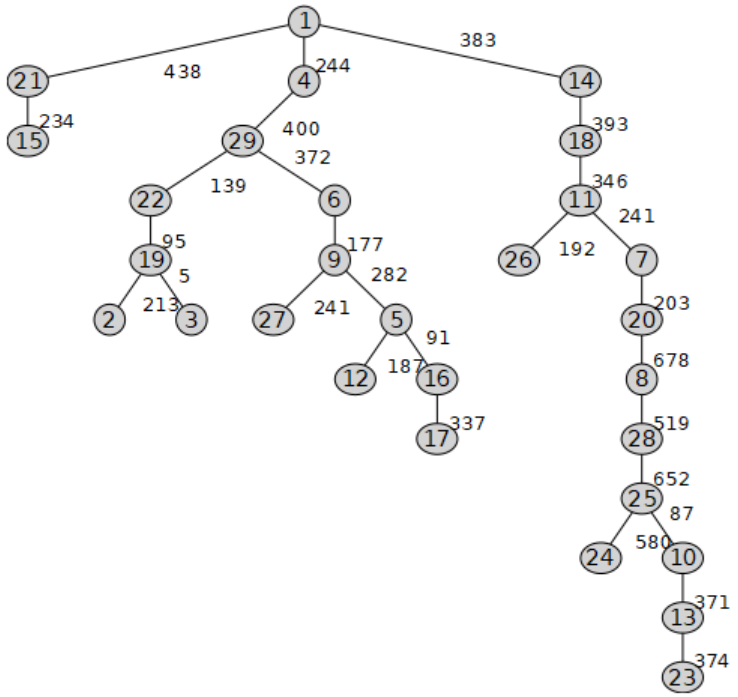


Figura 42: Árbol de expansión mínimo

El conjunto O de vértices de grado impar de T es $O = \{1, 2, 3, 5, 9, 11, 12, 15, 17, 19, 23, 24, 25, 26, 27, 29\}$, un número par de vértices de grado impar. Formamos el subgrafo inducido de G, es decir, el grafo completo K_{16} dado por los vértices de O cuyo emparejamiento perfecto de peso mínimo M es el formado por las rutas $\{1, 15\}$, $\{2, 19\}$, $\{3, 29\}$, $\{5, 12\}$, $\{9, 27\}$, $\{11, 26\}$, $\{17, 23\}$ y $\{24, 25\}$.

Ahora, formamos el multigrafo H combinando las aristas de M y de T, representado en la figura 43.

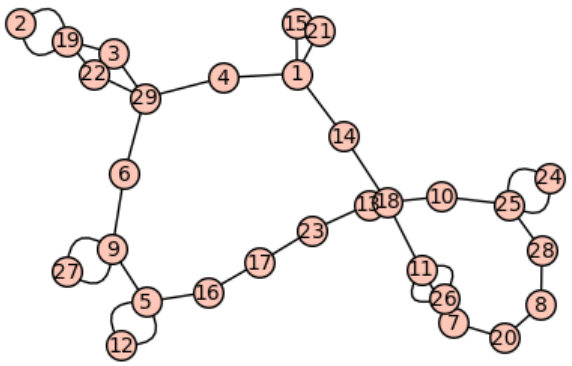


Figura 43: Multigrafo H

Formamos un circuito de Euler en H,

$$19, 2, 19, 3, 29, 6, 9, 27, 9, 5, 12, 5, 16, 17, 23, 13, 10, 25, 24, 25, 28, 8, 20, 7,$$

$$11, 26, 11, 18, 14, 1, 15, 21, 1, 4, 29, 22, 19.$$

Eliminando los vértices repetidos, obtenemos el circuito de Hamilton buscado, es decir, la ruta que debería seguir el aficionado, recorre 11797 millas de distancia.

$$19, 2, 3, 29, 6, 9, 27, 5, 12, 16, 17, 23, 13, 10, 25, 24,$$

$$28, 8, 20, 7, 11, 26, 18, 14, 1, 15, 21, 4, 22, 19.$$

La ruta por ciudades es, Boston, Brooklyn (NY), Washington D.C, Cleveland, Detroit, Toronto, Chicago, Indeanápolis, Milwaukee, Mineápolis, Phoenix, Los Ángeles, San Francico, Sacramento, Portland, Salt Lake City, Denver, Oklahoma City, Dallas, Houston, San Antonio, Nueva Orleans, Memphis, Atlanta, Miami, Orlando, Charlotte, Filadelfia, para posteriormente volver al Madison Square Garden (NY).

Veamos las diferentes rutas obtenidas por algunos de los métodos aproximados anteriormente explicados.

El método nearest insertion proporciona una ruta de 11458 millas, esta ruta es,

$$19, 3, 2, 27, 9, 17, 16, 5, 12, 6, 14, 20, 7, 8, 28, 24,$$

$$25, 10, 13, 23, 26, 11, 18, 15, 21, 1, 4, 29, 22, 19.$$

El método farthest insertion proporciona una ruta de 10017 millas, esta ruta es,

$$19, 3, 2, 15, 21, 4, 1, 18, 14, 20, 7, 11, 26, 23, 13, 10,$$

$$25, 24, 28, 8, 17, 16, 5, 12, 9, 6, 27, 29, 22, 19.$$

El método cheapest insertion proporciona una ruta de 11113 millas, esta ruta es,

$$19, 3, 2, 27, 9, 6, 17, 16, 5, 12, 14, 20, 7, 8, 28, 24$$

$$25, 10, 13, 23, 26, 11, 18, 15, 21, 1, 4, 29, 22, 19.$$

El método nearest neighbors proporciona una ruta de 13651 millas, esta ruta es,

$$19, 3, 22, 29, 27, 6, 9, 12, 5, 16, 17, 14, 18, 11, 7$$

$$20, 26, 8, 28, 23, 13, 10, 25, 24, 15, 21, 4, 1, 2, 19.$$

El método 2-opt proporciona una ruta de 11623 millas, esta ruta es,

$$19, 3, 22, 29, 2, 27, 9, 6, 4, 1, 21, 15, 18, 14, 20,$$

$$7, 11, 26, 23, 13, 10, 25, 24, 28, 8, 17, 16, 12, 5, 19.$$

Por lo tanto, el aficionado a la NBA deberá seguir la ruta proporcionada por el método farthest insertion, que proporciona una de 10017 millas para visitar todos los pabellones de la NBA.

Es decir, comenzar en el Madison Square Garden, para posteriormente visitar los pabellones de Brooklyn Nets, Boston Celtics, Miami Heat, Orlando Magic, Charlotte Hornets, Atlanta Hawks, New Orleans Pelicans, Memphis Grizzlies, Oklahoma City Thunder, Dallas Mavericks, Houston Rockets, San Antonio Spurs, Phoenix Suns, LA Lakers/Clippers, Golden State Warriors, Sacramento Kings, Portland Trail Blazers, Utah Jazz, Denver Nuggets, Minnesota Timberwolves, Milwaukee Bucks, Chicago Bulls, Indiana Pacers, Detroit Piston, Cleveland Cavaliers, Toronto Raptors, Washington Wizards y Philadelphia 76ers.

Para los algoritmos aproximados utilizados para la resolución del ejemplo, el tiempo de ejecución es instantáneo. Vemos que la distancia a recorrer por el algoritmo de Christofides es 11797 millas, muy similar a la distancia a recorrer por los métodos nearest insertion, cheapest insertion y el método 2-opt.

Nótese que en el método del vecino más cercano, la distancia a recorrer es mayor, ya que como se ha comentado, en una iteración se escoge la mejor opción disponible y esto puede llevar a malas elecciones en iteraciones finales.

Para este ejemplo, el método más rápido es farthest insertion, que comienza seleccionando la ciudad más lejana y toma repetidamente la ciudad fuera del circuito que se va creando más lejana disponible y la coloca entre las dos ciudades que hacen el circuito más corto posible. No se ha podido comprobar la diferencia con la ruta óptima dada por los algoritmos exactos ya que el tiempo de ejecución que se requiere para estos con un problema de 29 vértices es muy elevado.

La inserción más lejana generalmente (no siempre, como se verá a continuación) produce un mejor recorrido que la más cercana, más barata y el vecino más cercano.

Si ahora reducimos el problema a visitar los primeros 10 pabellones para poder aplicar los algoritmos exactos y compararlos, partiendo del pabellón de Golden State Warriors, el Chase Center, obtenemos las siguientes rutas mediante los diferentes métodos mencionados anteriormente.

Para el nearest insertion, se obtiene la ruta siguiente, que recorre 7187 millas,

$$10, 5, 9, 6, 3, 2, 4, 1, 7, 8, 10.$$

El método farthest insertion proporciona una ruta de 8328 millas,

$$10, 2, 3, 4, 1, 6, 9, 5, 7, 8, 10.$$

Para el algoritmo cheapest insertion se obtiene una ruta de 7522 millas,

$$10, 7, 5, 9, 6, 3, 2, 4, 1, 8, 10.$$

El método nearest neighbors, cuya ruta siguiente recorre 8154 millas,

$$10, 8, 7, 1, 4, 6, 9, 5, 3, 2, 10.$$

El método 2-opt proporciona una ruta de 7621 millas, este recorrido es,

$$10, 8, 7, 1, 4, 2, 3, 6, 5, 9, 10.$$

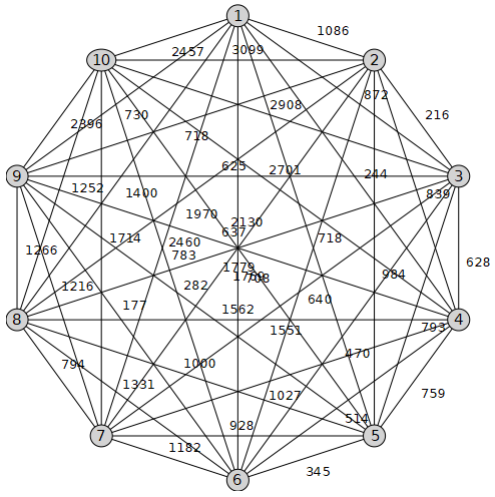


Figura 44: Grafo G

Procediendo mediante el algoritmo de Christofides, donde el grafo ponderado que representa este nuevo problema es $G = K_{10}$, representado en la figura 44, se obtiene tras el primer paso de crear el árbol de expansión mínima T, figura 45, que el conjunto de vértices de grado impar es $O = \{2, 5, 6, 10\}$.

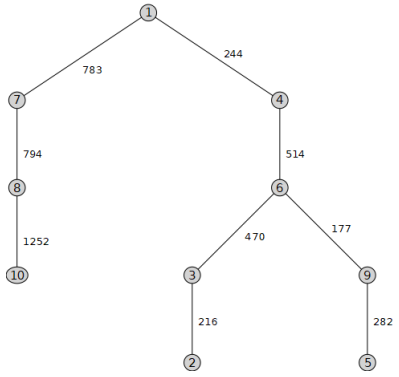


Figura 45: Árbol de expansión T

Formamos el grafo inducido, figura 46, cuyo emparejamiento perfecto de peso mínimo M es el formado por $\{2, 6\}$ y $\{5, 10\}$. Combinando las aristas de M y de T, se obtiene el multigrafo H, representado en la figura 46.

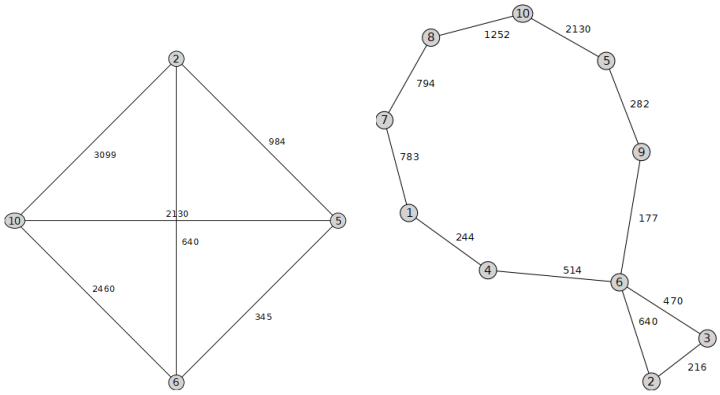


Figura 46: Grafo inducido y multigrafo H

Un circuito de Euler en H es,

$$10, 8, 7, 1, 4, 6, 2, 3, 6, 9, 5, 10.$$

Eliminando el vértice repetido, obtenemos el circuito de Hamilton, es decir, la ruta a seguir que recorre 7187 millas.

$$10, 8, 7, 1, 4, 2, 3, 6, 9, 5, 10.$$

La técnica branch and bound proporciona el mejor recorrido para este problema, ya que la siguiente ruta consta de 6936 millas.

$$10, 7, 1, 4, 3, 2, 6, 9, 5, 8, 10.$$

Mediante el algoritmo de fuerza bruta, cuyo código se recuerda que está disponible en el enlace [12], el tiempo de ejecución para este ejemplo de 10 vértices ha sido de 524 segundos (ejecución con un ordenador de 8 GB RAM, 1000 GB disco duro, 2.5 GHz de velocidad del procesador Intel Core i5). Evidentemente, al ser ambos algoritmos exactos, este método asigna la misma ruta.

En este caso, la salida de los algoritmos aproximados que más se acerca al mínimo número de millas a recorrer dada por los algoritmos exactos, es la ofrecida por el método nearest insertion y el algoritmo de Christofides, cuya salida en ambos casos es una ruta que consta de 7187 millas.

El método que más se aleja de la ruta óptima es el farthest insertion, y sin embargo en el caso anterior con 29 pabellones, este método brinda la mejor de las rutas.

A. Apéndice programación

Veamos como se ha realizado el ejemplo de la sección 5.5, además de algunos de los comandos que han sido utilizados a lo largo del trabajo.

En Maple y Sage se realiza el algoritmo de Christofides (ya que Maple no permite la posibilidad de crear multigrafos). En Rstudio, se han ejecutado los algoritmos aproximados de dicha sección, además de las gráficas 38, 39 mediante la función *plot*().

- **Maple**

```
with(GraphTheory)

## Grafo representativo de las distancias entre los 29 pabellones
G:=Graph({[{1,2},1086], [{1,3}, 872], ..., [{27,28},1901]})

## Arbol de expansion minima
T:=MinimalSpanningTree(G)
Edges(T, weights)
DrawGraph(T)

## Grafo inducido
G1:=Graph({[{1,2},1086], [{1,3}, 872], ..., [{27,29},482]})
DrawGraph(G1)
```

- **Sage**

```
## Grafo inducido
G = Graph()
G.add_edges([(1, 2, 1086), (1, 3, 872), ..., (27, 29, 482)])
G.perfect_matchings()
```

```
## Multigrafo
H = Graph(multiedges=True, sparse=True)
H.add_edges([(1, 4, 244), (1, 14, 383), ..., (24, 25, 580)])
H.show()
H.eulerian_circuit()
```

- **Rstudio**

```
V1 <- c(0, 1086, 872,..., 968, 1870, 637) ##Atlanta Hawks...
V29 <- c(637, 439, 229,..., 482, 2083, 0) ##Washington
matriz <- rbind(V1,..., V29)

library("TSP")
etsp <- ETSP(matriz)

## calculate a tour using some methods
methods <- c("nearest_insertion", "farthest_insertion",
             "cheapest_insertion", "nn", "2-opt")

tours <- sapply(methods, FUN = function(m) solve_TSP(etsp,
                 method = m), simplify = FALSE)
```

La función *sapply* en Rstudio es una función vectorizada de la familia de aplicaciones que permite iterar sobre una lista o vector sin la necesidad de usar el bucle for, en este caso, sobre los diferentes algoritmos aproximados del vector *methods* y FUN, función a la que se aplica y sus argumentos adicionales.

Algunos de los comandos (en Maple) que se han usado durante el trabajo son,

- Crear grafos completos o ciclos.

```
CompleteGraph(n)
CycleGraph(n)
```

- Matriz de ayacencia e incidencia, ejemplos 4 y 5.

```
## Formar grafos dirigidos
H := Digraph([1, 2, 3, 4, 5, 6], {[1, 2], [1, 4], [1, 6],
                 [2, 3], [3, 5], [4, 5], [5, 6], [6, 2]})
AdjacencyMatrix(H)

G := Graph({{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 5}, {2, 6},
              {3, 4}, {4, 5}, {5, 6}})
IncidenceMatrix(G)
```

- Remarcar vértices o aristas, como en el ejemplo 8.

```
H := Graph({{1, 2}, {1, 5}, {2, 3}, {3, 4}, {3, 6}, {4, 5},
            {6, 7}, {6, 9}, {7, 8}, {7, 9}, {8, 9}})
HighlightedVertices(H, 3)
DrawGraph(H)

G := Graph({{1, 2}, {1, 3}, {2, 3}, {2, 4}, {4, 5},
            {4, 6}, {5, 6}, {6, 7}})
HighlightEdges(G, {2, 4})
DrawGraph(G)
```

- Contemplar subgrafos, como en el ejemplo 2 o también para remarcar los emparejamientos de la figura 8.

```
G := Graph({{1, 2}, {1, 3}, {2, 3}, {3, 4}, {4, 5},
            {4, 6}, {5, 6}})
H := Graph([1, 2, 3], {{1, 2}, {1, 3}, {2, 3}})
G0 := HighlightSubgraph(G, H, inplace = false)
DrawGraph(G0)
```

- Algoritmos para árboles de expansión mínima, como en el ejemplo 17.

```
G := Graph({[{a, b}, 2], [{a, e}, 3], [{b, c}, 3],
[{b, f}, 1], [{c, d}, 1], [{c, g}, 2], [{d, h}, 5],
[{e, f}, 4], [{e, i}, 4], [{f, g}, 3], [{f, j}, 2],
[{g, h}, 3], [{g, k}, 4], [{h, l}, 3], [{i, j}, 3],
[{j, k}, 3], [{k, l}, 1]})
T := KruskalsAlgorithm(G, w) ## w es la ruta
T1 := PrimsAlgorithm(G, w1)
G0 := HighlightSubgraph(G, T, inplace = false)
G1 := HighlightSubgraph(G, T1, inplace = false)
DrawGraph(G0)
DrawGraph(G1)
```

- Encontrar el TSP mediante un algoritmo de ramificación y acotación, ejemplo 21.

```
G := Graph({[{1, 2}, 2], [{1, 3}, 3], [{1, 4}, 5],
[{1, 5}, 1], [{1, 6}, 7], [{2, 3}, 1], [{2, 4}, 6],
[{2, 5}, 3], [{2, 6}, 11], [{3, 4}, 5], [{3, 5}, 4],
[{3, 6}, 9], [{4, 5}, 6], [{4, 6}, 8], [{5, 6}, 1]})
w, tour := TravelingSalesman(G)
HighlightTrail(G, tour, red) ## Marcar ruta
DrawGraph(G)
```

Referencias

[1] José Vicente ÁLvarez. Complejidad Algorítmica. Departamento de Informática, Universidad de Valladolid, Campus de Segovia.

https://www2.infor.uva.es/~jvalvarez/docencia/tema5.pdf

[2] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report (1976).

[3] Francesc Comellas, Josep Fàbrega, Anna Sànchez y Oriol Serra. Matemática discreta. Edicions de la Universitat Politècnica de Catalunya (2001).

https://www.academia.edu/38586061/Matem%C3%A1tica_discreta

[4] William Cook. Concorde Home (2020).

https://www.math.uwaterloo.ca/tsp/concorde.html

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to algorithms. Second Edition McGraw-Hill and The MIT Press (2001).

[6] Donald Davendra. Traveling Salesman Problem, Theory and Applications. InTech (2010).

[7] Harold Neil Gabow. Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs. Ph.D. Thesis, Stanford University (1974).

[8] B. Golden, L. Bodin, T. Doyle and W. Stewart. Traveling Salesman Problem Insertion Algorithms. Operations Research (1980).

https://www2.isye.gatech.edu/~mgoetsch/cali/VEHICLE/TSP/TSP003__.HTM

[9] Eugene Lawler. Combinatorial Optimization: Networks and Matroids. New York: Holt, Rinehart and Winston (1976).

[10] José Jesús Mendoza. Diseño de algoritmos heurísticos y metaheurísticos eficientes para resolver el problema del agente viajero. Universidad Nacional Autónoma de Nicaragua (2017).

https://1library.co/document/qmo7344y-raveling-salesman-problem-algoritmos-heuristicos-metaheuristicos-eficientes-problema.html

[11] Julio José Moyano Fernández. Elementos de Matemática Discreta. Colección Sapientia 170, Publicacions de la Universitat Jaume I, Primera edición (2021).

http://repositori.uji.es/xmlui/handle/10234/195193

[12] Gerhard Reinelt. TSP Brute. TSPLIB, A Traveling Salesman Problem Library (1991).

people.sc.fsu.edu/~jburkardt/c_src/tsp_brute/tsp_brute.html

[13] Kenneth H. Rosen. Discrete Mathematics and Its Applications. Seventh Edition McGraw-Hill (2012).

[14] Francisco J. Veiga. El problema del viajante. Universidad de Santiago de Compostela (2013).

http://eio.usc.es/pub/mte/descargas/proyectosfinmaster/proyecto_774.pdf

[15] Lawrence Weru. 11 Animated Algorithms for the Traveling Salesman Problem. Stem Lounge.

https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/

[16] Douglas B. West. Introduction to graph theory. Second edition Pearson Education, Inc (2001).

[17] r-NBA. Reddit (2018).

github.com/rv94/NBA-Heatmaps/blob/master/League%20Distances.csv#L715

[18] Complejidad temporal. Wikipedia, Wikimedia Foundation (2022).

[19] Problema P contra NP. Wikipedia, Wikimedia Foundation (2022).

[20] Problema del viajante. Wikipedia, Wikimedia Foundation (2022).

[21] National Basketball Association. Wikipedia, Wikimedia Foundation (2022).