Departamento de Matemáticas, Facultad de Ciencias
Universidad Autónoma de Madrid

Algoritmos Voraces

Algoritmos Voraces

Grado en Matemáticas

Autor: Alejandro Sánchez Sanz

Tutores: Daniel Ortega Rodrigo, Fernando Quirós Gracián

Curso 2020-2021

Resumen

Existen muchos tipos de algoritmos para resolver problemas de optimización o búsqueda, pero posiblemente los más directos y simples sean los algoritmos voraces. En este trabajo vamos a tratar de estudiarlos en detalle. En el capítulo 1 nos plantearemos diversas cuestiones acerca de sus características y su uso, y proporcionaremos al lector unas definiciones preliminares de conceptos que usaremos posteriormente. En el capítulo 2 enunciaremos dos problemas clásicos: el problema del árbol recubridor mínimo y el problema del viajante; donde los algoritmos voraces funcionan de manera óptima o no, respectivamente. Durante el capítulo 3 profundizaremos en qué son los algoritmos voraces, sus características generales y la estrategia que suelen seguir. Los capítulos 4 y 5 explicarán en detalle los problemas de ejemplo del segundo capítulo, mostrando dos algoritmos voraces para cada uno de ellos. En todos los casos daremos demostraciones y pseudocódigos de posibles implementaciones de estos algoritmos. Llegados a este punto, en el capítulo 6, cerraremos el trabajo con las conclusiones obtenidas tras todo nuestro análisis. Finalmente, en los apéndices listaremos parte del código que hemos implementado en *Sage* para ver los algoritmos en acción.

Abstract

There are many types of algorithms to solve optimization or search problems, but possibly the simplest and most straightforward are the greedy algorithms. In this project we will try to study them in detail. In chapter 1 we will ask ourselves various questions about their features and usage, and we will provide the reader with some preliminary definitions of concepts for later use. In chapter 2 we will state two classic problems: the minimum spanning tree problem and the travelling salesman problem, where the greedy algorithms find an optimal solution or not, respectively. Along chapter 3 we will dig into what greedy algorithms are, their general features and the strategy they usually follow. Chapters 4 and 5 will explain in detail the model problems of the second chapter, showing two greedy algorithms for each of them. For each case, we will present proofs and pseudocodes of possible implementations of these algorithms. At this point, in chapter 6, we will close the project with the conclusions obtained after all our analysis. Finally, in the appendixes we will list part of the code that we have implemented in *Sage* to see the algorithms in action.

Índice general

CAPÍTULO 1
Introducción y preliminares

1.1. Introducción

Los algoritmos voraces solucionan problemas de búsqueda por medio de **elecciones locales óptimas** (en algún sentido) en cada paso con la intención de encontrar una solución global óptima. Pero hay problemas donde esta estrategia voraz nos da una solución subóptima (una aproximación) o incluso, la peor solución. Este TFG trata de estudiar las razones por las que esto ocurre y realizar implementaciones y pruebas sobre algunos de estos algoritmos.

Los resultados tratarán de contestar a las siguientes preguntas:

1. ¿Para qué problemas producen los algoritmos voraces una solución global óptima?
2. ¿Para qué problemas producen los algoritmos voraces una buena aproximación a una solución global óptima?
3. ¿Para qué problemas se sabe que los algoritmos voraces no producen una solución global óptima, ni una buena aproximación?

Además se van a estudiar más en detalle distintos algoritmos voraces para problemas clásicos como el del viajante o el de la mochila, o también para problemas relacionados con grafos (como encontrar el árbol recubridor mínimo). Se implementarán en *Sage* y se realizarán pruebas para medir su complejidad (pudiendo compararla con la obtenida de modo teórico).

Este trabajo nos permitirá comprender mejor estos algoritmos y entender por qué funcionan tan bien en muchos casos a pesar de su simplicidad. Pero de igual manera, comprender en qué casos no y por qué. Cuando esto suceda querremos ver la aproximación que proporcionan y cuánto de buena es, pues en algunos problemas puede merecer la pena usar estos algoritmos por su velocidad. Dicho esto, se espera poder verlos *en acción* tras programarlos y realizar pruebas sobre ellos.

1.2. Definiciones preliminares

- **Algoritmo**: Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución.
- **Árbol**: Grafo no dirigido en el que cualquier par de vértices está conectado por exactamente un camino.
- **Árbol binario**: Árbol en el que cada nodo tiene como mucho dos hijos, uno izquierdo y uno derecho.
- **Árbol completo**: Árbol en el que todos sus niveles están llenos salvo el último, que está ocupado de izquierda a derecha.
- **Bottom-Up**: Técnica de resolución de problemas que va buscando soluciones a los subproblemas de menor a mayor, hasta terminar con la solución del problema completo. Cuando se resuelve un subproblema particular, ya se han resuelto los subproblemas menores de los que depende su solución.
- **Camino**: Sucesión de enlaces adyacentes en un grafo.
- **Ciclo**: Camino cerrado en un grafo en el que no se repite ningún vértice a excepción del primero, que aparece dos veces, como principio y fin del camino.
- **Ciclo hamiltoniano**: Ciclo en el que se visita cada vértice del grafo una sola vez. Está formado por tantos enlaces como número de vértices tenga el grafo.
- **Complejidad algorítmica**: Métrica teórica que nos ayuda a describir el comportamiento de un algoritmo en términos de sus recursos (tiempo de ejecución y memoria requerida).
- **Componente conexa (de un grafo)**: Cada uno de los subgrafos conexos de un grafo no conexo. En el caso de un grafo conexo, coincide con él mismo.
- **Diccionario**: Estructura de datos mutable compuesta de un conjunto (no necesariamente ordenado) de pares clave-valor.
- **Grafo**: Representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas que reciben el nombre de aristas o enlaces. Lo representaremos por $G =< N, A >$, donde N es el conjunto de vértices y A el conjunto de enlaces.
- **Grafo completo**: Grafo donde cada par de vértices está conectado por un enlace, es decir, existen enlaces uniendo *todos* los pares posibles de vértices.
- **Grafo conexo**: Grafo donde cada par de vértices está conectado por al menos un camino. Posee una única componente conexa.
- **Grafo denso**: Grafo cuyo número de enlaces es cercano a su número máximo, es decir, a los que tendría si fuera completo.

- **Grafo dirigido**: Grafo donde los enlaces son unidireccionales. Sus enlaces se representan como pares ordenados (u, v), siendo u, v vértices del grafo.
- **Grafo disperso**: Grafo cuyo número de enlaces es muy bajo, cercano a los que tendría si fuera vacío.
- **Grafo no dirigido**: Grafo donde los enlaces son bidireccionales. Sus enlaces se representan como pares no ordenados $\{u, v\}$, siendo u, v vértices del grafo.
- **Grafo ponderado**: Grafo donde cada enlace tiene asignado un peso, valor o coste.
- **Memoización**: Técnica de optimización del tiempo de ejecución de algoritmos. Consiste en almacenar los resultados intermedios de ciertas funciones para no tener que ejecutarlas de nuevo en futuras llamadas.
- **Memoria requerida (en un algoritmo)**: Cantidad de memoria necesaria para procesar las instrucciones que solucionan dicho problema.
- **Notación asintótica**: Notación que permite analizar la complejidad de un algoritmo identificando su comportamiento si el tamaño de entrada para el algoritmo aumenta.
- **Notación Ω (*Omega-grande*)**: Cota inferior asintótica.
- **Notación O (*O-grande*)**: Cota superior asintótica.
- **Notación Θ (*Theta-grande*)**: Cota ajustada asintótica. Es la igualdad entre la O y la Ω.
- **Tamaño (de un algoritmo)**: Número de datos elementales sobre los que actuará el algoritmo. Es dependiente del problema.
- **Tiempo de ejecución (de un algoritmo)**: Tiempo que tarda un algoritmo en resolver un problema. Puede medirse en tiempo real o calcularse multiplicando el número de instrucciones por el tiempo que tarda cada una de ellas. Suele darse como estimación.
- **Top-Down**: Técnica de resolución de problemas que va buscando soluciones a los subproblemas de mayor a menor, siguiendo un modo recursivo y generalmente empleando memoización.

CAPÍTULO 2
Algunos problemas clásicos

2.1. Encontrar el árbol recubridor mínimo en un grafo

El problema del árbol recubridor mínimo (*Minimum Spanning Tree*) es formulado como un problema de teoría de grafos: Dado un **grafo conexo ponderado y no dirigido** $G =< N, A >$, encontrar el menor subárbol de G que incluya a todos sus vértices. Por *menor*, consideramos aquel que minimice la suma del coste de sus enlaces. La solución óptima no tiene por qué ser única, ya que podríamos encontrar subárboles con el mismo coste óptimo pero con distintos enlaces. Nuestros algoritmos buscarán una de ellas.

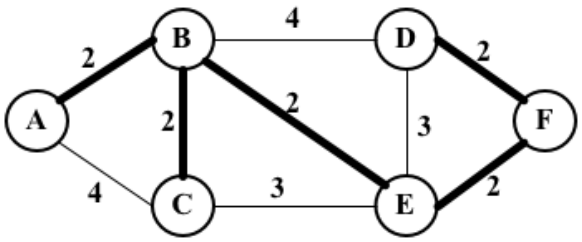

Figura 2.1: Ejemplo de grafo conexo ponderado y no dirigido

Buscamos un subconjunto T de enlaces que encuentren la solución óptima de conectar todos los vértices. Consideremos el grafo parcial $G' =< N, T >$ y supongamos que hay n vértices en N. Como G' es conexo, debe tener al menos $n-1$ enlaces, así que este es el número mínimo de enlaces que puede tener T. Para ver que este es de hecho el número exacto, solo tenemos que ver que si en T hubiera alguno más, entonces existiría al menos un ciclo. Y en ese caso, quitando uno de los enlaces de ese ciclo, G' seguiría siendo conexo y el coste total sería menor, por lo que este nuevo conjunto de enlaces sería preferible. Por tanto, T debe tener exactamente $n-1$ enlaces, y como G' es conexo, debe ser un árbol.

Las aplicaciones de este problema son variadas. Es muy típico el ejemplo en telecomunicaciones. Podemos modelar la comunicación telefónica de una ciudad suponiendo que los vértices son las casas y los enlaces son las líneas telefónicas entre ellas. El cos-

te de cada enlace sería el coste real de desplegar esa línea telefónica. Resolver este problema supondría encontrar el modo más barato de conectar telefónicamente todas las casas (utilizando solo enlaces directos entre ellas, porque el problema se podría complicar si consideramos la posibilidad de crear vértices intermedios en repetidores o torres de telefonía).

Para encontrar un algoritmo voraz que resuelva este problema, parece que hay dos caminos posibles a primera vista:

- Comenzar con un conjunto vacío de enlaces y escoger a cada paso el enlace menos costoso del grafo que esté disponible, es decir, que no haya sido elegido o rechazado todavía.
- Comenzar con un nodo arbitrario del grafo y tratar de ir construyendo el árbol desde él. Para eso, hay que escoger a cada paso el enlace menos costoso que esté disponible y pueda extender el árbol.

Parece sorprendente, pero en este problema ambos caminos funcionan por medio de los algoritmos de **Kruskal** y **Prim**, respectivamente. Los mostraremos en detalle en el capítulo 4.

2.2. Problema del viajante (TSP)

El problema del viajante (o también conocido como TSP por sus siglas en inglés, *Travelling Salesman Problem*) consiste en encontrar el camino único más corto que, dada una lista de ciudades y las distancias entre ellas, visita todas las ciudades una sola vez y regresa a la ciudad de origen. A este camino con origen y fin en la misma ciudad, lo llamaremos *tour*.

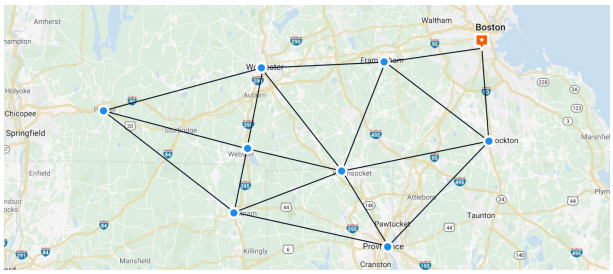

Figura 2.2: Ejemplo del TSP

No tiene un origen claro, aunque se menciona en un manual alemán para viajantes de 1832. La primera formulación matemática fue realizada en el siglo XIX por William Rowan Hamilton y Thomas Kirkman. El primero creó *el Juego Icosian*, que era un rompecabezas recreativo basado en encontrar un ciclo hamiltoniano en él. Su relación con nuestro problema es que la solución del TSP sería el ciclo hamiltoniano de menor longitud en un grafo conexo.

Algunas de las aplicaciones más interesantes de esta casuística se incluyen en problemas de *scheduling*, producción y planificación de rutas en empresas de distribución logística y transporte.

El problema tiene distintas variantes como pueden ser el *TSP simétrico* (cuando el coste de ir de una ciudad a otra es el mismo que el de vuelta), *TSP asimétrico* (cuando el coste entre la ida y la vuelta no es el mismo) y el *TSP múltiple* (cuando hay más de un viajante y el objetivo es encontrar ciclos que partan de una misma ciudad y pasen por cierta parte de las otras, como podría pasar con un almacén del que salen 3 camiones distintos a repartir por distintas ciudades cada uno). También se suele distinguir aquellos en los que la distancia entre ciudades es la euclídea, conociéndose como *TSP euclideano*. Nosotros vamos a centrarnos después en el simétrico.

El TSP simétrico puede ser modelado como un grafo ponderado no dirigido, de manera que las ciudades sean los vértices del grafo, los caminos son los enlaces y las distancias de los caminos son los pesos de dichos enlaces. Esto es un problema de minimización que comienza y termina en un vértice específico y se visita el resto de los vértices exactamente una vez (lo cual es, exactamente, encontrar un ciclo hamiltoniano de longitud mínima dentro del grafo). Con frecuencia, el modelo es un grafo completo. Si no existe camino entre un par de ciudades, se añade arbitrariamente un enlace largo para completar el grafo sin afectar el recorrido óptimo.

En el problema se presentan $N!$ rutas posibles, aunque se puede simplificar ya que dada una ruta nos da igual el punto de partida y esto reduce el número de rutas a examinar en un factor N quedando $(N-1)!$. Como no importa la dirección en que se desplace el viajante, el número de rutas a examinar se reduce nuevamente en un factor 2. Por lo tanto, hay que considerar $\frac{(N-1)!}{2}$ rutas posibles.

En la práctica, para un problema del viajante con 5 ciudades hay $\frac{(5-1)!}{2} = 12$ rutas diferentes y no necesitamos un ordenador para encontrar la mejor ruta, pero la simplicidad aparente es engañosa, ya que apenas aumentamos el número de ciudades las posibilidades crecen factorialmente:

- Para 10 ciudades hay $\frac{(10-1)!}{2} = 181\,440$ rutas diferentes.
- Para 30 ciudades hay más de $4 * 10^{30}$ rutas posibles. Un ordenador que calcule un millón de rutas por segundo necesitaría 10^{17} años para resolverlo. Dicho de otra forma, si se hubiera comenzado a calcular al comienzo de la creación del universo (hace unos 13 400 millones de años) todavía no se habría terminado.

Puede comprobarse que por cada ciudad nueva que incorporemos, el número de rutas se multiplica por el factor N y crece factorialmente. Este problema es muy famoso y ha sido estudiado en gran profundidad, así que se conoce que posee una gran complejidad y, de hecho, pertenece a la clase de problemas ***NP-completos***. En el capítulo 5 explicaremos qué significa esto y veremos que no existen algoritmos que puedan calcular la solución óptima en tiempo polinomial, pero sí existen buenas aproximaciones en forma de algoritmos voraces, como son el **Algoritmo del MST** (referido a *Minimum Spanning Tree*) o el **Algoritmo del Vecino Más Próximo**.

CAPÍTULO 3

Algoritmos voraces: qué son, para qué se usan, características...

Los algoritmos voraces se caracterizan por buscar una solución local óptima a cada paso, con el fin de encontrar una solución global óptima al final del proceso. Para cada decisión no requieren de más información de la que se posee en ese momento (no miran todo el espacio del problema) y no se preocupan del efecto que estas decisiones pueden tener en el futuro. Por tanto, son muy directos, eficientes, fáciles de implementar y, para muchos problemas, óptimos. Sin embargo, el mundo no es tan simple y muchos problemas no pueden solucionarse del mejor modo con este tipo de algoritmos.

Los algoritmos voraces suelen usarse para resolver problemas de optimización (de diversas áreas). Ya hemos comentado el problema del árbol recubridor mínimo y el del viajante, pero hay muchos otros ejemplos clásicos como el de la mochila, el del planificador de tareas o el de dar cambio. Veamos este último como ejemplo:

- Supongamos que tenemos monedas de distintos valores. El problema busca pagar a alguien una cantidad concreta, empleando el menor número de monedas. Para que exista siempre solución exacta, vamos a considerar que tenemos suficientes monedas de cada tipo para poder escoger a cada paso. En caso contrario, el problema se complica.
- Por ejemplo, si estamos en España tenemos 9 tipos de monedas: 2€ (200 céntimos), 1€ (100 céntimos), 50 céntimos, 20 céntimos, 10 céntimos, 5 céntimos, 2 céntimos y 1 céntimo. Si tuviéramos que pagar 3,41€ (341 céntimos), la mejor solución sería usar 1 moneda de 2€, 1 moneda de 1€, 2 monedas de 20 céntimos y 1 moneda de 1 céntimo.
- Esto puede darse en un día cotidiano y lo resolvemos sin darnos cuenta de que estamos empleando un algoritmo voraz: a cada paso, escogemos la moneda de mayor valor que esté disponible con la que no nos pasemos de la cantidad.

Si queremos resolver un problema con un algoritmo de este tipo, tenemos que desarrollar una estrategia voraz. Esta suele seguir los siguientes pasos:

1. Considerar el problema de optimización como uno en el que hacemos una elección y nos quedamos con un subproblema que resolver.
2. Probar que siempre hay una solución óptima al problema original que realiza la elección voraz, es decir, que hacer la elección es seguro.
3. Demostrar la subestructura óptima de modo que, al hacer la elección voraz, lo que queda es un subproblema con la propiedad de que ese elemento elegido, unido con la solución del subproblema, proporciona una solución óptima al problema original.

¿Pero cómo podemos asegurar que un problema va a poder ser resuelto de manera óptima por un algoritmo voraz? Para empezar, el problema debe ser capaz de descomponerse en subproblemas más pequeños. Después, sobre eso, hay que garantizar básicamente dos propiedades:

- **Propiedad de elección voraz**: Se puede formar una solución global óptima a base de hacer elecciones locales "voraces" óptimas. Es decir, poder ir eligiendo a cada paso lo que decidimos es mejor (dejando un subproblema menor a cada paso), sin considerar los resultados del resto de subproblemas. En el ejemplo del cambio, elegimos en cada paso la moneda de mayor valor disponible con la que no nos pasemos del precio.
- **Subestructura óptima**: Una solución óptima del problema debe contener en ella las soluciones óptimas a los subproblemas. Para esto, es imprescindible que los subproblemas sean **independientes**, es decir, que la solución de uno de ellos no afecte a la solución de otro. Dicho de otro modo, dos subproblemas son independientes cuando no comparten recursos. En el ejemplo del cambio, está claro que después de escoger una moneda a cada paso, tenemos un problema similar al original pero con una cantidad de dinero objetivo menor. Y la elección anterior no afecta a la solución de este nuevo subproblema (recordemos que estamos considerando que tenemos suficientes monedas como para no quedarnos sin un tipo a cada paso).

Es obvio que la primera propiedad va de la mano con la estrategia que siguen los algoritmos voraces que hemos mencionado al comienzo del capítulo, pero es fundamental garantizarla para que la solución obtenida siguiendo esa estrategia sea óptima. Para poder mejorar la eficiencia del algoritmo, es recomendable preprocesar los datos de entrada o emplear estructuras de datos apropiadas (como ya veremos, suelen ser *heaps* o *colas de prioridad*), ya que permitirá realizar estas elecciones voraces en menor tiempo.

En cuanto a la segunda propiedad, en los algoritmos voraces solo tendremos que demostrar que una solución óptima a un subproblema, junto con la elección voraz ya hecha, proporciona una solución óptima al problema original. Para esto, se usa inducción en los subproblemas. Esta propiedad aparece también en la **programación dinámica**, pues esos algoritmos y los voraces tienen muchas similitudes. De hecho, para cada algoritmo voraz suele haber otra solución más enrevesada basada en programación dinámica.

A la hora de resolver un problema usando programación dinámica hay dos maneras: *bottom-up* o *top-down*. Los algoritmos voraces siguen la segunda, ya que van haciendo elecciones (voraces) en problemas más grandes hasta los más pequeños.

Ahora estudiemos más en detalle los algoritmos voraces. Veamos algunas características generales que poseen:

1. Buscan la solución óptima para un problema. Tenemos un conjunto de candidatos para formar esa solución. En el ejemplo del cambio, el conjunto de candidatos son las monedas de cada tipo.

2. Vamos acumulando dos conjuntos más: uno con los candidatos elegidos y otro con los rechazados.

3. Hay una función, digamos *solucion(S)*, que comprueba si el conjunto de elegidos soluciona el problema, ignorando la optimalidad por el momento. En el ejemplo, comprueba si la suma del valor de las monedas elegidas es *exactamente* la cantidad total a pagar.

4. Otra función, *factible(S)*, comprueba si el conjunto de candidatos es factible, es decir, si puede producir una solución añadiendo más candidatos a ese conjunto o no (de nuevo sin mirar optimalidad). Solemos esperar que exista al menos una solución con los candidatos que tenemos. En el cambio, comprobamos que no se exceda la cantidad total.

5. Y otra función, *seleccionar(C)*, comprueba qué candidato de los restantes es el más adecuado para añadirse a la lista de elegidos. En el caso de las monedas, selecciona la moneda de mayor valor disponible.

6. Finalmente, hay una función objetivo que nos da el valor de la solución obtenida. A diferencia de las tres funciones anteriores, esta no aparece explícitamente en el algoritmo voraz. En nuestro ejemplo, esta función cuenta el número de monedas usadas en la solución.

Para solucionar el problema, buscamos un conjunto de candidatos que constituya una solución y optimice (maximice o minimice, según el caso) el valor de la función objetivo. Un algoritmo voraz trabaja paso a paso:

- El conjunto de elegidos empieza vacío.
- Según la función de selección, consideramos añadir a este conjunto el mejor candidato restante. Decimos "consideramos" porque solo se añade en caso de que el conjunto de elegidos resultante siga siendo factible. Si no lo fuera, entonces mandamos ese candidato al conjunto de rechazados.
- Cada vez que añadimos un candidato al conjunto de elegidos, comprobamos si constituye una solución a nuestro problema. Si no, volvemos al paso anterior considerando añadir a otro candidato.

- Cuando un algoritmo voraz funciona correctamente, la primera solución obtenida de esta manera es siempre óptima.

Es obvio el nombre de "voraz", ya que escoge el "mejor" candidato a cada paso, sin importar nada más. Si es elegido, siempre estará incluido en la solución, y si es rechazado, nunca lo estará.

Este podría ser el pseudocódigo genérico de un algoritmo voraz:

```
funcion voraz(C: conjunto): conjunto
    # C es el conjunto de candidatos
    S ← ∅ # Construimos la solución en el conjunto S
    mientras C ≠ ∅ y no solucion(S), hacer
        x ← seleccionar(C)
        C ← C \ {x}
        si factible(S ∪ {x}) entonces S ← S ∪ {x}
    si solucion(S) entonces
        devolver S
    en otro caso
        devolver "No hay soluciones"
```

Las funciones de selección y objetivo suelen estar relacionadas (dependiendo de si queremos maximizar o minimizar, por ejemplo) y pueden existir distintas funciones de selección posibles.

Para terminar el capítulo volvamos al ejemplo del cambio. Este sería el posible algoritmo que hemos comentado:

```
funcion dar_cambio(n): conjunto de monedas
    '''
      Dar cambio de una cantidad n, usando el menor número de
      monedas. La constante C especifica los tipos de monedas
      en céntimos
    '''
    const C = {200, 100, 50, 20, 10, 5, 2, 1}
    S ← ∅ # Construimos la solución en el conjunto S
    s ← 0 # s es la suma de los elementos de S
    mientras s ≠ n, hacer
        x ← el elemento mayor de C tal que s + x ≤ n
        si no hay tal elemento entonces
            devolver "No se encuentra solución"
        S ← S ∪ {una moneda de valor x}
        s ← s + x
    devolver S
```

CAPÍTULO 4
Problema del árbol recubridor mínimo

Comencemos con el esquema general ya mencionado de los algoritmos voraces en este problema:

- Los candidatos son los enlaces de G.
- Un conjunto de enlaces forma una solución si es un árbol recubridor para los vértices en N.
- Un conjunto de enlaces es factible si no contiene ciclos.
- La función de selección varía con el algoritmo.
- La función objetivo que tratamos de minimizar es el coste total de los enlaces en la solución.

Ahora necesitaremos unas definiciones y un lema antes de poder entrar en los detalles de los algoritmos:

Definición 4.1. Un conjunto factible de enlaces es **prometedor** si puede extenderse hasta producir una solución óptima.

En particular, el conjunto vacío es siempre prometedor. Además, si un conjunto prometedor de enlaces es ya una solución, entonces su extensión será vacía y eso querrá decir que es ya una solución óptima.

Definición 4.2. Un enlace **sale** de un determinado conjunto de vértices si tiene exactamente un extremo en ese conjunto.

Por tanto, un enlace no lo cumple si sus extremos están ambos en el conjunto o si ninguno lo está.

Lema 4.3. *Sea $G =< N, A >$ un grafo conexo no dirigido, donde cada coste es conocido. Sea $B \subset N$ un subconjunto estricto de vértices de N. Sea $T \subseteq A$ un conjunto prometedor de enlaces tal que ninguno de sus enlaces sale de B. Sea v el enlace más corto que sale de B (o si hay empates, uno de ellos). Entonces $T \cup \{v\}$ es prometedor.*

Demostración. Sea U un árbol recubridor mínimo de G tal que $T \subseteq U$. Como suponemos que T es prometedor, entonces ese U debe existir. Si $v \in U$, no hay nada que probar. En caso contrario, al añadir v a U estamos creando exactamente un ciclo, debido a que U es un árbol. En este ciclo, como v *sale* de B, entonces tiene que existir otro enlace que cierre el ciclo, llamémoslo u, que también sale de B. Si ahora quitamos u, el ciclo desaparece y tenemos un nuevo árbol recubridor V. Sin embargo, tal y como habíamos definido v, su coste no puede ser mayor que el de u, por lo que el coste total de los enlaces de V no puede exceder el de los enlaces de U. Por tanto, V es también un árbol recubridor mínimo de G, el cual contiene a v. Para terminar la prueba, hay que remarcar que $T \subseteq V$ porque u salía de B y, por la definición de T, este no podría haber sido uno de sus enlaces. □

Usaremos este lema para demostrar que los siguientes algoritmos encuentran la solución óptima al problema.

A continuación, vamos a mostrar una estructura de datos que será de utilidad para mejorar la eficiencia de estos algoritmos: el *heap*.

Definición 4.4. Un árbol cumple la **condición de heap** si es binario completo y si cada nodo padre tiene un valor mayor (o menor) que cualquiera de sus nodos hijos.

Definición 4.5. Un **heap** es una estructura de datos de tipo árbol con información perteneciente a un conjunto ordenado, que cumple la condición de heap.

Según si en la condición de *heap* el valor de los padres es mayor o menor que el de los hijos, tenemos dos tipos de *heaps*: *max heaps* y *min heaps*, respectivamente.

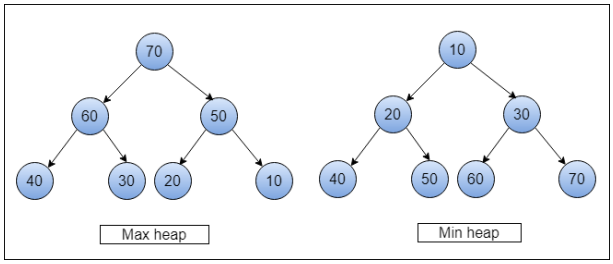

Figura 4.1: Ejemplos de heaps

Esta estructura de datos se emplea mayoritariamente para implementar *colas de prioridad*, ya que en la raíz de los *heaps* tenemos siempre el mayor (*max heap*) o menor (*min heap*) valor, y extraerlo es muy rápido. También es famoso por el algoritmo de ordenación *heapsort*.

Como es un árbol binario completo, puede representarse fácilmente como un vector donde, si hay n nodos y numeramos las posiciones del vector de 0 a $n-1$:

- El nodo raíz ocupa la posición 0 del vector.
- Los hijos de un nodo almacenado en la posición k se almacenan en las posiciones $2k+1$ y $2k+2$, respectivamente.

Dado un conjunto de n valores, sabemos que podemos construir un *heap* en tiempo lineal ($\Theta(n)$), y una vez extraída la raíz, podemos hacer las operaciones para volver a tener la condición de *heap* en tiempo logarítimo ($\Theta(\log n)$). Este último proceso suele conocerse como *heapify*.

Pasemos ahora a hablar de los algoritmos voraces de este capítulo.

4.1. Algoritmo de Kruskal

Como hemos dicho en el capítulo 2.1, este algoritmo sigue el primer camino de resolver el problema y comienza con un conjunto vacío de enlaces, llamémoslo T. En este conjunto vamos a ir formando distintas componentes conexas del grafo G y, de hecho, los enlaces de T de cada componente van a formar un árbol recubridor mínimo de los vértices de dicha componente. Al comenzar, podemos decir que tenemos $n = |N|$ componentes conexas triviales de G, una formada por cada vértice independientemente. Entonces, a cada paso, vamos a escoger el "mejor" (es decir, el de menor coste) enlace restante disponible y añadirlo a T siempre y cuando no forme un ciclo en alguna de las componentes, ya que entonces no tendríamos un árbol. Si al añadir este enlace conectamos dos de las componentes conexas, entonces estas se combinan. El algoritmo continúa hasta que queda una única componente conexa.

Vamos a mostrar un ejemplo con el grafo de la figura 4.2:

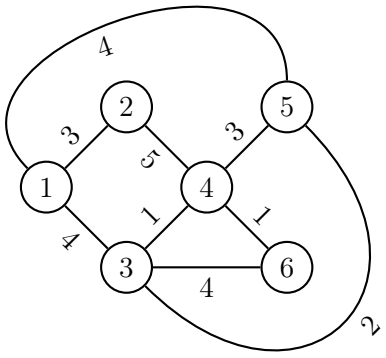

Figura 4.2: Grafo de ejemplo

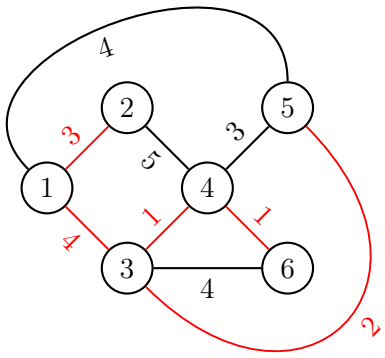

Figura 4.3: Árbol recubridor mínimo. Coste total: 11

Comencemos ordenando los enlaces de menor a mayor coste: {3,4}, {4,6}, {3,5}, {1,2}, {4,5}, {1,3}, {1,5}, {3,6}, {2,4}. Ahora veamos los pasos del algoritmo:

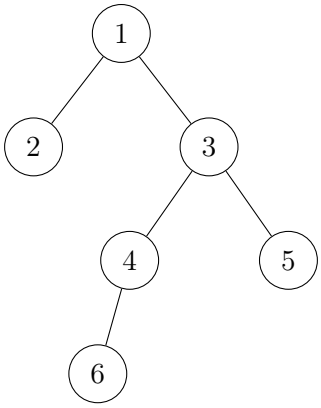

Figura 4.4: Representación más clara del árbol recubridor mínimo anterior

Paso	Enlace seleccionado	Componentes conexas
Inicialización	-	{1} {2} {3} {4} {5} {6}
1	{3,4}	{1} {2} {3,4} {5} {6}
2	{4,6}	{1} {2} {3,4,6} {5}
3	{3,5}	{1} {2} {3,4,5,6}
4	{1,2}	{1,2} {3,4,5,6}
5	{4,5}	rechazado
6	{1,3}	{1,2,3,4,5,6}

Al terminar el algoritmo, T contiene los enlaces {3,4}, {4,6}, {3,5}, {1,2} y {1,3}, con un coste total de 11, como mostramos en la figura 4.3. Estos forman un árbol recubridor mínimo de G, tal y como indica la figura 4.4.

Teorema 4.6. *El algoritmo de Kruskal encuentra un árbol recubridor mínimo.*

Demostración. Vamos a usar inducción sobre el número de enlaces del conjunto T. Debemos probar que si T es prometedor a cada paso del algoritmo, entonces también lo será cuando se añade un enlace adicional. Cuando el algoritmo pare, como la solución nos la da T y este es prometedor, entonces dicha solución será óptima.

- Caso base: El conjunto vacío es prometedor porque G es conexo, así que existe una solución al menos.
- Paso inductivo: Suponemos que T es prometedor antes de añadir un nuevo enlace $e = \{u, v\}$. Los enlaces de T dividen a los vértices de G (i.e. el conjunto N) en una o más componentes conexas, estando u y v en diferentes de ellas. Sea B el conjunto de vértices de la componente que contiene a u:
  - $B \subset N$, pues no contiene a v, por ejemplo.
  - T es un conjunto prometedor de enlaces tal que ninguno de ellos sale de B.
  - e es uno de los enlaces con menor coste que sale de B (ya que es el que el algoritmo quiere añadir a T en este paso).

Como vemos, se cumplen las condiciones del Lema 4.3, así que podemos concluir que el conjunto $T \cup \{e\}$ es prometedor.

Como hemos dicho antes, esta inducción demuestra que al terminar el algoritmo tenemos una solución óptima. □

A la hora de implementar este algoritmo, necesitaremos primero ordenar los enlaces de menor a mayor coste. Después, será importante tener una función que permita encontrar la componente conexa de un vértice y otra que permita combinar componentes conexas. El pseudocódigo sería así:

```
funcion kruskal(G = <N,A>: grafo): conjunto de enlaces
    # G es un grafo ponderado conexo no dirigido

    # Inicialización
    Ordenar A de menor a mayor coste
    n ← número de vértices de N
    T ← ∅ # Contendrá los enlaces de la solución
    Inicializar  n conjuntos,  cada uno con un vértice de N

    # Bucle voraz
    mientras T tenga menos de n−1 enlaces hacer
        e ← {u,v} ← enlace disponible de menor coste
        ucomp ← encontrar(u)
        vcomp ← encontrar(v)
        si ucomp ≠ vcomp entonces
            combinar(ucomp, vcomp)
            T ← T ∪ {e}
    devolver T
```

La manera de representar el grafo va a ser basada en la clase ***lisgraph*** mencionada en el capítulo 5 de [1]. Esta contiene un *diccionario* con las etiquetas de los vértices como claves, y listas de tuplas *(vecino, coste del enlace entre ellos)* como valores.

Estudiemos la complejidad del algoritmo. Si el grafo tiene n vértices y a enlaces, la complejidad del número de operaciones es:

- $\Theta(a \log a)$ para ordenar los enlaces, lo que es equivalente a $\Theta(a \log n)$ pues

$$n - 1 \leq a \leq \frac{n(n-1)}{2}.$$

- $\Theta(n)$ para inicializar los n conjuntos disjuntos.
- $\Theta(\log n)$ para todas las operaciones de *encontrar* y *combinar*.
- $O(a)$ en el peor caso, para el resto de operaciones.

Con todo esto, podemos concluir que la complejidad del algoritmo es $\Theta(a \log n)$. Teniendo en cuenta la dispersión del grafo, podemos ajustar la estimación un poco más. Si el grafo es disperso, a tiende a n, y si el grafo es denso, tiende hacia $\frac{n(n-1)}{2}$. Por tanto en cada caso, la complejidad del algoritmo de Kruskal es $\Theta(n \log n)$ y $\Theta(n^2 \log n)$, respectivamente.

Vale la pena mencionar que usando un *min heap* para almacenar los enlaces del grafo, conseguiríamos mejorar el tiempo especialmente en los casos donde haya muchos enlaces que no haga falta mirar para encontrar el árbol recubridor mínimo. Esto se debe a que el tiempo de creación del *heap* es $\Theta(a)$, aunque no se mejora el análisis del peor caso, ya que el tiempo de las operaciones del bucle voraz ahora es de $\Theta(\log a) = \Theta(\log n)$.

4.2. Algoritmo de Prim

En este segundo algoritmo vamos a seguir el otro camino: escoger un vértice arbitrario e ir añadiendo enlaces garantizando tener siempre un árbol a cada paso, hasta terminar con el árbol recubridor mínimo cuando alcancemos el último vértice.

Sea B un conjunto de vértices y T uno de enlaces. Comenzamos con B con un único vértice arbitrario y T vacío. A cada paso, el algoritmo de Prim añade a T el enlace $\{u, v\}$ disponible de menor coste, que tenga un extremo en B (digamos u) y otro en $N \setminus B$ (digamos v), y por tanto también añade v a B. De este modo, tendremos en T a cada paso un árbol recubridor mínimo de los vértices de B. Esto se repite hasta que todos los vértices del grafo están en B.

Mostremos el algoritmo en acción sobre el mismo grafo de ejemplo anterior de la figura 4.2, comenzando arbitrariamente por el vértice 1:

Paso	Enlace seleccionado $\{u, v\}$	Conjunto B
Inicialización	-	{1}
1	{1,2}	{1,2}
2	{1,3}	{1,2,3}
3	{3,4}	{1,2,3,4}
4	{4,6}	{1,2,3,4,6}
5	{3,5}	{1,2,3,4,5,6}

Al terminar el algoritmo, T contiene los enlaces {1,2}, {1,3}, {3,4}, {4,6} y {3,5}, con un coste total de 11. Forman un árbol recubridor mínimo de G, que coincide con el que encontramos con Kruskal y se muestra en la figura 4.4.

Teorema 4.7. *El algoritmo de Prim encuentra un árbol recubridor mínimo.*

Demostración. Tal y como hicimos para probar el teorema 4.6, vamos a usar inducción sobre el número de enlaces del conjunto T. Debemos probar que si T es prometedor a cada paso del algoritmo, entonces también lo será cuando se añade un enlace adicional. Cuando el algoritmo pare, como la solución nos la da T y este es prometedor, entonces dicha solución será óptima.

- Caso base: El conjunto vacío es prometedor (ya explicado antes).
- Paso inductivo: Suponemos que T es prometedor antes de añadir un nuevo enlace $e = \{u, v\}$. Tenemos que:
  - $B \subset N$, pues el algoritmo termina cuando $B = N$.
  - T es un conjunto prometedor de enlaces por la hipótesis de inducción.
  - e es uno de los enlaces con menor coste que sale de B (ya que es el que el algoritmo quiere añadir a T en este paso).

  Como vemos, se cumplen las condiciones del Lema 4.3, así que podemos concluir que el conjunto $T \cup \{e\}$ es prometedor.

Como hemos dicho antes, esta inducción demuestra que al terminar el algoritmo tenemos una solución óptima. □

Para implementar este algoritmo, es recomendable cambiar la manera de representar el grafo a la clase ***adjgraph***, también mencionada en el capítulo 5 de [1]. Esta contiene una lista con las etiquetas de los vértices (asumimos que están numerados de 1 a n) y una matriz de adyacencias simétrica L. En $L[i, j]$ tendremos el coste del enlace $\{i, j\}$, siendo ∞ en caso de no existir dicho enlace. El pseudocódigo sería así:

```
funcion prim(G = L[1..n, 1..n]: grafo):  conjunto de enlaces
    # G es un grafo ponderado conexo no dirigido

    # Inicialización: comenzamos arbitrariamente por el vértice 1
    T ← ∅ # Contendrá los enlaces de la solución
    desde i ← 2 hasta n hacer
        mas_cercano[i] ← 1
        dist_min[i] ← L[i,1]

    # Bucle voraz
    durante n−1 iteraciones hacer
        minimo ← inf
        desde j ← 2 hasta n hacer
            si 0 ≤ dist_min[j] < minimo entonces
                minimo ← dist_min[j]
                k ← j
        e ← {mas_cercano[k],k}
        T ← T ∪ {e}
        dist_min[k] ← −1 # Metemos k en B
        desde j ← 2 hasta n hacer
            si L[j,k]  < dist_min[j] entonces
                dist_min[j] ← L[j,k]
                mas_cercano[j] ← k
    devolver T
```

Hemos usado dos *arrays*. Para cada vértice $i \in N \setminus B$:

- $mas\_cercano[i]$ nos dice el vértice en B más cercano a i.
- $dist\_min[i]$ nos dice la distancia entre i y $mas\_cercano[i]$.

Y no hemos representado el conjunto B explícitamente, sino que este comienza solo con el vértice 1 e indicamos que un vértice $i \in B$ si $dist\_min[i]$ es -1.

Ver la complejidad del algoritmo es sencillo, ya que el bucle voraz tiene $n - 1$ iteraciones y sus otros bucles internos tardan un tiempo $\Theta(n)$. Por consiguiente, el algoritmo tiene una complejidad de $\Theta(n^2)$. Comparándolo con el algoritmo de Kruskal, vemos que Prim es más eficiente si el grafo es denso, pero menos si el grafo es disperso.

De todos modos, al igual que con Kruskal, usando un *min heap* para almacenar los enlaces del grafo, conseguiríamos mejorar el tiempo hasta $\Theta(a \log n)$.

CAPÍTULO 5
Problema del viajante

Dada la simplicidad de su formulación, puede resultar tentador en un primer intento de resolver el problema calcular todos los posibles tours del viajante y quedarnos con el que tenga menor coste (esto se conoce como *algoritmo de fuerza bruta* o *algoritmo ingenuo*). Pero como vimos en el capítulo 2.2, la realidad es que resulta muy complejo para conjuntos grandes de ciudades y, de hecho, no se han encontrado todavía algoritmos que encuentren la solución óptima en tiempo polinomial. Este problema es realmente famoso y se podría escribir un TFG solo acerca de él, pero aquí vamos a ver dos algoritmos voraces que no solucionan óptimamente el problema, pero que encuentran una aproximación razonable. Recordemos que hemos dicho que íbamos a tratar el TSP simétrico, modelándolo como un grafo ponderado completo no dirigido.

Antes de continuar, vamos a explicar más sobre estas clases de problemas según su complejidad algorítmica. Comencemos con una definición:

Definición 5.1. Supongamos que tenemos dos clases de problemas, P_1 y P_2. Decimos que P_1 se reduce a P_2 en tiempo polinomial si cualquier ejemplo de la clase P_1 se puede transformar en tiempo polinomial en un ejemplo de la clase P_2.

Ahora veamos las clases de problemas que hay:

- ***P***: Un problema pertenece a la clase de complejidad P, si existe un algoritmo que resuelve cualquier ejemplo de este problema en un tiempo polinomial.
- ***NP***: Un problema pertenece a la clase de complejidad NP, si existe un algoritmo que puede verificar cualquier solución de un ejemplo de esta clase en tiempo polinomial.
- ***NP-completo* (o *NPC*)**: Un problema P_1 pertenece a la clase de complejidad *NP-completo* si pertenece a NP y cualquier otro problema de la clase NP se puede reducir a P_1 en tiempo polinomial.
- ***NP-duro* (o *NP-Hard*)**: Un problema P_1 pertenece a la clase de complejidad *NP-duro* si cualquier problema de la clase NP se puede reducir a P_1 en tiempo polinomial. Por tanto, su diferencia con *NP-completo* es que no requiere que el problema pertenezca a NP.

Se conoce que $P \subset NP$ y $NPC \subset NP$, pero uno de los problemas más estudiados de la complejidad computacional, del que todavía no conocemos el resultado, es el de comprobar si $P = NP$ (de hecho, es uno de los *Problemas del Milenio*, cuyo premio por resolverlo es un millón de dólares). La creencia más extendida es que no es así, ya que en ese caso tendríamos $P = NP = NPC$, y del amplio rango de problemas *NP-completos* estudiados hasta la fecha, nunca se ha encontrado una solución en tiempo polinomial para ninguno de ellos. Y dada su definición, si se pudiera encontrar para uno, querría decir que se podrían encontrar para todos los demás, y eso sería realmente sorprendente. En la figura 5.1 mostramos un diagrama de Euler con las distintas clases, en ambos supuestos del problema del milenio previamente mencionado.

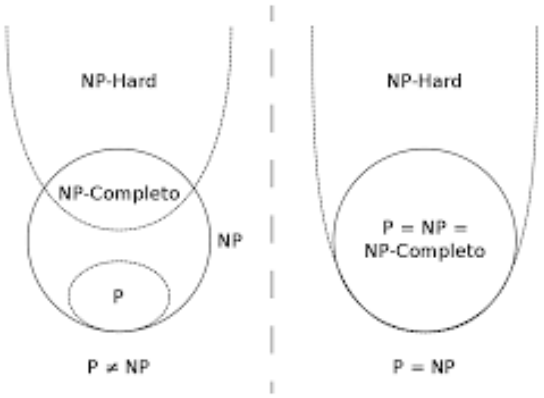

Figura 5.1: Clases de complejidad de problemas

En cuanto a nuestro problema, el TSP, calcular una solución general es difícil así que se considera *NP-duro*, pero también hemos dicho antes que es *NP-completo*. Para demostrar esto, necesitaríamos poder reducir el TSP a otro problema que sepamos es *NP-completo*. Aquí solo vamos a ver esto último, pues nos querremos centrar más en los algoritmos voraces que aproximan la solución. De todos modos, la demostración completa utilizaría otros teoremas y problemas auxiliares, y un ejemplo podría ser:

1. Demostrar el *Teorema de Cook*, es decir, que el *Problema de Satisfacibilidad* (SAT) es *NP-completo*.
2. Demostrar que el *Problema de 3-Satisfacibilidad* ($3\text{-}SAT$) es *NP-completo*, pues se puede reducir $3\text{-}SAT$ a SAT.
3. Demostrar que el *Problema del ciclo hamiltoniano* es *NP-completo*, encontrando una reducción de $3\text{-}SAT$ a este problema.
4. Demostrar que el *TSP* es *NP-completo*, encontrando una reducción del problema del ciclo hamiltoniano al TSP.

Demostremos solo este último paso:

Teorema 5.2. *El problema del viajante (TSP) es NP-completo.*

Demostración. Sea $G =< N, A >$ un grafo cualquiera del problema del ciclo hamiltoniano y sea $n = |N|$. Para el TSP, construimos $G' =< N, A' >$ como la completación del grafo G (es decir, añadiendo enlaces entre los vértices que no tuvieran originalmente ninguno conectándolos directamente), y damos coste 2 a estos enlaces añadidos, y 1 a los enlaces originales de G. Denotando al coste del enlace entre los vértices i y j como $c(i, j)$, se puede formular esto como:

$$c(i,j) = \begin{cases} 1 & \text{si} \quad (i,j) \in A, \\ 2 & \text{si} \quad (i,j) \notin A. \end{cases}$$

Entonces mostramos que G tiene un ciclo hamiltoniano $\Leftrightarrow$ G' tiene un tour de coste como mucho n.

- $\Rightarrow$: Supongamos que G tiene un ciclo hamiltoniano h. Como cada enlace de h pertenece a A, tiene coste 1 en G'. Por tanto, h es un tour de coste n (ya que recordemos que un ciclo hamiltoniano tiene tantos enlaces como número de vértices) en G'.
- $\Leftarrow$: Por otro lado, supongamos que G' tiene un tour h' de coste n como máximo. Como h' está formado por n enlaces y el coste de los enlaces de A' es 1 o 2, el coste total del tour h' debe ser exactamente n, y todos sus enlaces deben tener coste 1. Por tanto, h' solo contiene enlaces en A, y esto nos permite concluir que h' es un ciclo hamiltoniano en el grafo G.

Por tanto, como hemos visto que el problema del ciclo hamiltoniano es *NP-completo* y hemos encontrado una reducción de este al TSP, queda demostrado que el TSP es también *NP-completo.* □

Como ya dijimos, no se ha descubierto un algoritmo que encuentre la solución óptima para el TSP en tiempo polinomial en cualquier caso, pero conocemos ciertos algoritmos que servirán como buenas aproximaciones. A continuación vamos a mostrar dos algoritmos **voraces** cuya solución del TSP tiene, como máximo, el doble del coste de la solución óptima. Para que estos algoritmos funcionen correctamente, requieren añadir una condición más sobre el TSP, y es la **desigualdad triangular**, convirtiéndolo en el *TSP métrico*:

Definición 5.3. Sea $G =< N, A >$ un grafo ponderado no dirigido. Sea $c(i, j)$ el coste del enlace que conecta los vértices i y j. Decimos que cumple la desigualdad triangular si para 3 vértices cualesquiera $i, j, k \in N$,

$$c(i,k) \leq c(i,j) + c(j,k).$$

Esto se podría conseguir en cualquier grafo considerando como nuevo coste de un enlace (i, k) a

$$c^*(i,k) = \text{mín}\{c(P) | \text{P camino de i hasta k}\}.$$

Ahora sí, mostremos los algoritmos.

5.1. Algoritmo del MST

Este algoritmo comienza construyendo un árbol recubridor mínimo en el grafo del problema, tomando como raíz el vértice original desde donde sale el viajante. El coste total de este MST nos dará un límite inferior del coste del camino óptimo del viajante. Después usaremos ese MST para construir un tour cuyo coste no será mayor que el doble del coste total del MST, siempre y cuando el grafo cumpla la desigualdad triangular. Para construir el MST vamos a aprovechar el algoritmo de Prim ya estudiado en el capítulo 4.2. El tour lo construiremos a partir del orden en que visitamos los vértices en un **recorrido preorden** del MST.

El recorrido de árboles se refiere al proceso de visitar de una manera sistemática, exactamente una vez, cada nodo en una estructura de datos de árbol (examinando y/o actualizando los datos en los nodos). Tales recorridos están clasificados por el orden en el cual son visitados los nodos. Hay principalmente dos tipos:

- ***Recorrido en profundidad-primero***: Su funcionamiento consiste en ir visitando desde la raíz todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (*Backtracking*), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado. Es el empleado para la **Búsqueda en profundidad**. Dentro de este, hay tres subtipos, en función del orden en que se visitan los nodos en cada subárbol (por simplicidad, vamos a definirlos para el caso de árboles binarios):
  - *Recorrido preorden*: Visita primero la raíz, luego atraviesa el subárbol izquierdo y finalmente atraviesa el subárbol derecho.
  - *Recorrido inorden*: Atraviesa primero el subárbol izquierdo, luego visita la raíz y finalmente atraviesa el subárbol derecho.
  - *Recorrido postorden*: Atraviesa primero el subárbol izquierdo, luego atraviesa el subárbol derecho y finalmente visita la raíz.
- ***Recorrido en anchura-primero***: Se comienza en la raíz y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol. Es el empleado para la **Búsqueda en anchura**.

Si hablamos de un **recorrido completo** en uno de estos casos, estaríamos apuntando no solo la primera vez que visitamos cada nodo, sino también cada vez que volvemos a él tras visitar uno de sus subárboles.

Dicho esto, vamos a ilustrar un ejemplo del algoritmo en el grafo completo de ejemplo de la figura 5.2.

Comenzamos en el nodo 1 y, usando el algoritmo de Prim, obtenemos el árbol recubridor mínimo de la figura 5.3 (en esta figura y las dos siguientes vamos a omitir el resto de enlaces no afectados para mostrar más claro el resultado). Ahora, haciendo un recorrido preorden del árbol, obtendríamos el siguiente orden de nodos:

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

Añadimos el primer nodo al final también para cerrar el tour, y entonces nos quedaría esta lista de enlaces: {1,3}, {3,6}, {2,6}, {2,7}, {4,7}, {4,5}, {5,8} y {1,8}. Ya tendríamos la solución, cuyo coste total es 24. Lo vemos en la figura 5.5.

Como este grafo es pequeño (solo tiene 8 nodos), podemos calcular la solución óptima con el algoritmo de fuerza bruta, obteniendo un tour con los enlaces {1,3}, {3,5}, {2,5}, {2,7}, {4,7}, {4,8}, {6,8} y {1,6} y coste total 20. La podemos ver en la figura 5.6. Por tanto, el tour óptimo es aproximadamente un 17 % mejor que el nuestro (pues $\frac{20}{24} \approx 0,83$).

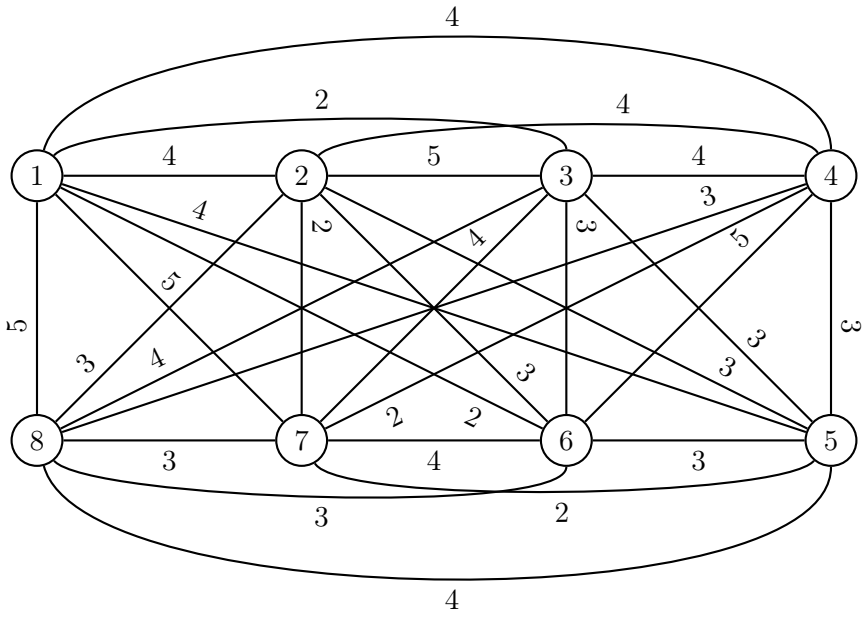

Figura 5.2: Grafo completo de ejemplo para el TSP

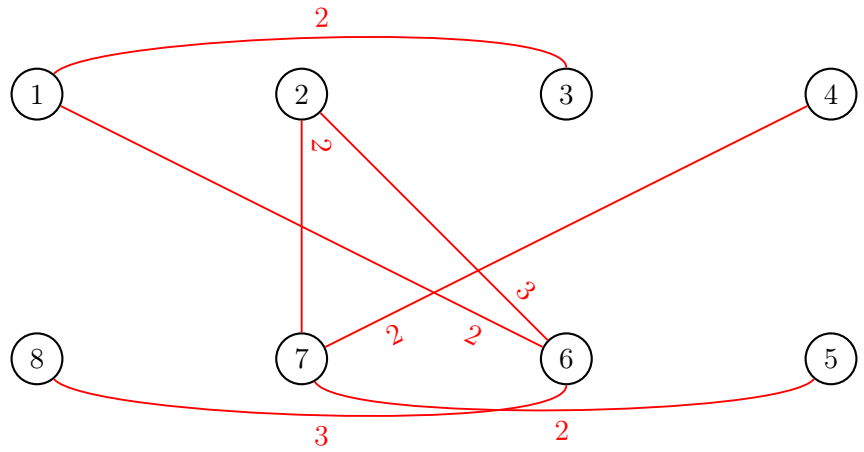

Figura 5.3: Árbol recubridor mínimo calculado con el algoritmo de Prim

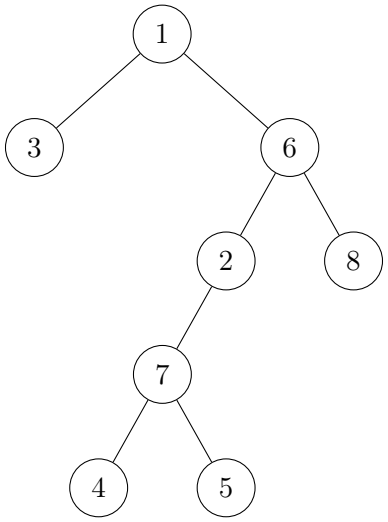

Figura 5.4: Representación más clara del árbol recubridor mínimo anterior

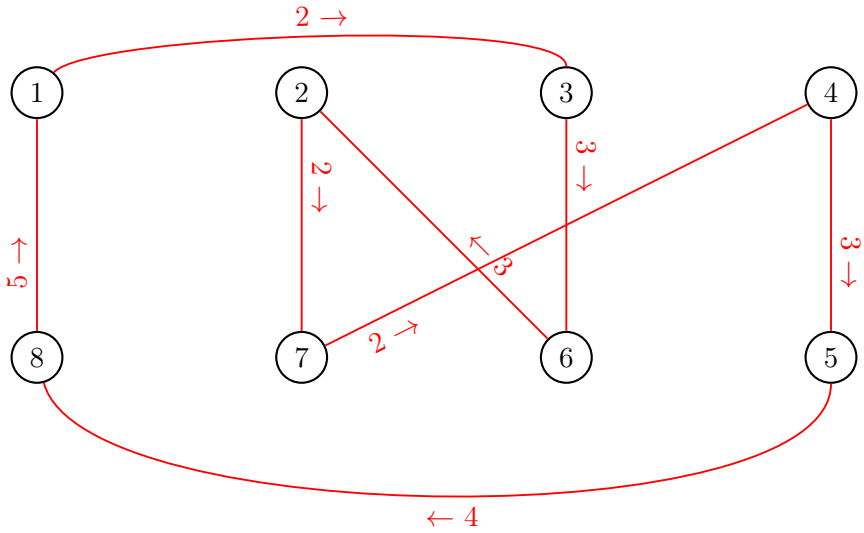

Figura 5.5: Solución obtenida por el algoritmo del MST. Coste total: 24

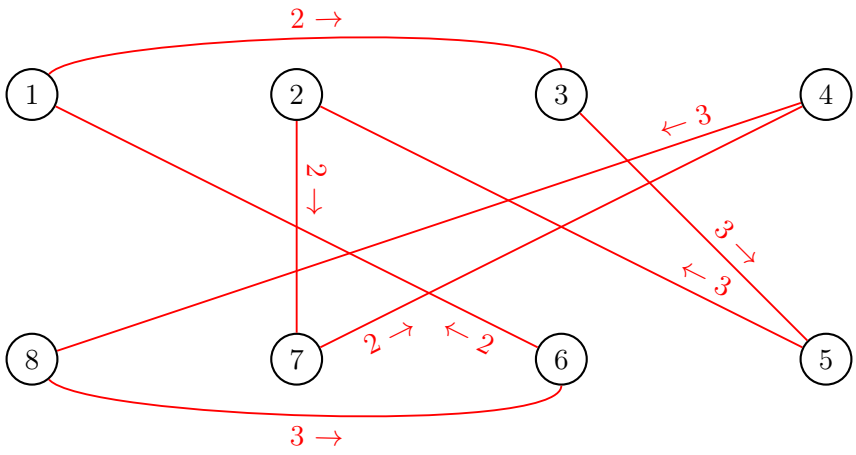

Figura 5.6: Solución óptima del problema del TSP. Coste total: 20

Teorema 5.4. *El algoritmo del MST es una 2-aproximación del TSP métrico.*

Demostración. Como el paso más costoso del algoritmo es la creación del MST, y sabemos que la complejidad de Prim es polinomial, partimos de la base de que el algoritmo del MST es polinomial.

Sea H^* un tour óptimo del TSP para el conjunto de vértices dado. Cada enlace tiene coste no negativo, y obtenemos un árbol recubridor quitando uno de esos enlaces de H^*. Por consiguiente, el coste total del árbol recubridor mínimo T calculado en el algoritmo del MST, proporciona un límite inferior del coste total del tour óptimo:

$$c(T) \leq c(H^*). \tag{5.1}$$

Sea W el recorrido completo de T. Como el recorrido completo pasa por cada enlace de T exactamente 2 veces, tenemos

$$c(W) = 2c(T). \tag{5.2}$$

Combinando (5.1) y (5.2), obtenemos

$$c(W) \leq 2c(H^*), \tag{5.3}$$

y por tanto, el coste total de W es como mucho el doble del coste del tour óptimo.

Desgraciadamente, el recorrido completo W no suele ser un tour, ya que visita algunos vértices más de una vez. Sin embargo, por la desigualdad triangular, podemos eliminar la visita a cualquier enlace de W y el coste no aumenta (si eliminamos de W un vértice v entre las visitas a u y w, el orden resultante diría de ir desde u hasta w directamente, y por la desigualdad triangular, esto es menor o igual). Aplicando este proceso repetidamente, podemos eliminar las segundas visitas a cada vértice de W hasta quedarnos con una lista sin repeticiones.

De hecho, este orden es el mismo que obtendríamos en un recorrido preorden de T. Sea H el ciclo correspondiente a este recorrido preorden (es decir, añadiendo el vértice

inicial al final). Es un ciclo hamiltoniano ya que cada vértice se visita una única vez, y es de hecho el obtenido en nuestro algoritmo del MST. Como H se obtiene eliminando vértices del recorrido completo, tenemos

$$c(H) \leq c(W). \tag{5.4}$$

Combinando las inecuaciones anteriores (5.3) y (5.4), nos queda

$$c(H) \leq 2c(H^*), \tag{5.5}$$

que es justo lo que queríamos demostrar. □

En el ejemplo anterior, el recorrido completo (profundidad-primero) sería

$$1, 3, 1, 6, 2, 7, 4, 7, 5, 7, 2, 6, 8, 6, 1$$

y quedándonos solo con las primeras visitas, tendríamos el recorrido preorden ya mencionado antes:

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

Este podría ser su pseudocódigo comenzando arbitrariamente por el primer nodo, representando el grafo igual que en Prim, como matriz de adyacencias:

```
funcion mst_viajante(G = L[1..n, 1..n]: grafo): conjunto de enlaces
    # G es un grafo ponderado completo no dirigido

    # Inicialización: comenzamos arbitrariamente por el vértice 1
    inicio ← 1

    '''
      Calculamos un MST usando el algoritmo de Prim,
      pero indicando por dónde empezamos
    '''
    T ← prim(G, inicio)

    '''
      Obtenemos la lista de vértices del MST ordenados
      según los hemos ido visitando en un recorrido preorden del MST
    '''
    V ← vertices_preorden(T)

    '''
      Obtenemos los enlaces para recorrer los vértices de V,
      cerrando el tour con el vértice inicial
    '''
    H ← enlaces_tour_viajante(G, V)

    devolver H
```

5.2. Algoritmo del Vecino Más Próximo

El algoritmo del vecino más próximo fue, en las ciencias de la computación, uno de los primeros algoritmos utilizados para determinar una solución para el problema del viajante. Este método genera rápidamente un camino corto, pero generalmente no el ideal. Al igual que antes, vamos a seguir considerando que el grafo que modela el TSP es completo no dirigido.

El tour se construye de la siguiente manera:

1. Empezar en un nodo arbitrario y añadirlo a la lista de visitados.
2. Buscar el nodo más cercano al último visitado, que no esté en la lista de visitados, y añadir el enlace que los conecta a nuestro tour.
3. Repetir estos pasos hasta que todos los nodos hayan sido visitados.
4. Una vez todos los nodos han sido visitados, añadir un enlace desde el último al primero, para cerrar el tour.

Como en otros algoritmos anteriores, en caso de empates en el segundo paso, podemos elegir uno arbitrariamente. Mostremos un ejemplo del algoritmo en acción, con el grafo de la figura 5.2 de nuevo. Comenzamos arbitrariamente en el nodo 1 (en la tabla no mostramos la lista de vecinos, ya que al ser completo el grafo, serán todos los demás nodos).

Paso	Nodo actual	Enlace seleccionado $\{u, v\}$	Conjunto de visitados
Inicialización	-	-	{1}
1	1	{1,3}	{1,3}
2	3	{3,5}	{1,3,5}
3	5	{5,7}	{1,3,5,7}
4	7	{2,7}	{1,2,3,5,7}
5	2	{2,6}	{1,2,3,5,6,7}
6	6	{6,8}	{1,2,3,5,6,7,8}
7	8	{4,8}	{1,2,3,4,5,6,7,8}

Una vez hemos visitado todos los nodos, cerramos el tour añadiendo el enlace {1,4}. Por tanto, nos quedaría una solución formada por {1,3}, {3,5}, {5,7}, {2,7}, {2,6}, {6,8}, {4,8} y {1,4}, con coste total 22. Como ya habíamos visto antes, la solución óptima de la figura 5.6 tiene coste total 20, y entonces volvemos a tener una solución peor que la óptima, esta vez en aproximadamente un 9 %.

La complejidad del algoritmo al programarla es del orden de n^2, siendo n el número de nodos, pero puede ser lineal en el tamaño de entrada si la entrada del algoritmo es una lista con todos los costes entre nodos.

Vamos a comparar la solución de este algoritmo con la solución óptima, pero primero necesitamos un lema. Necesitaremos que el grafo vuelva a cumplir la desigualdad

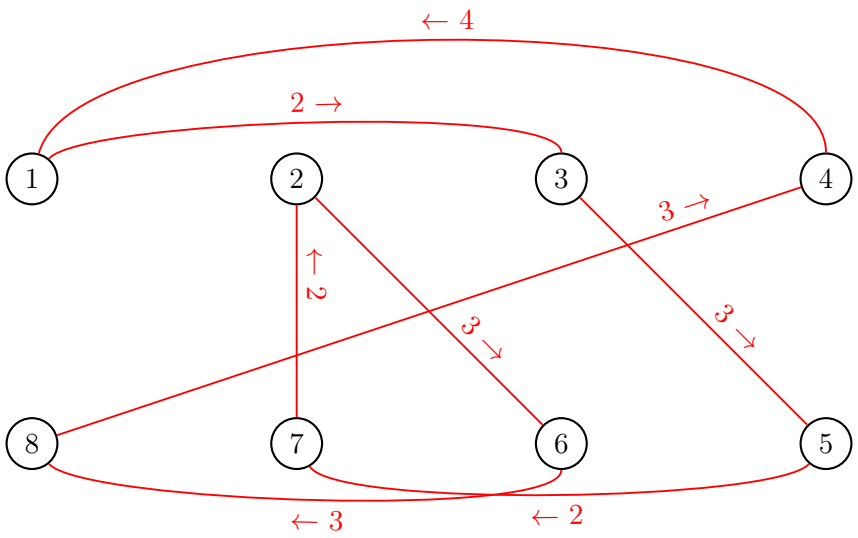

Figura 5.7: Solución obtenida por el algoritmo del vecino más próximo. Coste total: 22

triangular ya mencionada antes. Denotemos OPT al coste de la solución óptima y VMP al coste de la solución del algoritmo del vecino más próximo. Vamos a asumir también la convención de que OPT no puede ser trivial, es decir:

$$OPT > 0. \tag{5.6}$$

Enunciemos un lema:

Lema 5.5. *Supongamos que para un grafo $G =< N, A >$ con $n = |N|$ existe una función l que a cada nodo u le asigna un número $l(u) = l_u$, tal que cumple las siguientes condiciones:*

a) $c(u, v) \geq \min(l_u, l_v)$ *para cualesquiera nodos u y v.*

b) $l_u \leq \frac{1}{2} OPT$ *para cualquier nodo u.*

Entonces, $\sum l_u \leq \frac{1}{2}(\lceil \log_2 n \rceil + 1) OPT$.

Demostración. Podemos asumir sin pérdida de generalidad que el conjunto de nodos N es $\{u | 1 \leq u \leq n\}$ y que $l_u \geq l_v$ siempre que $u \leq v$. La clave de la prueba es la siguiente desigualdad:

$$OPT \geq 2 \sum_{u=k+1}^{\min(2k,n)} l_u, \tag{5.7}$$

para todo k que cumpla $1 \leq k \leq n$.

Para demostrar (5.7), llamemos G^* al subgrafo completo definido sobre el conjunto de nodos

$$\{u | 1 \leq u \leq \min(2k, n)\}.$$

Sea T el tour en G^* que visita los nodos de H en el mismo orden que se visitan en el tour óptimo del grafo original. Sea $COSTE$ el coste total de T. Por la desigualdad triangular, cada enlace $\{u, v\}$ de T debe tener un coste menor o igual que el coste del camino de u a v en el tour óptimo. Como los enlaces de T tienen un coste total de $COSTE$ y sus correspondientes caminos en el grafo original tienen un coste total de OPT, podemos afirmar que

$$OPT \geq COSTE. \tag{5.8}$$

Por la condición *a)* del lema, para cada enlace $\{u, v\}$ en T, $c(u, v) \geq \text{mín}(l_u, l_v)$. Por consiguiente,

$$COSTE \geq \sum_{\{u,v\} \in T} \text{mín}(l_u, l_v) = \sum_{u \in G^*} \alpha_u l_u \tag{5.9}$$

donde α_u es el número de enlaces $\{u, v\}$ en T en los que $u > v$ (y por tanto, $l_u = \text{mín}(l_u, l_v)$).

Queremos obtener una cota inferior en el lado derecho de (5.9). Obsérvese que α_u es como máximo 2 (porque u es el extremo de solo 2 enlaces en el tour T) y que los α_u suman el número de enlaces en T. Debido a que k es al menos la mitad del número de enlaces de T, podemos conseguir una cota inferior del lado derecho de (5.9) si asumimos que los k l_u más grandes tienen $\alpha_u = 0$ y los restantes $\text{mín}(2k, n) - k$ l_u tienen $\alpha_u = 2$. Por suposición, los k más grandes son $\{l_u | 1 \leq u \leq k\}$, así que la cota inferior estimada es

$$\sum_{u \in G^*} \alpha_u l_u \geq 2 \sum_{u=k+1}^{\text{mín}(2k,n)} l_u \tag{5.10}$$

y (5.8), (5.9) y (5.10) juntas, implican (5.7).

Ahora, sumemos las desigualdades de (5.7) para todos los valores de k iguales a una potencia de 2 menores que n. Esto es:

$$\sum_{v=0}^{\lceil \log_2 n \rceil - 1} OPT \geq \sum_{v=0}^{\lceil \log_2 n \rceil - 1} 2 \cdot \sum_{u=2^v+1}^{\text{mín}(2^{v+1}, n)} l_u,$$

lo que se reduce a

$$\lceil \log_2 n \rceil \cdot OPT \geq 2 \cdot \sum_{u=2}^{n} l_u. \tag{5.11}$$

Ahora bien, la condición *b)* del lema implica que

$$OPT \geq 2 \cdot l_1 \tag{5.12}$$

y (5.11) y (5.12) juntas concluyen la demostración del lema. □

Una vez probado este lema, veamos el teorema que nos interesa:

Teorema 5.6. *Para un problema del viajante en un grafo con n nodos,*

$$\frac{VMP}{OPT} \leq \frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}.$$

Demostración. Para cada nodo u, sea l_u el coste del enlace que conecta u con el nodo seleccionado como su vecino más próximo. Queremos probar que l_u satisface las condiciones del lema 5.5.

Si el nodo u fue seleccionado por el algoritmo del vecino más próximo antes que el nodo v, entonces v era un candidato para el nodo no seleccionado más cercano a u. Esto significa que el enlace $\{u, v\}$ no es más corto que el enlace seleccionado y por tanto

$$c(u, v) \geq l_u. \tag{5.13}$$

De manera inversa, si v fue seleccionado antes que u, entonces

$$c(u, v) \geq l_v. \tag{5.14}$$

Como uno de los nodos fue seleccionado antes que el otro, o (5.13) o (5.14) debe cumplirse y, por tanto, la condición *a)* del lema se satisface.

Para demostrar la condición *b)* es suficiente probar que para cualquier enlace $\{u, v\}$,

$$c(u, v) \leq \frac{1}{2} \cdot OPT. \tag{5.15}$$

El tour óptimo puede considerarse formado por dos partes disjuntas, siendo cada una un camino entre los nodos u y v. Por la desigualdad triangular, el coste de cualquier camino entre u y v no puede ser estrictamente menor que $c(u, v)$, implicando entonces (5.15).

Como l_u son los costes de los pares que componen el tour T, tenemos

$$\sum l_u = VMP. \tag{5.16}$$

La conclusión del lema 5.5 junto con (5.15) y (5.6), implican la desigualdad del teorema. □

Teorema 5.7. *El algoritmo del vecino más próximo es una $(\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2})$-aproximación del TSP métrico. Para una formulación más simple, podemos relajarlo un poco hasta una $(\log n)$-aproximación.*

Demostración. Por el teorema 5.6 sabemos que

$$\frac{VMP}{OPT} \leq \frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}.$$

Por (5.6), podemos pasar OPT multiplicando al otro lado y tendríamos

$$VMP \leq (\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}) \cdot OPT.$$

Por definición de *VMP* y *OPT*, tenemos que si H es el tour obtenido por el algoritmo del vecino más próximo y H^* es el tour óptimo:

$$c(H) \leq (\frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}) \cdot c(H^*),$$

lo que probaría el teorema.

Para simplificar la formulación a cambio de una aproximación ligeramente peor, podemos ver con las gráficas de $f(x) = \frac{1}{2}\lceil \log_2 n \rceil + \frac{1}{2}$ y $g(x) = \log x$, que para $x > 34$, tenemos que $g(x) < f(x)$, así que en esos casos tendríamos

$$c(H) \leq (\log n) \cdot c(H^*).$$

□

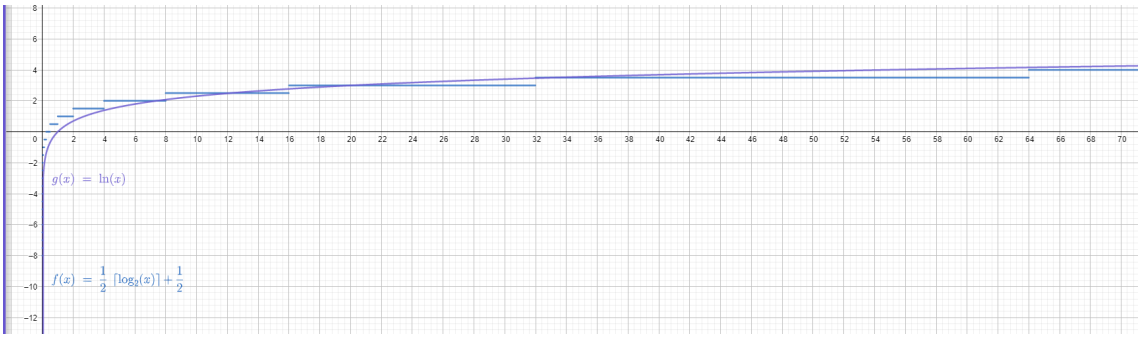

Figura 5.8: Gráficas de las funciones f(x) y g(x) anteriores, representadas en *GeoGebra*

El pseudocódigo del algoritmo es sencillo. Podría ser así:

```
funcion vecino_proximo_viajante
    (G = <N,A>: grafo;
     distancia: A -> R+;
     comienzo: vértice): (lista de enlaces, distancia total)
    # G es un grafo ponderado completo no dirigido

    # Inicialización
    n ← número de nodos de N
    T ← [ ] # Lista vacía para guardar la solución
    vertice_actual ← comienzo
    # Conjunto con los nodos visitados. Inicializado con comienzo
    visitados ← {comienzo}

    # Bucle voraz
    durante n−1 iteraciones hacer
        '''
          Buscamos el vecino más próximo a vertice_actual
          que no esté en visitados
        '''
        vecino ← seleccionar()
        añadir enlace {vecino, vertice_actual} a T
        añadir vecino a visitados
        vertice_actual ← vecino

    # Cerramos el tour
    añadir enlace {vertice_actual, comienzo} a T
    devolver T
```

CAPÍTULO 6
Conclusiones

Nos preguntamos al comienzo del trabajo en qué casos los algoritmos voraces daban una solución óptima y en cuáles no, viendo entonces cómo de buena era la aproximación obtenida. Haciendo un estudio general de dichos algoritmos, hemos observado que resultan simples y, en ciertas ocasiones, hasta intuitivos (como en el problema del cambio). Pero volviendo a las preguntas, hemos visto que la respuesta pasaba por garantizar que el problema cumplía las propiedades de **elección voraz** y de **subestructura óptima**. Uno de los problemas que las cumple es el del árbol recubridor mínimo en un grafo, el cual hemos estudiado mostrando dos algoritmos voraces: el de Kruskal y el de Prim.

Pero parecía más interesante estudiar los casos donde no obteníamos una solución óptima, así que hemos hablado del problema del viajante. Para él, hemos explicado dos algoritmos voraces (el del MST y el del vecino más próximo) que nos proporcionaban soluciones aproximadas. Debemos recordar que para demostrar el nivel de aproximación de estos algoritmos, hemos restringido el problema al TSP métrico (aquel donde el grafo que modela el problema, cumple la desigualdad triangular), y esto nos podría hacer pensar que para un grafo cualquiera, deberíamos estudiar primero si cumple esta propiedad o no, antes de aplicar estos algoritmos. Con esta incertidumbre, podríamos tener una aproximación peor de la esperada, pero merece la pena mencionar que una de las aplicaciones más comunes de este problema se da con puntos en el mapa y las distancias reales entre ellos. Aquí el peso entre cada nodo es la distancia euclídea (estaríamos en el TSP Euclidiano), que sabemos que cumple dicha desigualdad triangular. Por otro lado, siempre se puede implementar un algoritmo para comprobar si un grafo cumple esta propiedad, aunque su complejidad pueda ser alta para grafos muy densos.

Existe una gran variedad de problemas de optimización o búsqueda de los que no hemos hablado, que pueden resolverse por medio de algoritmos voraces, como pueden ser el problema de la mochila o el del camino más corto dentro de un grafo dirigido. Sin embargo, estos algoritmos no son los únicos que permiten resolver esta clase de problemas. Otros ejemplos con los que guardan relación son los que emplean **programación dinámica**. Por lo general son más complejos y potentes, pero por su uso de la recursión, no disfrutan de la gran ventaja de los algoritmos voraces: su rapidez.

En conclusión, esta es la principal razón del uso tan extendido de los algoritmos voraces, no solo para resolver óptimamente algunos problemas más deprisa, sino en especial para resolver problemas de modo aproximado en un tiempo razonable, en lugar de emplear uno más complejo que tarde un tiempo desorbitado debido a la limitada capacidad de computación que tenemos actualmente. Y no solo para casos extremos, sino porque hoy en día el mundo va a gran velocidad y quiere resultados rápidos, por lo que en determinadas situaciones merece la pena sacrificar precisión a cambio de eficiencia.

Apéndices

Todo el código implementado se encuentra en este repositorio de GitHub.

https://github.com/Jandrov/TFG-Matematicas

Aquí en los apéndices vamos a mostrar parte de él (la referida a los algoritmos tratados durante los capítulos anteriores).

APÉNDICE A

Algoritmos del capítulo 4

```
#
# KRUSKAL
#
'''
    Clase que representa un grafo ponderado NO DIRIGIDO
    como un diccionario con:
    – Las etiquetas de los vértices como claves
    – Una lista con tuplas de (etiqueta, longitud) de los vecinos
      de dicho vértice y con cada longitud de esos enlaces
    Se implementan luego algunos métodos útiles
'''

class lisgraph(object):

    def __init__(self, grafo_dic=None):
        '''
            Inicializa un objeto lisgraph.
            Si no se pasa el diccionario,
            se usará uno vacío.
        '''
        if grafo_dic == None:
            grafo_dic = {}
        self.__grafo_dic = grafo_dic

    def vertices(self):
        ''' Devuelve los vértices del grafo '''
        return list(self.__grafo_dic.keys())

    def enlaces(self):
        ''' Devuelve los enlaces del grafo '''
        return self.__generar_enlaces()

    def vecinos(self, vertice):
```

```
        ''' Devuelve los vecinos de ese vértice '''
        return self.__grafo_dic[vertice]

    def anadir_vertice(self, vertice):
        '''
            Si el vértice "vertice" no está en
            self.__grafo_dic, una clave "vertice" con una
            lista vacía como valor es añadida al diccionario.
            En otro caso, no debe hacerse nada.
        '''
        if vertice not in self.__grafo_dic:
            self.__grafo_dic[vertice] = []

    def anadir_enlace(self, vertice1, vertice2, longitud):
        '''
            Aquí como el grafo es no digirido, el orden de
            los vértices no es relevante.
            Se crea un enlace entre los vértices de longitud
            "longitud", añadiendo dichos vértices si es
            necesario y posible.
        '''
        self.anadir_vertice(vertice1)
        self.anadir_vertice(vertice2)
        self.__grafo_dic[vertice1].append((vertice2, longitud))
        self.__grafo_dic[vertice2].append((vertice1, longitud))

    def __generar_enlaces(self):
        '''
            Un método estático que genera los enlaces del
            grafo. Los enlaces son representados como
            diccionarios con "e" (enlace) y "l" (longitud)
        '''
        enlaces = []
        for vertice in self.__grafo_dic:
            for vecino, longitud in self.__grafo_dic[vertice]:
                if {"e" : {vecino, vertice},
                    "l" : longitud} not in enlaces:
                    enlaces.append({"e" : {vertice, vecino},
                                    "l" : longitud})
        return enlaces

    def __str__(self):
        res = "vértices"
        for k in self.__grafo_dic:
            res += str(k) + " "
        res += "\nenlaces: "
```

```
        for enlace in self.__generar_enlaces():
            res += str(enlace) + " "
        return res

import numpy as np

'''
 Función que ordena la lista de enlaces por longitud ascendente

 Entrada:
     enlaces: list -> lista de enlaces
 Salida:
     list -> Lista de enlaces ordenada
'''
def ordenar_lista_enlaces(enlaces: list) -> list:
    return sorted(enlaces, key=lambda x: x["l"])

'''
 Función que selecciona el enlace más corto,
 lo extrae de la lista de enlaces y lo devuelve

 Entrada:
     enlaces: list -> lista de enlaces ya ordenada
 Salida:
     dict -> enlace (como diccionario) de la primera
             posición de la lista
'''
def seleccionar_kruskal(enlaces: list) -> dict:
    return enlaces.pop(0)

'''
 Función que busca la componente que contiene al
 vértice v y la devuelve

 Entrada:
     bosque: list -> bosque que contiene todas las
                     componentes
     v: str -> vértice a buscar
 Salida:
     set -> componente (en forma de conjunto) donde
            se encuentra el vértice v
'''
def encontrar(bosque: list, v: str) -> set:
    for comp in bosque:
        if v in comp:
            return comp
```

```
'''
 Función que combina dos componentes en un bosque

 Entrada:
     bosque: list  -> bosque que contiene todas las
                      componentes
     comp1: set -> primera componente
     comp2: set -> segunda componente
 Salida:
     Nada
'''
def combinar(bosque: list, comp1: set, comp2: set):
    bosque.remove(comp1)
    bosque.remove(comp2)
    bosque.append(comp1.union(comp2))

'''
 Algoritmo de Kruskal para encontrar el árbol recubridor mínimo
 en un grafo conexo no dirigido

 Entrada:
     grafo: lisgraph  -> grafo donde buscamos el árbol recubridor
                         mínimo
 Salida:
     (list, float)  -> tupla formada por la lista de enlaces del
                       árbol recubridor mínimo y su longitud total
'''
def kruskal(grafo: lisgraph) -> (list, float):
    longitud_total = 0 # Longitud total del MST
    T = [] # Lista de enlaces del MST

    # Inicializamos un conjunto por cada vértice del grafo
    bosque = [{vertice} for vertice in grafo.vertices()]
    n = len(bosque) # Número de vértices del grafo

    # Generamos la lista ordenada de enlaces
    enlaces_ordenados = ordenar_lista_enlaces(grafo.enlaces())

    while len(T) < n-1:
        e_optimo = seleccionar_kruskal(enlaces_ordenados)
        (u, v) = e_optimo["e"]
        ucomp, vcomp = encontrar(bosque, u), encontrar(bosque, v)
        if ucomp != vcomp:
            combinar(bosque, ucomp, vcomp)
            T.append(e_optimo)
```

```
            longitud_total += e_optimo["l"]

    return T, longitud_total


#
# KRUSKAL HEAPS
#
import heapq as hq

'''
 Función que crea un min heap según la longitud
 de los enlaces

 Entrada:
     enlaces: list  -> lista de enlaces
 Salida:
     list  -> heap (como una lista) de enlaces
              (como tuplas (longitud, enlace))
'''
def crear_min_heap(enlaces: list) -> list:
    heap = []
    for enlace in enlaces:
        hq.heappush(heap, (enlace["l"], enlace["e"]))
    return heap

'''
 Función que selecciona el enlace más corto,
 lo extrae del heap y lo devuelve

 Entrada:
     heap: list  -> lista de enlaces con estructura de min heap
 Salida:
     dict  -> enlace de la raíz del heap, en forma de diccionario
'''
def seleccionar_kruskal_heaps(heap: list) -> dict:
    longitud, enlace = hq.heappop(heap)
    return { "e" : enlace, "l" : longitud }

'''
 Algoritmo de Kruskal para encontrar el árbol recubridor mínimo
 en un grafo conexo no dirigido, usando un heap para representar
 la lista de enlaces

 Entrada:
     grafo: lisgraph  -> grafo donde buscamos el árbol recubridor
```

```
                        mínimo
 Salida:
     (list,  float)  -> tupla formada por la lista de enlaces del
                        árbol recubridor mínimo y su longitud total
'''
def kruskal_heaps(grafo: lisgraph) -> (list, float):
    longitud_total = 0 # Longitud total del MST
    T = [] # Lista de enlaces del MST

    # Inicializamos un conjunto por cada vértice del grafo
    bosque = [{vertice} for vertice  in grafo.vertices()]
    n = len(bosque) # Número de vértices del grafo

    # Generamos el min heap de enlaces
    heap = crear_min_heap(grafo.enlaces())

    while len(T) < n-1:
        e_optimo = seleccionar_kruskal_heaps(heap)
        (u, v) = e_optimo["e"]
        ucomp, vcomp = encontrar(bosque, u), encontrar(bosque, v)
        if ucomp != vcomp:
            combinar(bosque, ucomp, vcomp)
            T.append(e_optimo)
            longitud_total += e_optimo["l"]

    return T, longitud_total


#
# PRIM
#
'''
    Clase que representa un grafo ponderado NO DIRIGIDO
    de n vértices como:
    - Una lista con las etiquetas de los vértices (ordenados
      de 1 a n)
    - Una matriz de adyacencias simétrica con la longitud de cada
      enlace e infinito en los enlaces que no existan
    Se implementan luego algunos métodos útiles
'''

class adjgraph(object):

    def __init__(self, vertices=None, matriz=None):
        '''
            Inicializa un objeto adjgraph.
```

```
            Si no se pasan la matriz o los vértices,
            se usarán unos vacíos.
        '''
        if vertices == None:
            vertices = []
        if matriz == None:
            matriz = []
        self.__vertices = vertices
        self.__matriz = matriz

    def vertices(self):
        ''' Devuelve los vértices del grafo '''
        return self.__vertices

    def matriz(self):
        ''' Devuelve la matriz de adyacencias del grafo '''
        return self.__matriz

    def enlaces(self, vertice=None):
        '''
            Devuelve los enlaces del grafo.
            Si se pasa un vértice como argumento,
            se devuelven los enlaces que contienen
            a dicho vértice. Si no, se devuelven
            todos los enlaces del grafo
        '''
        return self.__generar_enlaces(vertice)

    def anadir_vertice(self, vertice):
        '''
            Si el vértice "vertice" no está en
            self.__vertices, se añade a la lista al
            final (debe ser el vértice n+1 tal y como
            hemos definido los vértices en este grafo)
            se añade en la matriz de enlaces con inf,
            y se devuelve True
            En otro caso, devuelve False.
        '''
        if vertice not in self.__vertices:
            num_v = len(self.__vertices)
            if int(vertice) == num_v+1:
                self.__vertices.append(vertice)
                for lista in self.__matriz:
                    lista.append(inf)
                self.__matriz.append([inf]*(num_v+1))
                return True
```

```
        return False

    def anadir_enlace(self, vertice1, vertice2, longitud):
        '''
            Aquí como el grafo es no digirido, el orden de
            los vértices solo es relevante a la hora de
            insertar los nuevos vértices en orden.
            Se crea un enlace entre los vértices de longitud
            "longitud", añadiendo dichos vértices si es
            necesario y posible.
        '''
        indice1, indice2 = int(vertice1), int(vertice2)
        if vertice1 in self.__vertices and vertice2 in self.__vertices:
            i, j = indice1-1, indice2-1
            if self.__matriz[i][j] == inf:
                self.__matriz[i][j] = longitud
                self.__matriz[j][i] = longitud
        elif vertice1 in self.__vertices:
            if self.anadir_vertice(vertice2):
                i, j = indice1-1, indice2-1
                self.__matriz[i][j] = longitud
                self.__matriz[j][i] = longitud
        else:
            if abs(indice1-indice2) == 1:
                if self.anadir_vertice(vertice1):
                    self.anadir_vertice(vertice2)
                elif self.anadir_vertice(vertice2):
                    self.anadir_vertice(vertice1)
                else:
                    return
                i, j = indice1-1, indice2-1
                self.__matriz[i][j] = longitud
                self.__matriz[j][i] = longitud


    def __generar_enlaces(self, vertice=None):
        '''
            Un método estático que genera los enlaces del
            grafo. Si se pasa un vértice como argumento,
            se generan los enlaces que contienen
            a dicho vértice. Si no, se generan todos.
            Los enlaces son representados como
            diccionarios con "e" (enlace) y "l" (longitud)
        '''
        enlaces = []
        n = len(self.__vertices)
```

```
        if vertice is None:
            for i in range(n):
                for j in range(i, n):
                    enlace = self.__crear_enlace(i, j)
                    if enlace is not None:
                        enlaces.append(enlace)

        else:
            j = int(vertice - 1)
            for i in range(n):
                enlace = self.__crear_enlace(i, j)
                if enlace is not None:
                    enlaces.append(enlace)

        return enlaces

    def __crear_enlace(self, fila, columna):
        '''
            Un método estático que crea un enlace
            accediendo a la matriz de adyacencias
            por una fila y columna determinadas
        '''
        if self.__matriz[fila][columna] != inf:
            return {"e" : {fila + 1, columna + 1},
                    "l" : self.__matriz[fila][columna]}
        else:
            return None

    def __str__(self):
        res = "vértices: "
        for k in self.__vertices:
            res += k + " "
        res += "\nenlaces: "
        for enlace in self.__generar_enlaces():
            res += str(enlace) + " "
        return res

'''
Algoritmo de Prim para encontrar el árbol recubridor mínimo
en un grafo conexo no dirigido

Entrada:
    grafo: adjgraph -> grafo donde buscamos el árbol recubridor
                       mínimo
Salida:
    (list, float) -> tupla formada por la lista de enlaces del
```

```
                    árbol recubridor mínimo y su longitud total
'''
def prim(grafo: adjgraph) -> (set, float):
    longitud_total = 0 # Longitud total del MST
    T = [] # Lista de enlaces del MST
    # Asumimos comenzar por el primer vértice
    primero = int(grafo.vertices()[0])
    n = len(grafo.vertices())  # Número de vértices del grafo
    matriz = grafo.matriz() # Matriz de adyacencias del grafo

    # Inicializamos nuestros arrays según el vértice por el que empezamos
    mas_cercano = [primero] * n
    dist_min = [-1]
    for i in range(1, n):
        dist_min.append(matriz[i][primero - 1])

    for _ in range(n-1):
        min = inf
        for j in range(1, n):
            if 0 <= dist_min[j] and dist_min[j] < min:
                min = dist_min[j]
                k = j
        T.append({ "e" : {mas_cercano[k], k + 1},
                   "l" : min })
        longitud_total += min
        dist_min[k] = -1
        for j in range(1, n):
            if matriz[j][k]  < dist_min[j]:
                dist_min[j] = matriz[j][k]
                mas_cercano[j] = k + 1

    return T, longitud_total


#
# PRIM HEAPS LISGRAPH
#
'''
 Algoritmo de Prim para encontrar el árbol recubridor mínimo
 en un grafo conexo no dirigido, usando un heap para representar
 la lista de enlaces sin explorar

 Entrada:
     grafo: lisgraph -> grafo donde buscamos el árbol recubridor
                        mínimo
 Salida:
```

```
        (list,  float)  -> tupla formada por la lista de enlaces del
                           árbol recubridor mínimo y su longitud total
    '''

def prim_heaps_lis(grafo: lisgraph) -> (list, float):
    longitud_total = 0 # Longitud total del MST
    explorados = set() # Conjunto de vértices del MST
    # Asumimos comenzar por el primer vértice
    primero = grafo.vertices()[0]
    T = [] # Lista de enlaces del MST

    # Enlaces no explorados ordenados por longitud (será nuestro heap)
    no_explorados = [(0, primero)]

    while no_explorados:
        longitud, ganador = hq.heappop(no_explorados)
        if explorados:
            (vertice1, vertice2) = tuple(ganador)
            if vertice1 not in explorados:
                nuevo_vertice = vertice1
                T.append({ "e" : ganador,
                           "l" : longitud })
            elif vertice2 not in explorados:
                nuevo_vertice = vertice2
                T.append({ "e" : ganador,
                           "l" : longitud })
            else:
                continue
        else:
            nuevo_vertice = ganador
        explorados.add(nuevo_vertice)
        longitud_total += longitud
        vecinos = grafo.vecinos(nuevo_vertice)
        for vecino, longitud in vecinos:
            if vecino not in explorados:
                hq.heappush(no_explorados, (longitud, {nuevo_vertice, vecino}))
    return T, longitud_total


#
# PRIM HEAPS ADJGRAPH
#
'''
 Algoritmo de Prim para encontrar el árbol recubridor mínimo
 en un grafo conexo no dirigido, usando un heap para representar
 la lista de enlaces sin explorar
```

```
    Entrada:
        grafo: adjgraph -> grafo donde buscamos el árbol recubridor
                           mínimo
    Salida:
        (list, float) -> tupla formada por la lista de enlaces del
                         árbol recubridor mínimo y su longitud total
    '''

def prim_heaps_adj(grafo: adjgraph) -> (list, float):
    longitud_total = 0 # Longitud total del MST
    explorados = set() # Conjunto de vértices del MST
    # Asumimos comenzar por el primer vértice
    primero = grafo.vertices()[0]
    T = [] # Lista de enlaces del MST
    matriz = grafo.matriz() # Matriz de adyacencias del grafo

    # Enlaces no explorados ordenados por longitud (será nuestro heap)
    no_explorados = [(0, primero)]

    while no_explorados:
        longitud, ganador = hq.heappop(no_explorados)
        if explorados:
            (vertice1, vertice2) = tuple(ganador)
            if vertice1 not in explorados:
                nuevo_vertice = vertice1
                T.append({ "e" : ganador,
                           "l" : longitud })
            elif vertice2 not in explorados:
                nuevo_vertice = vertice2
                T.append({ "e" : ganador,
                           "l" : longitud })
            else:
                continue
        else:
            nuevo_vertice = ganador
        explorados.add(nuevo_vertice)
        longitud_total += longitud
        vecino = primero
        for longitud in matriz[int(nuevo_vertice)-1]:
            if longitud < inf and vecino not in explorados:
                hq.heappush(no_explorados, (longitud, {nuevo_vertice, vecino}))
            vecino = str(int(vecino) + 1)
    return T, longitud_total
```

Algoritmo adicional en el notebook de Sage greedyAlgorithmsGraphs.ipynb: ***Dijkstra***

APÉNDICE B
Algoritmos del capítulo 5

```
#
# MST
#
'''
    Clase que representa un grafo ponderado NO DIRIGIDO
    de n vértices como:
    - Una lista con las etiquetas de los vértices (ordenados
      de 1 a n)
    - Una matriz de adyacencias simétrica con la longitud de cada
      enlace e infinito en los enlaces que no existan
    Se implementan luego algunos métodos útiles
'''

class adjgraph(object):
    '''
      Ya está su código antes en este apéndice
    '''
    ...


'''
 Algoritmo de Prim para encontrar el árbol recubridor mínimo
 en un grafo conexo no dirigido, con una raíz determinada

 Entrada:
     grafo: adjgraph -> grafo donde buscamos el árbol recubridor
                        mínimo
     comienzo: str -> nodo desde el que comenzamos el MST
 Salida:
     (list, float) -> tupla formada por la lista de enlaces del
                      árbol recubridor mínimo y su longitud total
'''
def prim(grafo: adjgraph, comienzo: str) -> (set, float):
```

```
    longitud_total = 0 # Longitud total del MST
    T = [] # Lista de enlaces del MST
    primero = int(comienzo) # Vértice en el que comenzamos
    primera_posicion = primero - 1 # Posición del vértice donde comenzamos
    n = len(grafo.vertices())  # Número de vértices del grafo
    matriz = grafo.matriz() # Matriz de adyacencias del grafo

    # Inicializamos nuestros arrays según el vértice por el que empezamos
    mas_cercano = [primero] * n
    dist_min = []
    for i in range(n):
        if i == primera_posicion:
            dist_min.append(-1)
        else:
            dist_min.append(matriz[i][primera_posicion])

    for _ in range(n-1):
        min = inf
        for j in range(n):
            if j == primera_posicion:
                continue
            if 0 <= dist_min[j] and dist_min[j] < min:
                min = dist_min[j]
                k = j
        T.append({ "e" : {mas_cercano[k], k + 1},
                   "l" : min })
        longitud_total += min
        dist_min[k] = -1
        for j in range(n):
            if j == primera_posicion:
                continue
            if matriz[j][k]  < dist_min[j]:
                dist_min[j] = matriz[j][k]
                mas_cercano[j] = k + 1

    return T, longitud_total

'''
Función que devuelve una lista con los vrtices del
árbol recibido, recorrindolo en preorden

Entrada:
    arbol: list -> lista de enlaces del arbol

Salida:
    list -> lista con los vrtices del recorrido preorden
```

```
'''
def vertices_preorden(arbol: list) -> list:
    vertices = [] # Lista de vertices del recorrido preorden

    for enlace in arbol:
        for vertice in enlace["e"]:
            if vertice not in vertices:
                vertices.append(vertice)

    return vertices

'''
 Función que devuelve los enlaces del ciclo que recorre
 los vrtices recibidos en ese mismo orden, y la longitud
 total de dicho ciclo

 Entrada:
     grafo: adjgraph -> grafo donde buscamos el ciclo óptimo
     vertices: list -> lista de vertices en determinado orden

 Salida:
     (list, float) -> tupla formada por la lista de enlaces del
                      ciclo y su longitud total
'''
def enlaces_tour_viajante(grafo: adjgraph, vertices: list) -> (list, float):
    longitud_total = 0 # Longitud total del ciclo
    ciclo = [] # Lista de enlaces del ciclo
    matriz = grafo.matriz() # Matriz de adyacencias del grafo
    n = len(vertices) # Número de vértices
    for pos in range(n-1):
        vertice1, vertice2 = vertices[pos], vertices[pos + 1]
        longitud = matriz[vertice1 - 1][vertice2 - 1]
        ciclo.append({ "e" : {vertice1, vertice2},
                       "l" : longitud })
        longitud_total += longitud

    # Cerramos el ciclo
    longitud = matriz[vertice2 - 1][vertices[0] - 1]
    ciclo.append({ "e": {vertice2, vertices[0]},
                   "l": longitud })
    longitud_total += longitud

    return ciclo, longitud_total

'''
 Algoritmo del MST para encontrar una buena aproximación a la
```

```
    solución del problema del viajante en un grafo completo no dirigido

    Entrada:
        grafo: adjgraph -> grafo donde buscamos el ciclo óptimo
        comienzo: str -> nodo desde el que comenzamos el MST

    Salida:
        (list, float) -> tupla formada por la lista de enlaces del
                         camino encontrado y su longitud total
    '''
def mst_viajante(grafo: adjgraph, comienzo: str) -> (set, float):
    mst, _ = prim(grafo, comienzo)

    vertices = vertices_preorden(mst)

    H, longitud_total = enlaces_tour_viajante(grafo, vertices)

    return H, longitud_total


#
# VECINO MÁS PRÓXIMO
#
import heapq as hq

'''
    Función que genera los enlaces del grafo que contienen
    a dicho "vertice" y no a los que estn en "filtro".

    Entrada:
        matriz: list -> matriz de adyacencias del grafo
        vertice: int -> vrtice del que buscamos los enlaces
        filtro: set -> conjunto de vrtices que no queremos en
                       los enlaces

    Salida:
        list -> heap (como una lista) de enlaces
                (como tuplas (longitud, enlace))
    '''
def generar_enlaces_filtrados(matriz: list, vertice: int, filtro: set) -> list:
    distancias = matriz[vertice - 1]
    heap = []
    for pos in range(len(distancias)):
        vecino = pos + 1
        distancia = distancias[pos]
        if distancia != inf and vecino not in filtro:
```

```
                hq.heappush(heap, (distancia, {vertice,  vecino}))

    return heap

'''
 Función que selecciona el enlace más corto,
 lo extrae del heap y lo devuelve

 Entrada:
     heap: list  -> lista de enlaces con estructura de min heap
 Salida:
     dict -> enlace de la raíz del heap, en forma de diccionario
'''
def seleccionar_viajante(heap: list) -> dict:
    longitud, enlace = hq.heappop(heap)
    return { "e" : enlace, "l" : longitud }

'''
 Algoritmo del vecino más próximo para encontrar una buena aproximación a
 la solución del problema del viajante en un grafo completo no dirigido

 Entrada:
     grafo: adjgraph -> grafo donde buscamos el ciclo óptimo
     comienzo: str -> nodo desde el que comenzamos el ciclo
 Salida:
     (list, float) -> tupla formada por la lista de enlaces del
                      camino encontrado y su longitud total
'''
def vecino_proximo_viajante(grafo: adjgraph, comienzo: str) -> (set, float):
    longitud_total = 0 # Longitud total del camino
    T = [] # Lista de enlaces del camino
    primero = int(comienzo) # Vértice en el que comenzamos
    n = len(grafo.vertices())  # Número de vértices del grafo
    matriz = grafo.matriz() # Matriz de adyacencias del grafo
    vertice_actual = primero # Vértice en el que estamos
    visitados = {primero} # Conjunto de vértices visitados

    # Bucle voraz
    for _ in range(n-1):
        enlaces = generar_enlaces_filtrados(matriz, vertice_actual, visitados)
        enlace_optimo = seleccionar_viajante(enlaces)
        T.append(enlace_optimo)
        longitud_total += enlace_optimo["l"]
        vecino_mas_proximo = (enlace_optimo["e"] - {vertice_actual}).pop()
        visitados.add(vecino_mas_proximo)
        vertice_actual = vecino_mas_proximo
```

```
    # Cerramos el ciclo
    longitud = matriz[vertice_actual - 1][primero - 1]
    T.append({ "e": {vertice_actual, primero},
               "l": longitud })
    longitud_total += longitud

    return T, longitud_total
```

Algoritmos adicionales en el notebook de Sage greedyAlgorithmsTSP.ipynb: ***Algoritmo ingenuo*** *y* ***Comprobar desigualdad triangular***

APÉNDICE C

Algoritmo voraz para resolver el problema de la mochila

En el notebook de Sage greedyAlgorithmsKnapsack.ipynb

Bibliografía

[1] Brassard, G.; Bratley, P.: Fundamentals of algorithmics. *s. Prentice Hall, Inc., Englewood Cliffs, NJ* (1996)

[2] Cormen, T. H.; Leiserson, Ch. E.; Rivest, R. L.; Stein, C: Introduction to algorithms. Third edition. *MIT Press, Cambridge, MA* (2009)

[3] Rosenkrantz, Daniel Stearns, Richard II, Philip.: An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM J. Comput.. 6. 563-581. 10.1137/0206041.* (1977)

[4] Moshe Zadka: The Python heapq Module: Using Heaps and Priority Queues. *Recuperado 15 de septiembre de 2020, de RealPython website: https://realpython.com/python-heapq-module/* (2020)

[5] python-course.eu: Graphs in Python: Graphs as a Python Class. *Recuperado 14 de septiembre de 2020, de PythonCourse website: https://www.python-course.eu/graphs_python.php* (2020)

[6] Nishant-Singh: Traveling Salesman Problem (TSP) Implementation. *Recuperado 17 de noviembre de 2020, de GeeksforGeeks website: https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/* (2020)

[7] jddeep003: Travelling Salesman Problem | Greedy Approach. *Recuperado 17 de noviembre de 2020, de GeeksforGeeks website: https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/* (2020)