5

Algoritmos “Greedy”

Introducción

La idea intuitiva y básica en la que se apoya esta técnica de diseño de algoritmos consiste en seleccionar en cada momento lo mejor de entre un conjunto de candidatos, sin tener en cuenta lo ya hecho, hasta obtener una solución para el problema, y así cuando el algoritmo que resuelve un problema está diseñado de acuerdo a la técnica greedy, se dice que es un algoritmo greedy.

Generalmente estos algoritmos son bastante simples y suelen usarse para resolver problemas de optimización: encontrar el mejor orden para ejecutar un cierto conjunto de tareas en un ordenador, calcular el camino más corto sobre un grafo, determinar la mínima estructura necesaria para asegurar la conectividad de una red, etc.

Respecto a la aplicabilidad de esta técnica, cuando en un cierto problema podemos identificar los cinco siguientes elementos,

1. un conjunto (una lista) de candidatos: las tareas a ejecutar, los nodos del grafo, etc... y el conjunto de los candidatos que ya se han usado,
2. un criterio que nos dice cuando un conjunto particular de candidatos es una solución de nuestro problema, ignorando en principio las cuestiones de optimalidad,
3. una función que indica cuando un conjunto de candidatos es factible, es decir, cuando o no es posible completar el conjunto de tal forma que se obtenga al menos una solución (no necesariamente óptima) para el problema,
4. una función de selección que da en cada etapa el candidato más prometedor de los no usados todavía, y
5. una función objetivo que da el valor asociado a una solución y que es la función que intentamos optimizar,

se dice que dicho problema es resoluble mediante un algoritmo cuyo diseño se basa en la técnica greedy, y el algoritmo resultante se llama entonces algoritmo greedy.

Según esto, para resolver un problema de optimización que se ajuste a las anteriores características, tendremos que buscar un conjunto de candidatos que constituyan una solución que optimice (minimice o maximice, según el caso) el valor de la función objetivo, e identificar el resto de los elementos que antes hemos introducido.

No obstante, hay problemas que sin tener todas las anteriores características, también puede resolverse según un enfoque greedy.

El denominado enfoque greedy corresponde a lo siguiente. Un algoritmo greedy procede etapa por etapa. Inicialmente, el conjunto de candidatos elegido es vacío.

Entonces, en cada etapa, intentamos añadir a este conjunto el mejor entre los candidatos restantes, estando guiada esta elección por la función de selección. Si el conjunto aumentado de los candidatos seleccionados no es factible, eliminamos el candidato que acabamos de incorporar y no lo volvemos a considerar nunca más. Sin embargo, si el conjunto aumentado sigue siendo factible, entonces el candidato recién añadido queda en el conjunto de los candidatos elegidos desde ese momento. Cada vez que aumentamos el conjunto de los candidatos elegidos, vemos si constituye una solución para el problema. Cuando un algoritmo greedy trabaja correctamente, la primera solución encontrada por este método es la óptima.
Esta forma de actuación es siempre la misma y, por tanto, es susceptible de ser descrita en términos algorítmicos, como se describe a continuación

```
FUNCION GREEDY (C: conjunto): conjunto
{C es el conjunto de todos los candidatos}
{S es un conjunto en el que construimos la solución}
Mientras S NO SEA una solución y C ≠ ∅ hacer
        X = elemento de C que maximiza SELEC (X)
        C = C – {X}
        Si (S ∪ {X} es factible Entonces S = S∪ {X}
Si S es una solución entonces return S
En caso contrario return "no hay soluciones"
```

Es fácil ver porque estos algoritmos se llaman greedy: En cualquier etapa, el procedimiento elige el mejor pedazo que puede digerir sin preocuparse por el futuro. Nunca cambia su esquema: cuando un candidato se incluye en la solución, queda ahí para siempre, pero cuando un candidato resulta excluido de la solución, nunca más es siquiera considerado. La función de selección se basa usualmente en el objetivo; incluso pueden ser idénticas. Sin embargo puede haber varias funciones plausibles, de entre las que tendremos que elegir la más apropiada para que el algoritmo trabaje correctamente.
Ilustremos el funcionamiento de estos algoritmos con algunos sencillos ejemplos. Consideremos en primer lugar el conocido problema del Cambio de Monedas, según el cual si el sistema monetario de un país está constituido por monedas de valores m_1, m_2, m_3, …, m_n, el problema que queremos resolver es dar cambio de M unidades monetarias a un cliente usando el menor número posible de monedas, es decir en descomponer cualquier cantidad dada *M* en monedas de ese país utilizando el menor número posible de monedas.
Comencemos notando que es fácil implementar un algoritmo ávido para resolver este problema, que es el que sigue el proceso que usualmente utilizamos en nuestra vida diaria. En efecto, los elementos del problema son,

1. Los candidatos: un conjunto finito de monedas de por ejemplo 1, 5, 10 y 25 unidades. conteniendo al menos una moneda de cada. A partir de ese conjunto, será trivial ir determinando las unidades que ya hayamos usado en la solución que estemos construyendo.
2. Una solución: el valor total del conjunto elegido de monedas debe ser exactamente la cantidad que queremos pagar.
3. Un conjunto factible: el valor total del conjunto elegido no puede exceder la cantidad a pagar.

4. La función de selección: elegir la moneda de mayor valor entre las que queden en el conjunto de candidatos, y
5. La función objetivo: el número de monedas usadas en la solución.

Como el problema reúne las cinco características exigidas, es un típico problema greedy, y por tanto podemos construir un algoritmo greedy para resolverlo, pero tal algoritmo va a depender del sistema monetario utilizado ya que puede comprobarse como con los valores sugeridos para las monedas en el anterior ejemplo, el algoritmo greedy siempre encontrara una solución óptima dado que existe. también se puede demostrar, por otro lado, dando contraejemplos específicos, que el algoritmo greedy no daría una solución óptima en cualquier caso si también existieran monedas de 12 unidades, o si alguno de los tipos iniciales de monedas desapareciera del conjunto inicial.

Concretamente, si tenemos, por ejemplo, una moneda de 100 que queremos cambiar y nuestro sistema dispone de 3 monedas de 25, 1 de 10, 2 de 5 y 25 de 1, entonces la primera solución que alcanza el algoritmo greedy directo es justamente la Solución Óptima: 3 de 25, 1 de 10, 2 de 5 y 5 de 1. Pero si tenemos 10 monedas de 1, 5 de 5, 3 de 10, 3 de 12 y 2 de 25, la solución del algoritmo para una moneda de 100 sería 2 de 25, 3 de 12, 1 de 10 y 4 de 1, en total diez monedas. Evidentemente la solución no es óptima, ya que existe otra mejor que utiliza nueve monedas en vez de diez, que es 2 de 25, 3 de 10 y 4 de 5.

Veamos otro ejemplo. Se trata del Problema del Reparador de Equipos Informáticos (PREI). Un técnico en reparaciones de equipamiento informático debe hacer *n* reparaciones urgentes, sabiendo por experiencia lo que va a tardar en cada una de ellas. Sea el t_i el tiempo que empleará en la reparación i. Como él gana en función de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de sus clientes, es decir que si por E_i notamos el tiempo que tarda en esperar a la reparación el cliente *i*-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n) = \sum_{1..n} E_i$$

Es evidente que este problema, como el anterior, reúne todos los ingredientes para resolverlo con un algoritmo greedy, y por tanto vamos a diseñar un algoritmo greedy que resuelva el problema, pero en este caso además probaremos su corrección.

Observemos en primer lugar que el técnico siempre tardará el mismo tiempo total

$$T = t_1 + t_2 + \ \ldots + t_n$$

en realizar todas las reparaciones, independientemente de la forma en que las ordene. Sin embargo, los tiempos de espera de los clientes sí dependen de esta ordenación. En efecto, si mantiene la ordenación original de las tareas (1, 2, ..., *n*), la expresión de los tiempos de espera de los clientes viene dada por:

$$E_1 = t_1$$

$$E_2 = t_1 + t_2$$

.....

$$E_n = t_1 + t_2 + \ \ldots + t_n$$

Lo que buscamos es una permutación de las reparaciones en donde se minimice la expresión de E(n) que, basándonos en las ecuaciones anteriores, viene dada por:

$$E(n) = \sum_{1..n} E_i = \sum_{1..n} (n - i + 1)t_i$$

Probaremos que la permutación óptima es en la que los avisos de avería se atienden en orden creciente de sus tiempos de reparación. Para ello, notamos $X = (x_1, x_2, \ldots, x_n)$ a una permutación de los elementos (1,2,...,n). Sean $(s_1, s_2, \ldots, s_n)$ sus respectivos tiempos de ejecución, es decir, $(s_1, s_2, \ldots, s_n)$ es una permutación de los tiempos originales $(t_1, t_2, \ldots, t_n)$. Supongamos que no está ordenada en orden creciente de tiempo de reparación, es decir, que existen dos números $x_i < x_j$ tales que $s_i > s_j$. Sea $Y = (y_1, y_2, \ldots, y_n)$ la permutación obtenida a partir de X intercambiando x_i con x_j, es decir, $y_k = x_k$ si $k \neq i$ y $k \neq j$, $y_i = x_j$, $y_j = x_i$.
Si se demuestra que $E(Y) < E(X)$ se habrá probado lo que buscamos, pues mientras más ordenada esté la permutación, menor tiempo de espera habrá. Pero para ello, basta darse cuenta que

$$E(Y) = (n - x_i + 1)\, s_j + (n - x_j + 1)\, s_i + \sum_{k=1..n(k \neq i,\, k \neq j)} (n - k + 1)s_k$$

y que, por tanto,

$$E(X) - E(Y) = (n - x_i + 1)(s_i - s_j) + (n - x_j + 1)(s_j - s_i) = (x_j - x_i)(s_i - s_j) > 0.$$

Con lo que la solución de nuestro problema consiste en atender a las llamadas en orden inverso a su tiempo de reparación, y obrando de esa forma el correspondiente algoritmo siempre proporciona una solución optima
Se comprueba de esta forma que los algoritmos greedy no dan siempre la solución óptima, a menos que concurran circunstancias que así lo garanticen. Sin embargo siempre proporcionan buenas soluciones, porque aunque no obtengan la mejor solución, proporcionan lo que se denominan soluciones óptimas locales. Comentemos la importancia que toman los conceptos de óptimo global y óptimo local, en relación con los algoritmos greedy, porque como se ha dicho explican la naturaleza que tienen algunas de las soluciones que pueden dar estos algoritmos en la práctica.
Acabamos de constatar sobre los anteriores ejemplos que el modo de trabajar de estos algoritmos no garantiza el que siempre se alcance un mínimo global, como sería deseable. Esto es debido a que siempre progresan hacia el mejor valor posible, sin tener en cuenta otras alternativas, lo que les puede conducir a conseguir un óptimo local que no represente la solución (óptimo global) del problema considerado. Recuérdese que la definición de óptimo global y local es la siguiente: Sea $f: R^n \rightarrow R$ y consideremos el problema de minimizar la función f(x) sujetas sus variables a la condición de que $x \in S$. Un punto $x \in S$ se llama solución factible del problema. Si $x^* \in S$ es tal que $f(x) > f(x^*)$ para cada $x \in S$, entonces x^* se llama solución óptima, o solución óptima global, o simplemente óptimo global del problema. Si $x \in S$ y existe un entorno $E(x^*)$ de x^* tal que $f(x) > f(x^*)$ para cada $x \in E(x^*)$, entonces x se llama óptimo local.
Es importante destacar el siguiente resultado que establece las condiciones bajo las que un óptimo local es óptimo global, y cuya demostración omitimos por apartarse mucho de los objetivos de este curso (ver por ejemplo el libro de M.S. Bazaraa y

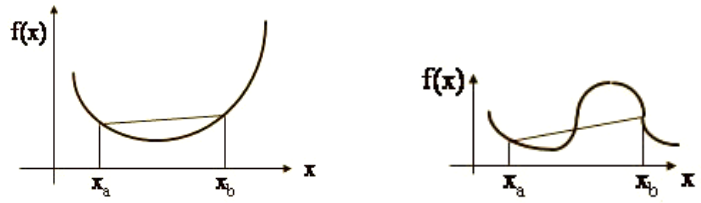
C.M. Shetty, Nonlinear Programming. Theory and Algorithms, John Wiley and Sons, 1979).

Sea $S \subset R^n$ un conjunto convexo no vacío, y f: $R^n \rightarrow R$. Consideremos el problema de minimizar f(x) sujeta a $x \in S$, y supongamos que $x^* \in S$ es un óptimo local del problema. Entonces, si f es una función convexa, x* es un óptimo global del problema.

Para entender un poco más este resultado, recordemos que una función f: $R^n \rightarrow R$ se llama convexa en el intervalo (a,b) cuando se verifica que

$$\forall x_1, x_2 \in (a,b), \forall \lambda \in [0,1] \Rightarrow f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$$

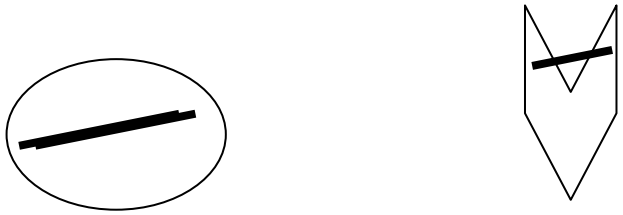
Gráficamente, la función de la izquierda de la siguiente figura sería una función convexa, ya que el segmento lineal que une los dos puntos (x_1 , $f(x_1)$) y (x_2, $f(x_2)$) está completamente dentro de la "sombra" superior de la función, mientras que como es evidente la de la derecha no cumple esa propiedad.



En el mismo sentido, un conjunto S se dice que es un conjunto convexo cuando

$$\forall\ x_1, x_2 \in S \text{ y } \forall\ \lambda \in [0,1] \rightarrow \lambda\ x_1 + (1 - \lambda)\ x_2 \in S$$

lo que se traduce como que dados dos puntos del conjunto S, todo el segmento lineal que los une está en el conjunto. Esto lo refleja la siguiente figura. Como resulta evidente, la de la izquierda se corresponde con un conjunto convexo, mientras que en la de la derecha se ve que no se cumple la propiedad de convexidad,

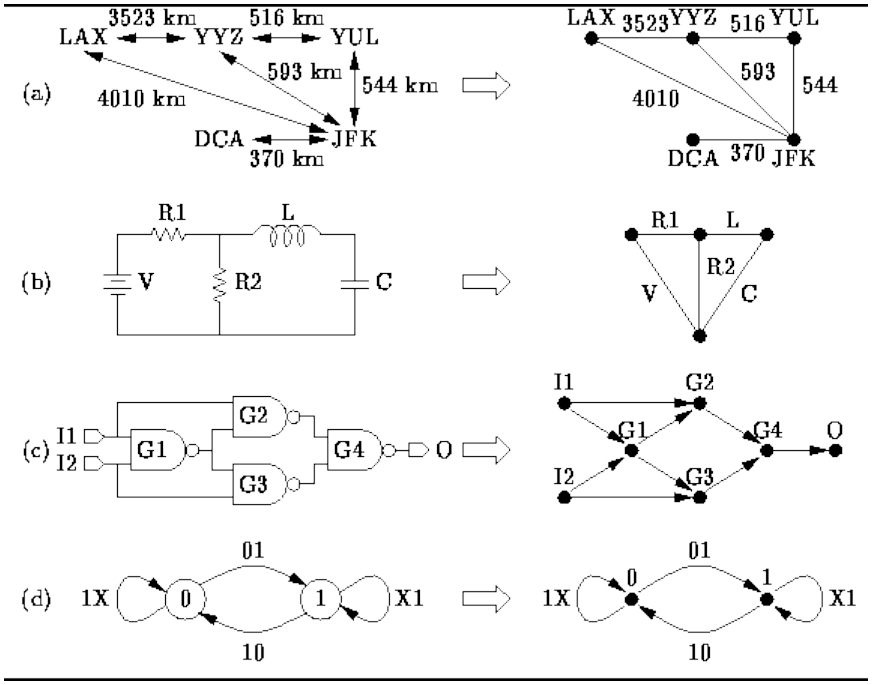


así queda clara la importancia que tiene el concepto de convexidad, y en particular el que la función objetivo sea convexa, para que estos algoritmos consigan alcanzar el óptimo global en algún problema, de forma que cuando se verifiquen las hipótesis del anterior resultado, podremos asegurar que el óptimo que alcance nuestro algoritmo será la solución óptima del problema en cuestión, mientras que si no es así, no tendremos garantías de obtener la solución óptima del problema. Esto no quiere decir, por supuesto, que no se pueda alcanzar ese óptimo global, sino que no habrá garantías de que se pueda hacer, teniendo entonces que buscar condiciones adicionales que nos permitan obtenerlo y, en definitiva, garantizar el buen funcionamiento del algoritmo que estemos considerando.

Podemos concluir por tanto que, salvo que se demuestre la corrección del algoritmo en cuestión, los algoritmos greedy no alcanzan soluciones óptimas siempre. Pero esto, que pudiera parecer una desventaja, se convierte en una ventaja en problemas en los que es imposible (o muy difícil) alcanzar el óptimo, lo que justifica su uso como algoritmos heurísticos. Un poco más adelante incidiremos más detenidamente en este aspecto.

Conceptos básicos sobre grafos

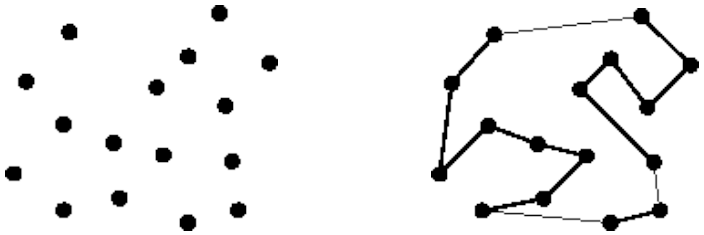
Uno de los campos donde más se aplican este tipo de algoritmos, y en general uno de los dominios más importantes en Teoría de Algoritmos, es el de los Grafos, que intuitivamente se definen con dos conjuntos: un conjunto de vértices (nodos), y un conjunto de aristas. Cuando las aristas tienen origen y final (dirección), se habla de Grafos Dirigidos, y en lugar de aristas tendremos arcos. También hay Grafos Ponderados.
Los grafos sirven para dar fundamento teórico a un gran número de problemas prácticos importantes en Inteligencia Artificial, Robótica, Internet, etc., como por ejemplo se ilustra en la siguiente figura



en la que se muestra el grafo que correspondería a: a) un problema del viajante de comercio, b) un circuito eléctrico, los puntos indican donde se conectan las componentes, que son las aristas, c) un circuito lógico, los nodos son puertas lógicas

y los arcos marcan flujos, y d) una máquina de estado finito, los nodos son los estados y los arcos las transiciones posibles

Las aplicaciones de los grafos son muy diversas, y los problemas que se pueden resolver con ellos, por tanto, múltiples. Uno de los más conocidos, el del camino mínimo, podemos plantearlo como sigue: supongamos que tenemos un robot que maneja un soldador. Para que el robot haga las soldaduras que queremos, debemos darle el orden en que debe visitar los puntos de soldadura, de modo que visite (y suelde) el primer punto, luego el segundo, etc., hasta que concluya su tarea. Gráficamente, nuestro robot tendría que soldar todos los puntos de la siguiente figura (izquierda),



Pero como los robots son caros, necesitamos encontrar el orden en que se realicen las soldaduras, de forma que se minimice el tiempo (es decir, la distancia recorrida) que tarda en realizar todas las soldaduras del panel, es decir, buscamos un algoritmo que nos proporcione el recorrido (mínimo) que se muestra en la anterior figura (derecha)

Por todas estas razones, en lo que sigue vamos a detenernos en los conceptos y definiciones más relevantes dentro de este tema, antes de pasar a describir con precisión algunos de los problemas sobre grafos que son resolubles mediante el enfoque greedy.

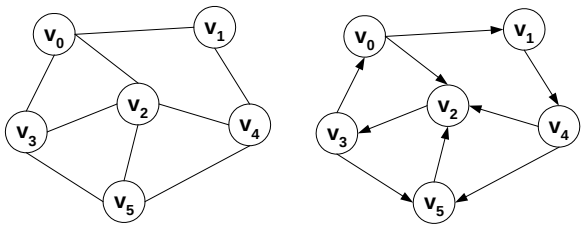
Formalmente, sea $X = \{x_1, x_2, \ldots, x_n\}$ un conjunto finito, no vacío. Sea $A \subseteq X \times X$ una relación. Al par $G = (X,A)$ se le llama grafo dirigido.

Sea $I_X = \{(x_i, x_i), i = 1, 2, ..., n\}$ y $X\_^2 = (X \times X) - I_X$ (para eliminar lazos). Definimos en $X\_^2$ una relación R (para quitar la dirección de los puntos) tal que:

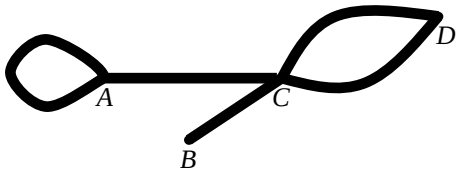
$$(x_i, x_j)\ R\ (x_h, x_k) \Leftrightarrow (x_i, x_j) = (x_h, x_k) \text{ ó } (x_i, x_j) = (x_k, x_h)$$

R es una relación de equivalencia. A continuación definimos el conjunto cociente $(X\_^2 / R)$ para esa relación. Entonces al par $G = (X, B)$, con $B \subseteq X\_^2 / R$ se le llama grafo no dirigido.

Un ejemplo de grafo no dirigido y dirigido se muestra respectivamente en la figura izquierda y derecha siguiente



En un arco (x_i, x_j) a x_i se le llama vértice inicial y a x_j se le llama vértice final. Ambos son vértices terminales. Cuando dos vértices están conectados por una arista o un arco se llaman adyacentes. Del mismo modo, dos aristas son adyacentes si comparten un vértice. Cuando sobre cada par de vértices puede haber más de un arco o de una arista, entonces tenemos un multigrafo
:



y sobre cada vértice o arista, siempre, se podrá definir una función que asocie a cada uno un peso, distancia o valor. En este caso hablaremos de grafo (dirigido o no) ponderado. Es interesante notar que a un grafo G dirigido siempre se le puede asociar un G´ no dirigido quitando la dirección a los arcos.

Un camino sobre un grafo dirigido es una sucesión de arcos $(a_1, ..., a_q)$ tal que el vértice final de un arco coincide con el inicial del siguiente. Una cadena en un grafo no dirigido es una secuencia de aristas adyacentes. Un camino (cadena) simple es un camino (cadena) que no pasa por el mismo arco (arista) más de una vez. Un camino (cadena) elementales un camino (cadena) que no pasa por el mismo vértice más de una vez. Un camino o cadena también se pueden representar por la secuencia de vértices por la que pasa.

Un bucle es un arco cuyos vértices inicial y final coinciden. Un circuito es un camino $(a_1, ..., a_q)$ en el que el vértice inicial de a_1 y el final de a_q coinciden. Un ciclo es una cadena en la que los vértices inicial y final coinciden.

El número de arcos que tienen a un vértice x como vértice inicial se llama grado de salida de x; el número de arcos que tienen a un vértice x como vértice final se llama grado de entrada de x. El grado de un vértice x en un grafo no dirigido es el número de aristas que contienen a x.

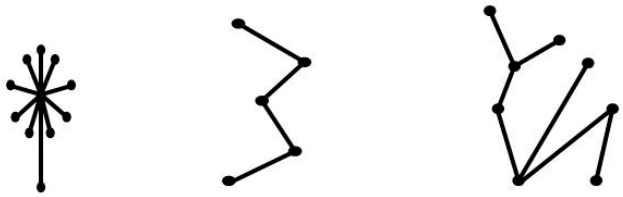
Dado un grafo G = (X,A), un grafo parcial G_P de G es un grafo (X, A_P) con $A_P \subseteq A$, es decir, es un grafo con los mismos vértices pero solo un subconjunto de los arcos (aristas) del grafo original. Dado un grafo G = (X,A), un subgrafo G_S es un grafo (X_S, A_S) con $X_S \subseteq X$ y $A_S \subseteq A$, de modo que cada arco o arista de A_S que esté formado por vértices de X_S tiene que estar en A_S. Dicho de otro modo, un subgrafo tiene solo un subconjunto de los vértices del grafo original, pero contiene todos los arcos o aristas cuyos vértices terminales están en ese subconjunto. Un subgrafo parcial de un grafo G es un grafo parcial de un subgrafo de G (un subconjunto de vértices y un subconjunto de arcos o aristas).

Nos interesará distinguir entre los grafos que usaremos en nuestros algoritmos algunos tipos distinguidos de ellos. Así, un grafo no dirigido es completo si todo par de vértices están unidos por una arista. También, un grafo dirigido es completo si para todo par de vértices hay al menos un arco que los une, es decir, si el grafo no dirigido asociado es completo. Un grafo es regular si todos los vértices tienen el mismo grado. Si el grado común es r, se habla de un grafo regular de grado r. En particular, un grafo es conexo si para todo par de vértices hay al menos una cadena

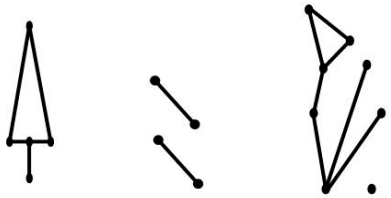
que los une. Un arco o arista tal que si se elimina incrementa el número de componentes conexas del grafo, se llama un puente.
Un grafo no dirigido G = (X,A) se llama bipartido si el conjunto de vértices puede dividirse en dos subconjuntos X_A y X_B disjuntos y exhaustivos, es decir, tales

$$X_A \cap X_B = \varnothing, X_A \cup X_B = X$$

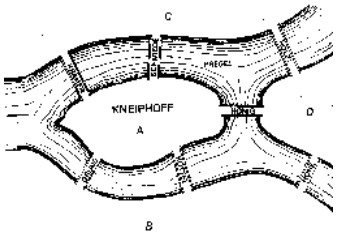
de modo que todas las aristas tengan un vértice terminal en X_A y el otro en X_B.
Particularmente, un grafo dirigido es bipartido si su grafo no dirigido asociado es bipartido.
Un grafo es plano si puede dibujarse en un plano de forma que ninguna de sus aristas interseque a otra.
Un grafo que no tiene ningún ciclo o circuito es un grafo acíclico o bosque. Un árbol es un grafo acíclico y conexo. Ejemplos de árbol son los siguientes grafos,



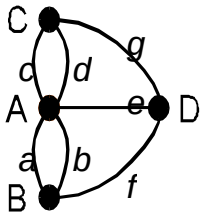
Mientras que los siguientes, claramente, no lo son



Dado un grafo G = (X, A), un subgrafo (X_S ,A_S) es una componente conexa de G si (X_S ,A_S) es un árbol conexo. Intuitivamente se puede decir que una componente conexa es la parte más grande de un grafo que es conexa.
Un multigrafo es euleriano si contiene un circuito (ciclo) que pasa por cada arco (arista) una y solo una vez. El origen de este concepto, y en particular el de Circuito Euleriano se remonta a mediados del siglo XVIII. En efecto, en la ciudad de Königsberg en Austria, hay una isla llamada "Kneiphoff" bordeada por el río Pregel. Como se ilustra en la siguiente figura, hay siete puentes conectando las orillas



El problema que se planteaba era saber si una persona podía recorrer todos estos puentes, pasando por todos y cada uno de ellos solamente una vez, y volviendo a su punto de partida.
Cuando Euler llego a Königsberg, había consenso en la imposibilidad de hacer aquel recorrido, pero nadie lo aseguraba con certeza. Euler planteo el problema como uno de grafos, de modo que cada parte de tierra supondría un vértice, y cada puente representaría una arista,



y en 1736 demostró la imposibilidad de dar un paseo como el que se quería mediante lo que se conoce con el nombre de Teorema de Euler, según el cual si todos los vértices de un grafo son de grado impar, entonces no existen circuitos eulerianos. Además si un grafo es conexo y todos sus vértices son de grado par, existe al menos un circuito euleriano. Concretamente, la determinación de un circuito euleriano mínimo es lo que se conoce con el nombre del Problema del Cartero Chino, que obviamente consiste en encontrar el circuito de longitud mínimo que recorre cada arista de un grafo al menos una vez.
Otro concepto importante es el de Circuito Hamiltoniano, un circuito que pasa a través de cada vértice de un grafo una y solo una vez, y termina en el mismo vértice en el que comenzó. Bajo condiciones muy sencillas, que un poco más adelante detallaremos, la determinación del Circuito Hamiltoniano mínimo de un grafo se conoce con el nombre del Problema del Viajante de Comercio, y esencialmente se plantea en los siguientes términos: Se conocen las distancias entre un cierto número de ciudades. A partir de una de ellas, un viajante debe visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible. Un poco más formalmente, el Problema del Viajante de Comercio puede plantearse como sigue: dado un grafo g conexo y ponderado y dado uno de sus vértices v, encontrar el ciclo Hamiltoniano de coste mínimo que comienza y termina en v.
Por último, para representar un grafo usaremos:
a) La Matriz de Adyacencia: Si suponemos un grafo G = (X, E) con n vértices, entonces su matriz de adyacencia se define por:

$$A_G(i,j) = \begin{cases} 1.....si(x_i, x_j) \in E \\ 0.....si(x_i, x_j) \notin E \end{cases}$$

Como es evidente, A_G es una matriz cuadrada n-dimensional.
b) La Matriz de Incidencia: Si se trata de un grafo no dirigido, entonces la matriz es:

$$B_G(i,j) = \begin{cases} 1.....si...x_i..esta..en..a_j \\ 0.....en..otro..caso \end{cases}$$

donde a_j es la arista que conecta x_i con x_j.
Si se trata de un grafo dirigido, entonces la matriz es:

$$B(i,\ j) = \begin{cases} +1.....si..x_i..es..inicial..en..a_j \\ 0.....en..otro..caso..(bucle) \\ -1.....si..x_i..es..final..en..a_j \end{cases}$$

donde a_j es el arco que conecta x_i con x_j.
En lo que sigue nos detendremos en algunos importantes problemas de grafos, que son resolubles con algoritmos greedy , y que demuestran en definitiva la validez y eficacia de esta técnica de diseño de algoritmos. El lector interesado en el tema de los grafos puede profundizar en las definiciones anteriores, y consultar los resultados anteriormente comentados en cualquier libro de especializado en este tema, pero en particular se recomienda el libro de N. Christofides: Graph Theory. An Algorithmic Approach, Academic Press, Londres, 1975.

El árbol Generador Mínimo de un grafo

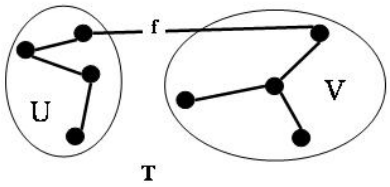
Sea G = (N,A) un grafo conexo no dirigido en el que N es el conjunto de nodos y A el conjunto de aristas. Supongamos que el grafo es ponderado, de forma que cada arista tiene una longitud no negativa. El problema es encontrar un subconjunto T de las aristas de G tal que todos los nodos permanezcan conectados cuando solo se usan las aristas de T, siendo la suma de las aristas de T tan pequeña como sea posible. Como es obvio, en lugar de hablar de longitudes, podríamos hablar de costos. En este caso el problema sería encontrar un subconjunto T cuyo costo total fuera mínimo. Al grafo (N,T) se le llama un Árbol Generador Mínimo (AGM) del grafo G.
Una relación simple de posibles aplicaciones prácticas de este problema revela inmediatamente su importancia, ya que es útil en campos tan diversos como el diseño de redes físicas (telefónicas, eléctricas, hidráulicas, TV por cable, computadores, carreteras, ...), el análisis de clusters (eliminación de aristas largas entre vértices irrelevantes, búsqueda de cúmulos de quásares y estrellas), la solución aproximada de problemas difíciles (entre ellos el Problema del Viajante de Comercio, los de Steiner, etc.), la distribución de mensajes entre agentes y otras aplicaciones indirectas como el plegamiento de proteínas o el reconocimiento de células cancerosas.
El problema reúne todas las características que exige el enfoque greedy ya que

1. Tenemos una lista de aristas, y a partir de ella pueden darse las listas de candidatos o no a solución.
2. Podemos tomar una solución como un conjunto de aristas que forme un árbol generador.
3. La condición de factibilidad es que la arista que se vaya a incluir no forme un ciclo con las ya incluidas.
4. El criterio de selección será escoger en cada momento la arista de mínimo peso.
5. La función objetivo será la definida por la suma de los pesos de las aristas en el árbol, para que sea mínima.

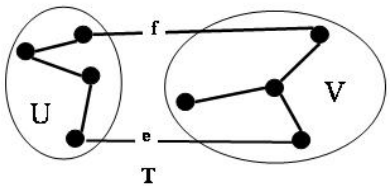
Daremos dos algoritmos para resolver correctamente este problema. Para ello el siguiente resultado, que se conoce con el nombre de Propiedad del AGM será crucial puesto que servirá para garantizar en este caso el correcto funcionamiento de la técnica greedy.

Propiedad del AGM. Sea G = (V,A) un grafo no dirigido y conexo donde cada arista tiene una longitud conocida. Sea U $\subset$ V un subconjunto propio (lo que significa que U no puede coincidir con V) de los nodos de G. Si (u,v) es una arista tal que u $\in$U y v $\in$ V – U, y además es la arista del grafo que verifica esa condición con el menor peso, entonces existe un AGM T que incluye a (u,v).

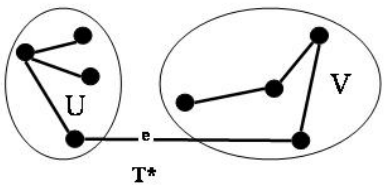
La demostración de esta propiedad se realiza por reducción al absurdo, es decir, partiremos de una hipótesis falsa, para llegar a una contradicción, con lo que aseguraremos la veracidad de la hipótesis que establece el resultado que queremos demostrar. Así, supongamos que T es un AGM. T no contiene a e = (u,v), ya que evidentemente si T contuviera a (u,v) ya estaría demostrada la propiedad que buscamos.



Como T es un AGM, si le añadimos la arista e a T se crea un ciclo C, como muestra la siguiente figura



Si rompemos ese ciclo (eliminando una arista f conectando U y V, obtenemos un nuevo árbol generador T*.



Pero T* por incluir a la arista e tendría menor longitud que T. Eso es una contradicción que demuestra la Propiedad del AGM, como queríamos.

Con esto ya podemos plantear el primero de los algoritmos que sirven para resolver el problema que estamos considerando.

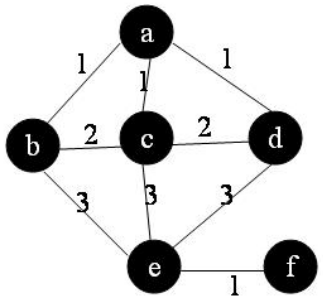
Algoritmo de Kruskal. Siguiendo el enfoque greedy que hemos descrito antes, este algoritmo considera inicialmente que el conjunto T de las aristas está vacío. Conforme el algoritmo progresa, se añaden aristas a T. En cualquier instante, el

grafo parcial formado por los nodos de G y las aristas de T está constituido por varias componentes conexas (al principio, cuando T es vacío, cada nodo de G forma una componente conexa distinta trivial). Los elementos de T que están incluidos en una componente conexa dada forman un árbol generador mínimo para los nodos en esta componente. Al final solo se tiene una componente conexa, de modo que T es entonces un árbol generador mínimo para todos los nodos de G.

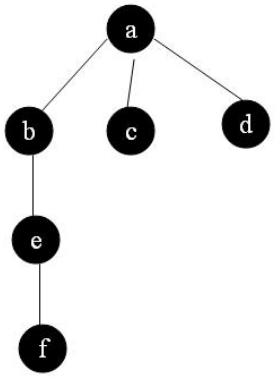
Para construir componentes conexas cada vez mayores, a partir de la Propiedad del AGM, examinamos las aristas de G en orden de longitud creciente. Si una arista une dos nodos en diferentes componentes conexas, la añadimos a T, y consecuentemente, las dos componentes conexas forman ahora una sola componente. En otro caso, la arista se rechaza porque une dos nodos en la misma componente conexa y no puede por tanto ser añadida a T sin evitar la formación de un ciclo, ya que las aristas en T forman un árbol generador mínimo en cada componente. El algoritmo para cuando solo queda una componente conexa.

Es importante destacar que la solución que ofrece este algoritmo, aunque óptima por la Propiedad del AGM, no tiene por qué ser única, ya que como puede haber más de una arista con el mismo peso, y que no formen ciclos al seleccionarla, puede haber más de un AGM asociado a un mismo grafo. De todos modos, nótese que aunque haya más de un AGM, todos deberán tener la misma longitud total.

Para ilustrar como trabaja este algoritmo, consideramos el grafo de la figura,



Si ordenamos sus aristas de forma creciente, tenemos: (a,b), (a,c), (a,d), (e,f), (b,c), (c,d), (b,e), (c,e), (d,e). Ahora, seleccionando teniendo en cuenta que no se formen ciclos cada vez que escogemos una arista, nos queda el siguiente AGM,



Por tanto el Algoritmo de Kruskal, en esencia, actúa del siguiente modo:

1) Ordenar las aristas de forma creciente de costo.
2) Repetir
   Seleccionar la arista más corta.
   Borrar la arista seleccionada de E
   Aceptar la arista si no forma un ciclo en el árbol parcial, y rechazarla en caso contrario,
 Hasta que tengamos $|V|-1$ aristas correctas.

de modo que claramente puede implementarse conforme a lo siguiente

ALGORITMO KRUSKAL (G: GRAFO): Conjunto de aristas.

```
Ordenar las aristas por longitudes crecientes
n = |V|
T = ∅
Inicializar n conjuntos (1 elemento cada uno)
Repetir
        {U,V} = arista más corta aun no considerada
        COMP U = BUSCA (U)
        COMP V = BUSCA (V)
        SI COMPU ≠ COMPV ENTONCES
                UNIR (COMPU, COMPV)
                T = T ∪ ({U,V})
Hasta que |T| = n – 1
DEVOLVER (T)
```

Donde n es el número de vértices en el grafo, T es el conjunto donde construimos el AGM, la función BUSCA lleva a cabo la búsqueda de la arista que se considere y UNIR efectúa la unión de dos conjuntos, y es una función de O (log n). Por tanto en lo que se refiere a la eficiencia del algoritmo, la podemos estimar como sigue. En un grafo con n nodos y a aristas, el número de operaciones es

- O(a log a) para ordenar las a aristas,
- O(n) para inicializar los n conjuntos disjuntos,
- En el peor caso O((2a + n - 1)lg n) para todas las operaciones de búsqueda y unión, ya que a lo más hay 2a operaciones de búsqueda, y n-1 operaciones de unión en un universo que contiene n elementos; y,
- A lo más, O(a) para las restantes operaciones.

Concluimos por tanto que la eficiencia del Algoritmo de Kruskal es O(a log a). Ahora bien, como en cualquier grafo siempre se verifica que

$$n-1 \le a < \frac{n(n-1)}{2}$$

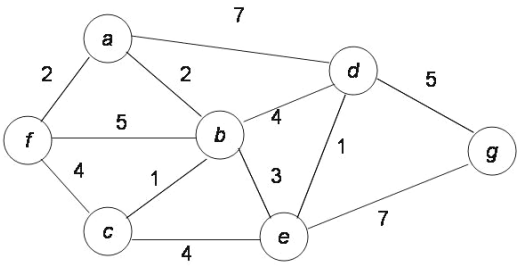
es inmediato obtener que la eficiencia del Algoritmo de Kruskal es O(a log n).

Algoritmo de Prim. En el anterior algoritmo elegíamos las aristas sin tener en cuenta su conexión a aristas previamente elegidas, excepción hecha del cuidado que se pone en no formar un ciclo. Resulta así un bosque de árboles que crece de un modo un tanto fortuito. El algoritmo de Prim se va a construir también en función de la propiedad del AGM, sin embargo el árbol solución va a crecer de manera natural a partir de una raíz arbitraria. En cada etapa, añadiremos una nueva rama al árbol ya construido, y el algoritmo parará cuando se hayan alcanzado todos los nodos.

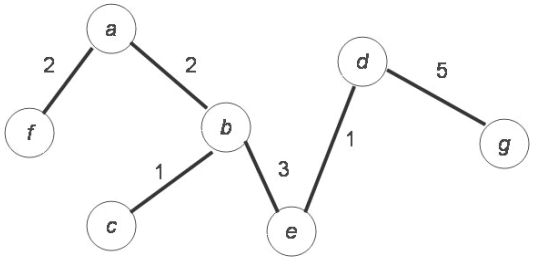
La idea básica del Algoritmo de Prim es la siguiente: Se toma un conjunto U de nodos, que inicialmente contiene al nodo raíz. Formamos el conjunto T de soluciones (aristas). En cada etapa el algoritmo busca la arista más corta que conecta U con V – U, siendo V el conjunto de candidatos, y se añade el vértice obtenido al conjunto U y la arista obtenida a T. En cada instante, las aristas que están en T constituyen un AGM para los nodos que están en U. Esto lo hacemos hasta que U = n. Lo siguiente es un programa informal para este algoritmo.

FUNCION PRIM (G = (V, A)) conjunto de aristas.
(Inicialización)
T = ∅ (Contendrá las aristas del AGM que buscamos).
U = un miembro arbitrario de V
MIENTRAS |U| ≠ N HACER
BUSCAR e = (u,v) de longitud mínima tal que
u ∈ U y v ∈ V – U
T = T + e
U = U + v
DEVOLVER (T)

Aplicando este esqueleto del algoritmo de Prim al siguiente grafo, en el que tomamos como raíz el vértice b,



se obtiene el siguiente AGM,



en el que las aristas se han ido añadiendo en el orden: (b,c), (b,a), (a,f), (b,e), (e,d) y (d,g), con una longitud final de 14.
Para estudiar la eficiencia del algoritmo es necesario elaborar un poco más su implementación. Para ello notamos L [I,J] ≥ 0 una matriz de distancias, MasProximo [x] un vector que nos da el nodo U que está más cercano al vértice x., y DistMin [x] a otro vector que nos da la distancia entre x y MasProximo [x]. Entonces, podemos proponer el siguiente algoritmo,

```
FUNCION PRIM (L[1...n, 1...n]: conjunto de aristas)
{al comienzo solo el nodo 1 se encuentra en U}
        T = ∅ (contendrá las aristas del AGM)
        Para i = 2 hasta n hacer
                MasProximo [i] = 1; DistMin [i] = L[i, 1]
        Repetir n – 1 veces
                min = ∞
                Para j = 2 hasta n hacer
                        Si 0 ≤ DistMin [j] < min entonces min = DistMin [j]
                T = T + (MasProximo [k], k)
                DistMin [k] = -1 (estamos añadiendo k a U)
                para j = 2 hasta n hacer
                        si L [j, k] < DistMin [j] entonces
                                DistMin [k] = L [j, k];
                    MasProximo [j] = k
        Devolver T
```

Respecto a la eficiencia del algoritmo, nótese que el bucle principal del algoritmo se ejecuta n – 1 veces, y que en cada iteración, el bucle "para" anidado requiere un tiempo O(n). Por tanto, el algoritmo de Prim requiere un tiempo O (n^2).
A la vista de estos dos algoritmos, la cuestión que surge es cual de ambos emplear. Vimos que el algoritmo de Kruskal tenía una eficiencia de O(a log n), siendo a el número de aristas en el grafo. Para un grafo muy denso, a tiende hacia n(n-1)/2. Luego en ese caso, el algoritmo de Kruskal consumiría un tiempo O(n log n), por lo que el algoritmo de Prim sería probablemente mejor. Sin embargo para un grafo poco denso, a tiende hacia n. En ese caso, el algoritmo de Kruskal consumiría un tiempo de O(n log n), y el algoritmo de Prim sería probablemente menos eficiente.

Caminos Mínimos

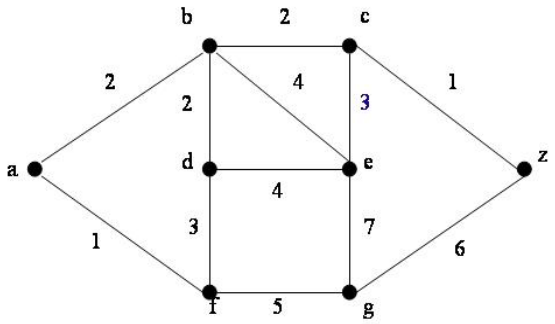
Consideremos ahora un grafo dirigido G = (N, A), en el que como habitualmente N es el conjunto de los nodos y A el de los arcos (aristas dirigidas). Cada arista tiene una longitud no negativa. A uno de los nodos lo designaremos nodo origen. El problema es determinar la longitud del camino más corto desde el origen hasta cada uno de los otros nodos del grafo.
El problema del cálculo del camino mínimo entre dos puntos, o entre varios vértices de un grafo es de una importancia extraordinaria en todas las ramas de las Ingenierías, y la multiplicidad y variedad de sus aplicaciones excusa su listado. El algoritmo que resuelve este problema de manera más eficiente es el conocido Algoritmo de Dijkstra, que diseño el mismo Edsger W. Dijkstra en 1959, cuando tenía 29 años (Dijkstra falleció el 6 de agosto del pasado año 2002). Se trata de un típico problema greedy, en el que la identificación de las cinco características que así lo definen es un asunto trivial.
El algoritmo va a considerar dos conjuntos C y S, que son respectivamente el conjunto de nodos candidatos disponible y el conjunto de nodos ya elegidos. En cualquier momento S contiene aquellos nodos cuya distancia mínima desde el

vértice origen ya se conoce, mientras que C contiene todos los otros. Al comienzo, S solo contiene al origen; cuando el algoritmo termina, S contiene todos los nodos del grafo y nuestro problema queda resuelto. En cada etapa elegimos el nodo en C cuya distancia al origen es menor, y lo añadimos a S.
Decimos que un camino desde el origen a algún otro nodo es "especial" si todos los nodos intermedios en dicho camino pertenecen a S. En cada etapa del algoritmo, un vector D contiene en cada posición la longitud del camino especial más corto al nodo del grafo que corresponde a esa posición. En el momento en que añadimos un nuevo nodo v a S, el camino especial más corto a v es también el más corto de todos los caminos a v. Al final, todos los nodos del grafo están en S, y por tanto todos los caminos desde el origen a cualquier otro nodo son especiales. Consecuentemente, los valores en D dan la solución a nuestro problema.
Si suponemos que los nodos del grafo están numerados de 1 a n, N = {1,2,...,n}, que el nodo 1 es el origen, y que la matriz L da la longitud de cada arco, L[i,j] ≥0 si existe la arista (i,j) y L[i,j] =∞ en otro caso, el algoritmo es el siguiente

FUNCION DIJKSTRA

```
C = {2, 3, ..., N }
Para I = 2 hasta N hacer D[I] = L[1, I]
I = 2, ..., N
P[I] = 1
Repetir N – 2 veces
        V = algún elemento de C que minimice D[V]
        C = C – (V)
        Para cada w ∈ C hacer
                SI D[w] > D[V] + L[v, w] entonces
                D[w] = D[V] + L[v ,w]
                P[w] = v
devolver D
```

donde P es un vector que nos permite conocer por donde pasa cada camino de longitud mínima desde el origen sin más que tener en cuenta que P[v] contiene el nodo que precede a v en el camino más corto. Para encontrar el camino completo, simplemente seguimos al puntero P hacia atrás desde el destino hasta la fuente.
Desarrollemos, como ilustración, el algoritmo sobre el grafo de la siguiente figura,



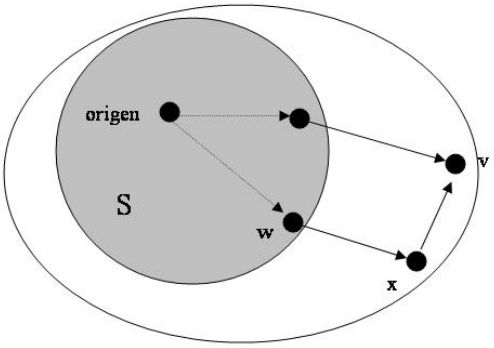
y para no extendernos, desarrollémoslo solo entre los vértices a (origen) y z.

Etapa	v	C	D
Inicialización	a	{b,c,d,e,f,g,z}	{2,100,100,100,1,100,100}
1	f	{b,c,d,e,g,z}	{2, 100, 4, 100, 1, 6, 100}
2	b	{c,d,e,g,z}	{2, 4, 4, 6, 1, 6,,100}
3	d	{c,e,g,z}	{2, 4, 4, 6, 1, 6, 100}
4	c	{e,g,z}	{2, 4, 4, 6, 1, 6, 5}
5	e	{g,z}	{2, 4, 4, 6, 1, 6, 5}
6	g	{z}	{2, 4, 4, 6, 1, 6, 5}

de donde obtenemos que el camino mínimo entre a y z tiene de longitud 6.
La corrección del algoritmo se demuestra a partir de la siguiente hipótesis Inducción T(i):

- En la i-ésima iteración del lazo del algoritmo, el valor D[v] del vértice v se hace permanente
- D[v] siempre es el valor mínimo entre los valores temporales
- El valor D[v] da el camino mínimo desde el origen hasta el vértice v

En efecto, la base (i = 1) de la inducción está clara: Inicialmente D[origen] = 0, y todos los demás valores en D son mayores que cero, por tanto se elige el origen y D[origen] da el valor del camino más corto desde el origen a sí mismo. Consecuentemente T(1) es cierta.
Por otro lado D[v] da la longitud del camino mínimo desde el origen hasta v. Por la hipótesis de inducción D[v] da la longitud del camino especial mínimo. Tenemos por tanto que verificar que el camino mínimo desde el origen hasta v no pasa a través de un nodo que no pertenezca a S. Supongamos lo contrario, de modo que cuando sigamos el camino mínimo desde el origen hasta v, el primer nodo que nos encontremos que no pertenece a S sea algún nodo x distinto de v, como muestra la figura,

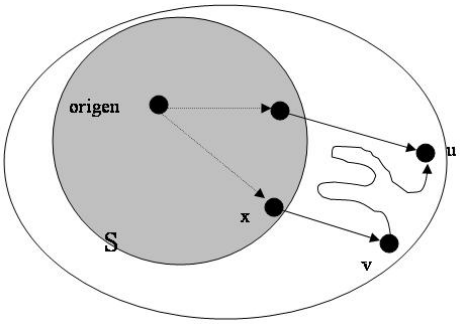


Pero si eso es así, como hemos supuesto que las longitudes son positivas, resulta que D[x] tendría un valor menor que D[v], ya que

$$D[v] = D[x] + L[x,v]$$

lo que iría en contradicción con el hecho de que D[v] fuera el valor mínimo, es decir, con que se hubiera elegido v antes que x.

Finalmente, supongamos que T(k) es cierta para todo k < i. Entonces en la iteración k+1, seleccionamos el nodo v, y lo que tenemos que comprobar es si D(v) es el camino más corto desde el origen hasta v.
Observemos la siguiente figura,



y consideramos un nodo u ∈ S diferente del v. Cuando v se añade a S, hay dos posibilidades para el camino mínimo especial entre el origen y u: o no cambia, o en caso contrario ahora pasa a través de v. En este último caso parece a primera vista que, de nuevo, hay otras dos posibilidades: o v es el último nodo visitado en S antes de llegar a u, o no lo es. Tenemos que comparar explícitamente la longitud del antiguo camino especial que iba a u con la longitud del camino especial que va a v e inmediatamente después a u; el algoritmo hace esto. Pero podemos ignorar la posibilidad de que v sea visitado, para luego ir hasta u, ya que: un camino de este tipo no puede ser más corto que el camino de longitud

$$D[x] + L[x,u]$$

que hemos examinado en la etapa previa cuando x se añadió a S, ya que D[x] < D[v].
Como nuestra hipótesis de inducción era que si se seleccionaba en la iteración k+1 el nodo v, entonces D(v) daba el camino más corto desde el origen hasta v, acabamos de demostrar que esa hipótesis es cierta, y por tanto que T(k+1) es cierta. Como la base y la hipótesis de inducción son ciertas, T(i) es cierta para todo i, y se asegura la corrección del Algoritmo de Dijkstra.
En lo que concierne a la eficiencia, supongamos que el algoritmo se aplica a un grafo que tiene n nodos y a aristas. La inicialización toma un tiempo O(n). En una implementación directa, elegir v en el ciclo repetir requiere examinar todos los elementos de C, de modo que buscamos en n-1, n-2,..., 2 valores de D en las sucesivas iteraciones, dando un tiempo total en $O(n^2)$. El lazo interno para (for) hace n-2, n-1,..., 1 iteraciones, realizando un total también en $O(n^2)$. Por tanto el tiempo requerido para esta versión del algoritmo es $O(n^2)$ en el peor caso, siendo esencial la hipótesis de que las distancias sean no negativas, ya que si hay alguna distancia negativa, el algoritmo no funciona correctamente porque la hipótesis de inducción no puede demostrarse al no verificarse la propiedad triangular.
Recordemos finalmente que esta versión del algoritmo de Dijkstra calcula la distancia entre un vértice y todos los demás, pero que si quisiéramos extenderlo para que calculara las distancias entre todos los pares de vértices (un problema que volveremos a considerar un poco más adelante en el tema dedicado a la Programación Dinámica), lo conseguiríamos muy fácilmente sin más que incorporar

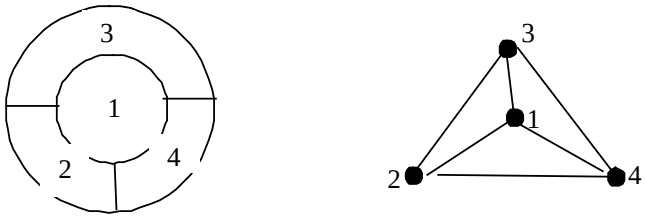
un nuevo lazo externo que recorriera todos los vértices, tomando cada uno de ellos en cada ocasión como origen. En tal caso, la eficiencia del correspondiente algoritmo sería $O(n^3)$.
Como comentamos al principio de este capítulo, dada la facilidad que los algoritmos greedy tienen para ser implementados, así como para alcanzar una solución (aunque sea local), a menudo se usan como heurísticas en situaciones en las que podemos (o debemos) aceptar una solución aproximada en lugar de una solución exacta óptima, lo que suele darse siempre que quien está buscando la solución del problema prefiere soluciones que le satisfagan, antes que soluciones óptimas. Este es el caso del empleo de los algoritmos greedy en problemas clásicos y complejos como son los del coloreo de un grafo, el del viajante de comercio o de la mochila, que a continuación estudiaremos.

El coloreo de un grafo.

Los orígenes de este problema son muy antiguos. Los cartógrafos renacentistas sabían ya que les bastaban cuatro colores para representar sus mapas de manera que dos países vecinos tuvieran distintos colores, logrando así que sus mapas fueran claros y fáciles de entender. Sin embargo, hasta el siglo XIX, a nadie se le había ocurrido que este hecho tuviera que ver con matemáticas y mucho menos que se podía o debía demostrar. Parece ser que el llamado "Problema de los Cuatro Colores" se convirtió formalmente en un problema matemático cuando en 1850 un estudiante inglés, Francis Guthrie, se dio cuenta de que siempre podía representar correctamente los mapas sin usar más de cuatro colores. Intuyendo que esto podía ser demostrado, él y su hermano Frederick se lo plantearon a un prestigioso matemático inglés de la época llamado Augustus De Morgan.
Durante muchos años, matemáticos y no matemáticos, expertos y novatos intentaron resolver el problema de los cuatro colores, es decir, demostrar que bastan cuatro colores para dar una coloración correcta a cualquier mapa. El problema de los cuatro colores se hizo tan famoso en el medio matemático, que en 1878 el matemático inglés Arthur Cayley lo propuso oficialmente a la Sociedad Matemática de Londres, una de las sociedades de matemáticos más importantes del mundo en esa época, como un problema a resolver.
Aunque no se conseguía la demostración, lo que sí se logró con el paso de los años y el trabajo de tanta gente, fue demostrar dos cosas fundamentales: La primera que tres colores son insuficientes para colorear cualquier mapa, es decir, existen mapas que no pueden colorearse de ningún modo usando únicamente tres colores. La segunda que con cinco colores se puede colorear cualquier mapa correctamente. De manera que aunque no se había probado nada respecto a los cuatro colores por lo menos ya se sabía que con tres faltaba y con cinco sobraba, así el número cuatro era el candidato ideal. Había entonces que probarlo o refutarlo.
Finalmente en 1976, 124 años después de proponer el problema, dos matemáticos de la Universidad de Illinois en Estados Unidos, Kenneth Appel y Wolfgang Haken, usando una computadora Cray de segunda generación, consiguieron demostrar lo que se conoce como Teorema de los Cuatro Colores, aunque la forma práctica en que consiguieron hacerlo, suscitó cierta polémica. Por último en 1996, Neil

Robertson, Daniel P. Sanders, Paul Seymour y Robin Thomas, de la Escuela de Matemáticas del Georgia Institute of Technology de Estados Unidos, publicaron una demostración que no tenía los inconvenientes imputados a la demostración de Appel y Haken, cerrando definitivamente el problema.

Si consideremos un "mapa" con 4 regiones de la forma que indica la figura de la izquierda, se observa claramente que se necesitan 4 colores para pintarlo de forma que las regiones contiguas tengan diferentes colores. El problema se traduce a pintar con colores distintos los vértices del grafo de la figura de la derecha.



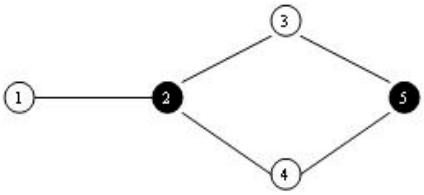
Por tanto, formalmente, el problema consiste en "pintar" los vértices de un grafo G de forma tal que vértices adyacentes tengan distinto color. Si, con este objeto, k colores son suficientes diremos que G es k-colorable

Se llama número cromático, $\chi(G)$, al mínimo número de colores que se necesitan para pintar los vértices del grafo G. Así $\chi(G)$ es el mínimo k tal que G es k-colorable. En el grafo del ejemplo anterior, está claro que $\chi(G) = 4$. Con esto, el Teorema de los Cuatro Colores, lo que establece es que si G es plano, entonces $\chi(G) \leq 4$.

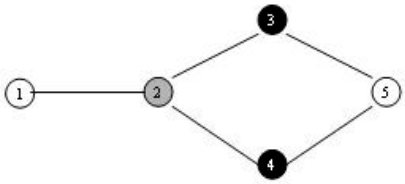
¿Porque podemos estar interesados en diseñar un algoritmo para un problema como este?. Bien, la respuesta puede plantearse desde dos puntos de vista al menos. En primer lugar, para el problema del coloreo, así como para otros que veremos en este mismo capítulo, y para otros muchos, resulta que todos los algoritmos exactos conocidos requieren un tiempo de computación exponencial, es decir, son problemas de los llamados NP-completos (no se conocen algoritmos exactos que funcionen en tiempo polinómico, y por tanto no se pueden conocer sus soluciones óptimas en tiempos razonables), y por tanto para casos de gran tamaño estos algoritmos no pueden usarse en la práctica, con lo que nos vemos obligados a trabajar con métodos aproximados (heurísticas en el caso que aquí estamos considerando).

Por otro lado, e independientemente de responder al desafío intelectual que supuso durante tiempo la Conjetura de los Cuatro Colores, el problema del coloreo de un grafo ha recibido tanta atención por la variedad e importancia de sus aplicaciones, entre las que, aparte de la obvia del coloreo de mapas geográficos o políticos, destacan el diseño de carreteras y cruces de tráfico, el diseño de circuitos, el almacenamiento de productos peligrosos, la asignación de frecuencias de radio o el diseño de páginas webs, junto con la no menos importante de ser un problema para el que no existen algoritmos exactos, es decir, que proporcionen la solución óptima en un tiempo polinómico. De ahí que el uso de heurísticas sea tan importante para encontrar sus soluciones, y por tanto que los algoritmos greedy jueguen un papel tan importante en su solución.

Sea G = (N,A) un grafo no dirigido cuyos nodos tienen que ser coloreados. Queremos usar tan pocos colores como nos sea posible. Es evidente que se trata de un problema que reúne las características necesarias para ser resuelto con un enfoque greedy. Por ejemplo, el grafo de la figura, puede colorearse usando solo dos colores: blanco para los nodos 1, 3 y 4 y negro para los nodos 2 y 5.



Un algoritmo tipo greedy obvio, que podríamos denominar algoritmo secuencial básico para la coloración de un grafo, consiste en elegir un color y un nodo inicial arbitrario, y entonces considerar cada nodo restante por fases, pintándolo con este color si es posible. Cuando no se puedan pintar más nodos, elegimos un nuevo color y un nuevo nodo inicial que no haya sido pintado, pintaremos tantos nodos como podamos con este segundo color, y así sucesivamente.
En nuestro ejemplo, si el nodo 1 se pinta en blanco, no podemos pintar el 2 con el mismo color, los nodos 3 y 4 pueden ser blancos, y por último el 5 no puede pintarse de ese color. Si comenzamos de nuevo con el nodo 2 usando el color negro, podemos colorear los nodos 2 y 5 y finalizar, por tanto, habiendo usado solo dos colores: esta es una solución óptima. Sin embargo, si consideramos los nodos en el orden 1, 5, 2, 3 y 4, obtenemos una respuesta diferente, como se muestra en la figura.



Los nodos 1 y 5 se pintan de blanco, el nodo 2 gris, pero ahora los nodos 3 y 4 requieren que usemos un tercer color: en este caso, el resultado no es óptima. Por tanto, el algoritmo no es más que una heurística que puede encontrar una buena solución, y para la que es fácil dar una versión algorítmica

```
función COLOREO
{COLOREO pone en NuevoColor los vértices de G que pueden tener el
mismo color}
Begin
        NuevoColor = ∅
        Para cada vértice no coloreado v de G Hacer
        Si v no es adyacente a ningún vértice en NuevoColor
        Entonces
            Marcar v como coloreado
            Añadir v a NuevoColor
End
```

Como resulta obvio, se trata de un algoritmo que funciona en O(n), pero que no siempre da la solución óptima. De hecho esta heurística es una de las más pobres que hay para resolver este problema.

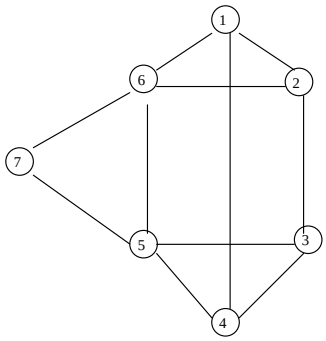
Hay un gran catálogo de algoritmos heurísticos para resolver el Problema del Coloreo de un grafo. Los más simples son coloraciones secuenciales aproximadas. En general, primero se supone que los vértices están ordenados. Entonces el primer vértice se colorea con el color de la clase uno. Los vértices restantes se consideran en orden, y cada uno es coloreado con el color de la primera clase que no es adyacente con vértices que ya tienen asignado el color de dicha clase. Si no hay suficientes clases, se van creando sucesivamente las que sean necesarias.

El orden en que se van seleccionando los vértices es decisivo. Según se haga esa selección, o según estén ordenados los vértices, entre estos métodos destacan:

a) el algoritmo de Welsh y Powell, que ordena los vértices por sus grados de modo descendente, y
b) el algoritmo de Matula, Marble e Isaacson, que lista los vértices de menor grado en orden inverso de forma que un vértice de grado mínimo se coloca en el último lugar de la lista. Se supone que todos los vértices están ordenados así.

Ambos métodos son muy fáciles de implementar y muy rápidos, y normalmente producen una coloración muy cercana a la coloración óptima, de ahí su adecuación para ser empleados como heurísticas.

Para ilustrar su funcionamiento, consideremos el grafo de la siguiente figura:



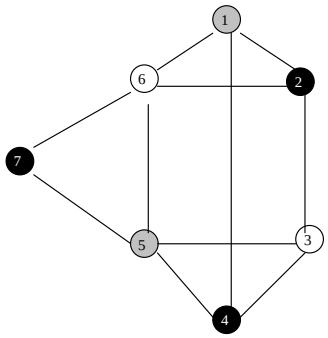
a) método de Welsh Powell (Primero el de Mayor Grado). Para comenzar, el orden de coloración que plantea el algoritmo es primero el de mayor grado, así que tendremos el siguiente orden para los nodos: 6, 5, 1, 2, 3, 4, 7.

El algoritmo se desarrolla conforme a los siguientes pasos: Ponemos en 6 un color, sea este el blanco. Entonces seleccionamos el segundo color, gris, que lo asignamos al siguiente vértice (5). Coloreamos el vértice 1 con el mismo color que el 5, puesto que no son adyacentes.

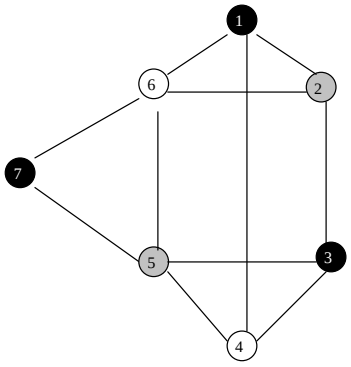
Ahora, para el siguiente vértice (2) tenemos que seleccionar ya un color nuevo. Sea este el negro.

Coloreamos el vértice 3 de color blanco, ya que no es adyacente a ningún vértice de este color, y a continuación el 4 de nuevo de color negro. Finalizamos coloreando el vértice 7 también de color negro, ya que es la primera clase de color que no tiene vértices adyacentes a él.

La solución que obtenemos es la que se muestra a continuación:



b) método de Matula-Marble-Isaacson (El de Menor Grado el Último). Los nodos se ordenan de modo que el de menor grado sea el último, con lo que estos nos quedan ordenados del siguiente modo: -6, 5, 4, 3, 2, 1, 7.
Entonces, coloreamos el vértice 6 con la primera clase de color, blanco. Luego el vértice 5 con otro color, ya que éste vértice es adyacente al anterior, sea este el gris. A continuación el vértice 4 de color blanco, como el 6, ya que no es adyacente a este. El siguiente paso consiste en colorear el vértice 3 de un nuevo color, porque ya es adyacente a vértices coloreados con las dos clases ya usadas, y luego en colorear el vértice 2 de color gris, porque es la primera clase de color que no contiene vértices adyacentes al vértice 2. Se colorea el vértice 1 con la primera clase de color usada no adyacente (negro) y terminamos coloreando el vértice 7 del mismo modo que los anteriores, es decir, negro.
Con esto la solución que se obtiene es la siguiente:



El problema del viajante de comercio.

El Problema del Viajante de Comercio (PVC) se plantea en los siguientes intuitivos términos. Supongamos que conocemos las distancias entre un cierto número de ciudades. Un viajante de comercio que reside en una de esas ciudades, tiene que trazar una ruta que partiendo de su ciudad, visite todas las ciudades, a las que tiene que ir una y sólo una vez, volviendo al origen y realizando un recorrido total mínimo en distancia.

Es un problema NP, es decir, para el que no existen algoritmos en tiempo polinómico que lo resuelvan en un tiempo asumible, aunque si los hay exactos que encuentran una solución para grafos con un máximo de alrededor de 40 vértices, pero ineficaces en grafos de mayor dimensión. Por tanto para más de 40 nodos, es necesario utilizar heurísticas, ya que el problema se hace intratable en el tiempo.

El PVC es uno de los más importantes en Teoría de Algoritmos. Sus orígenes no están muy claros. Aunque en el siglo XIX W.E. Hamilton y T.P. Kirkman consideraron algunos problemas de tipo matemático que se pueden entender como los antecedentes del PVC, parece que fue K. Menger en 1932 el primero que planteó formalmente el PVC.. Desde entonces, probablemente su popularidad se haya debido a lo sencillo de su planteamiento, a lo difícil que es de resolver, y a sus múltiples aplicaciones, que van desde las más intuitivas asociadas a su planteamiento inicial, a las más recientes en el terreno de la Bioinformática, asociadas al reconocimiento de genomas.

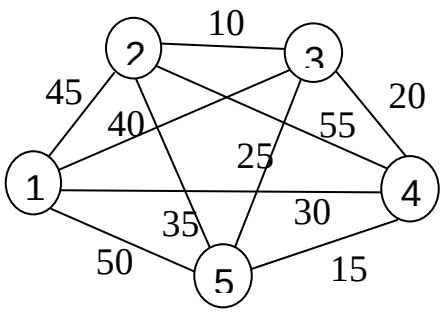
Mas formalmente, supongamos un grafo no dirigido y completo G = (N, A) y que L es una matriz de distancias no negativas referida a los nodos de G. Se quiere encontrar un Circuito Hamiltoniano Mínimo. El grafo también puede ser dirigido si la matriz de distancias no es simétrica. .

Este es un problema Greedy típico, que presenta las 5 condiciones para poder ser enfocado con un algoritmo greedy. En efecto, podemos considerar una lista de candidatos (los nodos del grafo o las correspondientes aristas), una solución (el orden en el que se han de visitar los nodos o en el que hay que recorrer las aristas), una condición de factibilidad: Cada vez que escojamos un candidato para incorporarlo a la solución que estemos formando, se deberá cumplir que

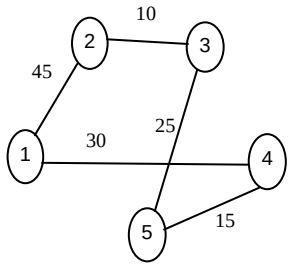
1) no forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completara el recorrido del viajante),
2) no sea la tercera arista elegida incidente a algún nodo.

Además tenemos como función objetivo que la suma de las longitudes de las aristas que constituyan la solución sea mínima, y diferentes funciones de selección, de modo que según cada una de ellas, podremos tener distintas soluciones para un mismo problema. Por ahora, el hecho importante es que como el PVC presenta las cinco características que buscábamos, es resoluble con un enfoque greedy. Otra cosa es que, según la elección que hagamos de la función de selección o de los candidatos que vayamos a usar, podamos obtener distintas soluciones para un mismo problema. En particular, veamos cómo influye la función de selección que elijamos en la solución que buscamos.

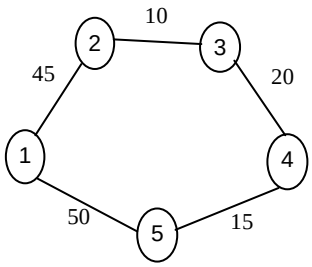
Consideremos por ejemplo el grafo de la siguiente figura,



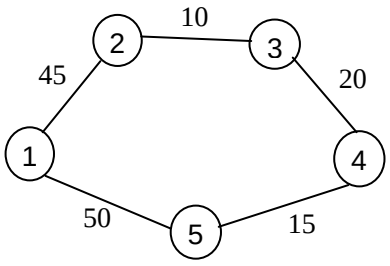
Si los nodos fueran los candidatos, podríamos empezar en un nodo cualquiera y en cada paso movernos al nodo no visitado más próximo al último nodo seleccionado, teniendo en cuenta la condición de factibilidad. En este caso, muy probablemente el nodo que seleccionemos como primer nodo del circuito influya en la solución final. Pero, desde otro punto de vista, también podríamos considerar como candidatas a las aristas. En ese caso podríamos actuar como en el Algoritmo de Kruskal, pero garantizando que a final se forme un ciclo.
Si desarrollamos la primera heurística sobre el anterior grafo, si comenzamos en el nodo 1, se obtiene



es decir, el circuito es el (1, 4, 5 ,3, 2) con una longitud total de 125.
Si ahora empezamos por el nodo 3, el circuito que obtenemos es



es decir (5,4,3, 2,1) con longitud total de 140.
Ahora, si empleamos la segunda heurística, es decir, vamos seleccionando en función de las aristas de menor costo como en el Algoritmo de Kruskal, el circuito que se obtiene es el de la siguiente figura



es decir ((2, 3), (4, 5), (3, 4), (1, 2), (1, 5)), con un costo final de 140.
Queda Claro por tanto como el enfoque greedy sirve para resolver el PVC, dando soluciones que, siendo buenas no hay garantía de que sean óptimas. Por otro lado, es importante destacar que, en todos los casos estos algoritmos heurísticos tendrán una eficiencia igual a la del algoritmo de ordenación de candidatos que nos indique la función de selección que consideremos.

El Problema de la Mochila

Intentemos aplicar el método greedy a la resolución de un problema que, a pesar de su complejidad, o quizás debido a ella, es uno de los bancos de pruebas para todos los nuevos algoritmos heurísticos, y por tanto un referente en el conjunto de las Ciencias de la Computación: El problema de la mochila. Su planteamiento es extraordinariamente intuitivo: Tenemos n objetos y una mochila. El objeto i tiene un peso w_i y la mochila tiene una capacidad M. Si se coloca en la mochila una fracción x_i del objeto i, $0 \leq x_i \leq 1$, se produce un beneficio de $p_i x_i$. El objetivo es obtener una composición de la mochila que maximice el beneficio total. Como la capacidad de la mochila es M, necesitamos que el peso total de todos los objetos que escojamos no supere M.
Formalmente el problema, en el que siempre se supone que los pesos y los beneficios están dados por números positivos, puede establecerse como,

$$\text{Max } \{\sum_{i=1..n} pi \cdot x_i \,/\, \sum_{i=1..n} w_i \cdot x_i \leq M;\ x_i \in 0 \leq x_i \leq 1,\ i = 1...n\}$$

y se denomina Problema de la Mochila Fraccional.
Por ejemplo, si tenemos una mochila de capacidad M = 20, en la que podemos incluir hasta n = 3 objetos, que respectivamente pesan $(w_1, w_2, w_3) = (18,15,10)$ y tienen beneficios de $(p_1, p_2, p_3) = (25, 24,15)$, el correspondiente modelo del Problema de la Mochila se plantearía como,

$$\begin{aligned} \text{Max: } & 25\,x_1 + 24\,x_2 + 15\,x_3 \\ \text{s.a:} & \\ & 18\,x_1 + 15\,x_2 + 10x_3 \leq 20 \\ & x_1,\ x_2,\ x_3 \geq 0 \end{aligned}$$

El Problema de la Mochila en general, y el de tipo fraccional en particular, es un claro exponente greedy. Tenemos una lista obvia de candidatos (los ítems a considerar para incluir en la mochila), un concepto claro de solución (las cantidades a incluir de cada objeto), un criterio de factibilidad (que lo que se incluya en la mochila, no supere la cantidad M), un objetivo (maximizar el beneficio de lo incluido) y una función de selección, que nos indicará la forma en que podemos ir escogiendo los candidatos a ser incluidos. así, una solución factible (una composición de la mochila) es cualquier conjunto $(x_1, x_2, ..., x_n)$ que satisfaga las anteriores restricciones. Una solución óptima es una solución factible para la que la función objetivo alcanza su máximo valor.
En el caso de que la suma de todos los pesos sea menor o igual que la capacidad M, entonces claramente $x_i = 1$, $1 \leq i \leq n$, es una solución óptima. Por tanto, para no considerar casos triviales como este, supondremos en lo que sigue que la suma de los pesos supera a M. Entonces todos los x_i no pueden valer 1.
Por otro lado en el Problema de la Mochila Fraccional, la solución óptima rellenara la mochila exactamente. Esto es cierto porque siempre podemos aumentar en una cantidad fraccional la contribución del último objeto a introducir hasta que el peso total sea exactamente M. Por consiguiente este problema podría plantearse exactamente igual si la restricción de capacidad se formula en términos de igualdad, en lugar de con el símbolo menor o igual que antes hemos escrito.

Igual que en otros casos de problemas greedy, tenemos diferentes posibilidades para diseñar una función de selección. Analicémoslas siguiendo un ejemplo. Supongamos el Problema de la Mochila Fraccional definido por los siguientes datos,

Precio (euros)	20	30	65	40	60
Peso (kilos)	10	20	30	40	50

Junto con una capacidad total de valor M = 100. Pueden sugerirse diversas estrategias greedy simples para obtener soluciones factibles que sumen 100.
Primero podemos intentar rellenar la mochila escogiendo aquel objeto con mayor beneficio. Si un objeto bajo consideración no se ajusta, entonces se incluye una fracción de el para llenar la mochila. así cada vez que se incluye un objeto en la mochila (excepto quizás cuando escojamos el ultimo objeto) obtenemos el mayor incremento posible del beneficio. nótese que si se incluye solo una fracción del último objeto, entonces puede ser posible obtener un mayor beneficio escogiendo otro objeto. Así, con este criterio de selección, la solución que obtendríamos sería la siguiente,

$$\text{Costo Total} = 65 + 60 + 20 = 145$$
$$\text{Peso Total} = 30 + 50 + 20 = 100$$

En la que del último objeto que hemos escogido, solo hemos tomado 20 unidades porque si hubiéramos tomado más, habríamos sobrepasado la capacidad de la mochila. Es muy fácil comprobar que esta solución es sub-óptima. Es evidente que el método usado para obtener esta solución es un auténtico método greedy, pero desde luego no da la solución óptima.
Podemos formular al menos otras dos estrategias greedy para intentar obtener soluciones óptimas. A partir del anterior ejemplo observamos que el considerar objetos en orden decreciente de beneficio no produce soluciones óptimas, ya que aunque la función objetivo efectúa grandes incrementos en cada etapa, el número de etapas es bajo porque la capacidad de la mochila se agota rápidamente. Por tanto intentemos ser greedy en la capacidad y agotarla tan despacio como sea posible. Esto exigirá que consideremos los objetos en orden de peso no decreciente. así, sobre el anterior ejemplo, la solución que se encuentra es la siguiente

$$\text{Costo total} = 20 + 30 + 65 + 40 = 155$$
$$\text{Peso total} = 10 + 20 + 30 + 40 = 100$$

que también es claramente sub-óptima. En esta ocasión aunque la capacidad se agotó despacio, los beneficios no crecieron tan rápidamente.
así nuestro siguiente intento será un algoritmo que equilibre la tasa a la que el beneficio aumenta, con la tasa a la que la capacidad se agota. En cada etapa incluiremos aquel objeto que dé el máximo beneficio por unidad de capacidad usada, lo que significa que los objetos abr que considerarlos en el orden que fije la razón p_i / w_i. De acuerdo con esta estrategia tenemos

Precio (euros)	20	30	65	40	60
Peso (Kilos)	10	20	30	40	50
Precio/Peso	2	1,5	2,1	1	1,2

es decir,

$$\text{Costo Total} = 65 + 20 + 30 + 48 = 163$$
$$\text{Peso Total} = 30 + 10 + 20 + 40 = 100$$

que además es la solución óptima del ejemplo que hemos considerado.
En definitiva, si

a) definimos la densidad del objeto A_i por p_i/w_i
b) se usan objetos de tan alta densidad como sea posible, es decir, los seleccionaremos en orden decreciente de densidad,
c) se coge todo lo que se pueda del objeto i. Si no es posible, se rellena el espacio disponible de la mochila con una fracción del objeto en curso, hasta completar la capacidad, y se desprecia el resto.
d) se ordenan los objetos por densidad no creciente, es decir,

$$p_i/w_i \geq p_{i+1}/w_{i+1} \text{ para } 1 \leq i \leq n.$$

el siguiente algoritmo da las soluciones correspondientes a esta estrategia.

PROCEDIMIENTO MOCHILA_GREEDY (P,W,M,X,n)
//P(1:n) y W(1:n) contienen los costos y pesos respectivos de los n objetos ordenados como P(I)/W(I) > P(I+1)/W(I+1). M es la capacidad de la mochila y X(1:n) es el vector solución//
real P(1:n), W(1:n), X(1:n), M, cr;
integer I,n;

```
        x = 0; //inicializa la solución en cero //
        cr = M;  // cr = capacidad restante de la mochila //
        Para i = 1 hasta n Hacer
                Si W(i) > cr Entonces exit endif
                X(I) = 1;
                cr =  c - W(i);
        Repetir
                Si I < n Entonces X(I) = cr/W(I) endif
End MOCHILA_GREEDY
```

nótese que usando este algoritmo, si los objetos estuvieran en el orden apropiado, podrían obtenerse las soluciones correspondientes a las dos primeras estrategias que, sin considerar el tiempo que se consuma inicialmente ordenando los objetos, solo consumen un tiempo O(n) cada una de ellas.
Hemos visto que cuando se aplica el método greedy al problema de la mochila hay al menos tres medidas diferentes con las que se puede intentar la optimización cuando se quiere determinar que objeto elegir. Dichas medidas son el beneficio total, la capacidad usada y la razón entre el beneficio acumulado y la capacidad usada. Cuando se ha elegido una de esas medidas, el método greedy sugiere elegir los objetos que hay que incluir en la solución de tal forma que cada elección maximice esa medida cada vez. así un método greedy que usa el beneficio como medida, en cada etapa elegirá aquel objeto que le incremente cuanto más el

beneficio. Si lo que usamos es la medida de capacidad, el objeto que elijamos incrementara esta lo menos posible.
Mientras que los métodos greedy basados en las dos primeras medidas no garantizan soluciones óptimas para este problema, vamos a demostrar sin embargo que un algoritmo greedy que use la tercera estrategia siempre obtiene una solución óptima. En efecto, si

$$p_i/w_i \geq p_{i+1}/w_{i+1},\ 1 \leq i \leq n$$

entonces el anterior algoritmo genera una solución óptima para el caso dado del problema de la mochila.
Demostrémoslo. Sea $X = (x_1, x_2, ..., x_n)$ la solución generada por el anterior algoritmo greedy, y sea $Y = (y_1, y_2, ..., y_n)$ una solución factible cualquiera. Vamos a probar entonces que

$$\sum_{i=1}^{n}(x_i - y_i)p_i \geq 0$$

En efecto, si todos los x_i son iguales a 1, entonces claramente la solución que nos dio el algoritmo es la óptima. Por tanto sea k el menor índice tal que $x_k < 1$. Entonces tenemos

$$\sum_{i=1}^{n}(x_i - y_i)p_i = \sum_{i=1}^{k-1}(x_i - y_i)w_i\frac{p_i}{w_i} + (x_k - y_k)w_k\frac{p_k}{w_k} + \sum_{i=k+1}^{n}(x_i - y_i)w_i\frac{p_i}{w_i}$$

Si consideramos cada uno de esos bloques, resulta que

$$\sum_{i=1}^{k-1}(x_i - y_i)w_i\frac{p_i}{w_i} \geq \sum_{i=1}^{k-1}(x_i - y_i)w_i\frac{p_k}{w_k}$$

$$(x_k - y_k)w_k\frac{p_k}{w_k} = (x_k - y_k)w_k\frac{p_k}{w_k}$$

$$\sum_{i=k+1}^{n}(x_i - y_i)w_i\frac{p_i}{w_i} \geq \sum_{i=k+1}^{n}(x_i - y_i)w_i\frac{p_k}{w_k}$$

Luego

$$\sum_{i=1}^{n}(x_i - y_i)p_i \geq \sum_{i=1}^{n}(x_i - y_i)w_i\frac{p_k}{w_k} \qquad \frac{p_k}{w_k}\sum_{i=1}^{n}(x_i - y_i)w_i$$

Evidentemente $\sum_i x_i w_i = M$ por hipótesis, pero $\sum_i y_i w_i \leq M$, luego como siempre

$$W\frac{p_k}{w_k}\sum_{i=1}^{n}(x_i - y_i)w_i$$

es mayor que cero, entonces

$$\sum_{i=1}^{n}(x_i - y_i)p_i \geq 0$$

como queríamos demostrar.