ALGORITMOS "GREEDY"

Algoritmo greedy -> Seleccionar en cada momento lo mejor de entre un conjunto de candidatos, sin tener cuenta lo ya hecho, hasta obtener solución al problema. Procede etapa por etapa y en un principio el conjunto de candidatos elegido está vacío. Si el candidato seleccionado no es factible con el resto, se elimina y no se considera nunca más. Si funciona correctamente, la primera solución es la óptima.

Se aplica a problemas si en estos se identifica:

1. Conjunto de candidatos y el conjunto de candidatos ya usados
2. Un criterio que nos dice cuando un conjunto de candidatos es una solución
3. Una función que indica cuando un conjunto de candidatos es posible o no para obtener una solución (no necesariamente más óptima)
4. Una función de selección que da en cada etapa el candidato más prometedor de los no usados
5. Una función objetivo (que es la que intentamos optimizar) que da una solución

Problemas que no se correspondan que las características anteriores pueden tener soluciones con greedy.

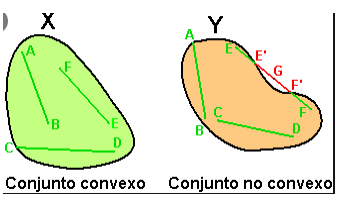
Un técnico tiene n reparaciones urgentes sabiendo que va a tardar t_i en la tarea i. Necesita minimizar el tiempo medio de espera de los clientes (E_i es el tiempo de espera del cliente i). Se quiere minimizar la expresión: $E(n)= \sum_{1..n} E_i$ -> $E(n) = \sum_{1..n} E_i = \sum_{1..n}(n-i+1)t_i$

El tiempo total siempre será el mismo: $T = t_1 + t_2 + \dots + t_n$

Primero se prueba a reparar por orden creciente de sus tiempos y luego de mayor tiempo de reparación a menos. Tras algunas comprobaciones matemáticas vemos que la segunda es la solución óptima. Se comprueba así que siempre da soluciones óptimas locales, pero no siempre globales.

No se garantiza que siempre se obtenga el mínimo global como sería deseable. Las definiciones de óptimo global y local son:

Sea f: R^n -> R y se quiere minimizar la función f(x) cumpliendo que xϵS (x pertenece a las Soluciones). Si existe una x'ϵS (otra x perteneciente al conjunto de soluciones posibles) que cumple que f(x) > f(x') para cada xϵS (que cumpla que cualquier solución que cojas, sea la que sea, va a hacer que la función (que se pretende minimizar siempre) sea mayor), entonces se cumple que x' es la solución óptima o solución óptima global. Si se cumple que xϵS y que cada solución x dentro de un entorno E de soluciones posibles (cada xϵE(x')) provoca que f(x) > f(x') (que la función a minimizar vuelve a ser mayor), entonces x es una solución, pero es óptimo local.

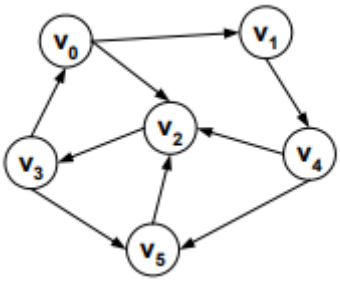


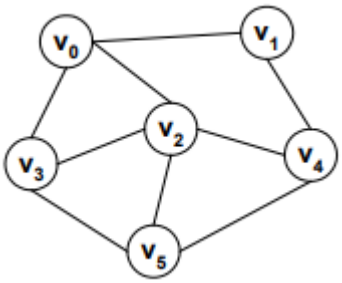
Consideramos a las soluciones (S) un conjunto convexo no vacío y f(x) con xϵS la función que se quiere minimizar. Suponemos que x'ϵS es un óptimo local (buena solución, pero no la mejor). Si f es una función convexa (imagen de la derecha como guía (Dados dos puntos del conjunto S, cualquier segmento lineal que los une está en el conjunto)), x' se convierte en una solución global del problema (que, dentro de su entorno, no van a existir soluciones mejores).

Por tanto, se desea que la función objetivo (f(x)) sea convexa para alcanzar el óptimo global, si no, no tendríamos garantías. No alcanzar soluciones óptimas siempre, se convierte en una ventaja con problemas en los que es imposible alcanzar el óptimo, que es por lo que se usa como algoritmo heurístico (como técnica o método para resolver un problema).

CONCEPTOS BÁSICOS SOBRE GRAFOS

Grafos dirigidos ->

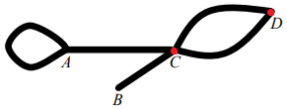




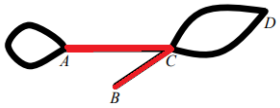
<- Su equivalente no dirigido

Un arco se representa como (xi, xj), siendo xi el vértice inicial y xj el vértice final (ambos terminales).

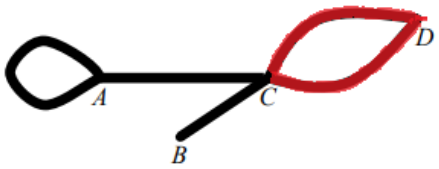
Vértices adyacentes -> Conectados por una arista o un arco.



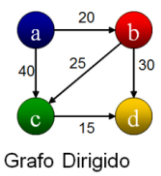
Aristas adyacentes -> Comparten un vértice



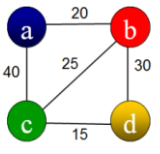
Multigrafo -> Sobre cada par de vértices puede haber más de una arista o arco



Grafo ponderado -> (Dirigido o no) Puede tener sobre cada vértice o arista asociado un peso, distancia o valor.

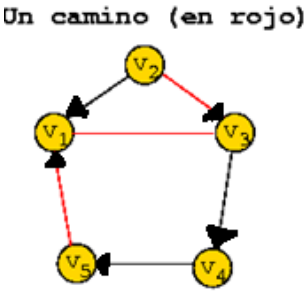


Grafo Dirigido Etiquetado

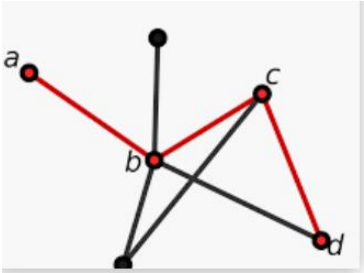


Grafo No Dirigido Etiquetado

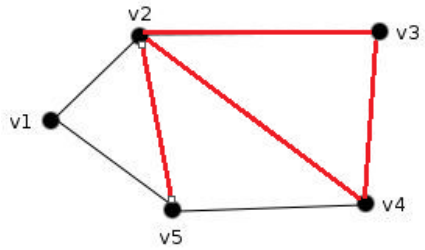
Camino sobre grafo dirigido -> Sucesión de arcos ($a_1,\ldots,a_q$) tal que el vértice final de cada arco, es el inicial del siguiente.



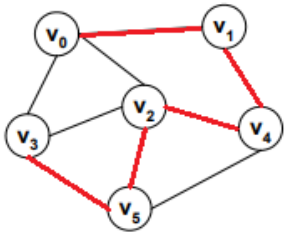
Cadena sobre grafo no dirigido -> Secuencia de aristas adyacentes



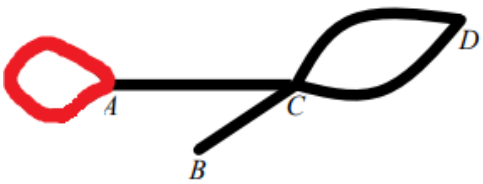
Cadena/camino simple -> No pasa por el mismo arco/arista más de una vez. (Pasa dos veces por V2)



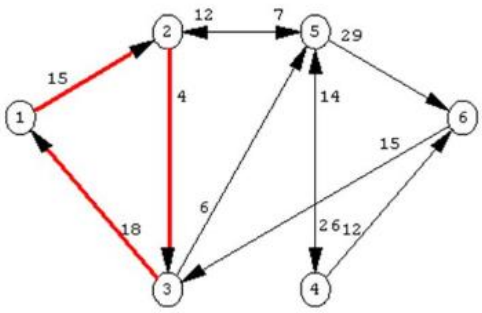
Cadena/camino elemental -> No pasa por el mismo vértice más de una vez (V0, V1, V4, V2, V5, V3)



Bucle -> Arco cuyo vértice inicial y final coinciden

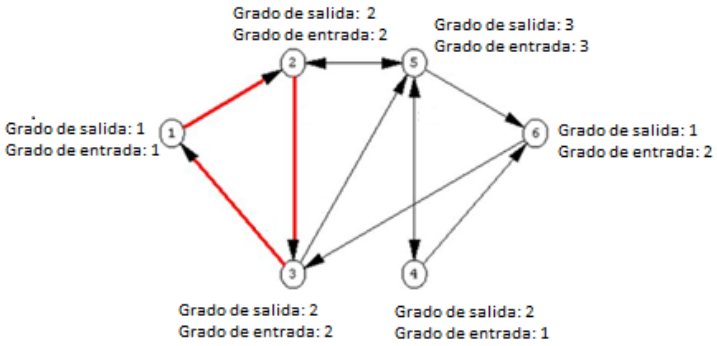


Circuito/ciclo -> Camino/cadena en el que el vértice inicial y el final coinciden (Camino cerrado).

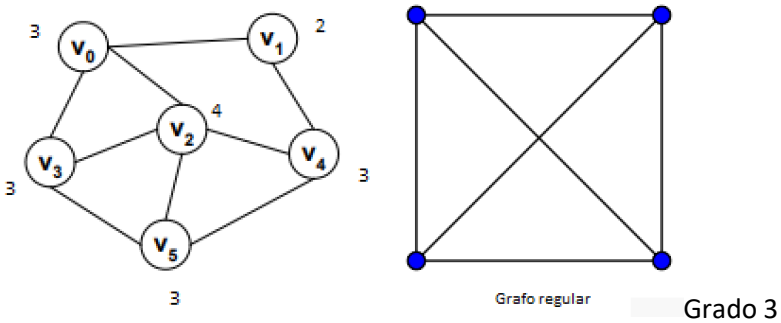


Grado de salida de x -> Número de arcos que tienen a x como vértice inicial

Grado de entrada de x -> Número de aristas que tienen a x como vértice final

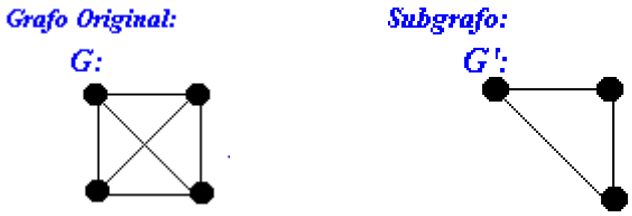


Grado de un vértice en grafo no dirigido -> Número de aristas que contienen a x. Si todos los vértices tienen el mismo grado (r), grafo regular de grado r.



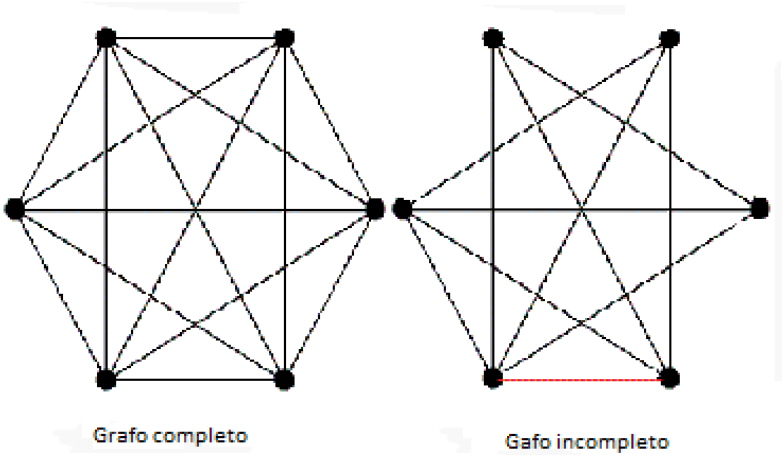
Grado 3

Grafo parcial/Subgrafo -> Un grafo G_p es un grafo parcial de G sin tiene un subconjunto de los vértices originales y un subconjunto de los arcos. Los arcos cuyos vértices terminales estén en el subconjunto de vértices del subgrafo, deben estar contenidos en el subgrafo.

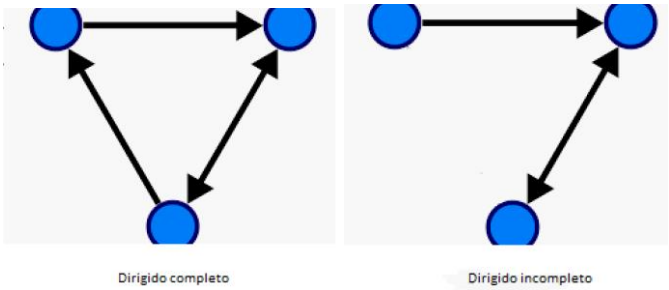


Un subgrafo parcial de un grafo G es un grafo parcial de un subgrafo de G

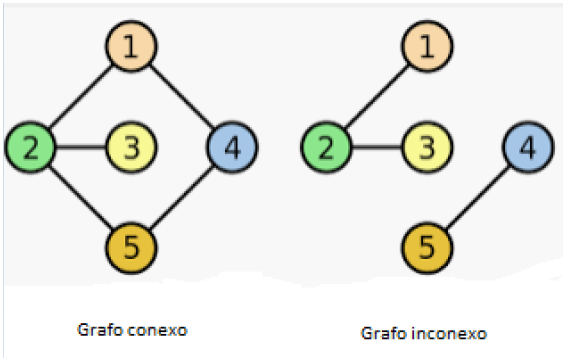
Grafo no dirigido completo -> Todos los pares de vértices unidos por una arista



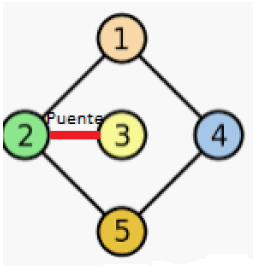
Grafo dirigido completo -> Todos los pares de vértices unidos por una arista. Es completo si su grafo no dirigido asociado es completo



Grafo conexo -> Todo par de vértices está unido por al menos una cadena.

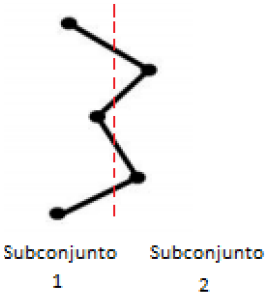


Puente -> Arco o arista que al ser borrada incrementa el número de componentes conexas (Al quitarlo se separa un grafo en dos que no están conectadas entre sí)

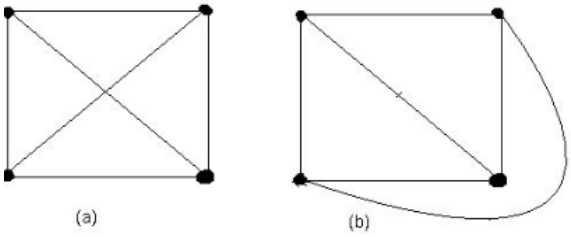


Grafo bipartido -> Si el conjunto de vértices puede dividirse en dos subconjuntos disjuntos y exhaustivos, de manera que todas las aristas tengan un vértice terminal en uno de los conjuntos y el otro vértice terminal en el otro conjunto. Un grafo dirigido es bipartido si su grafo asociado ni dirigido es bipartido.

BIPARTIDO ->



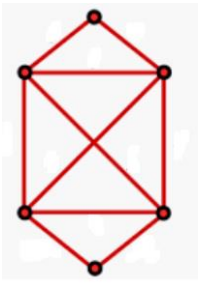
Grafo plano -> Si puede pintarse en un plano sin que ninguna de sus aristas corte a otra.

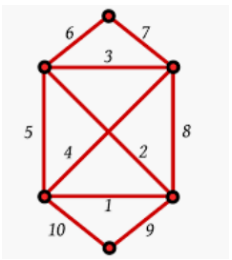


Grafo acíclico o bosque -> Grafo que no tiene circuito o ciclo

Grafo acíclico y conexo <- Árbol

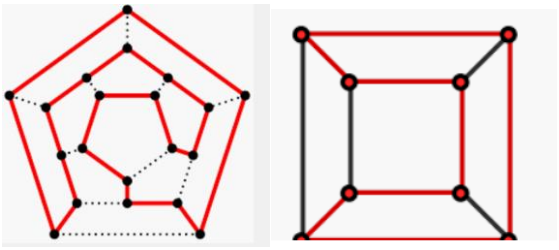
Multigrafo euleriano -> Contiene un circuito/ciclo que pasa por cada arco/arista una sola vez. En el ejemplo vemos que un grafo normal contiene un circuito euleriano siguiendo el orden numérico.





Teorema de Euler -> Si todos los vértices de un grafo son impares, no existe circuito euleriano. Además, si el grafo es conexo y todos sus vértices son de grado par, existe al menos un circuito euleriano. (Problema del cartero chino)

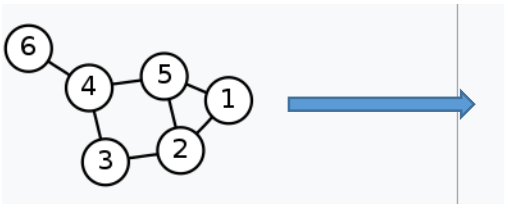
Circuito Hamiltoniano -> Circuito que pasa por cada vértice una única vez y termina en el mismo vértice del que empezó. (Problema del viajante de comercio)



Resumen del problema del viajante de comercio: Un grafo conexo y ponderado que, dado uno de sus vértices, se trata de encontrar el ciclo Hamiltoniano de coste mínimo que empiece en este.

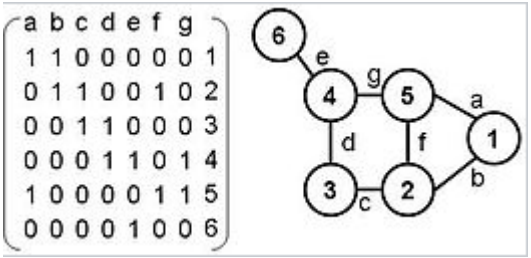
REPRESENTACION DE GRAFOS

Matriz de adyacencia (para grafo con n vértices): A es una matriz cuadrada de dimensión n con $A_G(i,j)$ cero si no existe arista que una los vértices i y j, y uno si lo están. (Adicional: Si la arista es un bucle y el grafo es no dirigido, entonces se suma 2 en vez 1.)

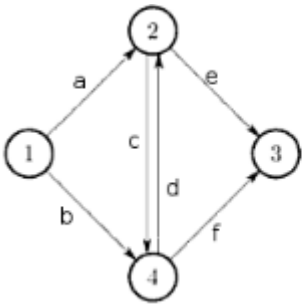


$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Matriz de incidencia. En esta, las columnas representan las aristas del grafo y las filas representan los distintos nodos. Por cada nodo unido por una arista ponemos un 1, y el resto de 0.



Matriz de incidencia para grafos dirigidos. En esta, las columnas representan las aristas del grafo y las filas representan los distintos nodos. Sumas uno en una posición si esa arista sale de ese nodo, sumas -1 si esa arista termina en ese nodo y no sumas nada en cualquier otro caso (en un bucle no sumas, porque al salir de un nodo sumas 1 y al llegar a él mismo restas 1)



	a	b	c	d	e	f
1	1	1	0	0	0	0
2	−1	0	1	−1	1	0
3	0	0	0	0	−1	−1
4	0	−1	−1	1	0	1

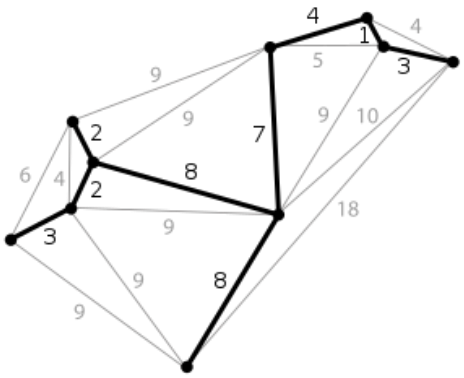
El árbol generador mínimo de un grafo

Sea G = (N,A) un grafo conexo no dirigido donde N es el conjunto de nodos y A es el conjunto de aristas. Además, este grafo es ponderado, por lo que la longitud de cada arista tiene longitudes no negativas (1,12,24…). Pero tenemos un problema a la hora de encontrar un subconjunto T cuyo costo total sea mínimo (es decir que la suma de las longitudes de sus aristas sea lo más pequeña posible). Al grafo (N,T) se le llama un Árbol Generador Mínimo (AGM) del grafo G.

Este problema reúne todas las características que existe en el enfoque de los algoritmos greedy por lo que vamos a explicar dos de ellos que nos ayuden con esta cuestión. Antes, deberemos definir la Propiedad del AGM para asegurar que nuestro algoritmo Greedy funcione correctamente.

Propiedad del AGM

Sea G = (V,A) un grafo no dirigido y conexo donde cada arista tiene una longitud conocida (es decir, ponderado). Un AGM es un subgrafo que tiene que ser un árbol y contener todos los vértices del grafo inicial. Este grafo va a tener un peso menor o igual que el árbol que lo recurre (el grafo original) y NO VA A TENER CICLOS.



Este grafo es un AGM. Cada punto representa un vértice, cada arista tiene su peso indicado. No tiene ciclos. Cumple todas las propiedades.

Algoritmo de Kruskal

La solución que ofrece este algoritmo no tiene por qué ser única, puede haber más de una arista con el mismo peso y tener que decantarnos por un sitio o por otro, de modo que no

existan ciclos al seleccionarla, pero siempre van a quedar con la misma longitud total (con el mismo peso). Los pasos a seguir en este algoritmo son:

1. Ordenar las aristas de forma creciente de peso (1,2,3,4,5…)
2. Repetir la siguiente secuencia
   2.1. Seleccionar la arista con menor peso
   2.2. Borrar la arista seleccionada de E
   2.3. Aceptarla si no forma un ciclo en el árbol o rechazarla en caso contrario

Se repetirá el punto 2 hasta conseguir que tengamos | V | - 1 aristas correctas. De modo gráfico se ve paso por paso en la carpeta algoritmo Kruskal.

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 ((2*a + n – 1)log(n)) para todas las operaciones de búsqueda y unión, ya que como máximo habrá 2*a operaciones de búsqueda y n-1 operaciones de unión
- A lo más tendrá O (a) para el resto de las operaciones

Como conclusión podemos decir que el algoritmo de Kruskal es O (a*log(a)). Además, como todo grafo verifica que

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

Con esto vemos que es inmediato que **LA EFICIENCIA DE KRUSKAL ES O (a*log(n))**

Algoritmo de Prim

Este algoritmo también se va a construir 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. Para ver de forma gráfica como funciona este algoritmo entrar en la carpeta de fotos de algoritmo de Prim

LA EFICIENCIA DEL ALGORITMO DE PRIM ES O (n^2)

La explicación de esta eficiencia se basa en el desarrollo del algoritmo de forma más profunda por lo que no veo necesario perder tiempo en entender el por qué, aunque si estás interesado en él váyase a la página 16/30 del PDF del tema.

Como antes hemos dicho, el algoritmo de Kruskal tiene un tiempo de O (n*log(n)) por lo que a priori el algoritmo de Prim parece 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

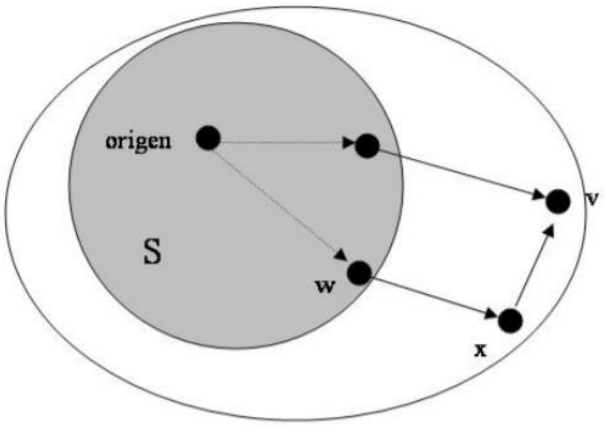
Sea G = (N,A) un grafo dirigido en el que N es el conjunto de nodos y A el de arcos (aristas dirigidas). Cada arista tiene una longitud no negativa. A uno de los nodos le llamaremos nodo origen. El problema es determinar la longitud del camino más corto desde el origen hasta cada uno de los otros nodos del grafo.

El algoritmo que resuelve este problema de la forma más eficiente es el Algoritmo de Dijkstra. Este algoritmo va a considerar dos conjuntos C y S, que son respectivamente el conjunto de nodos candidatos posibles (C) y el conjunto de nodos ya elegidos (S). En cualquier momento S contiene aquellos nodos cuya distancia mínima desde el vértice origen ya se conoce, mientras C contiene todos los otros. Cuando el algoritmo comienza, S solo contiene al origen; cuando el algoritmo termina, S contiene todos los nodos del grafo y el 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, para entender mejor este algoritmo, en la carpeta de fotos hay una carpeta para el algoritmo de Dijkstra para verlo de forma gráfica

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. En la página 19 hay una pequeña demonstración de ello con una nueva figura (2 en la carpeta de fotos) pero solo sirve para reforzar la corrección del Algoritmo de Dijkstra (poca cosa que aprender de ahí vaya).

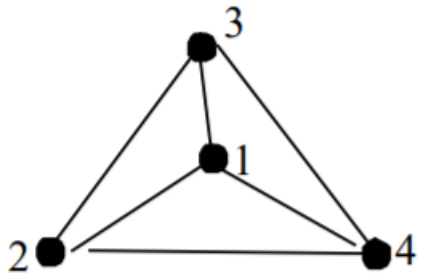
En lo que concierne a la **eficiencia**, supongamos que el algoritmo se aplica a un grafo que tiene n nodos y a aristas. La inicialización toma un tiempo de O (n). En una implementación directa, elegir v en el ciclo repetir requiere examinar todos los elementos de C, de modo que buscamos en n-1, n-2,…, 2 valores de D en las sucesivas iteraciones, dando un **tiempo total en O (n^2)**. El bucle for que se plantea en la implementación de este algoritmo también da un tiempo total de 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.

Como conclusión, 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, 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).

El coloreo de un grafo

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. 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** x(G), al mínimo número de colores que se necesitan para pintar los vértices del grafo G. Así x(G) es el mínimo k tal que G es k colorable.



En este grafo está claro que x(G) es 4. Con esto, el Teorema de los Cuatro Colores establece que, si G es plano, entonces x(G) ≤ 4.

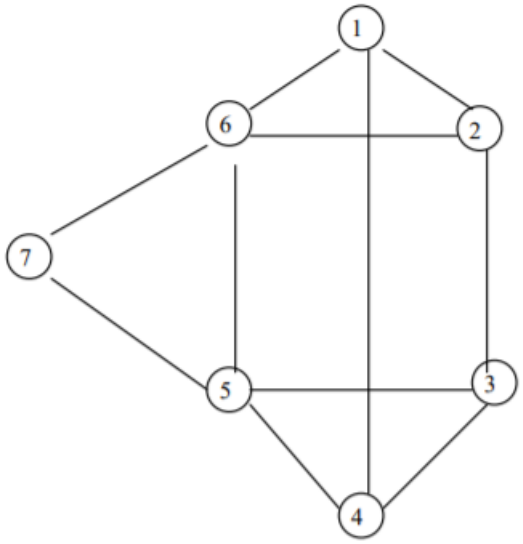
Sea G = (N,A) un grafo no dirigido cuyos nodos tienen que ser coloreados. Queremos usar tan pocos colores como nos sea posible. Un algoritmo greedy (secuencial básico) puede consistir en elegir un color y un nodo inicial arbitrario, y entonces considerar cada nodo restante por fases, pintándolo con este 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.

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

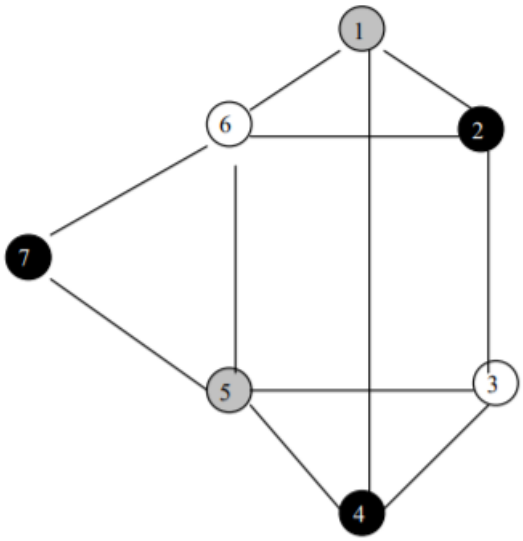
El orden en el que se van seleccionando los vértices es decisivo. Según se haga esa selección, o según estén ordenados los vértices, entre estos métodos destacan:

a) El algoritmo de Welsh y Powell, que ordena los vértices por sus grados de modo descendente
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.

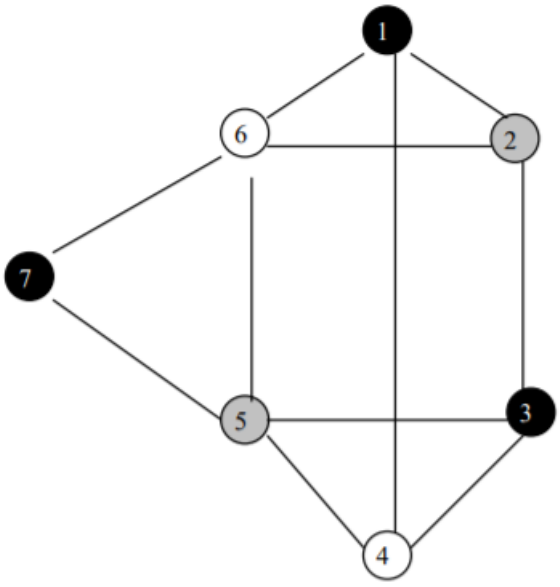
Para ilustrar ambos métodos, consideraremos la siguiente figura sobre la cual ejecutaremos ambos algoritmos:



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 tendríamos el siguiente orden para los nodos: 6, 5, 1, 2, 3, 4, 7
Primero ponemos en 6 un color, sea este blanco, por ejemplo. Entonces seleccionamos otro color (gris) y se lo asignamos al vértice 5. Coloreamos el vértice 1 con el mismo color que el 5, puesto que no son adyacentes.
Ahora para el vértice 2 tenemos que seleccionar ya un color nuevo (negro).
Coloreamos el vértice 3 de 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 de negro también, ya que es la primera clase de color que no tiene vértices adyacentes a él. De forma que el grafo nos queda coloreado así:



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 nos queda la ordenación así: 6, 5, 4, 3, 2, 1, 7.
Entonces coloreamos el vértice 6 con el primer color (blanco), luego el vértice 5 de otro color ya que es adyacente al anterior (gris). A continuación, el vértice 4 se pinta de blanco, como el 6, ya que no es adyacente a este. El siguiente paso es 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 con el vértice 7 del mismo modo que los anteriores, es decir de negro. De forma que el grafo nos queda coloreado así:



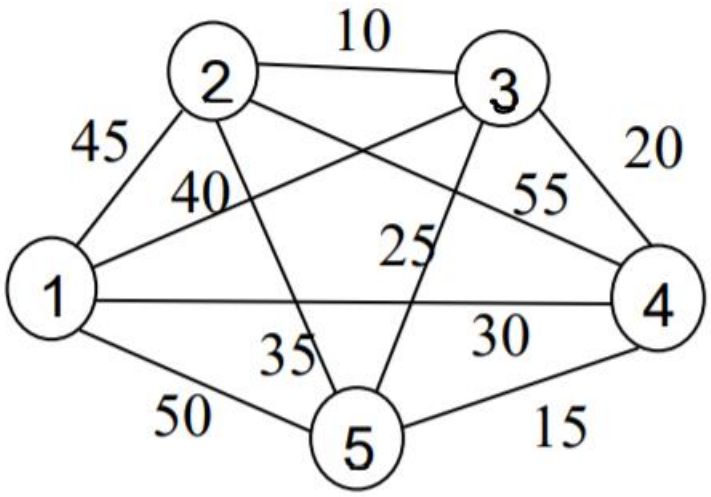
El problema del viajante de comercio

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. Se trata de un problema NP (no polinómico), es decir, para el que no existen algoritmos en tiempo polinómico que lo resuelvan en un tiempo asumible, aunque si los hay exactos pero ineficaces en grafos de dimensiones grandes. Para más de 40 nodos, es necesario utilizar heurísticas, ya que el problema se hace intratable en el tiempo.
Para definir este problema más formalmente, suponemos 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.
Cada vez que escojamos un candidato para incorporarlo a la solución que estemos formando, se deberá cumplir que

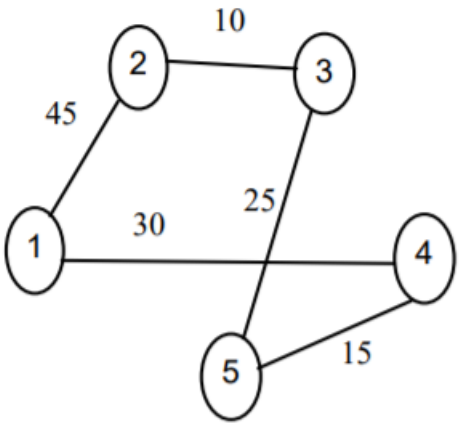
1) No forme un ciclo con las aristas ya escogidas (excepto para la última arista elegida, que completara el recorrido del viajante)
2) No sea la tercera arista elegida incidente a algún nodo

Además, tenemos como objetivo que la suma de las longitudes de las aristas que constituyen 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. Como el PVC (problema del viajante comercio) cumple con las 5 características que buscábamos, es resoluble con un enfoque greedy.

<u>Consideremos por ejemplo el grafo de la siguiente figura:</u>

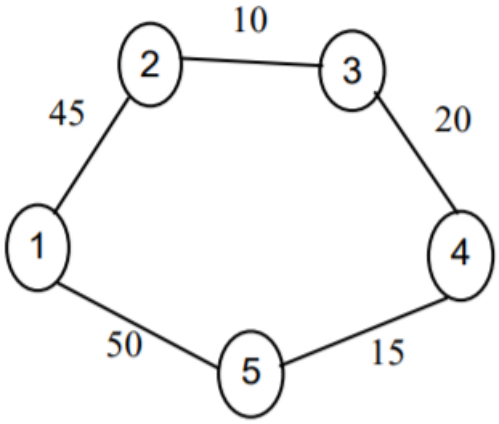


Si empezáramos por el nodo 1 se obtiene



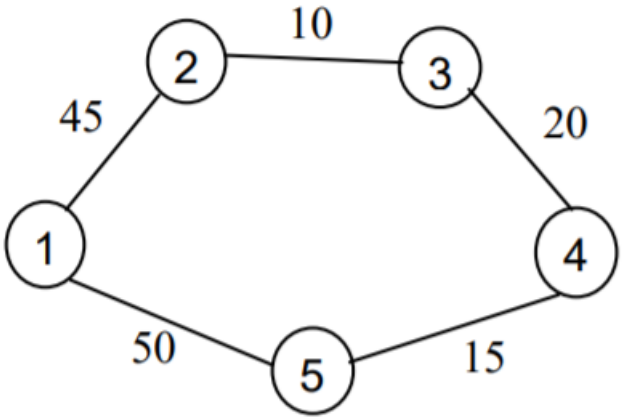
es decir, el circuito es el (1, 4, 5, 3, 2) con una longitud total de 125.

Si empezáramos por el nodo 3 se obtiene



el circuito que se obtiene es (5, 4, 3, 2, 1) con una longitud total de 140.

Si escogemos las aristas de menor costo como en el algoritmo de Kruskal, el circuito que se obtiene es el siguiente



Es decir ((2,3), (4,5), (3,4), (1,2), (1,5)), con un costo final de 140.

Queda claro como el enfoque greedy sirve para resolver el PVC, dando soluciones que, siendo buenas **no hay garantía de que sean óptimas.** 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.

PROBLEMA DE LA MOCHILA

Por su complejidad, este problema se utiliza como prueba para todos los nuevos algoritmos heurísticos. Tenemos n objetos y una mochila. El objeto i pesa wi y la capacidad de la mochila es M. Si se mete en la mochila una fracción de i (fracción de i = xi), que sea 0 <= xi <= 1, se obtiene un beneficio de pi*xi. Se quiere obtener el máximo beneficio en la mochila. El peso total no puede superar M que es la capacidad máxima. Suponemos que los beneficios y los pesos son positivos:

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

En resumen y en cristiano: El beneficio es la importancia que se le da a los objetos. Se pretende maximizar el valor de los objetos sin pasarnos del límite. Empezamos por los objetos que tengan más importancia (beneficio). Por tanto, buscamos valores x1, x2, …, xn (que estén entre 0 y 1) que maximicen

$$f(x) = \sum_{i=1..n} pi * xi \qquad \text{(la suma de beneficios)}$$

teniendo en cuenta la restricción de que:

$$\sum_{i=1..n} pi * xi <= M$$

Ejemplo: La mochila tiene capacidad M = 20, en la que podemos incluir n = 3 objetos, que pesan (w1,w2,w3) = (18,15,10) y sus beneficios son (p1,p2,p3)=(25,24,15). Se plantea como:

Maximizar: 25x1 + 24x2 + 15x3 teniendo en cuenta que:

18x1 + 15x2 + 10x3 <= 20

X1, x2, x3 >=0

Cumple con los requisitos de las primeras páginas. Hay una lista de candidatos (cosas a incluir en la mochila), un concepto de solución (cantidades a incluir de cada objeto), criterio de factibilidad (no superar M), objetivo (maximizar beneficio) y una función de selección. Una solución óptima es una solución para la que la función objetivo alcanza su máximo valor. Como son fracciones, del último objeto que se escoja se puede meter la cantidad que quieras hasta rellenar al completo la mochila, por lo que la restricción de la capacidad M podría ser de igualdad, siempre se puede llenar al máximo.

Supongamos los siguientes datos:

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

M = 100

Podemos rellenar la mochila eligiendo los mayores beneficios, y si el objeto elegido no encaja como debería, se fracciona y se coge solo una parte. Así, obtenemos el mayor beneficio (valor o precio) posible en cada caso. Con esa idea, se obtienen estos datos:

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

Como es greedy, de todos los candidatos coge la mayor cantidad posible del que mayor precio tiene (los 30kg de 65 euros), luego el mayor beneficio posible y con la mayor cantidad posible de él (los 50kg de 60 euros) y el siguiente con más valor es de 40, pero como no se pueden coger los 40 kg porque se superarían los 100, se cogen todos los kg que se puedan (son 20) lo que aumenta el precio en 20.

Se observa que, en el peso total, en cada etapa se incrementa muchísimo la cantidad total, por lo que se intuye que no es la mejor solución de todas. De ese modo, se plantea coger por precio decreciente, quedando:

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

Tampoco es la más óptima, la capacidad sube más lentamente ahora, pero los beneficios crecen más lento.

El siguiente intento sería equilibrar el ritmo al que aumenta el costo y la capacidad. Se hace una fracción de precio/peso y se va escogiendo el que mayor beneficio de por unidad de peso. Ese nuevo dato se representa así:

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

lo que resulta como:

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

Se selecciona la mayor cantidad posible de aquellos que tienen la densidad (precio/peso) más alta (y vamos decreciendo esa densidad).

ALGORITMO DE LA MOCHILA_GREEDY

Variables

(tipo real) Vector P(1:n) -> contiene los precios de forma decreciente

(tipo real) Vector W(1:n)-> contiene los pesos ordenados de forma decreciente

(tipo real) M = capacidad de la mochila

(tipo real) Vector X(1:n) -> solución

(tipo real) cr -> capacidad restante

(tipo int) i, n(nº de objetos)

x = 0; //Inicializamos la solución en 0

cr = M //Al principio la capacidad restante es el total

Para i=1 hasta n hacer //Ambos incluidos para comprobar todos los objetos

Si W(i) > cr Entonces exit endif //Si lo que pesa ese objeto es mayor que la capacidad que queda en la mochila, no lo mete ni hace nada y se sale del bucle.

X(i)=1; //Si no ha entrado en la condición anterior, no hace exit y pasa por aquí, por tanto ese objeto cabía. De esa forma, en el vector de soluciones se pone un 1 en la posición de ese objeto indicando que sí ha entrado en la mochila

cr = cr – W(i) //Se actualiza la capacidad restante quitándole el peso que del objeto que se acaba de añadir.

Repetir //Vuelta a la siguiente iteración del bucle que compruebe el siguiente objeto.

Si i<n entonces X(i) = cr/W(i) endif //Si al terminar el bucle, la i se ha quedado con un valor menor que n significa que no ha comprobado todos los objetos ya que se ha llegado a alguno que ya superaba la capacidad de la mochila. De esta forma, lo que se hace es dividir el espacio que ha quedado entre el peso de ese objeto para ver qué fracción de este es la que cabe, y se pone en la posición que corresponda a ese objeto en el vector de soluciones (no se pone uno porque no se coge entero).

Si los objetos están ordenados correctamente, vemos que las dos primeras soluciones tienen O(n). Había 3 formas de solucionar el problema: priorizando valor, capacidad y densidad; y greedy trabaja maximizando cualquiera de esas medidas que elijamos. Demostramos que la tercera opción óptima si con 1<= i <= n, se cumple: $p_i/w_i \geq p_{i+1}/w_{i+1}$.

DEMOSTRACION:

X = (x1, x2, …, xn) -> Solución al tercer algoritmo.

Y = (y1, y2, …, yn) -> Solución factible cualquiera

Probemos que: $\sum_{i=1}^{n}(x_i - y_i)p_i \geq 0$ * //Se resta la cantidad que se coge en el tercer algoritmo con la cantidad que se cogería en otra solución y esa diferencia se multiplica por el valor que tiene ese objeto. Si se comprueba lo que queremos, la cantidad que se haya cogido en el tercer algoritmo será mayor que las otras, por lo que esas restas darán positivo y la multiplicación por pi dará mayor o igual que 0.

Si todos los xi son iguales a 1, entonces claramente la solución es la óptima (Sería que todas las soluciones cogidas con el tercer algoritmo son 1, o sea, haber cogido el 100% de todos los objetos, opción inmejorable). Por tanto, si el objeto que tiene menor densidad es k (p_k/w_k es un valor muy pequeño), el porcentaje que se ha cogido de ese objeto es menor que 1 porque es el último objeto en ser seleccionado y por tanto se coge solamente la cantidad que quepa ($x_k < 1$). De esa forma tenemos:

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

Se ha partido de la ecuación original *. Se añade la multiplicación y división del peso en cada factor (lo que a fin de cuentas es dejarlo igual, multiplicas y divides por lo mismo, por lo que no cambia el significado de la ecuación). Antes se sumaban n veces, ahora se suma igual, pero en 3 partes. Una primera desde 1 hasta k-1, luego se suma la de k y luego desde k+1 hasta n. Analizándolas por separado vemos que:

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

Como p_k/w_k es la división de peso/valor que da la cantidad más pequeña

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

Estas son iguales.

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

Igualmente, como p_k/w_k es la división de peso/valor que da la cantidad más pequeña, cualquier otro valor para esa división dará mayor o igual.

Como cada una de las 3 partes es mayor o igual que $\sum_{i=1}^{n}(x_i - y_i)w_i\frac{p_k}{w_k}$

Entonces la suma de las 3 ($\sum_{i=1}^{n}(x_i - y_i)p_i$) va a ser mayor o igual seguro:

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

Sacando factor común pk/wk:

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

Sabemos que la solución más óptima coge tal cantidad de cada objeto (xi) que rellena al completo la mochila ($\sum_i x_i w_i = M$), pero cualquier otra solución, al no ser tan buena, no tiene por qué cumplir ese requisito ($\sum_i y_i w_i \leq M$). De esta forma, sacando factor común la wi y restando las xi – yi, siempre será un valor positivo. Por tanto:

$$\sum_{i=1}^{n}(x_i - y_i)p_i \geq \frac{p_k}{w_k}\sum_{i=1}^{n}(x_i - y_i)w_i \Rightarrow \sum_{i=1}^{n}(x_i - y_i)p_i \geq 0$$

Que es lo que queríamos demostrar.