Técnicas de diseño de algoritmos
Algoritmos voraces

Luis Javier Rodríguez Fuentes
Amparo Varona Fernández

Departamento de Electricidad y Electrónica
Facultad de Ciencia y Tecnología, UPV/EHU
luisjavier.rodriguez@ehu.es
amparo.varona@ehu.es

OpenCourseWare 2015
Campus Virtual UPV/EHU

Algoritmos voraces: Indice

Algoritmos voraces: Introducción

Los así llamados algoritmos voraces o ávidos (en inglés greedy algorithms), deben su nombre a su comportamiento: en cada etapa *toman lo que pueden* sin analizar las consecuencias, es decir, son glotones por naturaleza.

- El enfoque es *miope*, las decisiones se toman utilizando únicamente la información disponible en cada paso, sin tener en cuenta los efectos que estas decisiones puedan tener en el futuro.
- Los algoritmos voraces resultan *fáciles de diseñar*, fáciles de implementar y, cuando funcionan, son eficientes.
- Se usan principalmente para resolver problemas de optimización:
  - Dado un problema con n entradas, se trata de obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema.
  - Cada uno de los subconjuntos que cumplan las restricciones son soluciones factibles o prometedoras.
  - Una solución factible que maximice o minimice una función objetivo se denominará solución óptima.

Algoritmos voraces: Esquema de diseño

El funcionamiento de los algoritmos voraces puede resumirse como sigue:

1. Para resolver el problema, un algoritmo voraz tratará de tomar el conjunto de decisiones (escoger el conjunto de candidatos) que, cumpliendo las restricciones del problema, conduzca a la solución óptima.
2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parezca mejor, sin considerar las consecuencias futuras.
3. De acuerdo a esta estrategia, tomará aquella decisión (escogerá a aquel candidato) que produzca un óptimo local en cada etapa, suponiendo que ello conducirá, a su vez, a un óptimo global para el problema.
4. Antes de tomar una decisión, el algoritmo voraz comprobará si es prometedora. En caso afirmativo, la decisión se ejecuta (es decir, se da un nuevo paso hacia la solución) sin posibilidad de retroceder; en caso negativo, la decisión será descartada para siempre.
5. Cada vez que se toma una nueva decisión, el algoritmo voraz comprobará si la solución construida hasta ese punto cumple las restricciones de una solución al problema.

Algoritmos voraces: Esquema de diseño

```
def Algoritmo_Voraz(Conjunto_Entrada):
    Conjunto = Conjunto_Entrada
    Solucion = []
    encontrada = False
    while not EsVacio(Conjunto) and not encontrada:
        x = Seleccionar_Mejor_Candidato(Conjunto)
        Conjunto = Conjunto - [x]
        if EsFactible(Solucion + [x]):
            Solucion = Solucion + [x]
            if EsSolucion(Solucion):
                encontrada = True
    return Solucion
```

Algoritmos voraces: Consideraciones sobre su aplicación

- **¿Por qué no utilizar siempre algoritmos voraces?**
  1. porque no todos los problemas admiten esta estrategia de resolución;
  2. y porque la búsqueda de óptimos locales no tiene por qué conducir a un óptimo global.

- La **estrategia voraz** trata de ganar todas las batallas sin darse cuenta de que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.

- Desgraciadamente, y como en la vida misma, pocos hechos hay para los que podamos afirmar sin miedo a equivocarnos que lo que parece bueno hoy será bueno también en el futuro. Y aquí radican las limitaciones de este tipo de algoritmos.

Algoritmos voraces: Consideraciones sobre su aplicación

- Es fundamental encontrar la función de selección adecuada que garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión.
- Deberá proporcionarse la demostración formal de que la función de selección conduce a un óptimo global, cualquiera que sea la entrada.
- Así pues, al plantear un algoritmo voraz (que por diseño será rápido y eficiente), habrá que demostrar que conduce a la solución óptima del problema en todos los casos.
- Un algoritmo voraz podría conducir a soluciones subóptimas. En tales casos, la demostración de suboptimalidad consistirá en presentar un contraejemplo para el que el algoritmo no proporcione la solución óptima.

Algoritmos voraces: Consideraciones sobre su aplicación

- Precisamente, debido a su eficiencia, los algoritmos voraces se usan incluso en casos en los que no necesariamente encuentran la solución óptima.
- En algunas ocasiones, es imprescindible encontrar pronto o en un tiempo finito una solución razonablemente buena, aunque no sea la óptima.
- En otras ocasiones, un algoritmo voraz proporciona una solución rápida (subóptima) del problema y, a partir de ella, aplicando algoritmos más sofisticados, la solución óptima se obtiene más rápidamente.
- En resumen, los algoritmos voraces se aplican en muchas circunstancias debido a su eficiencia, aunque sólo den una solución *aproximada* al problema de optimización planteado.

Cambio de monedas: Planteamiento

Enunciado: Considerese un sistema monetario con n monedas distintas, de valores $v_1, v_2, \ldots, v_n$, y supóngase que hay un número ilimitado de monedas de cada valor. El problema del cambio consiste en descomponer una cierta cantidad a devolver M en el menor número posible de monedas.

- Es fácil implementar un algoritmo voraz para resolver este problema, aplicando el método que utilizamos habitualmente en nuestra vida diaria (primero monedas grandes, usando tantas como se pueda sin pasarse de la cantidad a devolver).
- Sin embargo, la optimalidad de dicho algoritmo dependerá del sistema monetario utilizado.
- Se plantean dos situaciones distintas, en cada una de las cuales se comprobará si dicho algoritmo voraz encuentra siempre la solución óptima:
  1. Cada moneda del sistema monetario vale al menos el doble que la moneda de valor inmediatamente inferior y existe una moneda de valor unitario.
  2. El sistema monetario está compuesto por monedas de valores $1, p, p^2, p^3, \ldots, p^n$, donde $p > 1$ y $n > 0$.

Cambio de monedas: Conceptos básicos

- **Conjunto de candidatos**

  Los valores de las distintas monedas de que consta el sistema monetario.

- **Solución**

  Un conjunto de monedas cuyo valor total es igual al importe a devolver.

- **Completable**

  La suma de los valores de las monedas escogidas en un momento dado, que no supera el importe a devolver.

- **Función de selección**

  Elegir la moneda de mayor valor del sistema monetario tal que sumada al completable no supere el importe a devolver.

- **Función objetivo**

  Número total de monedas utilizadas en la solución **(debe minimizarse)**.

Cambio de monedas: Algoritmo

```
monedas=[500,200,100,50,25,5,1]
def Devolver_Cambio(cantidad, monedas):
    n = len(monedas)
    cambio = [0 for i in range(n)]
    for i in range(n):
        while monedas[i] < cantidad:
            cantidad = cantidad - monedas[i]
            cambio[i] = cambio[i] + 1
    return cambio
```

Este algoritmo es de complejidad lineal respecto a n: número de monedas del sistema monetario.

¿Se nos ocurre alguna mejora?

Cambio de monedas: Análisis de la optimalidad

PRIMER SUPUESTO (demostración de suboptimalidad)

- Supongamos que nuestro sistema monetario está compuesto por las siguientes monedas:

$$\text{monedas} = [11, 5, 1]$$

- Este sistema verifica las condiciones del primer supuesto, ya que disponemos de una moneda de valor unitario y cada una de las monedas vale más del doble que la moneda de valor inmediatamente inferior.
- Supongamos que la cantidad a devolver es $M = 15$. El algoritmo voraz del cambio de monedas conduce a una descomposición en 5 monedas:

$$15 = 11 + 1 + 1 + 1 + 1$$

- Sin embargo, existe una descomposición que utiliza menos monedas (exactamente 3):

$$15 = 5 + 5 + 5$$

Cambio de monedas: Análisis de la optimalidad

SEGUNDO SUPUESTO (demostración de optimalidad - 1/5)

- Para demostrar que el algoritmo voraz propuesto encuentra la solución óptima, vamos a apoyarnos en una propiedad general de los números naturales:

> *Si p es un número natural mayor que 1, todo número natural x puede expresarse de forma única como:*
>
> $$x = r_0 + r_1 p + r_2 p^2 + \ldots + r_n p^n \qquad (1)$$
>
> *con $0 \leq r_i < p$, y siendo n el menor natural tal que $x < p^{n+1}$.*

- Nuestro algoritmo calcula los r_i, que indican el número de monedas a devolver de cada valor p_i. Se trata de demostrar que esa descomposición es óptima. Para ello, consideremos otra descomposición distinta:

$$x = s_0 + s_1 p + s_2 p^2 + \ldots + s_m p^m$$

entonces deberá ser:

$$\sum_{i=0}^{n} r_i < \sum_{i=0}^{m} s_i \qquad (2)$$

Cambio de monedas: Análisis de la optimalidad

SEGUNDO SUPUESTO (demostración de optimalidad - 2/5)

- Demostraremos la desigualdad (2) para $p = 2$ (el caso general es análogo).
- Consideremos en primer lugar la descomposición obtenida por el algoritmo voraz:

$$x = r_0 + 2r_1 + 2^2 r_2 + \ldots + 2^n r_n$$

donde se tiene que $x < 2^{n+1}$ y los coeficientes r_i toman los valores 0 o 1.

- Consideremos ahora otra descomposición distinta:

$$x = s_0 + 2s_1 + 2^2 s_2 + \ldots + 2^m s_m$$

- **Paso 1:** En primer lugar, $x < 2^{n+1} \Rightarrow m \leq n$. Definimos entonces $s_{m+1} = s_{m+2} = \ldots = s_n = 0$, para tener n términos en ambas descomposiciones.

Cambio de monedas: Análisis de la optimalidad

SEGUNDO SUPUESTO (demostración de optimalidad - 3/5)

- **Paso 2:** Como ambas descomposiciones son distintas, sea k el primer índice tal que $r_k \neq s_k$. Podemos suponer sin pérdida de generalidad que $k = 0$, puesto que si no lo fuera podríamos restar a ambos lados de la desigualdad los términos iguales y dividir por la potencia de 2 adecuada.

  Demostraremos a continuación que si $r_0 \neq s_0$, entonces $r_0 < s_0$.

  - Si x es par entonces $r_0 = 0$. Como $s_0 \geq 0$ y estamos suponiendo que $r_0 \neq s_0$, ha de ser $s_0 > 0$ y por tanto $r_0 < s_0$.
  - Si x es impar entonces $r_0 = 1$. Pero la segunda descomposición de x ha de contener también al menos una moneda de una unidad, y por tanto $s_0 \geq 1$. Puesto que $r_0 \neq s_0$, se deduce que $r_0 < s_0$.

  Consideremos ahora la cantidad $s_0 - r_0 > 0$. Tal cantidad ha de ser par, puesto que $x - r_0$ lo es. Y por ser par, siempre podremos **mejorar** la segunda descomposición $(s_0, s_1, \ldots, s_n)$ cambiando $s_0 - r_0$ monedas de una unidad por $(s_0 - r_0)/2$ monedas de 2 unidades, de donde se deduce que:

$$s_0 + s_1 + s_2 + \ldots + s_n \;>\; r_0 + \left(s_1 + \frac{s_0 - r_0}{2}\right) + s_2 + \ldots + s_n \qquad (3)$$

Cambio de monedas: Análisis de la optimalidad

SEGUNDO SUPUESTO (demostración de optimalidad - 4/5)

- **Paso 3:** El razonamiento del paso 2 nos ha llevado a una nueva descomposición, mejor que la segunda, tal que:

$$x = r_0 + 2\left(s_1 + \frac{s_0 - r_0}{2}\right) + 2^2 s_2 + \ldots + 2^n s_n$$

- Podemos volver a aplicar el mismo razonamiento sobre la nueva descomposición, y así sucesivamente, viendo que $s_i \geq r_i \ \forall i \in [0, n-1]$, y obteniendo nuevas descomposiciones, cada una mejor que la anterior, hasta llegar en el último paso a una descomposición de la forma:

$$x = r_0 + 2r_1 + 2^2 r_2 + \ldots + 2^{n-1} r_{n-1} + 2^n (s_n + A_{n-1}) \qquad (4)$$

donde:

$$A_i = \sum_{j=0}^{i} \frac{s_j - r_j}{2^{(i-j+1)}}$$

teniéndose que:

$$s_0 + s_1 + \ldots + s_n > r_0 + r_1 + \ldots + r_i + (s_{i+1} + A_i) + \ldots + s_n \qquad (5)$$

Cambio de monedas: Análisis de la optimalidad

SEGUNDO SUPUESTO (demostración de optimalidad - 5/5)

- **Paso 4:** Llegados a este punto, la demostración es sencilla. A partir de la expresión (4) y aplicando la propiedad de unicidad (1), se ha de verificar:

$$s_n + A_{n-1} = r_n$$

- Esto, junto a la cadena de desigualdades (5), viene a demostrar la desigualdad (2). El razonamiento sería idéntico para $p > 2$.
- Resta sólo preguntarnos **por qué esta demostración no funciona para cualquier sistema monetario**. La explicación se encuentra en el paso 2, concretamente en la expresión 3, que en este sistema siempre permite intercambiar un número de monedas de un valor por un número inferior de monedas del valor inmediatamente superior, lo cual no es posible en otros sistemas monetarios.

El problema de la mochila (continuo)

Enunciado: Considerese una mochila capaz de albergar un peso máximo M, y n elementos con pesos $p_1, p_2, \ldots, p_n$ y beneficios $b_1, b_2, \ldots, b_n$. Se trata de encontrar las proporciones de los n elementos $x = (x_1, x_2, ..., x_n)$ (con $0 \leq x_i \leq 1$) que deberán introducirse en la mochila para maximizar el beneficio.

Por tanto, buscamos los valores $x_1, x_2, ..., x_n$ (con $0 \leq x_i \leq 1$) que maximizan la siguiente función objetivo:

$$f(x) = \sum_{i=1}^{n} b_i x_i$$

con la restricción:

$$\sum_{i=1}^{n} p_i x_i \leq M$$

El problema de la mochila (continuo)

Algoritmo propuesto

```
def Mochila(peso, beneficio, M):
    n=len(peso)
    solucion = [0.0 for i in range(n)]
    peso_actual = 0.0
    while peso_actual < M:
        i = mejor_objeto_restante(peso, beneficio)
        if peso[i] + peso_actual <= M:
            solucion[i] = 1
            peso_actual = peso_actual + peso[i]
        else:
            solucion[i] = (M - peso_actual) / peso[i]
            peso_actual = M
     return solucion
```

El problema de la mochila (continuo)

Supongamos el siguiente ejemplo:

$n = 5,\ M = 100$					
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

Podemos considerar tres enfoques distintos en la selección del mejor candidato:

Selección	x_i					Valor
Máximo b_i	0.0	0.0	1.0	0.5	1.0	146
Mínimo p_i	1.0	1.0	1.0	1.0	0.0	156
Máximo b_i/p_i	1.0	1.0	1.0	0.0	0.8	164

El problema de la mochila (continuo)

A continuación demostraremos el siguiente enunciado:

Si se seleccionan los objetos por orden decreciente b_i/p_i, el algoritmo es óptimo

- Supongamos, sin pérdida de generalidad, que los objetos disponibles están ordenados por valor decreciente de beneficio por unidad de peso:

$$b_1/p_1 \geq b_2/p_2 \geq \ldots \geq b_n/p_n$$

- Sea $x = (x_1, x_2, \ldots, x_n)$ la solución encontrada por el algoritmo voraz.
  - Si $x_i = 1$ para todo i, la solución es óptima.
  - En caso contrario, sea j el menor índice tal que $x_j < 1$. Por la forma en que trabaja el algoritmo, $x_i = 1$ para todo $i < j$, $x_i = 0$ para todo $i > j$, y además $\sum_{i=1}^{n} p_i x_i = M$.
- Sea $f(x) = \sum_{i=1}^{n} b_i x_i$ el beneficio que se obtiene para esa solución.

El problema de la mochila (continuo)

- Consideremos una solución distinta $y = (y_1, y_2, \ldots, y_n)$, y sea $f(y) = \sum_{i=1}^{n} b_i y_i$ su beneficio.
- Por ser solución, cumple que $\sum_{i=1}^{n} p_i y_i \leq M$. Entonces, puesto que $\sum_{i=1}^{n} p_i x_i = M$, podemos afirmar que:

$$\sum_{i=1}^{n} (p_i x_i - p_i y_i) = \sum_{i=1}^{n} p_i (x_i - y_i) \geq 0 \tag{6}$$

- La diferencia de beneficios es:

$$f(x) - f(y) = \sum_{i=1}^{n} (b_i x_i - b_i y_i) = \sum_{i=1}^{n} \frac{b_i}{p_i} p_i (x_i - y_i) \tag{7}$$

El problema de la mochila (continuo)

- Considerese ahora de nuevo el menor índice j tal que $x_j < 1$ en la solución x obtenida mediante el algoritmo voraz propuesto:
  - Si $i < j$ entonces $x_i = 1$, y por tanto $(x_i - y_i) \geq 0$. Además, por la forma en que se han ordenado los elementos, $\frac{b_i}{p_i} \geq \frac{b_j}{p_j}$.
  - Si $i > j$ entonces $x_i = 0$, y por tanto $(x_i - y_i) \leq 0$. Además, por la forma en que se han ordenado los elementos, $\frac{b_i}{p_i} \leq \frac{b_j}{p_j}$.
  - Por último, si $i = j$, entonces $\frac{b_i}{p_i} = \frac{b_j}{p_j}$.
- Así pues, para todo i se tiene que: $\frac{b_i}{p_i}(x_i - y_i) \geq \frac{b_j}{p_j}(x_i - y_i)$.
- Por último, teniendo en cuenta este último resultado y la desigualdad (6), la diferencia de beneficios (7) queda:

$$f(x) - f(y) = \sum_{i=1}^{n} \frac{b_i}{p_i} p_i (x_i - y_i) \geq \frac{b_j}{p_j} \sum_{i=1}^{n} p_i (x_i - y_i) \geq 0$$

es decir, $f(x) \geq f(y)$, tal como queríamos demostrar.

El problema de la mochila (discreto)

- En este caso, el conjunto solución no puede contener fracciones de elementos, es decir, x_i sólo podrá ser 0 o 1, para todo $i \in [1, n]$.
- Como en el problema continuo, el objetivo es maximizar el beneficio, definido como $\sum_{i=1}^{n} b_i x_i$, con la restricción $\sum_{i=1}^{n} p_i x_i \leq M$.
- **¿Seguirá dando la solución óptima en este caso el algoritmo voraz propuesto para el caso continuo?**

El problema de la mochila (discreto)

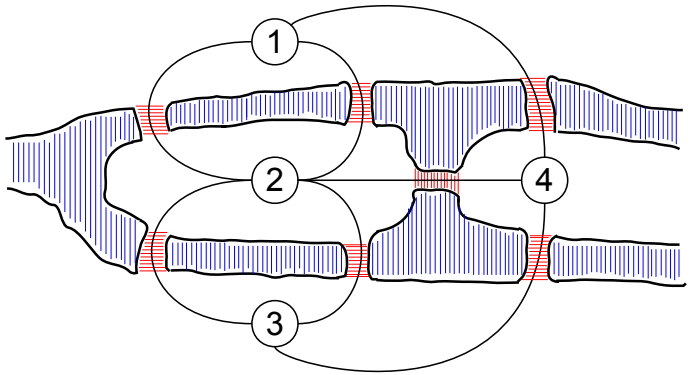
- **Respuesta:** Lamentablemente no, como lo demuestra el siguiente **contraejemplo**.
- Supongamos una mochila de capacidad $M = 6$, con los siguientes elementos (almacenados en orden decreciente de ratio beneficio/peso):

$$\frac{b_i}{p_i} : \frac{11}{5}, \frac{6}{3}, \frac{6}{3}$$

- Nuestro algoritmo voraz elige como mejor candidato el primer elemento, y como ya no caben más elementos, la solución tiene un beneficio de 11.
- Sin embargo, hay una solución mejor: podemos introducir en la mochila los dos últimos elementos, que no superan su capacidad, con un beneficio total de 12.

Grafos: Introducción

- La teoría de grafos surge de un trabajo del matemático suizo Leonhard Euler en 1736, basado en el problema de los puentes de Königsberg.
- La ciudad de Königsberg (hoy Kaliningrado), era famosa por los siete puentes que unen ambas márgenes del río Pregel con dos de sus islas (véase el esquema inferior). El problema se planteaba como sigue: ¿es posible, partiendo de un lugar arbitrario, regresar al lugar de partida cruzando cada puente una sola vez?



- Euler consigue demostrar que el grafo asociado al esquema de puentes de Königsberg no tiene solución.
- De hecho, Euler resuelve un problema más general: ¿qué condiciones debe satisfacer un grafo para garantizar que se puede regresar al nodo de partida sin pasar por la misma arista más de una vez?
- Se define **grado** de un nodo como el número de aristas (caminos que lo conectan con otro nodo) que alcanzan dicho nodo. Pues bien, sólo es posible hacer un recorrido cíclico de un grafo, pasando exactamente una vez por cada arista, si todos los nodos tienen grado par.

Grafos: Definiciones

- Un **grafo simple** G es un par (V, E) tal que:
  - V es un conjunto finito no vacío de nodos o vértices. Se llama *orden de G* al número de vértices de G y se denotará n.
  - E es un subconjunto del producto cartesiano $V \times V$ (pares de vértices), cuyos elementos se denominan arcos (grafos dirigidos) o aristas (grafos no dirigidos), que conectan los vértices de V.
- En un **grafo dirigido**, cada arco viene dado por un par ordenado $< u, v >$ donde $u \in V$ es el vértice inicial del arco y $v \in V$ es el vértice final.
- En un **grafo no dirigido**, no se distinguen los vértices inicial y final. Dados dos vértices $u, v \in V$, la arista (u, v) conecta dichos vértices en ambos sentidos.
- En un grafo no existen autociclos ni multi-arcos.
- El máximo número de aristas de un grafo no dirigido es $n(n-1)/2$. El máximo número de arcos de un grafo dirigido es $n(n-1)$. En ambos casos, se dice que el grafo es completo.

Grafos: Definiciones

- El **grado de un vértice** es el número de arcos o aristas que inciden en dicho vértices.
- En grafos dirigidos, se definen el **grado de entrada**: número de arcos que entran al vértice, y el **grado de salida**: número de arcos que salen del vértice.
- Dos arcos o aristas se dice que son **adyacentes** si tienen un vértice en común. Dos vértices son adyacentes si existe algún arco o arista que los une.
- **Grafos ponderados**. En muchas aplicaciones, los arcos de un grafo llevan asignado un cierto valor real, denominado **peso**, que representa la distancia o el coste de desplazamiento entre dos vértices adyacentes.
- Llamamos **subgrafo** de $G = (V, E)$ a un grafo $G_b = (V_b, E_b)$ tal que $V_b \subseteq V$ y $E_b = E \cap (V_b \times V_b)$. En el subgrafo G_b están todos los arcos o aristas que hay en E_b y que unen los vértices de V_b.

Grafos: Definiciones

- Un **camino** entre los los vértices u y v de un grafo no dirigido $G = (V, E)$ es una secuencia de nodos $u, w_1, \ldots, w_k, v$ tal que las aristas $(u, w_1), \ldots, (w_k, v) \in E$. En el caso de grafos dirigidos, un camino se define de la misma manera, pero usando arcos en lugar de aristas.
- Se denomina **camino simple** a un camino en el cual todos sus los vértices son distintos salvo, quizás, el primero y el último.
- Se denomina **ciclo** a un camino simple en el cual los vértices primero y último son iguales.
- La **longitud de un camino** es el número de arcos/aristas que lo componen.
- En un grafo no dirigido, dos vértices u y v están **conectados** si existe al menos un camino entre ambos. Un grafo no dirigido es **conexo** si existe al menos un camino entre todo par de vértices.
- En un grafo dirigido, un vértice u es **accesible** desde otro vértice v si existe al menos un camino que parte de v y termina en u. Un grafo dirigido es **fuertemente conexo** si todo vértice u es accesible desde cualquier otro vértice v.

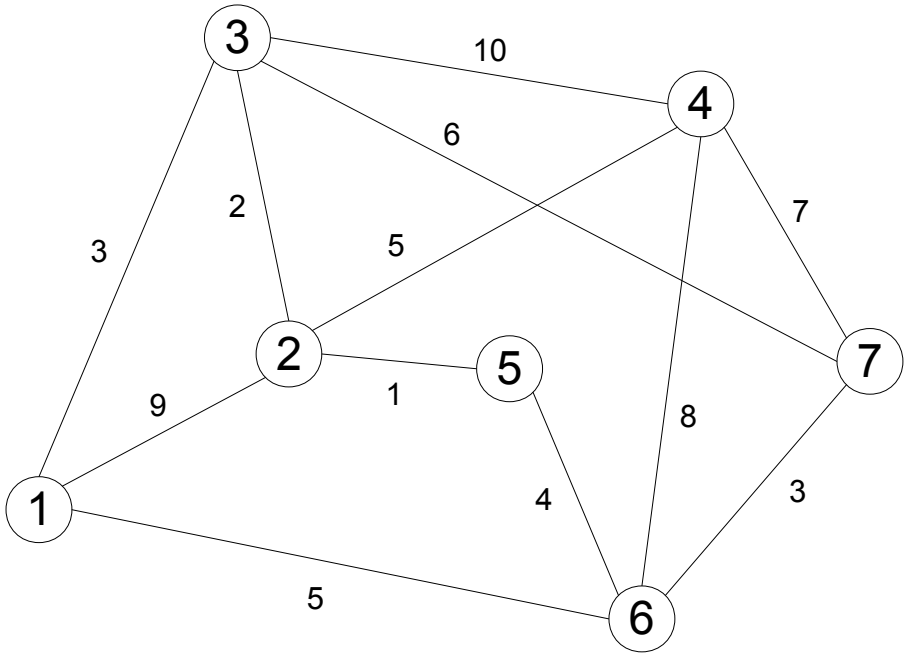
Representación de grafos

Las dos representaciones de grafos más utilizadas son:

- Matriz de Adyacencia: Se utiliza una matriz M de tamaño $n \times n$ (n: número de vértices del grafo) donde filas y columnas hacen referencia a los vértices, y cada elemento $M[i][j]$ almacena o bien un valor booleano que indica la conexión o no entre los vértices i y j, o bien el peso/coste de dicha conexión (que sería ∞ si no existiera). Suele hacerse $M[i][i] = 0$, aunque por coherencia es más correcto hacer $M[i][i] = \infty$.

- Lista de Adyacencia: Se utiliza una estructura L de tamaño n (un elemento por cada vértice) donde $L[i]$ contiene la lista de vértices adyacentes a i y, en el caso de grafos ponderados, también el peso/coste asociado a cada conexión.

Representación de grafos: Matriz de Adyacencia



	1	2	3	4	5	6	7
1	0	9	3	∞	∞	5	∞
2	9	0	2	5	1	∞	∞
3	3	2	0	10	∞	∞	6
4	∞	5	10	0	∞	8	7
5	∞	1	∞	∞	0	4	∞
6	5	∞	∞	8	4	0	3
7	∞	∞	6	7	∞	3	0

Representación de grafos: Matriz de Adyacencia

Matrices en Python

El siguiente código:

```
table = [[0 for i in range(3)] for j in range(3)]
print(table)

for d1 in range(3):
    for d2 in range(3):
        table[d1][d2] = d1+d2+2
print(table)
```

produce la salida:

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[2, 3, 4], [3, 4, 5], [4, 5, 6]]
```


Representación de grafos: Matriz de Adyacencia

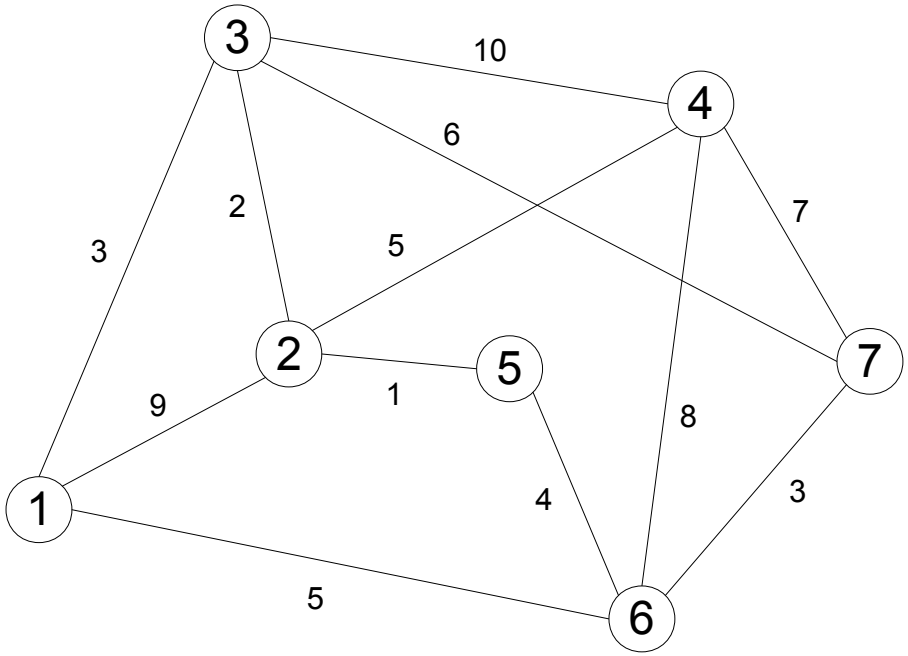
Ejemplo de implementación en Python

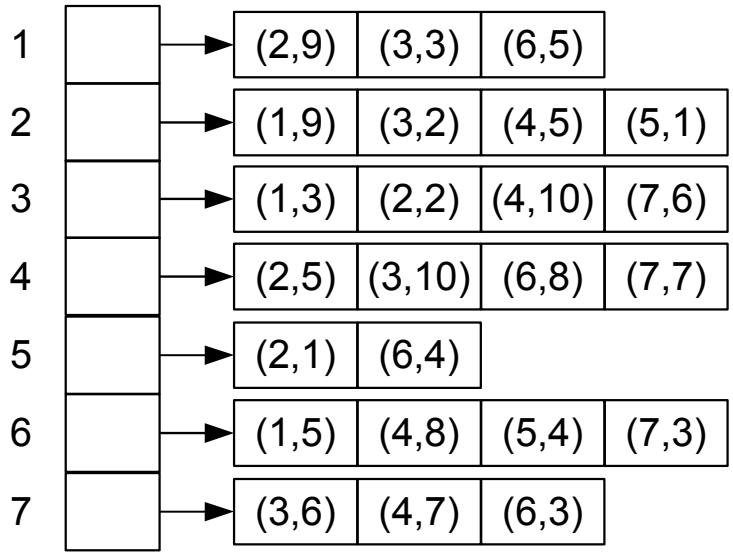
```
Inf = 1e100
G = [[Inf,50,30,100,10],
     [Inf,Inf,Inf,Inf,Inf],
     [Inf,5,Inf,Inf,Inf],
     [Inf,20,50,Inf,Inf],
     [Inf,Inf,Inf,10,Inf]]
for i in G:
    print(i)
```

El código anterior produce la salida:

```
[1e+100, 50, 30, 100, 10]
[1e+100, 1e+100, 1e+100, 1e+100, 1e+100]
[1e+100, 5, 1e+100, 1e+100, 1e+100]
[1e+100, 20, 50, 1e+100, 1e+100]
[1e+100, 1e+100, 1e+100, 10, 1e+100]
```

Representación de grafos: Lista de Adyacencia





Representación de grafos: Lista de Adyacencia

Diccionarios en Python

Considerese el siguiente código:

```
def histograma(s):
    d={}
    for c in s:
        if c not in d:
            d[c]=1
        else:
            d[c]+=1
    return d
```

A continuación vemos un ejemplo de aplicación:

```
>>> h=histograma("brontosaurus")
>>> print(h)
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't':1}
```

Representación de grafos: Lista de Adyacencia

Ejemplo de implementación en Python (mediante diccionarios)

```
G = { 1 : [(2,50), (3,30), (4,100), (5,10)],
      3 : [(2,5)],
      4 : [(2,20), (3,50)],
      5 : [(4,10)] }
```

```
for i in G:
    print (G[i])
```

```
[(2, 50), (3, 30), (4, 100), (5, 10)]
[(2, 5)]
[(2, 20), (3, 50)]
[(4, 10)]
```

```
for i in G:
    for j,c in G[i]:
        print(i,j,c)
```

```
1 2 50
1 3 30
1 4 100
1 5 10
3 2 5
4 2 20
4 3 50
5 4 10
```

Representación de grafos: tablas asociativas

Tabla asociativa con la función de coste/pertenencia de las aristas

En Python, podemos usar una variable n con el número de vértices del grafo (implícitamente, los vértices se representan mediante enteros) y un diccionario c indexado por tuplas de enteros (i, j), donde i y j representan vértices, con la información relativa a cada arista.

Si el grafo es ponderado, el valor asociado a cada arista es el coste de la conexión.

Si el grafo es no dirigido, el orden de los vértices en la tupla es irrelevante y sólo se almacenan las conexiones en un sentido (por ejemplo, sólo las tuplas (i, j) tales que $i < j$):

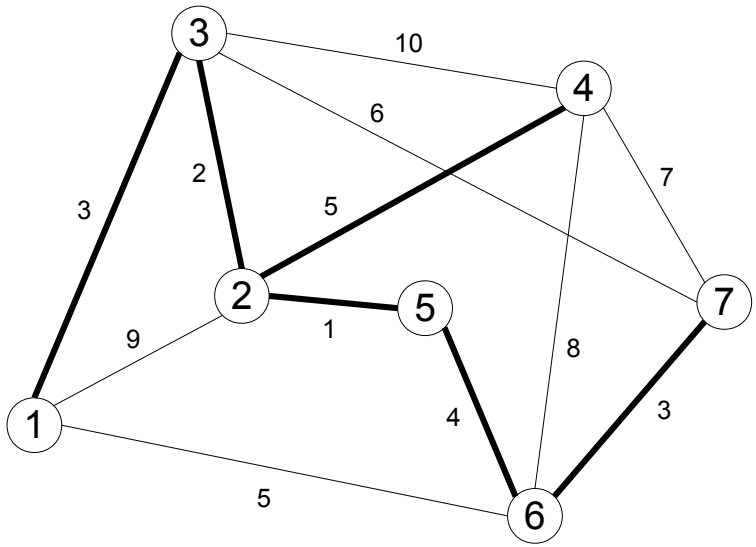
```
c = { (1,2) : 50, (1,3) : 30, (1,4) : 100, (1,5) : 10,
      (2,3) : 5, (2,4) : 20, (3,4) : 50, (4,5) : 10 }

for arista in c:
    if 2 in arista:
        print(arista,c[arista])
```

```
(1, 2) 50
(2, 3) 5
(2, 4) 20
```


Grafos: árboles de recubrimiento de coste mínimo

- Dado un grafo G no dirigido y conexo, un árbol de recubrimiento de G es un subgrafo conexo acíclico (es decir, un *árbol*) que contiene todos los vértices de G.
- Si el grafo es ponderado, cada arista tiene asignado un valor, que denominaremos *coste* y que habitualmente es mayor o igual que 0. En tal caso, el coste total de un árbol de recubrimiento se obtiene sumando los costes de las aristas que lo forman.

Un **árbol de recubrimiento de coste mínimo** sobre un grafo G es, como su propio nombre indica, aquel cuyo coste total es menor o igual que el de los demás árboles de recubrimiento que pueden definirse sobre G.



Grafos: árboles de recubrimiento de coste mínimo

Ejemplo de aplicación

Una compañía telefónica tiende cable en un nuevo vecindario

- Sólo puede tender cable por ciertas rutas, por lo que define un grafo que representa dichas rutas (que serán las aristas) y sus intersecciones (que serán los vértices).
- El coste de cada ruta vendrá dado por su longitud o por otros factores físicos o económicos.
- Un árbol de recubrimiento para ese grafo es un subconjunto de rutas sin ciclos que permite alcanzar y conectar todas las zonas habitadas.
- Evidentemente, existirán muchos posibles árboles de recubrimiento. Entre ellos, habrá uno o varios cuyo coste sea mínimo.
- Si no hay dos aristas con el mismo coste, existirá un único árbol de recubrimiento de coste mínimo (se puede demostrar fácilmente por inducción). Esto ocurre frecuentemente en la práctica, pues es extraño que dos rutas tengan exactamente el mismo coste.

Grafos: árboles de recubrimiento de coste mínimo

Enunciado: Considerese un grafo no dirigido, ponderado y conexo $G = (V, E, c)$, donde $c : E \to \mathbb{R}^+$ es la función que asigna a cada arista (u, v) un coste $c(u, v) \geq 0$. Se trata de escribir un algoritmo que obtenga el árbol de recubrimiento de G de coste mínimo.

- **Algoritmo de Prim**. Inicializa el conjunto de vértices visitados con un vértice cualquiera y el conjunto solución se inicializa vacío. En cada paso, escoge la arista de menor peso tal que uno de sus vértices ya ha sido visitado y el otro no. La arista elegida se añade a la solución y el segundo vértice se añade al conjunto de vértices visitados. Así hasta que todos los vértices se han visitado.

- **Algoritmo de Kruskal**. Tanto el conjunto solución como el conjunto de vértices visitados se inicializan vacíos y las aristas se colocan en una lista en orden creciente de coste. En cada paso, se añade al conjunto solución la siguiente arista de la lista tal que al menos uno de sus vértices no ha sido visitado, y se añaden al conjunto de vértices visitados los vértices de la nueva arista que no estaban ya en él. Así hasta que todos los vértices se han visitado.

Grafos: árboles de recubrimiento de coste mínimo

- Esquema general
  - Los candidatos son las aristas de G.
  - Un conjunto de aristas es *factible* si no contiene ningún ciclo.
  - Un conjunto de aristas factible es *solución* si contiene todos los vértices de G.
  - La función de selección varía con el algoritmo.
  - La función objetivo que hay que minimizar es la suma del coste de las aristas de la solución.
- Diremos que un conjunto de aristas factible es *prometedor* si añadiéndole más aristas es posible obtener un árbol de recubrimiento de coste mínimo.
- Diremos que una arista *sale* de un conjunto de vértices si uno y sólo uno de sus extremos está en dicho conjunto de vértices.
- Tendremos en cuenta también ciertas *propiedades de los árboles*:
  - Un árbol con n nodos tiene exactamente $n - 1$ aristas.
  - Si se añade una nueva arista entre dos de los vértices de un árbol, el grafo resultante contiene un ciclo (y por tanto ya no es un árbol).

Grafos: árboles de recubrimiento de coste mínimo

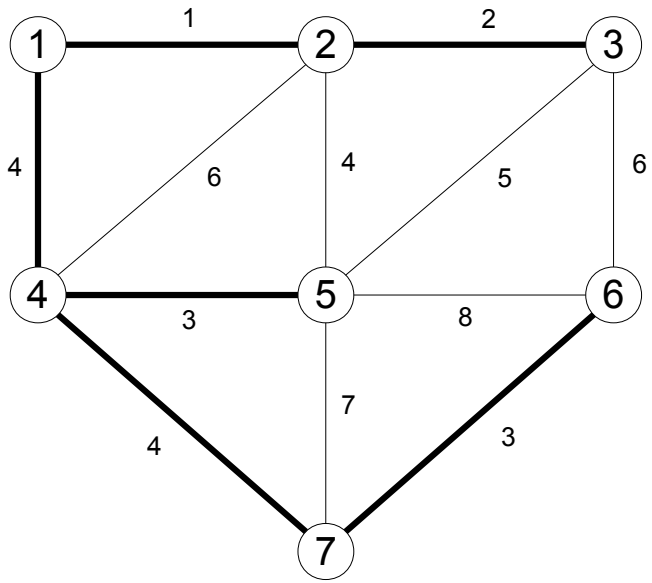
Lema Sea $G = (V, E, c)$ un grafo no dirigido, ponderado y conexo. Sea $B \subset V$ un subconjunto estricto de los vértices de G. Sea $T \subseteq E$ un conjunto de aristas prometedor, tal que no hay ninguna arista de T que salga de B. Sea e una arista de coste mínimo que sale de B. Entonces el conjunto $T \cup \{e\}$ es prometedor.

Demostración:

- Puesto que T es prometedor, existe U: árbol de recubrimiento de coste mínimo de G, tal que $T \subseteq U$.
- Si $e \in U$, no hay nada que probar.
- Si $e \notin U$, al añadir e a U se creará un ciclo. Puesto que e sale de B, U debe contener necesariamente otra arista f que salga de B, o el ciclo no se cerraría.
- Por tanto, si eliminamos f, el ciclo desaparece y obtenemos un nuevo árbol W que recubre G.
- Por definición, el coste de e es menor o igual que el de f; por tanto, el coste de W será, a su vez, menor o igual que el de U. De ahí se deduce que W es un árbol de recubrimiento de coste mínimo de G, y además contiene a e.
- La demostración se completa observando que $T \subseteq W$, ya que la arista f que se ha eliminado de U también salía de B y, por tanto, no podía formar parte de T.

Algoritmo de Kruskal

Para construir el árbol de recubrimiento de coste mínimo de un grafo, el algoritmo de Kruskal recorre las aristas en orden creciente de costes.



Paso	Arista considerada	Arista aceptada	Componentes conexas
Inicialización	–	–	{1}{2}{3}{4}{5}{6}{7}
1	{1, 2}	Sí	{1, 2}{3}{4}{5}{6}{7}
2	{2, 3}	Sí	{1, 2, 3}{4}{5}{6}{7}
3	{4, 5}	Sí	{1, 2, 3}{4, 5}{6}{7}
4	{6, 7}	Sí	{1, 2, 3}{4, 5}{6, 7}
5	{1, 4}	Sí	{1, 2, 3, 4, 5}{6, 7}
6	{2, 5}	No	{1, 2, 3, 4, 5}{6, 7}
7	{4, 7}	Sí	{1, 2, 3, 4, 5, 6, 7}

Algoritmo de Kruskal

Implementación en lenguaje Python

```
# n: número de vértices del grafo
# c: diccionario de costes, indexado por aristas: tuplas (i,j)
# Se suponen definidas las funciones argmin() y componente()

def Kruskal(n,c):
    T = []                          # inicialización del árbol de recubrimiento
    B = [[i] for i in range(n)]     # inicialmente: n componentes conectadas,
                                    # cada una con un vértice
    while len(T) < n-1:
        e = argmin(c)               # se extrae la arista de coste mínimo:
        c.pop(e)
        i = componente(B,e[0])      # componente donde está el vértice e[0]
        j = componente(B,e[1])      # componente donde está el vértice e[1]
        if i != j:
            B[i].extend(B[j])       # se fusionan las componentes
            B.pop(j)
            T.append(e)             # la arista e entra en la solución
    return T
```

Algoritmo de Kruskal

El algoritmo de Kruskal devuelve un árbol de recubrimiento de coste mínimo.

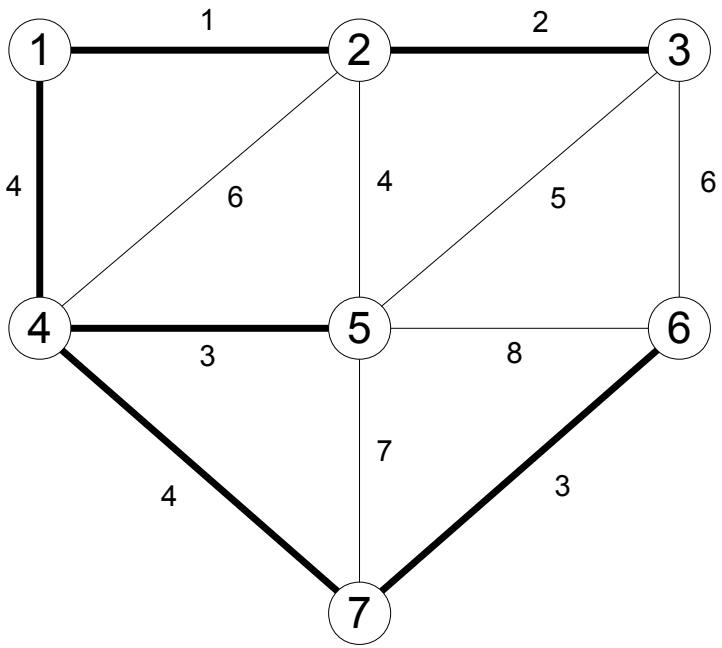
La demostración se hace por inducción sobre el número de aristas en T.
Si T es prometedor, sigue siendo prometedor tras añadir una nueva arista.
Cuando al algoritmo termina, T es la solución al problema.

- **Base:** el conjunto vacío es prometedor porque G es conexo y por tanto tiene que existir solución.
- **Hipótesis:** T es prometedor antes de añadir una nueva arista $e = (u, v)$.
- **Prueba:** Las aristas de T dividen a los nodos de G en dos o más componentes conexas. Por definición, el vértice u de la arista añadida se encuentra en una de esas componentes y el vértice v en otra. Sea B el conjunto de vértices de la componente que contiene a u:
  - El conjunto B es un subconjunto estricto de los vértices de G y no contiene a v.
  - T es un conjunto prometedor tal que ninguna arista de T sale de B.
  - La arista e tiene coste mínimo entre las aristas que salen de B.

  Esto hace que se cumplan las condiciones del lema demostrado anteriormente, y por tanto el conjunto $T \cup \{e\}$ es prometedor.

Algoritmo de Prim

El algoritmo de Prim construye el árbol de recubrimiento de coste mínimo añadiendo un nuevo vértice a cada paso, comenzando por un vértice arbitrario.



Paso	Arista elegida	B
Inicialización	–	{1}
1	{1, 2}	{1, 2}
2	{2, 3}	{1, 2, 3}
3	{1, 4}	{1, 2, 3, 4}
4	{4, 5}	{1, 2, 3, 4, 5}
5	{4, 7}	{1, 2, 3, 4, 5, 7}
6	{7, 6}	{1, 2, 3, 4, 5, 6, 7}

Algoritmo de Prim

Implementación en lenguaje Python

```
# n: número de vértices del grafo
# c: diccionario de costes, indexado por aristas: tuplas (i,j)
# Se supone definida la función argmin()

def Prim(n,c):
    T = []                  # inicialización del árbol de recubrimiento
    B = [0]                 # B se inicializa con un vértice cualquiera
    while len(B) < n:
        e = argmin(B,n,c) # e: arista de menor coste t.q. e[0]∈B y e[1]∉B
        B.append(e[1])      # el vértice e[1] entra en B
        T.append(e)         # la arista e entra en la solución
    return T
```

Algoritmo de Prim

El algoritmo de Prim devuelve un árbol de recubrimiento de coste mínimo.

La demostración se hace por inducción sobre el número de aristas en T.
Si T es prometedor, sigue siendo prometedor tras añadir una nueva arista.
Cuando al algoritmo termina, T es la solución al problema.

- **Base:** El conjunto vacío ($T = \emptyset$) es prometedor.
- **Hipótesis:** T es prometedor antes de añadir una nueva arista $e = (u, v)$.
- **Prueba:** Sea $\bar{B} = V - B$ el conjunto de vértices de G que no están en B.
  - $\bar{B}$ es un subconjunto estricto de los vértices de G (ya que B lo es).
  - T es un conjunto prometedor tal que ninguna arista de T sale de $\bar{B}$.
  - La arista e tiene coste mínimo entre las aristas que salen de $\bar{B}$ y conectan con B.

  Esto hace que se cumplan las condiciones del lema demostrado anteriormente, y por tanto el conjunto $T \cup \{e\}$ es prometedor.

El problema del viajante

Problema: Se conocen las distancias entre un cierto número de ciudades. Un viajante debe visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Enunciado formal: Dado un grafo G no dirigido, ponderado, conexo y completo, y un vértice v_0, encontrar el *ciclo hamiltoniano* de coste mínimo que comienza y termina en v_0.

Algoritmo voraz propuesto

- Sea $C(v_0, v)$ el camino construido hasta el momento, que comienza en v_0 y termina en v. Si $C(v_0, v)$ contiene todos los vértices de G, el algoritmo añade la arista (v, v_0) (para cerrar el ciclo) y termina. Si no, el algoritmo añade la arista (v, w) de coste mínimo desde v a vértices w que no estén en el camino. Inicialmente $C = \emptyset$ y $v = v_0$.
- Cualquiera que sea la forma de seleccionar la siguiente arista (v, w) del camino, deberá cumplir las siguientes condiciones: (1) no formar un ciclo con las aristas ya seleccionadas; y (2) no incidir por tercera vez en uno de los vértices ya recorridos.

El problema del viajante

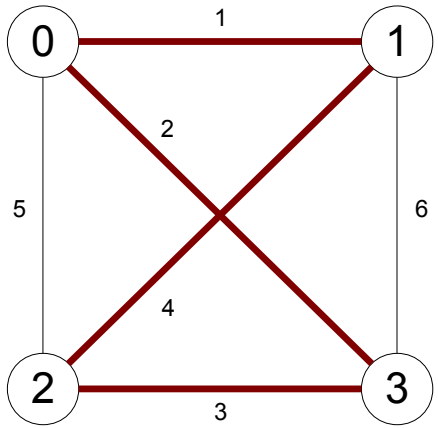
Algoritmo voraz: implementación en Python

```
# n: número de vértices del grafo
# c: diccionario de costes, indexado por aristas (i,j) con i<j
def viajante(n,c,v0=0):
    T = []
    B = [v0]
    v = v0
    while len(B) < n:
        minimo = 1e+100      # coste infinito
        for j in range(n):
            # las aristas aparecen en c como tuplas (i,j) con i<j
            coste = c[(min(v,j),max(v,j))]
            if j not in B and coste < minimo:
                minimo = coste
                w=j
        B.append(w)
        T.append((v,w))
        v=w
    T.append((v,v0))
    return T
```

El problema del viajante

Ejemplo de aplicación 1

```
n = 4

c = { (0,1): 1, (0,2): 5, (0,3): 2,
      (1,2): 4, (1,3): 6, (2,3): 3 }
```

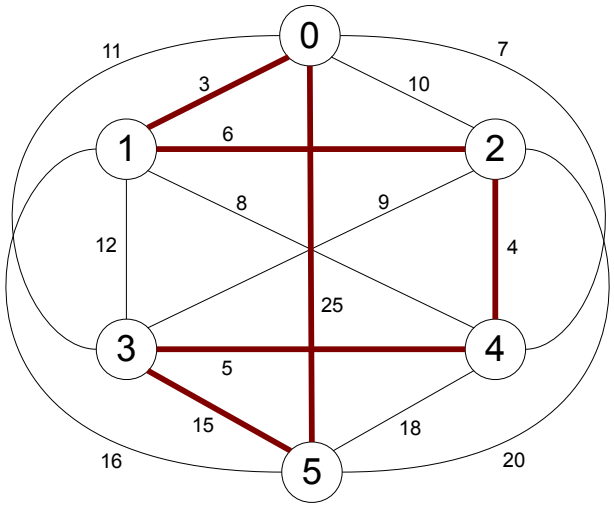


Partiendo del vértice 0, **el algoritmo encuentra una solución óptima**, que está formada por las aristas (0,1), (1,2), (2,3) y (3,0), lo que da lugar al ciclo (0, 1, 2, 3, 0), cuyo coste es $1 + 4 + 3 + 2 = 10$, que es óptimo, ya que el resto de soluciones arrojan costes iguales o superiores: 15, 17, 14, 17 y 10.

El problema del viajante

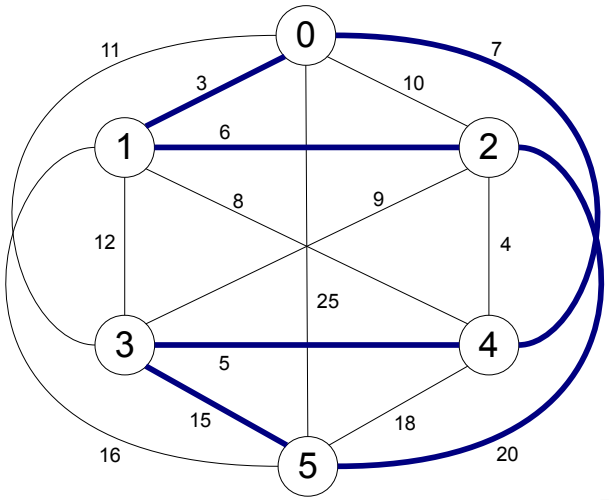
Ejemplo de aplicación 2

```
n = 6

c = { (0,1): 3, (0,2): 10, (0,3): 11,
      (0,4): 7, (0,5): 25, (1,2): 6,
      (1,3): 12, (1,4): 8, (1,5): 16,
      (2,3): 9, (2,4): 4, (2,5): 20,
      (3,4): 5, (3,5): 15, (4,5): 18 }
```



Partiendo del vértice 0, el algoritmo escoge la secuencia de aristas (0,1), (1,2), (2,4), (4,3), (3,5) y (5,0), que da lugar al ciclo (0, 1, 2, 4, 3, 5, 0), cuyo coste es 58.

Sin embargo, el camino definido por las aristas (0,1), (1,2), (2,5), (5,3), (3,4) y (4,0) tiene un coste menor (56).



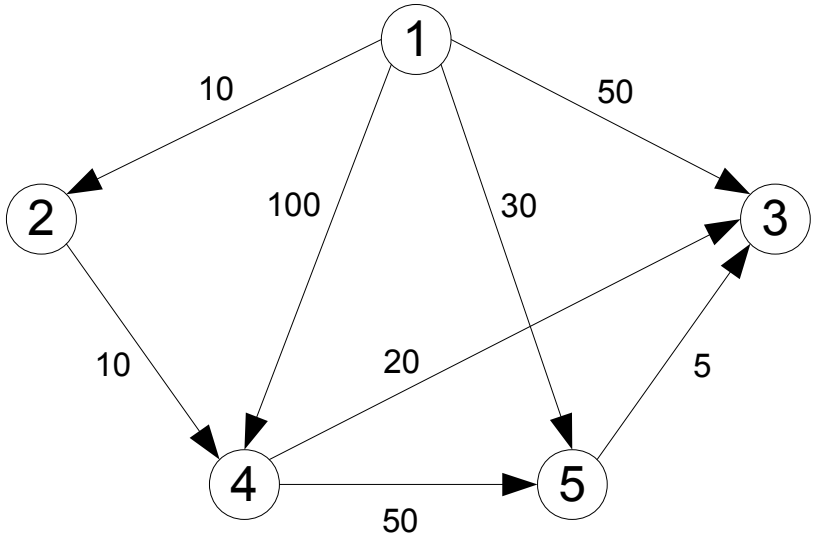
Caminos de coste mínimo: algoritmo de Dijkstra

Enunciado: Sea $G = (V, E, c)$ un grafo **dirigido** y ponderado. Uno de sus vértices, que llamaremos v_0, será el *vértice origen*. Se trata de encontrar el camino de coste mínimo desde v_0 a cualquier otro vértice v_i de G.

Solución: algoritmo de Dijkstra

- C: conjunto de vértices para los que aún no se ha calculado el camino de coste mínimo desde v_0. Inicialización: $C = V - \{v_0\}$.
- S: conjunto de vértices para los que ya se ha calculado el camino de coste mínimo desde v_0. En todo momento, se tiene: $S = V - C$.
- Un camino desde v_0 hasta un vértice $v \in C$ se dice que es **especial** si todos los vértices intermedios pertenecen a S.
- En cada iteración, el algoritmo calcula y almacena en un vector D el coste y en otro vector P el último vértice de S en el camino especial de coste mínimo a cada vértice del grafo. Inicialización: $D[v_i] = c(v_0, v_i)$ y $P[v_i] = v_0$.
- En cada iteración, se añade a S el vértice de C con valor mínimo en D: w. D se actualiza haciendo $D[v] = \text{mín}\,(D[v], D[w] + c(w, v))$ para todo $v \in C$; además, si $D[w] + c(w, v) < D[v]$, se hace $P[v] = w$.
- El algoritmo termina cuando todos los vértices del grafo están en S.

Caminos de coste mínimo: algoritmo de Dijkstra



Paso	w	S	D	P
0	–	{1}	[0, 10, 50, 100, 30]	[1, 1, 1, 1, 1]
1	2	{1, 2}	[0, 10, 50, 20, 30]	[1, 1, 1, 2, 1]
2	4	{1, 2, 4}	[0, 10, 40, 20, 30]	[1, 1, 4, 2, 1]
3	5	{1, 2, 4, 5}	[0, 10, 35, 20, 30]	[1, 1, 5, 2, 1]
4	3	{1, 2, 4, 5, 3}	[0, 10, 35, 20, 30]	[1, 1, 5, 2, 1]

Caminos de coste mínimo: algoritmo de Dijkstra

Implementación en Python

```
def dijkstra(n,c,v0):
    # D[v]: coste del camino especial optimo a v
    D = [c[(v0,v)] if v!=v0 else 0 for v in range(n)]
    # P[v]: vertice anterior en el camino especial optimo a v
    P = [v0 for v in range(n)]
    C = [v for v in range(n) if v!=v0]
    while len(C) > 0:
        minimo = 1e+100 # infinito
        for v in C:
            if D[v] < minimo:
                minimo = D[v]
                w=v
        C.remove(w)
        for v in C: # actualizacion de D y P
            if (w,v) in c and D[w] + c[(w,v)] < D[v]:
                D[v] = D[w] + c[(w,v)]
                P[v] = w
    return D,P
```

Caminos de coste mínimo: algoritmo de Dijkstra

El algoritmo de Dijkstra halla los caminos de coste mínimo desde un vértice origen fijo v_0 a todos los demás vértices del grafo

Demostraremos por inducción matemática que:

(a) $v_i \in S$ $(v_i \neq v_0)$ $\Rightarrow$ $D[i]$ es el coste del camino óptimo desde v_0 a v_i; y

(b) $v_i \notin S$ $\Rightarrow$ $D[i]$ es el coste del camino especial óptimo desde v_0 a v_i.

Demostración

- **Base:** Inicialmente $S = \{v_0\}$. El único camino especial a cualquier nodo v_i es el directo, y D se inicializa con esos costes (y $P[v_i] = v_0$ $\forall v_i$).
- **Hipótesis:** Las propiedades (a) y (b) son válidas justo antes de añadir a S el vértice w que minimiza $D[v]$ entre los vértices $v \in C$.

(continúa...)

Caminos de coste mínimo: algoritmo de Dijkstra

(...viene de la página anterior)

(...y continúa en la siguiente)

- **Prueba de la propiedad (a):**
  - Si $v_i \in S$ antes de añadir w a S, $v_i \in S$ también después de hacerlo, por lo que la propiedad (a) seguirá siendo válida.
  - No obstante, antes de añadir w a S, es preciso comprobar que $D[w]$ es efectivamente el coste del camino óptimo de v_0 a w.
  - La hipótesis de inducción dice que $D[w]$ es el coste del camino especial óptimo de v_0 a w. Por tanto, basta comprobar que el camino óptimo de v_0 a w no incluye ningun vértice que no pertenezca a S.
  - Procederemos por reducción al absurdo. Supongamos que el camino óptimo de v_0 a w incluye uno o más vértices que no pertenecen a S.
  - Sea u el primero de esos vértices. El tramo del camino que lleva desde v_0 a u es un camino especial y su coste, aplicando la hipótesis de inducción (propiedad (b)), será $D[u]$.
  - Claramente, el coste del camino que va de v_0 a w a través de u es mayor o igual que $D[u]$, es decir, $D[w] \geq D[u]$, ya que los costes de los arcos son positivos o nulos.
  - Por otra parte, $D[u] \geq D[w]$, ya que $w = \arg\min_{\forall v \in C}(D(v))$.
  - Por tanto, el coste del camino que va de v_0 a w a través de u es, como mínimo, igual a $D[w]$ y no mejora el del camino especial óptimo de v_0 a w.

Caminos de coste mínimo: algoritmo de Dijkstra

(...viene de la página anterior)

- **Prueba de la propiedad (b):**
  - Considérese un nodo $u \notin S$, distinto de w. Cuando w se añade a S, hay dos posibilidades para el camino especial de coste mínimo desde v_0 hasta u: o bien no cambia, o bien pasa a través de w (y posiblemente, también a través de otros vértices de S).
  - En el segundo caso, sea x el último nodo de S visitado antes de llegar a u. El coste de ese camino es $D[x] + c(x, u)$.
  - A primera vista, podría parecer que para calcular el nuevo valor de $D[u]$ deberíamos comparar el valor actual de $D[u]$ con el de todos los posibles caminos a través de S: $D[x] + c(x, u)$ para todo $x \in S$. Sin embargo, dicha comparación ya fue realizada en el momento de añadir x a S, para todo $x \neq w$, y $D[x]$ no ha variado desde entonces. Por tanto, basta comparar el valor actual de $D[u]$ con $D[w] + c(w, u)$ y quedarse con el mínimo.
  - Puesto que el algoritmo hace precisamente eso, la propiedad (b) seguirá siendo cierta después de añadir a S un nuevo vértice w.