Programa de teoría

ALGORÍTMICA

1. Análisis de algoritmos.
2. Divide y vencerás.
3. **Algoritmos voraces.**
4. Programación dinámica.
5. Backtracking.
6. Ramificación y poda.

ALGORÍTMICA

Tema 3. Algoritmos voraces.

3.1. Método general.

3.2. Análisis de tiempos de ejecución.

3.3. Ejemplos de aplicación.

- 3.3.1. Problema de la mochila.
- 3.3.2. Planificación de tareas.

3.4. Heurísticas voraces.

- 3.4.1. El problema del viajante.
- 3.4.2. Coloración de grafos.

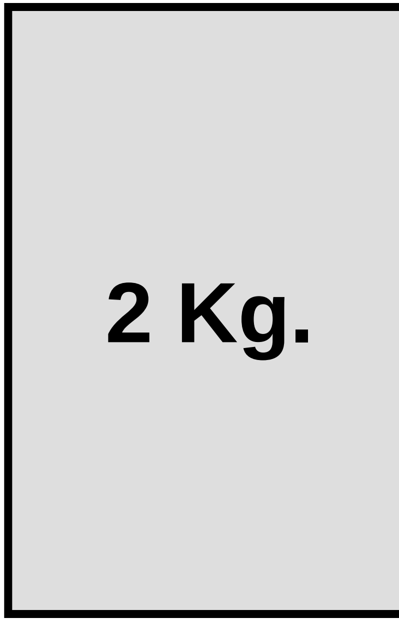
3.1. Método general.

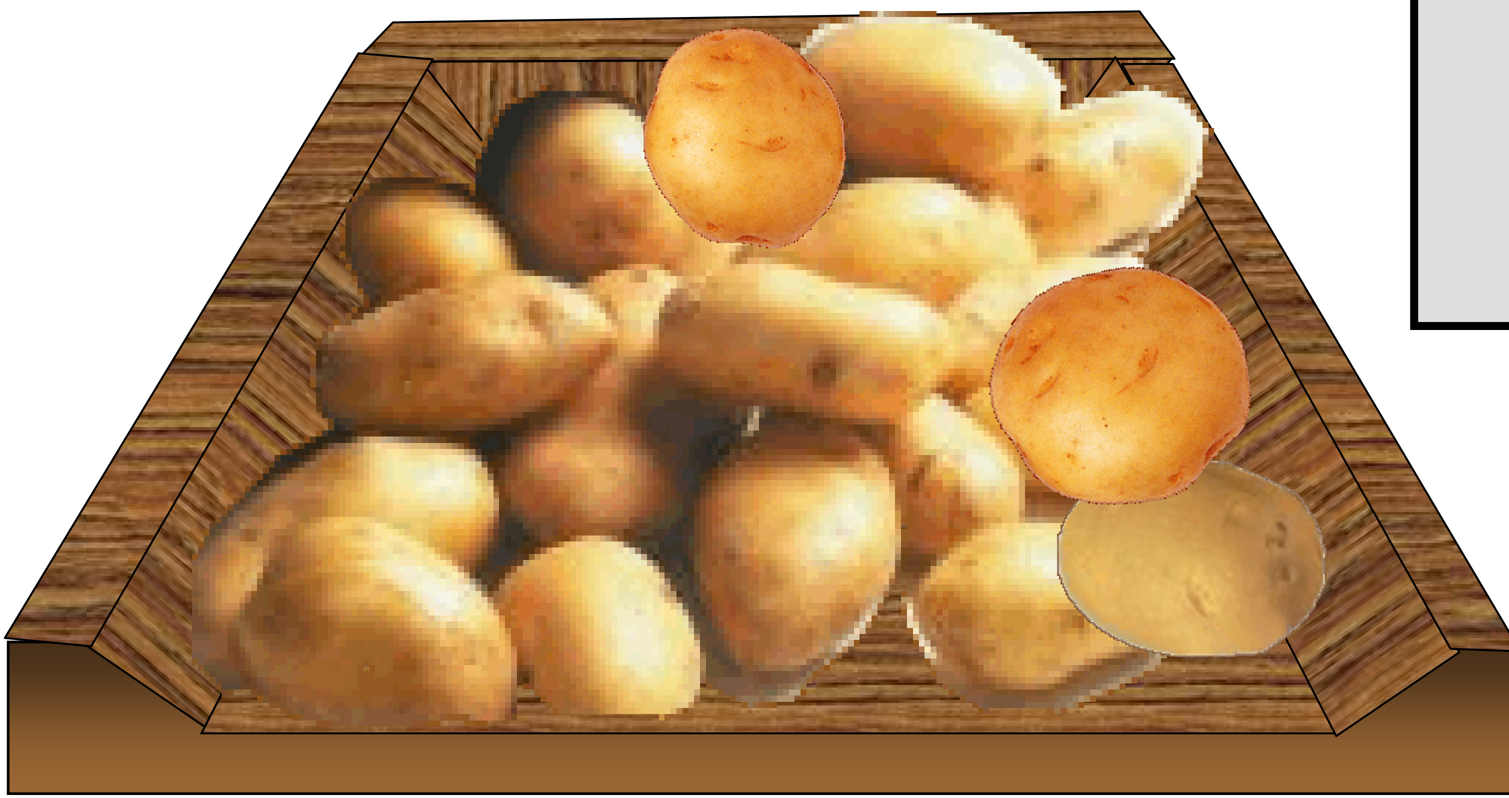
- Los algoritmos **voraces**, **ávidos** o de **avance rápido** (en inglés **greedy**) se utilizan normalmente en problemas de optimización.
  - El problema se ***interpreta*** como: “tomar algunos elementos de entre un conjunto de candidatos”.
  - El **orden** el que se cogen puede ser importante o no.
- Un **algoritmo voraz** funciona por pasos:
  - Inicialmente partimos de una solución vacía.
  - En cada paso se escoge el siguiente elemento para añadir a la solución, entre los candidatos.
  - Una vez tomada esta decisión no se podrá deshacer.
  - El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.

3.1. Método general.

- **Ejemplo:** “el viejo algoritmo de comprar patatas en el mercado”.

¿Sí o no?





3.1. Método general.

Características básicas del algoritmo:

- Inicialmente empezamos con una solución “vacía”, sin patatas.
- Función de selección: seleccionar la mejor patata del montón o la que “parezca” que es la mejor.
- Examinar la patata detenidamente y decidir si se coge o no.
- Si no se coge, se aparta del montón.
- Si se coge, se mete a la bolsa (y ya no se saca).
- Una vez que tenemos 2 kilos paramos.

3.1. Método general.

- Se puede generalizar el proceso intuitivo a un esquema algorítmico general.

- El esquema trabaja con los siguientes conjuntos de elementos:
  - **C:** Conjunto de elementos **candidatos, pendientes** de seleccionar (inicialmente todos).
  - **S:** Candidatos seleccionados para la **solución**.
  - **R:** Candidatos seleccionados pero **rechazados** después.

- ¿Qué o cuáles son los candidatos? Depende de cada problema.

3.1. Método general.

- **Esquema general de un algoritmo voraz:**

voraz (C: CjtoCandidatos; var S: CjtoSolución)

S:= Ø
mientras (C $\neq$ Ø) Y NO **solución**(S) hacer
    x:= **seleccionar**(C)
    C:= C - {x}
    si **factible**(S, x) entonces
        insertar(S, x)
    finsi
finmientras
si NO **solución**(S) entonces
    devolver “No se puede encontrar solución”
finsi

3.1. Método general.

Funciones genéricas

- **solución (S).** Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- **seleccionar (C)**. Devuelve el elemento más “prometedor” del conjunto de candidatos pendientes (no seleccionados ni rechazados).
- **factible (S, x)**. Indica si a partir del conjunto **S** y añadiendo **x**, es posible construir una solución (posiblemente añadiendo otros elementos).
- **insertar (S, x)**. Añade el elemento **x** al conjunto solución. Además, puede ser necesario hacer otras cosas.
- Función **objetivo (S)**. Dada una solución devuelve el coste asociado a la misma (resultado del problema de optimización).

3.2. Análisis de tiempos de ejecución.

- El orden de complejidad de ... del número de candidatos, de ... del número d...
- **n**: núm... ntos de un...
- Repe...
  - Con... Norm...
  - Se... **g(n)**.
    O(1) y O(n).
  - La función **factible** es parecida a **solución**, pero con una solución parcial **h(m)**.
  - La unión de un... ión puede requerir otras c... **m)**.

¡EH! ¡¡QUE SE VA SIN PAGAR LAS PATATAS!!

¡...!

3.1. Método general.

SON 1,11 EUROS

AHÍ VAN 5 EUROS

TOMA EL CAMBIO,
3,89 EUROS

- **Problema del cambio de monedas.**

Construir un algoritmo que dada una cantidad **P** devuelva esa cantidad usando el menor número posible de monedas.

Disponemos de monedas con valores de 1, 2, 5, 10, 20 y 50 céntimos de euro, 1 y 2 euros (€).

3.1. Método general.

- **Caso 1.** Devolver 3,89 Euros.
  1 moneda de 2€, 1 moneda de 1€, 1 moneda de 50 c€, 1 moneda de 20 c€, 1 moneda de 10 c€ , 1 moneda de 5 c€ y 2 monedas de 2 c€. Total: 8 monedas.

- El método intuitivo se puede entender como un **algoritmo voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta llegar a **P**.

3.1. Método general.

Problema del cambio de monedas

- **Conjunto de candidatos**: todos los tipos de monedas disponibles. Supondremos una cantidad ilimitada de cada tipo.
- **Solución**: conjunto de monedas que sumen **P**.
- **Función objetivo:** minimizar el número de monedas.

Representación de la solución:

- **(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)**, donde x_i es el número de monedas usadas de tipo **i**.
- Suponemos que la moneda **i** vale c_i.
- **Formulación:** Minimizar $\sum_{i=1..8} x_i$, sujeto a $\sum_{i=1..8} x_i \cdot c_i = P$, $x_i \geq 0$

3.1. Método general.

Funciones del esquema:

- **inicialización.** Inicialmente $x_i = 0$, para todo i= 1..8
- **solución**. El valor actual es solución si $\sum x_i \cdot c_i = P$
- **seleccionar**. ¿Qué moneda se elige en cada paso de entre los candidatos?
- **Respuesta:** elegir en cada paso la moneda de valor más alto posible, pero sin sobrepasar la cantidad que queda por devolver.
- **factible**. Valdrá siempre verdad.
- En lugar de seleccionar monedas de una en una, usamos la división entera y cogemos todas las monedas posibles de mayor valor.

3.1. Método general.

- **Implementación.** Usamos una variable local **act** para acumular la cantidad devuelta hasta este punto.
- Suponemos que las monedas están ordenadas de menor a mayor valor.

DevolverCambio (P: entero; C: array [1..n] de entero;
var X: array [1..n] de entero)

```
act:= 0
j:= n
para i:= 1,...,n hacer
    X[i]:= 0
mientras act ≠ P hacer
    mientras (C[j] > (P - act)) AND (j>0) hacer j:= j - 1
    si j==0 entonces devolver "No existe solución"
    X[j]:= ⌊(P - act) / C[j]⌋
    act:= act + C[j]*X[j]
finmientras
```

inicialización

no solución(X)

seleccionar(C,P,X)

no factible(j)

insertar(X,j)

3.1. Método general.

- ¿Cuál es el orden de complejidad del algoritmo?
- ¿Garantiza siempre la solución óptima?
- Para este sistema monetario sí. Pero no siempre...

- **Ejemplo.** Supongamos que tenemos monedas de 100, 90 y 1. Queremos devolver 180.

  – **Algoritmo voraz**. 1 moneda de 100 y 80 monedas de 1: total 81 monedas.

  – **Solución óptima**. 2 monedas de 90: total 2 monedas.

3.2. Análisis de tiempos de ejecución.

- El orden de complejidad depende de:
  - El número de candidatos existentes.
  - Los tiempos de las funciones básicas utilizadas.
  - El número de elementos de la solución.
  - ...

- **Ejemplo. n**: número de elementos de C. **m**: número de elementos de una solución.
- Repetir, como máximo **n** veces y como mínimo **m**:
  - Función solución: **f(m)**. Normalmente O(1) ó O(m).
  - Función de selección: **g(n)**. Entre O(1) y O(n).
  - Función **factible** (parecida a **solución**, pero con una solución parcial): **h(m)**.
  - Inserción de un elemento: **j(n, m)**.

3.2. Análisis de tiempos de ejecución.

- Tiempo de ejecución **genérico**:
  t(n,m) $\in$ O(n*(f(m)+g(n)+h(m)) + m*j(n, m))

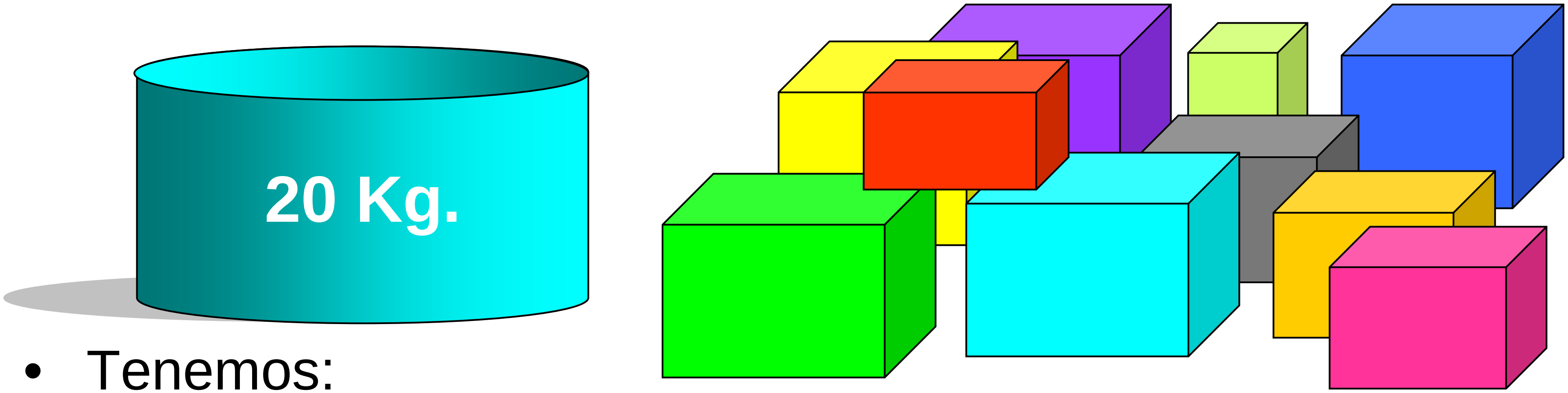
- Ejemplo:
  – Devolución de monedas (ordenando las monedas): O(n).

- El análisis depende de cada algoritmo concreto.

- En la práctica los algoritmos voraces **suelen ser** bastante **rápidos**, encontrándose dentro de órdenes de complejidad **polinomiales**.

3.3. Ejemplos de aplicación.

3.3.1. Problema de la mochila.



- Tenemos:
  - **n** objetos, cada uno con un peso ($\mathbf{p_i}$) y un beneficio ($\mathbf{b_i}$)
  - Una mochila en la que podemos meter objetos, con una capacidad de peso máximo **M**.
- **Objetivo:** llenar la mochila, maximizando el beneficio de los objetos transportados, y respetando la limitación de capacidad máxima **M**.
- Los objetos se pueden partir en trozos.

3.3.1. Problema de la mochila.

Datos del problema:

- **n**: número de objetos disponibles.
- **M**: capacidad de la mochila.
- $\mathbf{p = (p_1, p_2, ..., p_n)}$ pesos de los objetos.
- $\mathbf{b = (b_1, b_2, ..., b_n)}$ beneficios de los objetos.

Representación de la solución:

- Una solución será de la forma $\mathbf{S = (x_1, x_2, ..., x_n)}$, con $0 \leq x_i \leq 1$, siendo cada $\mathbf{x_i}$ la fracción escogida del objeto **i**.

Formulación matemática:

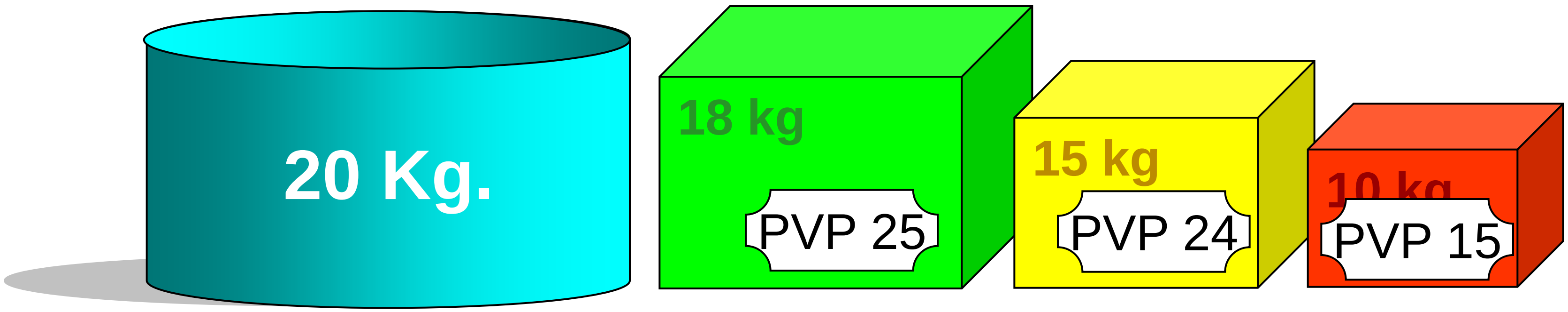
- Maximizar $\sum_{i=1..n} x_i b_i$; sujeto a la restricción $\sum_{i=1..n} x_i p_i \leq M$, y $0 \leq x_i \leq 1$

3.3.1. Problema de la mochila.

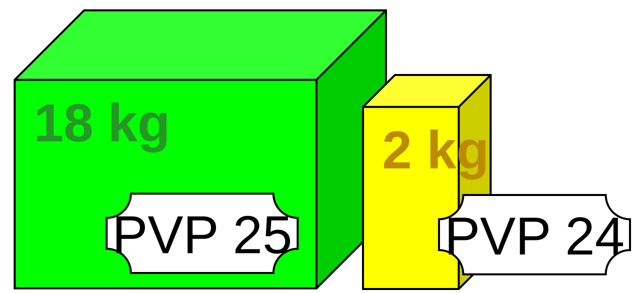
- **Ejemplo:** n = 3; M = 20

  p = (18, 15, 10)

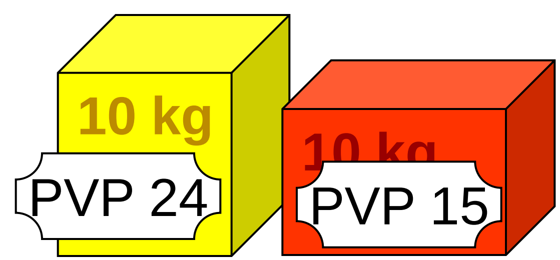
  b = (25, 24, 15)



- **Solución 1**: S = (1, 2/15, 0)
  Beneficio total = 25 + 24*2/15 = 28,2



- **Solución 2**: S = (0, 2/3, 1)
  Beneficio total = 15 + 24*2/3 = 31



3.3.1. Problema de la mochila.

- El problema se ajusta bien a la idea de algoritmo voraz.
- **Diseño de la solución:**
  - **Candidatos**: Cada uno de los **n** objetos de partida.
  - Función **solución**: Tendremos una solución si hemos introducido en la mochila el peso máximo **M**, o si se han acabado los objetos.
  - Función **seleccionar**: Escoger el objeto más prometedor.
  - Función **factible**: Será siempre cierta (podemos añadir trozos de objetos).
  - **Añadir** a la solución: Añadir el objeto entero si cabe, o en otro caso la proporción del mismo que quede para completarla.
  - Función **objetivo**: Suma de los beneficios de cada candidato por la proporción seleccionada del mismo.
- Queda por definir la función de selección. ¿Qué criterio podemos usar para seleccionar el objeto más prometedor?

3.3.1. Problema de la mochila.

```
Mochila (M: entero; b, p: array [1..n] de entero;
                    var X: array [1..n] de entero)
  para i:= 1, ..., n hacer
      X[i]:= 0
  pesoAct:= 0
  mientras pesoAct < M hacer
      i:= el mejor objeto restante
      si pesoAct + p[i] ≤ M entonces
          X[i]:= 1
          pesoAct:= pesoAct + p[i]
      sino
          X[i]:= (M - pesoAct)/p[i]
          pesoAct:= M
      finsi
  finmientras
```

3.3.1. Problema de la mochila.

- Posibles criterios para seleccionar el mejor objeto de los restantes:
  1. El objeto con más beneficio $\mathbf{b_i}$: $\underset{i=1,\ldots,n}{\text{argmax}}\ \mathbf{b_i}$
  2. El objeto menos pesado $\mathbf{p_i}$ (para poder añadir muchos objetos): $\underset{i=1,\ldots,n}{\text{argmin}}\ \mathbf{p_i}$
  3. El objeto con mejor proporción $\mathbf{b_i/p_i}$ (beneficio por unidad de peso): $\underset{i=1,\ldots,n}{\text{argmax}}\ \mathbf{b_i/p_i}$
- ¿Cuál es el mejor criterio de selección?
- ¿Garantiza la solución óptima?

3.3.1. Problema de la mochila.

- **Ejemplo 1:** n = 4; M = 10

  p = (10, 3, 3, 4)

  b = (10, 9, 9, 9)

  – Criterio 1: S = (1, 0, 0, 0). Beneficio total = 10
  – Criterio 2 y 3: S = (0, 1, 1, 1). Beneficio total = 27
- **Ejemplo 2:** n = 4; M = 10

  p = (10, 3, 3, 4)

  b = (10, 1, 1, 1)

  – Criterio 1 y 3: S = (1, 0, 0, 0). Beneficio total = 10
  – Criterio 2: S = (0, 1, 1, 1). Beneficio total = 3
- Los criterios 1 y 2 pueden dar soluciones no muy buenas.
- El criterio 3 garantiza siempre una solución óptima.

3.3.1. Problema de la mochila.

- **Demostración** (por reducción al absurdo):
  Supongamos que tenemos una solución óptima $x=(x_1, x_2, ..., x_n)$, que incluye un objeto **i**, pero no incluye (o incluye con menor proporción) otro objeto **j** con mejor proporción: $(x_i > x_j)$ y $(b_i/p_i < b_j/p_j)$.
- Si quitamos un trozo de peso de **i** y lo metemos de **j** entonces obtendríamos más beneficio. Por ejemplo, si quitamos un peso **r**, con $0 < r \leq x_i \cdot p_i$, $r \leq (1-x_j) \cdot p_j$:

$$b_{NUEVO} = b_{ANTIGUO} – r\ b_i/p_i + r\ b_j/p_j =$$
$$b_{ANTIGUO} + r\ (b_j/p_j - b_i/p_i) > b_{ANTIGUO}$$

- ¿Cuánto es el orden de complejidad del algoritmo?
- ¿Cómo calcular el beneficio total?
- ¿Qué ocurre si no se pueden partir los objetos?

3.3.2. Planificación de tareas.

Problema de secuenciamiento de trabajos con plazos:

- Tenemos un procesador y **n** tareas disponibles.
- Todas las tareas requieren 1 unidad de tiempo para ejecutarse y tienen:
  - $\mathbf{b_i}$: Beneficio obtenido si se ejecuta la tarea **i**.
  - $\mathbf{d_i}$: Plazo máximo de ejecución de la tarea **i**.

- **Significado:** la tarea **i** sólo puede ejecutarse si se hace en un instante igual o anterior a $\mathbf{d_i}$. En ese caso se obtiene un beneficio $\mathbf{b_i}$.

- En general puede que no sea posible ejecutar todas las tareas.

3.3.2. Planificación de tareas.

- **Objetivo:** dar una planificación de las tareas a ejecutar (s_1, s_2, ..., s_m) de forma que se maximice el beneficio total obtenido.

$$b_{TOTAL} = \sum_{i=1..m} b_{s_i}$$

T	1	2	3	4
S	s_1	s_2	s_3	s_4

- **Problemas a resolver:**
  1) ¿Qué tareas ejecutar?
  2) ¿En qué orden ejecutarlas?

3.3.2. Planificación de tareas.

- **Ejemplo**: n = 4

  b = (100, 10, 15, 27)

  d = (2,   1,   2,   1)

- Posibles soluciones:

T	1	2
S	1	3
b	100	15
d	2	2

b_{TOTAL}= 115

T	1	2
S	4	3
b	27	15
d	1	2

b_{TOTAL}= 42

T	1	2
S	1	4
b	100	27
d	2	1

No factible

T	1	2
S	4	1
b	27	100
d	1	2

b_{TOTAL}= 127

- **Algoritmo sencillo:** comprobar todos los posibles órdenes de las tareas y quedarse con el mejor (y que sea factible).
- Tendría una complejidad de Ω(n!) ...

3.3.2. Planificación de tareas.

- **Aplicamos avance rápido:** empezamos con una planificación sin tareas y vamos añadiéndolas paso a paso.
- Una solución estará formada por un conjunto de candidatos, junto con un orden de ejecución de los mismos.
- **Representación de la solución:** $S = (s_1, s_2, ..., s_m)$, donde $\mathbf{s_i}$ es la tarea ejecutada en el instante **i**.
- Función de **selección**: de los candidatos restantes elegir el que tenga mayor valor de beneficio: argmax b_i.
- ¿Cómo es la función **factible (S, x)**?

3.3.2. Planificación de tareas.

x
b_x
d_x

Planifi-
cación
actual

T	1	2	3	4
S	s_1	s_2	s_3	s_4
b	b_{s_1}	b_{s_2}	b_{s_3}	b_{s_4}
d	d_{s_1}	d_{s_2}	d_{s_3}	d_{s_4}

- ¿Dónde debería ser colocada **x** dentro de la planificación?
- ¿Es factible la solución parcial que incluye a **x**?
- **Idea 1:** Probar todas las posibles colocaciones. → MAL
- **Idea 2:** Ordenar las tareas por orden de plazo d_x. Que las tareas con plazos más tempranos se ejecuten antes. → OK

3.3.2. Planificación de tareas.

- **Lema:** Sea **J** un conjunto de **k** tareas. Existe una ordenación factible de **J** (es decir que respeta los plazos) si y sólo si la ordenación S = (s_1, s_2, ..., s_k), con $d_{s_1} \leq d_{s_2} \leq \ldots \leq d_{s_k}$ es factible.

T	1		2		...		k
S	s_1		s_2		...		s_k
b	b_{s_1}		b_{s_2}		...		b_{s_k}
d	d_{s_1}	$\leq$	d_{s_2}	$\leq$	...	$\leq$	d_{s_k}

- Es decir, sólo es necesario probar la planificación en orden creciente de plazo de ejecución, comprobando que cada $\mathbf{d_{s_i}} \geq \mathbf{i}$ (la tarea ejecutada en la **i**-ésima posición tiene un plazo de **i** ó más).

3.3.2. Planificación de tareas.

- **Demostración del lema:**

  $\Leftarrow$) Si el orden **S** (antes definido) es factible entonces existe una ordenación factible de **J**: Trivial.

  $\Rightarrow$) Si existe alguna ordenación factible de **J**, **S** es factible.

  – Supongamos (por reducción al absurdo) que existe esa ordenación factible pero que **S** no es factible.

  – Entonces debe existir una tarea $\mathbf{s_r}$ tal que $\mathbf{d_{s_r}} < \mathbf{r}$.

T	1	2	r	...
S	s_1	s_2	s_r	...
d	$d_{s_1} \leq$	$d_{s_2} \leq$	$d_{s_r} \leq$	...

  – Puesto que las **r-1** tareas anteriores tienen $\mathbf{d_{si}} \leq \mathbf{d_{sr}} < \mathbf{r}$, habrán **r** tareas cuyo plazo es menor que **r**.

  – En conclusión, no puede existir ningún orden de las tareas **J**, de forma que se ejecuten dentro de su plazo → Contradicción.

3.3.2. Planificación de tareas.

Estructura del algoritmo voraz:

- **Inicialización:** Empezar con una secuencia vacía, con todas las tareas como candidatas.
- Ordenar las tareas según el valor de $\mathbf{b_i}$.
- En cada paso, hasta que se acaben los candidatos, repetir:
  - **Selección:** Elegir entre los candidatos restantes el que tenga mayor beneficio.
  - **Factible:** Introducir la nueva tarea en la posición adecuada, según los valores de plazo **d**.
    - Si el nuevo orden (s_1, s_2, ..., s_k) es tal que $\mathbf{d_{s_i}} \geq \mathbf{i}$, para todo **i** entre **1** y **k**, entonces el nuevo candidato es factible. Añadirlo a la solución.
    - En otro caso, rechazar el candidato.

3.3.2. Planificación de tareas.

- **Ejemplo**: n=6

  b = (20, 15, 10, 7, 5, 3)

  d = (3, 1, 1, 3, 1, 3)

T	1	2	3
S	2	4	1
b	15	7	20
d	1	3	3

- Es posible demostrar que este algoritmo obtiene la solución óptima.
- **Idea**: suponer una solución óptima y comprobar que tiene el mismo beneficio que la calculada por el algoritmo.

3.3.2. Planificación de tareas.

- **Orden de complejidad** del algoritmo, suponiendo **n** tareas:
  - Primero, ordenar las tareas por orden creciente de plazo: **O(n·log n)**
  - Repetir para **i** desde 1 hasta **n**:
    - Elegir el próximo candidato: **O(1)**
    - Comprobar si la nueva planificación es factible, y añadirla a la solución en caso afirmativo: **O(i)** en el peor caso.
  - En total, el algoritmo es un **O(n^2)**
- **Ejercicio:** en lugar de desplazar tareas, planificar cada tarea lo más tarde posible según su plazo y la ordenación actual.

3.4. Heurísticas voraces.

- **Problemas NP-completos:** la solución exacta puede requerir órdenes factoriales o exponenciales (el problema de la *explosión combinatoria*).
- **Objetivo:** obtener “buenas” soluciones en un tiempo de ejecución corto (*razonable*).
- **Algoritmos de aproximación:** garantizan una solución más o menos buena (o una cierta aproximación respecto al óptimo).
- Un tipo son los **algoritmos heurísticos**[1]: algoritmo basado en el conocimiento “intuitivo” o “experto” del programador sobre determinado problema.

[1]DRAE. *Heurística*: Arte de inventar.

3.4. Heurísticas voraces.

PROBLEMA COMPLEJO

+ tiempo
+ preciso

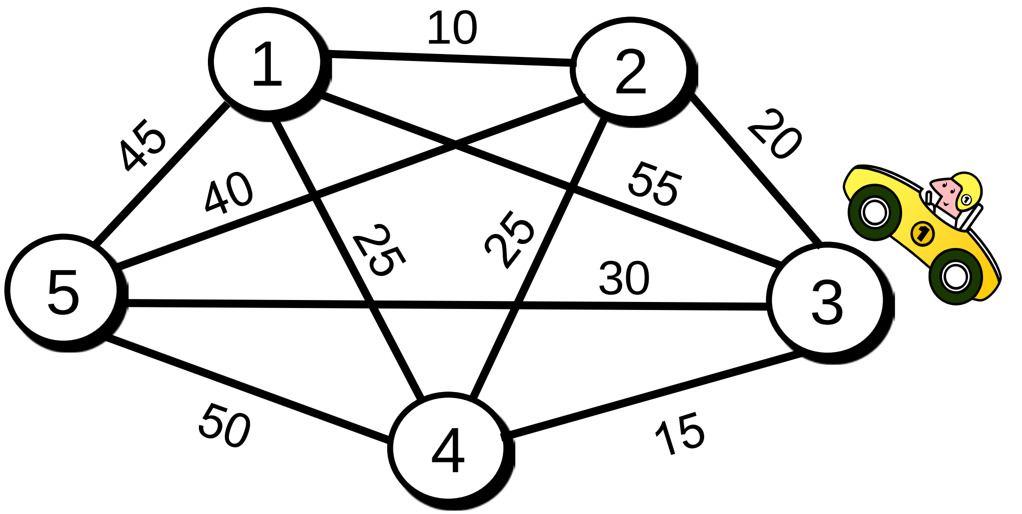
- tiempo
- preciso

Algoritmo exacto	**Algoritmo heurístico 1**	**Algoritmo heurístico 2**
Backtracking Fuerza bruta Ram. y Poda	Heur. voraz Alg. genético Alg. probabilista	Solución inmediata o fija

- La estructura de algoritmos voraces se puede utilizar para construir procedimientos heurísticos: hablamos de **heurísticas voraces**.
- **La clave:** diseñar buenas funciones de selección.

3.4.1. El problema del viajante.

- **Problema:** Dado un grafo no dirigido, completo y ponderado G = (V, A), encontrar un ciclo de coste mínimo que pase por todos los nodos.



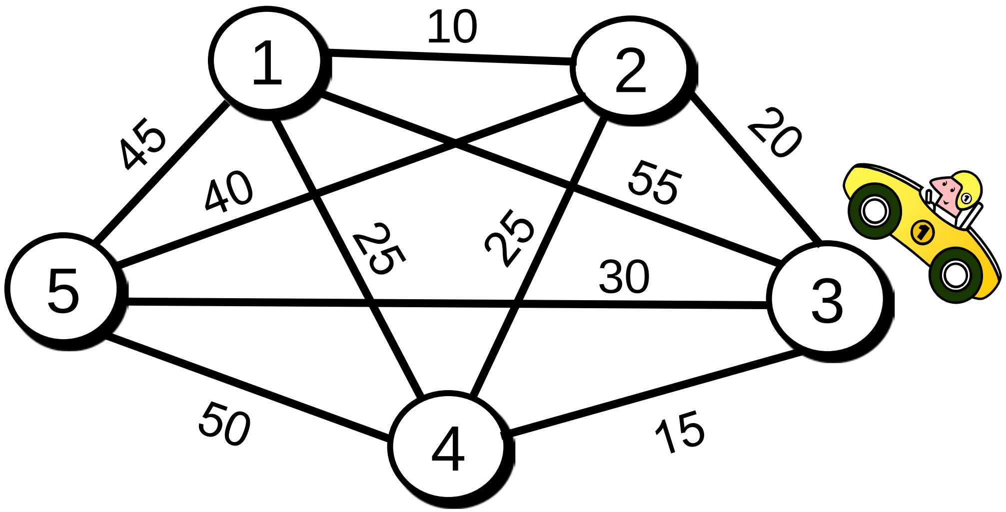
- Es un problema NP-completo, pero necesitamos una solución eficiente.
- Problema de optimización, la solución está formada por un conjunto de elementos en cierto orden: podemos aplicar el esquema voraz.

3.4.1. El problema del viajante.

- Primera cuestión: ¿Cuáles son los candidatos?
- **Dos posibilidades**:

1) Los nodos son los candidatos. Empezar en un nodo cualquiera. En cada paso moverse al nodo no visitado más próximo al último nodo seleccionado.

2) Las aristas son los candidatos. Hacer igual que en el algoritmo de Kruskal, pero garantizando que se forme un ciclo.

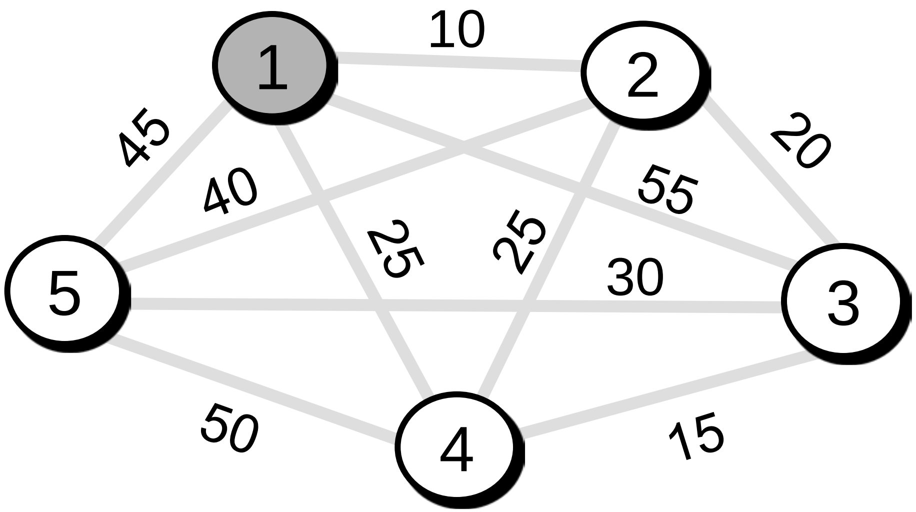


3.4.1. El problema del viajante.

- **Heurística voraz 1) Candidatos = V**
  - Una solución será un cierto orden en el conjunto de nodos.
  - **Representación de la solución: s** = $(\mathbf{c_1}, \mathbf{c_2}, \ldots, \mathbf{c_n})$, donde $\mathbf{c_i}$ es el nodo visitado en el lugar **i**-ésimo.
  - **Inicialización**: empezar en un nodo cualquiera.
  - Función de **selección**: de los nodos candidatos seleccionar el más próximo al último (o al primero) de la secuencia actual $(c_1, c_2, \ldots, c_a)$.
  - Acabamos cuando tengamos **n** nodos.

3.4.1. El problema del viajante.

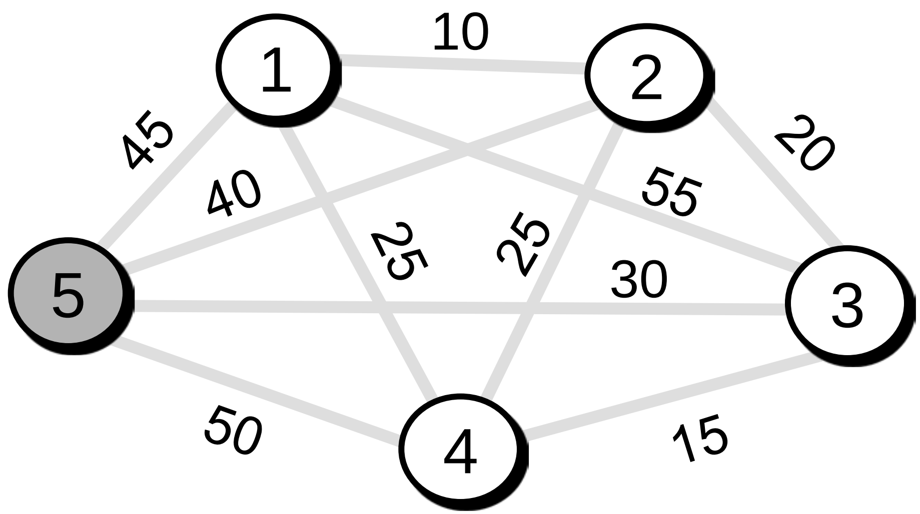
- **Ejemplo 1.** Empezando en el nodo 1.



Solución: (5, 1, 2, 3, 4)
Coste total: 45+10+20+15+50=140

3.4.1. El problema del viajante.

- **Ejemplo 2.** Empezando en el nodo 5.



Solución: (5, 3, 4, 2, 1)

Coste total: 30+15+25+10+45 = 125

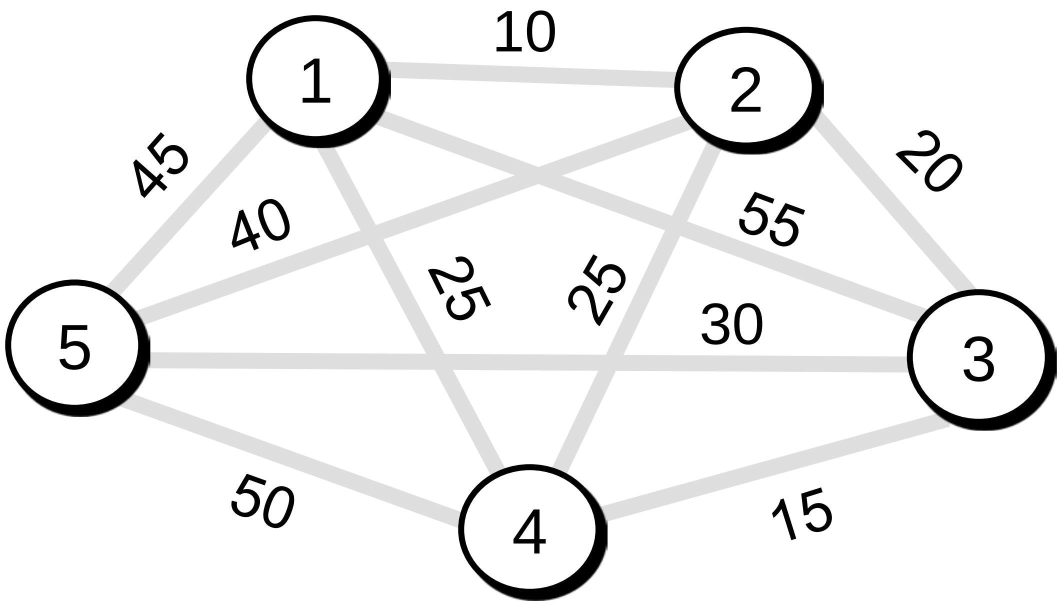
- **Conclusión:** el algoritmo no es óptimo.

3.4.1. El problema del viajante.

- **Heurística voraz 2) Candidatos = A**
  - Una solución será un conjunto de aristas $(a_1, a_2, ..., a_n)$ que formen un ciclo hamiltoniano, sin importar el orden.
  - **Representación de la solución: $s = (a_1, a_2, ..., a_n)$**, donde cada $\mathbf{a_i}$ es una arista, de la forma $\mathbf{a_i} = (\mathbf{v_i}, \mathbf{w_i})$.
  - **Inicialización:** empezar con un grafo sin aristas.
  - **Selección**: seleccionar la arista candidata de menor coste.
  - **Factible**: una arista se puede añadir a la solución actual si no se forma un ciclo (excepto para la última arista añadida) y si los nodos unidos no tienen grado mayor que 2.

3.4.1. El problema del viajante.

- **Ejemplo 3.**



Solución: ((1, 2), (4, 3), (2, 3), (1, 5), (4, 5))

Coste total = 10+15+20+45+50 = 140

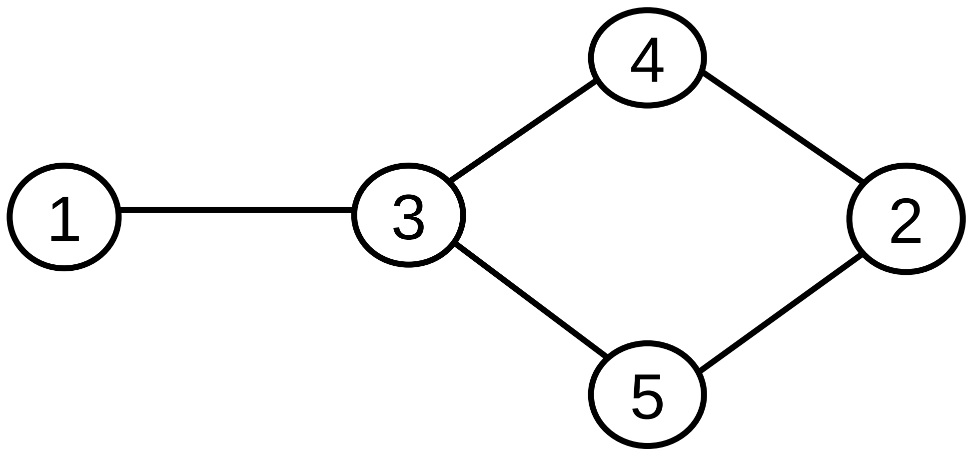
3.4.1. El problema del viajante.

Conclusiones:

- Ninguno de los dos algoritmos garantiza la solución óptima.
- Sin embargo, “normalmente” ambos dan soluciones buenas, próximas a la óptima.
- Posibles mejoras:
  - Buscar heurísticas mejores, más complejas.
  - Repetir la heurística 1 con varios orígenes.
  - A partir de la solución del algoritmo intentar hacer **modificaciones locales** para mejorar esa solución.

3.4.2. Coloración de grafos.

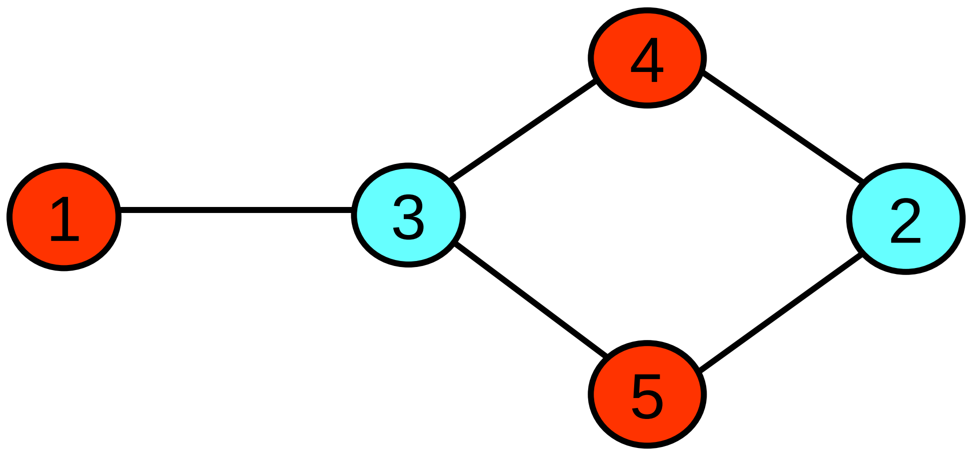
- **Coloración de un grafo:** asignación de un color a cada nodo, de forma que dos nodos unidos con un arco tengan siempre distinto color.
- **Problema de coloración:** dado un grafo no dirigido, realizar una coloración utilizando el número mínimo de colores.



- También es un problema NP-completo.
- ¿Cómo interpretar el problema? ¿Cómo representar una solución?

3.4.2. Coloración de grafos.

- **Representación de la solución:** una solución tiene la forma ($\mathbf{c_1}$, $\mathbf{c_2}$, ..., $\mathbf{c_n}$), donde $\mathbf{c_i}$ es el color asignado al nodo **i**.
- La solución es válida si para toda arista (**v**, **w**) $\in$ A, se cumple que $\mathbf{c_v} \neq \mathbf{c_w}$.



- S = (1, 2, 2, 1, 1), Total: 2 colores

3.4.2. Coloración de grafos.

- Podemos usar una **heurística voraz** para obtener una solución:
  - Inicialmente ningún nodo tiene color asignado.
  - Tomamos un color *colorActual*:= 1.
  - Para cada uno de los nodos sin colorear:
    - Comprobar si es posible asignarle el color actual.
    - Si se puede, se asigna. En otro caso, se deja sin colorear.
  - Si quedan nodos sin colorear, escoger otro color (*colorActual*:= *colorActual* + 1) y volver al paso anterior.

3.4.2. Coloración de grafos.

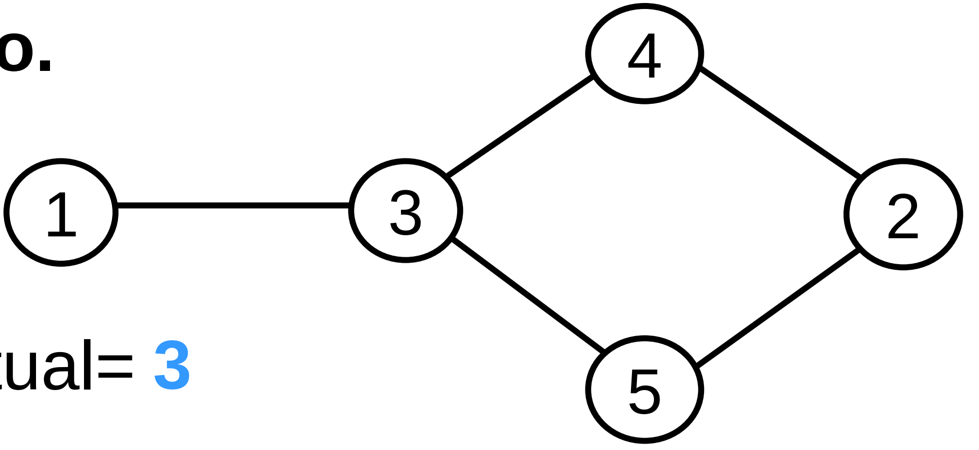
- La estructura básica del esquema voraz se repite varias veces, una por cada color, hasta que todos los nodos estén coloreados.
- Función de **selección**: cualquier candidato restante.
- **Factible(x)**: se puede asignar un color a **x** si ninguno de sus adyacentes tiene ese mismo color.

para todo nodo y adyacente a x **hacer**
        si c_y == *colorActual* **entonces**
                devolver false
finpara
devolver true

- ¿Garantiza el algoritmo la solución óptima?

3.4.2. Coloración de grafos.

- **Ejemplo.**



colorActual= **3**

c_1	c_2	c_3	c_4	c_5

- **Resultado:** se necesitan 3 colores. Recordar que el óptimo es 2 colores.
- **Conclusión:** el algoritmo no es óptimo.
- ¿Cuál será el tiempo de ejecución en el peor caso?

3. Algoritmos voraces.

Conclusiones:

- Avance rápido se basa en una idea intuitiva:
  – Empezamos con una solución “vacía”, y la construimos paso a paso.
  – En cada paso se selecciona un candidato (el más prometedor) y se decide si se mete o no (función factible).
  – Una vez tomada una decisión no se vuelve a deshacer.
  – Acabamos cuando tenemos una solución o cuando no queden candidatos.

3. Algoritmos voraces.

Conclusiones:

- **Primera cuestión:** ¿cuáles son los candidatos?, ¿cómo se representa una solución al problema?
- **Cuestión clave:** diseñar una función de selección adecuada.
  - Algunas pueden garantizar la solución óptima.
  - Otras pueden ser más heurísticas...
- **Función factible:** garantizar las problema.
- En general los algoritmos voraces son la solu ón rápida a muchos problemas (a veces óptimas, otras o).
- ¿Y si podemos deshacer decisiones...?

BACKTRACKING