Algoritmos voraces

Algoritmos y estructuras de datos II

José Antonio Hernández López[1]

[1]Departamento de Informática y Sistemas
Universidad de Murcia

11 de marzo de 2025

Índice

Algoritmo voraz

Definición

Los algoritmos voraces (o de avance rápido) son un tipo de algoritmos que construyen la solución paso a paso y, en cada paso, toma un decisión que es *localmente óptima* con la esperanza de que, al final, lleguemos a la solución óptima global.

1. Una vez que se toma la decisión, no hay vuelta atrás.
2. Se suelen utilizar en problemas de optimización.
3. Hay veces que obtenemos la solución óptima usando un algoritmo voraz y otras que no (en la gran mayoría no).
4. Suelen ser métodos iterativos muy rápidos y eficientes (suele ser posible tener una implementación recursiva).

Algoritmo voraz: esquema *puro*

Dado un problema P y una solución S inicialmente vacía, el algoritmo voraz hace lo siguiente:

1. Se toma una **decisión voraz** en base a *información local* de P y se añade a la solución S. Si es la solución final, terminamos.
2. Se construye un subproblema P' en base a la decisión voraz de la misma naturaleza que P.
3. Volver a ejecutar 1 y 2 para P' hasta que tengamos la solución final.

Ejemplo: el problema de la rana 🐸

Descripción del problema

- La rana empieza en la posición 0 y quiere llegar a la posición n.
- Hay nenúfares en varias posiciones. Hay uno en la posición 0 y otra en la posición n.
- La rana puede saltar, como máximo, d unidades en un solo salto.
- **Objetivo:** Encontrar el camino que la rana debería seguir que minimize el número de saltos. Se asume que existe una solución.

Ejemplo: el problema de la rana 🐸

Ejemplo: el problema de la rana 🐸

Nuestra decisión voraz: en cada paso escoger el nenúfar más alejado.

Ejemplo: el problema de la rana 🐸

El problema P inicial está parametrizado por (i, f) donde i es el inicio y f es el final. Así pues, en cada paso,

1. La decisión voraz viene dada por el nenúfar l más alejado alcanzable desde i que no sobrepase a f. Si es f, entonces hemos terminado. Si no es f, se añade a S.
2. Construimos $P(l, f)$ (llegar de l a f).
3. Volver a ejecutar 1 y 2 para $P(l, f)$ hasta llegar a f.

Ejemplo: el problema de la rana 🐸

```python
def rana_voraz(nenufares, d):
    # inicializamos la solución como lista vacía
    S = []
    # posición actual
    x = 0
    # tamaño tablero
    n = len(nenufares) - 1

    while x < n:
        # si podemos saltar al final y terminar lo hacemos
        if x + d >= n:
            x = n
        else:
            # elección voraz: escogemos el nenúfar
            # más alejado de x al que podemos saltar
            eleccion_voraz = -1
            for j in range(d, 0, -1):
                if nenufares[x + j] == 1:
                    eleccion_voraz = x + j
                    break

            # añadimos la elección voraz a S
            S.append(eleccion_voraz)
            # transformamos el problema en uno más pequeño avanzando la pos actual
            x = eleccion_voraz

    return S

nenufares = [1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1]
d = 3
rana_voraz(nenufares, d)
# salida: [3, 5, 8]
```

Teorema del algoritmo voraz

Para este problema, ¿el algoritmo voraz devuelve la solución óptima?

Teorema del algoritmo voraz

Para este problema, ¿el algoritmo voraz devuelve la solución óptima?
En general, probar que un algoritmo voraz devuelve la solución óptima, no es trivial...

Teorema del algoritmo voraz

Teorema del algoritmo voraz

Dado un algoritmo voraz diseñado para un problema de optimización P, si se cumplen estas dos condiciones:

- **Propiedad de la decisión voraz**: existe una solución óptima de P que contiene la decisión voraz $\Rightarrow$ tomando la decisión voraz vamos encaminados a la solución óptima final
- **Subestructura óptima**: Sea I la decisión voraz para P y S' la solución óptima al problema P', entonces I unido a S' es una solución óptima de P $\Rightarrow$ podemos resolver el subproblema que queda de la misma manera.

entonces el agoritmo devuelve la solución óptima.

Ejemplo: el problema de la rana 🐸

Teorema del algoritmo voraz aplicado a 🐸

Dado el problema $P(i, f)$:

- Propiedad de la decisión voraz: si estoy en la posición i y escojo el más alejado l, existe una solución S de $P(i, f)$ que empieza por l.
- Subestructura óptima: Sea l la decisión voraz y S' la solución de $P(l, f)$, entonces l concatenado con S' es la solución óptima de $P(i, f)$.

Ejemplo: el problema de la rana 🐸

Lema

Propiedad de la decisión voraz: si estoy en la posición i y escojo el más alejado l, existe una solución óptima S de $P(i, f)$ que empieza por l.

Ejemplo: el problema de la rana 🐸

Lema

Propiedad de la decisión voraz: si estoy en la posición i y escojo el más alejado l, existe una solución óptima S de $P(i, f)$ que empieza por l.

Demostración. Esta propiedad se suele demostrar partiendo de una solución óptima y construyendo otra que contenga a la decisión voraz. Sea S una solución óptima de $P(i, j)$. Tenemos varias opciones:

1. l está en S como primer elemento. En este caso hemos terminado.
2. l está en S pero no como primer elemento. Esto no puede ser, pues entonces podríamos construir un $|S'| < |S|$ quitando todos los que hay antes que l.
3. l no está en S. Entonces S se puede descomponer en $L_1||L_2$ donde L_1, L_2 son los que están antes, después de l respectivamente. Si construimos $S' = l||L_2$ tenemos que $|S'| \leq |S|$ ya que $|L_1| \geq 1$.

Ejemplo: el problema de la rana 🐸

Lema

Subestructura óptima: Sea I la decisión voraz y S' la solución óptima de $P(I, f)$, entonces I concatenado con S' es la solución óptima de $P(i, f)$.

Ejemplo: el problema de la rana 🐸

Lema

Subestructura óptima: Sea l la decisión voraz y S' la solución óptima de $P(l, f)$, entonces l concatenado con S' es la solución óptima de $P(i, f)$.

Demostración. Por reducción al absurdo. Supongamos que existe W solución óptima de $P(i, f)$ tal que $|W| < |S' \cup \{l\}| = |S'| + 1$. Entonces:

1. Si W contiene a l como primer elemento, entonces $W - \{l\}$ sería una solución válida de $P(l, f)$ tal que $|W - \{l\}| = W - 1 < |S'|$. Lo que sería una contradicción pues S' es una solución óptima de $P(l, f)$.
2. Si W no contiene a l, entonces tiene que haber nenúfares L en W antes que l ya que l es el nenúfar más alejado posible alcanzable desde i. $W - L$ es una solución válida $P(l, f)$ tal que $|W - L| = |W| - |L| < |S'|$, contraducción.
3. Si W contiene a l pero no como primer elemento, entonces entonces tiene que haber nenúfares L en W antes que l. Así pues, $W - L - \{l\}$ es una solución válida de $P(l, f)$ tal que $|W - L - \{l\}| = |W| - |L| - 1 < |S'|$, contradicción.

Mochila no 0/1

Descripción del problema

- Tenemos $O = \{1, \ldots, n\}$, n objetos con pesos $p_i > 0$ beneficios $b_i > 0$.
- En la mochila tenemos que meter objetos, con un peso máximo de M. Se pueden fraccionar los objetos.
- **Objetivo:** Llenar la mochila maximizando el beneficio sin superar la capacidad máxima. Se asume que el problema no es trivial (es decir, $\sum_{i=1}^{n} p_i > M$).

Mochila no 0/1

Descripción del problema

- Tenemos $O = \{1, \ldots, n\}$, n objetos con pesos $p_i > 0$ beneficios $b_i > 0$.
- En la mochila tenemos que meter objetos, con un peso máximo de M. Se pueden fraccionar los objetos.
- **Objetivo:** Llenar la mochila maximizando el beneficio sin superar la capacidad máxima. Se asume que el problema no es trivial (es decir, $\sum_{i=1}^{n} p_i > M$).

Por ejemplo: $n = 3$, $M = 20$ y

$$p = (18, 15, 10)$$

$$b = (25, 24, 15)$$

1. $S_1 = (1, 2/15, 0)$, beneficio $= 28,2$
2. $S_2 = (0, 2/3, 1)$, beneficio $= 31$

Las soluciones se representan como una tupla de n, $0 \leq x_i \leq 1$.

Mochila no 0/1

El problema P está parametrizado por (M, O) donde M es el peso máximo de la mochila y O son los objetos. Así pues, en cada paso,

1. Tomamos una decisión voraz y la añadimos a S. La decisión voraz es el objeto I que metemos con su proporción x_I. Si I es el único objeto, terminamos. Si ya no nos queda espacio en la mochila, terminamos.
2. Construimos $P(M - x_I p_I, O - \{I\})$
3. Volver a ejecutar 1 y 2 para $P(M - x_I p_I, O - \{I\})$.

Mochila no 0/1 🎒

El problema P está parametrizado por (M, O) donde M es el peso máximo de la mochila y O son los objetos. Así pues, en cada paso,

1. Tomamos una decisión voraz y la añadimos a S. La decisión voraz es el objeto I que metemos con su proporción x_I. Si I es el único objeto, terminamos. Si ya no nos queda espacio en la mochila, terminamos.
2. Construimos $P(M - x_I p_I, O - \{I\})$
3. Volver a ejecutar 1 y 2 para $P(M - x_I p_I, O - \{I\})$.

Nota

Vamos a asumir que, una vez seleccionado el objeto, la proporción que vamos a añadir va a ser la máxima posible. Es decir, si el objeto seleccionado cabe entero, lo metemos. Si no cabe, metemos lo máximo posible de eso objeto hasta llenar la mochila.

Mochila no 0/1

```python
def seleccion_voraz(O, B, P, capacidad_restante):
    ...

def mochila(O, B, P, M):
    # solución como array de 0s de longitud el número de objetos
    S = [0] * len(O)
    # peso actual
    m = 0
    # los objetos disponibles en cada paso
    O_disponibles = set(O)
    while m < M and len(O_disponibles)!=0:
        capacidad_restante = M - m
        # selecciono el objeto y la cantidad que quiero meter
        x_l, l = seleccion_voraz(O_disponibles, B, P, capacidad_restante)
        # añado la solución
        S[l] = x_l

        # transformo el problema en uno más pequeño
        # elimino el objeto seleccionado de los disponibles
        O_disponibles.remove(l)
        # aumento el peso actual
        m += x_l*P[l]
    return S
```

Mochila no 0/1 🎒

Posibles criterios:

- El objeto con más beneficio

Posibles criterios:

- El objeto con más beneficio ❌
  $n = 4; M = 10$

$$p = (10, 3, 3, 4)$$
$$b = (10, 9, 9, 9)$$

Si ejecutamos el algoritmo con este criterio obtendríamos un beneficio total de 10 pues solo escogeríamos el primer elemento. Esta solución dista de la óptima...

Mochila no 0/1 🎒

Posibles criterios:

- El objeto con más beneficio ❌
  $n = 4; M = 10$
  $$p = (10, 3, 3, 4)$$
  $$b = (10, 9, 9, 9)$$
  Si ejecutamos el algoritmo con este criterio obtendríamos un beneficio total de 10 pues solo escogeríamos el primer elemento. Esta solución dista de la óptima...
- El objeto menos pesado

Mochila no 0/1 🎒

Posibles criterios:

- El objeto con más beneficio ❌
  $n = 4; M = 10$
  $$p = (10, 3, 3, 4)$$
  $$b = (10, 9, 9, 9)$$
  Si ejecutamos el algoritmo con este criterio obtendríamos un beneficio total de 10 pues solo escogeríamos el primer elemento. Esta solución dista de la óptima...
- El objeto menos pesado ❌ $n = 2; M = 10$
  $$p = (10, 9)$$
  $$b = (10, 1)$$
  Siguiendo un razonamiento similar, al ejecutar el algoritmo no obtenemos la solución óptima.
- El objeto con mejor proporción b_i/p_i ... ✅

Mochila no 0/1 🎒

Teorema del algoritmo voraz aplicado a 🎒

Dado el problema $P(M, O)$ y el algoritmo voraz de la mejor proporción:

- Propiedad de la decisión voraz: existe una solución óptima que contiene al elemento con mejor proporción y en la mayor cantidad posible.
- Subestructura óptima: sea x_l la decisión voraz escogida para $P(M, O)$ y S' la solución óptima de $P(M - x_l p_l, O - \{l\})$, entonces $\{x_l\} \cup S'$ es solución óptima de $P(M, O)$.

Mochila no 0/1

Lema

Propiedad de la decisión voraz: existe una solución óptima que contiene al elemento con mejor proporción y en la mayor cantidad posible.

[1] ya que si no existiera al menos uno significaría que o bien, la mochila está vacía o bien solo está i en la solución pero no en la mayor cantidad posible

Mochila no 0/1 🎒

Lema

Propiedad de la decisión voraz: existe una solución óptima que contiene al elemento con mejor proporción y en la mayor cantidad posible.

Demostración. Sea S la solución óptima de P e i el objeto con mejor proporción. Entonces tenemos dos opciones:

1. S contiene a i y en la mayor cantidad posible. No hay nada que hacer y hemos terminado
2. S no contiene a i o lo contiene pero no en la mayor cantidad posible.

En este segundo caso, podemos suponer que existe otro objeto en la solución j con $0 \neq x_j \in S$[1]. Este objeto cumple que

$$\frac{b_i}{p_i} \geq \frac{b_j}{p_j}.$$

[1] ya que si no existiera al menos uno significaría que o bien, la mochila está vacía o bien solo está i en la solución pero no en la mayor cantidad posible

Mochila no 0/1 🎒

Dado un $r > 0$, construimos un S' quitando un peso r de j y poniéndoselo a i. Sea x'_j y x'_i las cantidades de cada objeto que se quitan/añaden tales que $p_j x'_j = r$ y $p_i x'_i = r$ (lo que se añade de cada objeto es igual a r). Así pues

- Si $\frac{b_i}{p_i} > \frac{b_j}{p_j}$, entonces

$$b(S') = b(S) - b_j x'_j + b_i x'_i = b(S) + r\left(\frac{b_i}{p_i} - \frac{b_j}{p_j}\right) > b(S).$$

  Esto no puede darse ya que hemos supuesto que S es la solución óptima.

- No nos queda más remedio que suponer que $\frac{b_i}{p_i} = \frac{b_j}{p_j}$, entonces

$$b(S') = b(S) - b_j x'_j + b_i x'_i = b(S) + r\left(\frac{b_i}{p_i} - \frac{b_j}{p_j}\right) = b(S)$$

Esto podría darse y corresponde al caso de que haya objetos que tengan la misma proporción beneficio-peso que i. En ese caso, como para cada r siempre podemos pasar ese peso de un objeto $j \neq i$ a i y obtener una solución igual de buena (óptima), pues pasamos peso de todos los objetos distintos a i hasta conseguir la mayor cantidad posible.

Mochila no 0/1 🎒

Lema

Subestructura óptima: sea x_l la decisión voraz escogida para $P(M, O)$ y S' la solución óptima de $P(M - x_l p_l, O - \{l\})$, entonces $\{x_l\} \cup S'$ es solución óptima de $P(M, O)$.

Mochila no 0/1

Lema

Subestructura óptima: sea x_l la decisión voraz escogida para $P(M, O)$ y S' la solución óptima de $P(M - x_l p_l, O - \{l\})$, entonces $\{x_l\} \cup S'$ es solución óptima de $P(M, O)$.

Por reducción a lo absurdo, supongamos que existe W solución óptima de $P(M, O)$ tal que $b(W) > b(\{x_l\} \cup S') = b(S') + x_l b_l$. Sea $W' = W - \{x_l\}$, entonces:

$$p(W') = p(W) - x_l p_l = M - x_l p_l$$

$$b(W') = b(W) - x_l b_l > b(S')$$

de este modo, llegamos a una contradicción.

Mochila no 0/1

```python
def seleccion_voraz(O, B, P, capacidad_restante):
    l = -1
    l_prop = -1

    for o in O:
        if B[o]/P[o] > l_prop:
            l_prop = B[o]/P[o]
            l = o

    if capacidad_restante > P[l]:
        return 1, l
    else:
        return capacidad_restante/P[l], l
```

Mochila 0/1

El algoritmo voraz no funciona en la Mochila 0/1 donde no podemos fraccionar los objetos...

Mochila 0/1 🎒

El algoritmo voraz no funciona en la Mochila 0/1 donde no podemos fraccionar los objetos... Supongamos $n = 3; M = 50$

$$p = (10, 20, 30)$$

$$b = (60, 100, 120)$$

$$b/p = (6, 5, 4)$$

$S_{\text{voraz}} = (1, 1, 0),\ B(S_{\text{voraz}}) = 160$
$S = (0, 1, 1),\ B(S) = 220$

Mochila 0/1 🎒

El algoritmo voraz no funciona en la Mochila 0/1 donde no podemos fraccionar los objetos... Supongamos $n = 3; M = 50$

$$p = (10, 20, 30)$$

$$b = (60, 100, 120)$$

$$b/p = (6, 5, 4)$$

$S_{\text{voraz}} = (1, 1, 0),\ B(S_{\text{voraz}}) = 160$
$S = (0, 1, 1),\ B(S) = 220$
La Mochila 0/1 es NP-completo. La única forma (conocida) de encontrar la solución óptima es probar todas las combinaciones y eso tiene un orden exponencial.

Cambio de monedas

Descripción del problema

- Tenemos n monedas (identificadas por $O = \{1, \ldots, n\}$) con valores $V = \{v_1, \ldots, v_n\}$.
- Nos dan una cantidad C que tenemos que devolver usando ese sistema monetario.
- **Objetivo:** Devolver la cantidad C minimizando el número de monedas.

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $3,89$.

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $3,89$. Las soluciones se representan como una tupla de n, $0 \leq x_i$ que indica la cantidad de monedas de valor i.

$$S = (0, 2, 1, 1, 1, 1, 1, 1), \text{Total de monedas} = 8$$

Cambio de monedas 🪙

El problema P está parametrizado por:

Cambio de monedas

El problema P está parametrizado por:

- Monedas disponibles O
- Cantidad a devolver C

Cambio de monedas

El problema P está parametrizado por:

- Monedas disponibles O
- Cantidad a devolver C

Así pues, en cada paso,

1. Tomamos una decisión voraz y la añadimos a S. La decisión voraz es la moneda I que devolvemos y el número de monedas de esa cantidad x_I. Si I es la última modena, terminamos. Si ya hemos devuelto C, terminamos.
2. Construimos $P(C - x_I v_I, O - \{I\})$
3. Volver a ejecutar 1 y 2 para $P(C - x_I v_I, O - \{I\})$.

Cambio de monedas

El problema P está parametrizado por:

- Monedas disponibles O
- Cantidad a devolver C

Así pues, en cada paso,

1. Tomamos una decisión voraz y la añadimos a S. La decisión voraz es la moneda I que devolvemos y el número de monedas de esa cantidad x_I. Si I es la última modena, terminamos. Si ya hemos devuelto C, terminamos.
2. Construimos $P(C - x_I v_I, O - \{I\})$
3. Volver a ejecutar 1 y 2 para $P(C - x_I v_I, O - \{I\})$.

Decisión voraz

Seleccionamos la moneda más alta posible y devolvemos la máxima cantidad de monedas posible.

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $5,89$, ¿cómo se ejecutaría el algoritmo?

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $5,89$, ¿cómo se ejecutaría el algoritmo?

1. P(todas las monedas, $C = 5,89) \rightarrow 2$ monedas de 2 euros $+$ P(todas las monedas menos la de 2, $C = 5,89 - 4 = 1,89$)

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $5,89$, ¿cómo se ejecutaría el algoritmo?

1. P(todas las monedas, $C = 5,89$) $\rightarrow 2$ monedas de 2 euros $+$ P(todas las monedas menos la de 2, $C = 5,89 - 4 = 1,89$)
2. P(todas las monedas menos la de 2, $C = 1,89$) $\rightarrow 1$ moneda de 1 euro $+$ P(todas las monedas menos la de 2 y 1, $C = 0,89$)

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $5, 89$, ¿cómo se ejecutaría el algoritmo?

1. P(todas las monedas, $C = 5, 89) \rightarrow 2$ monedas de 2 euros $+$ P(todas las monedas menos la de 2, $C = 5, 89 - 4 = 1, 89)$
2. P(todas las monedas menos la de 2, $C = 1, 89) \rightarrow 1$ moneda de 1 euro $+$ P(todas las monedas menos la de 2 y 1, $C = 0, 89)$
3. P $\rightarrow 1$ moneda de $0{,}5 +$ P(todas menos la de 2, 1, 0.5, $C = 0, 39)$

Cambio de monedas

Por ejemplo: en el Euro tenemos,

$$V = \{0{,}01, 0{,}02, 0{,}05, 0{,}1, 0{,}2, 0{,}5, 1, 2\}$$

y queremos devolver la cantidad $5,89$, ¿cómo se ejecutaría el algoritmo?

1. P(todas las monedas, $C = 5,89) \rightarrow 2$ monedas de 2 euros $+$ P(todas las monedas menos la de 2, $C = 5,89 - 4 = 1,89$)
2. P(todas las monedas menos la de 2, $C = 1,89) \rightarrow 1$ moneda de 1 euro $+$ P(todas las monedas menos la de 2 y 1, $C = 0,89$)
3. $P \rightarrow 1$ moneda de $0{,}5 +$ P(todas menos la de 2, 1, 0.5, $C = 0,39$)
4. ...

Cambio de monedas

```python
def monedas_voraz(C, V):
    # se asume que V está ordenado
    n = len(V)
    S = []

    cantidad_actual = C
    for l in range(0, n):

        # selección voraz
        # la moneda es la l porque están ordenados
        # falta la cantidad
        x_l = 0
        while V[l]*(x_l + 1) < cantidad_actual:
            x_l += 1

        # añado a la solución
        S.append(x_l)
        # transformo el problema
        cantidad_actual -= V[l]*x_l

    return S

C = 27.89
V = [2, 1, 0.5, 0.2, 0.1, 0.05, 0.02, 0.01]
S = monedas_voraz(C, V)
# [13, 1, 1, 1, 1, 1, 2, 0]
```

Cambio de monedas

El algoritmo voraz del cambio de monedas satisface la subestructura óptima y, dependiendo del sistema monetario, puede llegar a satisfacer la propiedad de la decisión voraz.

Cambio de monedas

El algoritmo voraz del cambio de monedas satisface la subestructura óptima y, dependiendo del sistema monetario, puede llegar a satisfacer la propiedad de la decisión voraz.

- Para el Euro y el Dollar, el algoritmo satisface la propiedad de la decisión voraz.
- Para $V = \{1, 3, 4\}$, el algoritmo voraz no devuelve la solución óptima. Por ejemplo, ejecutad el algoritmo con $C = 6$ y V como sistema monetario.
- En la literatura, los sistemas monetarios para los cuales el algoritmo voraz devuelve la solución óptima se les llaman **sistemas monetarios canónicos**.

Planificación de tareas 📝

Descripción del problema

- Tenemos un procesador y n tareas disponibles.
- Todas las tareas requieren una unidad de tiempo para ejecutarse.
- Todas tienen un beneficio b_i si se ejecutan y un plazo máximo de ejecución d_i.
- La tarea i ha de ejecutarse antes de d_i si se quiere obtener el beneficio.
- **Objetivo:** Dar una secuencia de tareas que maximice el beneficio obtenido.

Por ejemplo $n = 6$,

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

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

Dos soluciones podrían ser:

Instante	1	2	3	4	5	6
S_1	1	3	0	2	4	5
S_2	0	1	2	3	4	5

donde

$$B(S_1) = 15 + 7 + 20 + 0 + 0 + 0 = 42$$

y

$$B(S_2) = 20 + 0 + 0 + 0 + 0 + 0 = 20.$$

El problema P está parametrizado por:

Planificación de tareas 📝

El problema P está parametrizado por:

- Instante actual: i
- Tareas disponibles: T

Así pues, en cada paso,

Planificación de tareas 📝

El problema P está parametrizado por:

- Instante actual: i
- Tareas disponibles: T

Así pues, en cada paso,

1. Tomamos una decisión voraz y la añadimos a S. Si ya no quedan tareas, terminamos.
2. Construimos $P(i+1, T-\{l\})$
3. Volver a ejecutar 1 y 2 para $P(i+1, T-\{l\})$.

Planificación de tareas

El problema P está parametrizado por:

- Instante actual: i
- Tareas disponibles: T

Así pues, en cada paso,

1. Tomamos una decisión voraz y la añadimos a S. Si ya no quedan tareas, terminamos.
2. Construimos $P(i+1, T-\{l\})$
3. Volver a ejecutar 1 y 2 para $P(i+1, T-\{l\})$.

Decisión voraz

De las actividades cuyo plazo no se ha pasado y se va a pasar antes, seleccionar la que da más beneficio. Si a todas se les ha pasado el plazo, seleccionar la que sea.

Planificación de tareas 📝

Supongamos $n = 6$,

$T =$	0	1	2	3	4	5
$b =$	20	15	10	7	5	3
$d =$	3	1	1	3	1	3

Supongamos $n = 6$,

$T =$	0	1	2	3	4	5
$b =$	20	15	10	7	5	3
$d =$	3	1	1	3	1	3

- $P(1, T) \rightarrow$ Tarea 1 $+ P(2, T - \{1\})$

Planificación de tareas 📝

Supongamos $n = 6$,

$T =$	0	1	2	3	4	5
$b =$	20	15	10	7	5	3
$d =$	3	1	1	3	1	3

- $P(1, T) \rightarrow$ Tarea 1 $+ P(2, T - \{1\})$
- $P(2, T - \{1\}) \rightarrow$ Tarea 0 $+ P(3, T - \{1, 0\})$

Planificación de tareas 📝

Supongamos $n = 6$,

$T =$	0	1	2	3	4	5
$b =$	20	15	10	7	5	3
$d =$	3	1	1	3	1	3

- $P(1, T) \rightarrow$ Tarea 1 $+ P(2, T - \{1\})$
- $P(2, T - \{1\}) \rightarrow$ Tarea 0 $+ P(3, T - \{1, 0\})$
- $P(3, T - \{1, 0\}) \rightarrow$ Tarea 3 $+ P(4, T - \{3, 1, 0\})$

Planificación de tareas 📝

Supongamos $n = 6$,

$T =$	0	1	2	3	4	5
$b =$	20	15	10	7	5	3
$d =$	3	1	1	3	1	3

- $P(1, T) \to$ Tarea 1 $+ P(2, T - \{1\})$
- $P(2, T - \{1\}) \to$ Tarea 0 $+ P(3, T - \{1, 0\})$
- $P(3, T - \{1, 0\}) \to$ Tarea 3 $+ P(4, T - \{3, 1, 0\})$
- A todas se les ha pasado el plazo, seleccionar cualquiera.
- ...

Supongamos $n = 6$,

$T =$	0	1	2	3	4	5
$b =$	20	15	10	7	5	3
$d =$	3	1	1	3	1	3

- $P(1, T) \rightarrow$ Tarea 1 $+ P(2, T - \{1\})$
- $P(2, T - \{1\}) \rightarrow$ Tarea 0 $+ P(3, T - \{1, 0\})$
- $P(3, T - \{1, 0\}) \rightarrow$ Tarea 3 $+ P(4, T - \{3, 1, 0\})$
- A todas se les ha pasado el plazo, seleccionar cualquiera.
- ...

No devuelve la solución óptima pero es una aproximación con sentido.

Planificación de tareas 📝

```python
def tareas(T, instante_inicial, B, D):
    S = []
    T_restantes = set(T)
    instante_actual = instante_inicial

    while len(T_restantes) != 0:
        # mientras queden tareas por hacer, tomamos la decisión voraz
        l = decision_voraz(T_restantes, B, D, instante_actual)
        # la añadimos a la solución
        S.append(l)
        # reducimos el problema
        T_restantes.remove(l)
        instante_actual += 1
    return S

T = {0,1,2,3,4,5}
instante_inicial = 1
B = [20, 15, 10, 7, 5, 3]
D = [3, 1, 1, 3, 1, 3]

tareas(T, instante_inicial, B, D)
# [1, 0, 3, 2, 4, 5]
```

Planificación de tareas 📝

```python
def decision_voraz(T, B, D, instante):
    # vemos si quedan tareas a las cuales no se les ha pasado el plazo
    primer_filtro = set([t for t in T if D[t] >=instante])

    # si no quedan, devolvemos la que sea
    if len(primer_filtro) == 0:
        return list(T)[0]
    else:
        # para aquellas tareas a los que no se les ha pasado el plazo
        # devolvemos la que se va a pasar antes y
        # si hay varias que se pasan a la vez, devolvemos la que da más beneficio
        l = -1
        l_d = float('inf')
        l_b = float('-inf')
        for t in primer_filtro:
            if l_d > D[t]:
                l = t
                l_d = D[t]
                l_b = B[t]
            elif l_d == D[t] and l_b < B[t]:
                l = t
                l_d = D[t]
                l_b = B[t]
        return l
```

Esqueva voraz v2.1

- Hay veces que es difícil aplicar el esquema voraz puro: $P \rightarrow I + P'$ donde P' es de la misma naturaleza que P.
- Os puede resultar conveniente usar el siguiente esquema voraz *equivalente*:

```
voraz (var S: CjtoSolución)
    S:= Ø
    C = generarCandidatos(S, … )
    mientras (C ≠ Ø) Y NO solución(S) hacer
        x:= seleccionar(C)
        C:= C - {x}
        si factible(S, x) entonces
            insertar(S, x)
            C = generarCandidatos(S, … )
        finsi
    finmientras
    si NO solución(S) entonces
        devolver "No se puede encontrar solución"
    finsi
```

Esquema voraz v2.2

Luego hay otra versión donde los candidatos no cambian y vienen dados al principio:

```
voraz (C: CjtoCandidatos; var S: CjtoSolución)
    S:= Ø
    mientras (C ≠ 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
```

Nota

Seguramente uséis estos dos esquemas en las prácticas.

Esquema voraz v2.2

- S es la solución
- C es el conjunto de candidatos
- `solución`(S) indica si S es una solución final al problema
- `seleccionar`(C) devuelve al elemento más prometedor (similar a la decisión voraz)
- `factible`(S, x) indica si es posible construir una solución añadiendo x a S
- `insertar`(S, x) inserta x en S
- `generarCandidatos`($S, \ldots$) genera los candidatos que son seleccionables en cada paso

Problema de la rana 🐸 y esquema v2

```
voraz (var S: CjtoSolución)
    S:= Ø
    C = generarCandidatos(S, … )
    mientras (C ≠ Ø) Y NO solución(S) hacer
        x:= seleccionar(C)
        C:= C - {x}
        si factible(S, x) entonces
            insertar(S, x)
            C = generarCandidatos(S, … )
        finsi
    finmientras
    si NO solución(S) entonces
        devolver "No se puede encontrar solución"
    finsi
```

- S lista de posiciones inicializada con el primer elemento.
- `generarCandidatos`(S) da las posiciones que son alcanzables desde la posición actual (dado por el último elemento de S).
- `seleccionar`(C) devuelve la posición más alejada.
- `factible`(S,x) devuelve `true` si hay un nenúfar en esa posición.
- `insertar`(S,x) inserta el elemento al final de S.
- `solución`(S) indica si el último elemento de S es la posición n.

Problema de la rana 🐸 y esquema v2

Siguiendo este esquema, el algoritmo quedaría así:

1. $S = [0]$ ▷inicializamos la solución al primer elemento
2. $C = \texttt{generarCandidatos}(S)$ ▷posiciones alcanzables desde 0
3. mientras $C \neq \emptyset$ y no $\texttt{solución}(S)$ hacer
    1. $x := \texttt{seleccionar}(C)$ ▷la posición más alejada
    2. $C := C - \{x\}$
    3. si $\texttt{factible}(S, x)$ ▷¿hay un nenúfar en x?
        1. $\texttt{insertar}(S, x)$
        2. $C = \texttt{generarCandidatos}(S)$ ▷regeneramos candidatos, alcanzables desde el último elemento de S
4. si no $\texttt{solución}(S)$ hacer
    1. devolver `NO HAY SOLUCIÓN`
5. devolver S

```
voraz (C: CjtoCandidatos; var S: CjtoSolución)
   S:= Ø
   mientras (C ≠ 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
```

- S array inicializado de tamaño n inicializado a 0s
- C los candidatos son los objetos restantes
- `seleccionar`(C) devuelve el objeto con más b/p
- `factible`(S, x) siempre es cierto.
- `insertar`(S, x) calcula la cantidad que se inserta a S
- `solución`(S) si hemos llenado ya la mochila

El problema del viajante ✈️

Descripción del problema

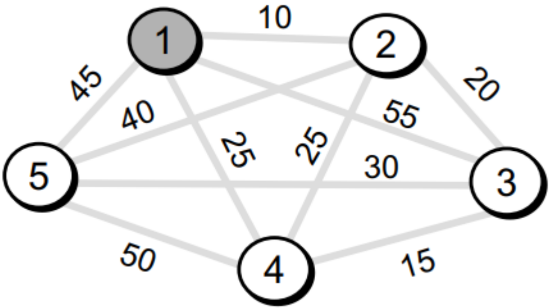
- Tenemos un grafo completo no dirigido $G = (V, E)$ donde cada arista e tiene un peso $w(e)$.
- **Objetivo:** Encontrar el ciclo hamiltoniano de peso mínimo.

El problema del viajante ✈️

Descripción del problema

- Tenemos un grafo completo no dirigido $G = (V, E)$ donde cada arista e tiene un peso $w(e)$.
- **Objetivo:** Encontrar el ciclo hamiltoniano de peso mínimo.

Este problema es NP-completo, pero podemos da una buena solución con una heurística voraz...


1
2
3
4
5
10
20
55
45
40
25
25
30
50
15

El problema del viajante ✈️

```
voraz (var S: CjtoSolución)
    S:= Ø
    C = generarCandidatos(S, … )
    mientras (C ≠ Ø) Y NO solución(S) hacer
        x:= seleccionar(C)
        C:= C - {x}
        si factible(S, x) entonces
            insertar(S, x)
            C = generarCandidatos(S, … )
        finsi
    finmientras
    si NO solución(S) entonces
        devolver "No se puede encontrar solución"
    finsi
```

- S lista de nodos inicializada con un elemento al azar.
- `generarCandidatos(`S`)` devuelve los vecinos del nodo actual.
- `seleccionar(`C`)` devuelve el nodo más cercano.
- `factible(`S`,`x`)` devuelve `true` si no está visitado (en S).
- `insertar(`S, x`)` inserta el elemento al final de S.
- `solución(`S`)` devuelve `true` si están todos los nodos visitados.

El problema del viajante ✈️

Algoritmo voraz:

1. Seleccionar un nodo v al azar como punto de partida.
2. Sea $S = [v]$ la solución.
3. Hasta que todos los nodos no estén en S:
    1. Seleccionar el siguiente nodo l.
    2. Añadir l a S.
4. Devolver S.

Heurísrica voraz

Heurísrica voraz $1-$NN: escogemos el nodo más cercano al nodo actual (el último de S) y que no haya sido visitado.

Coloración de grafos 🖍️

Descripción del problema

- Tenemos un grafo completo no dirigido $G = (V, E)$.
- **Objetivo:** Colorear el grafo con el mínimo número de colores manera que nodos adyacentes no tengan el mismo color.

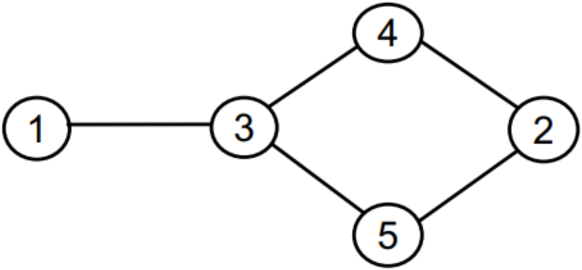
Coloración de grafos 🖍️

Descripción del problema

- Tenemos un grafo completo no dirigido $G = (V, E)$.
- **Objetivo:** Colorear el grafo con el mínimo número de colores manera que nodos adyacentes no tengan el mismo color.

Este problema es NP-completo, pero podemos da una buena solución con una heurística voraz...

Coloración de grafos



Coloración de grafos 🖍️

Algoritmo voraz:

1. $S = (-1, \ldots, -1)$ de dimensión V
2. Para cada vértice v hacer:
    1. Sea l el color más pequeño que no esté siendo usado por ninguno de los vecinos de v.
    2. $S[v] = l$
3. Devolver S

Algoritmo voraz:

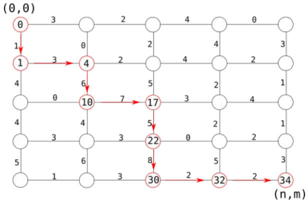
1. $S = (-1, \ldots, -1)$ de dimensión V
2. Para cada vértice v hacer:
    1. Sea I el color más pequeño que no esté siendo usado por ninguno de los vecinos de v.
    2. $S[v] = I$
3. Devolver S

Este algoritmo no devuelve la solución óptima...

Recorrido turístico por Manhattan 🏛️

Descripción del problema

- Tenemos un grafo $G = (V, E)$ en forma de *grid*. Cada arista tiene un peso asignado.
- Estamos en una posición inicial $(0, 0)$ y queremos llegar a la posición (n, n).
- Solo podemos movernos hacia abajo y hacia la derecha siempre que se pueda.
- **Objetivo:** Encontrar el camino más largo.



A este problema se le puede aplicar el esquema voraz puro de manera muy natural.

A este problema se le puede aplicar el esquema voraz puro de manera muy natural. El problema P está para metrizado por:

Recorrido turístico por Manhattan 🏛️

A este problema se le puede aplicar el esquema voraz puro de manera muy natural. El problema P está para metrizado por:

- Posición de inicio (x_0, y_0).
- Posición final (x_f, y_f).

Así pues, en cada paso,

Recorrido turístico por Manhattan 🏛️

A este problema se le puede aplicar el esquema voraz puro de manera muy natural. El problema P está para metrizado por:

- Posición de inicio (x_0, y_0).
- Posición final (x_f, y_f).

Así pues, en cada paso,

- Tomamos la decisión voraz (x_l, y_l) que puede ser hacia abajo o a la derecha (siempre que se pueda) y se añade a S. Si es la final, terminamos.
- Construimos $P((x_l, y_l), (x_f, y_f))$.
- Volvemos a ejecutar 1 y 2 para $P((x_l, y_l), (x_f, y_f))$.

Recorrido turístico por Manhattan 🏛

A este problema se le puede aplicar el esquema voraz puro de manera muy natural. El problema P está para metrizado por:

- Posición de inicio (x_0, y_0).
- Posición final (x_f, y_f).

Así pues, en cada paso,

- Tomamos la decisión voraz (x_I, y_I) que puede ser hacia abajo o a la derecha (siempre que se pueda) y se añade a S. Si es la final, terminamos.
- Construimos $P((x_I, y_I), (x_f, y_f))$.
- Volvemos a ejecutar 1 y 2 para $P((x_I, y_I), (x_f, y_f))$.

Decisión voraz

Cogemos el (x_I, y_I) más alejado.

Recorrido turístico por Manhattan 🏛️

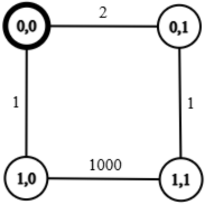
Este problema es muy interesante porque:

1. No es NP-completo
2. El esquema voraz no da la solución óptima
3. Se cumple la propiedad de la subestructura óptima pero no la de la decisión voraz.

Recorrido turístico por Manhattan 🏛️

Este problema es muy interesante porque:

1. No es NP-completo
2. El esquema voraz no da la solución óptima
3. Se cumple la propiedad de la subestructura óptima pero no la de la decisión voraz.



Complejidades de los algoritmos voraces

- Suelen ser bastante rápidos y tener órdenes polinomiales.
- El orden depende mucho del problema y el algoritmo.
- Una vez aplicado el esquema, lo normal es que se hagan optimizaciones.

Complejidades de los algoritmos voraces

```python
def mochila(O, B, P, M):
    # solución como array de 0s de longitud el número de objetos
    S = [0] * len(O)
    # peso actual
    m = 0
    # los objetos disponibles en cada paso
    O_disponibles = set(O)
    while m < M len(O_disponibles)!=0:
        capacidad_restante = M - m
        # selecciono el objeto y la cantidad que quiero meter
        x_l, l = seleccion_voraz(O_disponibles, B, P, capacidad_restante)
        # añado la solución
        S[l] = x_l

        # transformo el problema en uno más pequeño
        # elimino el objeto seleccionado de los disponibles
        O_disponibles.remove(l)
        # aumento el peso actual
        m += x_l*P[l]
    return S
```

Complejidades de los algoritmos voraces

```python
def mochila(O, B, P, M):
    # solución como array de 0s de longitud el número de objetos
    S = [0] * len(O)
    # peso actual
    m = 0
    # los objetos disponibles en cada paso
    O_disponibles = set(O)
    while m < M len(O_disponibles)!=0:
        capacidad_restante = M - m
        # selecciono el objeto y la cantidad que quiero meter
        x_l, l = seleccion_voraz(O_disponibles, B, P, capacidad_restante)
        # añado la solución
        S[l] = x_l

        # transformo el problema en uno más pequeño
        # elimino el objeto seleccionado de los disponibles
        O_disponibles.remove(l)
        # aumento el peso actual
        m += x_l*P[l]
    return S
```

Esto puede llegar a tener $O(n^2)$ ($\sim$dos bucles anidados), con n el número de objetos.

Complejidades de los algoritmos voraces

```
def mochila(O, B, P, M):
    # solución como array de 0s de longitud el número de objetos
    S = [0] * len(O)
    # peso actual
    m = 0
    # los objetos disponibles en cada paso
    O_disponibles = set(O)
    while m < M len(O_disponibles)!=0:
        capacidad_restante = M - m
        # selecciono el objeto y la cantidad que quiero meter
        x_l, l = seleccion_voraz(O_disponibles, B, P, capacidad_restante)
        # añado la solución
        S[l] = x_l

        # transformo el problema en uno más pequeño
        # elimino el objeto seleccionado de los disponibles
        O_disponibles.remove(l)
        # aumento el peso actual
        m += x_l*P[l]
    return S
```

Esto puede llegar a tener $O(n^2)$ ($\sim$dos bucles anidados), con n el número de objetos.
Si ordenamos antes los objetos usando el criterio b/p y luego recorremos los mismos calculando x_i tenemos $O(n \log n)$.

Conclusiones 🧑‍🎓

- Un algoritmo voraz busca la solución al problema tomando una sucesión de decisiones localmente óptimas sin preocuparse por la estructura global del problema.
- Son rápidos (tiempo polinomial) y se suelen implementar de forma iterativa.
- No siempre devuelven la solución óptima pero, con una buena heurística voraz, dan buenas soluciones al problema.
- Hemos visto dos esquemas generales: $P \to P' + I$ y esquema v2 con varias funciones a implementar (`generarCandidatos`, `factible`, etc.)
- El primer esquema es recomendable para demostrar optimalidad (subestructura óptima y decisión voraz).