Algoritmos voraces

Algoritmos y estructuras de datos II

José Antonio Hernández López[1]

[1]Departamento de Informática y Sistemas
Universidad de Murcia

2 de marzo de 2025

Índice

1. Algoritmo voraz y el problema de la 🐸
2. Mochila no 0/1 🎒

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

En cada paso de un algoritmo voraz hace lo siguiente:

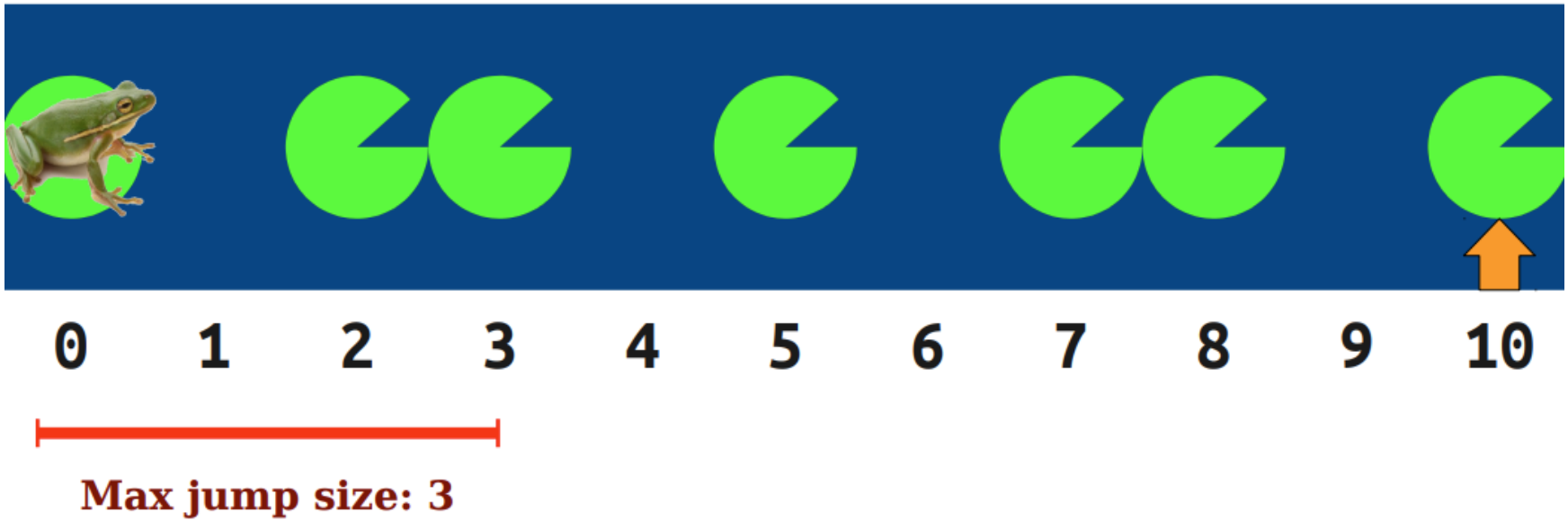
1. Se toma una **decisión voraz** en base a *información local* de P y se añade a la solución S. Si es la solución final, terminamos.
2. Se construye un subproblema P' en base a la decisión voraz de la misma naturaleza que P.
3. Volver a ejecutar 1 y 2 para P' hasta que tengamos la solución final.

Ejemplo: el problema de la rana 🐸

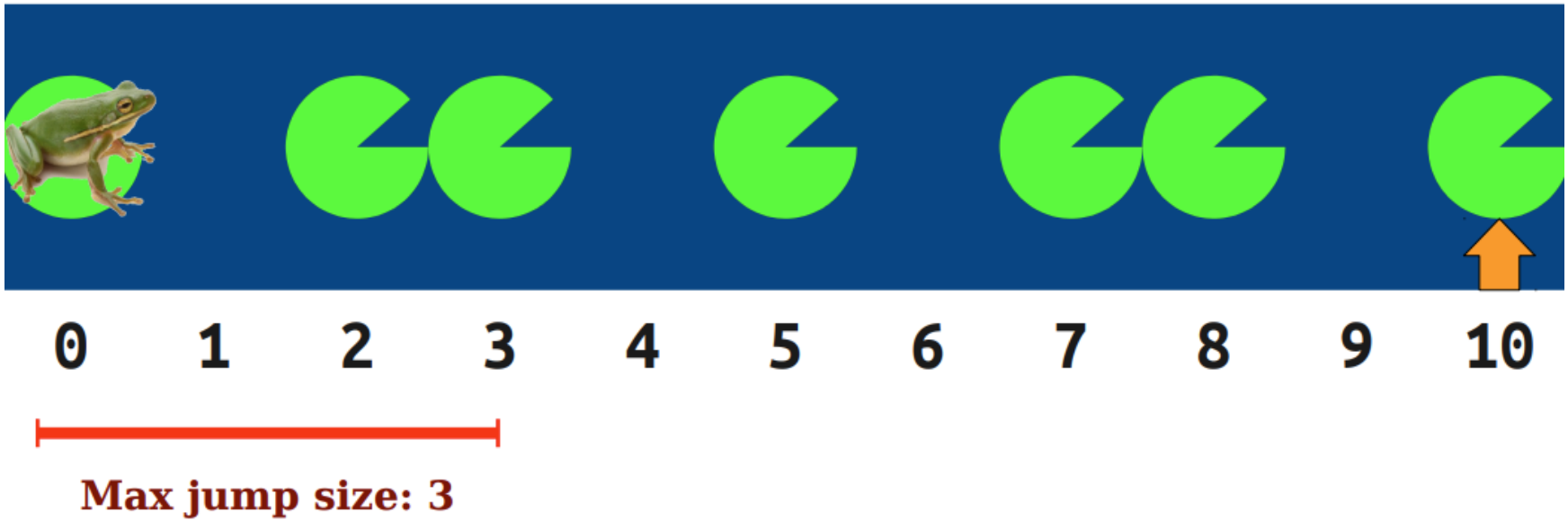
Descripción del problema

- 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

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 l la decisión voraz para P y S' la solución al problema P', entonces l 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 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 S de $P(i, f)$ que empieza por l.

Demostración. Sea S' una solución óptima que no contiene a l. Como l es el elemento más alejado de i, habrá uno o más elementos en S' que estén antes que l. Si es solo un elemento $l' < l$, entonces creamos $S = \{l\} \cup (S' - \{l'\})$. Esta S es:

- Una solución válida del problema porque si de l' se puede llegar al segundo de S' desde l porque estaría más cerca.
- Es una solución óptima porque $|S| = |S'|$ ya que estamos solo cambiando un elemento.

Si hay más de un elemento $l'_1, \ldots, l_k < l$, entonces estaríamos ante una contradicción. Pues creando $S = \{l\} \cup (S' - \{l'_1, \ldots, l_k\})$. Se tendría que $|S| < |S'|$ y S' no sería una solución óptima.

Ejemplo: el problema de la rana 🐸

Lema

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

Subestructura óptima: Sea I la decisión voraz y S' la solución de $P(I, f)$, entonces I 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 $|W| < |S' \cup \{I\}| = |S'| + 1$. Entonces:

1. Si W contiene a I como primer elemento, entonces $W - \{I\}$ sería una solución válida de $P(I, f)$ tal que $|W - \{I\}| = W - 1 < |S'|$. Lo que sería una contradicción pues S' es una solución óptima de $P(I, f)$.
2. Si W no contiene a I, entonces tiene que haber nenúfares L en W antes que I ya que I es el nenúfar más alejado posible alcanzable desde i. $W - L$ es una solución válida $P(I, f)$ tal que $|W - L| = |W| - |L| < |S'|$, contraducción.
3. Si W contiene a I pero no como primer elemento, entonces entonces tiene que haber nenúfares L en W antes que I. Así pues, $W - L - \{I\}$ es una solución válida de $P(I, f)$ tal que $|W - L - \{I\}| = |W| - |L| - 1 < |S'|$, 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.
- **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.
- **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/array de n elementos $0 \leq s_i \leq 1$. En este caso S_1 es mejor que S_2.

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 l que metemos con su proporción x_l.
2. Construimos $P(M - x_l p_l, O - \{l\})$
3. Volver a ejecutar 1 y 2 para $P(M - x_l p_l, O - \{l\})$.

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 l que metemos con su proporción x_l.
2. Construimos $P(M - x_l p_l, O - \{l\})$
3. Volver a ejecutar 1 y 2 para $P(M - x_l p_l, O - \{l\})$.

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:
        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

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...

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 poníendoselo 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.

Mochila no 0/1 🎒

- 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
```