Ramificación y Poda

Algoritmos y Estructuras de Datos
2023

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

Dada una mochila con 25 kg de capacidad, ¿cuál es el valor máximo que se puede cargar?

Item	Item 0	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6
Peso	22	9	10	6	4	2	1
Valor	19	11	9	6	1	0,5	0,1

Soluciones Óptimas

- Un problema puede contener un **gran número de entradas** y, en algunos casos, **restricciones definidas** para su solución.
- En estos casos, para alcanzar la solución al problema es necesario seleccionar un **subconjunto de estas entradas**.
- Cada uno de los subconjuntos que cumplan las restricciones diremos que son **soluciones prometedoras**.
- Una solución prometedora que **maximice o minimice** una **función objetivo** la denominaremos **solución óptima**.

Problema de la mochila (*knapsack problem*)

- Entradas: pesos y valores de cada ítem.
- Restricción: los pesos pueden sumar, a lo sumo, 25.
- Función a maximizar: suma de valores.

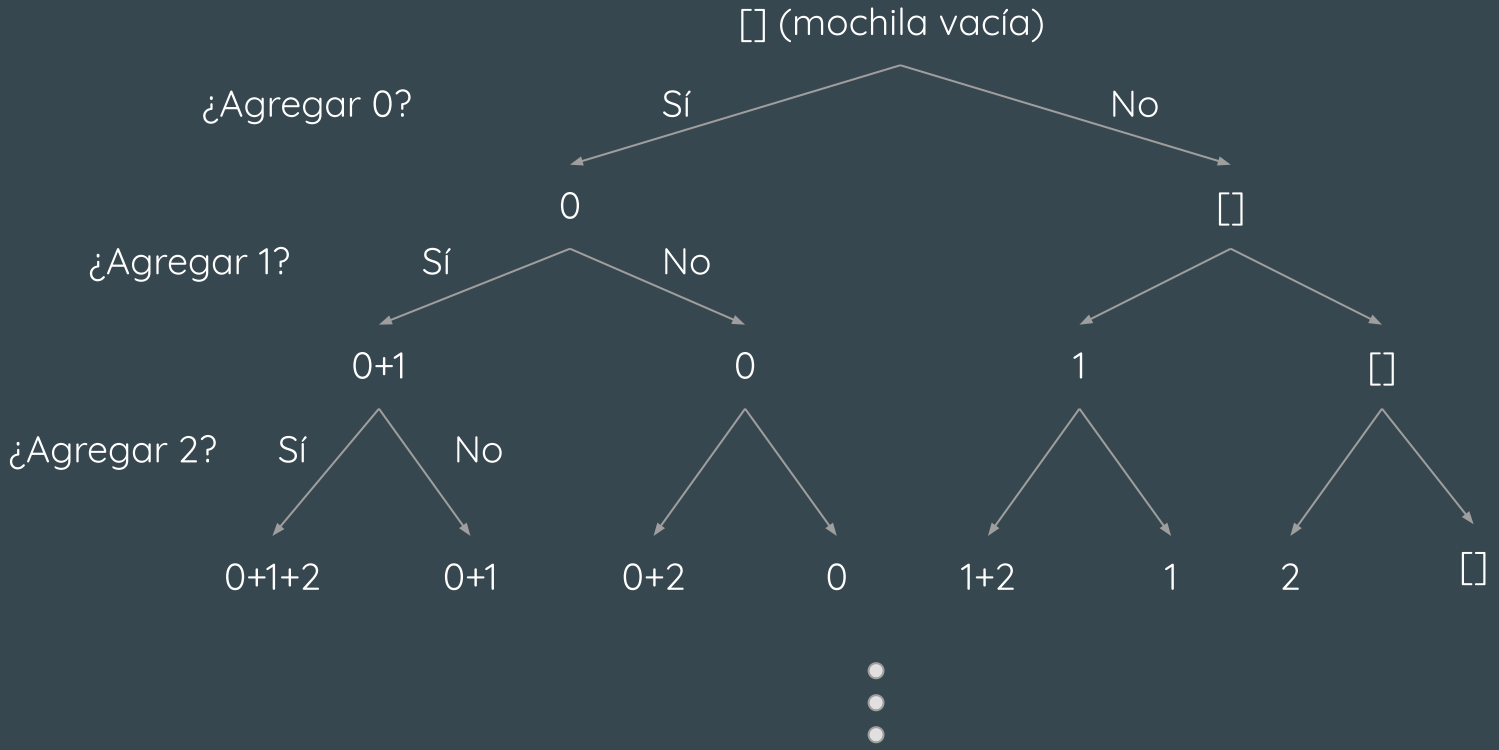
¿Cómo solucionar un problema?

1. Utilizar **fuerza bruta**

    En este caso, tenemos 7 elementos, cada uno puede estar o no en la mochila, por lo que el número de soluciones prometedoras es $2^7 = 128$ (crece exponencialmente!)

2. Utilizar **ramificación y poda**
3. Utilizar **algoritmos ávidos**
4. Utilizar **programación dinámica**

Calculando soluciones (no siempre) prometedoras



Fuerza bruta

- Implica calcular todas las posibles soluciones y después compararlas para encontrar la óptima.
- Es la alternativa que más cálculos conlleva, muchas veces de soluciones no prometedoras.
- Por ejemplo, se calcula la solución Item 0 + Item 1 + Item 2, cuando se sabe que agregar el Item 1 o 2 al Item 0 supera el peso permitido.

Codificando la solución por fuerza bruta

```
def suma(pesos, valores, soluciones, peso = 0, valor = 0, indice = 0,
combinacion = ""):
  if indice == len(pesos):
    combinacion = combinacion[:-1]
    soluciones[combinacion] = {"Peso": peso, "Valor": valor}

  else:
    # Incluyendo el valor
    comb_si = combinacion + str(indice) + "+"
    peso_si = peso + pesos[indice]
    valor_si = valor + valores[indice]
    suma(pesos, valores, soluciones, peso_si, valor_si, indice+1, comb_si)

    # No incluyendo el valor
    suma(pesos, valores, soluciones, peso, valor, indice+1, combinacion)
```

Resultados de la solución por fuerza bruta

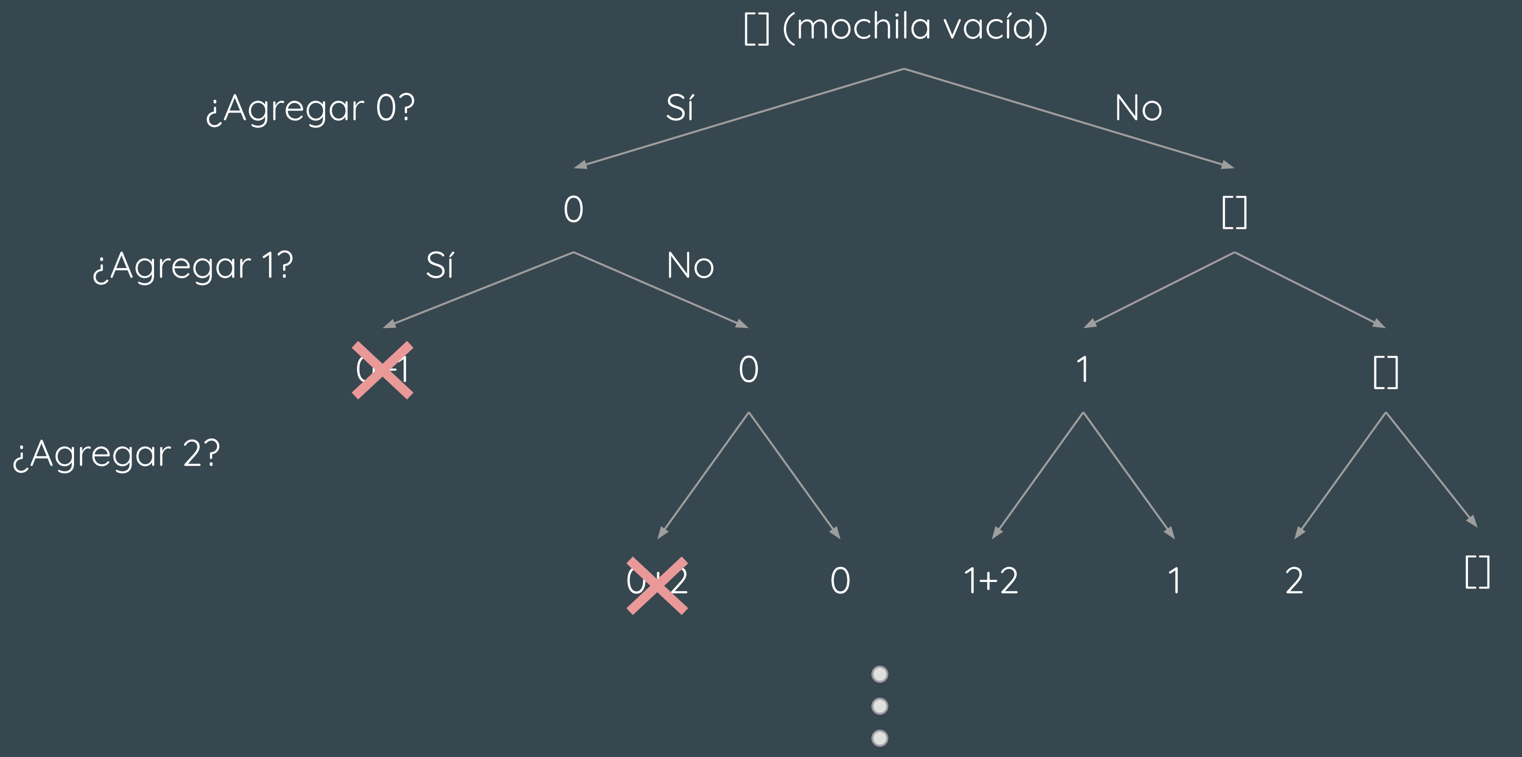
Combinación	Peso	Valor
0+1+2+3+4+5+6	54	46.6
0+1+2+3+4+5	53	46.5
0+1+2+3+4+6	52	46.1
0+1+2+3+4	51	46
0+1+2+3+5+6	50	45.6
0+1+2+3+5	49	45.5

⋮

Ramificación y Poda

- Cuando hay limitaciones para las soluciones, es deseable evitar calcular todas aquellas combinaciones que no representen soluciones prometedoras.
- De esta manera, se reduce el conjunto de soluciones calculadas, y por lo tanto, el número de cálculos totales.
- Este proceso de eliminar ramas no prometedoras del árbol de recursión se conoce como **poda**.

Calculando soluciones prometedoras



Codificando la solución usando Poda

```
def suma_poda(pesos, valores, soluciones, peso_max, peso = 0, valor = 0,
indice = 0, combinacion = ""):
  if indice == len(pesos):
    combinacion = combinacion[:-1]
    soluciones[combinacion] = {"Peso": peso, "Valor": valor}

  else:
    # Incluyendo el valor
    comb_si = combinacion + str(indice) + "+"
    peso_si = peso + pesos[indice]
    valor_si = valor + valores[indice]
    if peso_si <= peso_max:
      suma_poda(pesos, valores, soluciones, peso_max, peso_si, valor_si,
indice+1, comb_si)

    # No incluyendo el valor
    suma_poda(pesos, valores, soluciones, peso_max, peso, valor, indice+1,
combinacion)
```

Resultados de la solución usando Poda

Combinación	Peso	Valor
0	22	19
0+5	24	19.5
0+6	23	19.1
1	9	11
1+2	9	20
1+2+4	23	21

⋮

Repasando estrategias de resolución

- El uso de **fuerza bruta** es la estrategia más **simple** de plantear, pero requiere muchos cálculos (más tiempo y más memoria), ya que evalúa **todo el universo** de soluciones posibles.
- La **poda** de ramas no prometedoras es una estrategia que permite **disminuir la cantidad de cálculos** y acelerar un algoritmo (en este ejemplo, el número de soluciones baja de 128 a 57).

Repasando estrategias de resolución

- Usando **algoritmos ávidos** es posible llegar a un resultado **satisfactorio** (no siempre óptimo) calculando **una sola solución**.
- Estos algoritmos se basan en el uso de **heurísticas** para seleccionar el candidato más adecuado, en base a la búsqueda de **óptimos locales**.
- La **programación dinámica** es una estrategia que busca optimizar un algoritmo vía la **reducción del número de cálculos**.
- Para ello, se almacenan todos los resultados ya calculados en memoria y se los reutiliza en caso de que sea necesario. Este proceso se denomina **memoización**.

Bibliografía

- Guerequeta, R., & Vallecillo, A. (2019). Técnicas de diseño de algoritmos. Segunda edición. Capítulos 7