Técnicas de Diseño de Algoritmos (Ex Algoritmos y Estructuras de Datos III)

Segundo cuatrimestre 2025

Repaso 2, fuerza bruta y backtracking

Problemas bien resueltos

Convención. Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

Problemas bien resueltos

Convención. Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

No obstante ...

Problemas bien resueltos

Convención. Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

No obstante ...

- Si los tamaños de instancia son pequeños, ¿es tan malo un algoritmo exponencial?
- ¿Cómo se comparan $O(n^{85})$ con $O(1{,}001^n)$?
- ¿Puede pasar que un algoritmo de peor caso exponencial sea eficiente en la práctica? ¿Puede pasar que en la práctica sea *el mejor*?
- ¿Qué pasa si no encuentro un algoritmo polinomial?

Problemas de optimización

- Un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

Problemas de optimización

- Un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

- La función $f : S \to \mathbb{R}$ se denomina función objetivo del problema.

Problemas de optimización

- Un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

- La función $f : S \to \mathbb{R}$ se denomina función objetivo del problema.
- El conjunto S es la región factible y los elementos $x \in S$ se llaman soluciones factibles.

Problemas de optimización

- Un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

- La función $f : S \to \mathbb{R}$ se denomina función objetivo del problema.
- El conjunto S es la región factible y los elementos $x \in S$ se llaman soluciones factibles.
- El valor $z^* \in \mathbb{R}$ es el valor óptimo del problema, y cualquier solución factible $x^* \in S$ tal que $f(x^*) = z^*$ se llama un óptimo del problema.

Problemas de optimización combinatoria

- Un problema de optimización combinatoria es un problema de optimización cuya región factible es un conjunto definido por consideraciones combinatorias (!).

Problemas de optimización combinatoria

- Un problema de optimización combinatoria es un problema de optimización cuya región factible es un conjunto definido por consideraciones combinatorias (!).
- La combinatoria es la rama de la matemática discreta que estudia la construcción, enumeración y existencia de configuraciones de objetos finitos que satisfacen ciertas propiedades.

Problemas de optimización combinatoria

- Un problema de optimización combinatoria es un problema de optimización cuya región factible es un conjunto definido por consideraciones combinatorias (!).
- La combinatoria es la rama de la matemática discreta que estudia la construcción, enumeración y existencia de configuraciones de objetos finitos que satisfacen ciertas propiedades.
- Por ejemplo, regiones factibles dadas por todos los subconjuntos/permutaciones de un conjunto finito de elementos (posiblemente con alguna restricción adicional), todos los caminos en un grafo, etc.

Algoritmos de fuerza bruta

- Un algoritmo de fuerza bruta para un problema de optimización combinatoria consiste en generar todas las soluciones factibles y quedarse con la mejor.

Algoritmos de fuerza bruta

- Un algoritmo de fuerza bruta para un problema de optimización combinatoria consiste en generar todas las soluciones factibles y quedarse con la mejor.

  1. Se los suele llamar también algoritmos de búsqueda exhaustiva o generate and test.
  2. Se trata de una técnica trivial pero muy general.
  3. Suele ser fácil de implementar, y es un algoritmo exacto: si hay solución, siempre la encuentra.

Algoritmos de fuerza bruta

- Un algoritmo de fuerza bruta para un problema de optimización combinatoria consiste en generar todas las soluciones factibles y quedarse con la mejor.
    1. Se los suele llamar también algoritmos de búsqueda exhaustiva o generate and test.
    2. Se trata de una técnica trivial pero muy general.
    3. Suele ser fácil de implementar, y es un algoritmo exacto: si hay solución, siempre la encuentra.
- El principal problema de este tipo de algoritmos es su complejidad. Habitualmente, un algoritmo de fuerza bruta tiene una complejidad exponencial.

Ejemplo: El problema de la mochila

Datos de entrada:

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{Z}_+$ de objetos.
- Peso $p_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \ldots, n$.
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \ldots, n$.

Ejemplo: El problema de la mochila

Datos de entrada:

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{Z}_+$ de objetos.
- Peso $p_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \ldots, n$.
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \ldots, n$.

Problema: Determinar qué objetos debemos incluir en la mochila sin excedernos del peso máximo C, de modo tal de maximizar el beneficio total entre los objetos seleccionados.

Ejemplo: El problema de la mochila

- ¿Cómo es un algoritmo de fuerza bruta para el problema de la mochila?

Ejemplo: El problema de la mochila

- ¿Cómo es un algoritmo de fuerza bruta para el problema de la mochila?
- ¿Cómo se implementa este algoritmo?

Ejemplo: El problema de la mochila

- ¿Cómo es un algoritmo de fuerza bruta para el problema de la mochila?
- ¿Cómo se implementa este algoritmo?

```
MOCHILA(S ⊆ {1, ..., n}, k : ℤ)
if k = n + 1 then
    if peso(S) ≤ C ∧ beneficio(S) > beneficio(B) then
        B ← S
    end if
else
    MOCHILA(S ∪ {k}, k + 1);
    MOCHILA(S, k + 1);
end if
```

Ejemplo: El problema de la mochila

- ¿Cómo es un algoritmo de fuerza bruta para el problema de la mochila?
- ¿Cómo se implementa este algoritmo?

$\textsc{Mochila}(S \subseteq \{1, \ldots, n\},\ k : \mathbb{Z})$

```
if k = n + 1 then
    if peso(S) ≤ C ∧ beneficio(S) > beneficio(B) then
        B ← S
    end if
else
    MOCHILA(S ∪ {k}, k + 1);
    MOCHILA(S, k + 1);
end if
```

- Iniciamos la recursión con $B \leftarrow \emptyset$; $\textsc{Mochila}(\emptyset,\ 1)$.

Ejemplo: El problema de la mochila

- ¿Cómo es un algoritmo de fuerza bruta para el problema de la mochila?

- ¿Cómo se implementa este algoritmo?

$\textsc{Mochila}(S \subseteq \{1, \ldots, n\},\ k : \mathbb{Z})$

---

if $k = n + 1$ **then**
    if peso$(S) \leq C \wedge$ beneficio$(S) >$ beneficio(B) **then**
        $B \leftarrow S$
    end if
else
    $\textsc{Mochila}(S \cup \{k\},\ k + 1)$;
    $\textsc{Mochila}(S,\ k + 1)$;
end if

- Iniciamos la recursión con $B \leftarrow \emptyset$; $\textsc{Mochila}(\emptyset,\ 1)$.

- ¿Cuál es la complejidad computacional de este algoritmo?

Ejemplo: El problema de la mochila

- **Idea.** Podemos interrumpir la recursión cuando el subconjunto actual excede la capacidad de la mochila!

Ejemplo: El problema de la mochila

- **Idea.** Podemos interrumpir la recursión cuando el subconjunto actual excede la capacidad de la mochila!

$\textsc{Mochila}(S \subseteq \{1, \dots, n\},\ k : \mathbb{Z})$

---

if $k = n + 1$ **then**
  if $\text{peso}(S) \leq C \wedge \text{beneficio}(S) > \text{beneficio}(B)$ **then**
    $B \leftarrow S$
  end if
else if $\text{peso}(S) \leq C$ **then**
  $\textsc{Mochila}(S \cup \{k\},\ k + 1)$;
  $\textsc{Mochila}(S,\ k + 1)$;
end if

- Con este agregado, decimos que tenemos un backtracking.
- ¿Cuál es la complejidad computacional de este algoritmo?

Ejemplo: El problema de la mochila

- Podemos implementar alguna otra poda?

Ejemplo: El problema de la mochila

- Podemos implementar alguna otra poda?

$\textsc{Mochila}(S \subseteq \{1, \ldots, n\},\ k : \mathbb{Z})$

---

if $k = n + 1$ **then**
    if peso$(S) \leq C \wedge$ beneficio$(S) >$ beneficio(B) **then**
        $B \leftarrow S$
    end if
else if peso$(S) \leq C \wedge$ benef$(S) + \sum_{i=k+1}^{n} b_i >$ benef(B)
then
    $\textsc{Mochila}(S \cup \{k\},\ k + 1)$;
    $\textsc{Mochila}(S,\ k + 1)$;
end if

- Este tipo de algoritmos se denomina habitualmente branch and bound.

Backtracking

Idea: Recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional, eliminando las configuraciones parciales que no puedan completarse a una solución.

Backtracking

Idea: Recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional, eliminando las configuraciones parciales que no puedan completarse a una solución.

- Habitualmente, utiliza un vector $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata, cada a_i pertenece un dominio/conjunto ordenado y finito A_i.
- El espacio de soluciones es el producto cartesiano $A_1 \times \ldots \times A_n$.

Backtracking

- En cada paso se extienden las soluciones parciales $a = (a_1, a_2, \ldots, a_k)$, $k < n$, agregando un elemento más, $a_{k+1} \in S_{k+1} \subseteq A_{k+1}$, al final del vector a. Las nuevas soluciones parciales son sucesores de la anterior.
- Si S_{k+1} es vacío, se *retrocede* a la solución parcial $(a_1, a_2, \ldots, a_{k-1})$.
- Se puede pensar este espacio como un árbol dirigido, donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y.
- Permite descartar configuraciones antes de explorarlas (podar el árbol).

Backtracking: Todas las soluciones

```
algoritmo BT(a,k)
    si  a es solución entonces
        procesar(a)
        retornar
    sino
        para cada a' ∈ Sucesores(a,k)
            BT(a',k+1)
        fin para
    fin si
    retornar
```

Backtracking: Una solución

```
algoritmo BT(a,k)
    si a es solución entonces
        sol ← a
        encontro ← true
    sino
        para cada a' ∈ Sucesores(a,k)
            BT(a',k+1)
            si encontro entonces
                retornar
            fin si
        fin para
    fin si
    retornar
```

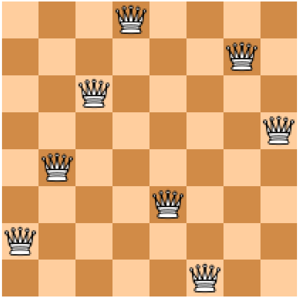
Backtracking - Resolver un *sudoku*

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

El problema de resolver un *sudoku* se resuelve en forma muy eficiente con un algoritmo de *backtracking* (no obstante, el peor caso es exponencial!).

Fuerza bruta - Problema de las n damas



Problema: Ubicar n damas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna dama amenace a otra.

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- Un algoritmo de fuerza bruta (también llamado de búsqueda exhaustiva) analiza todas las posibles "configuraciones", lo cual habitualmente implica una complejidad exponencial.

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- Un algoritmo de fuerza bruta (también llamado de búsqueda exhaustiva) analiza todas las posibles "configuraciones", lo cual habitualmente implica una complejidad exponencial.
- Por ejemplo, para $n = 8$ una implementación directa consiste en generar todos los subconjuntos de casillas.

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- Un algoritmo de fuerza bruta (también llamado de búsqueda exhaustiva) analiza todas las posibles "configuraciones", lo cual habitualmente implica una complejidad exponencial.
- Por ejemplo, para $n = 8$ una implementación directa consiste en generar todos los subconjuntos de casillas.

$$2^{64} = 18,446,744,073,709,551,616 \text{ combinaciones!}$$

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- Un algoritmo de fuerza bruta (también llamado de búsqueda exhaustiva) analiza todas las posibles "configuraciones", lo cual habitualmente implica una complejidad exponencial.
- Por ejemplo, para $n = 8$ una implementación directa consiste en generar todos los subconjuntos de casillas.

$$2^{64} = 18,446,744,073,709,551,616 \text{ combinaciones!}$$

- Sabemos que dos damas no pueden estar en la misma casilla.

Fuerza bruta - Problema de las n damas

- Solución por **fuerza bruta**: hallar todas las formas posibles de colocar n damas en un tablero de $n \times n$ y luego seleccionar las que satisfagan las restricciones.
- Un algoritmo de fuerza bruta (también llamado de búsqueda exhaustiva) analiza todas las posibles "configuraciones", lo cual habitualmente implica una complejidad exponencial.
- Por ejemplo, para $n = 8$ una implementación directa consiste en generar todos los subconjuntos de casillas.

$$2^{64} = 18,446,744,073,709,551,616 \text{ combinaciones!}$$

- Sabemos que dos damas no pueden estar en la misma casilla.

$$\binom{64}{8} = 4,426,165,368 \text{ combinaciones.}$$

Fuerza bruta - Problema de las n damas

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

Fuerza bruta - Problema de las *n* damas

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

Tenemos ahora $8^8 = 16,777,216$ combinaciones.

Fuerza bruta - Problema de las n damas

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

  Tenemos ahora $8^8 = 16,777,216$ combinaciones.

- Adicionalmente, cada fila debe tener exactamente una dama.

Fuerza bruta - Problema de las n damas

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

  Tenemos ahora $8^8 = 16,777,216$ combinaciones.

- Adicionalmente, cada fila debe tener exactamente una dama.

  Se reduce a $8! = 40,320$ combinaciones.

Fuerza bruta - Problema de las n damas

- Sabemos que cada columna debe tener exactamente una dama. Cada solución parcial puede estar representada por $(a_1, \ldots, a_k)$, $k \leq 8$, con $a_i \in \{1, \ldots, 8\}$ indicando la fila de la dama que está en la columna i.

  Tenemos ahora $8^8 = 16,777,216$ combinaciones.

- Adicionalmente, cada fila debe tener exactamente una dama.

  Se reduce a $8! = 40,320$ combinaciones.

- Esto está mejor, pero se puede mejorar observando que no es necesario analizar muchas de estas combinaciones (¿por qué?).