Algoritmos y Estructuras de Datos III

Segundo cuatrimestre 2022

(bienvenidos!)

Programa

1. Técnicas de diseño de algoritmos.
2. Introducción a la teoría de grafos y algoritmos sobre grafos.
3. Problema de árbol generador mínimo.
4. Problema de camino mínimo.
5. Problemas de flujo en redes.
6. Introducción a la teoría de NP-completitud.

Bibliografía

1. G. Brassard and P. Bratley, *Fundamentals of algorithmics*, Prentice-Hall, 1996.
2. F. Harary, *Graph theory*, Addison-Wesley, 1969.
3. R. Ahuja, T. Magnanti and J. Orlin, *Network flows: theory, algorithms, and applications*, Prentice-Hall, 1993.
4. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W. Freeman and Co., 1979.

Régimen de cursada

- Cursada:
  1. Lunes: clases teóricas.
  2. Miércoles y viernes: clases prácticas y de laboratorio.
     (con algunas excepciones!)
- Evaluaciones:
  1. Dos parciales (individuales).
  2. Dos trabajos prácticos (en grupo).
  3. Un examen final (individual).
- Espacio en el campus virtual (campus.exactas.uba.ar), y comunicación por correo electrónico:
  1. Lista de docentes: `algo3-doc@dc.uba.ar`.
  2. Lista de alumnos: `algo3-alu@dc.uba.ar`.

Complejidad computacional: Las reglas del juego

- En el contexto de la teoría de complejidad computacional, llamamos problema a la descripción de los datos de entrada y la respuesta a proporcionar para cada dato de entrada.
- Una instancia de un problema es un juego válido de datos de entrada.
- Ejemplo:
  1. **Entrada:** Un número n entero no negativo.
  2. **Salida:** ¿El número n es primo?
- En este ejemplo, una instancia está dada por un número entero no negativo.

Complejidad computacional: Las reglas del juego

- Suponemos una Máquina RAM (*random access memory*).
  1. La memoria está dada por una sucesión de celdas numeradas. Cada celda puede almacenar un valor de b bits.
  2. Supondremos habitualmente que el tamaño b en bits de cada celda está fijo, y suponemos que todos los datos individuales que maneja el algoritmo se pueden almacenar con b bits.
  3. Se tiene un programa imperativo no almacenado en memoria, compuesto por asignaciones y las estructuras de control habituales.
  4. Las asignaciones pueden acceder a celdas de memoria y realizar las operaciones estándar sobre los tipos de datos primitivos habituales.

Complejidad computacional: Las reglas del juego

- Cada instrucción tiene un tiempo de ejecución asociado.

  1. El acceso a cualquier celda de memoria, tanto para lectura como para escritura, es $O(1)$.
  2. Las asignaciones y el manejo de las estructuras de control se realiza en $O(1)$.
  3. Las operaciones entre valores lógicos son $O(1)$.

- Las operaciones entre enteros/reales dependen de b:

  1. Las sumas y restas son $O(b)$.
  2. Las multiplicaciones y divisiones son $O(b \log b)$.

$\Rightarrow$ Si b está fijo, estas operaciones son $O(1)$. En cambio, si no se puede suponer esto, entonces hay que contemplar que el costo de estas operaciones depende de b.

Complejidad computacional: Las reglas del juego

- **Tiempo de ejecución de un algoritmo A:**
  $T_A(I) =$ suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la *instancia* I.

- Dada una instancia I, definimos $|I|$ como la cantidad de bits necesarios para almacenar los datos de entrada de I.

  1. Si b está fijo y la entrada ocupa n celdas de memoria, entonces $|I| = bn = O(n)$.

- **Complejidad de un algoritmo A:**
  $f_A(n) = \text{máx}_{I:|I|=n} T_A(I)$.

Repaso: Notación O

Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$, decimos que:

- $f(n) = O(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que $f(n) \leq c\, g(n)$ para todo $n \geq n_0$.
- $f(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que $f(n) \geq c\, g(n)$ para todo $n \geq n_0$.
- $f(n) = \Theta(g(n))$ si $f = O(g(n))$ y $f = \Omega(g(n))$.

Repaso: Notación O

- Si un algoritmo es $O(n)$, se dice lineal.
- Si un algoritmo es $O(n^2)$, se dice cuadrático.
- Si un algoritmo es $O(n^3)$, se dice cúbico.
- Si un algoritmo es $O(n^k)$, $k \in \mathbb{N}$, se dice polinomial.
- Si un algoritmo es $O(\log n)$, se dice logarítmico.
- Si un algoritmo es $O(d^n)$, $d \in \mathbb{R}_+$, se dice exponencial.

- Cualquier función exponencial es *peor* que cualquier función polinomial: Si $k, d \in \mathbb{N}$ entonces k^n no es $O(n^d)$.

- La función logarítmica es *mejor* que la función lineal (no importa la base), es decir $\log n$ es $O(n)$ pero no a la inversa.

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:

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

- 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 (!).
- 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.

  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 $i = 1, \ldots, n$.

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?
- ¿Cómo se implementa este algoritmo?

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

---

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
    $\textsc{Mochila}(S \cup \{k\},\ k + 1)$;
    $\textsc{Mochila}(S,\ k + 1)$;
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!

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

---

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?

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

---

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.

- 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
```

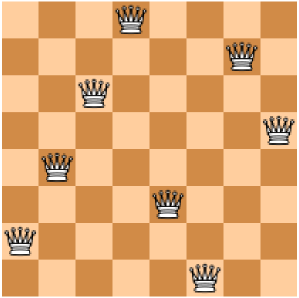
Backtracking - Resolver un *sudoku*

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.
- 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.

  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?).