Fuerza bruta + Backtracking

Tecnología Digital V: Diseño de Algoritmos

Universidad Torcuato Di Tella

Motivación: GPS, aplicaciones y movilidad

Problema

- Queremos ir desde UTDT al Obelisco.
- Vamos a usar una plataframa o una *app* (Google Maps, Waze, etc.) para tomar la decisión sobre como movernos.
- Queremos llegar en el menor tiempo posible.

Preguntas

- ¿Cómo podemos abordar el problema?
- ¿Cuántos caminos posibles tenemos?
- ¿Cuánto estamos dispuestos a esperar por una respuesta?

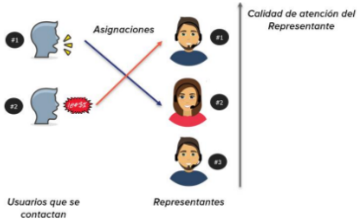
Motivación: Customer Service vs. Customer Experience

Customer Service

Es posible mejorar la *experiencia del usuario* para minimizar el número de **detractores** en una plataforma mediante el manejo eficiente de quejas y reclamos por los equipos de soporte al cliente?

En la práctica

En un determinado instante, un conjunto de usuarios que contactan a soporte al cliente tienen que ser atendidos por representantes con diferentes habilidades, experiencia, etc.



Preguntas

- ¿Cómo podemos abordar el problema?
- ¿Cuántas asignaciones posibles tenemos?
- ¿Cuánto estamos dispuestos a esperar por una respuesta?

Motivación: Problema de la mochila

Problema

- Tenemos una mochila con un peso máximo limitado.
- Tenemos un conjunto de ítems, cada uno de ellos proporciona un beneficio y tiene un peso asociado.
- Buscamos elegir qué elementos incluir en la mochila de forma tal que el beneficio sea máximo sin exceder el peso máximo.

Preguntas

- ¿Cómo podemos abordar el problema?
- ¿Cuántas soluciones posibles tenemos?
- ¿Cuánto estamos dispuestos a esperar por una respuesta?

Definición

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

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

donde

- 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.

Pregunta

¿Cómo se define S en cada uno de los problemas mencionados anteriormente?

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.

Primera aproximación: algoritmos de fuerza bruta

Definición

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.

Observación

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

Knapsack-01 (KP-01)

Debemos llenar una mochila eligiendo entre varios objetos posibles. Cada producto tiene un peso, una medida de comfort (beneficio) y la mochila tolera un peso máximo de carga. **Los objetos no pueden ser fraccionados, y solo se puede elegir una unidad de cada objeto.**

Una instancia del KP-01 está dada por

- $N = \{1, \ldots, n\}$ el conjunto de objetos (o productos).
- $p_i \in \mathbb{Z}_+$ el peso del objeto i, para $i = 1, \ldots, n$.
- $b_i \in \mathbb{Z}_+$ el beneficio del objeto i, para $i = 1, \ldots, n$.
- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).

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

Instancia

- $n = 8$
- $b = (b_i) = (15, 100, 90, 60, 40, 15, 10, 1)$
- $p = (p_i) = (2, 20, 20, 30, 40, 30, 60, 10)$
- $C = 102$

Solución factible

- $S = \{1, 2, 3, 4, 6\}$
- $z^* = 280$
- Capacidad usada: 102

Preguntas

1. Qué podemos decir de un objeto con $b_i < 0$?
2. Cómo podemos explorar todas las soluciones?

Acá en S se agarra el objeto 1, 2, 3, 4, 6.
Es decir, se suman los beneficios 15+100+90+60+15 = 280
Y sus pesos <= C 2+20+20+30+30 = 102

Probás todas las combinaciones posibles de los elementos para poner en la mochila.

- Cómo es un algoritmo de fuerza bruta para el problema de la mochila?
- Cómo se implementa este algoritmo?
- Representamos una solución como una secuencia de decisiones $S = (s_1, \ldots, s_n)$. La decisión $s_i \in \{0, 1\}$ consiste en determinar si el objeto i está incluido en la solución o no.
- Un algoritmo de fuerza bruta consiste en generar todas las secuencias posibles de decisiones que representen soluciones factibles, y quedarse con la mejor.

Hay n objetos, cada uno con su peso y beneficio

$\textsc{Mochila}(S \in \{0,1\}^k, k \in \mathbb{N})$ S (vector binario), k (índice del objeto)

if $k = n$ **then**
    if $\text{peso}(S) \leq C \wedge \text{beneficio}(S) > \text{beneficio}(B)$ **then**
        $B \leftarrow S$
    end if
else Caso recursivo: no sabemos qué hacer con el objeto k
    $S' \leftarrow S + [1];$
    $\textsc{Mochila}(S', k+1);$ ▷ Corresponde a $s_k = 1$
    $S' \leftarrow S + [0];$
    $\textsc{Mochila}(S', k+1);$ ▷ Corresponde a $s_k = 0$
end if

beneficio(S) = suma de beneficios

B es la mejor solución encontrada hasta ahora

- Iniciamos la recursión con $B \leftarrow []$; $\textsc{Mochila}([], 1)$.
- Cuál es la complejidad computacional de este algoritmo?

El algoritmo recorre todas las combinaciones posibles de los n objetos.
Cada camino es una solución candidata.
Se guarda la mejor solución global (con fuerza bruta)

Idea

Podemos interrumpir la recursión si una solución parcial cumple ciertas propiedades.

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

Ejemplo: El problema de la mochila

$\textsc{Mochila}(S \in \{0,1\}^k, k \in \mathbb{N})$

if $k = n$ **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**   Nueva condición de parada
  $S' \leftarrow S + [1];$
  $\textsc{Mochila}(S', k+1);$ ▹ Corresponde a $s_k = 1$
  $S' \leftarrow S + [0];$
  $\textsc{Mochila}(S', k+1);$ ▹ Corresponde a $s_k = 0$
end if

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

Idea

Podemos interrumpir la recursión si una solución parcial cumple ciertas propiedades.

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

- **Poda por optimalidad.** Podemos evitar explorar soluciones descendientes de un subconjunto si lo mejor que podemos obtener no supera la mejor solución conocida.

Ejemplo: El problema de la mochila

$\textsc{Mochila}(S \in \{0,1\}^k, k \in \mathbb{N})$

```
if k = n then
    if peso(S) ≤ C ∧ beneficio(S) > beneficio(B) then
        B ← S
    end if
else if peso(S) ≤ C ∧ benef(S) + Σ_{i=k+1}^{n} b_i > benef(B) then
    S' ← S + [1];
    Mochila(S', k + 1);                          ▷ Corresponde a s_k = 1
    S' ← S + [0];
    Mochila(S', k + 1);                          ▷ Corresponde a s_k = 0
end if
```

Nueva condición: $\wedge\ \text{benef}(S) + \sum_{i=k+1}^{n} b_i > \text{benef}(B)$

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

Generalizando el ejemplo

- Intentemos abstraer los elementos de este algoritmo.

BACKTRACKING($S \in D_1 \times \cdots \times D_k$)

---

if $k = n$ **then**
  if S es factible $\wedge$ S es mejor que B **then**
    $B \leftarrow S$
  end if
else
  for cada continuación posible s de S **do**
    $S' \leftarrow S + [s]$;
    BACKTRACKING(S');
  end for
end if

Generalizando el ejemplo

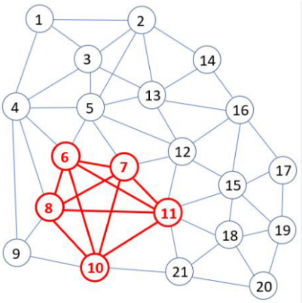
- En muchos problemas alcanza con encontrar una solución factible. En estos casos, podemos interrumpir la recursión apenas encontramos la primera solución factible.

BACKTRACKING($S \in D_1 \times \cdots \times D_k$)

---

if $k = n$ **then**
    if S es factible **then**
        Print(S);
        encontramos $\leftarrow$ **true**;
    end if
else
    for cada continuación posible s de S **do**
        $S' \leftarrow S + [s]$;
        BACKTRACKING(S');
        if encontramos == **true then**
            return
        end if
    end for
end if

Ejemplo: Clique máxima en grafos



Definición

Recordemos que una clique en un grafo $G = (V, E)$ es un conjunto $K \subseteq V$ tal que $ij \in E$ para todo $i, j \in K$, $i \neq j$.

Max-Clique

Dado un grafo G, encontrar una clique de tamaño máximo de G.

Ejemplo: Clique máxima en grafos

- Cómo es un algoritmo de fuerza bruta para el problema de clique máxima?
- Cómo se implementa este algoritmo?
- Representamos una solución como una secuencia de decisiones $S = (s_1, \dots, s_n)$. La decisión $s_i \in \{0, 1\}$ consiste en determinar si el vértice i está incluido en la solución o no.
- Un algoritmo de fuerza bruta consiste en generar todas las secuencias posibles de decisiones que representen soluciones factibles, y quedarse con la mejor.

Ejemplo: Clique máxima en grafos

```
MaxClique(S ∈ {0,1}^k)
if k = n then
    if esClique(S) ∧ |S| > |B| then
        B ← S
    end if
else
    S′ ← S + [1];
    MaxClique(S′);                    ▷ Corresponde a s_k = 1
    S′ ← S + [0];
    MaxClique(S′);                    ▷ Corresponde a s_k = 0
end if
```

- Iniciamos la recursión con $B \leftarrow []$; MaxClique([]).
- Cuál es la complejidad computacional de este algoritmo?
- Podemos aplicar alguna poda?

Ejemplo: Clique máxima en grafos

$\textsc{MaxClique}(S \in \{0,1\}^k)$

if $k = n$ **then**
  if esClique(S) $\wedge |S| > |B|$ **then**
    $B \leftarrow S$
  end if
else if esClique(S) **then**
  $S' \leftarrow S + [1]$;
  $\textsc{MaxClique}(S')$; ▷ Corresponde a $s_k = 1$
  $S' \leftarrow S + [0]$;
  $\textsc{MaxClique}(S')$; ▷ Corresponde a $s_k = 0$
end if

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

Ejemplo: Clique máxima en grafos

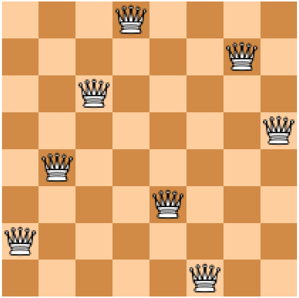
MaxClique($S \in \{0,1\}^k$)

---

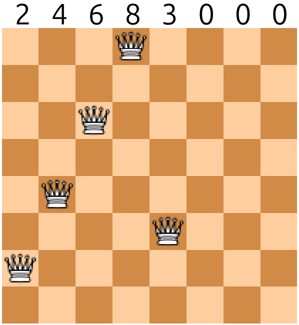
if $k = n$ **then**
    if esClique(S) $\wedge |S| > |B|$ **then**
        $B \leftarrow S$
    end if
else if esClique(S) $\wedge |S| + (n - k) > |B|$ **then**
    $S' \leftarrow S + [1]$;
    MaxClique(S'); ▷ Corresponde a $s_k = 1$
    $S' \leftarrow S + [0]$;
    MaxClique(S'); ▷ Corresponde a $s_k = 0$
end if

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

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.
- ¿Cómo podemos representar una solución como una secuencia de decisiones $S = (s_1, \ldots, s_n)$?



- **Ejemplo:** Representamos esta solución parcial con la secuencia de decisiones $S = (2, 4, 6, 8, 3)$.
- ¿Qué valores son válidos para s_5?

- Podemos plantear el siguiente pseudocódigo, recordando que representamos las filas y columnas comenzando desde el índice 0.

DAMAS($S \in \{1, \ldots, 8\}^k$)

---

if $k = 8$ **then**
    Print(S);
    return
else
    for $j = 1, \ldots, 8$ **do**
        if la casilla $(k + 1, j)$ no está amenazada en S **then**
            $S' \leftarrow S + [j]$;
            DAMAS(S');
        end if
    end for
end if

- En muchos casos podemos evitar la generación de S' a partir de S modificando el mismo S (y deshaciendo el cambio!).

```
DAMAS(S ∈ {1, ..., 8}^k)
if k = 8 then
    Print(S);
    return
else
    for j = 1, ..., 8 do
        if la casilla (k + 1, j) no está amenazada en S then
            Agregar j al final de S;
            DAMAS(S);
            Eliminar j de S;
        end if
    end for
end if
```

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!).