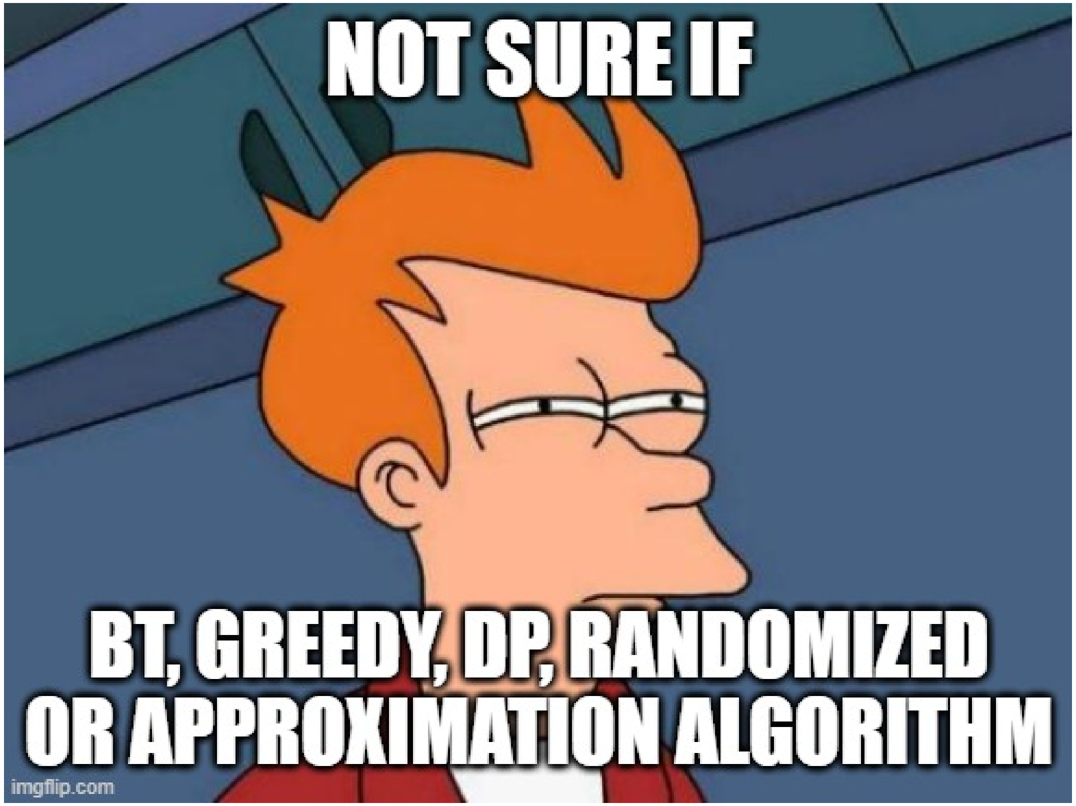
Algoritmos y complejidad



Ayudantía 10

Temario

- Repaso Programación Dinámica
  - 0-1 Knapsack problem
- Algoritmos aproximados
  - Traveling Salesman Problem
- Polynomial approximation schemes
  - Subset sum

Programación dinámica

- Aplicamos la estrategia dividir y conquistar cuando podemos separar un problema en subproblemas. La idea es resolver los problemas recursivamente y luego combinar las soluciones.
- Cuando estos subproblemas se solapan y terminamos solucionando los mismos una y otra vez es mejor utilizar **programación dinámica**.
- Es un método para resolver probemas (de optimización generalmente) con un enfoque Bottom-up; cuando una solución se requiera ya estará calculada. ❤️

> Recordar Fibonnaci, o Dijsktra.

Cuándo podemos aplicarla?

Necesitamos que el problema cumpla dos propiedades:

1. Optimal subestructure: Mediante óptimos para subproblemas, se puede llegar a un óptimo para el problema general.
2. Overlapping problems.

> Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (s. f.). Introduction to algorithms (3.a ed.). MIT Press. ❤️

0-1 Knapsack problem

- Consideremos una mochila con capacidad máxima W.
- Tenemos n items, cada item i tiene un valor v_i y un peso w_i asociado.
- Cada item, lo podemos llevar o no. No podemos tomar una fracción de él. 🤔
- Buscamos maximizar $\sum_{i=0}^{n} v_i$ sujeto a $\sum_{i=0}^{n} w \leq W$.

- La idea será construir una matriz $V[0..n, 0..W]$.

> Memoization data structure.

- El elemento $V[i, j]$ almacena el **máximo valor** para cualquier subconjunto de items $\{1 \ldots i\}$ que **pueden caber** en una mochila de capacidad j.
- Notar que la solución vendrá dada por $V[n, W]$.
- Notar que $V[0, j] = 0, \forall j \in 1 \ldots W$.

Para cada item i, podemos considerarlo o no:

- Si lo consideramos, el valor óptimo vendrá dado por cómo llenar la mochila de capacidad $j - w_i$ con los items restantes $\{1, \ldots, i-1\}$:

$$v_i + V[i-1, j-w_i]$$

- De lo contrario, el valor óptimo vendrá dado por cómo llenar la mochila de capacidad j con los items restantes $\{1, \ldots, i-1\}$:

$$V[i-1, j]$$

- Asi, obtenemos la siguiente recurrencia:

$$V[i,j] = \begin{cases} V[i-1,j] & \text{if } w_i > j \\ max(v_i + V[i-1, j-w_i], V[i-1,j]) & \text{if } w_i \leq j \end{cases}$$

- Ademas, sabemos $V[0,j] = 0$

> Si no tenemos items, entonces no tendremos ningún valor asociado.

- Contamos con una recurrencia. Los siguientes pasos son estandar para disenar el algoritmo basado en DP:

1. Formular el problema recursivamente. ✅
2. Identificar subproblemas.
3. Elegir la estructura de datos para la memoización.
4. Identificar dependencias.
5. Encontrar un buen orden de evaluación.
6. Analizar la complejidad espacial y temporal.
7. Escribir el algoritmo.

Solución en Python

```
def solve_knapsack(v,w,n,W):
    V = np.zeros((n, W))
    for i in range(1, n):
        for j in range(W):
            leave_val = V[i - 1][j]
            if( j >= w[i]):
                take_val = v[i] + V[i - 1][j - w[i]]
            else:
                take_val = - np.inf
            V[i][j] = max(leave_val, take_val)
    return V[n - 1][W - 1]
```

Algoritmos aproximados

Definicion

Algoritmo con tiempo de ejecución polinomial idealmente, que produce una solución que está garantizada a estar dentro de un factor de optimalidad. ❤️

Decimos que un algoritmo para un problema de tamano n tiene un ratio de aproximación $\rho(n)$, si para cada input produce una solución con costo C tal que:

$$max(\frac{C}{C*}, \frac{C*}{C}) \leq \rho(n)$$

Donde C^* es el costo óptimo.

> $\rho(n)$ puede ser constante, o bien una función creciente.
> Notar que hablamos de costos debido a el enfoque hacia los problemas de optimización.

¿Por qué algoritmos aproximados?

- Estamos interesados en aproximar soluciones a problemas NP completos con algoritmos polinomiales.
- Plantearemos una heuristica greedy interesante y demostraremos que en cualquier caso obtendremos una solución dentro de un factor de optimalidad.
- Por ejemplo, para un problema de minimización podriamos obtener un algoritmo que entrega un resultado que es a lo más el doble del optimo.

Idea general

1. Definir el problema.
2. Pensar en una heurística interesante.
3. Demostrar el factor de optimalidad.

> Las demostraciones ya no serán (tan) complicadas (en general).

Traveling Salesman Problem

Enunciado

- $G(U, V)$ grafo no dirigido rotulado con pesos positivos dados por la función $w(u, v)$.
- Buscamos un ciclo C que visite todos los vértices de G al mínimo costo $W(C)$: la suma de los costos de todos los arcos en C.
- Consideremos $w(u, v) \leq w(u, x) + w(x, v)$.

> Para ir de un nodo al otro, el mejor camino siempre sera el directo.

Aproximación

- Sea C^* el ciclo óptimo para el TSP, al eliminar uno de sus arcos obtenemos un árbol recubridor (spanning tree).
- $W(C^*)$ es a lo menos $W(T)$, con T el árbol recubridor minimo.

$$W(T) \leq W(C^*)$$

- De que forma podemos utilizar esto para obtener un algoritmo aproximado que resuelva TSP?

- La idea será pararnos en un vertice u y realizar un viaje de **ida y vuelta** en T.
- Este viaje tiene un costo de $2 \cdot W(T)$.
- Buscamos un ciclo C en donde solo se repita el nodo inicial, nuestro viaje tiene nodos repetidos dos veces.
- Necesitamos $W(C) \leq 2 \cdot W(T)$.
- Así, tendriamos $W(C) \leq 2 \cdot W(T) \leq 2 \cdot W(C^*)$.

- Para eliminar los nodos de nuestro viaje **ida y vuelta** y asi formar C recordemos $w(u,v) \leq w(u,x) + w(x,v)$.
- Podemos partir en cualquier nodo u y "tratar de seguir" el viaje ida y vuelta. Cada vez que debamos devolvernos a un nodo x para ir a v desde u, lo haremos directamente (sin los nodos intermedios).

- Esta estrategia garantiza que encontramos un ciclo C tal que $W(C) \leq \mathbf{2} \cdot W(T)$ debido a nuestro supuesto. 🙏
- Como ya sabiamos $\mathbf{2} \cdot W(T) \leq W(C)$, obtenemos:

$$\frac{W(C)}{W(C^*)} \leq 2$$

- Nuestro algoritmo tiene un ratio de aproximación constante de 2. A lo más tendremos una solución que es el doble del óptimo. 🤔

Pseudocódigo

```
def approx_TSP(G):
    T = kruskal(G) # arbol recubridor minimo
    u = select_initial_node(T)
    C = pre_order(u,T) # analogo a nuestra tecnica
    return C
```

Polynomial time approximation schemes (PTAS)

- Con algoritmos aproximados obtenemos soluciones dentro de un factor de optimalidad.
- Nos gustaria poder controlar la precisión. Esto lo haremos mediante un input $\epsilon > 0$ que garantizará una solución con error relativo de a lo más ϵ.
- A medida que ϵ tiende a 0, tendremos un algoritmo con un mayor tiempo de ejecución. Por ejemplo un algoritmo con complejidad $O(2^{1/\epsilon}n^2)$.

- Un **fully polinomial time approximation scheme** (FPTAS) es un algoritmo cuyo tiempo de ejecución es polinomial para n y $1/\epsilon$. Por ejemplo un algoritmo con complejidad $O((n/\epsilon)^2)$
- Para tales casos aproximaciones razonablemente precisas son computacionalmente factibles. ❤️
- Lamentablemente muy pocos problemas NP completos admiten este tipo de algoritmos. 😢

Subset sum

- Sea S un conjunto de enteros positivos $\{x_1, \ldots, x_n\}$, y s un valor objetivo.
- Buscamos el subconjunto $S' \subseteq S$ tal que la suma de sus elementos sea exactamente o menor a s, pero no mayor.

Algoritmo determinista

- Un algoritmo no muy eficiente para este problema es el siguiente:

```
def subset_sum(l, suma):
    s = set([0])
    for i in range(0, len(l)):
        s = s | set(map(lambda x: x + l[i], s))
        s = set(filter(lambda x: x <= suma, s))
    return max(s)
```

- El tiempo de ejecución viene dado por $\Omega(2^n)$.

> Debido a que se analiza el conjunto potencia de un arreglo de n elementos.

- Para una ejecución con $l = [1, 4, 6]$ y suma 8, se obtiene:

$$
\begin{aligned}
s_0 &= [0] \\
s_1 &= [0] \cup [0+1] = [0, 1] \\
s_2 &= [0, 1] \cup [0+4, 1+4] = [0, 1, 4, 5] \\
s_3 &= [0, 1, 4, 5] \cup [0+6, 1+6, 4+6, 5+6] \\
&= [0, 1, 4, 5, 6, 7]
\end{aligned}
$$

- El optimo es 7.

> Qué pasa si agregamos valores a l muy similares a los ya presentes?

Approximation algorithm

- La idea será "podar" elementos que sean **relativamente** cercanos.
- El optimo para $[1, 1.01, 4, 6, 6.2]$ es de 7.21, con la versión "podada" (trimmed) $[1, 4, 6]$ es 7. 🤔
- Qué tanto podemos podar (disminuir el espacio de busqueda) manteniendo una solucion **suficientemente** buena? 🤔
- Podemos reducir el espacio de búsqueda de exponencial en n a lineal? 🤔

- El error relativo que estamos dispuestos a asumir es ϵ.
- Qué tanto podemos podar viene dado por $\delta = \frac{\epsilon}{n}$, notar que $0 < \delta < 1$
- Asumiendo L ordenado, iteraremos sobre la lista.
- Sea z el último elemento no podado en L e $y \geq z$ el siguiente elemento. Podaremos a y si:

$$\frac{y - z}{y} \leq \delta$$

- Es decir, en L no habrán elementos y y z tal que:

$$(1-\delta)y \leq z \leq y$$

- Al podar estamos forzando que no hayan elementos muy similares en L.
- Considere $L = [10, 11, 12, 15, 20, 21, 22, 23, 24, 29]$, al podar obtenemos $L = [10, 12, 15, 20, 23, 29]$ con $\delta = 0.1$.

- Agregando la función trim que se encarga de podar en cada iteración, obtenemos el siguiente algoritmo PTAS:

```
def app_subset_sum(l, suma, epsilon):
    s = set([0])
    delta = epsilon/len(l)
    for i in range(0, len(l)):
        s = s | set(map(lambda x: x + l[i], s))
        aux = list(s)
        aux.sort()
        s = set(trim(aux, delta))
        s = set(filter(lambda x: x <= suma, s))
    return max(s)
```

```python
def trim(l, delta):
    l_prime = [l[0]]
    last = l[0]
    for i in range(1, len(l)):
        if (last < (1 - delta) * l[i]):
            l_prime.append(l[i])
            last = l[i]
    return l_prime
```

Analisis alternativo

- La idea será ver la acción de podar (trimming) como dividir el intervalo $[1 \ldots s]$ en subintervalos de largo exponencialmente creciente.
- Sea $d = 1/(1-\delta)$ (notar $d > 1$) y consideramos los intervalos:

$$[1, d], [d, d^2], [d^2, d^3], \ldots, [d^{k-1}, d^k]$$

> donde $d^k \geq s$.

- Notemos que si y y $z \leq y$ están en el mismo subintervalo $[d^{n-1}, d^n]$ entonces:

$$\frac{y-z}{y} \leq \frac{d^n - d^{n-1}}{d^n} = 1 - \frac{1}{d} = \delta$$

- Es congruente con lo planteado anteriormente.
- Entonces, visto de esta manera al hacer trimming no tendremos más de un elemento en un mismo sub intervalo.

Demostración PTAS

- Debemos demostrar que el tiempo de ejecución es polinomial en n.
- Primero demostraremos que el tamano de la lista podada (trimmed) es $O((n \cdot log\ s)/\epsilon)$
- Cualquier par consecutivo de datos diferen al menos por un ratio $d = 1/(1-\delta)$. Pues:

$$\frac{y-z}{y} \leq \delta = 1 - \frac{1}{d}$$
$$1 - \frac{z}{y} \leq 1 - \frac{1}{d}$$
$$\frac{z}{y} \geq \frac{1}{d}$$

- El elemento no cero más pequeno difiere del máximo por un ratio de al menos d^{k-1}, con k el número de elementos en la lista podada.
- Como el elemento no cero más pequeno es al menos 1, y el máximo a lo más s tenemos que:

$$d^{k-1} \leq s/1$$

- Tomando logaritmo natural obtenemos:

$$(k-1)\ln d \leq \ln s$$

- Sabemos $\delta = \epsilon/n$, además utilizaremos la identidad $\ln(1+x) \leq x$.

- Así tenemos:

$$k - 1 \leq \frac{\ln s}{\ln d} = \frac{\ln s}{-\ln(1-\delta)} \leq \frac{\ln s}{\delta} = \frac{n \ln s}{\epsilon}$$

- Concluimos complejidad espacial $k = O(\frac{n \log s}{\epsilon})$.
- El tiempo de ejecucion sera $O(n \cdot k) = O(\frac{n^2 \ln s}{\epsilon})$

> n debido a las iteraciones y $\frac{n \ln s}{\epsilon}$ como cota superior al tamano de la lista a la cual aplicaremos la función trim.

Conclusiones

- Aproximamos un algoritmo determinista con complejidad temporal (y espacial) exponencial, con un un algoritmo con complejidad temporal (y espacial) lineal en n. Esto ha sido demostrado.
- Demostramos que la solución efectivamente cumple con el maximo error ϵ que asumimos? ... 😢, no. Esta demostracion es también necesaria.
- La demostración restante se puede encontrar en el siguiente documento, en el cual se basó fuertemente esta ayudantia: https://www.cs.umd.edu/class/fall2013/cmsc451/Lects/cmsc451-fall13-lects.pdf ❤️

- Contamos con un set de estrategias para disenar algoritmos:
    i. Dividir y conquistar.
    ii. Backtracking.
    iii. Greedy.
    iv. Programacion dinámica.
    v. Algoritmos Aleatorizados.
    vi. Algoritmos Aproximados
- El documento citado contiene las todas estrategias 😮, un verdadero tesoro.