Repaso I2

Rojo-negro, Hash, Grafos y Estrategias

Joaquín Peralta - Elías Ayaach - Alex Infanta - Amelia Gonzalez - Paula Grune

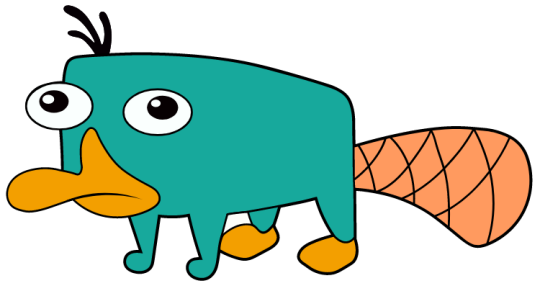
Hashing

Tablas de Hash 2019-2

La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores existentes en todos los multiversos, y necesita ayuda para hacer más eficiente la atención de la millonaria cantidad de pedidos que recibe por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando la pizza esta lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra el pedido del sistema.

Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?



Solución

Para eso usamos una tabla de hash que nos permita guardar múltiples *values* para un mismo *key*. En este caso *key* corresponde al tipo de pizza y un *value* corresponde al nombre de la persona que lo pidió.

Los *values* de un mismo *key* se deben guardar en una cola FIFO, de manera de agregar un nuevo *value* o extraer el siguiente, sea de O(1) y se atienda en orden de llegada.

La eliminación del pedido sale automática con esta estructura, ya que obtener el siguiente elemento lo extrae de la estructura.

Como el dominio de las *keys* es infinito, debemos ir despejando las celdas de la tabla cuando una *key* se queda sin *values*, ya que no tenemos memoria infinita. Para esto es necesario usar **encadenamiento** ya que permite eliminar *keys* de la tabla, sin perjudicar el rendimiento de esta. De esta manera, en caso de que varios sabores de pizza sean hasheados a la misma celda, las colas FIFO de los nombres de las personas que pidieron esas pizzas deberán ser encadenadas en una lista doblemente ligada que las contenga

Grafos

DFS y Orden Topológico

a) En clases estudiamos una versión recursiva del recorrido DFS de un grafo. Proponga el pseudocódigo de un algoritmo iterativo que implemente DFS en un grafo dirigido G = (V, E), que opere en tiempo O(V + E). Indique precisamente qué estructura(s) de datos necesita para su solución

Este es una modificación del algoritmo DFS estudiado en clases, se reemplaza el DFS-visit recursivo:

```
DFS-Visit(s):
    for u ∈ V − {s} :
        u.color ← blanco;  u.δ ← ∞;  π[u] ← ∅
    s.color ← gris;  s.δ ← 0;  π[s] ← ∅
    Q ← stack LIFO vacío
    Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        for v ∈ α[u] :
            if v.color = blanco :
                v.color ← gris;  v.δ ← u.δ + 1
                π[v] ← u
                Insert(Q, v)
        u.color ← negro
```

b) Un algoritmo alternativo para determinar un orden topológico en un grafo dirigido G = (V, E) es extraer un nodo sin aristas que apunten a él, eliminar todas sus aristas y repetir el proceso hasta que no queden nodos

(i) Proponga el pseudocódigo de este algoritmo de forma que en tiempo O(V + E) entregue un orden topológico para un grafo acíclico.
Especifique qué estructura(s) de datos necesita.

Solución.

```
Algoritmo(V, E):
    L ← lista ligada vacía
    while V ≠ ∅ :
        u ← Extract(Q)
        v ← nodo de V sin aristas incidentes Extraer v de V
        for e ∈ E :
            if v es mencionado en e :
                Extraer arista e de E
        Insertar v en L
    return L
```

(ii) Explique cómo este algoritmo puede detectar un ciclo en G. No requiere dar pseudocódigo

Si L no contiene todos los nodos de V, entonces G tiene un ciclo dirigido

Backtracking
vs
Greedy vs DP

Las 4 técnicas vistas

Para atacar los problemas complejos que se nos han presentado, contamos con las siguientes estrategias:

- Dividir para conquistar (D&C)
- Backtracking
- Algoritmos codiciosos (Greedy)
- Programación dinámica (DP)

Un breve repaso...

Dividir para conquistar

- Dividir el problema en problemas más pequeños hasta que sea trivial resolverlos
- Los problemas son **disjuntos**: Cada subproblema es independiente del resto y se resuelve aparte
- La solución del problema grande se construye a partir de las soluciones de los problemas pequeños

EJEMPLO: Ordenar un array con MergeSort

Backtracking

- Definimos variables, dominios y restricciones
- Asignamos valores a las variables de forma ordenada
- Si alguna restricción se rompe, deshacemos la última asignación y probamos nuevamente
- Iteramos hasta resolver el problema
- GRAN complejidad → $O(K^n)$

EJEMPLO: Salir de un laberinto dándose golpes contra la pared

Un breve repaso...

Greedy

- Iremos haciendo decisiones de forma secuencial
- Para cada paso, tomaremos la decisión que parezca mejor en el momento
- ¿Cuál es la mejor decisión? La que minimice/maximice cierta función
- Eficientes, pero podría llevarnos a mínimos/máximos locales

EJEMPLO: Queriendo encontrar el camino más corto en un grafo, tomar siempre la arista de menor peso

Programación Dinámica

- Al igual que Dividir para Conquistar, dividir el problema en subproblemas pequeños que tienen **subestructura óptima**
- Estos subproblemas NO son disjuntos, si no que se repiten al descomponer otros problemas
- Resolvemos los problemas sencillos y guardamos sus soluciones para usarlas al componer las soluciones de problemas mayores

EJEMPLO: Dar vuelto con la menor cantidad de monedas

¿Cuál debemos usar?

Dependiendo del problema, todos podrían encontrar solución, pero algunas soluciones podrían ser mejores que otras

Hay que hacerse preguntas tales como:

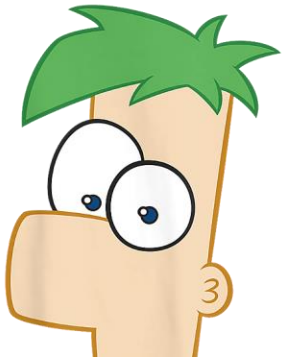
a) ¿Queremos maximizar minimizar algo?
b) ¿Hay restricciones?
c) ¿Hay problemas similares más pequeños? ¿Se repiten?

Queremos encontrar la mejor estrategia para el problema

Knapsack problem

El problema de la mochila consiste en seleccionar un subconjunto de objetos (indivisibles y unitarios), cada uno con un **valor v** y un **peso w,** de manera que se maximice el valor total de los objetos elegidos sin superar la capacidad **máxima W** de la mochila.

Cree una solución con **backtracking**, **PD** y **greedy** (Para este último no se pide que la solución sea globalmente óptima).



Knapsack problem - Backtracking

Variables y Restricciones

$$\sum_{i=1}^{n} x_i w_i \leq W$$

X_i : **1** si se considera el objeto i en la solución, **0** en otro caso.

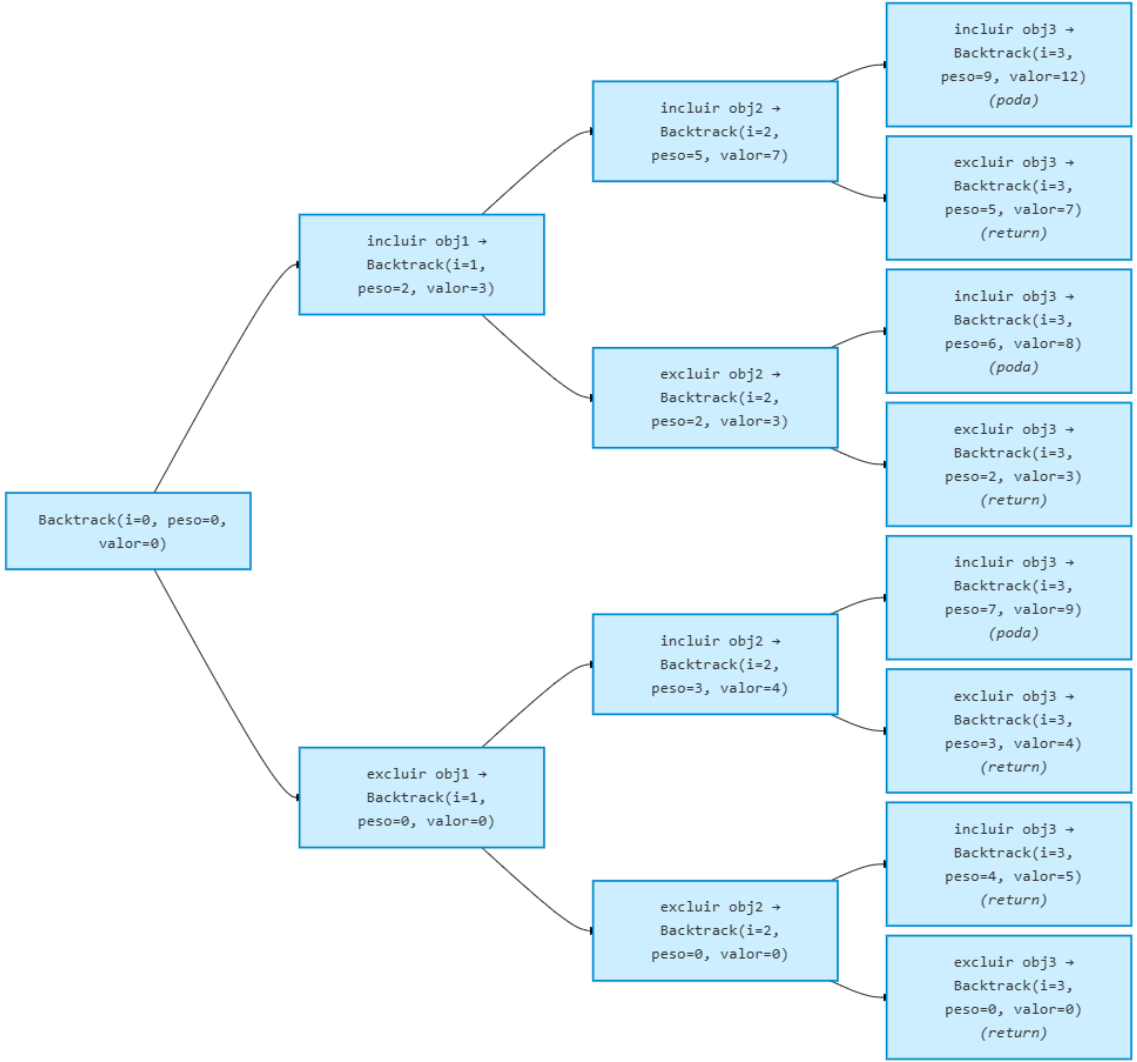
Knapsack problem - Backtracking

```
KnapsackBacktracking (w, v, W) :
1.      n ← len(w)
2.      max_value ← 0
3.      Backtrack (i, current_weight, current_value) :
4.          if current_weight > W :
5.              return
6.          if current_value > max_value :
7.              max_value ← current_value
8.          if i = n :
9.              return
10.         Backtrack (i + 1, current_weight + w[i], current_value + v[i])
11.         Backtrack (i + 1, current_weight, current_value)
12.     Backtrack (0, 0, 0)
13.     return max_value
```

O(2^n)

Ejemplo

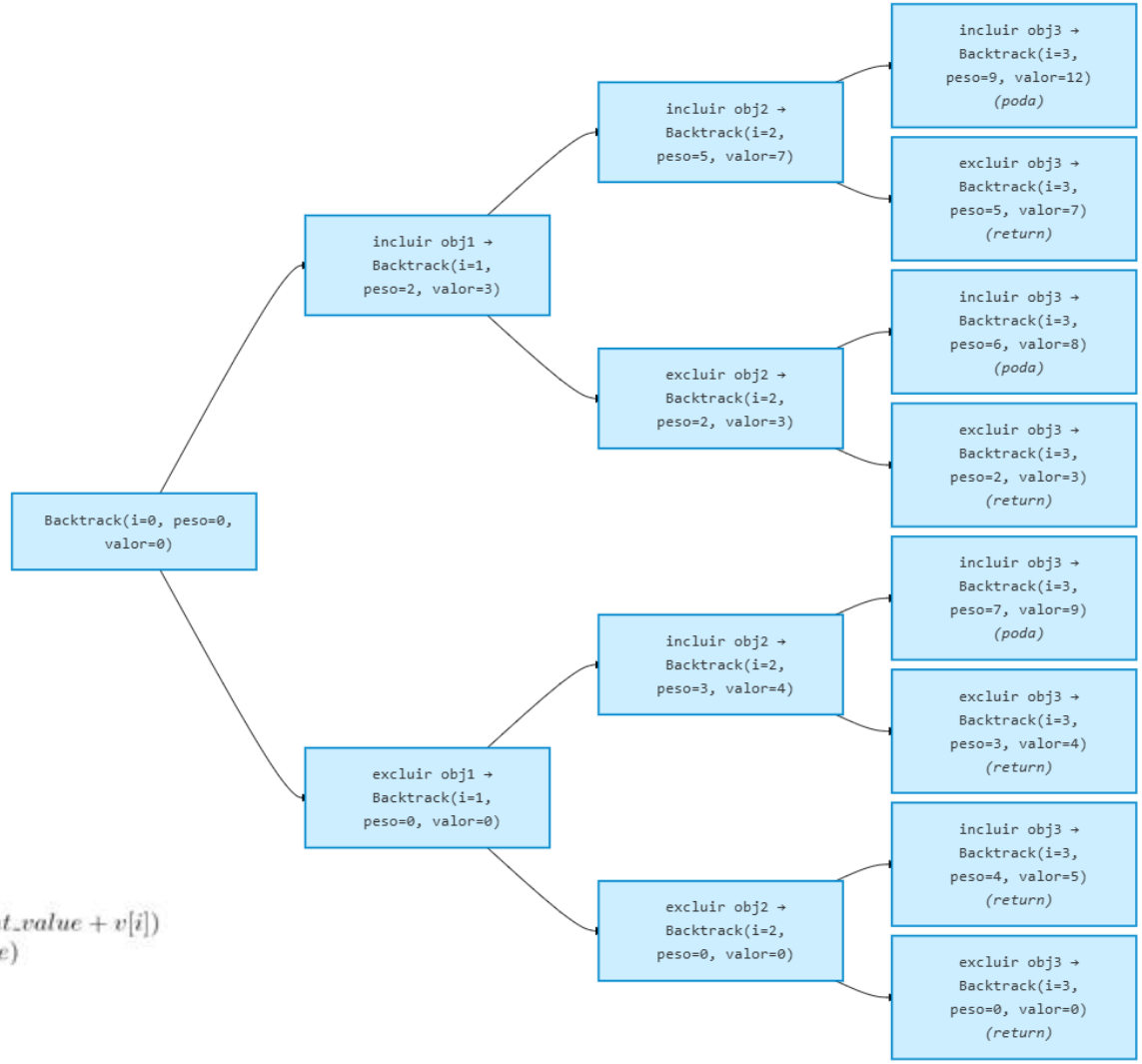
- Número de objetos: n=3
- Pesos: w=[2, 3, 4]
- Valores: v=[3, 4, 5]
- Capacidad mochila: W=5



Ejemplo

- Número de objetos: n=3
- Pesos: w=[2, 3, 4]
- Valores: v=[3, 4, 5]
- Capacidad mochila: W=5

```
KnapsackBacktracking (w, v, W) :
1.    n ← len(w)
2.    max_value ← 0
3.    Backtrack (i, current_weight, current_value) :
4.        if current_weight > W :
5.            return
6.        if current_value > max_value :
7.            max_value ← current_value
8.        if i = n :
9.            return
10.       Backtrack (i + 1, current_weight + w[i], current_value + v[i])
11.       Backtrack (i + 1, current_weight, current_value)
12.   Backtrack (0, 0, 0)
13.   return max_value
```



Knapsack problem - PD

$$DP[i][w] = \max\{DP[i-1][w], \text{valor}[i] + DP[i-1][w - \text{peso}[i]]\}$$

Valor máximo considerando los primeros i objetos y una capacidad w

Subproblema donde no se considera el objeto i

Subproblema donde sí se considera el objeto i, entonces hay menos capacidad

Knapsack problem - PD

```
KnapsackDP (w, v, W) :
1.   n ← len(w)
2.   Crear una matriz dp[0...n][0...W] ← 0
3.   for i = 1...n :
4.       for j = 0...W :
5.           if w[i − 1] ≤ j :
6.               dp[i][j] ← max(dp[i − 1][j], dp[i − 1][j − w[i − 1]] + v[i − 1])
7.           else :
8.               dp[i][j] ← dp[i − 1][j]
9.   return dp[n][W]
```

O(n x W)

Ejemplo

- Número de objetos: n=4
- Pesos: w=[1, 3, 4, 2]
- Valores: v=[1, 3, 5, 3]
- Capacidad mochila: W=7

```
KnapsackDP (w, v, W) :
1.  n ← len(w)
2.  Crear una matriz dp[0...n][0...W] ← 0
3.  for i = 1...n :
4.      for j = 0...W :
5.          if w[i − 1] ≤ j :
6.              dp[i][j] ← max(dp[i − 1][j], dp[i − 1][j − w[i − 1]] + v[i − 1])
7.          else :
8.              dp[i][j] ← dp[i − 1][j]
9.  return dp[n][W]
```

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0

Ejemplo

- Número de objetos: n=4
- Pesos: w=[1, 3, 4, 2]
- Valores: v=[1, 3, 5, 3]
- Capacidad mochila: W=7

```
KnapsackDP (w, v, W) :
1.  n ← len(w)
2.  Crear una matriz dp[0...n][0...W] ← 0
3.  for i = 1...n :
4.      for j = 0...W :
5.          if w[i − 1] ≤ j :
6.              dp[i][j] ← max(dp[i − 1][j], dp[i − 1][j − w[i − 1]] + v[i − 1])
7.          else :
8.              dp[i][j] ← dp[i − 1][j]
9.  return dp[n][W]
```

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	1	3	4	4	4	4
3	0	1	1	3	5	6	6	8
4	0	1	3	4	5	6	8	9

Ejemplo

- Número de objetos: n=4
- Pesos: w=[1, 3, 4, 2]
- Valores: v=[1, 3, 5, 3]
- Capacidad mochila: W=7

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	1	3	4	4	4	4
3	0	1	1	3	5	6	6	8
4	0	1	3	4	5	6	8	9

El resultado óptimo es:
dp[4][7] = 9

- Al elegir los objetos ({1,3,4})
- Peso total (1+4+2=7)
- Valor (1+5+3=9)

Knapsack problem - Greedy

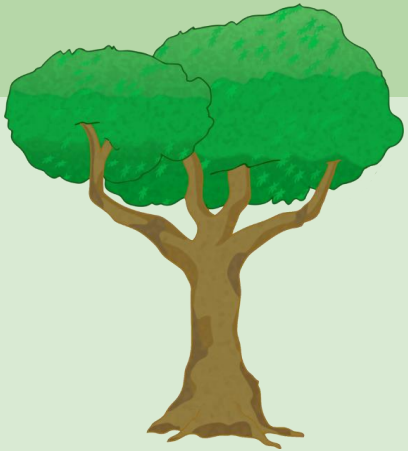
Probar insertar el elemento con el valor más denso (valor/peso) y repetir hasta no cumplir con la restricción de carga.

Knapsack problem - Greedy

```
KnapsackGreedy (v, w, W) :
1.   n ← len(w)
2.   Crear una lista density[0 ... n − 1] donde density[i] ← v[i] / w[i]
3.   Ordenar los objetos en orden descendente según density
4.   current_weight ← 0
5.   total_value ← 0
6.   for i = 0 ... n − 1 :
7.       if current_weight + w[i] ≤ W :
8.           current_weight ← current_weight + w[i]
9.           total_value ← total_value + v[i]
10.      else :
11.          break
12.  return total_value
```

O(nlogn)

Arboles Rojo-Negro



Un breve repaso...

Propiedades

- Cada nodo es rojo o negro
- La raíz del árbol es negra
- Si un nodo es rojo, entonces sus hijos deben ser negros
- La cantidad de nodos negros en el camino a cada hoja debe ser la misma

Extra

- Los nodos nulos/hojas vacías se consideran negros
- Las inserciones son con nodos rojos

Éxito en la I2 :3