Tarea 4 - Knapsack 0/1

Jonathan de Jesús Chávez Tabares A01636160
Carolina Pérez Alvarado A01631526 Agustín Quintanar de la Mora A01636142
Luis Delgado A01630534

24 de abril del 2020

1. Introducción

En esta tarea se realiza una comparación de la implementación de dos algoritmos para resolver el problema *Knapsack 0/1*, uno por medio de programación dinámica y el otro de manera greedy.

Definición formal del problema: Dado conjunto de n elementos numerados de 1 a n, cada elemento tiene un peso w_i, un beneficio b_i, se pueden elegir x_i de ellos y existe una mochila con capacidad W.

Maximizar:

$$\sum_{i=1}^{n} b_i x_i$$

Sujeto a las restricciones:

$$\sum_{i=1}^{n} w_i x_i \leq W,\ x_i \in \{0,1\}$$

2. Algoritmos

2.1. Programación dinámica

Algorithm 1: DP

```
dp = [n+1][W+1];
for i = 1 → i = n do
    for j = 1 → j = W do
        if currentweight ¿j then
            dp_{i,j} ← dp_{i-1,j};
        else
            dp_i j ←
              max(dp_{i-1,j}, dp_{i-1,j-currentweight}+
              currentvalue);
        end
    end
end
```

Lo que plantea la programación dinámica es una optimización de recursión, ya que se almacenan los resultados de los subproblemas, y de esta manera es más sencillo acceder a estos; esto puede reducir la complejidad de exponencial a polinomial. En este problema el resultado de un subproblema es almacenado en una matriz, lo que permite hacer la comparación con el actual subproblema y previos de manera más eficaz.

Complejidad temporal: Donde n=items y W=capacidad

$$\sum_{i=1}^{n}\sum_{j=1}^{W} 1$$

$$\sum_{i=1}^{n} W$$

$$= O(nW)$$

Complejidad espacial: Se utiliza una matriz de $n * W$ para almacenar la respuesta de los subproblemas.

$$= O(nW)$$

Aunque se puede hacer una optimización donde solo se usan dos filas, cuando se calcula una par, se guardan los resultados en la impar y viceversa. (Viene en el código implementado)

$$= O(W)$$

Sin embargo de esta manera sería imposible poder obtener el conjunto de los objetos que caben en la mochila.

2.2. Greedy

En este caso implementamos el algoritmo greedy para que elija el mejor item basado en su beneficio por unidad de peso (B / W) lo cual genera una solucionno optima como con programacion dinamica pero muy cercana a esta.

Algorithm 2: Greedy

Input items[], n (numItems), W (bagLimit)
for $i = 1 \rightarrow i = n$ **do**
  if w_i + $carriedWeight > W$ **then**
    break;
  else
    $maxprofit \leftarrow maxprofit + b_i$;
    $carriedWeight \leftarrow carriedWeight + w_i$
Output *maxprofit, usedItems (List)*

Complejidad temporal (Worst Case): Donde n=items

Division + IntroSort + ChecarSiCabe

$$\sum_{i=1}^{n} 1 + nlog(n) + \sum_{i=1}^{n} 1$$

$$n + nlog(n) + n$$

$$n + log(n))$$

$$= O(nlog(n))$$

Complejidad espacial: Se utilizan 2 vectores de tamaño n.

$$= O(2n)$$

$$= O(n)$$

3. Experimentación

3.1. Condiciones experimentales

- Lenguaje: C++ 14
- Compilador: clang 11.0.0
- Dispositivo: Macbook Pro 2014
- Procesador: 2.6 GHz Intel Core i5 dual core
- RAM: 8 GB 1600 MHz DDR3

Banderas de compilación:

- -O2: Enciende la optimización. Es la opción default y la más recomendada.
- -std=c++14: Especifica la versión de C++ o versión estándar de ISO.
- -o: Sirve para personalizar el nombre del archivo generado por el compilador.

3.2. Pruebas

Las pruebas se realizaron solamente imprimendo a stdout el beneficio máximo ya que si se imprimen los otros datos, estos causarían el mayor tiempo de ejecución (95 % se debe al I/O).

3.2.1. Pruebas Greedy

# Prueba	Objetos	Capacidad	Beneficio	Tiempo de ejecución
ga04	10000	62892889	63630728	530 μs
ga05	1000000	940162831	1026778680	10941 μs
ga06	3000000	3684401028	3150068305	24023 μs

3.2.2. Pruebas programación dinámica

# Prueba	Objetos	Capacidad	Beneficio	Tiempo de ejecución
dp01	3	5	6	.00001s
dp04	24	1404100	3028529	0.18s
dp05	10000	10000	378533	5.55s

3.3. Diferencia de optimalidad

En algunas instancias el greedy no cumple con la optimalidad de Bellman, este es un ejemplo:

Capacidad = 50

Peso	Beneficio
1	10
20	30
40	50

Resultado greedy

- Peso: 40
- Elementos seleccionados son 1 y 2

Resultado programación dinámica

- Peso: 80
- Elementos seleccionados son 2 y 3

4. Conclusiones

Observamos que el orden de crecimiento de algoritmos greedy es menor pero no siempre llega a la solución óptima, en cambio el de programación dinámica sí, a cambio de tener mayor complejidad temporal.

Implementamos una reducción de complejidad espacial en programación dinámica que utiliza únicamente dos arreglos en lugar de la matriz completa, pasando así en complejidad espacial de $O(nW)$ a $O(W)$.

5. Referencias

University of Washington. (s.f.). Compiling with g++. Recuperado de https://courses.cs.washington.edu/courses/cse373/99au/unix/g++.html

Weimer, F. (2018). Recommended compiler and linker flags for GCC. RedHat. Recuperado de https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/