Algoritmos y Estructuras de Datos III

Segundo cuatrimestre 2018

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo práctico Nº1

Integrante	LU	Correo electrónico
Christian Rivera	184/15	christiannahuelrivera@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción **3**

2. Soluciones provistas **3**
2.1. Fuerza bruta 3
2.2. Backtracking 4
2.3. Programación dinámica 5

3. Experimentación **6**
3.1. Experimento 0 6
3.2. Experimento 1 6
3.3. Experimento 2 7
3.4. Experimento 3 8
3.5. Otros experimentos que no entraron 8

4. Experiencia con este trabajo **8**

1. Introducción

El problema que analizaremos en este trabajo es el llamado Suma de subconjunto o mejor conocido por su nombre en ingles, Subset-Sum. Este problema tiene muchas aplicaciones en el mundo de la computación y en su definición general no se conocen algoritmos polinomiales que lo resuelvan de manera óptima. La definición de la Suma de subconjuntos es, dado un conjunto N de números Reales y un Valor objetivo V se quiere saber la mínima cantidad de elementos de N que sumados den V, si es que esta existe. Este problema tiene muchas aplicaciones muy interesantes entre las que se encuentran modelado de problemas en física donde se buscan cantidades de partículas óptimas para el modelado de un problema, elección de tareas para nivelar la carga de procesos en un procesador y la que particularmente más me llamó la atención a mí, sus aplicaciones en criptografía.

2. Soluciones provistas

El problema que encaré es la versión de Suma de subconjuntos donde N son un conjunto de numeros enteros no negativos y V es un entero no negativo. En este trabajo se proveen tres algoritmos para obtener la solución exacta para instancias que cumplen las precondiciones antes mencionadas.

2.1. Fuerza bruta

Primero encararemos el problema de la forma más simple e intuitiva que se me ocurrio. Si enumeramos todas las posibles permutaciones de los elementos de N y nos quedamos con la mejor de todas la posibles combinaciones que resuelve el problema entonces abremos encontrado la solución al problema de la suma de subconjuntos. A continuación se proveé el algoritmo que implementa la idea recién descripta.

```
1: function sumaDeSubconjuntos(arreglo N, entero V, arreglo respuestaParcial, entero indice, entero mejorSolucion)
2:     if indice = tamaño(N) then
3:         if Sumatoria(respuestaParcial) = V ∧ tamaño(respuestParcial) < mejorSolucion then
4:             mejorSolucion ← tamaño(respuestaParcial)
5:         else
6:             sumaDeSubconjuntos(N,V,respuestaParcial,indice+1,mejorSolucion)
7:             sumaDeSubconjuntos(N,V,respuestaParcial ⋃ N[indice],indice+1,mejorSolucion)
8:         end if
9:
```

En el algoritmo propuesto la respuestaParcial son los elementos que se fueron eligiendo hasta este momento en el arbol de recursión y el indice es un entero que indica en que lugar del arreglo N se debe mirar en la actual iteración. Indice se mueve entre [0,n) con lo que esta es la altura del arbol de recursión.

La complejidad de este algorimo es del orden de

$$O(2^n \cdot n)$$

Siendo n = tamaño(N). Esto es porque el arbol de recursión tiene altura n y en cada nivel crea 2 hojas más que a su vez crean dos hojas más. Finalmente el producto por n se obtiene porque cuando se llega a las hojas estas, se analiza si estas son o no soluciones haciendo la sumatoria de todos sus elementos y esto es del orden de O(n).

2.2. Backtracking

Luego de obtener todas las posibles soluciones con fuerza bruta si analizamos el algoritmo podemos observar que estamos analizando caminos que no dan soluciones posibles o que no mejoran soluciones que ya tenemos. En un segundo enfoque realizaremos una serie de podas sobre las ramas del arbol que no cumplan con una serie de reglas.

Primero cortaremos ramas que no den soluciones factibles, osea ramas que analizandolas no es posible que nos den una solución válida para el algoritmo y por lo tanto analizarlas es una perdida de tiempo y de poder computacional.

```
1: function FACTIBILIDAD(arreglo N, entero V, arreglo respuestaParcial, entero indice, entero mejorSolucion)
2:     cortarRama ← Falso
3:     if sumatoria(respuestaParcial) > V then
4:         cortarRama ← Verdad
5:     else if sumatoria(respuestaParcial) + sumatoriaDeLosProximos(indice,N) < V then
6:         cortarRama ← Verdad
7:     end ifreturn cortarRama
8: end function=0
```

Con esta primera poda cortamos si la suma de los elementos hasta el momento actual es mayor que el valor objetivo o si la suma de los elementos hasta ahora más la suma de los proximos elementos en N son menores que V. De ambos modos debo cortar porque es evidente que por más que siga analizando todas las permutaciónes posteriores no puedo alcanzar mi objetivo.

Como segunda idea puedo pensar en una poda donde si ya sé que mi respuestaParcial es peor que la mejor que obtuve en un momento dado entonces corto la ejecución, ya que por más que continue analizando las proximas combinaciones no podré mejorar el optimo aunque si lo iguale.

```
1: function OPTIMALIDAD(arreglo N, entero V, arreglo respuestaParcial, entero indice, entero mejorSolucion)
2:     cortarRama ← Falso
3:     if tamaño(respuestParcial) >= mejorSolucion then
4:         cortarRama ← Verdad
5:     end ifreturn cortarRama
6: end function
```

La complejidad de estas podas depende mucho de como las implemente por lo tanto en el pseudocódigo expresé la idea subyacente de las mismas y ahora voy a hablar de implementaciones. Para la sumatoriaDeLos-Proximos precalculé un arreglo con la suma desde un indice a los i-esimos proximos con complejidad

$$O(n^2)$$

De este modo no tengo que pagar O(n) en cada nivel del arbol. Luego la sumatoria de la respuestaParcial la tengo almacenada en una variable, por lo que respuestaParcial no es más un arreglo y puedo saber su suma de manera instantánea. Con estas consideraciones, las optimizaciones cuestan O(1) ya que son meras comparaciones de valores enteros. Finalmente la complejidad de el nuevo algoritmo es

$$O(2^n \cdot n + n^2) = O(2^n \cdot n)$$

2.3. Programación dinámica

En una tercera instancia refomulé completamente el algoritmo utilizado hasta ahora para implementar un solución que no vuelva a recalcular soluciones que ya calculó previamente. La estrategia utilizada aquí fue la propuesta por programación dinámica donde se debe pensar una función recursiva que sea óptima porque utilice resultados que ya hayan sido calculados por sub intancias de la misma.

Mi función recursiva la llamaré F y esta será:

F(i,N,v) = "Mínima cantidad de elementos de N entre [0..i) que suman v ".

Luego defino a F como:

F(i,N,v) = min(F(i-1,N,v) , F(i-1,N,v - N[i]) + 1)

Esta función F es recursiva porque utiliza el resultado de si misma para una instancia menor y es óptima porque construye una solución a partir de soluciones que a su vez ya son optimas porque F(i-1,N,v) es la mejor solución para el problema de sumar v con los elementos entre [0,i-1) y F(i-1,N,v - N[i]) es la mejor solución de sumar v - N[i] con los elementos entre [0,i-1). Finalmente el mínimo entre los dos será el mínimo para el problema de sumar v con i elementos.

De modo que la respuesta al problema es F(tamaño(N),N,V).

Para la implementación de esta idea tengo el siguiente algoritmo que guarda los resultados en una matriz de n * V llamada R:

```
1:  function sumaDeConjuntos(entero i, arreglo N, entero v, matriz R)
2:      if  R[i,v] = −∞  then
3:          if i > 0 ∧ v > 0  then
4:              R[i,v] ← min(sumaDeConjuntos(i-1,N,v,R) , sumaDeConjuntos(i-1,N,v - N[i], R) + 1)
5:          else if v = 0  then
6:              R[i,v] ← 0
7:          else
8:              R[i,v] ← ∞
9:          end if
10:     end ifreturn R[i,v]
11: end function
```

La complejidad de este algoritmo es

$$O(n \cdot V)$$

Donde n = tamaño(N). Esto lo obtengo ya que en cada iteración llamo a la subinstancia de i-1 elementos con i entre [0,n) y para cada uno analizo los posibles v que quedan de por cada camino haber o no elegido el i-esimo elemento y restarlo al valor objetivo. Como en el peor caso para encontrar una suma que de 0 tengo que recorrer los n elementos de N obtengo la complejidad ya mencionada.

3. Experimentación

En esta sección voy a jugar con las implementaciones dadas a los algoritmos previamente expuestos en este trabajo para analizar su comportamiento en distintos contextos de uso que se le pueden dar a los mismos. Para conseguir este objetivo e pensado una serie de casos de prueba donde en cada uno de ellos construyo una entrada específica para los casos de prueba y posteriormente corro los algoritmos y les calculo los tiempos de ejecución.

Cabe aclarara que los mismos fueron calculados en una:

MacbookPro modelo 2012

Memoria de 16 GB a una frecuencia de 1333 MHz DDR3

Disco de estado sólido de 128 GB.

Este equipo no es óptimo para calcular tiempos de ejecución ya que es un portatil y en estos gracias a las políticas de ahorro de bateria del procesador, este último se puede apagar por intervalos pequeños para consumir menor batería. De todos módos fué utilizada solo para esta taréa siendo cargada y con la mínima cantidad de subprocesos corriendo en segundo plano.

Para cada uno de los experimentos, con el objetivo de tenér el menor ruido en mis datos, corrí cada instancia 100 veces y de estas tomé la mediana de estos datos para tener no tener outliers que me perjudiquen los gráficos sin razón. Aún así no se puede evitar el ruido en las muestras ya que hay muchas variables que pueden impactarlos.

Los datos fueron obtenidos con un steady clock de la libreria chrono de c++ con la cual se obtuvieron los ticks del procesador que luego los traduje a millisegundos para tener una medida que sea humanamente entendible.

3.1. Experimento 0

En este experimento corro los tres algoritmos con unos datos que no deberían perjudicar a ninguno en especial. De este modo podría verse su comportamiento en un contexto real.

Para eso elegí un V arbitrario para todas las instancias y fuí aumentando el n entre [1,20]. Los datos obtenidos se pueden ver a continuación en la imagen.

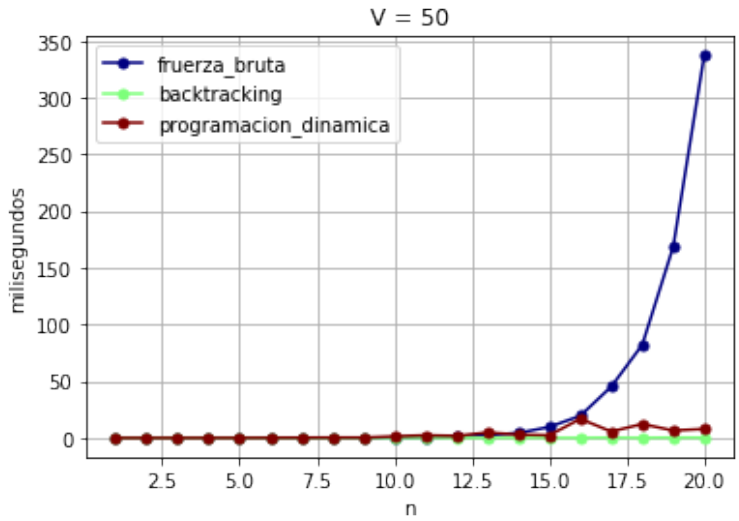

Figura 1: Comparando los tres algoritmos planteados

En el gráfico se puede ver como , si bien fuerza bruta y backtracking tienen la misma complejidad asintótica, en la práctica las podas del algoritmo cortan muchisimas ramas que sinó se deberían explorar. Del mismo modo se puede observar que Programación dinámica tiene un comportamiento mucho mejor que la primer implementación que resolvía el problema se la suma de subconjuntos.

3.2. Experimento 1

Como vimos en el experimento 0, fuerza bruta tiene un comportamiento exponencial tanto en su complejidad asitótica y en la práctica. Por eso en este y los siguientes experimentos no lo tendremos en cuenta.

Para las instancias de este experimento quise perjudicar a PD poniendo un valor objetivo V muy grande, de modo que la matriz que tiene que construir este algoritmo esa inmensa y tenga que pedir mucha memoria para alocarla.

Como primeras versiones elegí un V igual a INT MAX (en c++ el máximo valor representable con una variable de tipo int) pero esto fué demasiado para el algoritmo por que le costaba mucho crear las matrices por lo que tuve que bajarlo a un V igual a 100^2 que fué lo máximo que me dejó el experimento para correr en tiempos razonables.

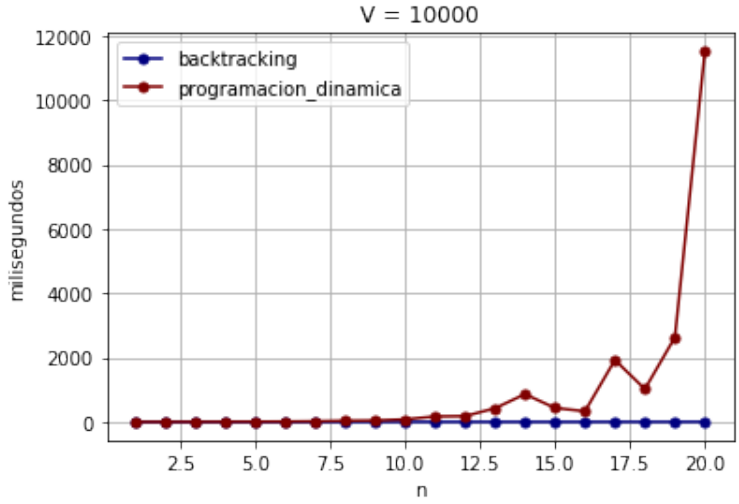

Figura 2: Compara BT y PD con instancias que perjudiquen a PD.

En la imagen se puede ver que estas instancias realmente afectaron la performance de la implementación de PD. De este modo la hipótesis de que mis instancias lo afectarian se cumplío satisfactoriamente.

3.3. Experimento 2

En este experimento cree instancias que beneficien a BT. Las mismas las hice con elementos cuyo valor esta entre [0,V*2] y luego las ordené de menor a mayor de modo que BT puede hacer podas mucho más efectivas (poda de factibilidad) ya que la suma de la respuesta parcial, más los i-esimos proximos elementos sea muy grande y pode las ramas que no correspondan.

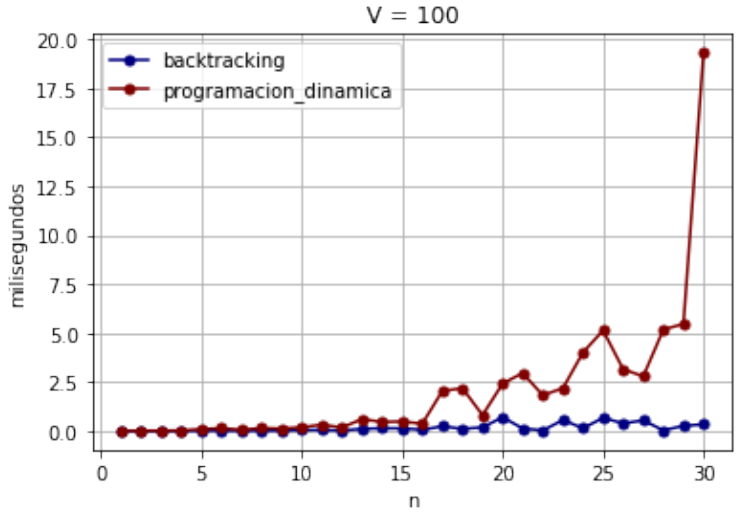

Figura 3: Compara BT y PD con instancias que benefician a BT.

Como se puede observar en la imagen BT le volvió a ganar a PD ya que las podas cortaron muchas ramas innecesarias. Los saltos que se observan en el gráfico son culpa del ruido de la muestra y es de esperar que si se

corría en lugar de 100 veces 1000 o 10000 veces el gráfico fuese lo más similar a una función continua.

3.4. Experimento 3

Para este experimento obtuve la muestra del mismo modo que en el experimento anterior pero luego ordené los datos de mayor a menor. De este modo mí hipótesis es que BT va a tener que entrar mucho más profundo en el arbol de recursión y saldrá perfudicado ante PD.

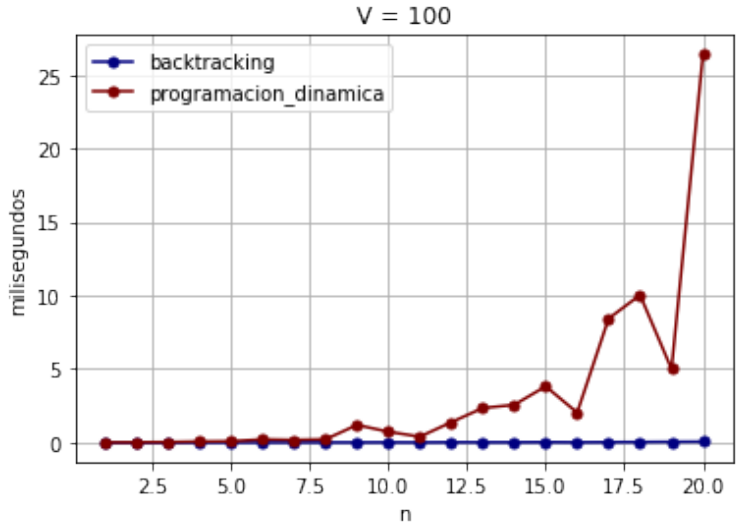

Figura 4: Compara BT y PD con instancias que perjudican a BT.

Observando la los datos de la imagen se puede observar que claramente la hipotisis planteada fue erronea ya que en la práctica se ve que PD corrio significativamente peor que BT. Analizando los mismos llegué a la conclusión que es porque no tuve en cuenta que una de las podas de factibilidad corta las ramas cuando el valor actual es mayor que el valor objetivo, por lo qué esta poda estaba cortando mucho antes que lo que hacía PD. Así pude ver por mis propios medio la importancia del método científico, donde se plantea una hipótesis, se experimenta y posteriormente se interpreta. De este modo se puede refutar o no la hipotesis inicial.

3.5. Otros experimentos que no entraron

Adicionalmente a estos experimentos, pensé y llevé a cabo otros más que cuando análicé sus resultados no me parecieron tan relevantes y no quise segúir expendiendo el informe por este motivo.

Las ideas de estos fueron, por un lado, llenar el arreglo N con valores superiores a V y luego en el último lugar de N ubicar a V de modo que se tuviese que recorrer todo el arbol en BT pero PD lo tuviese más fácil. de todos modos PD analiza todas las convinaciones y corta cuando completa los valores relevantes en la matriz. Otro fue llenar el arreglo de la misma forma pero que en el primer lugar ubico a V de modo que BT encuentre una solucion de tamaño 1 con muy poco trabajo.

4. Experiencia con este trabajo

Haciendo este trabajo, ya sea el análisis del problema, pensar los algoritmos, implementar estos en el lenguaje c++ o experimentando y graficandolos con jupyter, aprendí mucho de cada una de estas etapas y traté de hacer cada una con la mayor calidad posible.

Cuando pensaba los algoritmos primero los escribí en papel y luego fuí discutiendo si estas ideas respondian el problema. Luego tenía que analizar la complejidad asintótica de los mismos y en esta instancia encontraba errores en la primera implementación ya que en el caso de BT como primer versión del mismo me guardaba un vector de los elementos que fui eligiendo, la solución parcial, pero cuando quería podar tenía que hacer la sumatoria de todos ellos y luego la sumatoria de todos los proximos, por lo que la complejidad del algoritmo se me veía aumentada en

$$O(2^n \cdot n^3)$$

Esta instancia me resultó muy útil ya la ví facilmente extraible a mi trabajo en la industria donde hay dias que pienso soluciones donde hacer un ciclo de más cuando no es necesario implica muchos llamados distintos a

una base de datos. De este modo tomé consiencia de que debo siempre tener una actitud análitica y tratar de mejorar siempre las ideas iniciales.

Luego cuando hice la implementación de los algoritmos en código fuente en c++ me tuve que interiorizar con herramientas que no uso a diario ya que en el trabajo utilizo Java y para testear JUnit. Para hacer pruebas unitacias en c++ tuve que aprender a manerjar GTest que es el framwork de Google para hacer pruebas automatizadas. Esto me resultó muy util ya que sin este último no podría haber desarrollado siguiendo la metodología que a mí más me gusta, que es Test Driven Development o TDD. Me pareció una experiencia muy enrriquecedora aprender nuevas tecnologias y agregarlas a mis conocimientos.

Cuando tuve que realizar experimentos sobre las instancias que se me ocurrioron, para graficar los datos tuve que aprender a manejar pandas y jupyter. Como futuro trabajo voy a aprender a manejar mejor estas dos herramientas porque me parece que tienen una utilidad enorme.