Problema de la suma de subconjuntos (Subset-Sum)

Definición

Problema relativo al conjunto de los números enteros (N), En este problema tenemos una colección de números. $S = \{x_1, x_2, \dots, x_k\}$ y un número objetivo t Queremos determinar si el conjunto S contiene un subconjunto que suma t.

$$SUBSETSUM =$$

$$\left\{\langle S, t\rangle \mid S = \{x_1, x_2, \dots, x_k\}\ y\ para\ algún\ \{y_1, y_2, \dots, y_m\} \subseteq S, tal\ que \sum y_i = t \right\}$$

Por ejemplo $\langle\{4, 11,16,21,27\}, 25\rangle \in SUBSETSUM$, porque 4 + 21 = 25.

Tenga en cuenta que $\{x_1, x_2, \dots, x_k\}\ y\ \{y_1, y_2, \dots, y_m\}$ se consideran multisets así que se permiten la repetición de elementos (Sipser, 2006).

Como con cualquier problema aritmético, es importante recordar que la codificación estándar supone que los enteros de entrada están codificados en binario. Con esta suposición en mente, podemos demostrar que es improbable que el problema de la suma de subconjuntos tenga un algoritmo rápido (Cormen, Leiserson, Rivest, & Stein, 2009).

Características

- SUBSET-SUM está en NP
- SUBSET-SUM está en NP – Completo

Teorema.

$$SUBSETSUM\ está\ en\ NP$$

Prueba, Referencia Pág. 269 de (Sipser, 2006)**.**

Lo siguiente es un verificador V para SUBSETSUM.

V = "Para la entrada $\langle\langle S, t\rangle, c\rangle$:

1. Probar si c es un conjunto de números que suman t

2. Probar si S contiene todos los números en c.

3. Si ambos pasan, *aceptar*; caso contrario, *rechazar*. "

Teorema.

$$SUBSETSUM\ está\ en\ NP\ Completo$$

Prueba #1, Referencia Pág. 292 de (Sipser, 2006)**.** La idea general es que se puede reducir a 3SAT, por lo tanto tiene complejidad polinomial.

Prueba #2, Referencia Pág. 1097 de (Cormen, Leiserson, Rivest, & Stein, 2009).

El enfoque utilizado para probar que es NP Completo es de reducción basado en el siguiente gráfico.

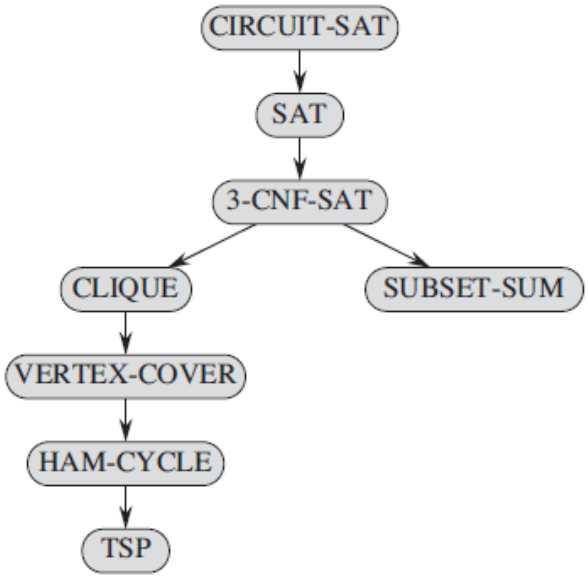

Figura 1: Estructura de pruebas, la prueba está basada en reducir el problema a CIRCUIT-SAT que es NP Completo, Fuente: Pág. 1087 de **(Cormen, Leiserson, Rivest, & Stein, 2009)**

El problema de optimización asociado con este problema de decisión tiene aplicaciones prácticas, por ejemplo deseamos encontrar un subconjunto de $\{x_1, x_2, \dots, x_n\}$ tal que la suma sea lo más grande posible pero no mayor que t.

Podemos tener un camión que no puede llevar más de t kg. y n diferentes cajas para trasladar, la i ava. caja pesa x_i kg. Queremos llenar el camión tanto como sea posible sin exceder el límite de carga del camión.

Describimos un algoritmo de tiempo exponencial que calcula el óptimo valor para este problema de optimización y luego mostramos la versión que usa un esquema de aproximación con complejidad polinomial, las siguientes páginas fueron traducidas de (Cormen, Leiserson, Rivest, & Stein, 2009)

Algoritmo exacto de tiempo exponencial para SUBSET-SUM.

Supongamos que hemos calculado, para cada subconjunto S' de S, la suma de los elementos en S' y luego seleccionamos, entre los subconjuntos cuya suma no exceda t, entre estos subconjuntos seleccionados alguno tiene la suma más cercana a t.

Este algoritmo devolvería la solución, pero podría tomar un **tiempo exponencial**.

Para implementar este algoritmo, utilizaremos un procedimiento iterativo, que en cada iteración i, calcule las sumas de todos los subconjuntos de $\{x_1, x_2, \dots, x_n\}$ utilizando como punto de partida las sumas de todos los subconjuntos $\{x_1, x_2, \dots, x_{i-1}\}$. Al hacerlo, nos daríamos cuenta de que una vez que un subconjunto particular S' tenga una suma superior a t, no habría razón para mantenerlo, ya que ningún superconjunto de S' podría ser la solución óptima, implementación de esta estrategia.

El procedimiento EXACT-SUBSET-SUM toma un conjunto de entrada $\{x_1, x_2, \dots, x_n\}$ y un valor objetivo *t.*

Este procedimiento calcula iterativamente L_i, la lista de sumas de todos los subconjuntos de $\{x_1, x_2, \dots, x_i\}$ que no excedan *t*, y luego devuelve el valor máximo en L_n.

Si L es una lista de enteros positivos y x es otro entero positivo, entonces $L + x$ denota la lista de enteros formados por L incrementado en *x, p*or ejemplo, si $L = \langle 1,2,3,5,9 \rangle$ entonces $L + 2 = \langle 3,4,5,7,11 \rangle$. También usamos esta notación para conjuntos, de modo que:

$$S + x = \{s + x : s \in S\}$$

También usamos un procedimiento auxiliar MERGE-LIST(L, L') que devuelve la lista ordenada que es la fusión de sus dos listas de entrada ordenadas L y L' con valores duplicados eliminados.

MERGE-LISTS se ejecuta en un tiempo $O(|L| + |L'|)$

```
EXACT-SUBSET-SUM(S, t)
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4       L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5       remove from L_i every element that is greater than t
6  return the largest element in L_n
```

Para ver cómo funciona EXACT-SUBSET-SUM, se denomina P_i al conjunto de todos los valores obtenidos de seleccionar un subconjunto (posiblemente vacío) de $\{x_1, x_2, \dots, x_i\}$ y sumando sus miembros. Por ejemplo, si s $S = \{1, 4, 5\}$ entonces:

$P_1 = \{0,1\}$

$P_2 = \{0,1,4,5\}$

$P_3 = \{0,1,4,5,6,9,10\}$

Dada la identidad

$P_i = P_{i-1} \cup (P_{i-1} + x_i)$

Podemos demostrar por inducción en *i* que la lista L_i es una lista ordenada que contiene cada elemento de P_i cuyo valor no es mayor que *t*. Dado que la longitud de $L_i$puede ser como máximo 2^i, EXACT-SUBSET-SUM es un algoritmo de tiempo exponencial en general, aunque es un algoritmo polinomial en los casos especiales en los que t es polinomial en $|S|$ o todos los números en *S* están limitados por un polinomio en $|S|$.

Algoritmo de aproximación para SUBSET-SUM.

Podemos derivar un algoritmo completamente polinomial "recortando" cada lista L_i después de ser creada. La idea detrás del recorte es que si dos valores en L están cerca unos de otros, podemos obtener una solución aproximada, no necesitamos mantener ambos de forma explícita.

Más precisamente, usaremos un parámetro de recorte δ tal que $0 < \delta < 1$. Cuando recortamos una lista L en δ eliminamos tantos elementos de L cómo es posible, de tal manera que si L' es el resultado de recortar L, entonces para cada elemento y que fue removido de L, hay un elemento z que se aproxima a y que se mantiene en L' esto es:

$$\frac{y}{1+\delta} \leq z \leq y$$

Podemos pensar que z está representando a y en la nueva lista L'. Cada elemento eliminado y está representado por un elemento restante z que satisface la desigualdad anterior.

Por ejemplo, si $\delta = 0.1$ y $L = \langle 10,11,12,15,20,21,22,23,24,29 \rangle$ y luego recortamos L obtenemos.

$L' = \langle 10, 12, 15, 20, 23, 29 \rangle$

El elemento eliminado 11 está representado por 10, los elementos 21 y 22 están representados por 20 y el elemento 24 está representado por 23.

Debido a que cada elemento de la versión recortada de la lista es también un elemento de la lista original, el recorte puede reducir drásticamente el número de elementos manteniendo un valor representativo cercano (y ligeramente menor) en la lista para cada elemento eliminado.

El siguiente procedimiento recorta la lista $L = \langle y_1, y_2, \ldots, y_m \rangle$ en un tiempo $O(m)$, dado L y δ, y suponiendo que L esta ordenado orden monotónicamente creciente. La salida del procedimiento es una lista recortada y ordenada.

```
TRIM(L, δ)
1  let m be the length of L
2  L' = ⟨y_1⟩
3  last = y_1
4  for i = 2 to m
5      if y_i > last · (1 + δ)        // y_i ≥ last because L is sorted
6          append y_i onto the end of L'
7          last = y_i
8  return L'
```

El procedimiento lee los elementos de L en un orden monotónicamente creciente. Un número se añade a la lista devuelta L' sólo si es el primer elemento de L o si este no puede ser representado por el número más reciente colocado en L'.

Dado el procedimiento TRIM, podemos construir nuestro esquema de aproximación como sigue. Este procedimiento toma como entrada un conjunto $S = \{x_1, x_2, \ldots, x_n\}$ de n enteros (en un orden arbitrario), un entero objetivo t, y un "parámetro de aproximación" ε, donde $0 < \varepsilon < 1$.

Esto devuelve un valor *z* cuyo valor está dentro de $1 + \varepsilon$ factor de la solución óptima.

```
APPROX-SUBSET-SUM(S, t, ε)
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4       L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5       L_i = TRIM(L_i, ε/2n)
6       remove from L_i every element that is greater than t
7  let z* be the largest value in L_n
8  return z*
```

La línea 2 inicializa la lista L_0 como la lista que contiene sólo el elemento 0. El bucle **for** en las líneas 3-6 calcula L_i como una lista ordenada que contiene una versión convenientemente recortada del conjunto P_i con todos los elementos mayores que *t* eliminados. Como creamos L_i a partir deL_{i-1}, debemos asegurarnos de que el recorte repetido no introduce demasiada inexactitud.

Observamos que APPROX-SUBSET-SUM devuelve una aproximación correcta si esta aproximación existe.

Por ejemplo, supongamos que tenemos el conjunto.

$S = \langle 104{,}102{,}201{,}101 \rangle$ con $t = 308$ y $\varepsilon = 0.40$. El valor de recorte $\delta = \frac{\varepsilon}{8} = 0.05$ APPROX-SUBSET-SUM calcula como sigue los valores en las líneas indicadas:

Línea 2: $L_0 = \langle 0 \rangle$

Línea 4: $L_1 = \langle 0, 104 \rangle$
Línea 5: $L_1 = \langle 0, 104 \rangle$
Línea 6: $L_1 = \langle 0, 104 \rangle$

Línea 4: $L_2 = \langle 0, 102, 104, 206 \rangle$
Línea 5: $L_2 = \langle 0, 102, 206 \rangle$
Línea 6: $L_2 = \langle 0, 102, 206 \rangle$

Línea 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
Línea 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$
Línea 6: $L_3 = \langle 0, 102, 201, 303 \rangle$

Línea 4: $L_4 = \langle 0, 101, 102, \ 201, 203, 302, 303, 404 \rangle$
Línea 5: $L_4 = \langle 0, 101, \ 201, 302, 404 \rangle$
Línea 6: $L_4 = \langle 0, 101, \ 201, 302 \rangle$

Calculo de aproximación para SUBSET-SUM.

El algoritmo de aproximación devuelve $z^* = 302$ como su respuesta para $\varepsilon = 40\%$ la respuesta óptima 307 = 104 + 102 + 101; de hecho, tiene una aproximación de.

$$\frac{Valor\ óptimo * 100\%}{Valor\ obtenido} = \frac{307}{302} = \mathbf{98.37}\ \%$$

RESULTADOS

Se puso a prueba el algoritmo con diferentes entradas de tamaño *n*; *n* representa el número de elementos del conjunto y se **comparó** con la implementación clásica con la finalidad de observar las características en función del tiempo del algoritmo de aproximación para SUBSET-SUM los resultados obtenidos son:

Se muestra una lista con los tiempos de ejecución totales para: Algoritmo de aproximación y algoritmo clásico.

Tamaño de entrada *n*	Conjunto Potencia 2^n	**Algoritmo de Aproximación (segundos)**	**Algoritmo Clásico (segundos)**	**Diferencia**
5	32	0.002146	0.001871	- 87.18%
10	1024	0.002518	0.004472	**+177.60%**
20	1048576	0.009648	1.895628	+196.47%
25	33554432	0.014034	75.07551	+5349.54%
26	67108864	0.015596	162.88581	+10444.07%
27	134217728	0.016705	331.98794	+19873.56%
28	268435456	0.017638	**691.35722**	+39197.03%
30	2^{30}	0.021640	Intratable	
40	2^{40}	0.034925	Intratable	
50	2^{50}	0.079997	Intratable	
100	2^{100}	0.481855	Intratable	
500	2^{500}	42.44653	Intratable	
1000	2^{1000}	**262.5024**	Intratable	
10000	2^{10000}	Intratable	Intratable	

Tabla 1: Tabla comparativa entre los dos algoritmos implementados para SUBSET-SUM, Fuente: Elaboración propia.

En los siguientes gráficos se muestran las funciones de crecimiento en función del tiempo para ambos algoritmos:

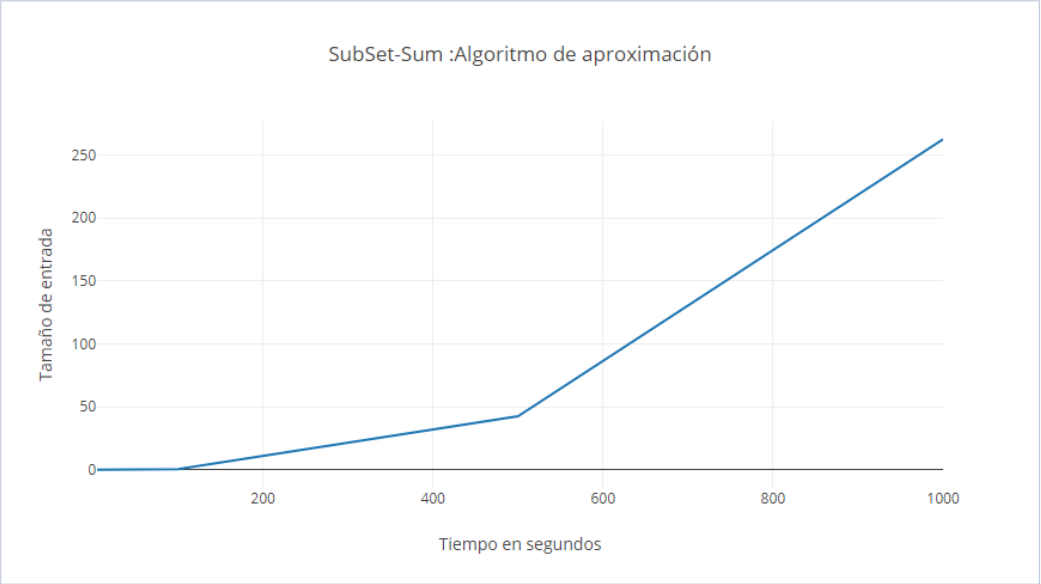

Figura 2: Función de crecimiento para algoritmo de aproximación, Fuente: Elaboración propia

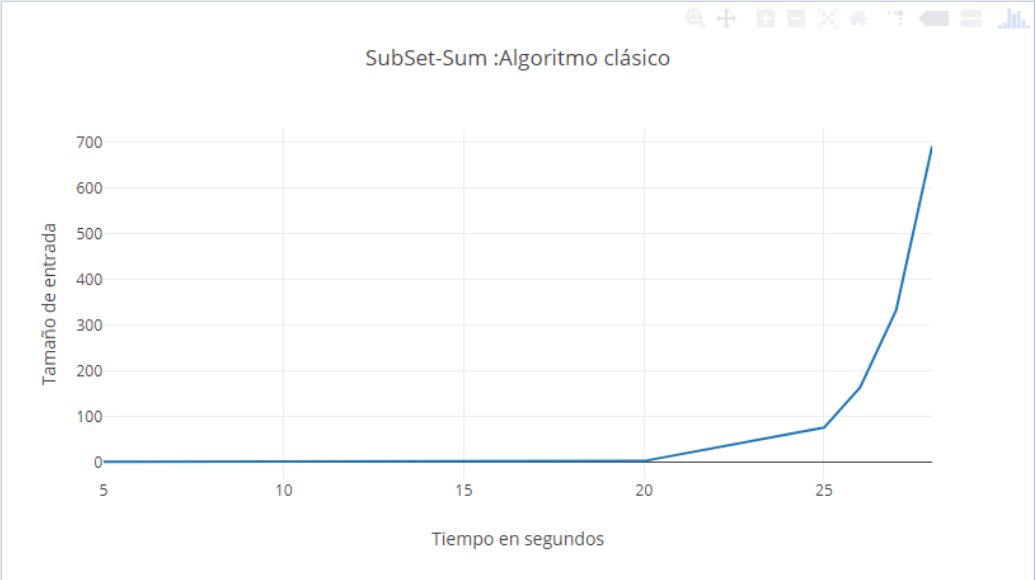

Figura 3: Función de crecimiento para algoritmo clásico, Fuente: Elaboración propia

BIBLIOGRAFÍA

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Third ed.). The MIT Press.

Goodrich, M. T., & Tamassia, R. (2002). *Algorithm Design: Foundations, Analysis, and Internet Examples.* John Wiley & Sons, Inc.

Goodrich, M. T., & Tamassia, R. (2014). *Algorithm Design and Applications.* John Wiley & Sons, Inc.

Linz, P. (2012). *An introduction to formal languages and automata* (Fifth ed.). Jones & Bartlett Learning.

Perras, J. (2014). *Bartering for Beers with Approximate Subset Sums*. Retrieved from Nerderati Blog: http://nerderati.com/2014/08/19/bartering-for-beers-with-approximate-subset-sums/

Sipser, M. (2006). *Introduction to the Theory of Computation* (Second ed.). Thomson Learning, Inc.

Vazirani, V. V. (2003). *Approximation Algorithms.* Springer-Verlag Berlin Heidelberg.

Williamson, D. P., & Shmoys, D. B. (2010). *The Design of Approximation Algorithms.* Cambridge University Press.