Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

SCC0205 – Teoria da Computação e Linguagens Formais

Trabalho 3 – Problema da Soma dos Subconjuntos

Explicação e Prova de **NP**-*completude*

Elias Italiano Rodrigues – 7987251
Gabriel Tessaroli Giancristofaro – 4321350
Paulo Augusto de Godoy Patire – 7987060

São Carlos, 1º de dezembro de 2014

Sumário

1 Introdução

1.1 NP-*completo*

Como visto na disciplina de Teoria da Computação e Linguagens Formais, não é possível resolver a grande maioria dos problemas em computação, ou porque são indecidíveis ou porque não se conhece algoritmo capaz de resolvê-los em tempo polinomial. Geralmente, consideramos os problemas que são resolvidos por algoritmos de tempo polinomial como tratáveis, ou fáceis, e os demais problemas que requerem tempo super-polinomial como intratáveis, ou difíceis.

Na classe de problemas **NP**-*completo* estão os problemas para os quais ainda não foi descoberto um algoritmo de tempo polinomial que os resolva, nem ainda alguém conseguiu provar que um algoritmo de tempo polinomial pode existir para algum desses problemas. A questão "**P** = **NP**?" é justamente sobre essa última afirmação e tem sido uma das questões abertas mais perplexas da Teoria da Computação desde que foi proposta em 1971.

Considerando que **P** $\neq$ **NP**, o diagrama da Figura 1 representa onde se encontram os problemas da classe **NP**-*completo*.

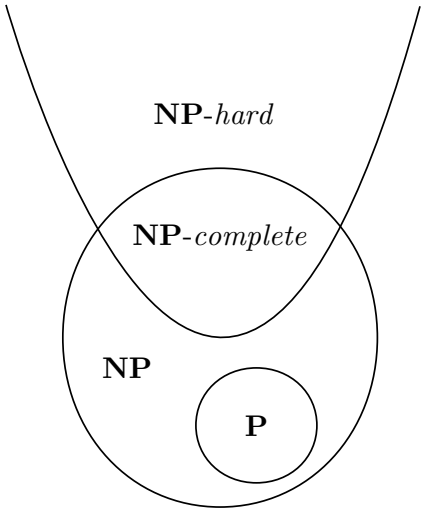

Figura 1: Diagrama das classes de problemas.

1.2 Objetivo

Este trabalho, dentro dos tópicos de Classes de Complexidade (*Complexity Classes*) da disciplina de Linguagens Formais e Teoria da Computação, consiste em explicar do que se trata o Problema da Soma dos Subconjuntos (*Subset Sum Problem*) e apresentar uma prova de sua **NP**-*completude*. Por ser um problema conhecido e já estudado, este trabalho é um **exercício de pesquisa** em que são consultadas referências na literatura e sintetizadas aqui pelo grupo.

Todo material pesquisado para este trabalho encontra-se nos capítulos 34 e 35 do livro *Introduction to Algorithms* [1] sugerido como referência nos *slides* de aula da disciplina.

2 Soma dos Subconjuntos

2.1 Enunciado

Assim como definido no enunciado do Trabalho 3 da disciplina, o problema é o seguinte:

SUBSET-SUM: dado um conjunto finito de números naturais S e um valor $t \in \mathbb{N}$, existe algum subconjunto não-vazio $S' \subseteq S$ em que seus elementos somados resultem em t?

Ou pode-se definir o problema mais formalmente como uma linguagem:

SUBSET-SUM $= \{\langle S, t\rangle : \text{existe um subconjunto } S' \subseteq S \text{ tal que } t = \sum_{s \in S'} s\}$.

Trata-se de um **problema de decisão**, uma vez que é esperada uma resposta *sim* ou *não*.

2.2 Explicação

Pensando em uma linguagem de programação, pode-se imaginar o conjunto S como um vetor de dados `unsigned int` de tamanho `n`, e procura-se por algum subconjunto dele em que a soma dos valores resulte em um valor de entrada `t`.

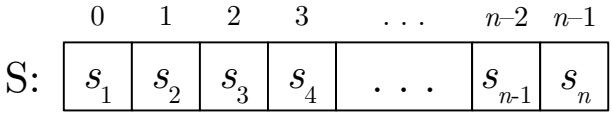

Figura 2: Ilustração de um vetor para o conjunto S.

Pode-se observar que há uma grande quantidade de subconjuntos para o vetor `S` em função do seu tamanho `n`, pois o total de subconjuntos é:

$$\sum_{i=1}^{n} \binom{n}{i} = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n-1} + \binom{n}{n} = 2^n - 1$$

Ou seja, é o número de subconjuntos com 1 elemento, mais o número de subconjuntos com 2 elementos etc, até o último subconjunto que é quando $S' = S$, e desconsiderando o conjunto vazio quando $i = 0$.

Dado esse total de subconjuntos possíveis, no pior caso, para encontrar um subconjunto que satisfaça o SUBSET-SUM e responda *sim*, um algoritmo ingênuo teria complexidade exponencial de $O(2^n)$, pois verificaria todos eles. Por exemplo, um conjunto S de tamanho $n = 100$ tem $2^{100} - 1 = 1\ 267\ 650\ 600\ 228\ 229\ 401\ 496\ 703\ 205\ 375$ subconjuntos não-vazios.

3 Provas

3.1 Subset-Sum está em NP

Dado que **NP** é o conjunto dos problemas de decisão em que uma possível solução pode ser verificada em tempo polinomial, para mostrar que SUBSET-SUM está em **NP** para uma instância $\langle S, t\rangle$ do problema, toma-se o subconjunto S' para ser verificado. O Algoritmo 1 de verificação pode checar se $\sum_{s \in S'} s = t$ em tempo polinomial $O(n)$.

Algoritmo 1: Verifica uma solução S' do SUBSET-SUM em tempo polinomial.

Input: $\langle S', t\rangle$
Output: *true* se S' é solução, *false* caso contrário

```
1 sum ← 0;
2 foreach s ∈ S' do
3 |   sum ← sum + s;
4 end
5 if sum = t then
6 |   return true;
7 end
8 return false;
```

3.2 Subset-Sum é NP-*completo*

3.2.1 Redução

Antes de provar a **NP**-*completude* do SUBSET-SUM, é necessário conhecer os problemas canônicos usados para prová-lo por **redução**.

A técnica de provar por redução que um problema B é **NP**-*completo*, requer conhecer um problema A que já foi provado ser **NP**-*completo* e então encontrar uma maneira de reduzir $A \rightarrow B$ "refraseando" A ou encontrando um algoritmo que converta qualquer instância de A em uma instância de B em tempo polinomial. Assim, pode-se dizer que B é tão difícil quanto A, ou ainda, que B não é mais do que um fator polinomial difícil do que A.

Ou seja, se $A \leq_{\mathbf{P}} B$ e $A \in$ **NP**-*completo* $\Rightarrow B \in$ **NP**-*completo*.

3.2.2 Problemas de Satisfatibilidade Conhecidos

Circuit-Sat é um problema canônico de satisfatibilidade provado pertencer a **NP**-*completo*. Seu enunciado é: dado um circuito booleano combinacional, ele é satisfatível?

Ou, mais formalmente como uma linguagem:

CIRCUIT-SAT $= \{\langle C\rangle : C$ é um circuito booleano combinacional satisfatível $\}$.

Um circuito booleano combinacional é um conjunto de variáveis que podem receber os valores 0 (*false*) ou 1 (*true*) e que de acordo com as operações lógicas AND ($\wedge$), OR ($\vee$) e NOT ($\neg$) pode resultar em uma resposta 0 ou 1. Dado um circuito C, pode-se determinar se ele é

satisfatível simplesmente testando todas as atribuições possíveis para suas n variáveis, que é um total de 2^n atribuições.

O Circuit-Sat foi provado ser **NP**-*completo*, pois, tratando-o mais abstratamente como uma linguagem $L \subseteq \{0,1\}^*$, satisfaz as seguintes propriedades:

1. $L \in \mathbf{NP}$, e
2. $L' \leq_{\mathbf{P}} L$ para todo $L' \in \mathbf{NP}$.

SAT é um problema de decidir se uma fórmula booleana ϕ é satisfatível.

Como linguagem: SAT = $\{\langle\phi\rangle : \phi$ é uma fórmula booleana satisfatível $\}$.

Um fórmula booleana, além dos operadores $\wedge$, $\vee$, $\neg$, contém também $\implies$ (implicação) e $\iff$ (se e somente se). O problema Circuit-Sat pode ser reduzido ao SAT e logo também SAT pertence a **NP**-*completo*.

3-CNF-SAT é um problema de decidir se uma fórmula booleana ϕ em 3-CNF é satisfatível.

Como linguagem: 3-CNF-SAT = $\{\langle\phi\rangle : \phi$ é uma fórmula booleana 3-CNF satisfatível $\}$.

A 3-CNF – Forma Normal Conjuntiva com 3 literais – é uma maneira canônica de expressar uma fórmula booleana como uma conjunção de disjunções em que cada cláusula tem exatamente 3 literais – um literal é a ocorrência de uma variável ou de sua negação. Por exemplo, a seguinte fórmula é 3-CNF: $(x_1 \vee \neg x_3 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$. Seguindo o diagrama da Figura 3, o 3-CNF-SAT é provado ser **NP**-*completo* por redução do SAT.

A prova de **NP**-*completude* dos problemas citados nesta Seção está fora do escopo deste trabalho, mas seu conhecimento é necessário para ser usado na prova de **NP**-*completude* do Subset-sum.

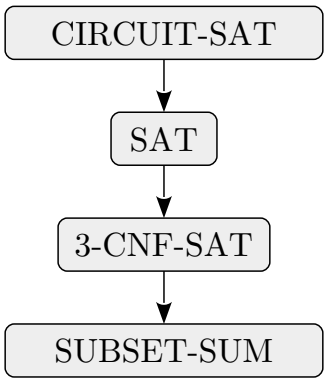

Figura 3: Dependência de problemas para a prova de **NP**-*completude* do Subset-Sum.

3.2.3 Subset-Sum usando 3-CNF-SAT

Dada uma fórmula ϕ sobre as variáveis $x_1, x_2, \ldots, x_n$ com cláusulas $C_1, C_2, \ldots, C_k$ cada uma contendo exatamente três literais distintos, o algoritmo de redução contrói uma instância $\langle S, t\rangle$ do Subset-sum tal que ϕ é satisfatível se, e somente se, existe um subconjunto de S cuja soma é exatamente t.

Toma-se os devidos cuidados para que ϕ não contenha um literal e sua negação em uma mesma cláusula e cada variável de ϕ apareça pelo menos uma vez em alguma cláusula.

A redução cria dois números em S para cada variável x_i e dois números em S para cada cláusula C_j. Assim construímos S e t da seguinte maneira. Para cada valor $s \in S$, nomeamos cada posição de seus dígitos ou por uma variável ou por uma cláusula de ϕ. Os k menos significantes dígitos de cada s são nomeados pelas cláusulas e seus n mais significantes dígitos são nomeados pelas variáveis.

- O valor t tem 1 em cada dígito nomeado por uma variável e 4 em cada dígito nomeado por uma cláusula.
- Para cada variável x_i, o conjunto S contém dois inteiros v_i e v'_i. Cada v_i e v'_i tem 1 no dígito nomeado por x_i e 0 nos demais dígitos de variáveis. Se o literal x_i aparece numa cláusula C_j, então o dígito nomeado por C_j em v_i contém 1. Se o literal $\neg x_i$ aparece numa cláusula C_j, então o dígito nomeado por C_j em v'_i contém 1. Todos demais dígitos nomeados por cláusula em v_i e v'_i são 0.
  Dessa maneira, tomado os devidos cuidados citados acima para a fórmula ϕ, todos os v_i e v'_i são únicos.
- Para cada cláusula C_j, o conjunto S contém dois inteiros r_j e r'_j. Cada r_j e r'_j tem 0 em todos os dígitos que não sejam aqueles nomeados por C_j. Para r_j, existe um 1 no dígito de C_j e r'_j tem um 2 neste dígito. Esses inteiros são usados para fazer somar até 4 nos dígitos nomeados por cláusulas.

Essa redução pode ser feita em tempo polinomial. O conjunto S contruído contém $2n + 2k$ valores s, cada um contendo $n + k$ dígitos, e o tempo para produzir cada dígito é $O(n + k)$. O valor t também tem $n + k$ dígitos e também é produzido em tempo polinomial.

Mostra-se agora que a fórmula ϕ é satisfatível se, e somente se, existe um $S' \subseteq S$ para os S e t construídos. Primeiro, suponha que existe uma atribuição que satisfaça ϕ. Para cada $i = 1, 2, \ldots, n$, se $x_i = 1$ nessa atribuição, então inclui-se v_i em S'. Caso contrário, inclui-se v'_i. Dessa maneira, a soma de todos os valores incluídos em S' até agora satisfazem os primeiros n dígitos de t. E, para atingir os dígitos 4 de t, inclui-se os subconjuntos não-vazios apropriados de variáveis $\{r_j, r'_j\}$ de acordo com as claúsulas C_j.

Agora, suponha que existe um subconjunto $S' \subseteq S$ que seus valores some t. O subconjunto S' tem que incluir exatamente um v_i ou um v'_i para cada $i = 1, 2, \ldots, n$. Caso contrário, os n primeiros dígitos de t não seriam alcançados na soma. E pode-se afirmar que todas claúsulas C_j, $j = 1, 2, \ldots, n$ também são satisfeitas, pois para alcançar a soma 4 nos k dígitos menos significativos de t, o subconjunto S' deve incluir pelo menos um v_i ou um v'_i que tem dígito 1 em C_j, e uma vez que as contribuições r_j e r'_j somam no máximo 3, consegue-se totalizar os valores 4 para esses k dígitos.

Por exemplo: para $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, onde $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$ e $C_4 = (x_1 \vee x_2 \vee x_3)$, uma atribuição que satisfaz ϕ é $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$. O conjunto S construído pela redução é $S = \{$ 1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2 $\}$ e $t = 1114444$. Como ilustrado na Figura 4.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
r_1	=	0	0	0	1	0	0	0
r'_1	=	0	0	0	2	0	0	0
r_2	=	0	0	0	0	1	0	0
r'_2	=	0	0	0	0	2	0	0
r_3	=	0	0	0	0	0	1	0
r'_3	=	0	0	0	0	0	2	0
r_4	=	0	0	0	0	0	0	1
r'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Figura 4: Ilustração do conjunto S para uma fórmula ϕ. As linhas em azul claro são os valores de S' que satisfazem a soma t.

4 Algoritmos

4.1 Exponencial

Para o Algoritmo 2 que responde certamente a pergunta do Subset-Sum, considere que:

- Os n elementos de S são numerados $s_1, s_2, \cdots, s_n$.
- Existe uma função Merge-List(L, L') de complexidade $O(|L| + |L'|)$ que faz a união de duas listas ordenadas L e L' e retorna em uma lista ordenada sem elementos repetidos.
- A notação $L + s$ denota uma lista com os números de L acrescidos de s. Por exemplo, se $L = \{1, 4, 5\}$, então $L + 3 = \{4, 7, 8\}$.
- Existe uma função Search-List(L, l) de complexidade $O(\log_2 |L|)$ que busca se um elemento l pertence a lista L e retorna *true* ou *false*.

Algoritmo 2: Responde a pergunta do SUBSET-SUM em tempo exponencial.

Input: $\langle S, t \rangle$
Output: *sim* se existe uma solução, *não* caso contrário

```
1 n ← |S|;
2 L_0 ← {0};
3 for i ← 1 to n do
4 |   L_i ← MERGE-LIST(L_{i-1}, L_{i-1} + s_i);
5 |   if SEARCH-LIST(L_i, t) then
6 |   |   return sim;
7 |   end
8 end
9 return não;
```

Como o tamanho da lista L_i pode chegar a 2^n, o Algoritmo 2 é em geral de tempo exponencial $O(2^n)$. Para explificá-lo, considere o exemplo dados como entrada $S = \{4, 1, 0, 6, 2\}$ e $t = 7$:

$$
\begin{aligned}
L_0 &= \{0\} \\
L_1 &= L_0 \cup L_0 + 4 = \{0, 4\} \\
L_2 &= L_1 \cup L_1 + 1 = \{0, 1, 4, 5\} \\
L_3 &= L_2 \cup L_2 + 0 = \{0, 1, 4, 5\} \\
L_4 &= L_3 \cup L_3 + 6 = \{0, 1, 4, 5, 6, \mathbf{7}, 10, 11\}
\end{aligned}
$$

Então o algoritmo para e retorna *sim*, pois na iteração 4 foi encontrada uma soma para o valor $t = 7$. Caso nenhuma soma com valor 7 fosse encontrada, no final o algoritmo retornaria *não*.

4.2 Pseudo-polinomial com Programação Dinâmica

Em Teoria da Complexidade, um algoritmo executa em tempo pseudo-polinomial se seu tempo de execução é polinomial *no valor numérico* da entrada, mas é exponencial no *comprimento* da entrada – número de bits requeridos para representá-lo.

No caso do SUBSET-SUM, pode-se fazer um algoritmo que execute em função do *valor numérico* da soma max de todos os n elementos de S.

O Algoritmo 3 demonstra esse algoritmo pseudo-polinomial usando a técnica de programação dinâmica que armazena os valores já calculados numa matriz (tabela) Q de tamanho $n \times (max + 1)$. Primeiramente é feita uma inicialização da linha 1 de Q e então o algoritmo itera em relação ao valor max para cada outra linha de Q.

Como é computado se existe resposta para todos os possíveis valores desde 0 até max, a resposta para t encontra-se na posição $Q(n, t)$ da matriz, mas todas as outras respostas para qualquer valor $0 \leq v \leq max$ encontram-se em $Q(n, v)$.

Algoritmo 3: Responde a pergunta do SUBSET-SUM em tempo pseudo-polinomial.

Input: $\langle S, t \rangle$
Output: *sim* se existe uma solução, *não* caso contrário

```
1  n ← |S|;
2  max ← 0;
3  for i ← 1 to n do
4  |  max ← max + s_i;
5  end
6  if t < 0 or t > max then
7  |  return não;
8  end
9  Seja Q uma matriz de tamanho n × (max + 1);
10 for j ← 0 to max do
11 |  Q(1, j) ← false;
12 end
13 Q(1, s_1) ← true;
14 for i ← 2 to n do
15 |  for j ← 0 to max do
16 |  |  if Q(i − 1, j) or j = s_i or Q(i − 1, j − s_i) then
17 |  |  |  Q(i, j) ← true;
18 |  |  else
19 |  |  |  Q(i, j) ← false;
20 |  |  end
21 |  end
22 end
23 if Q(n, t) then
24 |  return sim;
25 end
26 return não;
```

5 Aplicações

O SUBSET-SUM é um problema muito parecido com o Problema da Mochila (KNAPSACK) que é um **problema de otimização** enunciado como: dado um conjunto de n itens i, cada um com uma massa w_i e um valor v_i, determine quais itens pegar de modo que o total de peso seja menor ou igual a um limite W e o total de valor seja maior ou igual a K, desde que não ultrapasse o limite W.

Portanto, o SUBSET-SUM é um caso especial do KNAPSACK, onde $s_i = v_i = w_i$, $t = K = W$, e logo serve como base para seu entendimento – inclusive, poderia ser feita a redução KNAPSACK $\rightarrow$ SUBSET-SUM para provar sua **NP**-*completude*. Por sua vez, o KNAPSACK pode ser aplicado no mundo real para otimização no carregamento de cargas para transporte, empacotamento de produtos, orçamentos e investimentos de capital.

6 Conclusão

Neste trabalho foi apresentado e discutido o Problema da Soma dos Subconjuntos, nomeado aqui por SUBSET-SUM, e além de estudar sua **NP**-*completude* foi possível também analisar algoritmos para a solução do problema.

A realização deste trabalho possibilitou momentos de pesquisa e contato com o formalismo da Teoria da Complexidade e dos Algoritmos assim como é exposto nos livros-texto.

Referências

[1] Cormen, T.H. Leiserson, CE. Rivest, R.L. and Stein, C. *Introduction to Algorithms*. The MIT Press. (3rd ed.).