tiCOMPLEXIDADE DE ALGORITMOS: PROVA QUE O PROBLEMA DA SOMA DE SUBCONJUNTOS É NP-COMPLETO

Clemilson Luís de Brito Dias
Rafael Rebellato Trommenschläger

RESUMO

Este trabalho apresenta o problema da soma de subconjuntos (SUBSET-SUM) através da sua caracterização e identificação do problema. Em seguida, prova-se que o referido problema pertence à classe de problemas NP por meio de um algoritmo de verificação e a análise da complexidade desse algoritmo. Por fim, comprovamos que o problema foco deste artigo pertence à categoria dos problemas NP-Completos descrevendo o problema da satisfabilidade de fórmulas - com cada cláusula contendo exatos 3 literais - (3-SAT) e criando um algoritmo de redução do problema 3-SAT para uma instância do problema SUBSET-SUM, para provar que o problema é NP-Difícil, finalizando com a análise da complexidade desse algoritmo de redução.

Palavras-chave: Soma de subconjuntos. SUBSET-SUM. 3-SAT. NP-Completo.

INTRODUÇÃO

O estudo da análise da complexidade de algoritmos nos leva a trabalhar com algoritmos tradicionais que, em grande maioria, têm sua solução gerada num tempo polinomial em relação a sua entrada. Esses algoritmos resolvem problemas que pertencem à classe chamada P. No entanto, existem outras classes de problemas, como a NP, NP-Completo, NP-Difícil, etc. Para todas as classes, exceto a P, não existe um algoritmo de decisão que forneça uma solução para um problema num tempo polinomial.

Esse artigo trata de problemas da classe NP-Completo. Essa classe de problemas tem por característica fundamental o fato de sempre existir um algoritmo de decisão que verifica, em tempo polinomial, se uma suposta solução é válida para um problema.

O objeto é provar que um problema, no caso o problema da soma de subconjuntos (SUBSET-SUM), pertence à classe NP-Completo através da aplicação de um algoritmo de redução do problema 3-SAT, já conhecido por ser NP-Completo, a uma instância do SUBSET-SUM.

1. CONCEITOS IMPORTANTES

1. PROBLEMAS DE DECISÃO x PROBLEMAS DE OTIMIZAÇÃO

Quando nos referimos a problemas de otimização, estamos diante de um cenário em que temos várias soluções diferentes para determinado problema. Problema de otimização consiste em selecionar a melhor forma de resolver esse problema, ou seja, não basta resolvê-lo, temos que fazer isso da maneira mais eficiente.

Já os problemas de decisão consistem em responder se existe ou não uma solução a um determinado problema. Com isso, espera-se sempre de um problema de decisão uma resposta binária: "1" ou "0" (ou "sim" ou "não").

É possível transformar problemas de otimização em problemas de decisão impondo um limite sobre o valor a ser otimizado. Existem vários exemplos na literatura sobre isso.

Na classe de problemas NP-Completos, tratados nesse artigo, vamos considerar os problemas de decisão. Isso se deve ao fato de esses problemas terem a propriedade de ser codificados por linguagens. Com isso, o objeto seria descobrir se uma palavra pertence ou não a uma linguagem definida.

Para o problema de decisão do SUBSET-SUM, podemos dizer que é um problema computacional cujo objetivo seria decidir se existe um subconjunto de K tal que a soma dos k valores desse subconjunto seja igual a t.

2. REDUTIBILIDADE

A redutibilidade de problemas é um recurso muito importante para uma análise relativa dos problemas da classe NP, além de determinar sua afinidade.

Se um problema A pode ser reduzido a um problema B, isso significa que as respostas obtidas em B devem ser idênticas as que seriam obtidas em A, ou seja, um algoritmo de redução transforma qualquer instância de A numa instância de B.

Os algoritmos de redução que interessa a esse estudo são os executados em tempo polinomial.

Com a aplicação de um algoritmo de redução de A em B, podemos chegar a algumas conclusões:

- B é, no mínimo, tão difícil quanto A.
- Se B é solúvel, A também será.
- A pode ser uma parte fácil de B.
- Se A não tem solução em tempo polinomial, B também não terá.
- A jamais será mais difícil que B.

3. CLASSES DE PROBLEMAS

1. PROBLEMAS P

Essa classe de problemas consiste nos problemas que podem ser solucionados por um algoritmo em tempo polinomial, ou seja, existe um algoritmo que resolve o problema num tempo máximo $O(n^k)$, sendo k uma constante e n o tamanho da entrada de dados do problema. Outra forma de compreender essa definição é que, para essa classe de problemas, existe uma máquina de Turing determinística que fornece uma solução em tempo polinomial.

2. PROBLEMAS NP

Consiste nos problemas em que uma resposta dada previamente pode ser verificada rapidamente (em tempo polinomial), no entanto, não há nenhuma maneira conhecida para encontrar uma resposta em tempo polinomial. Um problema NP pode ser equivalentemente definido como um problema de decisão que pode ser solucionado em tempo polinomial em uma máquina de Turing não determinística.

3. PROBLEMAS NP-COMPLETOS

Os problemas da classe NP-Completo é um conjunto de problemas que podem ser reduzidos em tempo polinomial a partir de qualquer outro problema NP (todo problema em NP se pode reduzir ao NP-completo), mas cuja solução ainda pode ser verificada em tempo polinomial.

Se fosse possível encontrar uma maneira de resolver qualquer problema NP-Completo rapidamente (em tempo polinomial), então poderiam ser utilizados algoritmos para resolver todos os problemas NP rapidamente e, dessa forma, poderíamos afirmar que as classes P = NP. Esse é um dos maiores desafios da ciência da computação atualmente.

4. PROBLEMAS NP-DIFÍCIL

Essa última classe de problemas consiste na categoria em que existe um algoritmo de redução que pode ser executado em tempo polinomial

5. ORGANIZAÇÃO DAS CLASSES DE PROBLEMAS

Na literatura existem algumas variações com relação às classes de problemas e suas ligações. Consideramos para esse artigo aquela que é a mais aceita atualmente e trabalhada pela maioria dos autores. Na imagem abaixo, segue um diagrama da distribuição das classes de problemas:

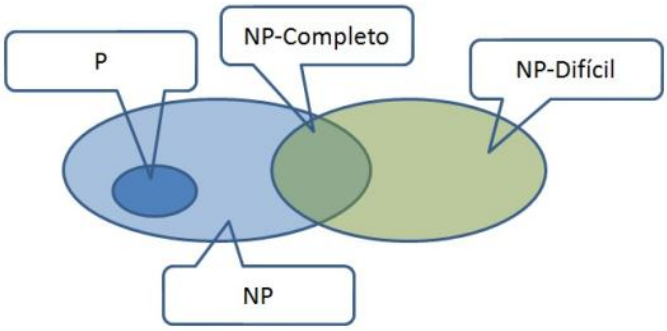


Figura 1 - Diagrama que relaciona as classes de problemas.

2. O PROBLEMA DA SOMA DE SUBCONJUNTOS (SUBSET-SUM)

Como já foi citado anteriormente, o objetivo desse artigo é provar que o problema da soma de subconjuntos, que vamos tratar de forma abreviada como SUBSET-SUM, é um problema NP-Completo.

Para relembrar, no SUBSET-SUM, dados números naturais *p1,...,pn* e *c*, questionamos se existe um subconjunto K de {1,...,*n*} tal que a soma dos *pk* para cada *k* em K seja igual a *c*.

1. ALGORITMO DE VERIFICAÇÃO E SUA COMPLEXIDADE

Um algoritmo de verificação consiste num algoritmo que recebe como parâmetro, no caso do SUBSET-SUM, um conjunto nomeado K, um subconjunto de valores K1, chamado de certificado, e um valor t, que é o valor do resultado esperado. Sua função é verificar se o certificado (K1) fornecido gera o resultado esperado (t) retornando, em tempo polinomial, *TRUE* ou *FALSE*, por exemplo. Além disso, verificamos se o subconjunto está dentro dos padrões esperados, como não possuir números repetidos em K1, nem possuir elementos que não estão contidos no conjunto K.

Com isso, fica comprovado que o problema SUBSET-SUM pertence à classe NP. O pseudocódigo do algoritmo de verificação é apresentado a seguir:

```
Function Verifica_SubsetSum (K, K1, t ) :
        int soma = 0;
        for each k in Kl :
                soma = soma + k;
        if (soma != t)
                return FALSE
        else
                int array[sizeof(K1)];
                array <= Fill with zeros;
                for i = 1 to sizeof(Kl):
                        for each n in K:
                                if( K1[i] == n)
                                        array[i]= array[i]+1;
                for i = 1 to sizeof(Kl) :
                        if( array[i] != 1)
                                return FALSE
                        else
```

```
                                    return TRUE
End Function Verifica_SubsetSum
```

Se considerarmos a soma como operação elementar, k o número de elementos de Kl e n o número de elementos de K, vemos que o código acima possui complexidade O(kn), uma vez que a iteração da linha 10 e 11 executará k * n vezes. As outras complexidades do problema não influenciam, pois essa é a complexidade dominante. Dessa forma, podemos considerar a complexidade do algoritmo como sendo O(kn), ou seja, polinomial.

Assim, mostramos que o problema da soma do subconjunto pertence à classe NP, já que seu algoritmo de verificação é executado em tempo polinomial.

Exemplos de verificação:

Seja K o conjunto {10, 20, 46, 62, 70, 90} e t o número alvo 226.

Consideremos o subconjunto K1 = {20, 46, 70, 90}. A soma dos valores de K1 é igual a 226. Como este valor é igual a "t" e todos os valores de K1 pertencem a K, temos um retorno positivo na verificação do problema.

2. SUBSET-SUM PERTENCE À CLASSE NP-COMPLETO

1. O PROBLEMA DE SATISFABILIDADE 3-SAT

Satisfabilidade: Consiste em determinar se existem valores-verdade para um conjunto de variáveis que tornam certa fórmula booleana verdadeira.

O problema 3-SAT consiste em verificar a satisfabilidade de fórmulas booleanas com três literais em cada termo.

Dada uma expressão lógica, na forma normal conjuntiva - CNF - com n variáveis booleanas e m conectivos lógicos NOT, AND e OR, onde cada termo contém exatamente três literais, a solução de um algoritmo de verificação consiste no conjunto de valores a serem atribuídos às variáveis da fórmula.

Após, avaliamos o resultado e se a atribuição é satisfatória, a solução é válida. Esse procedimento de verificação pode ser realizado em tempo polinomial ou até mais rápido. Com isso, conclui-se que o problema 3-SAT é NP.

1. 3-SAT É NP-COMPLETO

Conforme o teorema de Cook (1971), o problema da satisfabilidade booleana, conhecido como SAT, é NP-Completo. Esse foi o primeiro problema da literatura a ser provado NP-Completo.

Já existe na literatura um algoritmo de redução do problema SAT para uma instância do 3-SAT. Com isso, pode-se afirmar que o problema 3-SAT é NP-Completo.

2. A REDUÇÃO DE 3-SAT PARA UMA INSTÂNCIA DE SUBSET-SUM

Para que seja possível provar que o problema SUBSET-SUM é um problema NP-Completo, precisamos de um algoritmo que faça a redução de outro problema NP-Completo a uma instância de SUBSET-SUM. Para isso, vamos utilizar o problema 3-SAT, já provado ser NP-Completo como base para o nosso propósito.

3. IDEALIZAÇÃO DO ALGORITMO DE REDUÇÃO 3-SAT A UMA INSTÂNCIA DE SUBSET-SUM

Definições iniciais:

- K é o conjunto de número de SUBSET-SUM.
- K1 é um subconjunto de K que contém os valores cuja soma é igual a t.
- *t* é o número no qual desejamos encontrar a soma em K.
- c_i é uma cláusula das equações booleanas de 3-SAT.

- x_i é uma variável das equações booleanas de 3-SAT.

Para simplificar o processo do algoritmo de redução, faremos duas suposições iniciais que não trarão perda de generalidade ao processo. São elas:

- Nenhuma cláusula da fórmula booleana pode conter, ao mesmo tempo, uma variável e sua negação.
- Cada variável deve aparecer em, pelo menos, uma cláusula.

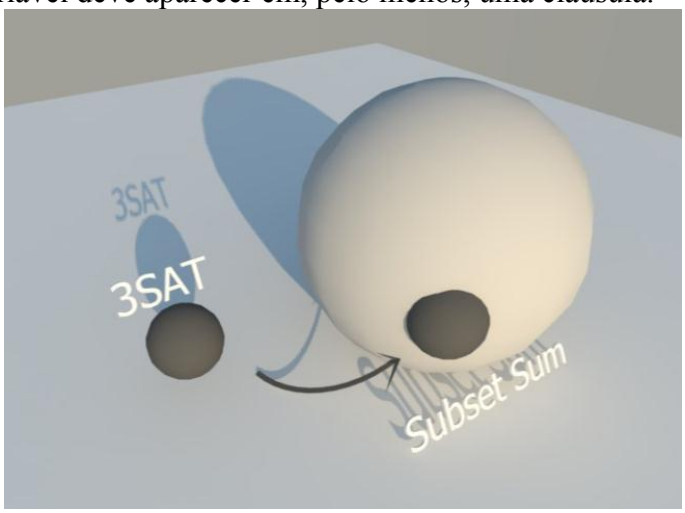


Figura 2: Redução do 3-SAT ao problema do SUBSET-SUM

Tais condições são óbvias, visto que se uma variável e sua negação estiverem na mesma cláusula, essa será verdadeira. E se uma variável não estiver em nenhuma cláusula, não faz sentido considerá-la para fins de validação da equação.

As *c* cláusulas e *x* variáveis das equações booleanas de 3-SAT serão convertidas em (2c + 2*x*) números no conjunto K de SUBSET-SUM.

O algoritmo de redução cria dois números em K para cada variável x_i e dois números em K para cada cláusula c_i.

Os números criados em K são na base 10.

Cada número criado em K contém *c* + *x* dígitos e cada dígito corresponde a uma variável ou uma cláusula.

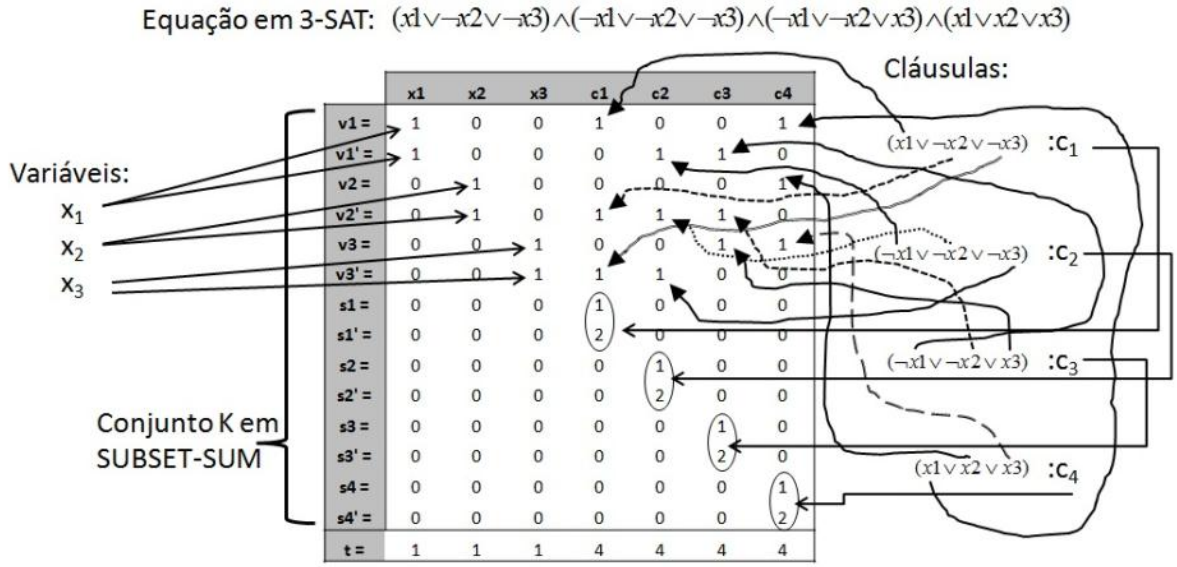
Os números em K e o valor de *t* são formados da seguinte forma:

- Os *x* dígitos mais significativos representam as *x* variáveis de 3-SAT e, no caso de *t*, serão iguais a 1.
- Os *c* dígitos menos significativos representam as *c* cláusulas de 3-SAT e, no caso de *t*, serão iguais a 4. Esse 4 refere-se ao fato de estarmos utilizando o 3-SAT. Resumindo, os dígitos menos significativos de t serão iguais a p+1, quando o algoritmo em uso for um p-SAT.
- Com isso, alguns exemplos de valores válidos de *t* são 111444, 1114444, 1111144, etc.
- Para cada variável *xi*, existem dois inteiros, *vi* e *vi′* em K. Cada um tem o valor 1 no dígito identificado por *xi*, e valores 0 nos outros dígitos de variáveis. Se o literal *xi* aparece na cláusula *cj*, então o dígito identificado por *cj* em *vi* contém um valor 1. Se o literal *xi* aparece na cláusula *cj*, então o dígito identificado por *cj* em *vi′* contém um valor 1. Todos os outros dígitos identificados por cláusulas em *vi* e *vi′* são 0.
- Para cada cláusula *cj* existem dois inteiros *sj* e *sj′* em K. Cada um tem valores 0 em todos os dígitos além do dígito identificado por *cj*. Para *sj*, existe um valor 1 no dígito *cj*, e *sj′* tem o valor 2 nesse dígito. Esses inteiros terão uma função de coringa ao montarmos a soma em busca de *t* conforme será visto mais adiante.

Devido às suposições iniciais feitas no início da descrição do algoritmo, pode-se garantir que não existem valores repetidos em K.

Caso, ao atribuirmos valores para as variáveis das equações, obtivermos um resultado positivo em 3-SAT, então existirá um subconjunto K1 em SUBSET-SUM, cuja soma de seus elementos será igual a *t*.

O gráfico abaixo representa o mapeamento da equação (variáveis e cláusulas) de 3-SAT para o conjunto K e o valor t em SUBSET-SUM:



	x1	x2	x3	c1	c2	c3	c4
v1 =	1	0	0	1	0	0	1
v1' =	1	0	0	0	1	1	0
v2 =	0	1	0	0	0	0	1
v2' =	0	1	0	1	1	1	0
v3 =	0	0	1	0	0	1	1
v3' =	0	0	1	1	1	0	0
s1 =	0	0	0	1	0	0	0
s1' =	0	0	0	2	0	0	0
s2 =	0	0	0	0	1	0	0
s2' =	0	0	0	0	2	0	0
s3 =	0	0	0	0	0	1	0
s3' =	0	0	0	0	0	2	0
s4 =	0	0	0	0	0	0	1
s4' =	0	0	0	0	0	0	2
t =	1	1	1	4	4	4	4

Figura 3 - Mapeamento 3-SAT para SUBSET-SUM

Exemplo 1:

Para facilitar a visualização, vejamos a seguinte tabela:

	x1	x2	x3	c1	c2	c3	c4
v1 =	1	0	0	1	0	0	1
v1' =	1	0	0	0	1	1	0
v2 =	0	1	0	0	0	0	1
v2' =	0	1	0	1	1	1	0
v3 =	0	0	1	0	0	1	1
v3' =	0	0	1	1	1	0	0
s1 =	0	0	0	1	0	0	0
s1' =	0	0	0	2	0	0	0
s2 =	0	0	0	0	1	0	0
s2' =	0	0	0	0	2	0	0
s3 =	0	0	0	0	0	1	0
s3' =	0	0	0	0	0	2	0
s4 =	0	0	0	0	0	0	1
s4' =	0	0	0	0	0	0	2
t =	1	1	1	4	4	4	4

Figura 4: Exemplo válido utilizando a tabela da execução do algoritmo

Supondo os valores x1 = 0, x2 = 0 e x3 =1 para as variáveis na equação booleana em 3-SAT, obteremos um resultado positivo. Veremos agora a resposta gerada pelo SUBSET-SUM:

As linhas claras na tabela correspondem aos números selecionados para K1 baseados nos valores das variáveis x1, x2 e x3. Assim, K1 será formado pelos seguintes números:

K1 = {1000110; 101110; 10011; 1000; 2000; 200; 10; 1; 2}

Com isso, ao somarmos os elementos de K1, obteremos o seguinte resultado:

$\sum_{s \in K1} = 1114444$ que é igual a t.

Podemos inferir que a combinação de valores lógicos de entrada gera uma saída válida para o problema 3-SAT. Após a redução feita do 3-SAT para o SUBSET-SUM, e a aplicação da mesma solução aplicada em 3-SAT, temos um resultado positivo, ou seja, a soma dos valores de K1 é igual a t.

Exemplo 2:

No exemplo a seguir, vamos supor uma nova solução para a equação 3-SAT: x1 = 0, x2 = 0 e x3 = 0.

Analisando a tabela, onde as linhas claras são as linhas selecionadas conforme a solução proposta, verificamos que não há nenhuma saída válida para a cláusula 4, pois a soma máxima dos termos seria 3, case considerássemos as duas linhas em vermelho selecionadas.

Abaixo segue a imagem da tabela:

Figura 5: Exemplo inválido utilizando a tabela da execução do algoritmo

4. DEFINIÇÃO DO ALGORITMO DE REDUÇÃO 3-SAT A UMA INSTÂNCIA DE SUBSET-SUM

Para o algoritmo de redução do 3-SAT para o SUBSET-SUM, vamos supor uma estrutura de dados chamada "Equation", que é composta por duas sub-coleções (arrays) chamadas "Literals", que contém as variáveis e "Clauses" que contém as cláusulas da equação booleana de 3-SAT. K é o conjunto dos valores para o SUBSET-SUM e t é o valor alvo da soma que é gerado através de uma string que vai sendo concatenada a cada variável/ cláusula processada.

A seguir, uma versão em pseudo código do algoritmo de redução do problema 3-SAT para SUBSET-SUM:

```
Function Reducao_3SATtoSubsetSum (Equation)
        if(!=verify_clauses(Equation.clauses);)
                return ERROR
        else
                        M = create_matrix(sizeof(Equation.clauses),
                                        sizeof(Equation.Literals));
                        for each different lit in Equation.literals
                                create_lits_T_F_column(M,lit);
                        for each clause in Equation.clauses
                                create_var_clause_column(M,clause));
                        if(everything_is_ok(M,K,t))
                                return OK
                        else
                                return ERROR
End Function Reducao_3SATtoSubsetSum
```

Algoritmos auxiliares:

```
Function verify_clauses(clauses)
        for each clause in clauses
                if (sizeof(clause.literals)) != 3) //verifica se tem 3
                                                   //literais em cada cláusula
```

```
                    return FALSE
            elseif (clause.literals(1)== clause.literals(2) ||
                     clause.literals(2)== clause.literals(3) ||
                     clause.literals(1)== clause.literals(3))
                    // verifica se todos são diferentes
                    return FALSE
            elseif (clause.literals(1)==not clause.literals(1) ||
                    clause.literals(2)==not clause.literals(2) ||
                    clause.literals(3)==not clause.literals(3))
        // verifica se nεo tem um literal e seu negado na mesma cláusula
                    return FALSE
            else
                    return TRUE

Function create_matrix(c,x);
        int matriz [sizeof(c + x)][sizeof(2*c + 2*x)]; //cria matriz de tamanho c+x por 2*c+2*x
        for (int i = 0; i < sizeof(c + x); i++)  // zera todos elementos da matriz
                for (int j = 0; j < sizeof(2*c + 2*x); j++)
                        matriz[i][j] = 0;
return matriz;
        <== na verdade, teria que ter entrado com um ponteiro para uma matriz.. mas para fins didбticos deixei assim

Function create_lits_T_F_column(M,lit)
        //adiciona na matriz em seu respectivo lugar o valor 1 em seu
        //literal e 1 na sua negação para seus "v" e "v'" respectivos
        //ex.: (    lit1=1 em i=0 e j=0,1
        //          lit2=1 em i=1 e j=2,3
        //          lit3=1 em i=2 e j=4,5
        //          etc...)

Function create_var_clause_column(M,clause)
        //para cada cláusula, adiciona na matriz em seu respectivo lugar o valor 1 nos lugares indicados pela
        //cláusula criar também os coringas, com os valores 1 e 2 para os seus "s" e "s'" respectivos

Function everything_is_ok(K)
        //concatena cada linha da matriz e gera um número em K. A concatenação da última linha gera t, logo, não é
        //adicionada a K. Se t não estiver no padrão 111*444* retorna FALSE, senão TRUE
```

Analisando a complexidade dos algoritmos auxiliares, resumidamente temos:

- *verify_clauses*: O(c), visto que percorre o array de cláusulas 1 vez.
- *create_matrix*: O((2c + 2x)(c+x)), já que é necessário zerar a matriz, percorrendo-a.
- *create_lits_T_F_column*: O(2c+2x)x, visto que apenas preenche a coluna da matriz para cada literal.
- *create_var_clause_column*: O(2c+2x)c, visto que apenas preenche a coluna da matriz para cada cláusula.
- *everything_is_ok*: O((2c + 2x)(c+x)), já que é necessário percorrer a matriz.

Fazendo a soma das complexidades do algoritmo de redução com seus algoritmos auxiliares, temos a complexidade $O(6c^2 + 6x^2 + 12cx + c)$. Nesse caso, conforme já dito anteriormente, *x* indica o número de literais e *c* o número de cláusulas da equação booleana. Com isso, a complexidade final do algoritmo de redução é $O(c^2 + x^2)$.

Assim, temos que o problema 3-SAT é reduzido ao problema da Soma dos Subconjuntos (3-SAT $\leq$p SUBSET-SUM) em tempo polinomial.

CONSIDERAÇÕES FINAIS

Diante do exposto, foi possível provar que o problema da Soma dos Subconjuntos (SUBSET SUM) é NP-Completo pois:

- Primeiramente, já que o problema 3-SAT é NP-Completo, conseguimos provar que o problema da Soma dos Subconjuntos é NP-Difícil, fazendo sua redução;

- Foi possível provar também que ele é NP pois obtivemos um resultado válido após entrar com um certificado e receber uma resposta positiva. Esse algoritmo de verificação possui sua resposta em tempo polinomial.

Assim, se temos que o problema da Soma dos Subconjuntos é tanto NP-Difícil quanto NP, sabemos que com essas características, ele faz parte dos problemas NP-Completos.

REFERÊNCIAS:

[1] http://pt.wikipedia.org/wiki/A_quest%C3%A3o_P_versus_NP

[2] http://whiletrue.com.br/4fase/TEC/Trabalho%20TEC%20-%20Artigo%20sobre%20Subset-sum.pdf

[3] http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-subsetsum.html

[4] http://pt.wikipedia.org/wiki/NP-completo

[5] http://en.wikipedia.org/wiki/Subset_sum_problem

[6] http://www.comp.ita.br/~mamc/slides/6.pdf

[7] http://forum.bgu.co.il/index.php?app=core&module=attach&section=attach&attach_id=75527

[8] http://www.inf.ufrgs.br/~cgdaudt/inf05515/art3.pdf

[9] http://www.cs.toronto.edu/~pmccabe/csc363-2005S/notes17.pdf

[10] CORMEN, THOMAS H. et al. **ALGORITMOS - Teoria e Prática**. Tradução da segunda edição [americana] Vandenberg D. de Souza. Rio de Janeiro:Campus 2002.