Problema NP-Completo: Coloração de Grafos

Ana Cláudia de Almeida Bordignon
Daniela Mendonça Cavalheiro
Instituto de Informática
Universidade Federal do Rio Grande do Sul
06/06/2012

Resumo

A proposta deste artigo é apresentar uma prova matemática da NP-completude do problema de coloração de grafos. Para isto será utilizada a redução do problema 3-SAT, sabidamente NP-completo, ao 3-coloração e a prova de que o problema em questão pode ser verificado em tempo polinomial.

1 Introdução

1. Definição do problema e aplicação

O problema de coloração de grafos consiste em determinar cores para os vértices de um grafo de forma que vértices adjacentes não recebam cores iguais. Ele pode ser tratado de um ponto de vista de problema de decisão, sendo então definido pela pergunta "é possível colorir o grafo com no máximo k cores, de forma a impedir que vértices adjacentes possuam cores iguais?", ou então do ponto de vista de problema de otimização, definido por "qual é o menor valor de k que permite a coloração do grafo, sem que vértices adjacentes recebam cores iguais?". Nós trataremos do problema do ponto de vista de decisão.

Podemos defini-lo formalmente como: dado um grafo G=(V, A) sem loops, bidirecional e não-valorado, onde V é o conjunto de vértices e A é o conjunto de arestas, e um conjunto C de cores, uma coloração de G é a atribuição de uma cn pertencente a C para cada vn pertencente a G, tal que vértices adjacentes não recebam uma mesma cor.

Uma k-coloração de G é uma coloração com conjunto C de k elementos.

Dentre as aplicações do problema de coloração, pode-se citar a coloração de mapas e o jogo Sudoku.

2 Prova

A prova da NP-completude do problema é feita em duas partes. Primeiro prova-se que o problema pode ser verificado em tempo polinomial - característica básica dos problemas da classe NP - e posteriormente prova-se que é possível reduzir o problema 3-SAT, sabidamente NP-completo, ao problema que estamos tratando, em tempo polinomial.

2.1 Verificação

- Instâncias:
    - V : número de vértices
    - A : matriz de adjacência
    - K : número de cores limite
- Certificado:
    - C: é um vetor de cores de tamanho V (número de vértices) que representa uma coloração;
- Saída:
    - 0 : C não é uma solução de para o Grafo G = (V,A)
    - 1 : C é uma solução para o Grafo G = (V,A)

Algoritmo:

```
VerificaColoracao(V,A, K, C)
{
        // Array que vai ter a relação de cores
        DiferentesCores[K];
        c = 0;
        // percorre o vetor de coloração para pegar a qtd de cores distintas
         for(i = 0; i < V; i++)
        {
                // função not_in(x, array) tem complexidade O(c)
                if( not_in(C[i], DiferentesCores))
                {
                        DiferentesCores[c] = C[i];
                        c++;
                }
        }
        // vai verificar se se c < = k
        if(c > k)
        {
                return 0;
        }

        // vai verificar se a cores de cada nodo são diferentes dos seus vizinhos
         for(i = 0; i < V; i++)
        {
                // vai verificar somente adjacência acima da diagonal principal
                for(j = i + 1; j < V; j++)
                {
                        if(Aij)
                        {
                                // compara cores
                                if(C[i] == C[j])
                                {
                                        return 0;
                                }
                        }
                }
        }
        // se para todas as adjacências as cores eram diferentes, a coloração é válida
        return 1;
}
```

2.1.2 Complexidade Verificação

Para o primeiro “for” do nosso algoritmo, temos a verificação do número de cores usado no certificado C, que tem complexidade O(Vc), pois para cada vértice, é verificado se a cor respectiva não existe no vetor DiferentesCores de tamanho c. A complexidade fica $O(V^2)$ pois c no máximo é igual a V.

No segundo “for”, é verificado na matriz de adjacência A se os vértices vizinhos tem cores diferentes. Para isso, percorremos a matriz A somente acima da diagonal principal para não validarmos 2 vezes o mesmo par de nodos. Com isso, temos que:

- No for de dentro, j varia de i+ 1 a V - 1, que é o mesmo que j variar de 1 a V-1 – i;
- No for de fora, i varia de 0 a V-1;

Assim a complexidade fica igual a:

$$\sum_{i=0}^{V-1}\left(\sum_{j=1}^{V-1-i}(1)\right)$$
$$= \sum_{i=0}^{V-1}(V-1-i)$$
$$= \sum_{i=0}^{V-1}(V) - \sum_{i=0}^{V-1}(1) - \sum_{i=0}^{V-1}(i)$$
$$= V(V-1) - (V-1) - (V(V+1))/2$$
$$= (V-1)^2 - (V^2+V)/2$$
$$= V^2 - 2V + 1 - V^2/2 + V/2$$
$$= V^2/2 - 3V/2 + 1$$
$$= O(V^2)$$

Para todo o algoritmo, temos a complexidade $O(V^2) + O(V^2) = O(V^2)$, que é polinomial, logo a coloração de grafos pertence a NP.

2.2 Redução: 3-SAT para 3-COLOR

2.2.1 Definição SAT

O problema 3-SAT consiste em, dada uma fórmula com n cláusulas, unidas pelo operador lógico " ^ " (AND), de exatamente 3 literais cada, unidos pelo operador lógico " v " (OR), definir valores para as suas variáveis, de forma que o resultado final para a expressão seja verdadeiro.

A NP-completude deste foi provada por Richard Karp, em 1972. Assim sendo, se conseguirmos reduzir, em tempo polinomial, o 3-SAT ao nosso problema, provamos que ele também é NP-completo.

2.2.2 Algoritmo de Redução

A redução do problema 3-SAT para 3-COLOR é feita através de um algoritmo que, a partir das entradas do problema 3-SAT, gera um grafo G que será colorável, com 3 cores distintas, quando o problema 3-SAT for satisfazível.

Para facilitar a elucidação do algoritmo de geração de grafo utilizamos um exemplo e imagens do mesmo. O exemplo se trata da expressão (x \/ !y \/ !z) /\ (x \/ !y \/ z) /\ (!x \/ y \/ z), para a qual tem-se solução válida com os valores x = 1, y = 1 e z = 1. Seus nodos foram coloridos conforme o algoritmo.

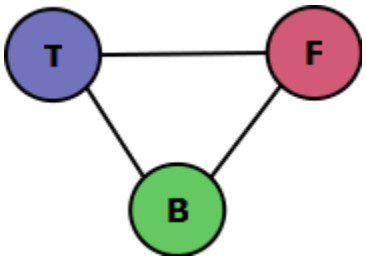
Após, para tornar mais claro o conceito de solução válida e solução inválida, apresentamos uma tentativa de coloração do grafo gerado, mas com valores de x, y e z que não satisfazem a expressão do problema 3-SAT.

É importante deixar claro que a coloração do grafo aqui apresentada *não* faz parte do algoritmo de redução. O resultado é um grafo que representa a expressão do 3-SAT sem coloração, as cores são somente para a melhor

visualização do exemplo.

2.2.2.1 Primeiro Passo: Triângulo True-False-Base (TFB)

Para ter uma base para o grafo ser colorável com 3 cores, a primeira estrutura criada é o triângulo TFB, onde cada nodo (True, False, Base) representaria uma cor.

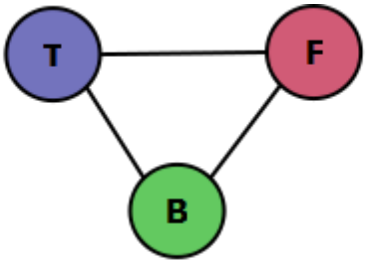


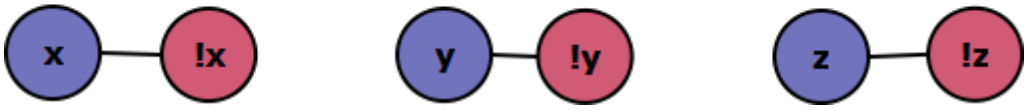
Para nosso exemplo, os nodos dos grafos, correspondentes aos literais, serão pintados conforme a solução do problema. Basicamente, os nodos cujos literais tiverem valor igual a 1, serão pintados da mesma cor que o nodo True “T”; os que tiverem valor igual a 0, serão pintados com a cor do nodo False “F”; e, de acordo com a adjacência dos nodos, outros nodos serão pintados com a cor do nodo Base “B”.

2.2.2.2 Segundo Passo: Ligação Complementares dos Literais

Como já vimos, a entrada do problema 3-SAT é uma expressão que possui K cláusulas com 3 literais cada. Como (x \/ !y \/ !z) /\ (x \/ !y \/ z) /\ (!x \/ y \/ z), que contém 3 cláusulas com 3 literais cada.

A partir dessa expressão, pegamos as variáveis x, y, z e ligamos elas aos seus complementares !x, !y e !z.





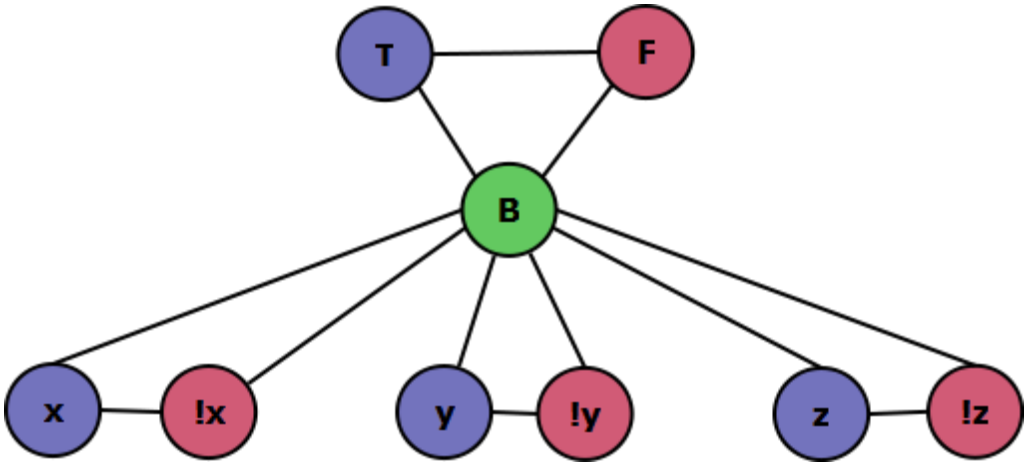
Os nodos são pintados de acordo com a validade das variáveis x, y e z. Como estamos ilustrando um exemplo, x=1, y=1, e z=1, os nodos x, y e z são pintados da mesma cor do nodo “T” e seus complementos da mesma cor do nodo “F”.

Note que nosso exemplo tem somente 3 variáveis. Mas poderíamos ter mais de 3 variáveis na expressão do 3-SAT, e, para todas as n variáveis do problema, iriamos fazer a ligação entre seus respectivos complementos.

2.2.2.3 Terceiro Passo: Ligação complementares com a Base

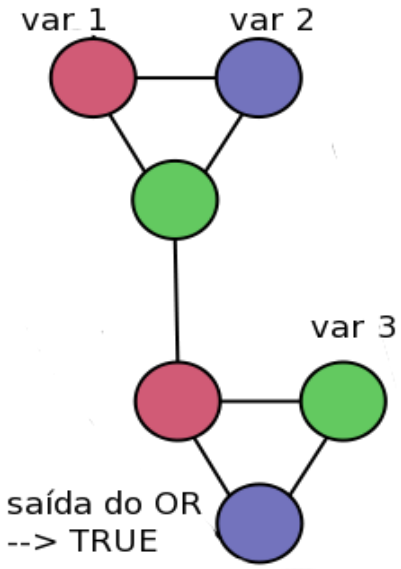
Como terceiro passo, fechamos um triângulo para cada dupla de

complementares com a nodo “B”, fechando as três cores.



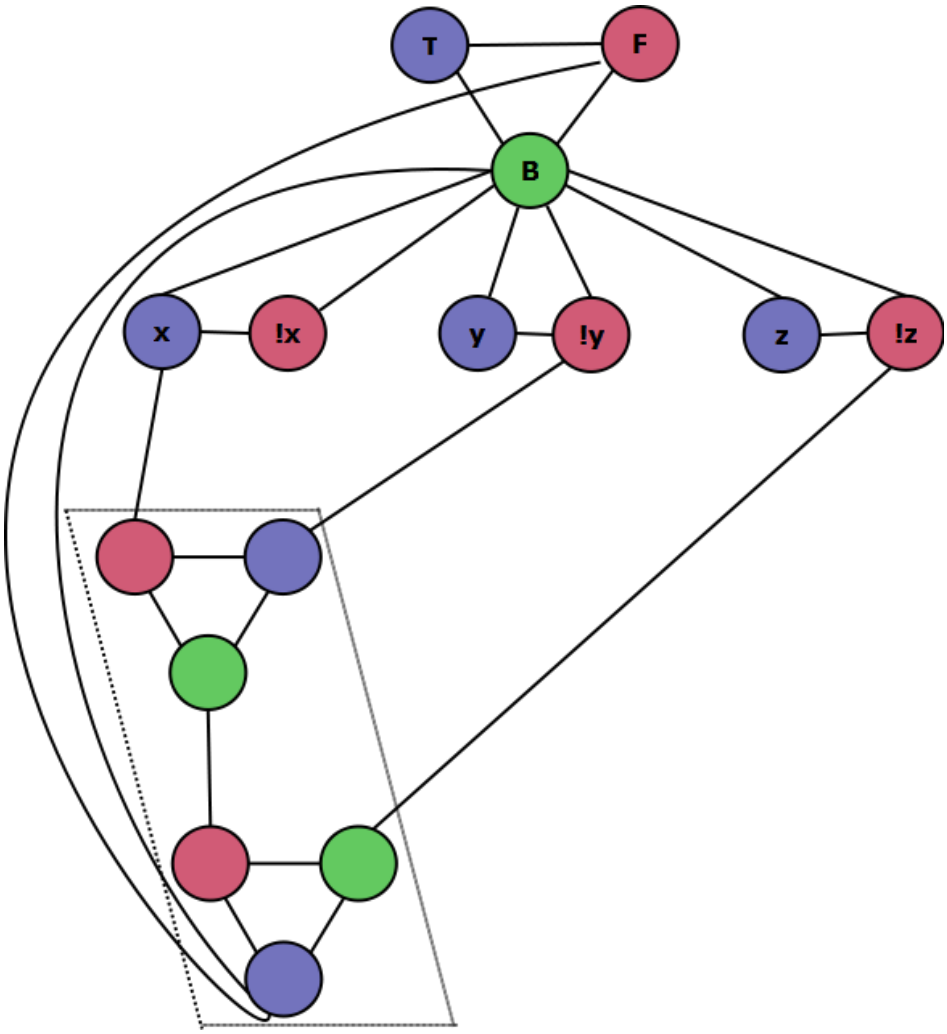
2.2.2.4 Quarto Passo: Cláusulas e estrutura OR

Agora vamos representar cada cláusula da expressão do problema 3-SAT. Para isso é criado uma estrutura, com dois triângulos, que representa as ligações OR (chamaremos neste texto de “estrutura OR”) entre as variáveis de cada cláusula, onde cada variável vai ser ligada a um dos nodos das variáveis e o nodo de saída do OR será ligado aos nodos “F” e “B”.



A cor dos nodos onde as variáveis da cláusula se ligam, dependem da validade do literal que a variável representa. Mas o nodo que representa a saída do OR *deve* ter a mesma cor do nodo T, pois cada cláusula deve ser verdadeira para uma solução válida.

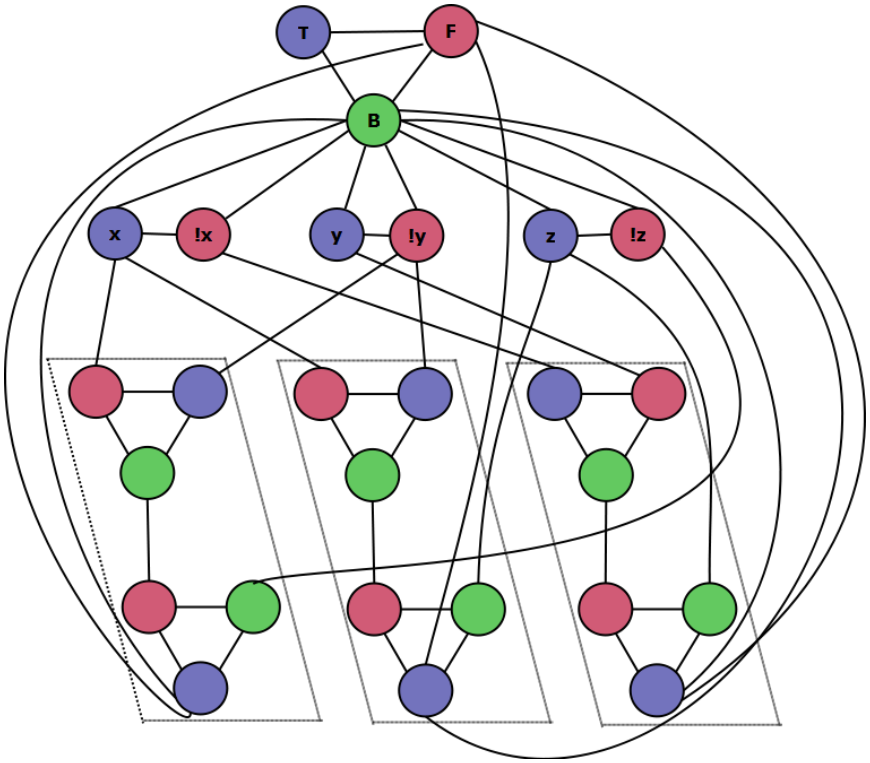
Assim conforme nosso exemplo, montamos a primeira cláusula (x \/ !y \/ !z) no grafo:



O nodo do literal “!z” se ligou ao nodo representado como “var 3” na figura anterior e, como o valor do literal !z = 0, sua cor é a mesma de “F”, assim resta ao nodo “var 3” ser da cor da base, já que ele está ligado ao nodo de saída. Com essa lógica seguimos pintando nosso grafo conforme as adjacências.

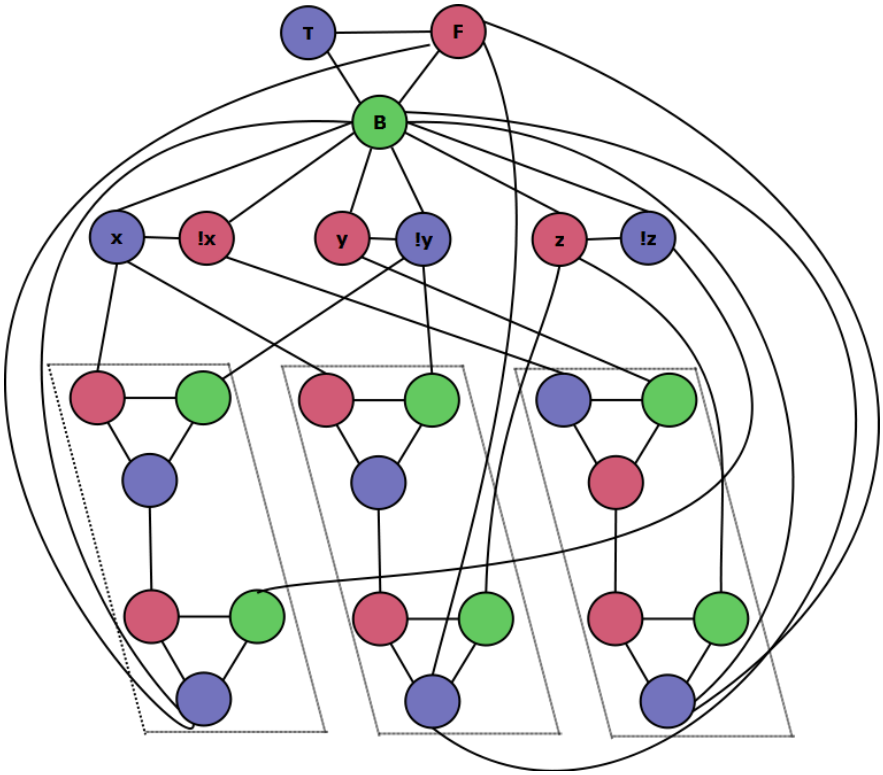
Para terminar o algoritmo de redução, as outras cláusulas devem ser representadas por outras estruturas OR e ligadas ao nodos “B” e “F” como descrito.

Completando nosso exemplo teremos no fim o seguinte grafo:



Como não houve nodos adjacentes com mesma cor, para uma saída valida da expressão do problema 3-SAT, a coloração do grafo é válida para um entrada válida.

E para um entrada inválida da expressão (x \/ !y \/ !z) /\ (x \/ !y \/ z) /\ (!x \/ y \/ z), como x = 1, y = 0 e z = 0, temos o seguinte grafo:



Note que ao tentarmos manter as saídas das estruturas OR válidas (True), tivemos dois nodos vizinhos com a mesma cor.

Segue o algoritmo em pseudo-linguagem:

Entradas:

vetor de Clausulas[k][3]; // k clausulas com 3 literais cada
vetor de Literais[n][2]; // representação Literais[3][1] = 'x3' Literais[3][0] = '¬x3'
--> Sendo:
k = numero de clausulas
n = numero de literais 'próprios' (sem contar complementos!)

Saídas:

vetor G =(V, E)
onde:

- *V[3+2n+6k]:= vetor com todos os vértices do grafo;*
- *E[3+3n+12k][2] := vetor de (3+3n+12k) arestas onde cada aresta possui*

dois nodos;

Programa:

```
// Total de vertices e total de arestas
totV = 3 + 2n + 6k;
totA = 3 + 3n + 12k;
// V é array de vértices de tamanho 3 + 2n + 6*k
V[totV];
// E é a matriz de adjacencia entre os vertices
E[totA][2];

// criacao do triangulo TFB
V[0] = F;
V[1] = T;
V[2] = B;
// contador para vertices
contV = 3;
// arestas do trinagulo
E[0][0] = F; E[1][0] = F;
E[0][1] = T; E[2][0] = T;
E[1][1] = B; E[2][1] = B;
// contador para arestas
contA = 3;

// criacao das ligacoes entre ligadores e seus complementares
for(i = 0; i < n; i++){
        // cria os nodo de um literal 'proprio' e seu complementar
        V[contV] = Literais[i][1];
        contV++;
        V[contV] = Literais[i][0];
        // cria as arestas entre os nodos complementares
        E[contA][0] = V[contV-1];
        E[contA][1] = V[contV];
        contA++;
        // cria as arestas entre cada nodo e o nodo base B, fechando os triangulos
        E[contA][0] = V[contV-1];
        E[contA][1] = B;
        contA++;
        E[contA][0] = V[contV];
        E[contA][1] = B;
        // incrementa contadores
        contA++;
        contV++;
}
```

```
// para cada clausula é criada a estrutura OR e suas ligacoes
for(i = 0; i < k; i++){
        // primeiro triangulo da estrutura
        V[contV] = Ai1;
        contV++;
        V[contV] = Ai2;
        contV++;
        V[contV] = Ai3;
        contV++;
        // liga os nodos do primeiro triangulo
        E[contA][0] = Ai1;
        E[contA][1] = Ai2;
        contA++;
        E[contA][0] = Ai1;
        E[contA][1] = Ai3;
        contA++;
        E[contA][0] = Ai2;
        E[contA][1] = Ai3;
        contA++;
        // liga as primeiras duas literais da clausula aos dois primeiros nodos do primeiro
triangulo
        E[contA][0] = Ai1;
        E[contA][1] = Clausulas[i][0];
        contA++;
        E[contA][0] = Ai2;
        E[contA][1] = Clausulas[i][1];
        contA++;

        // segundo traingulo da estrutura
        V[contV] = Ai4;
        contV++;
        V[contV] = Ai5;
        contV++;
        V[contV] = Ai6;
        contV++;
        // liga o nodo Ai3 do primeiro triangulo com o nodo Ai4 do segundo triagulo
        E[contA][0] = Ai3;
        E[contA][1] = Ai4;
        contA++;
        // liga o nodo Ai5 com o ultimo literal da clausula
        E[contA][0] = Ai5;
        E[contA][1] = Clausulas[i][6];
        contA++;
        // liga o ultimo nodo da estrutura OR, Ai6, com os nodos F e B
        E[contA][0] = Ai6;
        E[contA][1] = F;
        contA++;
        E[contA][0] = Ai6;
        E[contA][1] = B;
        contA++;
}

// Depois de listados os nodos e as arestas, é devolvido o grafo G = (V,E)
return Grafo(V,E);
```

1. Complexidade Redução

A complexidade do algoritmo de redução é dada pelas 15 atribuições iniciais para a criação do triângulo TFB, a iteração de 0 a n-1 de 13 atribuições para a criação dos nodos complementares, a iteração de 0 a k-1 de 39 atribuições para a criação das estrutura OR para cada uma das k cláusulas. Como se segue:

$$15 + (\sum_{i=0}^{n-1} 13) + (\sum_{i=0}^{k-1} 39) + 1$$
$$= 15 + 13n + 39k + 1$$
$$= O(1) + O(n) + O(k) + O(1)$$

Assim comprovamos que a complexidade do algoritmo de redução é polinomial.

3 Referências bibliográficas

Livros:

- F. HARARY, “Graph Theory”, Addison-Wesley, 1969.
- Complexidade de Algoritmos, Laira Vieira Toscani, Paulo A.S. Veloso.

Sites:

- http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/23Reductions.pdf
- http://www.shannarasite.org/kb/kbse60.html
- http://www.csc.liv.ac.uk/~igor/COMP309/3CP.pdf
- http://www.inf.ufrgs.br/~aesouza/Complexidade.pdf
- http://www.dsc.ufcg.edu.br/~abrantes/CursosAnteriores/TG051/coloracao.pdf