UNIVERSIDADE FEDERAL DO MATO GROSSO – CAMPUS ARAGUAIA II
CIÊNCIA DA COMPUTAÇÃO

INDEPENDENT SET PARA VERTEX COVER

BARRA DO GARÇAS
2023

ALDO TEIXEIRA DA SILVA JUNIOR
MARIA LUISE BRITTO
TAINÁ ISABELA MONTEIRO
WILLYAN JOSUÉ BASTOS

INDEPENDENT SET PARA VERTEX COVER

Trabalho de Projeto e Análise de Algoritmos sobre algoritmo do problema Independent Set para Vertex Cover.

Dr. Robson da Silva

BARRA DO GARÇAS
2023

SUMÁRIO

INTRODUÇÃO

O objetivo deste trabalho é demonstrar os problemas Independent Set e Vertex Cover, seus algoritmos e soluções, assim como suas respectivas classes dentre os problemas de decisão e busca, como também, a redução de um ao outro, em suma, mostrar que no fundo ambos são o mesmo problema com uma aparência diferente.

1. DEFINIÇÃO

O Vertex-Cover (conhecido também como Cobertura de Vértices), pode ser exemplificado como um conjunto de salas interligadas por corredores, onde um guarda postado numa sala é capaz de vigiar todos os corredores que convergem sobre a sala, queremos determinar o número de guardas capazes de vigiar todos os corredores. Ele é classificado como NP (classe de complexidade não determinística polinomial) e é representado por um grafo e um orçamento, onde o objetivo é encontrar os vértices que cobrem (toquem) todas as arestas (ou dizer que não existe).

Já o Independent Set(conhecido também como Conjuntos Independentes em Grafos) seria como, onde não pôr os guardas, já que estes estariam longe de todos os outros sem nenhuma conexão. Sendo assim, ele é classificado como NP-Completo e tem como entrada um grafo e um inteiro g, onde o objetivo é encontrar g vértices independentes (que não tem arestas entre eles), ou dizer que não existe.

Em termos simples, o problema do Independent-Set é inversamente complementar ao Vertex-Cover, em que ambos têm como entrada um grafo, e a partir da solução de Vertex-Cover, pode ser retirada uma subsolução, que seria a solução de Independent-Set e vice-versa.

Sob essa perspectiva, foram feitas implementações de algoritmos em c + + a fim de provar cada caso específico dentro da classe de complexidade.

2. DETERMINANDO SE B’(Vertex-Cover) está em NP

Um problema de decisão c é NP-completo se:

- ➢ c está em NP;
- ➢ Todo problema em NP é redutível para c em tempo polinomial.

Tendo isso em vista, deve-se mostrar a partir de uma implementação que verifique em tempo polinomial se, dada uma solução ***S*** para uma instância ***I*** (que no nosso caso seria um grafo), se ela é uma atribuição verdadeira ou não, quando o algoritmo a ser avaliado é o Vertex-Cover.

2.1. Implementação de função *C* que verifica em tempo polinomial se *S* é uma solução para instância *I* do problema *B’*.

```c
int existe(int vet[], int value, int tam)
{
    for (int i = 0; i < tam; i++)
    {
        if (vet[i] == value)
        {
            return 1;
        }
    }

    return 0;
}

int ehVertexCover(GRAFO *g, int S[])
{
    // Percorre toda o grafo verificando se solução passada está correta
    for (int i = 0; i < g->nVertices; i++)
    {
        for (int j = 0; j < g->nVertices; j++)
        {
            if (g->adj[i][j] == 1)
            {
                if (existe(S, i, g->nVertices) == 0 && existe(S, j, g->nVertices) == 0)
                {
                    printf("Está não é uma solução válida para a instância I do problema Vertex-Cover.\n");
                    return 0;
                }
            }
        }
    }
    printf("Está é uma solução válida para a instância I do problema Vertex-Cover.\n");
    return 1;
}
```

Implementação da Função ehVertexCover

A função *ehVertexCover* recebe uma instância *I* do problema Vertex-Cover que possui como entrada:

- ➢ Um grafo (*GRAFO *g*)
- ➢ Uma solução *S* representada por um vetor de inteiros, que contém os vértices selecionados como solução proposta pelo usuário.

Ela verifica se a solução *S* passada é um Vertex Cover válido para o grafo g, de maneira que percorre todos os vértices do grafo *g* usando dois loops aninhados.

Para cada par de vértices *(i, j)* no grafo, verifica se há uma aresta entre eles *(g->adj[i][j] == 1)*. Dentro do segundo loop, há um conjunto de condicionais para verificar se o par de vértices *(i, j)* não está coberto pelo conjunto *S*.

- ➢ O primeiro if verifica se o vértice *i* não está presente em *S* (ou seja, existe(*S, i, g->nVertices) == 0*).
- ➢ O segundo if verifica se o vértice *j* não está presente em *S* (ou seja, existe(*S, j, g->nVertices) == 0*).

Se ambos os vértices i e j não estão presentes em S, isso significa que a aresta entre eles não está coberta pelo Vertex Cover S, e uma mensagem de erro é impressa indicando que a solução S não é válida para a instância I do problema Vertex-Cover. A função retorna 0 para indicar que a solução não é válida.

A função *existe()* recebe:

- ➢ Um array vet
- ➢ Um valor value
- ➢ O tamanho tam do array

A função percorre todo o array e verifica se o valor *value* está presente nele, se encontrar o valor, a função retorna 1, indicando que ele existe no array, caso contrário, ao percorrer todo o array sem encontrar o valor, a função retorna 0, indicando que o valor não existe no array.

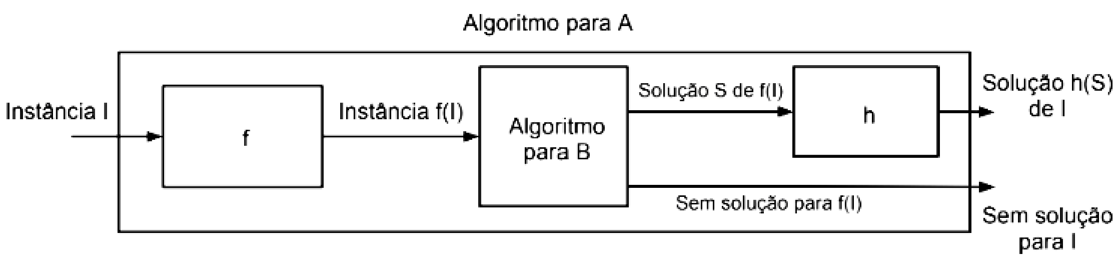
Sua complexidade é depende dos dois “for” aninhados que percorrem todos os vértices do grafo g. O for externo é controlado pela variável i, que itera de 0 até o número de vértices n de g e o interno é controlado pela variável j, que também itera de 0 até o número de vértices n de g.

Dentro do loop interno, há uma verificação condicional que verifica se existe uma aresta entre o vértice i e o vértice j. Essa verificação é feita acessando a matriz de adjacência de g, ou seja, g->adj[i][j].

A complexidade dessa verificação é constante, pois o acesso à matriz de adjacência é uma operação de tempo constante, independentemente do tamanho do grafo.

Portanto, a complexidade da função *ehVertexCover()* é O(n^2), onde n é o número de vértices do grafo g. Isso ocorre devido aos dois loops aninhados que percorrem todos os vértices do grafo.

3. DETERMINANDO SE NP É REDUTIVEL PARA C EM TEMPO POLINOMIAL



Modelo de Redução de Algoritmo A (Independent Set) para Algoritmo B (Vertex-Cover)

Uma redução de um problema de busca ***A*** para um problema de busca ***B*** são dois algoritmos de tempo polinomial ***f*** e ***h*** que:

- ***f*** transforma qualquer instância ***I*** de ***A*** em uma instância ***f(I)*** de ***B***,
- ***h*** transforma qualquer solução ***S*** de ***f(I)*** de volta em uma solução ***h(S)*** de ***I***,

Se ***f(I)*** não tem solução, então ***I*** não tem também.

Tendo isto em vista, pode-se definir as classes, as reduções e as complexidades dos problemas que estão sendo analisados neste trabalho. Supondo que o problema ***A'*** (Independent Set) é NP-Completo, vamos verificar se ***A'*** é redutível para ***B'*** (Vertex–Cover) em tempo polinomial.

Em suma, deve-se mostrar que dada uma instância ***I*** do Independent Set, para solucioná-lo, basta procurar um Vertex Cover de ***I*** com ***g*** vértices, e que os vértices que não estão conectados por arestas, há ***g*** vértices que não são do Vertex-Cover portanto formam um Independent Set.

3.1. Implementação de função que recebe uma instância *I* para o problema *A’* *(Independent Set)* utilizando o algoritmo B

```c
1  void solucionaA(GRAFO *g) {
2      int cont=0;
3      int *SA;
4      int *SBdeA = soluçãoOtimaVertexCover(g, g->nVertices);
5
6      while (SBdeA[cont] != 0) cont++;
7
8
9      if (SBdeA[0] != 0) {
10         SA = converteVertexParaIndempendt(g, SBdeA, cont);
11         printf("\nS de A' = ");
12         for (int i = 0; i < g->nVertices-cont-1; i++)
13         printf(" %d ", SA[i]);
14         printf("\n\n");
15     } else {
16         printf("Nao existe solucao.\n");
17     }
18 }
```

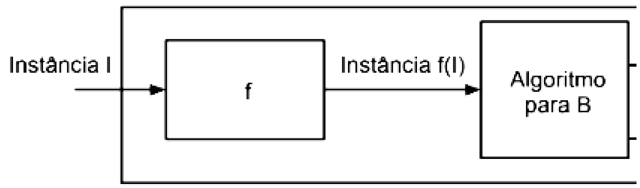
Implementação de instância I para o Independent Set

A função *solucionaA()* recebe um grafo g como entrada e tem como objetivo encontrar as soluções ótimas para o problema Vertex Cover para os conjuntos S' de A' e B' do grafo. Ela encontra as soluções ótimas para os conjuntos S' de A' e B' do problema Vertex Cover em um grafo g.

Primeiro, declara as variáveis cont (contador de número de elementos em *SBdeA*), *SA* (um array para armazenar *S’* de *A’*), depois a função entra no while (linha 6) que conta o número de elementos em *SBdeA* percorrendo até encontrar o valor 0, indicando o fim dos elementos válidos, em seguida verifica na condicional if(linha 9) o primeiro elemento de *SBdeA* que é diferente de 0, se sim, significa que é uma solução válida também e há novamente outra verificação com a chamada de função *converteVertexParaIndenpendent()*, que percorre os vértices do grafo e verifica quais vértices não estão presentes em SBdeA (conjunto S de B') e os adiciona em SA (conjunto S de A').

Finalmente, a função imprime os conjuntos *SA* e *SBdeA* na saída padrão. Se SBdeA[0] for igual a 0, significa que não existe solução válida para o problema Vertex Cover, e uma mensagem informando isso é impressa.

3.2. Implementação que faz redução da instância I do Independent Set para a instância f(I) do Vertex-Cover



Redução da Instância *I* para Instância *f(I)*

Seguindo o modelo acima o que precisaria ser feito basicamente é uma redução para que a instância do problema A fosse a transformada em uma instância do problema B.

Para o caso Independent Set → Vertex-Cover, percebe-se que há uma igualdade entre a instância *I* para o problema do Vertex-Cover e a instância *I* do Independent-Set, ambas possuem o mesmo tipo de instância, um grafo. Logo, na implementação do algoritmo que o grupo realizou não é necessário uma função f que faça a redução de uma instância para outra.

O caso em que poderia ser utilizado, é quando há realmente diferentes tipos de entrada em cada problema, por exemplo, a redução de uma instância CNF (Fórmula Booleana em Forma Normal Conjuntiva) para um problema que precisa de uma instância em grafo, como é feito em 3SAT → Independent Set.

3.3. Implementação de função *B* que recebe a instância *f(I)* do problema Vertex-Cover, com isso, encontrando a solução ótima por meio de um algoritmo determinístico para ele e retornar a resposta *S* para o problema Vertex-Cover;

```c
int *soluçãoOtimaVertexCover(GRAFO *g, int limit)
{
    GRAFO *aux = g;
    int *S = (int *)malloc(limit * sizeof(int));
    int maiorG = 0;
    int grauAtual;
    int K = 0;

    // Zera todas posições de S
    for (int i = 0; i < limit; i++)
        S[i] = 0;

    while (aux->nArestas > 0)
    {
        // encontra o vértice de maior grau
        for (int i = 0, max = 0; i < aux->nVertices; i++)
        {
            grauAtual = grauV(g, i);
            if (grauAtual > max)
            {
                maiorG = i;
                max = grauAtual;
            }
        }
        // adiciona o vértice de maior grau em S
        S[K++] = maiorG;
        // remove o maior vértice do grafo
        for (int i = 0; i < aux->nVertices; i++)
        {
            if (aux->adj[maiorG][i] == 1)
                removeAresta(aux, maiorG, i);
        }
    }

    printf("\nS de B' = ");
    for (int i = 0; i < K; i++)
        printf(" %d ", S[i]);
    printf("\n\n");

    return S;
}
```

Implementação de Algoritmo B Determinístico

A função *soluçãoOtimaVertexCover é um algoritmo determinístico que recebe como parâmetros:

- ➢ Um grafo (*GRAFO *g*)
- ➢ Um limite (*int limit*)

E retorna um ponteiro para um array de inteiros que seriam os vértices da solução ótima do Vertex-Cover.

A função tem como objetivo encontrar uma solução ótima para o problema do "Vertex Cover" no grafo fornecido, onde um "Vertex Cover" é um conjunto de vértices que abrange todas as arestas do grafo. O limite especifica o tamanho máximo do conjunto de vértices que a função pode retornar. Ela começa criando uma cópia do grafo e aloca um array de inteiros chamado S que se inicializa com zero em todas as posições, assim que entra no primeiro “for”, percorre todos os vértices do grafo e encontra o vértice de maior grau, este é armazenado em *maiorG* e em seguida é adicionada ao conjunto S da solução, em seguida no segundo “for”, esse vértice de grau maior é removido do grafo auxiliar assim como todas as arestas dele e assim o código continua pegando sempre o vértice de maior grau e o colocando na solução até não ter mais arestas.

Analisando o while, ele percorre todas as arestas do grafo auxiliar, portanto sua complexidade é O(E), onde E é a quantidade de arestas, em seguida temos o for que percorre todos os vértices do grafo auxiliar, ou seja, executa V iterações e dentro dele há uma chamada à função *"grauV"* para cada vértice, o que adiciona uma complexidade adicional de O(V). Além disso, a remoção de arestas no grafo auxiliar também pode ter uma complexidade de até O(V) em casos extremos, quando o grafo é uma árvore linear.

No geral, a complexidade do algoritmo é dominada pelo segundo loop (linha), resultando em uma complexidade de *O(V^2).* Em termos de espaço, a função aloca um array de tamanho limit para armazenar o conjunto de vértices, portanto sua complexidade de espaço é de *O(limit).*

3.4. Implementação de função *h* que recebe a resposta *S* do problema *B′* e converte para resposta do problema *A′*.

```c
int *converteVertexParaIndempendt(GRAFO *g, int *S, int K){
    int tamSA = (g->nVertices-1)-K;
    int *SA = (int*) malloc((tamSA)*sizeof(int));
    int aceito = TRUE;

    // verifica
    for (int i = 1; i < g->nVertices; i++){
        aceito = TRUE;
        for (int j = 0; j < K; j++) {
            if (i == S[j])
                aceito = FALSE;
        }
        if (aceito)
            SA[--tamSA] = i;
    }

    return SA;
}
```

A função *converteVertexParaIndempendent()* recebe um grafo *g*, um conjunto *S* e um inteiro *K* como parâmetros, tem como objetivo converter o conjunto *S* (que é uma solução ótima para o problema Vertex Cover) em um novo conjunto *SA*, que representa os vértices independentes de *S* no grafo *g*.

Ela percorre os vértices do grafo g e verifica se cada vértice está presente no conjunto S:

Se um vértice não estiver presente em *S*, ele é considerado independente e é adicionado ao conjunto SA, então, o tamanho do conjunto *SA* é calculado com base no número de vértices em g e em K. Em seguida, a memória necessária é alocada para a *SA*, E utilizando dois "for" aninhados, a função verifica se cada vértice é igual a algum elemento de *S*. Se não for igual, o vértice é considerado independente e é adicionado ao conjunto SA.

No final, a função retorna o conjunto SA contendo os vértices independentes de S no grafo g.

A complexidade da função converteVertexParaIndependent é O(n * m), onde n é o número de vértices do grafo g e m é o número de elementos em S. A função possui um for externo que percorre os vértices do grafo g de complexidade O(n), dentro dele há outro interno que percorre os elementos do conjunto S de complexidade O(m) e há também uma verificação simples de igualdade entre os vértices, que tem uma complexidade de tempo constante, O(1).

Portanto,a complexidade total da função é O(n * m), indicando que o tempo de execução da função cresce de forma proporcional ao produto do número de vértices n e do número de elementos em S m.

4. CONCLUSÃO

Conclui-se que a redução Independent Set → Vertex-Cover é um NP-Completo, pois

- c está em NP;
- Todo problema em NP é redutível para c em tempo polinomial.

Em termos simples, os problemas acima são os mesmos com algumas diferenças apenas.

5. BIBLIOGRAFIA

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/independent.html
https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/v-cover.html
CORMEN, Thomas H, Algoritmos – Teoria e Pratica, 3ed