SOLUÇÃO SEGUNDA LISTA

Norma Angélica María Tovar Uribe

Matrícula: 2018029559

1. [QUESTÃO – 01]

Especifique cada problema e calcule o M.C. (melhor caso), P.C. (pior caso), C.M. (caso médio) e a ordem de complexidade para algoritmos (os melhores existentes e versão recursiva e não-recursiva) para problemas abaixo. Procure ainda, pelo L.I. (Limite Inferior) de tais problemas:

(A) N-ésimo número da sequência de Fibonacci

É uma sucessão de números descrita pelo italiano Leonardo Fibonacci, ela é infinita e começa com 0 e 1. Os números seguintes são sempre a soma dos dois números anteriores. Portanto, depois de 0 e 1, vêm 1, 2, 3, 5, 8, 13, 21, 34…

Em termos matemáticos, a sequência é definida recursivamente pela fórmula abaixo, sendo os primeiros termo $F_0 = 0$ e $F_1 = 1$:

$$F_n = F_{n-1} + F_{n-2},$$

O problema do n-ésimo número da Sequência de Fibonacci está dado pela seguinte expressão:

$$F_n \begin{cases} 0 & \text{si } n = 0\ ; \\ 1 & \text{si } n = 1\ ; \\ F_{n-1} + F_{n-2} & \text{si } n \geq 2\ . \end{cases}$$

O trecho do código da função recursiva é:

```
int fibonacci(int n){
    if((n == 1) || (n == 0)) {
        return n;
    }
    if(n == 0){
        return 0;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

A complexidade do algoritmo está dada por $O(\phi^n)$ onde : ϕ é igual a $\frac{1+\sqrt{5}}{2}$. Com base a isso o melhor caso e o pior caso equivalem a 2^n.

O trecho do código da função interativa é:

```
int fibonacci (int n){
    int f, f1, f2;
    if((n == 1) || (n == 0)) {
        return n;
    }
    else{
        f1 = 0; f2 = 1;
        for(int i = 2, i <= n, i++ ){
            f = f1 + f2;
            f1 = f2;
            f2 = f;
        }
        return f;
    }
    return 0;
}
```

A complexidade do algoritmo está dada por $O(n)$. Com base a isso o melhor caso e o pior caso equivalem a n.

(B) Geração de todas as permutações de um número

As permutações de um número é incluir listas ordenadas sem repetição a partir dos dígitos do mesmo número. Exemplo: Se tiver o número 123, então:

```
Exemplo de 6 permutações de {1, 2, 3}:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

A complexidade para encontrar todas as permutações de um numero é n!

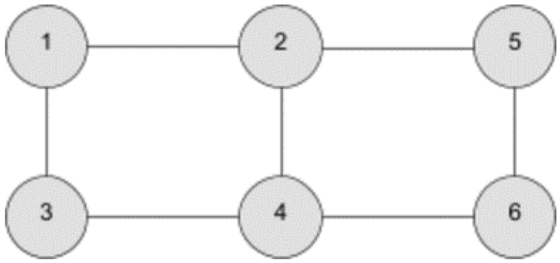
2. [QUESTÃO – 02]

Defina e dê exemplos:

(A) Grafos:

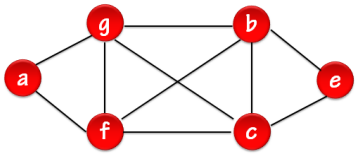
Um grafo é um par (V, A) em que V é um conjunto arbitrário e A é um subconjunto de V (2). Os elementos de V são chamados vértices e os de A são chamados arestas. Neste texto, vamos nos

restringir a grafos em que o conjunto de vértices é finito. Basicamente podemos definir como um conjunto de vértices (V) que são interligados por arestas (A).

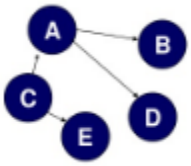


(B) Grafo conexo, acíclico e direcionado:

O grafo **conexo** é quando existe pelo menos um caminho entre qualquer vértice para todos os outros vértices. O grafo **acíclico** é o tipo de grafo em que os caminhos entre os vértices não contem vértices repetidos. E o grafo **direcionado** é um tipo de grafo onde as arestas possuem sentido, ou seja, não é permitido percorrer pelas arestas no sentido contrário.



Grado conexo

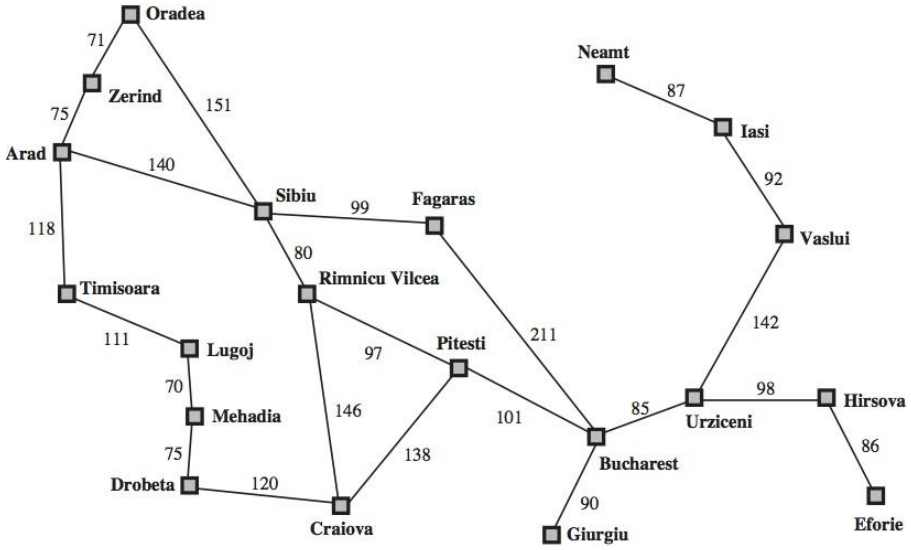


Grado acíclico e direcionado

(C) Adjacência x Vizinhança em grafos:

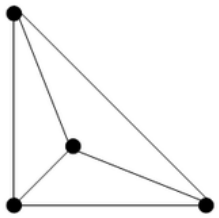
É o sub grafo formado por todos os vértices ligados diretamente a um vértice do grafo.

Um exemplo para vizinhança em grafos é um mapa rodoviário, onde os vértices são cidade e as arestas são as estradas, considerando uma cidade A as cidades vizinhas serão as cidades onde exista uma estrada ligada diretamente a A.



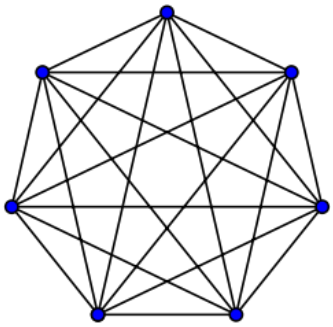
(D) Grafo planar:

Um grafo é planar se puder ser desenhado no plano sem que haja arestas se cruzando.

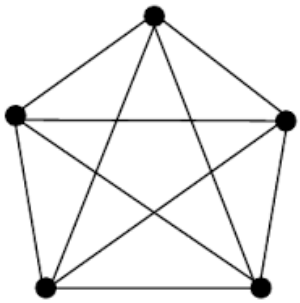


(E) Grafo completo, clique e grafo bipartido:

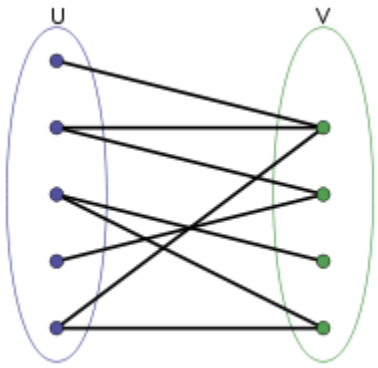
O grafo **completo** é um grafo simples em que cada par de vértices é conectado por uma aresta. **Clique** é um conjunto de vértices V tal que, para cada par de vértices de V, existe uma aresta que os conecta, em outras palavras, um clique é um sub grafo no qual cada vértice é conectado a cada outro vértice do sub grafo, ou seja, todos os vértices do sub grafo são adjacentes. E grafo **bipartido** é um tipo de grafo G = (N, E) onde seus vértices se podem separar em dois conjuntos disjuntos U e V.



Grafo completo



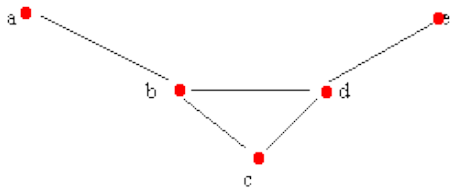
Sub grafo do grafo completo anterior onde os vértices formam um clique de tamanho 5.



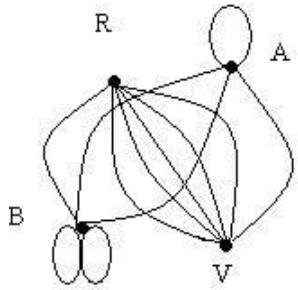
Grado bipartido

(F) Grafos simples x multigrafo x dígrafo:

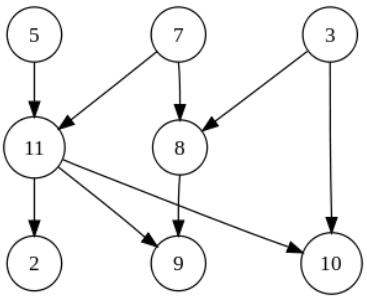
O **grafo simples** é um grafo que não contem nem laços nem arestas múltiplas. O **multigrafo** é um tipo de grafo onde os vértices possuem arestas múltiplas. E o **dígrafo** é um tipo de grafo direcionado e simples.



Grado simples



Multigrafo



Dígrafo

3. [QUESTÃO – 03]

Defina e apresente exemplos de matriz de incidência, matriz de adjacência e lista de adjacência. Adicionalmente, descreva o impacto (vantagens e desvantagens) da utilização de matriz de adjacência e lista de adjacência.

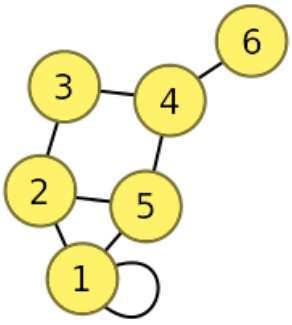
Matriz de incidência

Uma matriz de incidência representa computacionalmente um grafo através de uma matriz bidimensional, onde uma das dimensões são vértices e a outra dimensão são arestas.

Dado um grafo G com *n* vértices e m arestas, podemos representá-lo em uma matriz *n x m* M. A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se deseja representar, porém de forma geral guarda informações sobre como os vértices se relacionam com cada aresta.

Para representar um grafo sem pesos nas arestas e não direcionado, basta que as entradas da matriz M contenham 1 se a aresta incide no vértice, 2, caso seja um laço (incide duas vezes) e 0, caso a aresta não incida no vértice. Isto é, por exemplo, a matriz de incidência do grafo é representada na seguinte imagem.

$$\begin{bmatrix} 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$



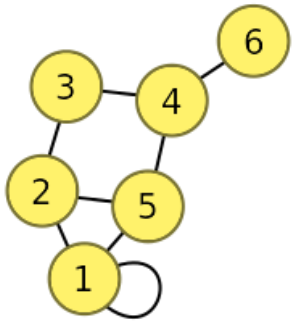
Matriz de adjacência

Uma matriz de adjacência é uma das formas de se representar um grafo.

Dado um grafo G com n vértices, podemos representá-lo em uma matriz *n x n* A(G)=[Aij] (ou simplesmente A). A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se deseja representar, porém de forma geral o valor Aij guarda informações sobre como os vértices Vi e Vj estão relacionados (isto é, informações sobre a adjacência de Vi e Vj).

Para representar um grafo não direcionado, simples e sem pesos nas arestas, basta que as entradas Aij da matriz A contenham 1 se Vi e Vj são adjacentes e 0, caso contrário. Se as arestas do grafo tiverem pesos, Aij pode conter, ao invés de 1 quando houver uma aresta entre Vi e Vj, o peso dessa mesma aresta. Exemplo:

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$



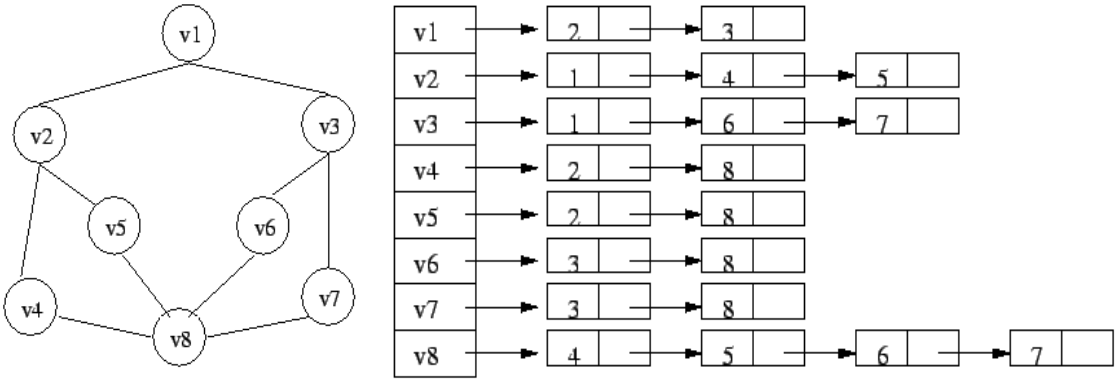
Lista de adjacência

Em teoria dos grafos, uma lista de adjacência, estrutura de adjacência ou dicionário é a representação de todas arestas ou arcos de um grafo em uma lista.

Se o grafo é não direcionado, cada entrada é um conjunto (ou multiconjunto) de dois nós contendo as duas extremidades da aresta correspondente; se ele for dirigido, cada entrada é uma tupla de dois nós, um indicando o nó de origem e o outro denotando o nó destino do arco correspondente. Normalmente, as listas de adjacência são desordenadas.

Nesta representação as *n* linhas da matriz de adjacência são representadas como n listas encadeadas. Existe uma lista para cada vértice em G. Os nós na lista i representam os vértices que são adjacentes ao vértice i.

Cada nó na lista possui pelo menos dois campos: VÉRTICE, que armazena o índice do vértice que é adjacente i e NEXT, que é um ponteiro para o próximo nó adjacente. As cabeças das listas podem ser armazenas em um arregro de ponteiros para facilitar o acesso aos vértices. É conveniente armazenarmos em cada entrada do arregro de cabeças de listas um campo FLAG que será utilizado posteriormente para indicar se um dado vértice possui alguma propriedade, por exemplo em buscas, este FLAG pode indicar se um dado vértice já foi visitado ou não. Exemplo:



Entre as vantagens e desvantagens da utilização de matriz de adjacência e lista de adjacência estão:

- Na matriz de adjacência a velocidade de leitura e escritura é constante.
- A lista de adjacência ocupa menos espaço na memória que a matriz de adjacência.
- Quando a quantidade de vértices é igual ao número de arestas, utilizar a matriz de adjacência é mais viável.
- Quando o número de arestas á bem inferior ao número de vértices pode ser melhor utilizar lista de adjacência.

4. [QUESTÃO – 04]

Comente sobre tabelas hash, apresentando a complexidade para as operações realizadas. Adicionalmente, implemente uma tabela hash com encadeamento separado usando: lista encadeada e árvore vermelho e preto. Apresente um estudo empírico para obter custos de inserção na medida em que o número de chaves aumenta. Gere gráficos para mostrar o custo de inserção para tamanhos distintos de N (exemplo: de 10 a 10000). Descreva uma análise de comparação em relação ao tempo de execução.

Uma tabela hash é uma estrutura de endereçamento que tem como finalidade a redução do espaço de armazenamento e o rápido acesso aos dados a partir de um espaço enorme de chaves. O mapeamento das chaves e feito através de uma função hash que gera a localização do item a partir de seu próprio valor. O tempo para pesquisar um elemento na tablela hash é *O(1).*

Porém, um nos grandes problemas da tabela hash são as colisões. Uma colisão ocorre quando duas ou mais chaves tem o mesmo valor hash. Para solucionar esse problema se pode utilizar técnicas de encadeamento aberto e encadeamento fechado. No caso de encadeamento fechado, se implementa uma subestrutura em cada posição da tabela hash para salvar as chaves com o mesmo valor hash ali, essas estruturas podem ser listas encadeadas, arvores ALV, arvores vermelho – preto e assim por diante.

- No pior caso a complexidade no acesso (busca, inserção e remoção) com hash por encadeamento será *O(n).*

- No caso das arvores balanceadas, as operações de busca, inserção e remoção levariam o temo *O(lg n)*.

5. [QUESTÃO – 05]

Defina, explicando as principais características e exemplifique:

(A) Enumeração explícita x implícita

Os dois métodos fazem uma enumeração das soluções para um determinado problema, porém, o método de enumeração explicita faz uma enumeração de todas as possíveis soluções, e a enumeração implícita faz uma enumeração "inteligente" com as melhores soluções para o problema.

Um exemplo de enumeração explicita x implícita. Tendo um baralho espalhado na mesa, o jogador já tem conhecimento do valor de cada carta, não irá precisar virar carta alguma para saber seu valor, já na implícita, o jogador teria que virar carta por carta para descobrir seu valor.

(B) Programação Dinâmica

É basicamente um esquema de enumeração de soluções que visa, através de uma abordagem de divisão e conquista, minimizar o montante de computação a ser feito. A programação dinâmica e aplicada quando um mesmo subproblema aparece diversas vezes ao longo do processo. Ela resolve o subproblema uma vez só e reutiliza a solução toda vez que o mesmo aparecer novamente.

- Transforma um problema de otimização complexo em uma sequência de subproblemas simples.
- Funciona de trás para frente (problemas menores a maiores)
- Arma relações de recorrência.
- Utiliza-se uma tabela auxiliar que contém uma entrada para cada subproblema distinto.

Programação dinâmica é muito utilizada em problemas de divisão e conquista que tendem a ter um número de subproblemas exponenciais, mas esses subproblemas se repetem frequentemente, assim a tabela irá reduzir bastante o tempo de execução. Um desses exemplos é a sequência de Fibonacci:

Método usando divisão e conquista

```
int fibonacci(int n){
    if((n == 1) || (n == 0)) {
        return n;
    }
    if(n == 0){
        return 0;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

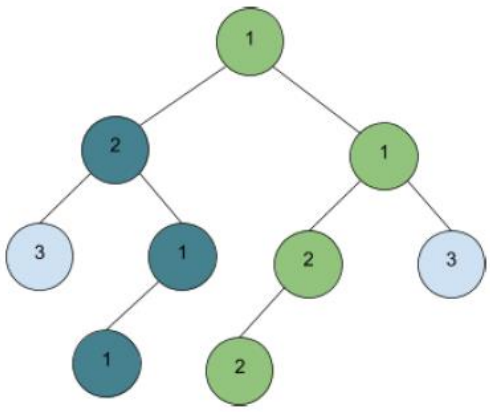
Método usando programação dinâmica

```
int fibonacci (int n){
    int f, f1, f2;
    if((n == 1) || (n == 0)) {
        return n;
    }
    else{
        f1 = 0; f2 = 1;
        for(int i = 2, i <= n, i++ ){
            f = f1 + f2;
            f1 = f2;
            f2 = f;
        }
        return f;
    }
    return 0;
}
```

(C) Algoritmo Guloso

É técnica de projeto de algoritmos que tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global. Uma vez que ele escolhe o próximo passo, ele não volta atrás.

- Jamais se arrepende de uma decisão, as escolhas realizadas são definitivas;
- Não leva em consideração as consequências de suas decisões;
- Podem fazer cálculos repetitivos;
- Nem sempre produz a melhor solução (depende da quantidade de informação fornecida);
- Quanto mais informações, maior a chance de produzir uma solução melhor.



O caminho verde é o caminho que o algoritmo escolhe. Perceba que o melhor caminho é o caminho azul, mas como ele escolhe a melhor opção de acordo com a situação, ele escolheu o caminho verde, porque na primeira bifurcação ir para direita era melhor do que ir para esquerda.

(D) Backtracking

Backtracking é um tipo de algoritmo que representa um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas.

- Para cada chamada recursiva existem diversas opções que podem ser seguidas
- Diversos dados do subconjunto de dados de entrada ainda não incluídos na solução são candidatos. Pode ser que todos sejam e pode existir uma restrição (constraint) reduzindo o número de candidatos. Ex.: Só vértices vizinhos são candidatos a serem o próximo.
- O algoritmo pode tentar uma chamada recursiva para cada um dos candidatos (solução para pesquisa exaustiva). Ex.: O Caixeiro Viajante tenta todos os caminhos.
- O algoritmo pode escolher um ou poucos dados segundo um critério qualquer.
- O processo de busca cria uma árvore de chamadas recursivas. Ex.: Todos os caminhos parciais do caixeiro viajante.

6. [QUESTÃO – 08]

Defina e exemplifique:

(A) Problema SAT x Teoria da NP-Completude

O problema da satisfatibilidade é o problema central da teoria da NP-completude, tal problema consiste em dada uma expressão booleana, pregunta-se se há alguma atribuição de valores para as variáveis que torne a expressão verdadeira. O algoritmo para a resolução desse problema consiste em testar todas as possibilidades de atribuição de valores para as variáveis. As soluções para esses problemas são exponenciais, que o classifica como problema NP-completo.

Exemplo:

$A = (a1 \vee \neg a2 \vee a3) \wedge (\neg a1 \vee a2 \vee \neg a3)$

a1 = 1, a2 = 1 e a3 = 1.

A = TRUE.

Essa solução é satisfativa para a expressão A.

(B) Classes P, NP, NP-Difícil e NP-Completude

As classes P, NP, NPCompleto e NPDifícil caracterizam problemas quanto à sua solução em tempo polinomial.

- P é solucionável em tempo polinomial.
- NP não é solucionável em tempo polinomial.
- NP-Difícil são problemas que não são solucionáveis em tempo polinomial.
- NP-Completo é a interseção entre NP e NP-Difícil: representa problemas que têm solução, mas não em tempo polinomial.

7. [QUESTÃO – 09]

Descreva a redução (prove a NP-Completude) do problema do SAT ao Clique. Apresente o pseudocódigo do algoritmo NP e mostre graficamente as instâncias e soluções, no processo de redução.

A redução do problema SAT ao Clique é transformar uma instancia X do SAT em uma instancia Y do clique, com a condição de que se X = True se somente se Y = True.

Algoritmo polinomial para, dada uma expressão booleana na FNC, α, (instância de SAT), gerar um grafo G(V,E) e um natural k ≤ |V| tal que:

α é satisfazível se existe um k-clique em G.

Algoritmo para gerar G(V,E) e k:

Seja = C1 ∧ C2 ∧ ... ∧ CM

V ← {vi j, onde i é a cláusula e j é a variável}

E ← { (vi j, v k l), onde I != k e vi j ¬ ∨ kl }.

Exemplo:

$$\alpha = (X \vee Y \vee \neg Z) \wedge (\neg Y \vee Z) \wedge (\neg X \vee Y \vee \neg Z)$$