LISTA 2 - Análise de Algoritmos 2018

Prof: Herbert Oliveira Rocha

Aluno: João Carlos Coelho Filho - 2017012584

[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

O cálculo do n-ésimo termo da sequência de Fibonacci é definido matematicamente, pela relação de recorrência abaixo, com os primeiros terms sendo $F_0 = 0$ e $F_1 = 1$:

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

A função recursiva é dada por:

```
#include<stdio.h>
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

A complexidade do algoritmo recursivo é dada por:

$$T(n) = 2T(n-1) + C$$
$$= 2^2 \cdot T(n-2) + (2+1)C$$
$$= 2^3 \cdot T(n-3) + (2^2+2+1)C$$
$$= \dots$$
$$= C \cdot \sum_{i=0}^{n-1} 2^i$$
$$= C \cdot \frac{2^n - 1}{2-1} = O(2^n)$$

A função interativa é dada por:

```
int fib(int n)
{
  int a = 0, b = 1, c, i;
  if( n == 0)
    return a;
  for (i = 2; i <= n; i++)
  {
     c = a + b;
     a = b;
     b = c;
  }
  return b;
}
```

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

(A) 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:

```
void permute(char *a, int l, int r)
{
   int i;
   if (l == r)
     printf("%s\n", a);
   else
   {
       for (i = l; i <= r; i++)
       {
          swap((a+l), (a+i));
          permute(a, l+1, r);
          swap((a+l), (a+i)); //backtrack
       }
   }
}
```

A complexidade para se encontrar todas as permutações de um número é O(N!) (Fatorial).

[QUESTÃO – 02]

Defina e dê exemplos:

(A) **Grafos:**

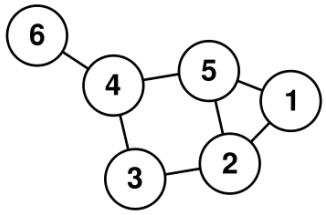
Um **grafo direcionado** consiste de:

- um conjunto *V* de *vértices*,
- um conjunto *E* de *arestas* e
- mapas *s*, *t* : *E* → *V*, onde *s*(*e*) é a *fonte* e *t*(*e*) é o *alvo* da aresta direcionada *e*.

Um **grafo não direcionado** (ou simplesmente **grafo**) é dado por

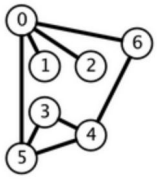
- um conjunto *V* de *vértices*,
- um conjunto *E* de *arestas* e
- uma função *w* : *E* → P(*V*) que associa a cada aresta um subconjunto de dois ou de um elemento de *V*, interpretado como os pontos terminais da aresta.

Ex:



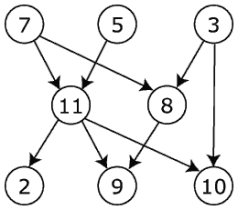
(B) Grafo conexo, acíclico e direcionado:

O grafo **conexo** é quando existe pelo menos um caminho entre qualquer vértice para todos os outros vértices.



Grafo conexo

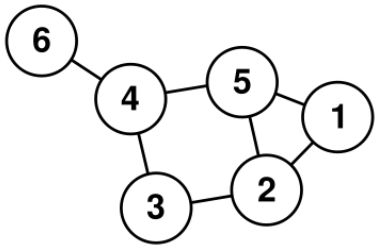
O grafo **acíclico** é o tipo de grafo no qual os caminhos entre os vértices não possuem vértices repetidos. E o grafo **direcionado** é um tipo de grafo onde as arestas possuem sentido, ou seja, não é permitido percorrer pelas arestas no sentido contrário.



Grafo acíclico e direcionado

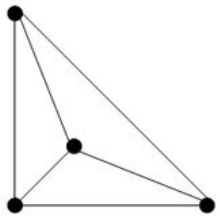
(C) Adjacência x Vizinhança em grafos:

Em teoria dos grafos, um vértice adjacente é aquele que possui uma aresta diretamente ligada a outro determinado vértice. Já a vizinhança de um vértice v consiste em um subgrafo contendo todos os vértices adjacentes a este dado grafo, assim como as arestas que ligam os vértices deste subgrafo. No grafo abaixo, considerando o vértice 4, sua vizinhança é formada pelos vértices 6, 5 e 3, assim como as arestas que os ligam.



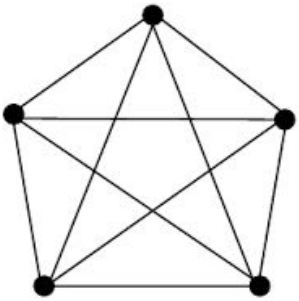
(D) Grafo planar:

É um grafo que pode ser representado no plano sem a sobreposição (cruzamento) de arestas.

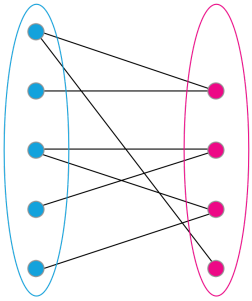


(E) Grafo completo, clique e grafo bipartido:

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 subgrafo 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.



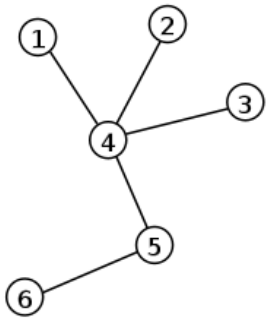
Grafo completo no qual os vértices formam um clique de tamanho 5.



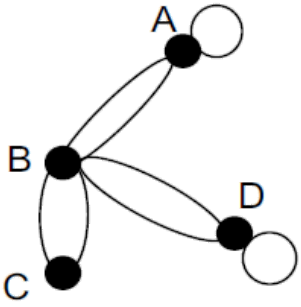
Grafo bipartido

(F) Grafos simples x multigrafo x dígrafo:

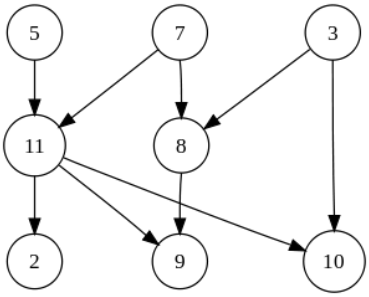
O **grafo simples** é um grafo que não possui arestas paralelas nem laços. O **multigrafo** é um tipo de grafo onde os vértices possuem arestas paralelas. E o **dígrafo** é um grafo direcionado e simples.



Grafo simples



Multigrafo



Dígrafo

[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.

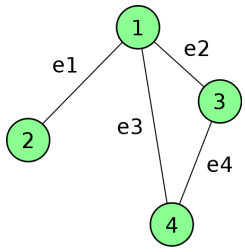
Dado um grafo G com *n* vértices e m arestas, podemos representá-lo em uma matriz *n x m* M. A definição precisa das entradas da matriz varia de acordo com as propriedades do grafo que se

deseja representar, porém de forma geral guarda informações sobre como os vértices se relacionam com cada aresta.

No exemplo abaixo a matriz armazena a incidência das arestas (e) nos vértices (1,2,3,4). Quando uma dada aresta se relaciona com um dado vértice, é marcado 1, senão 0.

	e_1	e_2	e_3	e_4
1	1	1	1	0
2	1	0	0	0
3	0	1	0	1
4	0	0	1	1

$$= \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

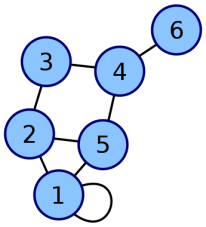


Matriz de adjacência

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:

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

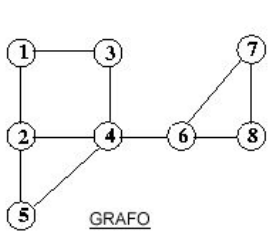


Lista de adjacência

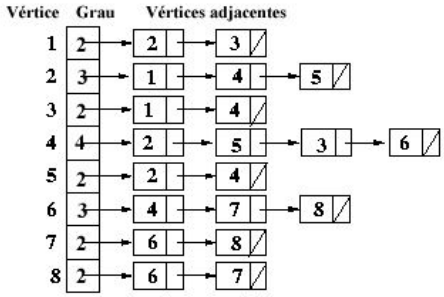
Uma lista de adjacência, é a representação de todas arestas ou arcos de um grafo em uma lista.

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.

Exemplo:



GRAFO



LISTA DE ADJACÊNCIAS

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 escrita 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.
- Se o número de arestas é muito inferior ao número de vértices pode ser melhor utilizar lista de adjacência.

[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.

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

Uma função de geração de hash pode gerar dois valores iguais para chaves diferentes, o que é chamado de colisão. Uma forma de solucionar este problema é a utilização do encadeamento fechado. Nele é implementada uma subestrutura em cada posição da tabela hash para armazenar chaves com o mesmo valor de hash, essas estruturas podem ser listas encadeadas, árvores ALV, árvores vermelho-preto, dentre outras.

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

- No caso das ávores balanceadas, as operações de busca, inserção e remoção teriam complexidade *O(lg n)*.

[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 explícita faz uma enumeração de todas as possíveis soluções, e a enumeração implícita faz uma enumeração "inteligente" com as melhores soluções para o problema.

Para resolver certos problemas computacionais é necessário enumerar — ou seja, fazer uma lista de — todos os objetos de um determinado tipo (por exemplo, todas as árvores binárias de busca com chaves 1, 2, ... , n). O número de objetos é tipicamente muito grande, e portanto a enumeração consome muito tempo.

Os problemas de enumeração estão relacionados com expressões como *backtracking*, *busca exaustiva*, *força bruta*, e *branch-and-bound*.

(B) Programação Dinâmica

Programação dinâmica é um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória. Ele é aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada - de forma a evitar recálculo - de outros subproblemas que, sobrepostos, compõem o problema original.

- Transforma um problema de otimização complexo em uma sequência de subproblemas simples.
- Funciona de trás para frente (problemas menores a maiores)
- 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 de Recursão simples

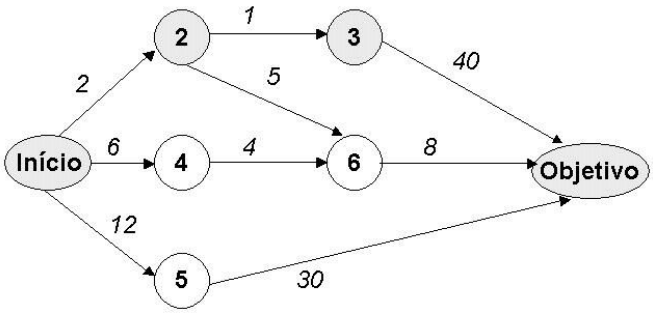
```
#include<stdio.h>
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

Método usando Programação Dinâmica

```
int fib(int n)
{

int f[n+2];
int i;

f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}
return f[n];
}
```

(C) Algoritmo Guloso

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

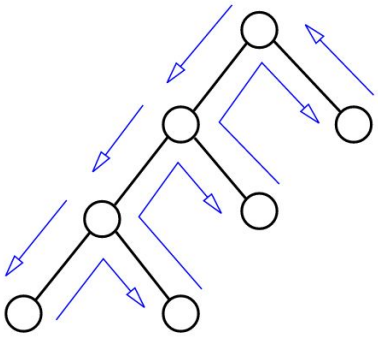


O Caminho escolhido pelo algoritmo guloso pode não representar a melhor solução global, conforme se verifica no grafo acima. O caminho escolhido resulta em 43, quando o melhor caminho teria peso total de apenas 18.

(D) Backtracking

Backtracking é um algoritmo genérico que busca, por força bruta, soluções possíveis para problemas computacionais (tipicamente problemas de satisfações à restrições).

- De maneira incremental, busca por candidatos à soluções e abandona cada candidato parcial C quando C não pode resultar em uma solução válida.
- Quando sua busca chega a uma extremidade da estrutura de dados, como um nó terminal de uma árvore, o algoritmo realiza um retrocesso tipicamente implementado através de uma recursão.



[QUESTÃO – 06]

Implemente uma solução para multiplicação de matrizes utilizando programação dinâmica, visando determinar uma ordem em que as matrizes sejam multiplicadas, de modo a minimizar o número de multiplicações envolvidas.

No repositório.

[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, o problema se resume em: dada uma determinada expressão booleana na forma conjuntiva, existe um conjunto de valores para as variáveis que torne a expressão verdadeira?.

Exemplo:

$$A = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

$$x=1, y=1, z=0$$

$$A = \text{Verdadeira}$$

(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 por uma Máquina de Turing Determinística

- NP - Tempo polinomial não determinístico, problemas que são decidíveis em tempo polinomial por uma máquina de Turing não determinística.
- 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.

[QUESTÃO – 10]

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 instância 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:

α = (X ∨ Y ∨ ¬Z) ∧ (¬Y∨ Z) ∧ (¬X ∨ Y ∨ ¬Z)