Resumo capítulo 3

Recursão

Recursão é o processo em que um objeto se repete dentro dele mesmo, ou seja, um objeto é definido em termos do próprio.

1. Conhecendo a teoria para programar

Na computação, a recursividade trata de dividir um problema maior em problemas menores, uma vez que a resolução é feita na mesma função que é recorrentemente chamada. Para a criação de uma função recursiva, deve-se:

- criar uma condição de parada.
- ter uma mudança de estado a cada chamada.

Como exemplo da utilização de recursividade na implementação de um código, o autor propõe o cálculo do fatorial de um número.

Embora a implementação recursiva seja mais simples que a versão iterativa, em muitos casos, consomem maior número de recursos, como memória e processamento e são muito mais difíceis de testar quando há muitas chamadas. Todavia, a sua utilização faz com que os códigos se tornem mais “enxutos” e com sua compreensão facilitada.

Sendo assim, deve-se utilizar a recursão quando:

1. o problema é naturalmente recursivo e a versão recursiva do algoritmo não gera ineficiência evidente, se comparada com a versão iterativa.
2. o algoritmo se torna compacto, sem perda de clareza ou generalidade.
3. é possível prever que o número de chamadas não vão provocar interrupção no processo.

Você NÃO deve utilizar a recursão quando:

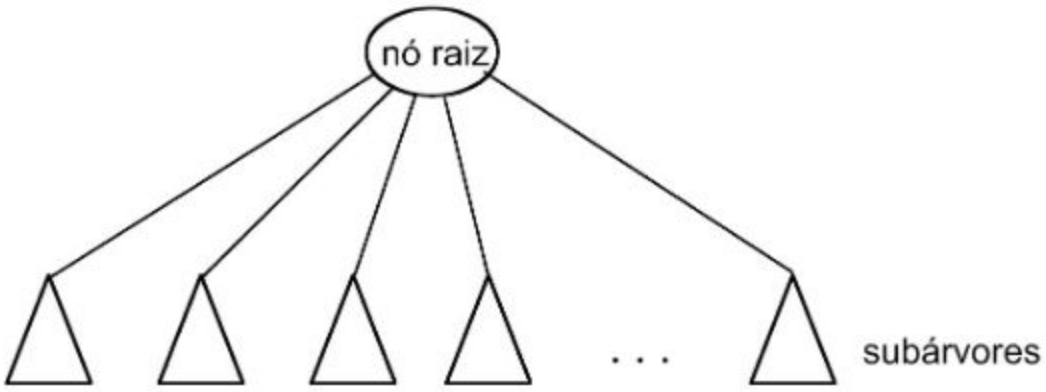
1. a solução recursiva causa ineficiência, se comparada com a versão iterativa.
2. existe uma única chamada do procedimento/função recursiva no fim ou no começo da rotina, com o procedimento/função podendo ser transformado numa iteração simples.
3. o uso de recursão acarreta número maior de cálculos que a versão iterativa.
4. parâmetros consideravelmente grandes têm que ser passados por valor.
5. não é possível prever o número de chamadas que podem causar sobrecarga na pilha.

A grande vantagem da utilização de algoritmos recursivos é que, em certos casos, estes proporcionam uma forma mais simples e clara de se expressar em linguagem computacional. Contudo, isso não garante a melhor solução, principalmente em termos de tempo e consumo de memória. Como é o caso do algoritmo que gera a sequência de Fibonacci, que em virtude da repetição de cálculos e chamadas exponenciais, a versão iterativa é muito mais eficiente.

Resumo capítulo 16

Árvores

Árvores são estrutura de dados usadas para a representação de hierarquias, sua definição é feita usando recursividade. Esse tipo de estrutura é composta por um conjunto de nós, existe um nó r, denominado raiz, que contém zero ou mais subárvores, cujas (sub)raízes são ligadas diretamente a r. Esses nós raízes das subárvores são ditos filhos do nó pai, r. Nós com filhos são comumente chamados de nós internos e nós que não têm filhos são chamados de folhas ou nós externos.



1. Árvores binárias

Um exemplo da utilização de árvores binárias é a avaliação de expressões. Nesse caso, os nós das árvores para representar uma expressão deve ter no máximo dois filhos. Nessa estrutura, os nós folhas representam os operando e os nós internos, os operadores.

Em uma árvore binária, cada nó pode apresentar zero, um ou dois filhos. A árvore é representada pelo seu nó raiz e, de maneira recursiva, pode-se definir a árvore como:

- uma árvore vazia (ponteiro para a raiz nulo);
- um nó raiz tendo duas subárvores, identificadas como a subárvore da direita (sad) e a subárvore da esquerda (sae).

1.1 Representação em C

Para a representação de uma árvore, cada nó deve armazenar três informações: a informação propriamente dita, no caso, um caractere, e dois ponteiros para as sub árvores, à esquerda e à direita. Assim, a estrutura de um nó é dada por:

```
typedef struct arvno ArvNo;
struct arvno {
  char info;
  ArvNo* esq;
  ArvNo* dir;
};
```

Cria-se, ainda, um tipo abstrato de dados que representa a árvore como um todo. Para tanto, precisa armazenar um ponteiro para o nó raiz. A partir do qual, tem acesso aos demais nós da árvore:

```
typedef struct arv Arv;
struct arv {
  ArvNo* raiz;
};
```

A interface da TAD irá depender essencialmente da forma de utilização que se pretende fazer da árvore. As funções básicas implementadas são a de criação da árvore e a de um nó, exibição do conteúdo através de funções recursivas que percorre a estrutura, liberação de memória alocada pela estrutura e localização de elementos.

1.2 Ordens de percurso em árvores binárias

Muitas operações em árvores binárias envolvem o percurso de todas as subárvores. Dessa forma, é comum percorrer a árvore, executando ações de tratamento em cada nó, das seguintes formas:

- pré-ordem: trata raiz, percorre sae, percorre sad;
- ordem simétrica: percorre sae, trata raiz, percorre sad;
- pós-ordem: percorre sae, percorre sad, trata raiz.

1.3 Altura de uma árvore

O definição de altura de uma árvore é o comprimento do caminho da raíz até a folha mais distante. Dessa forma, uma árvore que tem apenas o nó raíz sua altura é zero e uma árvore vazia sua altura é -1. Diz-se que uma árvore é cheia se todos os seus nós folhas estão no mesmo nível e não se pode acrescentar nenhuma nova folha sem aumentar a altura da árvore e é degenerada se todos os seus nós internos têm uma única subárvore associada.

2. Árvore binária de busca

As árvores binárias apresentam uma propriedade fundamental: o valor associado à raiz é sempre maior que o valor associado a qualquer nó da subárvore à esquerda e é sempre

menor que o valor associado a qualquer nó da subárvore à direita. Garantindo que, quando a árvore é percorrida em ordem simétrica, os valores são encontrados em ordem crescente. Usando essa propriedade de ordem, a busca de um valor em uma árvore pode ser feita de forma eficiente.

2.1 Operações em árvores binárias de busca

O tipo do nó da árvore binária para armazenar valores inteiros pode ser dado por:

```
struct arvno { int
  info;
  ArvNo* esq;
  ArvNo* dir;
};
```

A estrutura externa que representa a árvore contém um ponteiro para a raiz. Algumas funções usando essa estrutura foram implementadas, são elas:

- busca : função que busca um elemento na árvore, explora a propriedade de ordenação da árvore;
- insere : função que insere um novo elemento na árvore, usa-se sua estrutura recursiva e a ordenação especificada na propriedade fundamental;
- retira : função que retira um elemento da árvore, existem três situações a serem pensadas, a retirada de um elemento de nó folha, exclusão de um elemento com apena um nó filho ou apagar um elemento com dois nós filho.

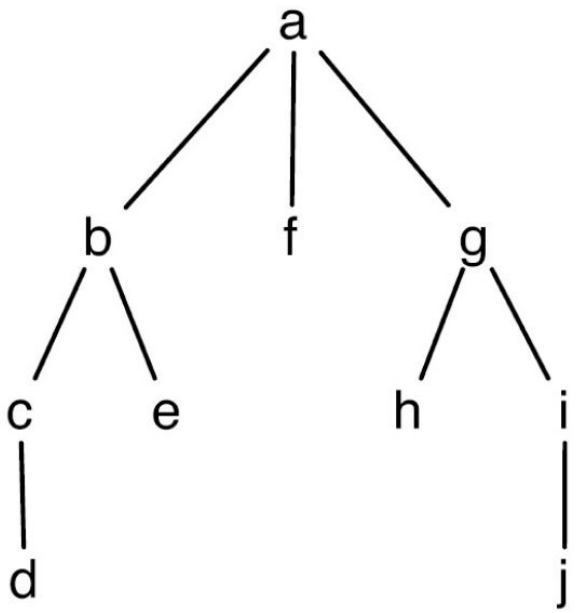
2.2 Árvores balanceadas

Após várias inserções/remoções, a árvore tende a ficar desbalanceada, uma vez que as operações implementadas na função não garantem o seu balanceamento.

A ideia central de um algoritmo para balancear uma árvore binária de busca pode ser a seguinte: se tivermos uma árvore com m elementos na *sae* e n ≥ m + 2 elementos na *sad*, podemos tornar a árvore menos desequilibrada movendo o valor da raiz para a sae, na qual ele se tornará o maior valor, e movendo o menor elemento da *sad* para a raiz. Dessa forma, a árvore continua com os mesmos elementos na mesma ordem. A situação em que a *sad* tem menos elementos que a *sae* é semelhante. Esse processo pode ser repetido até que a diferença entre os números de elementos das duas subárvores seja menor ou igual a 1.

3. Árvores com número variável de filhos

São estruturas de árvores nas quais cada nó pode ter mais do que duas subárvores associadas.



3.1 Representação em C

Dependendo da aplicação, pode-se usar diferentes estruturas para representar árvores, levando em consideração o número de filhos que cada nó pode apresentar. A interface da árvore apresentada no livro é composta pelas funções:

- criano : cria um nó folha, dada a informação a ser armazenada.
- insere : insere uma nova subárvore como filha de um dado nó.
- cria : cria uma árvore, dado seu nó raiz.
- imprime : percorre todos os nós e imprime suas informações.
- busca : verifica a ocorrência de determinado valor num dos nós da árvore.
- libera : libera toda a memória alocada pela árvore.
- altura : calcula o comprimento do caminho da raíz até a folha mais distante.

Questão 1

pares.c

```
void contador_par(ArvNo *r, int *count)
{
   if (r != NULL)
   {
       if(r->info%2 == 0) // entra se o numero r passado como paramentro for
divisivel por 2
           *count = *count + 1; // incrementa no contador

       // percorre a arvore pela esquerda e direita, por meio de chamadas
recursivas da funcao, ate o valor de r ser NULL
       contador_par(r->esq,count);
       contador_par(r->dir,count);
   }
}

int pares(Arv *a)
{
   int count = 0; // contador de pares inicia com 0
   contador_par(a->raiz, &count); // chamada de funcao para verificar a
                                  // quantidade de numeros pares na arvore

   return count;
}
```

pares.h

```
#ifndef ARVORE
#define ARVORE

typedef struct arvno ArvNo;
typedef struct arv Arv;

ArvNo *arv_criano(int c, ArvNo *esq, ArvNo *dir);
Arv *arv_cria(ArvNo *r);
int pares (Arv* a);
void contador_par(ArvNo *r, int* count);

#endif
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "pares.h"

int main()
{
   // cria a arvore
   Arv *a = arv_cria(
               arv_criano(1,
                       arv_criano(2,
                                  NULL,
                                  arv_criano(4, NULL, NULL)
                       ),
                       arv_criano(3,
                                  arv_criano(8, NULL, NULL), arv_criano(6,
NULL, NULL)
                       )

               )
           );

   printf("Quantidade de valores pares: %i\n", pares(a));
}
```

Questão 2

folhas.c

```
void contador_filhos(ArvNo *r, int *count)
{
   if (r != NULL)
   {
       if(r->esq == NULL && r->dir == NULL) // entra aqui se for um no folha
           *count = *count + 1; // incrementa no contador

       // percorre a arvore pela esquerda e direita, por meio de chamadas
recursivas da funcao, ate o valor de r ser NULL
       contador_filhos(r->esq,count);
       contador_filhos(r->dir,count);
   }
}
```

```
int folhas(Arv *a)
{
   int count = 0; // contador de folhas inicia com 0
   contador_filhos(a->raiz, &count); // chamada de funcao para verificar a
                                     // quantidade de folhas na arvore

   return count;
}
```

folhas.h

```
#ifndef ARVORE
#define ARVORE

typedef struct arvno ArvNo;
typedef struct arv Arv;

ArvNo *arv_criano(int c, ArvNo *esq, ArvNo *dir);
Arv *arv_cria(ArvNo *r);
int folhas(Arv *a);
void contador_filhos(ArvNo *r, int *count);

#endif
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "folhas.h"

int main()
{
   // cria a arvore
   Arv *a = arv_cria(
               arv_criano(1,
                       arv_criano(2,
                                  NULL,
                                  arv_criano(4, NULL, NULL)
                       ),
                       arv_criano(3,
                                  arv_criano(8, NULL, NULL), arv_criano(6,
NULL, NULL)
                       )

               )
```

```
            );

    printf("Quantidade de folhas: %i\n",folhas(a));
}
```

Questão 3

um_filho.c

```
void contador_filhosUnico(ArvNo *r, int *count)
{
    if (r != NULL)
    {
        if((r->esq == NULL && r->dir != NULL) || (r->esq != NULL && r->dir ==
NULL))
        // entra aqui se o no tiver um filho a esquerda ou a direita
            *count = *count + 1; // incrementa no contador

        // percorre a arvore pela esquerda e direita, por meio de chamadas
recursivas da funcao, ate o valor de r ser NULL
        contador_filhosUnico(r->esq,count);
        contador_filhosUnico(r->dir,count);
    }
}

int um_filho(Arv *a)
{
    int count = 0; // contador de filho unico inicia com 0
    contador_filhosUnico(a->raiz, &count); // chamada de funcao para verificar a
                                           // quantidade de nos com um unico
filho na arvore

    return count;
}
```

um_filho.h

```
#ifndef ARVORE
#define ARVORE

typedef struct arvno ArvNo;
typedef struct arv Arv;

ArvNo *arv_criano(int c, ArvNo *esq, ArvNo *dir);
Arv *arv_cria(ArvNo *r);
```

```
int um_filho (Arv* a);
void contador_filhosUnico(ArvNo *r, int *count);

#endif
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "um_filho.h"

int main()
{
   Arv *a = arv_cria(
               arv_criano(1,
                       arv_criano(2,
                                  NULL,
                                  arv_criano(4, NULL, NULL)
                       ),
                       arv_criano(3,
                                  arv_criano(8, NULL, NULL), arv_criano(6,
NULL, NULL)
                       )

               )
           );

   printf("Quantidade de nos com filho unico: %i\n",um_filho(a));
}
```

Questão 6

nfolhas_maiores.c

```
void busca(ArvNo *r, int x, int *count)
{
   if (r != NULL)
   {
       // percorre a arvore pela esquerda e direita, por meio de chamadas
recursivas da funcao, ate o valor de r ser NULL
       busca(r->esq, x, count);
       if ((r->info > x) && (r->esq == NULL && r->dir == NULL)) // entra se for
um no folha e com um valor maior que o de x
           *count = *count + 1;
       busca(r->dir, x, count);
```

```
    }
}

int nfolhas_maiores(Arv *a, int x)
{
    int count = 0; // inicia o count com 0
    busca(a->raiz, x, &count); // chamada de funcao para verficar o numero de
nos folhas
                               // maiores que x
    return count;
}
```

nfolhas_maiores.h

```
#ifndef ARVORE
#define ARVORE

typedef struct arvno ArvNo;
typedef struct arv Arv;

ArvNo *arv_criano(int c, ArvNo *esq, ArvNo *dir);
Arv *arv_cria();
static ArvNo *insere(ArvNo *r, int v);
void abb_insere(Arv *a, int v);
void busca(ArvNo *r, int x, int* count);
int nfolhas_maiores (Arv*a, int x);

#endif
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "nfolhas_maiores.h"

int main()
{
    // criacao da arvore
    Arv *a = arv_cria();

    // insersao dos elementos
    abb_insere(a, 6);
    abb_insere(a, 2);
    abb_insere(a, 8);
    abb_insere(a, 1);
```

```
   abb_insere(a, 4);
   abb_insere(a, 3);

   printf("Quantidade de nos folha com valores maiores que 1: %i\n",
nfolhas_maiores(a, 1));
   printf("Quantidade de nos folha com valores maiores que 3: %i\n",
nfolhas_maiores(a, 3));
   printf("Quantidade de nos folha com valores maiores que 5: %i\n",
nfolhas_maiores(a, 5));
   printf("Quantidade de nos folha com valores maiores que 8: %i\n",
nfolhas_maiores(a, 8));
}
```

Questão 7

soma_xy.c

```
void somatorio(ArvNo *r, int x, int y, int *total)
{
   if (r != NULL)
   {
       // percorre a arvore pela esquerda e direita, por meio de chamadas
recursivas da funcao, ate o valor de r ser NULL
       somatorio(r->esq, x, y, total);
       if (r->info > y && r->info < x) // entra se o valor do elemento estiver
entre X e Y
           *total = *total + r->info; // soma o valor do elemento no total
       somatorio(r->dir, x, y, total);
   }
}

int soma_xy(Arv *a, int x, int y)
{
   int total = 0; // inicia a soma com 0
   somatorio(a->raiz, x, y, &total); // chamada da funcao para a soma dos
                                     // elementos entre X e Y
   return total;
}
```

soma_xy.h

```
#ifndef ARVORE
#define ARVORE

typedef struct arvno ArvNo;
```

```
typedef struct arv Arv;

ArvNo *arv_criano(int c, ArvNo *esq, ArvNo *dir);
Arv *arv_cria();
static ArvNo *insere(ArvNo *r, int v);
void abb_insere(Arv *a, int v);
void somatorio(ArvNo *r, int x, int y, int* total);
int soma_xy (Arv*a, int x, int y);

#endif
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "soma_xy.h"

int main()
{
   // criacao da arvore
   Arv *a = arv_cria();

   // insersao dos elementos
   abb_insere(a, 6);
   abb_insere(a, 2);
   abb_insere(a, 8);
   abb_insere(a, 1);
   abb_insere(a, 4);
   abb_insere(a, 3);

   // De acordo com a questao X > Y
   // soma_xy(Arv,X,Y)
   printf("somatorio dos valores entre 8 e 1: %i\n", soma_xy(a, 8, 1));
   printf("somatorio dos valores entre 5 e 2: %i\n", soma_xy(a, 5, 2));
   printf("somatorio dos valores entre 7 e 3: %i\n", soma_xy(a, 7, 3));
}
```

Questão 8

nivel.c

```
int nivel_no(ArvNo *r, int x)
{
   int he, hd;
```

```
    if (r->info == x) // entra se o valor for o no raiz
        return 0;

    else if (r->info > x)
    // se o valor for menor que o no raiz o elemento esta na esqueda
    // pecorre pela esquerda ate achar o elemento e retornar a sua posicao
        return he = nivel_no(r->esq, x) + 1;

    else if (r->info < x)
    // se o valor for maior que o no raiz o elemento esta na direita
    // pecorre pela direita ate achar o elemento e retornar a sua posicao
        return hd = (nivel_no(r->dir, x) + 1);

}

int nivel(Arv *a, int x)
{
    return nivel_no(a->raiz, x); // chamada da funcao para pecorrer a arvore e
retornar o valor
}
```

nivel.h

```
#ifndef ARVORE
#define ARVORE

typedef struct arvno ArvNo;
typedef struct arv Arv;

ArvNo *arv_criano(int c, ArvNo *esq, ArvNo *dir);
Arv *arv_cria();
static ArvNo *insere(ArvNo *r, int v);
void abb_insere(Arv *a, int v);
int nivel_no(ArvNo *r, int x);
int nivel (Arv*a, int x);

#endif
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "nivel.h"

int main()
```

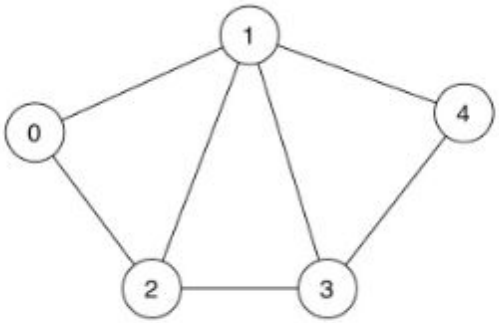
```
{
   // criacao da arvore
   Arv *a = arv_cria();

   // insersao dos elementos
   abb_insere(a, 6);
   abb_insere(a, 2);
   abb_insere(a, 8);
   abb_insere(a, 1);
   abb_insere(a, 4);
   abb_insere(a, 3);

   printf("Nivel do no do elemento 4: %i\n", nivel(a, 4));
   printf("Nivel do no do elemento 3: %i\n", nivel(a, 3));
   printf("Nivel do no do elemento 8: %i\n", nivel(a, 8));
}
```

Resumo capítulo 22

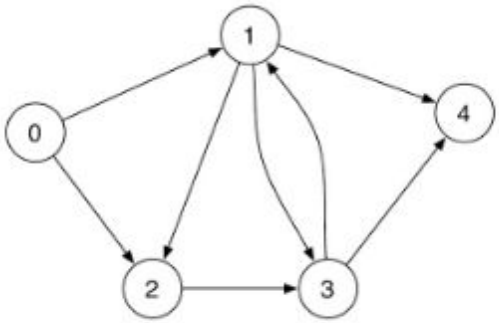
Grafos

Grafo é uma estrutura de dados que representa uma relação de conectividade entre nós, em que cada nó representa um vértice e cada conexão representa uma aresta do grafo.

Um grafo pode ser orientado ou não orientado. Em um grafo não orientado, cada aresta representa uma conexão entre os dois vértices ligados, nos dois sentidos. Já num grafo orientado, cada aresta tem uma direção.



Grafo não orientado



Grafo orientado

1. Representação

Os grafos podem ser representados de duas formas: lista de adjacência e matriz de adjacência. Usando-se lista de adjacência, representa-se um vetor de vértices, e cada vértice guarda uma lista de arestas que partem deste vértice. Na representação com matriz de adjacência, simboliza-se uma matriz V × V , onde cada elemento i_{xy} da matriz representa uma possível aresta conectando o vértice v_x ao vértice v_y .

2. Busca em profundidade

A visita nos vértices de um grafo a partir do vértice de origem pode ser feito por meio de busca em profundidade ou busca em amplitude.

Para percorrer o grafo em profundidade, parte-se do vértice v, visitando cada um dos vértices adjacentes, percorrendo todos os descendentes de um primeiro vértice adjacente antes de explorar o segundo e, assim, recursivamente. De forma a evitar que um vértice seja visitado mais de uma vez.

3. Busca em amplitude

Para percorrer o grafo em amplitude ou em largura, explora-se todos os vértices adjacentes a um dado vértice antes de prosseguir com a exploração dos descendentes destes vértices adjacentes. A exploração nesse formato faz com que a diferença entre o tempo inicial e final de exploração de um vértice diminua, uma vez que o algoritmo imediatamente explora todos os vértices adjacentes a um dado vértice.

4. Ordenação topológica

Ordenar topologicamente os vértices em relação às suas interdependências no grafo refere-se a extração da ordem de execução das tarefas a partir de uma tarefa origem. Isto pode ser feito a partir da enumeração dos vértices em ordem decrescente de “tempo final” associado em uma busca em profundidade ou fazendo a contabilização do número de arestas que chegam em cada vértice, enquanto existirem vértices no repositório que foi criado para o armazenamento dos V, retira-se um vértice e acessa-se seus vértices adjacentes.

5. Caminho mínimo

Existem três problemas de caminhos mínimos que podemos enumerar:

- Encontrar o caminho mínimo de um vértice origem a qualquer outro vértice do grafo.
- Encontrar o caminho mínimo de um vértice origem até determinado vértice destino.
- Encontrar os caminhos mínimos entre quaisquer pares de vértices do grafo.

Quando falamos de caminho mínimo, em geral associamos pesos (custos) arbitrários às arestas. Os algoritmos de caminho mínimo se baseiam no procedimento de relaxação de arestas: se existe a aresta $i \rightarrow j$, o custo do caminho mínimo para se chegar a j deve ser naturalmente menor do que o custo de se chegar a i, acrescido do custo da aresta $i \rightarrow j$; caso contrário, o caminho mínimo para se chegar a j deve passar pelo vértice i.

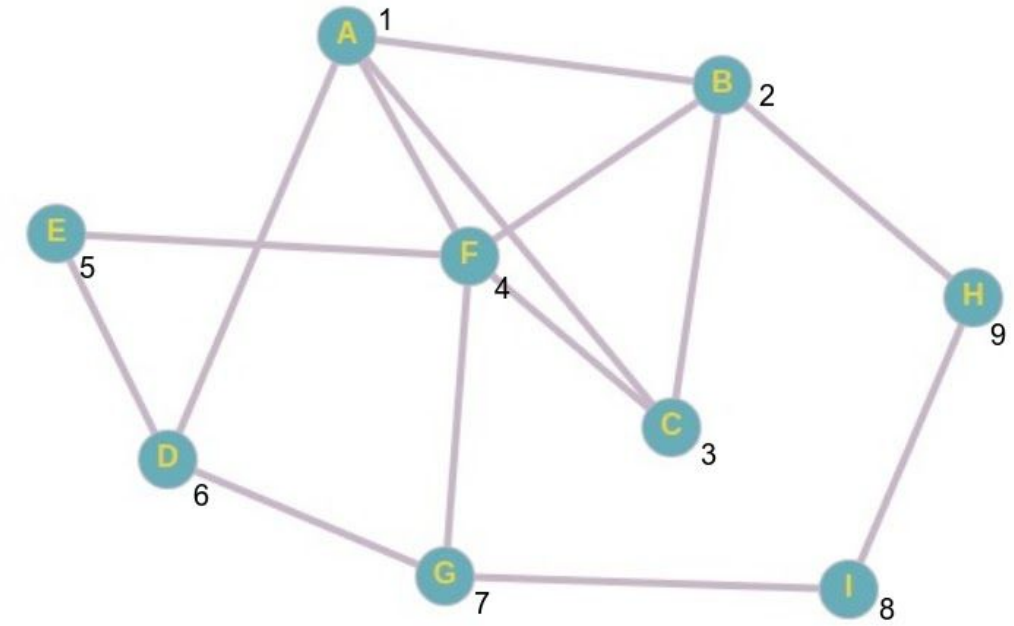
6. Fluxo máximo

Neste problema, cada aresta tem associada uma capacidade máxima de fluxo. O objetivo é obter um algoritmo que informe o fluxo máximo possível da rede do vértice origem (s) ao vértice destino, conhecido como escoadouro (t). Nesse caso, deve levar em consideração que o fluxo que está entrando em um vértice deve ser igual ao fluxo que irá sair, para qualquer vértice interno.

Letra B

	A	B	C	D	E	F	G	H	I
A	0	1	1	1	0	1	0	0	0
B	1	0	1	0	0	1	0	1	0
C	1	1	0	0	0	1	0	0	0
D	1	0	0	0	1	0	1	0	0
E	0	0	0	1	0	1	0	0	0
F	1	1	1	0	1	0	1	0	0
G	0	0	0	1	0	1	0	0	1
H	0	1	0	0	0	0	0	0	1
I	0	0	0	0	0	0	1	1	0

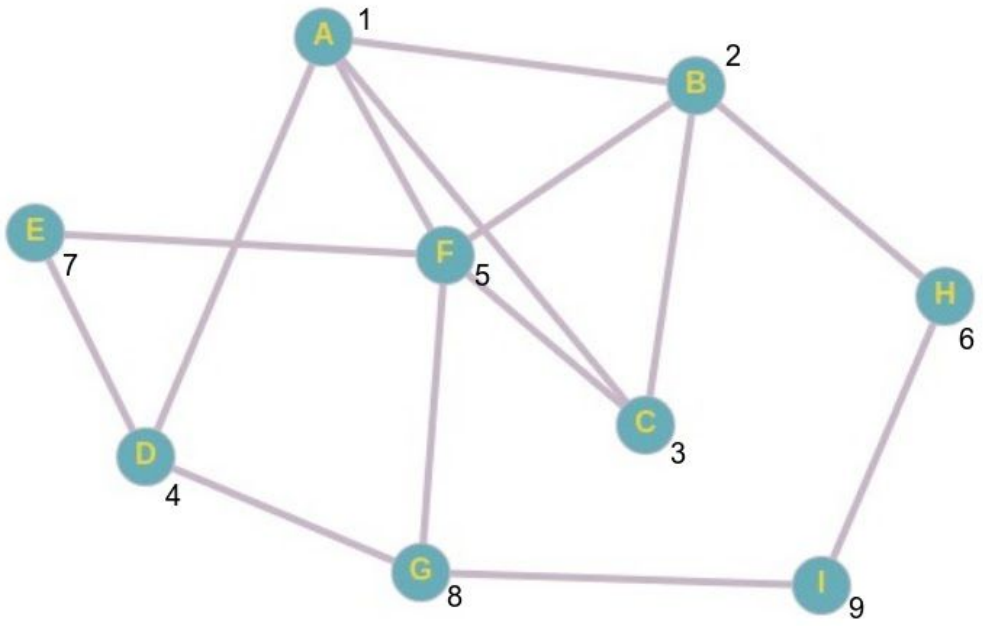
Letra C



Para percorrer um grafo em profundidade, intuitivamente, o algoritmo começa num nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder. Para explorar o grafo em questão leva-se em consideração a ordem alfabética e nó raiz em A. Dessa

forma, o primeiro elemento a se visitar é o A, ao analisar a ordem alfabética o próximo elemento é o B; seguindo o mesmo raciocínio neste ponto, segue para o C; dos ramos do ponto C falta explorar o F, tornando ele o 4ª elemento a ser visitado; no ponto F, tem-se 2 possibilidades, E e G, como o E aparece primeiro no alfabeto, é o que devemos considerar; do nó E falta visitar o ramo D, que se torna o próximo ponto; do nó D segue-se para G. Ao analisar esse nó, falta visitar o ramo I, que se torna o próximo ponto, seguindo, posteriormente, para o H, visitando assim todos os pontos do grafo.

Letra D



Para percorrer um grafo em largura, intuitivamente, você começa pelo vértice raiz e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim por diante. Para explorar o grafo em questão leva-se em consideração a ordem alfabética e nó raiz em A. Dessa forma, o primeiro ponto a se visitar é o nó A, ao explorar seus vizinhos, em ordem alfabética, temos a seguinte ordem: B, C, D e F, que serão explorados consecutivamente. Seguindo para o nó B, temos apenas o nó H como não explorado, tornando-se o 6ª a ser visitado. No nó C, não tem vizinhos a serem visitar, seguindo para o D. No D, pela ordem alfabética, temos o E e G, que serão o 7 e 8, respectivamente, pontos a serem visitados. No ponto F, nesse momento, todos os seus vizinhos já foram visitados. Sendo assim, analisa-se o nó H, onde falta explorar o vizinho I, tornando-o o 9ª ponto a ser visitado, percorrendo assim todo o grafo.