Exercícios de Fixação e Aprendizagem III

Victoria Monteiro Pontes

November 2020

1 Utilizando suas palavras, faça um resumo do capítulo 3 - ”Recursão” do livro ”Bianch. Estrutura de Dados e técnicas de programação”

Recursão é um conceito utilizado para definir um objeto dentro de outro de forma contínua. Ou seja, quando definimos um objeto de forma recursiva, o definimos usando suas propriedades. Esse tipo de definição remete à ideia de repetição de um objeto dentro dele mesmo. Ou seja, um laço de repetição. Vários exemplos desse tipo de conceito podem ser encontrados na matemática, computação e até mesmo nas artes. Nesta última área, o conceito é chamado de efeito Droste. Um exemplo disso é a seguinte fotografia:

Figure 1: Recursividade ou Efeito Droste

Na matemática, a recursividade está intimamente relacionada ao princípio de indução matemática. Um exemplo disso é como podemos definir os números naturais:

0 *está em* N*, sendo* N *o conjunto dos números naturais*
Se n *está em* N *então* $n + 1$ *está em* N*, sendo* n *um número qualquer*

Na computação, podemos desenvolver uma função que faça operações de forma recursiva utilizando um número *n* como parâmetro. Um exemplo é a função *Fibonacci* que descreve uma sequência de números, cujo o número seguinte é a soma dos dois termos anteriores, sendo 0 e 1 os dois primeiros termos da sequência.

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

Nessa função, quando temos $n > 1$, chamamos de forma recursiva a função para acharmos o valor que se encontra na posição (n - 1) e (n - 2) da sequência *Fibonacci*. A imagem abaixo ilustra como funciona a função Fibonacci quando n é igual a 5.

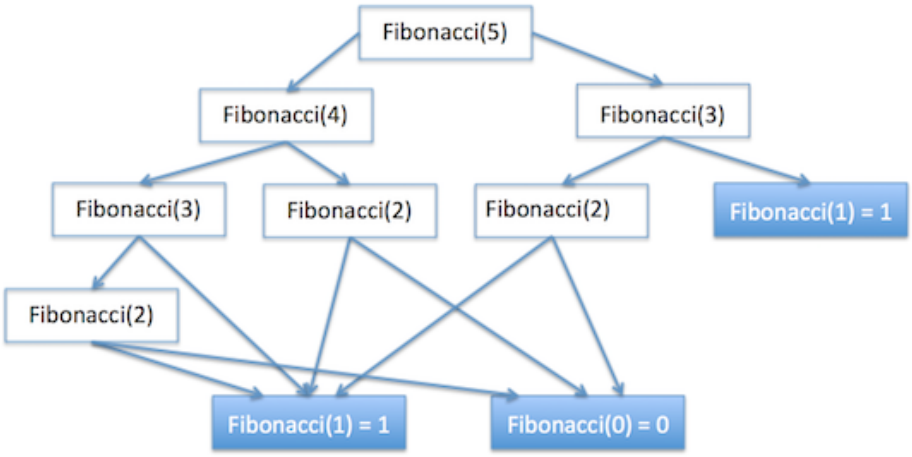

Figure 2: Ilustração da chamadas da função Fibonacci recursiva

Para a chamada da função e suas variáveis, é necessário utilizar alocação dinâmica no processo, mais especificamente, é utilizada uma pilha para empilhar as funções a cada chamada. Por isso, quando o parâmetro n é muito grande, ocorre o estouro da pilha, ou seja, não há mais espaço para empilhar mais nenhuma função e ocorre um erro de execução. Portanto, é importante saber de forma antecipada se o o tamanho do parâmetro n pode causar o estouro da pilha ou não. Outra observação importante é que normalmente, uma função recursiva exige um poder computacional e memória muito maiores do que uma versão iterativa da mesma função. Por isso, não há uma razão determinante para escolher a recursividade em vez da iteratividade. Entretanto, a vantagem de se utilizar uma versão recursiva é de que se torna muito mais fácil de se ler e compreender o código implementado.

De forma mais genérica, para implementar uma função recursiva, é necesssário estabelecer duas regras. Uma delas é que toda função recursiva deve ter uma condição de parada ou de encerramento da chamada. Podemos notar isso na função recursiva *Fibonacci* já citada, nos blocos:

```
if (n == 0){
    return 0;
}else if (n == 1){
```

```
    return 1;
}
```

Outra parte essencial da implementação recursiva é que deve haver uma mudança de estado a cada chamada, para que o parâmetro possa eventuamente chamar os blocos de condição de parada. Isso garante que a função não entre em um *loop* infinito. Percebemos isso na função *Fibonacci* no bloco:

```
    else{
    /*a mudanca de estado eh percebida quando chamamos a funcao
    fibonacci novamente utilizando (n - 1) e (n - 2)*/
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
```

Por fim, devemos evitar a utilização de recursões de cauda. Uma função possui uma recursão de cauda quando o caso recursivo é chamado no fim da função, ou seja, todo o processamento é feito antes do fim da chamada do caso recursivo. Um exemplo de recursão de cauda, é a função *Fibonacci*.

2 Questão 2

2.1 (a) Utilizando suas palavras, faça um resumo do ”capítulo 16 - Árvores” do livro ”Celes, W. et al.; Introdução a Estruturas de Dados com Técnicas de Programação em C, 2 ed., Elsevier, 2016”

Uma ”árvore”, na computação, é uma estrutura de dados com propriedades hierárquicas. Quando queremos representar uma pasta com várias pastas dentro dela e mais outras pastas dentro dessas, podemos utilizar uma árvores pra isso. Podemos pensar nessa estrutura como uma caixa, que pode conter várias caixas dentro de si e itens também. Dessa forma, criamos uma hierarquia para o acesso de dados. Uma árvore possui (análoga à uma árvore real), uma raiz e folhas, além de galhos que vamos chamar de subárvores. Todos esses elementos são organizados da seguinte forma: uma raiz é o nó inicial de uma estrutura de dados e dela

partem as subárvores, que se ligam a outros elementos, ou folhas que são nós que não possuem se ligam a nenhum outro nó. O crescimento desses elementos é normalmente mostrado, em grafos, com a raiz em cima e suas subárvores embaixo, se espalhando ao longo da estrutura.

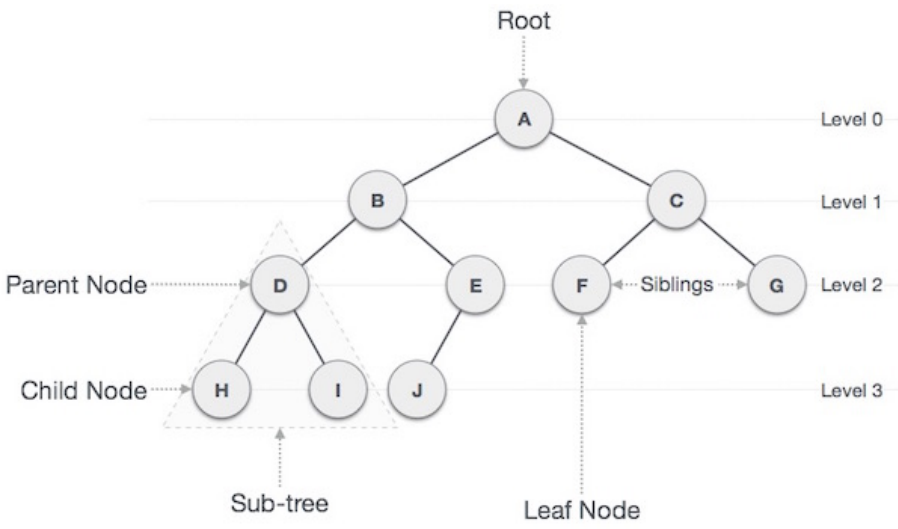

Figure 3: Ilustração de uma árvore

Como podemos olhar na figura 3, o nó A é a raiz (*root*) da árvore. Enquanto o nó D, por exemplo, é chamado de pai (*parent node*) e seus seguintes são chamados de filhos (*child node*). O conjunto de nós $\{$D,H,I$\}$ forma uma subárvore(*Sub-tree*). Vale lembrar que nós que possuem filhos também podem ser chamados de nós internos, enquanto nós sem filhos podem ser chamados de nós externos.

Um tipo muito importante de árvore, é a árvore binária, cujos nós só podem ter zero, um ou dois filhos. Para descrevê-las com notação textual, devemos descrever uma árvore vazia como $<>$, e árvores não vazias como $<$raiz (lado esquerdo) (lado direito)$>$. Vamos tomar o grafo a seguir que representa uma árvore binária:

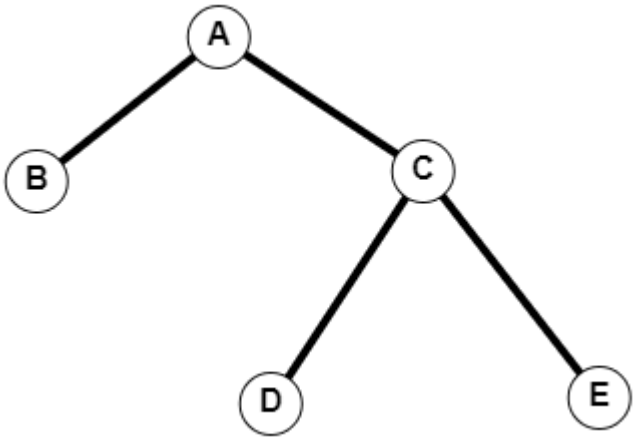

Figure 4: Grafo representando uma árvore binária

Neste exemplo, a árvore binária pode ser representada como:
<A<B<><><>><C<D<><><>><E<><><>>>>

A implementação de uma árvore, binária ou não, geralmente é feita com o auxílio da recursividade. Isso significa que muitas das funções que manipulam essa estrutura chamam a si mesmas e a outras funções. Tomamos como exemplo uma árvore binária, sendo representada na linguagem C. Podemos implementar um nó asssim:

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

Podemos também implementar um TAD para a árvore completa.

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

Às vezes, queremos percorrer uma árvore para fazer uma busca de um determinado item, ou queremos imprimir todos os valores da árvore,

ou mesmo deletá-la por completo. O modo mais intuitivo de fazer essa busca seria percorrer da raiz para as folhas, mas essa forma às vezes não é a mais eficiente para utilizar em uma função. Portanto, existem tipos de ordem de percurso (pré-ordem, ordem simétrica, pós-ordem), para árvores binárias. A pré-ordem é uma ordenação que começa pela raiz, percorrendo a subárvore esquerda e depois a subárvore direita. A ordem simétrica percorre a subárvore esquerda, depois a raiz e por fim a subárvore direita. A pós-ordem percorre primeiro a subárvore esquerda, depois a direita para então percorrer a raiz. Funções de liberar uma árvore binária, por exemplo, precisam utilizar pós-ordem, para que os nós mais externos sejam deletados primeiro, percorrendo a cadeia hierárquica no sentido contrário e evitando que algum nó deixe de ser deletado. Por exemplo:

```
static void libera (ArvNo* r){
    if (r != NULL){
        libera(r-> esq);
        libera(r-> dir);
        free(r);
    }
}

void arv_libera (Arv* a){
    libera(a-> raiz);
    free(a);
}
```

Perceba que as funções liberam os nós mais externos primeiro e por último a raiz utilizando a recursividade para atingir esse objetivo. Portanto, essas funções utilizam a pós-ordem. Um exemplo de uso de pré-ordem é uma função de impressão dos conteúdos de uma árvores, por exemplo. Esses tipos de ordens são muito úteis ao implementar as funções para manipulação de árvores.

Outra característica importante de uma árvore é a sua altura. Como só existe um caminho da raiz até um determinado nó, a altura é definida como o caminho entre a raiz e a folha mais externa da árvore. Para medir a altura de uma árvore, utilizamos níveis que são contados a partir do zero.

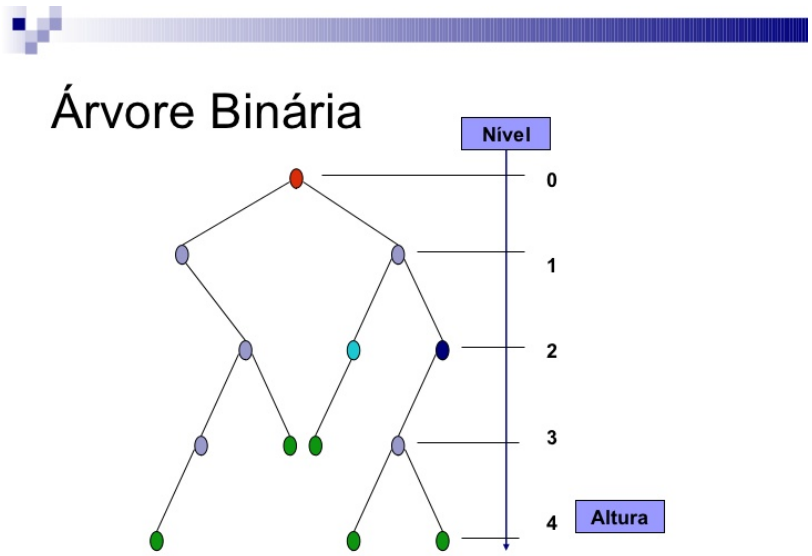

Figure 5: Níveis de uma árvore binária

Se não podemos inserir mais nenhum nó na árvore sem aumentar a altura da árvore e todos os seus nós e folhas estão no mesmo nível, dizemos que a árvore está cheia. Quando a árvore possui apenas uma subárvore associada a cada pai (nó interno), a chamamos de degenerada.

Os níveis da figura 5 são assim numerados devido à quantidade potencial de nós possíveis a cada nível de altura. Essa relação é expressa pela fórmula matemática $Q = 2^{h+1} - 1$, em que Q é a quantidade máxima de nós de uma árvore binária cheia e h é a sua altura (medida em níveis). Por causa dessa relação, uma árvore binária tem uma altura mínima de $log_2 n$, sendo n o número nós dessa árvore. Essa propriedade porém, se modifica quando a árvore é do tipo degenerada, pois a relação entre a quantidade de nós e a altura ganha um comportamento linear, de forma de $Q = h + 1$, sendo que Q é a quantidade de nós em uma árvore degenerada, consequentemente, o número mínimo de nós que podem ter em uma árvore binária de altura h.

A ligação entre esses conceitos é necessária para podermos compreender que, com circunstâncias favoráveis, podemos alcançar um nó qualquer da árvore, a partir da raiz, em até $O(log_n)$ passos, ou seja, de forma bastante eficiente. Entretanto, para manter tais condições fa-

voráveis, é necessário que as árvores binárias estejam balanceadas, ou seja, que o número de subárvores da esquerda seja próximo ao da direita, desta forma, tem-se o equilíbrio entre os dois ramos.

Uma árvore binária de busca tem como característica principal o fato de que o número armazenado na raiz sempre será o maior número dentre os nós da subárvore esquerda e o menor dentre os números pertencentes aos nós da subárvore direita. A partir daí, ao utilizarmos a ordem simétrica de percurso (subárvore esquerda $->$ raiz $->$ subárvore direita), percorremos os nós de forma que a sequência de valores armazenados nos nós será crescente. Entretanto, a ordem de percurso depende do tipo de aplicação. Se tivermos que armazenar informações mais complexas, a função de comparação também ganhará maior grau de complexidade e, dependendo do caso, os valores armazenados na árvore binária de busca podem ser repetidos.

Podemos também implementar uma árvore flexibilizando o número máximo de filhos que ela pode ter, de forma que a estrutura deixa de ser binária para ser uma estrutura com uma quantidade de filhos variável. Na linguagem C, a implementação de uma TAD com essa característica pode ser feita assim:

```
typedef struct arv3no Arv3No;
struct arv3no {
    char info;
    Arv3No *f1 , *f2 , *f3;
};
```

Como a quantidade é variável, podemos definir o número máximo de filhos que uma árvore pode ter e utilizar apenas algumas dessas subárvores disponíveis. No código acima, por exemplo, um pai pode se ligar a até 3 nós filhos, porém, caso eu queira criar somente duas subárvores, posso simplesmente deixar os ponteiros que sobram em NULL. E assim, a árvore teria apenas a primeira subárvore (começa em *f1) e a segunda (começa em *f2) e fazer o ponteiro da terceira subárvore (*f3) apontar para NULL. Entretanto, esse tipo de implementação se aplica a casos em que há um limite máximo de filhos. Quando queremos implementar uma estrutura de árvore que não possui limite máximo de filhos, podemos utilizar então uma ”lista de filhos” em que o nó pai aponta apenas para o primeiro filho e este filho aponta para o próximo filho (nó irmão). Dessa forma, formamos uma

lista encadeada que organiza os nós filhos já existentes e nos permite adicionarmos um novo filho a esse nó caso desejarmos.

A altura de uma árvore com número variável de filhos utilizam os mesmos conceitos aplicados à altura em árvores binárias. Portanto, o método para calcular o nível da altura é o mesmo.

Essas diferenças entre árvores binárias e de número de filhos variável é apenas conceitual. Na implementação do código, ainda serão utilizados os ponteiros com a mesma estrutura de nó e árvore. A diferença está apenas no que os ponteiros significam e sua representação. Ou seja, em uma árvore binária, enquanto os ponteiros apontavam para seus filhos em uma subárvore à esquerda e à direita, em uma árvore de número variável de filhos, que utiliza lista encadeada, terá um ponteiro apontando para o nó filho e outro apontando para o nó irmão.

2.2 (b) Resolva as questões 1, 2, 3, 6, 7, 8 do ”Capítulo 16 - Árvores” do livro ”Celes, W. et al.; Introdução à Estruturas de Dados com Técnicas de Programação em C, 2 ed., Elsevier, 2016”.

Exercicio 1

A implementação da função no arquivo arvore_binaria.c

```
/*QUESTAO 1*/
/*FUNCAO QUE CONTA QUANTOS NUMEROS PARES EXISTEM NA ARVORE*/
/*Chama recursivamente busca_par*/
int pares (Arv* a){
    int contador_numpares = 0;

    busca_par(a->raiz, &contador_numpares);

    return contador_numpares;
}

/*Faz a busca pelos numeros pares nos noh*/
int busca_par(ArvNo *no, int *contador_numpares){
    if (no == NULL) return 0;
```

```
    int aux = no->info % 2;

    if(aux == 0){
        *contador_numpares = *contador_numpares + 1;
    }

    busca_par(no->esq, contador_numpares);
    busca_par(no->dir, contador_numpares);

    return 1;
}
```

No arquivo exercicio1.c onde está a ”main” que testa a função.

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

int main (){

// cria a arvore
Arv* a = arv_cria(
    arv_criano(1,
            arv_criano(2, NULL ,
                    arv_criano(3, NULL , NULL)
            ),

        arv_criano(4,
            arv_criano(5, NULL , NULL), arv_criano(6, NULL , NULL)
        )

    )

);

/*Printa a quantidade de numeros pares*/
printf("Quantidade de numeros pares na arvore: %d\n",  pares(a));

/*Imprime os nohs da arvore*/
```

```
arv_imprime(a);

arv_libera(a);

return 0;
}
```

Exercicio 2

A implementação da função no arquivo arvore_binaria.c

```
/*QUESTAO 2*/
/*FUNCAO QUE CONTA QUANTAS FOLHAS TEM NA ARVORE*/
int folhas (Arv* a){
    int contador_folhas = 0;

    busca_folhas(a->raiz, &contador_folhas);

    return contador_folhas;
}

int busca_folhas(ArvNo* no, int *contador_folhas){

    if (no == NULL) return 0;

    if (no->esq == NULL && no->dir == NULL){
        *contador_folhas = *contador_folhas + 1;
    }

    busca_folhas(no->esq, contador_folhas);
    busca_folhas(no->dir, contador_folhas);

    return 1;
}
```

O teste está no arquivo exercicio2.c como ”main”

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

```c
#include "arvore_binaria.h"

int main (){

// cria a arvore
Arv* a = arv_cria(
    arv_criano(1,
            arv_criano(2, NULL ,
                    arv_criano(3, NULL , NULL)
            ),

        arv_criano(4,
            arv_criano(5, NULL , NULL), arv_criano(6, NULL , NULL)
        )

    )

);

/*Printa a quantidade de folhas*/
printf("Quantidade de folhas na arvore: %d\n",  folhas(a));

/*Imprime os nohs da arvore*/
arv_imprime(a);

arv_libera(a);

return 0;
}
```

Exercício 3

A implementação da função está no arquivo aarvore_binaria.c

```c
/*QUESTAO 3*/
/*FUNCAO QUE CONTA QUANTOS NOHS TEM APENAS UM FILHO*/
/*Chama recursivamente conta_filho*/
int um_filho(Arv* a){
    int contador_folhas = 0;
```

```
    busca_filho(a->raiz, &contador_folhas);
};


int busca_filho(ArvNo* no, int *contador_folhas){

    if (no == NULL) return 0;

    /*Nos dois ifs abaixo eh checado se o
    noh possui apenas um filho*/
    if (no->dir == NULL && no->esq != NULL){
        *contador_folhas = *contador_folhas + 1;

    }else if(no->dir != NULL && no->esq == NULL){
        *contador_folhas = *contador_folhas + 1;
    }

    busca_filho(no->dir, contador_folhas);
    busca_filho(no->esq, contador_folhas);

    return 1;
};
```

No arquivo exercio3.c está o teste da main

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

int main (){

// cria a arvore
Arv* a = arv_cria(
    arv_criano(1,
            arv_criano(2, NULL ,
                    arv_criano(3, NULL , NULL)
            ),

        arv_criano(4,
```

```
            arv_criano(5, NULL , NULL), arv_criano(6, NULL , NULL)
        )

    )

);

/*Printa a quantidade de numeros pares*/
printf("Quantidade de folhas na arvore: %d\n",  um_filho(a));


/*Imprime os nohs da arvore*/
arv_imprime(a);


arv_libera(a);

return 0;
}
```

Exercicio 6

3 Questão

3.1 (a) Utilizando suas palavras, faça o resumo do ”Capítulo 22 - Grafos” do livro ”Celes, W. et al.; Introdução a Estruturas de Dados e Técnicas de Programação em C, 2 ed., Elsevier, 2016”.

Um grafo é uma estrutura matemática que, na computação, também é uma estrutura de dados que utiliza o conceito de nós e de conectividade entre eles. Suas aplicações são variadas, sejam elas de representar mapas, árvores binárias ou de número variável de filhos, escalonamento de tarefas, entre outros. Um grafo é representado por vértices e arestas, sua notação é:

$G = (V, E)$, sendo V o conjunto de vértices e E o conjunto de arestas (*edges*). Um grafo pode ser orientado ou não. Isso significa que suas arestas podem ter um sentido único (orientado) ou não ter um sentido definido (não-orientado). Tomemos como exemplo a figura a seguir:

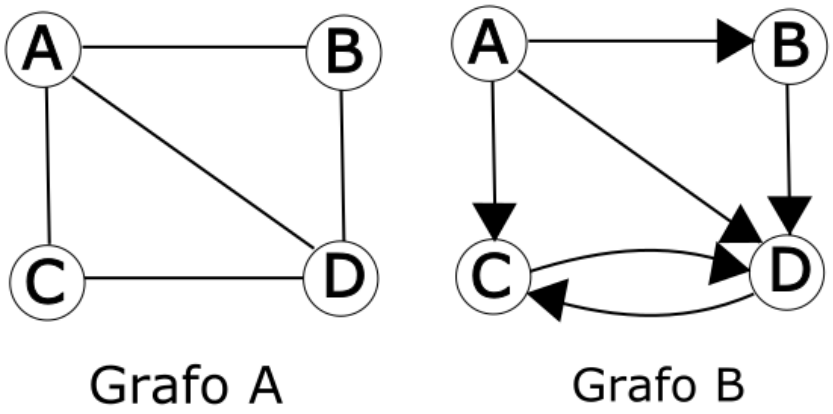

Figure 6: Grafos A e B, não-orientado e orientado, respectivamente

Às vezes, queremos representar grafos sem ter que construir uma rede de nós e arestas, especialmente na computação. Para isso, podemos representar um grafo utilizando uma lista ou uma matriz de adjacência. Em uma lista, armazenamos vértices, cada vértice guarda uma lista com as arestas que dele saem. Já uma matriz de adjacência é construída como uma matriz V x V em que a posição a_{ij} armazena algum valor relacionado aos dois nós i e j ou à aresta ij, cuja origem é o nó i e cujo destino é o nó j. Uma observação a ser feita é a eficiência da lista de adjacência para casos de arestas esparsas (poucas arestas em um grafo se comparado ao número tota de combinações de arestas possíveis), pois a sua alocação de memória necessária é $O(V + E)$ enquanto a matriz de adjacência requer uma alocação $O(V^2)$. Por esse mesmo motivo, também é preferível utilizar a matriz de adjacência para grafos que possuem uma quantidade de arestas próxima ao número total de combinações possíveis de arestas.

Existem dois métodos para percorrermos um grafo. Podemos fazer uma busca em profundidade ou uma busca em amplitude. O método de busca em profundidade consiste em, dado um vértice v, exploramos um filho seu e seus descendentes por completo antes de voltarmos e explo-

rarmos o segundo filho do vértice v. Imagine um labirinto, com muitas encruzilhadas (onde escolhemos ”para onde seguir”) e muitos caminhos. De forma simplificada, o método de busca em profundidade ”escolhe” um caminho para seguir até que ele chegue em um ”beco sem saída”, ou seja, não há mais como avançar. Então voltamos o caminho para a última encruzilhada, onde fizemos a decisão, e escolhemos um novo caminho. Como não queremos percorrer o mesmo caminho duas vezes para não percorrermos o labirinto infinitamente, marcamos o caminho com algumas migalhas. Da mesma forma, com algumas ressalvas, percorremos em profundidade um grafo. As encruzilhadas são equivalentes aos vértices e os caminhos a serem escolhidos são as arestas. Para não buscarmos em um grafo infinitamente, marcamos aqueles caminhos(arestas) que já percorremos e aquelas encruzilhadas(vértices) cujas possibilidades de caminhos já foram completamente exploradas (ou seja, todos os seus filhos e descendentes já foram explorados). Podemos também colocar uma marcação de tempo para que possamos saber o início e o fim da exploração de cada vértice.

Em um método de busca em amplitude ou largura, a ideia principal é que, dado um vértice v descoberto, este vértice é adicionado a uma fila, ao ser explorado, este vértice é retirado e seu nós vizinhos são adicionados à fila para reiniciar o processo. Dessa forma, se o vértice não está mais na fila, ele não pode ser tratado mais de uma vez.

Grafos conseguem representar uma grande gama de problemas de forma a facilitar sua resolução. Algumas dessas resoluções incluem a ordenação topológica no caso de representação de um cronograma de tarefas e algoritmos para encontrar o caminho mínimo no caso de um problema de transporte ou o fluxo máximo em um problema de fluxos em redes.

A ordenação topológica consiste em, a partir de uma primeira tarefa, obter uma ordem de execução das tarefas. Imagine três tarefas A, B e C. A tarefa C da tarefa B para ser executada e B precisa da tarefa A para ser executada. O grafo dessa situação será representado por $A->B->C$. Nessa situação, podemos ver claramente uma ordem de execução de tarefas. Primeiro executamos A, depois B e então C. Entretanto, com o crescimento da complexidade dos grafos, temos dificuldade de visualizar isso. Portanto, o algoritmo busca selecionar as tarefas que não dependem de nenhuma outra tarefa, ou seja, selecionar e guardar em um estoque os vértices que não possuem arestas de entrada. Enquanto houverem

vértices no estoque, exploramos a ele e seus vizinhos, e depois retiramos este vértice do estoque e marcamos as arestas que já foram exploradas. Se há um novo vértice cujas arestas de entradas já estão todas marcadas, o inserimos no estoque e o processo reinicia. Dessa forma, a ordem em que retiramos os vértices do estoque é a ordem topológica do grafo.

Uma observação importante é que este algoritmo considera um grafo com ciclos como um caso proibitivo, uma vez que, em um ciclo, todos os vértices dependem entre si e por isso o algoritmo não consegue explorar todos com sucesso.

Para problemas que exigem o caminho mínimo como resposta, temos alguns algoritmos que ajudam neste problema. Dentre eles, os algoritmos de Bellman-Ford, de Dijkstra e o algoritmo A*, ou ”A estrela”.

Para problemas de fluxo em rede que exigem o fluxo máximo como resposta, podemos utilizar o algoritmo de Ford-Fulkerson como ferramenta de resolução.

3.2 (b) Construa a matriz de adjacências para este grafo

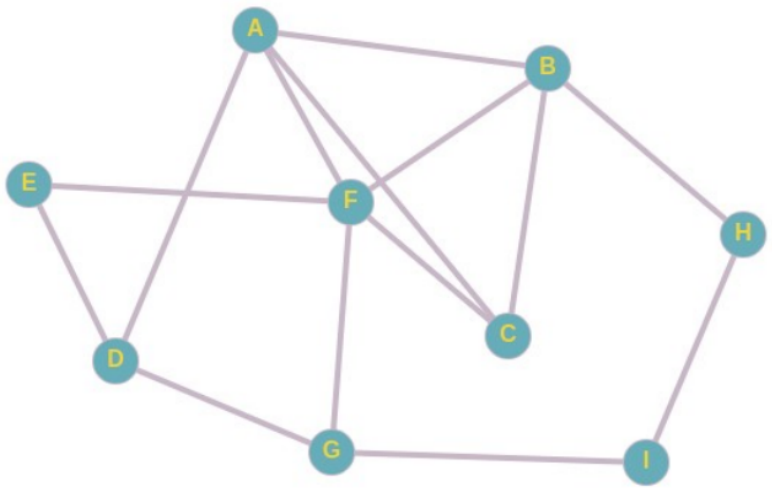

Figure 7: Grafo da questão 3

Nós	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

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

3.3 (c) Começando do nó A, explique como o grafo é percorrido em profundidade.

Para percorrer este grafo em profundidade a partir do nó A, devemos escolher um nó para seguir. Escolhemos então o nó B, para dar preferência à ordem alfabética. A partir do nó B, podemos escolher o nó H ou o nó C. Escolhemos o nó C. Quando estamos visitando o nó C, só podemos seguir por um caminho, o nó F, uma vez que os outros nós adjacentes já foram explorados exceto F. Como em F a mesma situação acontece, seguimos para o nó E. O nó E possui apenas uma adjacência que não foi visitada ainda, nó D, então escolhemos este nó. Logo depois, escolhemos o nó G, depois o nó I e logo depois o nó H. Quando chegamos ao nó H, percebemos que não há mais para onde seguir, uma vez que o nó B já está sendo visitado. Portanto, fazendo um *backtracking*, voltamos para o último nó antes de H e marcamos o caminho como já explorado (”jogamos as migalhas”). Como ainda não temos outra opção de

caminho para explorar, voltamos novamente até acharmos algum nó cujo filho ainda não foi explorado. Como não é encontrado outro nó que ainda não tenha sido explorado em nenhum dos vértices examinados ao fazer *backtracking*, desfazemos voltamos ao ponto de origem A, com todos os vértices já explorados e marcados devidamente.

Para percorrer o grafo com o método em largura a partir do nó A, utilizamos um pequeno estoque para conter os vértices que vamos explorar. Como A é o primeiro vértice, o inserimos no estoque. O estoque, ou fila, está assim: Fila = A. Após tratarmos o nó A, checamos quais são os nós vizinhos, ou seja, B, C, F, E e os adicionamos na fila. Como o nó A já foi tratado e o marcamos como já explorado, podemos tirá-lo da fila. Então Fila = B, C, E, F. Escolhemos o primeiro elemento do estoque (nó B) para ser tratado e explorado. Depois de explorado e marcado como tal, o retiramos da fila e adicionamos o seus nós adjacentes que ainda não foram explorados(sem marcação). Então Fila = C, E, F, H. Observe que o nó C já está na fila e não foi adicionado uma segunda vez ao analisarmos as adjacências de B. Com isto, passamos para o primeiro elemento da fila novamente (nó C), e o tratamos e marcamos como já explorado. Logo depois, adicionamos suas adjacências na fila com execeção das que já estão dentro dela ou que estão marcados como já explorados. Então Fila = E, F, H. Percebemos então que não adicionamos nenhum nó novo uma vez que todas as adjacências de C ou já foram exploradas/marcadas (caso dos nós A e B) ou já estão na fila (caso do nó F). Entã, partimos para o nó E (primeiro elemento da fila) e o tratamos e o marcamos como já explorado. Então adicionamos seu vizinho não explorado, o nó D, à fila e retiramos o nó E. Então Fila = F, H, D. Partimos para o nó F. Seguindo o mesmo procedimento já feito, temos que Fila = H, G. Refazemos o procedimento e tratamos o nó H. Então Fila = G, I. Partindo para o nó G, o tratamos e o marcamos como explorado e, como não há mais nenhum vizinho que ainda não foi explorado ou que não está na fila, retiramos G da lista e partimos para o nó I. O tratamos e marcamos como tratado e o retiramos da fila. Dessa forma a fila se encontra vazia, Fila = , então a busca se encerra, pois todos os vértices foram explorados e tratados