Universidade Federal de Minas Gerais
DCC- Departamento de Ciência da Computação
Engenharia de Sistemas

**PEDRO CAROLINO MALAQUIAS - 2016070115**

# TRABALHO PRÁTICO 0
# ALGORITMOS E ESTRUTUDAS DE DADOS 3

**BELO HORIZONTE**

**2018**

# 1 INTRODUÇÃO

O objetivo desse trabalho é a resolução do problema de um compartilhamento de uma música, onde pessoas que tenham abaixo de 35 anos e escutam a música, gostam da música e automaticamente começam a compartilhar a música com todos os seus familiares. Mas a música tem um problema, e pessoas que tem 35 anos ou mais, escutam a música e não gostam, sendo assim elas não fazem o compartilhamento dela com os seus familiares. Portanto o objetivo desse trabalho fica sendo, ajudar a contabilizar quantas pessoas gostam e compartilham a música. Sendo que as únicas informações disponíveis são as idades das pessoas, a relação dela com os seus familiares, e a primeira pessoa desse ciclo social a escutar a música.

Problemas bem parecidos com esses são resolvidos pelo Facebook, para divulgação de eventos, vídeos e até mesmo publicidades. Tendo em vista que, a probabilidade de você ter interesse de ir à festa que os seus amigos iram é alta.

Desta forma, cheguei à conclusão que a melhor forma de resolução desse problema, tendo em vista os meus conhecimentos prévios, seria criar uma matriz quadrática (nxn), onde n é o número de pessoas, para verificar as relações entre as pessoas que ouviam e os seus familiares, criar também um vetor de idades para que pudéssemos verificar se a pessoa em questão compartilha ou não a música, e criar uma pilha para que pudéssemos ter o controle de quantas pessoas compartilharam no final a música.

## 2 MODELAGEM DO PROBLEMA

A visualização da gráfica do problema pode ser vista como um grafo não direcional, onde cada vértice representa uma pessoa, e cada aresta representa o elo dela para os seus familiares. Está aresta contém duas informações de extrema importância que são, a idade de quem irá escutar a música, e a sua relação com os seus familiares. Já o vértice contém somente a informação do id da pessoa em questão.

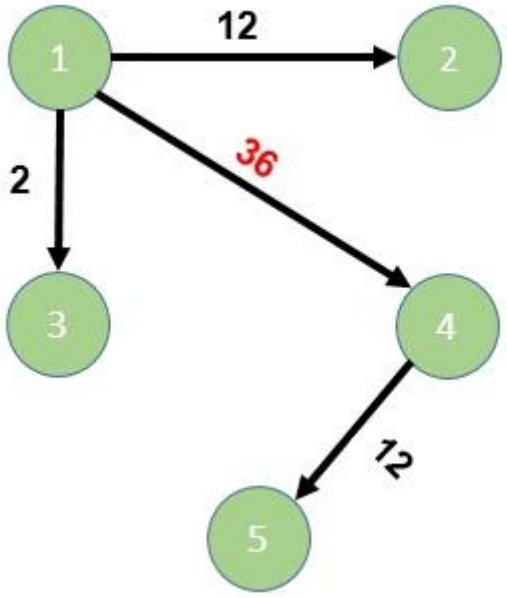


*Figura 2-1: Representação do problema por meio de Grafo(V,E)*

Para montarmos o grafo acima, utilizamos a ideia de criarmos uma Grafo de Matriz de Adjacência, mesmo o grafo sendo esparso, e realizarmos um método de busca recursiva dentro desse grafo para percorrermos somente aquelas posições desejadas, ou o mais próximo disso possível.

| Matriz de Relações entre familiares | | | | | | |
|---|---|---|---|---|---|---|
| nxn | 0 | 1 | 2 | 3 | 4 | → Linha |
| 0 | 0 | 1 | 1 | 1 | 0 | |
| 1 | 1 | 0 | 0 | 0 | 0 | |
| 2 | 1 | 0 | 0 | 1 | 0 | |
| 3 | 1 | 0 | 1 | 0 | 1 | |
| 4 | 0 | 0 | 0 | 1 | 0 | |
| ↓ Coluna | | | | | | |

*Figura 2-2: Representação da Matriz de Adjacência (nxn)*

Essa Matriz de Adjacência é quadrática, de tamanho (nxn), onde n é a quantidade de pessoas ou de vértices que o grafo terá. A Coluna da matriz representa os vértices da pessoa que escutou a música, e a Linha representa as relações com os seus familiares.

Como o grafo é não direcional, podemos perceber que sua matriz é simétrica, e tem função de armazenar as relações entre as pessoas e os seus familiares, ou seja, se for 1 a pessoa tem relação, se for 0 a pessoa não tem relação.

```
matrizrelacao = (int**)calloc(num_people,sizeof(int*)); //Criacao do grafo em memoria dinamica(Matriz adjacente de 2 dimensoes)
for(i = 0; i < num_people; i++)
{
    matrizrelacao[i] = (int*)calloc(num_people,sizeof(int));//alocação de uma coluna completa para o grafo; Todo o conteudo
cia com 0, por causa do calloc.
}


//Leitura da id e idade, feita numero de pessoas vezes.
for(i = 0; i < num_people; i++)
{
    scanf("%d %d", &id, &idade);

    vet_id[i] = id - 1; //Armazenamento das id no vetor, utiliza-se id-1 para as id seguirem em ordem as posicoes de memoria
vetor
    vet_idade[vet_id[i]] = idade; //Armazenamento das idades no vetor
    //printf("teste = %d\n", vet_idade[vet_id[i]]);
}

//Leitura das relacoes entre pessoas, feita numero de relacoes vezes
//Alem do preenchimento da matriz em relacao aos pesos(idade)
for(i = 0; i < num_relansh; i++)
{
    scanf("%d %d", &v1, &v2);

    matrizrelacao[v1-1][v2-1] = TRUE; //Preenchimento do grafo nao-direcional
    matrizrelacao[v2-1][v1-1] = TRUE;
}
```

*Figura 2-3:Código da Criação da Matriz de Relação e Alocação do Conteúdo*

Para buscarmos o conteúdo dessa matriz, utilizamos a ideia do algoritmo de Busca em Profundidade, esse algoritmo explora arestas partindo do vértice mais recentemente descoberto do qual ainda saem arestas inexploradas. Esse algoritmo como os demais algoritmos de busca em grafo, vão pintando os vértices durante sua busca, mas nesse trabalho essa condição não foi realizada, em vez de pintarmos os vértices, fizemos algo bem semelhante só que utilizando um vetor, se o vértice já tinha sido visitado armazenamos 1 na posição desejada no vetor de repetição, sabendo assim, se o algoritmo já tinha contado esse vértice ou não.

```c
//Funcao que verifica se a pessoa ja ouviu a musica ou nao
//Se ela já tive ouvido a musica retorna 1, se nao retorna 0
int verifica(int pessoa_inicial, int *vet_relacao)
{
    if(vet_relacao[pessoa_inicial] != 0)
    {
        //printf("pessoa ja foi visitada\n");
        return 1;//RETORNAR TRUE QUER DIZER QUE ESSA PESSOA JA FOI CONTADA
    }
    else
    {
        //printf("pessoa ainda nao foi visitada \n");
        vet_relacao[pessoa_inicial] = 1;
        return 0;//RETORNAR FALSE QUER DIZER QUE ESSA PESSOA NAO FOI CONTADA AINDA
    }
}
```

*Figura 2-4: Código da Função de Verificação*


*Figura 2-2-5: Representação do vetor de Repetição*

Para verificarmos se a pessoa compartilhava ou não a música, foi criado um outro vetor, um vetor de idade, para que pudéssemos realizar juntamente com o vetor de repetição, restrições para busca em profundidade e somente contarmos as pessoas que realmente compartilharam a música.


*Figura 2-6: Representação do Vetor de Idade*

```c
void DFS(int num_pessoas,int linha,int pessoa_inicial, int **matriz, int *vet_idade, int *vet_relacao, pilha *pilha)
{
    int j;
    vet_relacao[linha] = 1;

    for(j = 0; j < num_pessoas; j++)
    {
        //printf("matriz[%d][%d] = %d\n",linha, j , matriz[linha][j]);
        if(vet_idade[j] < 35 && matriz[linha][j] == 1)
        {
            if(vet_relacao[j] == 0)
            {
                //printf("Deu bom\n");
                //printf("matriz[%d][%d] = %d\n",linha, j , matriz[linha][j]);
                empilhaElemento(linha,pilha);
                DFS(num_pessoas,j,pessoa_inicial, matriz, vet_idade, vet_relacao,pilha); //Busca em Profundidade
            }
        }
    }
}
```

*Figura 2-7:Código da Busca Em Profundidade*

Para a contagem das pessoas que compartilhavam a música, foi criada uma pilha, e cada pessoa que compartilhava era empilhado, com o propósito de que ao final da busca em profundidade pudéssemos retornar quantas pessoas teriam compartilhado a música, por meio de saber o tamanho da pilha.

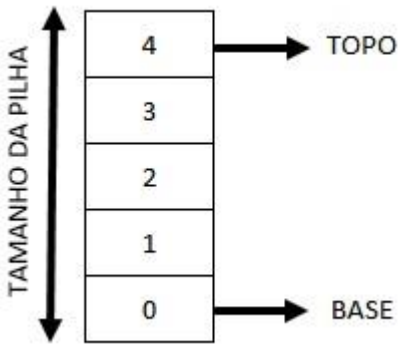


*Figura 2-8: Representação da Pilha de Pessoas*

```c
typedef struct nodo { //crio a estrutura do nodo da pilha, que contém o valor, o ponteiro para a próxima posição e para a posição anterior.
    int valor;
    struct nodo *proximo;
    struct nodo *anterior;
}nodo;

typedef struct pilha { //aqui, crio a estrutura da pilha, que contém um apontador para sua base e topo e um inteiro para medir.
    nodo *base;
    nodo *topo;
    int tamanho;
}pilha;

pilha *criaPilha();

void empilhaElemento(int elemento, pilha *p);

void LiberaPilha(pilha *p);
```

*Figura 2-9: Código do .h da Pilha*

```c
pilha *criaPilha()
{ //a função retorna uma pilha criada e vazia.
    pilha *p;
    p = (pilha*)(malloc(sizeof(pilha))); //aloco espaço na memória dinâmica para a estrutura da pilha.
    p->base = NULL; // direciono os ponteiros dela para NULL
    p->topo = NULL;
    p->tamanho = 1;

    return p; //retorno a pilha.
}

void empilhaElemento(int elemento, pilha *p){ //a função empilha o elemento dado na pilha. é do tipo void pois não precisa de retorno.
    nodo *n; //crio e aloco um nodo novo para a pilha.
    n = (nodo*)(malloc(sizeof(nodo)));
    n->valor = elemento; //o valor do nodo será o valor do elemento enviado na função. também direciono os ponteiros dele para NULL.
    n->proximo = NULL;
    n->anterior = NULL;

    if (p->base == NULL) { //caso a pilha estiver vazia, este nodo será sua base e, também, seu topo.
        p->base = n;
        p->topo = n;
    } else {
        n->anterior = p->topo; //caso tenha elementos na pilha, este nodo será organizado de forma a ser o novo topo.
        p->topo->proximo = n;
        p->topo = n;
    }

    p->tamanho++;
    //printf("tamanho da pilha = %d\n", p->tamanho);
    //printf("print pilha = %d\n",p->topo->valor);
}
```

*Figura 2-10: Código do .o da Pilha*

# 3 ANÁLISE TEORICA DO CUSTO ASSINTÓTICO

Nesta seção analisaremos os custos teóricos de tempo e espaço do algoritmo, tanto para a matriz de adjacência, quanto para a busca em profundidade e pilha.

## 3.1 ANÁLISE TEORICA DO CUSTO ASSINTOTICO DE TEMPO

- Matriz Adjacência: Para o algoritmo de Grafo em Matriz de Adjacência, avaliando todas os vértices e arestas, para realizar a sua criação e o seu preenchimento é necessário percorrer, o número de vértices (pessoas) vezes para criação da matriz, e para o seu preenchimento é necessário percorrer o número de arestas (relações) vezes. Sendo assim sua ordem de complexidade fica sendo: O(V + E).

- Busca em Profundidade: Para algoritmo de busca em profundidade DFS, se faz necessário um laço para o vértice percorrer as suas relações, e cada vez em que o vértice encontra uma relação ele chama a função de busca novamente para aquele outro vértice. Portanto no pior caso onde todos tem relação com todos, a sua ordem de complexidade fica sendo O(V + E), pois assim você irá percorrer todas as situações possíveis.

- Pilha: Para empilhar um inteiro em uma pilha, não é necessário realizar muitas operações, tendo em vista que, você so vai alterando o seu ponteiro de topo, portanto sua ordem de complexidade fica sendo O(1), para cada vértice que compartilhou a musica.

Desta forma conclui-se que a complexidade do algoritmo total fica sendo O(2*(V + E) + 1*C), onde C é a quantidade de pessoas que compartilharam a música.

## 3.2 ANÁLISE TEORICA DO CUSTO ASSINTOTICO DO ESPACO

- Matriz de Adjacência: Como o algoritmo se trata de uma matriz, e uma matriz é de tamanho nxm, onde m pode ou não ser igual a n, no caso é, se tornando uma matriz quadrática, o seu espaço necessário é de $O(n^2)$.
- Busca em Profundidade: Para realizarmos o algoritmo de Busca em profundidade foi necessário a criação de 2 vetores de tamanho n, para realizarmos as restrições do problema, sendo esses vetores, o vetor de idade e o vetor de visita, portanto o espaço necessário é de O(2*n).
- Pilha: O algoritmo de empilhar, ele vai alocando em posições descontiguas de memoria, portanto ele vai alocando cada vez que ele empilha, como no algoritmo de busca em profundidade so empilhamos as pessoas que compartilham a musica, o espaço gasto por esse algoritmo é a quantidade de pessoas empilhadas, O(1*P).

Desta forma, a ordem de complexidade de espaço do algoritmo é dada por:
$O(n^2 + 2*n + P)$.

## 4 ANÁLISE DOS EXPERIMENTOS

A análise experimental da implementação pode ser visualizada pela Tabela 1. Para realizar o experimento foi gerado um arquivo MAKEFILE onde, ele já faz a compilação com o atalho, digitado no terminal, “make tp0”, faz a limpeza do compilado com “make clean”, e também executa dos os testes com o atalho, “make testes”.

Para medir o tempo de execução do código, foi utilizada a biblioteca "<time.h>" para contar o número de clocks necessários para executar o código. Cada instancia foi gerada 4 vezes e foi retirada a média para obtermos um tempo com maior confiabilidade e precisão. Tendo em vista que os testes foram realizados em um computador com processador Core i5-7200 8GB de RAM e 2.71GHz.

| Tabela de Resultados do Algoritmo | | | | |
|---|---|---|---|---|
| Testes | Quantidade de Pessoas | Quantidade de Relações | Resultados | Tempo Médio(s) |
| Teste 1 | 5 | 5 | 3 | 0.22 |
| Teste 2 | 7 | 9 | 3 | 0.22 |
| Teste 3 | 100 | 1000 | 46 | 0.27 |
| Teste 4 | 500 | 7500 | 190 | 1.25 |
| Teste 5 | 1000 | 25000 | 369 | 0.32 |
| Teste 6 | 2000 | 400000 | 696 | 1.61 |
| Teste 7 | 5000 | 1000000 | 1737 | 4.26 |
| Teste 8 | 7500 | 1500000 | 2593 | 6.61 |
| Teste 9 | 10000 | 2000000 | 3455 | 9.71 |
| Teste 10 | 20000 | 8000000 | 6877 | 36.9 |

*Tabela 1: Tabela de Resultados do Algoritmo*

A Tabela 1, fornece o tempo médio do algoritmo em relação a quantidade de pessoas e a quantidade de relação dos testes feitos, percebemos que quanto maiores as entradas, maior o tempo do algoritmo, o que é totalmente coeso com o seu comportamento assintótico. Para explicar o crescimento praticamente linear do tempo, podemos perceber que tem muito a ver com a quantidade de relações que a pessoa tem, pois quanto mais relações ela tem, maiores quantidades de buscas em profundidade se fazem necessárias, o que aumenta o tempo do algoritmo.

## 5 CONCLUSÃO

Nesse trabalho, foi realizado o algoritmo de grafo de matriz de adjacência, juntamente com uma busca em profundidade e o empilhamento de pessoas, para a resolução do problema imposto, que era o de verificar quantas pessoas em um grupo familiar compartilhariam uma música.

Os resultados foram satisfatórios, tendo em vista que, todos os testes realizados obtivemos o resultado esperado.

O que se pode notar visualizando os tempos obtidos, foi que dependendo de como era dada a entrada e suas relações familiares o algoritmo demorava a mais para encontrar o resultado esperado, tendo em vista que, teríamos que fazer mais buscas em profundidades, o que é totalmente compreensível tendo em vista a analise assintótica do algoritmo de busca em profundidade.