UNIVERSIDADE FEDERAL DE MINAS GERAIS

DCC005 - ALGORITMOS E ESTRUTURAS DE DADOS III

HERNANE BRAGA PEREIRA

2014112627

TRABALHO PRÁTICO - 0

Belo Horizonte - MG
Abril/2018

1.1 Introdução

O objetivo deste trabalho é a resolução do problema do DJ Vitor, que deseja descobrir quantas pessoas gostaram de sua nova música, sendo que apenas pessoas com menos de 35 anos se interessam pela música e irão compartilha-la. Para ajuda-lo recebe-se uma lista com as idades de cada pessoa, as relações familiares dessas pessoas, e a primeira pessoa a ouvir a nova composição de Vitor.

A partir dos dados fornecidos e sabendo a primeira pessoa a ouvir a música, deve-se informar quantas pessoas desta rede gostaram da música.

2.1 Solução do Problema

Para solucionar o problema proposto, foi utilizada a estrutura de grafos, pois ela é a mais adequada para representar relacionamento entre pessoas, ou entidade. Na estrutura pensada, os vértices representam as pessoas da rede, enquanto que as arestas representam a conexão entre elas. Para escolher entre implementar um grafo por lista de adjacências, ou matriz de adjacências, foi realizada uma análise com os casos de testes fornecidos. A tabela abaixo compara o número de vértices, arestas e pessoas que gostaram da música, dos casos de teste um a dez.

Tabela 1: (V) - Número de vértices (A) - Número de arestas (T) – Arquivo caso de teste

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
V	5	7	100	500	1000	2000	5k	7500	10k	20k
A	5	9	900	7275	24375	360k	960k	1460k	1960k	7840k
< 35	4	5	46	190	369	696	1737	2593	3455	6877

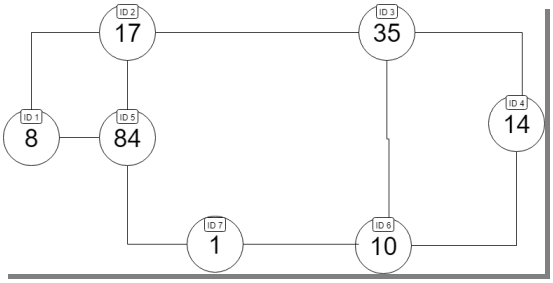
A partir da tabela acima, tira-se duas conclusões importantes:

1. O número de vértices mais arestas, é muito menor que o de vértices ao quadrado (V+A) << V^2. Portanto, o grafo é mais *esparso* e a implementação utilizando lista de adjacências é mais apropriada ao problema.
2. O número de pessoas com idade menor que 35 anos é consideravelmente menor que o número de vértices existentes.

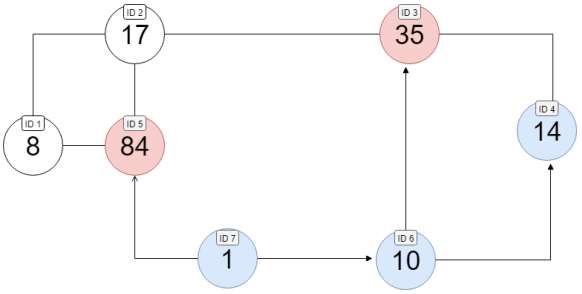
2.1.1 Modelagem do Problema

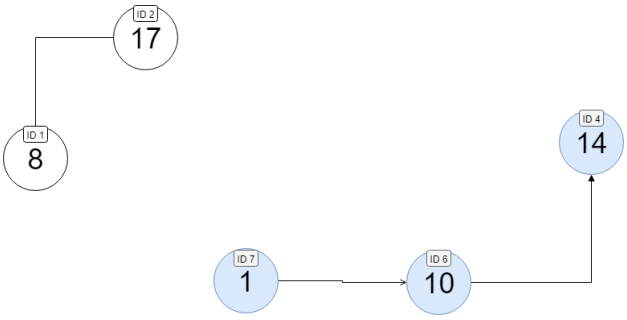
Abaixo está uma representação do caso de teste 2 (arquivo *teste_2.txt*). A partir dela, é possível notar que como as pessoas com mais de 35 anos de idade, não compartilham a música, é possível ignorar sua existência na construção do grafo, reduzindo assim o custo de memória alocada no grafo.

1. Representação do arquivo entrada teste_2.txt



2. Pessoa 7 começa ouvindo a música e os que estão em azul gostam





3. Representação do grafo sem as pessoas com idade > 35

Para criar o grafo apenas com pessoas com idade menor que 35 anos de idade, lê-se o arquivo de entrada até o momento em que todas as pessoas, com respectivas idades, são informadas, enquanto é contado o número de pessoas com idade menor que 35 anos. Feito isso, o grafo é criado com o número de vértices

igual ao número de pessoas que atendem o critério de gostar da música. Em seguida, a função *fseek* e retorna para a posição zero do arquivo e leitura é reiniciada, porém adicionando as pessoas com idade menor que 35 anos ao grafo.

```
//Verificar quantas pessoas tem idade < 35
//Assim criamos o grafo com a quantidade minima de espaco
for( i = 0 ; i < vertices ; i++){
    fscanf(arquivo,"%d %d", &id, &idade);
    if( idade < 35){
        vertices_validos++;
    }
}

//Voltar para o inicio do arquivo e comeca a ler-lo novamente.
fseek(arquivo, 0, SEEK_SET);
fscanf(arquivo,"%d %d", &vertices, &arestas);

//Releitura dos vertices
Graph graph = cria_grafo(vertices_validos);

for( aux = i = 0 ; i < vertices ; i++){
    fscanf(arquivo,"%d %d", &id, &idade);

    //Verificacao de quais pessoas tem idade < 35 e add ao grafo
    if( idade < 35 ){
        add_id_idade( graph, aux, id, idade);
        aux ++;
    }
}
```

Código 2: Verificação e adição de nova pessoa ao grafo

Em seguida, inicia-se a descrição do relacionamento entre as pessoas, dado por uma entrada *id1 id2* onde cada id é o número único de cada pessoa. Para realizar esta adição de arestas ao grafo, foi implementada a função *int verifica_id* que dado um id, ele retorna qual é o índice do vértice, associado ao identificador. O algoritmo realiza uma busca sequencial, partindo de um índice 0 até o número total de vértices, no pior caso.

A estrutura *grafo* é composta por um inteiro *V* que representa o número total de vértices existentes e por um campo *array*, que é do tipo lista encadeada. A lista encadeada por sua vez, possui os campos *id_pessoa, idade, visitado e grau* em sua

estrutura, onde *visitado* informa se aquele vértice já foi visitado na busca e *grau* diz quantas ligações aquele vértice possui.

```
    for ( i=0; i < arestas; i++){
        fscanf(arquivo,"%d %d", &id1, &id2);

        // Verifica se id1 existe no grafo
        indx_id1 = verifica_id( graph, id1);
        if ( indx_id1 >= 0){

            // Verifica se id2 existe no grafo
            indx_id2 = verifica_id( graph, id2);
            if ( indx_id2 >= 0 && indx_id1 >= 0 ){

                //Caso passe nas duas verificacoes, aresta e criada.
                add_aresta( graph, indx_id1, indx_id2);
            }
        }
    }
```

Código 2: Adição de nova aresta, com dupla verificação de id, poupando 1 chamada do algoritmo, caso o id 1 seja inválido e se repita sequencialmente.

```
// Verifica se o id informado e valido, e se sim,
// retorna seu indx de vertice
int verifica_id( Graph graph, int id_pessoa){
    int i;
    for ( i = 0; i < graph->V; i++){
        if ( graph->array[i].id_pessoa == id_pessoa)
            return i;
    }
    return -1;
}
```

Código 3: Implementação da função *verifica_id*

Após o grafo estar completo, com todas os vértices e arestas implementados, a função *conta_musica* faz uma busca no grafo à partir de um id informado, e conta quantas pessoas gostaram da música. A função é baseada no método de busca em profundidade (*DFS)* e é implementada utilizando chamadas recursivas.

```c
/*
Conta quantas pessoas gostaram da musica, a partir de um index (indx) de um
vertice
E pedido um contado (*cont) para contar quantas pessoas gostaram da musica
O parametro aux deve ser enviado com -1 por default
*/
void conta_musica( Graph graph, int indx, int *cont, int aux){
    int i;
    // Realiza i verificacoes, baseado no grau do vertice
    for ( i = 0; i < graph->array[indx].grau; i++)
    {
        // pCrawl recebe o valor do index da celula
       // visitada dentro da lista encadeada
        struct AdjListNode* pCrawl = graph->array[indx].head;
        graph->array[indx].visitado = 1; //index eh marcado como visitado

      // Loop ocorre enquanto nao puder apontar para mais
      // nada ( pCrawl != NULL )
        while (pCrawl)
        {
            aux = pCrawl->dest;
            if ( graph->array[aux].visitado == -1){
                conta_musica(graph, aux, cont, aux);
            }
            else{
                pCrawl = pCrawl->next;
            }
        }
    }
    cont[0] +=1;
}
```

Código 4: Implementação da função *conta_musica*

3.1 Análise Teórica do Custo Assintótico de Tempo

Podemos realizar a análise de custo assintótico avaliando cada um dos algoritmos apresentados:

- O código 3 (*verifica_id)* possui complexidade 2 * O(V), onde V é o número de vértices do grafo e para cada aresta A, é necessário realizar 2 chamadas do

algoritmo. Foi implementada uma verificação em cascata, para o caso de um id que não exista no grafo, se repita sequencialmente após uma chamada do algoritmo, esta verificação, quando atendida, economiza uma chamada da função. Durante toda a execução do trabalho, esta função será chamada pelo menos 2 * A * O(V). A complexidade aumentará linearmente caso os ids estiverem desordenados, pois assim a verificação em cascata não será tão efetiva, tornando a complexidade 2 * A * O(V) + D * O(V), onde D é o número de chamadas que não foram evitadas pela verificação em cascata.

- O código 4 (*conta_musica)* passará em todos os vértices e arestas que estão conectadas entre si, e marca os vértices já visitados. Sua complexidade total é de O(V+A).

Dessa forma, conclui-se que o custo total de tempo do algoritmo é 2 * A * O(V) + O(V+A) + D * O(V)

3.2 Análise Teórica do Custo Assintótico de Espaço

Para fazer a análise de custo de espaço, foca-se no tamanho do grafo criado, que por padrão é O(V+A). Devido ao pré-processamento das informações de entrada, explicitada em 2.1.1 e pelo *Código 1,* ocorre uma redução da complexidade de espaço, que passa de O(*V_entrada* + *A_entrada)* para O(*V_idade_valida* + *A_conexoes_validas*).

4.1 Testes do código

A máquina utilizada para rodar os testes possui 6,0 GB de memória RAM, com processador Core™ i7-5500U CPU 2.40Hz. Foi utilizado o comando *time ./tp0* <

teste_x.txt para contabilizar o tempo gasto pelo programa. Os resultados da tabela abaixo e do gráfico são a média de execuções do programa para cada caso de teste.

Tabela 3: Média de tempo gasto em cada arquivo de teste

teste_1.txt: 0.522 (s)	teste_6.txt: 0.940 (s)
teste_2.txt: 0.608 (s)	teste_7.txt: 2.013 (s)
teste_3.txt: 0.653 (s)	teste_8.txt: 3.315 (s)
teste_4.txt: 0.700 (s)	teste_9.txt: 5.309 (s)
teste_5.txt: 0.680 (s)	teste_10.txt: 39.181 (s)

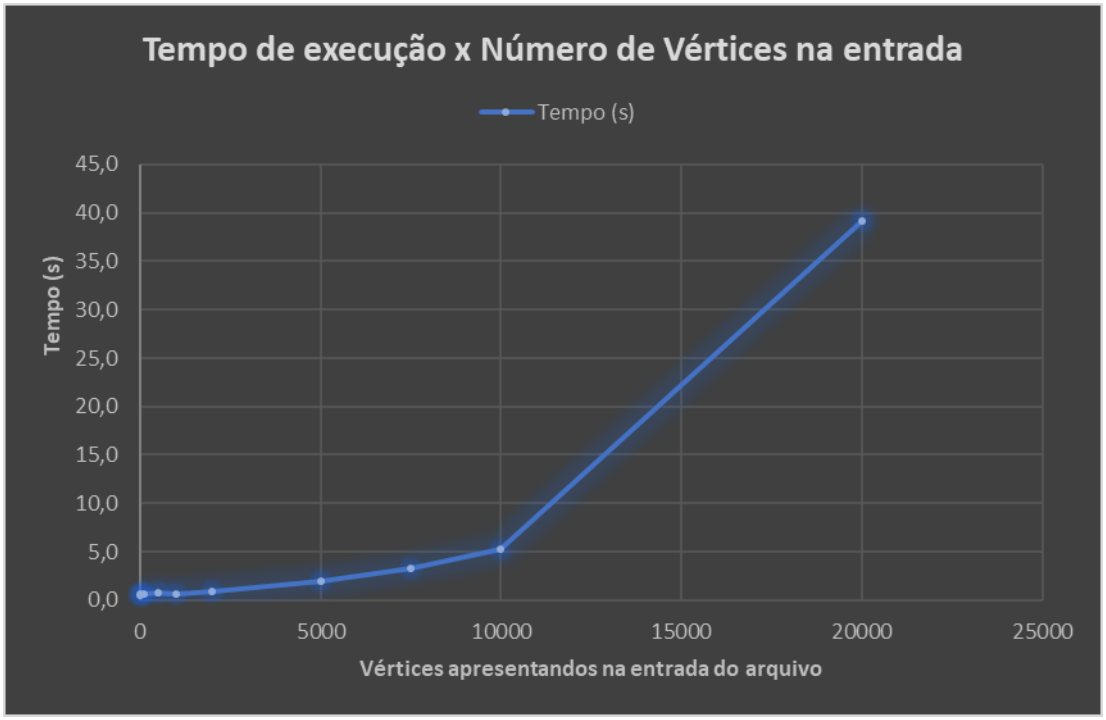


Gráfico 1: Tempo de execução do executável x Número de vértices fornecidos na entrada do arquivo

```
herna@DESKTOP-VOIMGII  /c/Users/herna/Desktop/hernane_braga_pereira_2014112627
$ make
gcc -lm -Wall -Wextra -std=c99 -g -pedantic -O2 grafo.h grafo.c main.c -lm -o tp0

herna@DESKTOP-VOIMGII  /c/Users/herna/Desktop/hernane_braga_pereira_2014112627
$ time ./tp0 < teste_1.txt
3
real    0m0.546s
user    0m0.000s
sys     0m0.015s

herna@DESKTOP-VOIMGII  /c/Users/herna/Desktop/hernane_braga_pereira_2014112627
$ time ./tp0 < teste_2.txt
3
real    0m0.678s
user    0m0.015s
sys     0m0.000s
```

Figura 4: Compilação do código e forma utilizada para cálculo de tempo gasto. Arquivos de teste 1 e 2.

```
herna@DESKTOP-VOIMGII  /c/Users/herna/Desktop/hernane_braga_pereira_2014112627
$ time ./tp0 < teste_8.txt
2593
real    0m3.423s
user    0m0.015s
sys     0m0.000s

herna@DESKTOP-VOIMGII  /c/Users/herna/Desktop/hernane_braga_pereira_2014112627
$ time ./tp0 < teste_9.txt
3455
real    0m5.405s
user    0m0.000s
sys     0m0.030s

herna@DESKTOP-VOIMGII  /c/Users/herna/Desktop/hernane_braga_pereira_2014112627
$ time ./tp0 < teste_10.txt
6877
real    0m40.711s
user    0m0.000s
sys     0m0.015s
```

Figura 5: Tempo gasto na execução dos arquivos de teste 8, 9, 10.

A partir do gráfico é possível notar que o programa tem tempo de execução linear baseado no número de vértices (V), o que comprova a análise teórica do item 3 desta documentação.

5.1 Conclusão

Neste trabalho, foi resolvido o problema de descobrir quantas pessoas gostaram da música do DJ Vitor. O problema foi resolvido modelando-o como grafo e aplicando um pré-processamento nos dados de entrada e utilizando um algoritmo baseado em DFS para contar quantas pessoas gostaram da música. A análise de complexidade teórica de tempo foi comprovada através de experimentos que utilizaram entradas grandes suficientes para analisar o comportamento assintótico. A solução apresentada poderia ser otimizada, utilizando-se uma busca binária, outro método de busca na função *verifica_id,* o que possibilitaria reduzir a complexidade de O(V) para O(log n).

REFERÊNCIAS

Ziviani, Nívio; Projeto de Algoritmos. 4ed. São Paulo: Pioneira. 1999

Algoritmos para grafos, IME-USP:
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphdatastructs.html#GRAPHinit-lists

Programação descomplicada em C, playlist do YouTube:
https://www.youtube.com/watch?v=gJvSmrxekDo&list=PL8iN9FQ7_jt4oQHq1TSeMgvVdolJvYsLO

Geeks for Geeks, Graph data structure and algorithms
https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/