Trabalho Prático 0

O Hit do Verão

Algoritmos e Estrutura de Dados III

Álvaro Augusto Alves de Carvalho

2015430231

1 Introdução

O objetivo deste trabalho prático é a resolução do problema proposto entitulado “O Hit do Verão”. O problema consiste em determinar o alcance de um novo “Hit” do verão de 2020, assumindo que a música será curtida e compartilhada por qualquer pessoa com menos de 35 anos de idade, e ela será compartilhada **somente** entre os familiares desta pessoa. Desta maneira, o objetivo é calcular quantas pessoas **gostaram** da música composta. Para resolver isto, são dadas como informações o número total de pessoas a ser considerado, a idade de cada pessoa, as relações familiares entre cada pessoa, e a primeira pessoa da amostra a ter ouvido a música encantadora.

Com isto em mente, e utilizando conceitos de estruturas de dados aprendidos nas disciplinas de Algoritmos e Estrutura de Dados I, II e III, chegou-se na conclusão de que a melhor maneira de abordar o problema seria criar um grafo, onde os vértices são as pessoas, e as arestas são as relações entre elas. Utilizando algoritmos conhecidos de busca para percorrer um grafo, com algumas simples adaptações, podemos encontrar a resposta para o problema proposto.

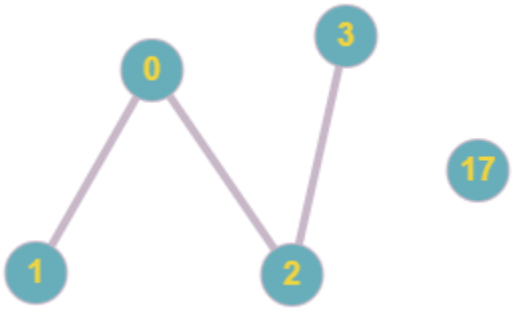

Figura 1: Exemplo de um grafo. supondo que a pessoa “0” é pai das pessoas “1” e “2”, “3” é um irmão perdido de “2” de outra família que desapareceu, e “17” é apenas um amigo de “1” (sem relação familiar), temos o grafo acima elaborado.

2 Modelagem do Problema e Implementação

Utilizando um grafo para representar os dados fornecidos, junto com algoritmos conhecidos para percorrer o mesmo, podemos determinar o alcance da música simplesmente verificando todas as conexões diretas e indiretas com o vértice que representa a pessoa que ouviu a música pela primeira vez. Podemos ver pela figura 1 que nem todas as pessoas podem estar relacionadas com a pessoa que primeiramente ouviu a música, direta ou indiretamente, e portanto a música não chegaria a elas, e nem compartilhada com seus familiares.

Além disso, nem todas as pessoas **gostam** do hit, e portanto não o compartilhariam e nem entrariam para o cálculo da resposta. Para essas pessoas, o grafo também estaria efetivamente "desconectado".

O problema proposto constitui em um grafo esparso, ou seja, com uma pouca quantidade de arestas em relação ao máximo número de arestas que um grafo poderia ter. Desta forma, a melhor estrutura de dados para representá-lo é uma lista de adjacências, representada na figura 2.

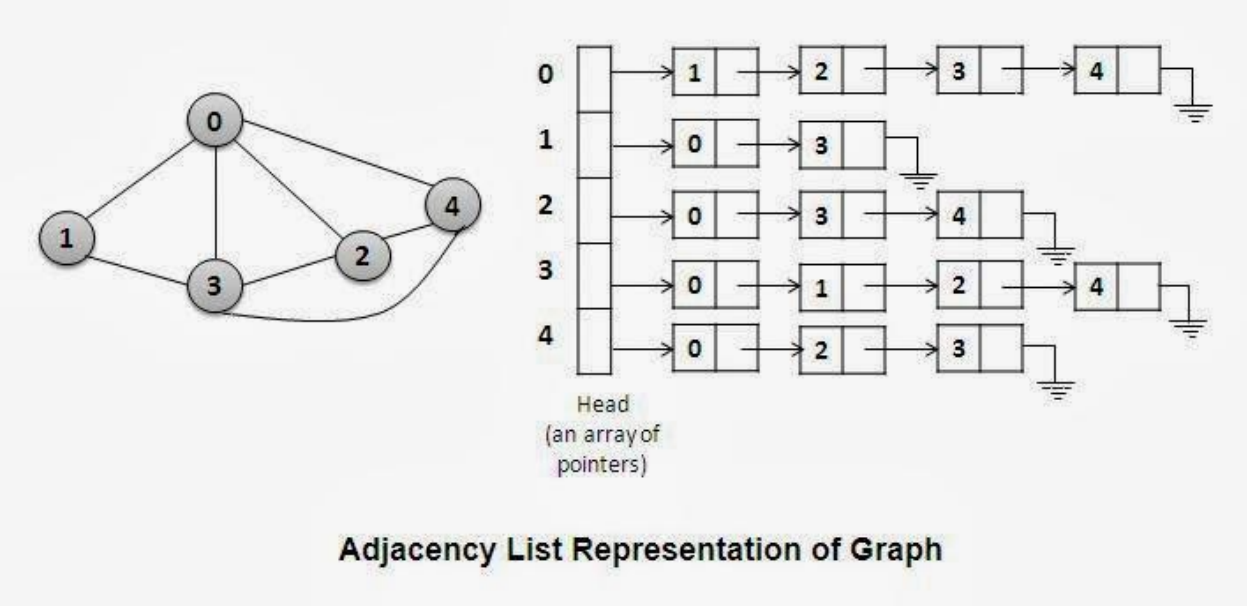

Figura 2: Representação de Grafo através de Lista de Adjacências

A lista de adjacências, apesar de possuir uma implementação conceitualmente mais complicada que a sua alternativa (matriz de adjacências), possui uma grande vantagem para tratar de grafos esparsos, em termos de uso de memória.

Antes de implementar o grafo, no entanto, deve ser considerado o problema que pessoas com 35 anos ou mais não gostam nem compartilham a música, ou seja, efetivamente não se

relacionam e nem precisariam ser parte do grafo para o objetivo deste trabalho. Desta forma, foi implementado um vetor para armazenar os ID's das pessoas que de fato gostarão e compartilharão o hit, descartando aquelas com idade maior ou igual à 35 anos.

```
for each ID { // ID da pessoa
    if (idade < IDADE_MAX) {
        pessoasQueGostam[ID] = true;
    }
    else {
        pessoasQueGostam[ID] = false;
    }
}
```

Pseudo-código da implementação do vetor de pessoas que gostam da música. Se a pessoa tiver menos que a idade máxima, então ela gostará da música. Caso contrário, ela não gostará.

No momento de receber como entrada as relações dos familiares, e com o vetor dos ID's das pessoas "compatíveis" em mãos, o programa primeiro realiza uma verificação. Só entrarão para o grafo as relações compostas por pessoas que gostarão e compartilharão a música.

```
for each Relationship {
    if (pessoasQueGostam[ID_1] && pessoasQueGostam[ID_2]) {
        acrescentaAresta(grafo, ID_1,ID_2);
    }
}
```

Pseudo-código da verificação de quem entrará no grafo com relacionamento. Para cada relacionamento dado, somente será adicionada a aresta no grafo caso ambos os ID's de pessoas estiverem como "Verdadeiro" no vetor de pessoas que gostam da música

Com estas considerações, o grafo implementado terá precisamente todas as pessoas que precisarão estar nele, ninguém a mais ou a menos, e para encontrar a solução do problema basta percorrer por todo o grafo a partir da primeira pessoa que ouvir a música, e contar quantas pessoas estão no caminho. Para isto, foi utilizado um algoritmo de busca em profundidade (Depth-First-Search).

A busca em profundidade é um algoritmo que data desde o século 19, utilizado por um matemático francês Charles Pierre Trémaux, como uma estratégia para solucionar labirintos. O algoritmo começa em um nó arbitrário do grafo e tenta explorar o grafo ao máximo ao longo dos ramos deste nó, antes de realizar um *backtracking*, ou seja, volta para o próximo candidato viável para continuar sua busca, ou exploração. Para isso, o algoritmo utiliza recursividade.

```
procedure DFS(G,v):
    v = visited;
    for all (arestas de v a w) in G.arestasAdjacentes(v) {
        if (vertice w != visited) {
            DFS(G,w); // recursividade
        }
    }
```

Pseudo-código: Algoritmo DFS iniciando em um vértice *v*.

Após rodar a busca em profundidade a partir do vértice especificado, basta percorrer por todo o vetor de vértices visitados e contar. O número de pessoas que gostaram do hit é igual ao número de vértices visitados pelo algoritmo DFS.

```
unsigned int curtiramOHit = 0;
for (i = 0; i<num_V; i++) {
    if (visited[i] == 1) {
        curtiramOHit++;
    }
}
return curtiramOHit;
}
```

Algoritmo que conta o número de pessoas que curtiram o hit do verão, a partir do vetor de visitados preenchido pela DFS.

3 Análise de Complexidade (teórica)

3.1 Análise de Complexidade de Tempo

Podemos realizar a análise de complexidade temporal baseando nas complexidades de cada algoritmo apresentado.

- Ambos os algoritmos usados para filtrar as pessoas com idade maior ou igual a 35 anos percorrem pelo número total de pessoas e tem uma complexidade fixa em O(N).

- O algoritmo DFS é um algoritmo de busca conhecido, que possui uma complexidade linear O(V+E), sendo **V = N** e **E = número de relações familiares** (desconsiderando as relações com pessoas acima de 35 anos). Neste trabalho a única possível alteração no DFS tradicional foi não imprimir o resultado da busca no final, o que não impacta na sua complexidade temporal.

- Os outros algoritmos envolvidos na implementação são pertinentes à implementação do grafo em si, utilizando lista de adjacências. O algoritmo que vale ser considerado para complexidade aqui é o de desalocação de memória do grafo, que percorre por todas as listas de adjacência relativas a cada vértice e, no pior caso, possui uma complexidade temporal de O(V*E).

  Por fim, temos que a complexidade do programa em si, considerando os algoritmos de maior complexidade, é $O_{max}\left(O(V*E), O(V+E), O(V)\right) = O(V*E)$.

3.2 Análise de Complexidade Espacial

Para a análise de complexidade espacial, vamos abordar a estrutura de dados do grafo em si, assim como os algoritmos utilizados.

- A lista de adjacências é uma lista de listas, onde cada lista corresponde a um vértice *v* e contém uma lista de arestas (*u*,*v*), originando de *u*. Desta forma, a lista de adjacências ocupa um espaço O(V+E) (caso médio). **Podemos considerar esta complexidade espacial principalmente quando sabemos que o grafo é esparso.**
- O algoritmo de busca DFS possui complexidade espacial O(V), no pior caso em que ele armazena a pilha de vértices do caminho atual e o *set* de vértices já visitados.
- Os algoritmos para filtrar ocupam um espaço O(V), que é o tamanho do vetor a ser utilizado.

Por fim, temos uma complexidade espacial do programa, considerando as estruturas e algoritmos abordados: $O_{max}\left(O(V), O(V+E)\right) = O(V+E)$

4 Análise dos Experimentos

Para a análise experimental do trabalho, foi utilizado o comando *time* (para análise de complexidade temporal), e o *valgrind* (para análise do consumo de memória do programa). Foram utilizados os testes fornecidos pelo moodle da disciplina na análise. Todos os testes foram realizados em uma máquina virtual Ubuntu através do Oracle VM VirtualBox, com 4gb de RAM alocados. A máquina física dispõe de um processador i7-4700MQ @2.40GHz, com 16gb de RAM.

4.1 Análise Temporal

Para a análise temporal, cada teste foi executado 10 vezes utilizando o comando *time*. Ex.: time ./tp0 < teste_i.txt

Os valores obtidos seguem na tabela 1.

Teste / Execucoes	1	2	3	4	5	6	7	8	9	10
1	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
2	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
3	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
4	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
5	0.005	0.005	0.005	0.005	0.005	0.006	0.005	0.005	0.005	0.006
6	0.115	0.116	0.126	0.111	0.127	0.117	0.119	0.125	0.117	0.117
7	0.235	0.271	0.274	0.271	0.266	0.268	0.265	0.263	0.27	0.285
8	0.355	0.366	0.401	0.393	0.371	0.378	0.37	0.361	0.381	0.374
9	0.533	0.542	0.546	0.541	0.539	0.533	0.528	0.529	0.555	0.545
10	1.632	1.707	1.719	1.663	1.676	1.662	1.663	1.673	1.743	1.679

Tabela 1: Valores de tempo de execução obtido por teste, em um total de 10 execuções. Valores expressos em segundos.

A partir da média dos valores obtidos (tabela 2), e do número de vértices em cada teste, o gráfico da figura 3 foi construído.

Teste	Tempo Médio	Vertices
1	0.001	5
2	0.001	7
3	0.001	100
4	0.002	500
5	0.0052	1000
6	0.119	2000
7	0.2668	5000
8	0.375	7500
9	0.5391	10000
10	1.6817	20000

Tabela 2: Tempo médio de execução x número de vértices, por teste

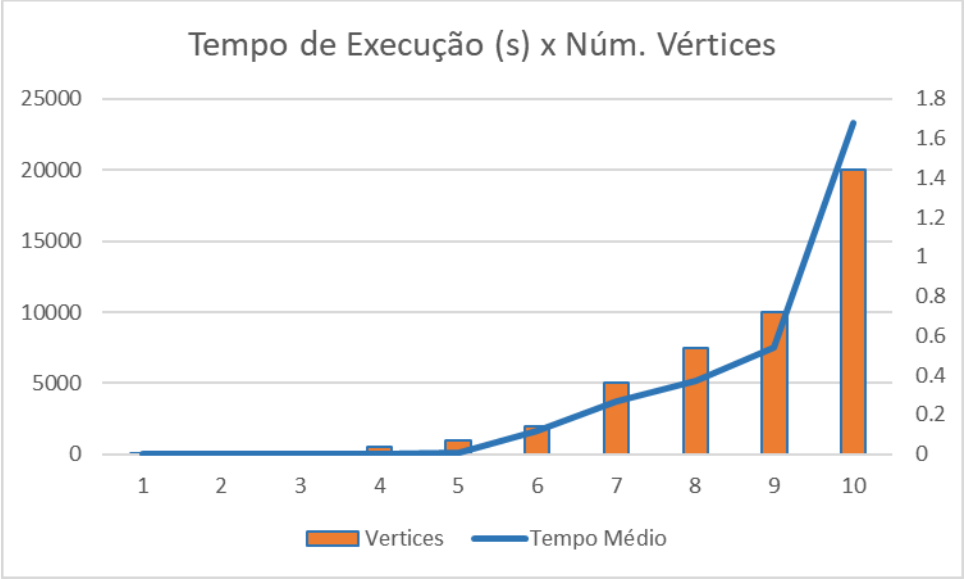

Figura 3: Gráfico de tempo de execução (em segundos) x Número de Vértices, por teste

Podemos ver que a complexidade do programa depende em grande parte do tamanho da entrada, ou seja, o número de vértices. Como verificado na parte teórica, o programa se comporta com uma relação diretamente proporcional ao número de vértices do problema.

4.2 Análise Espacial

Para a análise espacial foi utilizado o Valgrind para cada caso de teste apresentado, e foi criada uma tabela com os valores de memória utilizados em cada execução em relação ao número de vértices. Posteriormente, foi criada uma representação gráfica dos dados.

Testes	kBytes	Vértices
1	5.248	5
2	5.296	7
3	10.872	100
4	46.168	500
5	131.544	1000
6	1437.016	2000
7	3692.344	5000
8	5579.16	7500
9	7390.808	10000
10	30188.28	20000

Tabela 3: Espaço de memória utilizado (kBytes) x Número de Vértices, por teste

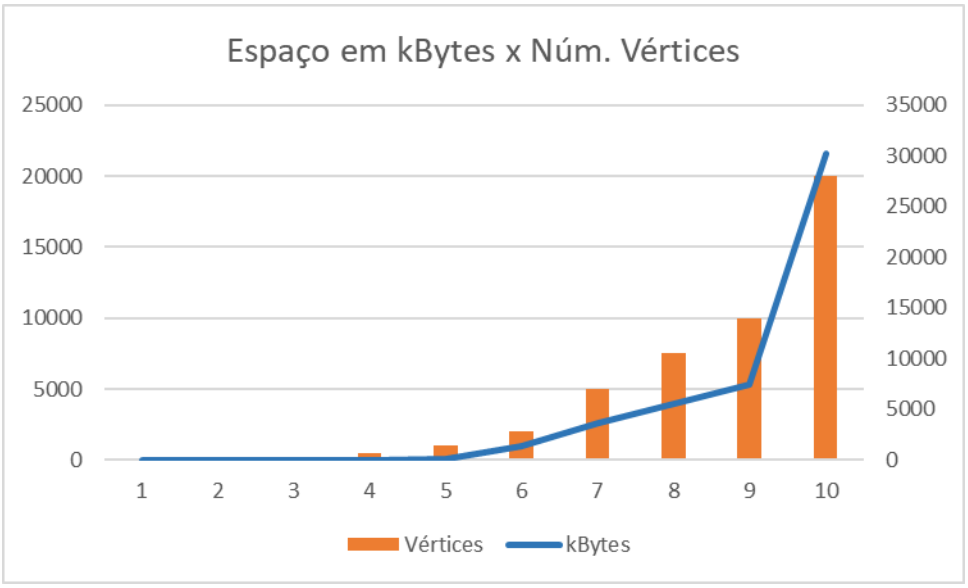

Figura 4: Gráfico de consumo de memória (kBytes) x Número de Vértices, por teste

Através dos dados apresentados, podemos verificar novamente que a complexidade espacial do programa se comporta em relação direta com o número de vértices no problema, seguindo a análise teórica.

5 Conclusão

O trabalho proposto teve como objetivo familiarizar o aluno de Algoritmos e Estrutura de Dados III com a matéria, que ainda se encontra no seu início. Além disso, teve o objetivo de testar alguns conhecimentos e noções básicas de estruturas de dados abordados na disciplina de Algoritmos e Estrutura de Dados II. Para a resolução do problema, foi implementada uma solução simples que depende principalmente do algoritmo de busca em profundidade (DFS), com algumas simples adaptações no momento de construção do grafo, para adequar-se ao problema apresentado.

Esta solução aqui apresentada, direcionada particularmente para o problema de idades das pessoas, foi bastante eficiente –considerando uma relação “simplicidade de implementação X eficiência de código”. Acredito que este foi um problema bastante direto, e a abordagem utilizada foi bastante adequada. As análises experimentais e teóricas, a princípio, também parecem condizentes com o problema proposto e a solução implementada.