Instituição: Universidade Federal de Minas Gerais
Disciplina: Algoritmos e Estruturas de Dados III
Professora: Olga Nikolaevna Goussevskaia
Aluna: Lorena Mendes Peixoto
Matrícula: 2017015002

Hit do Verão - Documentação

1. Introdução

Uma música agrada apenas às pessoas que têm menos que 35 anos. Tais pessoas a compartilham com outras dentro se seu ciclo de relacionamento, de modo que aquelas que possuem essa idade também a compartilham.

O presente trabalho propõe o desenvolvimento de um programa que calcule o número de pessoas que ouviram e gostaram da música, dada uma lista de pessoas envolvidas com suas respectivas idades, conforme mostrado no quadro 1.

Quadro 1 - Pessoas e idades

Pessoa 1 - 15 anos
Pessoa 2 - 26 anos
Pessoa 3 - 32 anos
Pessoa 4 - 40 anos
Pessoa 5 - 35 anos

Fonte: criado pela autora

O quadro 2 mostra uma segunda lista, que contém as relações entre as pessoas envolvidas. Essas relações são bilaterais, ou seja, se João se relaciona com Maria, então Maria se relaciona com João.

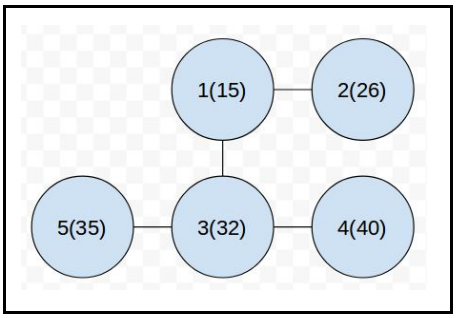
Quadro 2 - Relações entre pessoas

Pessoa 1 - Pessoa 2
Pessoa 1 - Pessoa 3
Pessoa 3 - Pessoa 4
Pessoa 3 - Pessoa 5

Fonte: criado pela autora

Um grafo é constituído de vértices e arestas. O problema pode ser representado por essa estrutura, de forma a facilitar a organização dos dados. A figura 1 mostra exatamente como seriam os quadros 1 e 2 relacionados em forma de grafo.

Figura 1 - Grafo de relações



Fonte: criado pela autora

2. Objetivos

Este trabalho tem por objetivo reforçar conceitos aprendidos em AEDS II - como listas encadeadas, ordenação e análise de complexidade -, bem como a introdução do uso de grafos e listas/matrizes de adjacência.

3. Desenvolvimento

Duas maneiras mais comumente utilizadas para resolver problemas que envolvem grafos são:

- Matrizes de adjacência: como o próprio nome sugere, ela representa os vértices do grafo em uma matriz de tamanho N, ocupando, na memória, um espaço de tamanho N². A figura 2 mostra uma matriz de adjacência de dimensões 10x10.

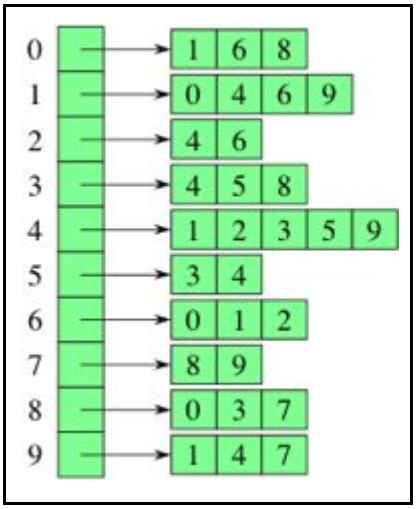
Figura 2 - Matriz de adjacência

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Fonte: Khan Academy, 2018

- Listas de adjacência: armazenam os vértices do grafo em listas encadeadas, de modo que cada aresta define as células a serem apontadas por cada item da lista. O espaço ocupado na memória é proporcional ao tamanho do grafo. A figura 3 mostra a representação de listas de adjacência.

Figura 3 - Listas de adjacência



Fonte: Khan Academy, 2018

O tipo de grafo utilizado para este trabalho é chamado de esparso, o que significa que possui menos arestas que um grafo completo teria. Portanto, foi mais compacto e prudente utilizar listas de adjacência na implementação, já que, para uma matriz, seria necessária a alocação de um espaço de memória muito maior e com desperdício.

A seguir, é mostrado o algoritmo que descreve a resolução do problema proposto:

- Recebo os a quantidade de pessoas a serem lidas;
- Recebo a quantidade de relações;
- Recebo o id da primeira pessoa a ouvir a música;
- Insiro cada pessoa em um vetor;
- Insiro as pessoas relacionadas a ela que tenham menos de 35 (idade mínima para não gostar da música) anos em uma lista correspondente;
- Marco a primeira pessoa a ouvir a música como pesquisada e faço uma busca em cada pessoa relacionada a ela;
- Nessa busca, se a pessoa já for pesquisada, não faço uma busca nela;
- Se a pessoa ainda não tiver sido pesquisada, faço, recursivamente, uma busca nela;
- Contabilizo o número de pessoas em que fiz busca;
- Retorno esse valor.

Para a implementá-lo foram utilizadas as seguintes estruturas de dados:

- Tipo Pessoa: uma pessoa é uma célula de uma lista encadeada que possui uma célula seguinte (prox), um id, uma idade e uma flag que indica se ela já foi pesquisada ou não (inicialmente, todas as pessoas possuem essa flag igual a zero);

O programa possui as seguintes funções:

- **InserirRelação()**
  É utilizada no início do programa, quando a lista de pessoas é montada. Essa função é do tipo *void* e recebe por parâmetros um objeto do tipo Pessoa (que será a lista na qual será feito o trabalho), dois ids e suas respectivas idades. O primeiro representa a pessoa “dona” da lista na qual o segundo será colocado.
  Esta função percorre a lista de pessoas relacionadas à passada por parâmetro primeiro até encontrar a última ou uma de id imediatamente menor. Encontrada, ela insere a nova pessoa.
  InserirRelação() possui complexidade O(n), sendo n o número de arestas que já estão na lista em que será feita a inserção. Nela, é feita apenas uma alocação dinâmica de memória, necessária para criar um novo objeto do tipo Relação com os atributos do objeto pessoa a ser inserido. São usadas três variáveis auxiliares do tipo Pessoa.

- **ImprimirLista()**
  Esta função não tem utilidade na execução do programa, é usada apenas para fins de teste. Para visualizar o que ela mostra, basta alterar a flag DEBUG de 0 para 1 no arquivo *main.c*. Ela recebe por parâmetros um objeto do tipo Relações e um inteiro com o tamanho desse objeto (que é uma lista), e o percorre imprimindo cada pessoa com seus relacionados e suas respectivas idades.
  A função possui complexidade O(n), sendo n a soma do número de vértices com o número de arestas. Essa complexidade se dá devido ao fato de a função percorrer todas as arestas de cada vértice do grafo.
  Não é feita qualquer alocação dinâmica de memória, sendo necessária apenas uma variável temporária do tipo Pessoa.

- **ProcuraLista()**
  O objetivo dessa função é fazer uma espécie de busca em profundidade no grafo. Ela recebe por parâmetros um objeto do tipo Pessoa (lista), um inteiro com seu tamanho, um id (que representa a pessoa a partir da qual será feita uma busca) e o endereço um inteiro que será usado para contar o número de pessoas que gostaram da música.
  ProcuraLista() é uma função recursiva que faz uma busca em cada elemento da lista de relações da pessoa passada por parâmetro. Se a pessoa não tiver sido pesquisada, a função é chamada para ela de modo a fazer a busca. No caso de uma pessoa já pesquisada, o procedimento não é chamado. A condição de parada para a recursão é de que uma pessoa não tenha relações ou que todas as suas relações já tenham sido pesquisadas.
  Esta função tem complexidade O(n) no pior caso, sendo n a soma do número de arestas e vértices do grafo. Essa pesquisa é feita somente uma vez por elemento.
  Não é feita qualquer alocação dinâmica de memória, sendo necessária apenas uma variável temporária do tipo Pessoa.

- **LiberaEspaço()**
  Função para desalocar o que foi alocado ao longo do programa. Ela desloca cada objeto do tipo Pessoa utilizado e, ao final, desaloca a lista. Para isso, utiliza a função free(). Tem complexidade O(n), onde n é a soma do número de vértices com o número de arestas.

Para a realização dos testes, foram disponibilizados 10 casos de entrada (com suas respectivas saídas) de tamanhos diferentes. Todos foram testados e obtive as saídas corretas. Na primeira versão pronta do trabalho, a execução dos testes demorava mais que o ideal. Por isso, fiz três otimizações e testei o tempo real de execução nos 10 casos. Com isso, montei o quadro 3.

Quadro 3 - Tempo real de execução por versão e caso

Casos	Versão 1	Versão 2	Versão 3	Versão 4
1	0,003s	0,003s	0,003s	0,003s
2	0,003s	0,003s	0,003s	0,003s
3	0,006s	0,006s	0,005s	0,004s
4	0,056s	0,051s	0,043s	0,008s
5	0,228s	0,172s	0,159s	0,022s
6	22,792s	19,264s	14,260s	0,126ss
7	3min19,542s	2min43,566s	2min11,795s	0,320s
8	7min41,821s	não testado	5min20,761s	0,597s
9	não testado	não testado	não testado	0,857s
10	não testado	não testado	não testado	8,45s

Fonte: criado pela autora

4. Considerações finais

Na primeira versão eu utilizava dois tipos: TipoRelações e TipoPessoa. A lista principal era do primeiro tipo, e carregava o segundo em cada célula. Devido a essa estrutura, eram necessárias alocações dinâmicas de memória que demandavam um alto custo de execução.
Na primeira versão, eu inseria todas as relações na lista. Na segunda versão, mantive toda a estrutura e criei um filtro que só inseria relações entre pessoas que tinham menos de 35 anos. Isso economizou muitas comparações e chamadas de funções, o que reduziu um pouco o tempo gasto.

Da versão 2 para a 3, retirei alguns atributos das estruturas de dados que não eram exatamente necessários e que gastavam certo tempo e espaço para configurar a cada entrada. Como foi uma alteração simples, que economizava alguns Kb sob entradas de tamanho muito grande, o tempo de execução teve uma redução não tão alta.
Da versão 3 para a versão 4, troquei toda a estrutura do programa de forma a utilizar somente uma estrutura de dados (TipoPessoa). Retirei o máximo de alocação dinâmica possível, deixando somente as estritamente necessárias. Essas alterações reduziram consideravelmente o tempo real de execução, mantendo meu programa dentro do limite de 1min30s para similares ao caso 10.
No geral, foi um trabalho interessante e abriu minha mente em relação à estrutura utilizada e sua relação com o tempo de execução demandado. Gastei cerca de 10 horas, ao todo, e minha maior dificuldade foi em realizar a recursão. O programa todo tem complexidade O(n), já que é a complexidade de todas as suas funções.

5. Referências bibliográficas

KHAN ACADEMY, Representando grafos. Disponível em <https://pt.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs>. Acesso em 09 de Abril de 2018.