TP2: Sistema de recomendação utilizando tabelas hash

1. Introdução

Um sistema de recomendação é um sistema pelo qual, através de determinados parâmetros de comparação, um usuário recebe recomendações de conteúdo que pode ser de seu agrado.

Nesse trabalho, o objetivo é criar uma variação do sistema de recomendação de filmes criado no TP0, porém utilizando tabelas hash para a organização dos dados, tornando o sistema mais ágil. Além disso, é necessário que nesse sistema possam ser recomendadas uma quantidade variada de filmes, enquanto no trabalho anterior, o sistema sempre fazia três recomendações, ou menos caso não fosse possível.

Pretende-se com esse trabalho entender melhor, de forma prática, o funcionamento da estrutura de tabela hash.

2. Implementação

Estrutura de Dados

Nesse trabalho, fizemos uso intenso da estrutura de tabela hash.
No início do programa, quando recebemos os filmes, eles são armazenados em uma lista linear. Em seguida, recebemos os usuários, e enquanto isso ocorre, o programa automaticamente contabiliza a popularidade de cada filme. Após esse processo, tranformamos a lista linear em uma tabela hash, utilizando a popularidade de cada filme para o cálculo do hash. Quando há colisão, os filmes são armazenados em uma árvore binária de busca, seguindo os critérios de desempate passados na especificação. Para descobrir os filmes mais populares, basta passar pela árvore em ordem central.
Para a recomendação personalizada, criamos uma hash table do elemento RegistroFilmeJaccard, que armazena os dados de um filme, um usuário e o coeficiente de jaccard entre esse usuário e o usuário para o qual devemos recomendar. Para gerar o hash utilizamos o coeficiente de jaccard, e em caso de colisão também criamos uma árvore binária de busca, usando os critérios de desempate passados na especificação.

Funções e procedimentos

As funções mais importantes estão nos TADs Filmes, Usuarios e HashFilmeJaccard:

Filmes
void inicializaFilmes(Filmes *lista, int tamanhoMaximo): Função que inicializa a lista de filmes alocando o espaço necessário e iniciando o tamanho da lista

int calculaHash(Filme filme, int numUsuarios, int tamanhoHash): Função que calcula o hash de determinada chave, obtendo a posição do elemento na tabela hash

void montaTabelaHash(Filmes *lista, int numUsuarios, int tamanhoHash): Função que transforma uma lista linear de filmes em uma tabela hash. Foi necessária a existência da lista linear inicialmente para que a leitura de filmes dos arquivos seja feita, em seguida sejam obtidos os dados de popularidade dos filmes, lendo o arquivo dos usuários, e só então a tabela hash se forme.

void finalizaTabelaHash(Filmes *lista): Libera a memória usada pela tabela e pelas árvores

void incluiFilmeFinal(Filmes *lista, Filme novo): Inclui um filme no final da lista de filmes. Deve ser usada quando a estrutura estiver organizando os filmes em uma lista linear

void insereArvore(Filme* raiz, Filme novo): Inclui um filme na árvore de filmes cuja raiz é o filme passado

void incluiFilmeHash(Filmes *lista, Filme novo, int numUsuarios): Inclui um filme na tabela hash na posição correta. Deve ser usada quando a estrutura estiver organizando os filmes em uma tabela hash

void recebeFilmesArquivo(Filmes *lista, char* nomeArquivo): Lê os filmes de um arquivo passado e os insere em uma lista linear

Usuarios
void inicializaUsuarios(Usuarios *lista, int tamanhoListaUsuarios): Função que inicializa a lista de usuários alocando o espaço necessário e iniciando o tamanho da lista

void incluiUsuario(Usuarios *lista, Usuario novo): Inclui um usuário na lista de usuários

Usuario getUsuario(Usuarios lista, int indexUsuario): Retorna o usuário na posição requisitada na lista

Usuario getUsuarioId(Usuarios lista, int id): Retorna o usuário cujo id seja igual ao requisitado

void recebeUsuariosArquivo(Usuarios *lista, char* nomeArquivo, int numFilmes): Lê os usuários de um arquivo passado e os insere na lista, armazenando os ids dos filmes que esse usuário assistiu

HashFilmeJaccard
HashFilmeJaccard* geraRelacaoRegistros(Usuarios usuarios, Filmes filmes, Usuario recomendar, int tamanhoHash, int tamanhoListaLinear): Recebe os usuários e filmes do sistema, e a partir disso gera os registros que armazenam a relação entre cada usuário e cada filme que assistiu, registrando o valor de jaccard para esse usuário e o usuário para o qual se vai recomendar

void inicializaHash(HashFilmeJaccard *table, int tamanhoHash): Inicializa o hash, alocando o espaço para a tabela, inicializando as posições no hash para que seu jaccard seja igual a -1, como forma de indicar que esse espaço não possui registro armazenado. Armazena no tamanho da lista o tamanho da tabela hash

int calculaHashRegistro(FilmeJaccard registro, int tamanhoHash): Calcula o hash de determinado registro na tabela, utilizando o jaccard da combinação para esse cálculo

void finalizaTabelaHashRegistro(HashFilmeJaccard *table): Desaloca a memória das árvores em cada posição da tabela hash, e depois desaloca a tabela hash

void insereRegistroArvore(FilmeJaccard* raiz, FilmeJaccard* novo): Inclui um registro na árvore de registros cuja raiz é o registro passado

void incluiRegistroHash(HashFilmeJaccard *lista, FilmeJaccard* novo): Insere um registro na tabela hash, e em caso de colisão insere o registro na árvore na posição correta da tabela hash

Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código é separado em três partes principais: parte de filmes (TAD de filme e de lista de filmes, que implementa tabela hash), de usuários (TAD de usuário e de lista de usuários) e de registro de filme-usuário (sendo essa uma forma de implementar a relação entre os usuários e os filmes que eles assistiram, tendo o TAD de registro de filme e a tabela hash de registros).

Uma decisão de implementação relevante foi a de criar uma lista linear de filmes no início do programa, e só criar a tabela hash após a leitura dos usuários. Dessa forma, podemos passar por cada usuário uma vez nesse trecho, incrementando a popularidade dos filmes sempre que encontramos seu id na lista de filmes de algum aluno. Organizando a lista linearmente, tendo o id como índice, o acesso aos filmes tem complexidade O(1), o que torna o programa mais eficiente.

3. Análise de complexidade

Filmes (considerando n a quantidade de filmes)
void inicializaFilmes(Filmes *lista, int tamanhoMaximo): Apenas operações de atribuição e alocação, **O(1)**.

int calculaHash(Filme filme, int numUsuarios, int tamanhoHash): Apenas cálculos, **O(1)**.

void montaTabelaHash(Filmes *lista, int numUsuarios, int tamanhoHash): Faz uma iteração de tamanho igual ao tamanho do hash (constante) e em seguida faz um loop pela lista de filmes anterior, portanto, **O(n)**.

void finalizaTabelaHash(Filmes *lista): Passa por cada posição na tabela hash (número constante) e finaliza as árvores recursivamente. No pior caso, todos os filmes estarão na mesma posição na tabela hash, e a árvore da colisão estará totalmente desbalanceada, portanto, **O(n)**.

void incluiFilmeFinal(Filmes *lista, Filme novo): Insere na última posição da lista, sem ter que verificar as outras posições, **O(1)**.

void insereArvore(Filme* raiz, Filme novo): Insere na árvore, e no pior caso essa árvore estará desbalanceada e terá todos os filmes, portanto, **O(n)**.

void incluiFilmeHash(Filmes *lista, Filme novo, int numUsuarios): Chama a função calculaHash (O(1)) e a função insereArvore (O(n)) caso haja colisão. Portanto, no pior caso, **O(n)**.

void recebeFilmesArquivo(Filmes *lista, char* nomeArquivo): Lê todos os filmes da lista passada, um por um. Chama a função incluiFilmeFinal (O(1)) para cada filme. Portanto, **O(n)**.

Usuarios (considerando n o número de usuários)
void inicializaUsuarios(Usuarios *lista, int tamanhoListaUsuarios): Apenas operações de atribuição e alocação, **O(1)**.

void incluiUsuario(Usuarios *lista, Usuario novo): Insere na última posição da lista, sem ter que verificar as outras posições, **O(1)**.

Usuario getUsuario(Usuarios lista, int indexUsuario): Encontra um usuário pelo índice, portanto, **O(1)**.

Usuario getUsuarioId(Usuarios lista, int id): Varre a lista linear em busca de um usuário de determinado id, e no pior caso passa por todos os usuários, portanto, **O(n)**.

void recebeUsuariosArquivo(Usuarios *lista, char* nomeArquivo, int numFilmes): Lê todos os usuários da lista passada, um por um. Chama a função incluiUsuario (O(1)) para cada usuário. Portanto, **O(n)**.

HashFilmeJaccard (considerando n o número de registros de relação usuário-filme)

void inicializaHash(HashFilmeJaccard *table, int tamanhoHash): Apenas aloca o espaço para a tabela hash e inicializa as posições na tabela com valores default. Considerando que o tamanho da tabela é uma constante, **O(1)**.

int calculaHashRegistro(FilmeJaccard registro, int tamanhoHash): Apenas cálculos, **O(1)**.

void finalizaTabelaHashRegistro(HashFilmeJaccard *table): Desaloca a memória usada pelas árvores nas posições da tabela hash. No pior caso, todos os registros estão na mesma posição, em uma árvore totalmente desbalanceada, portanto, **O(n)**.

void insereRegistroArvore(FilmeJaccard* raiz, FilmeJaccard* novo): Insere na árvore, e no pior caso essa árvore estará desbalanceada e terá todos os filmes, portanto, **O(n)**.

void incluiRegistroHash(HashFilmeJaccard *lista, FilmeJaccard* novo): Chama a função calculaHashRegistro (O(1)) e a função insereRegistroArvore (O(n)) caso haja colisão. Portanto, no pior caso, **O(n)**.

HashFilmeJaccard* geraRelacaoRegistros(Usuarios usuarios, Filmes filmes, Usuario recomendar, int tamanhoHash, int tamanhoListaLinear): Chama a função inicializaHash (O(1)), e descobre todos os registros necessários de serem criados, para cada registro possível, chama incluiRegistroHash (O(n)), portanto, **$O(n^2)$**.

4. Testes

Foram realizados alguns testes simples. Os testes foram realizados em um Intel Core i5, com 4GB de memória em um Ubuntu 14.04 LTS (Trusty Tahr).

Teste 1:

Entrada
bases/50_users_100_items_metadata.txt bases/50_users_100_items_ratingsN.txt 5 5

22957
42582
8881

Saída
bases/50_users_100_items_metadata.txt bases/50_users_100_items_ratingsN.txt 5 5

22957:
Most popular
Back to the Future Gladiator The Sixth Sense Terminator Salvation The Silence of the Lambs

Personalizada
Terminator Salvation Mission: Impossible III Catch Me If You Can Gladiator American Pie

42582:
Most popular
The Lord of the Rings: The Two Towers The Lord of the Rings: The Fellowship of the Ring Shrek Pirates of the Caribbean: The Curse of the Black Pearl The Lord of the Rings: The Return of the King

Personalizada
V for Vendetta Sin City Kill Bill: Vol. 2 Eternal Sunshine of the Spotless Mind Lost in Translation

8881:
Most popular
The Shawshank Redemption Batman Begins X-Men Braveheart Sin City

Personalizada
V for Vendetta Sin City Batman Begins Shrek 2 Kill Bill: Vol. 2

Teste 2:

Entrada
bases/5_users_10_items_metadata.txt bases/5_users_10_items_ratingsN.txt 2 5

1510

Saída
bases/5_users_10_items_metadata.txt bases/5_users_10_items_ratingsN.txt 2 5

1510:
Most popular
The Matrix Pulp Fiction

Personalizada
The Matrix Pulp Fiction

5. Conclusão

Esse trabalho foi bastante difícil, houveram muitos erros durante seu desenvolvimento, foi difícil manter o código organizado, e houveram muitas falhas de segmentação devido a ponteiros não inicializados. Mas ao longo do tempo esses problemas foram sendo resolvidos. O trabalho proporcionou um bom entendimento prático da estrutura de tabela hash, e abordou um problema com aplicação prática bem evidente hoje em dia.

Referências

Slides no "Moodle" da turma de AEDS II.

Anexos

- filme.c
- filme.h
- filmes.c
- filmes.h
- hashFilmeJaccard.c
- hashFilmeJaccard.h
- registroFilmeJaccard.c
- registroFilmeJaccard.h
- usuario.c
- usuario.h
- usuarios.c
- usuarios.h
- funcoesUsuariosFilmes.c
- funcoesUsuariosFilmes.h
- TP2.c