Tabela Hash para Deduplicação de Datasets

Wilson Albuquerque Ramos
DRE:118092402
Link do Repositório: https://github.com/wilsonufrj/TabelaHash

Introdução

Este projeto teve como objetivo principal o desenvolvimento e aplicação de uma **Tabela Hash** (ou tabela de dispersão) em C++ para resolver eficientemente o problema da deduplicação de registros em datasets. A deduplicação é uma etapa crítica no pré-processamento de dados para diversas aplicações, incluindo Ciência de Dados e Machine Learning, onde a presença de dados duplicados pode enviesar análises e modelos. A escolha da tabela hash como método para essa tarefa visa explorar sua capacidade de oferecer uma complexidade de tempo média de O(n), significativamente superior às abordagens tradicionais baseadas em ordenação (O(nlogn)) para grandes volumes de dados.

O sistema desenvolvido lê um arquivo CSV de entrada, utiliza uma coluna específica como chave para identificar e remover linhas duplicadas e, por fim, gera um novo arquivo CSV contendo apenas os registros únicos.

Estrutura do Projeto e Divisão de Módulos

O projeto foi modularizado para promover a organização, reusabilidade e manutenibilidade do código. A estrutura de diretórios e arquivos reflete essa divisão:

- **`main.cpp`**: Contém a lógica principal do programa. É responsável pela leitura do arquivo CSV de entrada (`dataset.csv`), pela orquestração do processo de deduplicação (interagindo com as classes `Registro` e `TabelaHash`), e pela escrita dos dados deduplicados no arquivo de saída (`dataset_sem_duplicatas.csv`).
- **`Registro.h` e `Registro.cpp`**: Definem e implementam a classe `Registro`. Esta classe encapsula os dados de uma única linha do dataset. Ela é fundamental para representar cada registro e fornecer métodos para acessar sua chave (a coluna utilizada para deduplicação) e os demais dados da linha. Métodos de comparação entre registros podem ser implementados aqui para uma lógica de deduplicação mais granular, comparando todos os campos e não apenas a chave.
- **`TabelaHash.h` e `TabelaHash.cpp`**: Contêm a definição e a implementação da classe `TabelaHash`, o cerne da solução. Esta classe é responsável por gerenciar a

estrutura da tabela hash, as funções de hashing e a estratégia de resolução de colisões.

Essa divisão em módulos permite que cada componente seja desenvolvido e testado de forma independente, facilitando futuras extensões e modificações.

Estruturas de Dados Utilizadas

A principal estrutura de dados utilizada neste projeto é a **Tabela Hash**, implementada como uma classe em C++. Internamente, a `TabelaHash` utiliza uma estrutura de dado linear para armazenar os *buckets* (ou "baldes"), que são as posições onde os registros serão armazenados após a aplicação da função de hashing.

Para a **resolução de colisões**, optou-se pela estratégia de **encadeamento exterior**. Isso significa que cada posição (índice) na estrutura linear da tabela hash pode conter uma lista (geralmente uma lista encadeada) de registros. Quando múltiplos registros geram o mesmo índice hash, eles são adicionados à lista correspondente àquele *bucket*. Essa abordagem é robusta e flexível, pois permite que um número arbitrário de colisões seja tratado sem a necessidade de redimensionamento imediato ou de estratégias complexas de busca por slots vazios.

A escolha do encadeamento exterior impacta a interface da `TabelaHash`, onde o operador de indexação (`[]`) pode ser utilizado para acessar o *bucket* correspondente a um índice, e a partir daí, iterar sobre a lista de registros naquele *bucket*.

Descrição das Rotinas e Funções Chave

Classe `Registro`

- **Construtor**: Inicializa um objeto `Registro` com os dados de uma linha do CSV, identificando qual coluna será tratada como chave.
- **`getChave()`**: Retorna o valor da chave do registro.
- **`getDadosCompletos()`**: Retorna a string completa que representa a linha do registro, útil para a escrita no arquivo de saída.
- **Operador de Comparação (`==`)**: (Sugestão para deduplicação mais robusta) Poderia ser sobrecarregado para permitir a comparação de todos os campos de dois objetos `Registro`, não apenas a chave, garantindo que "duplicatas" sejam verdadeiramente idênticas em todos os aspectos relevantes.

Classe `TabelaHash`

- **Construtor**: Inicializa a tabela hash com um determinado tamanho e, opcionalmente, com uma função de hashing padrão. A estrutura linear interna (como um `std::vector` de `std::list<Registro*>`) é alocada aqui.
- **Funções de Hashing (`hashFunction(...)`)**: Várias funções de hashing podem ser implementadas e testadas. A função padrão no projeto provavelmente se baseia em um dos métodos clássicos (divisão, multiplicação, etc.) aplicados ao valor da chave (string ou numérico).
    1. **Exemplos de Funções de Hashing a serem testadas:**
        - **Função de Divisão**: `hash(chave) = chave % tamanho_tabela` (para chaves numéricas).
        - **Função de Multiplicação**: Baseada em `floor(tamanho_tabela * (chave * A - floor(chave * A)))`, onde `A` é uma constante entre 0 e 1.
        - **Função para Strings**: Métodos que somam os valores ASCII dos caracteres, ou utilizam shifts bit-a-bit e XOR, para gerar um hash a partir de uma string.
- **`inserir(chave, dado)`**: Este é o método central para a deduplicação.
    1. Calcula o índice hash para a `chave` fornecida.
    2. Acessa o *bucket* correspondente.
    3. **Verificação de Duplicata**: Itera sobre os registros já presentes no *bucket*. Se um registro com a **mesma chave** for encontrado, o `dado` atual é considerado uma duplicata e **não é inserido**.
    4. Se a chave não for encontrada, o `dado` é adicionado à lista do *bucket*.
- **`buscar(chave)`**: Procura por um registro com a `chave` especificada. Calcula o índice hash, acessa o *bucket* e itera sobre a lista para encontrar o registro. Retorna o registro se encontrado, ou `nullptr` (ou similar) caso contrário.
- **`remover(chave)`**: Semelhante à busca, mas remove o registro da lista do *bucket* se encontrado.
- **`getRegistrosUnicos()`**: Percorre todos os *buckets* da tabela hash e coleta todos os registros únicos (aqueles que foram inseridos e não foram considerados duplicatas) em uma estrutura de dados (e.g., `std::vector<Registro*>`) para posterior escrita no arquivo de saída.

Complexidades de Tempo e Espaço

Complexidade de Tempo

- **Deduplicação (Inclusão/Busca)**:
    - **Melhor Caso (O(1))**: Ocorre quando a função de hashing distribui as chaves uniformemente e não há colisões no *bucket* do registro a ser inserido/buscado.
    - **Caso Médio (O(1))**: Em uma tabela hash bem projetada, com uma boa função de hashing e fator de carga adequado, o tempo médio para inserir ou

buscar um registro é constante, pois o número de elementos por *bucket* é, em média, muito pequeno.
    - **Pior Caso (O(Nk))**: Ocorre quando todos (ou a maioria) dos registros colidem e são mapeados para o mesmo *bucket*. Nesse cenário degenerado, a operação de inserção/busca degenera para uma busca linear na lista do *bucket*, onde Nk é o número de registros no *bucket* em questão, podendo ser tão grande quanto N (o número total de registros).
- **Deduplicação Global do Dataset (O(N))**: A principal vantagem de usar uma tabela hash para deduplicação é que, em média, cada registro é processado em tempo constante. Assim, o tempo total para processar N registros é $N \times O(1) = O(N)$. Isso é uma melhoria significativa em relação a métodos baseados em ordenação, que são $O(N \log N)$.

Complexidade de Espaço

- **Tabela Hash (O(N))**: A tabela hash armazena todos os registros únicos. Portanto, o espaço necessário é proporcional ao número de registros únicos presentes no dataset. Além disso, a estrutura linear interna da tabela consome um espaço proporcional ao seu tamanho (número de *buckets*), que é geralmente pré-definido e, em um bom cenário, proporcional a N.
- A escolha do encadeamento exterior adiciona um pequeno overhead de ponteiros ou cabeçalhos de listas para cada *bucket*.

Problemas e Observações Encontrados Durante o Desenvolvimento

- **Escolha da Função de Hashing**: A eficácia da tabela hash depende crucialmente de uma boa função de hashing. Uma função inadequada pode levar a um grande número de colisões, degradando o desempenho para o pior caso. Testar diferentes funções (divisão, multiplicação, etc.) e adaptá-las ao domínio das chaves (e.g., strings vs. inteiros) é um desafio e uma área importante para otimização.
- **Tratamento de Colisões**: Embora o encadeamento exterior seja robusto, um número excessivo de colisões em um único *bucket* ainda pode levar à degradação do desempenho. Isso pode ser mitigado escolhendo um tamanho de tabela adequado e, se necessário, implementando um mecanismo de redimensionamento da tabela (rehashing) quando o fator de carga excede um limite.
- **Representação de Registros**: A decisão de como um "registro" é representado (e.g., `std::string` da linha inteira, ou campos parseados individualmente) afeta tanto a complexidade da classe `Registro` quanto o processo de comparação para identificar duplicatas. No projeto atual, a identificação de duplicatas baseia-se primariamente na **chave primária**. Para uma deduplicação mais robusta, onde a "duplicata" é um registro idêntico em *todos* os campos (exceto talvez um ID único), a lógica de comparação na classe `Registro` e na função de inserção da

`TabelaHash` precisaria ser ajustada para comparar mais do que apenas a chave hash.
- **Leitura de CSV**: A parsing de arquivos CSV pode ser complexa devido a vírgulas dentro de campos, aspas, etc. Uma biblioteca robusta para parsing de CSV pode ser benéfica para projetos maiores. Neste projeto, assumiu-se um formato CSV relativamente simples.
- **Fator de Carga**: O fator de carga (número de itens / tamanho da tabela) é um indicador importante da eficiência. Monitorá-lo e, se necessário, implementar o rehashing pode otimizar o desempenho.

Conclusão a Respeito dos Resultados Obtidos

A implementação da tabela hash para deduplicação de datasets demonstrou ser uma solução eficaz e, em média, eficiente, validando o benefício de performance esperado (O(N) vs. O(NlogN)). A modularização do código em classes `Registro` e `TabelaHash` contribuiu para a clareza e organização do projeto, tornando-o mais fácil de entender e expandir.

Os resultados obtidos confirmam que, para a tarefa de deduplicação em grandes volumes de dados, uma tabela hash bem implementada, com uma função de hashing adequada e uma estratégia de tratamento de colisões eficiente como o encadeamento exterior, oferece uma performance superior em comparação com métodos que dependem de ordenação. A flexibilidade na escolha da função de hashing permite adaptar a solução a diferentes domínios de chaves, garantindo a robustez do algoritmo.

Futuros aprimoramentos poderiam incluir a implementação de diferentes funções de hashing para comparação de desempenho, a adição de um mecanismo de redimensionamento dinâmico (rehashing) para otimizar o fator de carga, e a expansão da lógica de comparação de registros para permitir a identificação de duplicatas baseada em múltiplos campos, proporcionando uma solução de deduplicação mais abrangente. No geral, o projeto estabelece uma base sólida para o uso de tabelas hash em problemas de pré-processamento de dados.