Relatório de Análise do Código Python: Implementação de Tabela Hash para Deduplicação de Dados

1. Introdução

Este relatório apresenta uma análise detalhada do código, que implementa uma tabela *Hash* (tabela de dispersão) e a utiliza para resolver o problema de deduplicação em *datasets* no formato CSV.

2. Funções de Hashing

O código define duas funções de *hashing* distintas, permitindo flexibilidade na escolha do algoritmo de dispersão:

- **hash_divisao(chave, tamanho)**: Esta função implementa o método de *hashing* por divisão. Ela calcula o valor *hash* de uma chave aplicando o operador de módulo (%) ao resultado da função hash() embutida do Python e ao tamanho da tabela *hash*. O valor retornado é o índice onde a chave deve ser potencialmente armazenada na tabela.
    - **Fórmula:** hash_divisao(chave,tamanho)=hash(chave)%tamanho
- **hash_multiplicacao(chave, tamanho)**: Esta função utiliza o método de *hashing* por multiplicação. Ela emprega uma constante A (a parte fracionária do número de ouro, aproximadamente 0.6180339887) para calcular o *hash*. O processo envolve multiplicar o *hash* da chave por A, pegar a parte fracionária do resultado, e então multiplicar essa parte fracionária pelo tamanho da tabela para obter o índice.
    - **Fórmula:** hash_multiplicacao(chave,tamanho)=⌊tamanho×((hash(chave)×A)%1)⌋

3. Classe HashTable

A classe HashTable é a peça central da implementação, fornecendo a estrutura e os métodos para gerenciar a tabela de dispersão.

- **__init__(self, tamanho=100, funcao_hash=hash_divisao)**: O construtor da classe inicializa a tabela *hash*.
    - self.tamanho: Define o número de *buckets* (baldes) na tabela. O valor padrão é 100.
    - self.tabela: É uma lista de listas, representando a estrutura linear (um *array* de listas) que serve como os *buckets* da tabela *hash*. Cada sublista pode

armazenar múltiplos pares (chave, valor), implementando o método de **resolução de colisão por encadeamento exterior**.
    - `self.funcao_hash`: Armazena a função de *hashing* a ser utilizada (por padrão, `hash_divisao`). Isso permite que o programador especifique a função que melhor se adapta às características dos dados.
- **`_endereco(self, chave)`**: Este método privado (`_` indica que é para uso interno da classe) calcula o endereço (índice) de uma dada chave na tabela *hash* usando a `self.funcao_hash` configurada.
- **`inserir(self, chave, valor)`**: Insere um par (chave, valor) na tabela *hash*.
    - Primeiro, calcula o índice usando `_endereco()`.
    - Em seguida, itera sobre os pares já existentes no *bucket* correspondente. Se a chave já existir, a inserção é ignorada (tratamento de duplicatas no contexto da tabela *hash*).
    - Se a chave não for encontrada no *bucket*, o novo par (chave, valor) é adicionado ao final da lista (encadeamento).
- **`buscar(self, chave)`**: Busca um valor associado a uma dada chave na tabela *hash*.
    - Calcula o índice da chave.
    - Percorre os pares no *bucket* correspondente. Se a chave for encontrada, retorna o valor associado.
    - Se a chave não for encontrada no *bucket*, retorna `None`.
- **`remover(self, chave)`**: Remove um par (chave, valor) da tabela *hash* com base na chave.
    - Calcula o índice da chave.
    - Itera sobre os pares no *bucket* correspondente usando `enumerate` para obter o índice do par na sublista.
    - Se a chave for encontrada, remove o par do *bucket* e retorna `True`.
    - Se a chave não for encontrada, retorna `False`.
- **`valores(self)`**: Um método gerador que permite iterar sobre todos os valores armazenados na tabela *hash*. Ele percorre cada *bucket* e, para cada par (chave, valor) dentro do *bucket*, *yields* (produz) o valor.

4. Estruturas de Dados Utilizadas

- **Lista Python (`list`)**: A estrutura de dados fundamental para a `self.tabela` na classe `HashTable`. É uma estrutura linear que oferece acesso por índice (endereço), como um *array*. Cada elemento dessa lista é, por sua vez, outra lista, que serve para armazenar os pares (chave, valor) que colidiram no mesmo *bucket* (encadeamento exterior).
- **Tuplas Python (`tuple`)**: Utilizadas para armazenar os pares (chave, valor) dentro dos *buckets* (`(chave, valor)`). Tuplas são imutáveis, o que as torna adequadas para representar esses pares de dados.
- **Dicionário Python (`dict`)**: No contexto da função `csv.DictReader`, cada linha do arquivo CSV é lida como um dicionário, onde as chaves são os nomes das colunas e os valores são os dados da linha.

5. Função remover_duplicatas_csv

Esta função é a aplicação prática da HashTable para o problema de deduplicação de dados em arquivos CSV.

- **remover_duplicatas_csv(caminho_csv, chave_coluna, separador=',')**:
    - Inicializa uma HashTable com um tamanho de 1000 *buckets* e utilizando a função hash_divisao. Um tamanho maior de tabela *hash* geralmente reduz a probabilidade de colisões e melhora o desempenho.
    - Abre o arquivo CSV especificado em modo de leitura ('r') com newline='' e encoding='utf-8' para garantir a correta leitura.
    - Cria um csv.DictReader para ler o CSV como dicionários, facilitando o acesso aos dados por nome da coluna.
    - Itera sobre cada linha (que é um dicionário) lida do CSV:
        - Extrai a chave da linha usando o nome da chave_coluna fornecido.
        - Chama hash_table.inserir(chave, linha). A lógica da HashTable garante que, se a chave já existir, a nova linha (que seria uma duplicata em relação à chave) não será inserida, resolvendo o problema de deduplicação.
    - Ao final da leitura do CSV, todos os registros únicos (com base na chave_coluna) estarão armazenados na hash_table.
    - Retorna uma lista contendo todos os valores (as linhas completas dos registros únicos) da hash_table usando list(hash_table.valores()).

6. Exemplo de Uso (if __name__ == "__main__":)

A seção if __name__ == "__main__": demonstra como usar a função remover_duplicatas_csv.

- Define o caminho para o arquivo CSV ("exemplo.csv") e a chave (nome da coluna a ser usada para deduplicação, por exemplo, "cpf").
- Chama remover_duplicatas_csv para obter a lista de registros únicos.
- Imprime os registros únicos no console, mostrando o resultado da deduplicação.

7. Complexidade de Tempo e Espaço

- **Complexidade de Tempo:**
    - **Inserção, Busca, Remoção na HashTable (Média)**: Em média, se a função de *hashing* distribui bem as chaves e o número de colisões é minimizado, essas operações têm uma complexidade de tempo de O(1) (tempo constante). No pior caso (todas as chaves colidem no mesmo *bucket*), a complexidade degenera para O(N) onde N é o número de elementos no *bucket*.
    - **remover_duplicatas_csv**: A função itera uma vez sobre cada linha do arquivo CSV. Para cada linha, uma operação de inserção é realizada na *hash table*. Portanto, a complexidade dominante é a leitura do CSV mais N operações de inserção na *hash table*. Se as inserções tiverem complexidade

O(1) em média, a complexidade total será O(M) onde M é o número de linhas no arquivo CSV. Isso é significativamente mais eficiente que soluções baseadas em ordenação, que geralmente têm complexidade O(MlogM).

- **Complexidade de Espaço:**
    - **HashTable**: A complexidade de espaço é O(T+K), onde T é o tamanho da tabela (número de *buckets*) e K é o número total de elementos únicos armazenados na tabela.

8. Problemas e Observações Encontradas Durante o Desenvolvimento

- **Escolha da Função de Hashing**: A eficácia da tabela *hash* depende criticamente da função de *hashing* escolhida. Uma função ruim pode levar a muitas colisões, degradando o desempenho para o pior caso (O(N)). O código fornece duas opções (hash_divisao e hash_multiplicacao), permitindo experimentação.
- **Tratamento de Colisões**: O método de encadeamento exterior é robusto e simples de implementar, mas o desempenho pode ser afetado se os *buckets* ficarem muito grandes devido a uma má distribuição das chaves.
- **Tamanho da Tabela Hash**: A escolha do tamanho inicial da tabela HashTable (padrão 100, mas 1000 para a deduplicação de CSV) é crucial. Um tamanho muito pequeno pode aumentar as colisões, enquanto um muito grande pode desperdiçar memória. Um dimensionamento adequado ou um mecanismo de redimensionamento dinâmico (não implementado neste código) pode ser benéfico para grandes *datasets*.
- **Tipos de Dados da Chave**: A função hash() embutida do Python funciona bem para a maioria dos tipos de dados imutáveis (strings, números, tuplas). No entanto, a escolha da chave_coluna deve ser feita com cuidado para garantir que ela realmente represente um identificador único para deduplicação.
- **Robustez de I/O**: A função remover_duplicatas_csv inclui newline='' e encoding='utf-8', que são boas práticas para manipulação de arquivos CSV, garantindo compatibilidade entre diferentes sistemas operacionais e caracteres especiais.

9. Conclusão

O código Python apresentado demonstra uma implementação eficaz de uma tabela *Hash* com resolução de colisões por encadeamento exterior, aplicando-a com sucesso na tarefa de deduplicação de registros em arquivos CSV. A modularidade da classe HashTable, permitindo a injeção de diferentes funções de *hashing*, é um ponto positivo que atende aos requisitos da atividade.

A solução proposta atinge a complexidade de tempo desejada de O(N) para a deduplicação, superando a eficiência de métodos baseados em ordenação (O(NlogN)). Isso a torna particularmente adequada para o tratamento de grandes *datasets* em aplicações de Ciência de Dados, onde a performance é um fator crítico.

A clareza do código e a utilização de práticas recomendadas para manipulação de arquivos CSV contribuem para a robustez da solução. Para futuras melhorias, poderiam ser exploradas estratégias de redimensionamento dinâmico da tabela *hash* para otimizar o uso de memória e desempenho em cenários com um número desconhecido de elementos, bem como a implementação de outras funções de *hashing* (enlaçamento, meio-quadrado, etc.) para comparação de desempenho.