Estudo das Tabelas de Dispersão

Caio Graça Gomes

13/03/2020

1 Introdução

O objetivo desse primeiro laboratório foi entender a implementação em $C++$ e a importância das tabelas de dispersão, compreendendo as razões que tornam essas estruturas de dados essenciais e analisando graficamente seus desempenhos sob diferentes circunstâncias com o uso do $MATLAB$.

2 Compreendendo as *Hash Tables*

1. **Por que necessitamos escolher uma boa função de *hashing*, e quais as consequências de escolher uma função ruim?**

   A escolha de uma função hash ruim pode gerar uma distribuição de elementos não uniforme na tabela. Concentrações de elementos em posições específicas devem ser evitadas para que as operações de busca, inserção e deleção operem em tempo aproximadamente $O(1)$.

2. **Por que há uma diferenca significativa entre considerar apenas o 1 caractere ou a soma de todos?**

   Pois as *strings* a serem distribuídas nas tabelas de dispersão usualmente não são totalmente aleatórias. As *strings* provavelmente serão palavras de uma determinada língua, viesando a escolha do primeiro caractere, dado que na maioria das línguas faladas algumas letras aparecem mais que outras como primeira das palavras. Enquanto isso, a soma dos caracteres não apresenta correlação muito relevante com as palavras da língua utilizada.

3. **Por que um *dataset* apresentou resultados muito piores do que os outros, quando consideramos apenas o 1 caractere?**

   Pois as palavras do *dataset startend* não são completamente aleatórias. É fácil ver que há uma imensa quantidade de palavras iniciadas com a letra *"c"* e também outras letras do início do alfabeto. Assim, a função *hash* baseada no primeiro caractere da *string* não alcança uma entropia tão desejável quanto a de uma distribuição ideal uniforme.

4. **Com uma tabela de *hash* maior, o *hash* deveria ser mais fácil. Afinal temos mais posições na tabela para espalhar as *strings*. *hash* com tamanho 30 não deveria ser sempre melhor do que com tamanho 29? Porque não é este o resultado?**

   Utilizar o tamanho 29 é mais desejável que tamanho 30 porque é preferível escolher um número primo para o tamanho da tabela *hash* e a contribuição de um *bucket* a mais não é suficientemente relevante para contrapor esse efeito.

5. **Uma regra comum é usar um tamanho primo (e.g. 29) e não um tamanho com vários divisores, como 30. Que tipo de problema o tamanho primo evita, e porque a diferença não é muito grande no nosso exemplo?**

   Utilizar uma tabela *hash* de tamanho não primo com vários divisores pode ser indesejável em algumas situações. Suponha que a tabela tem um tamanho T, cujos divisores são $\{d_1, d_2, ..., d_n\}$. Assim, se um elemento incrementa o valor obtido da função *hash* por um divisor $d_i, 1 \leq i \leq n$ ou múltiplo um dele, repetir esse elemento mais T/d_i vezes não alterará o *bucket* que a função retorna. Esse efeito deve ser considerado para todos os elementos que incrementam o valor da função por qualquer divisor d_i e também combinações de elementos que gerem esses divisores. Assim sendo o "tamanho efetivo" não é de fato a quantidade de *buckets* T, pois repetir certas combinações de elementos uma quantidade arbitrária de vezes não é uma condição suficiente para preencher todos os *buckets*, resultando em uma distribuição consideravelmente não uniforme.

   No entanto, nos exemplos dos casos testes utilizados foram utilizadas palavras de tamanho pequeno, o que minimiza o efeito acima citado pois não há repetição de caracteres ou combinações de caracteres suficientes para que isso tenha relevância considerável.

6. **Note que o arquivo mod30 foi feito para atacar um *hash* por divisão de tabela de tamanho 30. Como este ataque funciona?**

A maioria das palavras usadas no teste retornam o valor 4 quando computadas na função *hash*, assim, não haverá uma distribuição uniforme das palavras na tabela. Essas palavras foram escolhidas justamente com o intuito de a soma dos caracteres deixar um resto fixo na divisão por 30.

7. **Com tamanho 997 (primo) para a tabela de *hash* ao invés de 29, não deveria ser mais fácil? afinal, temos 997 posições para espalhar números ao invés de 29. Porque às vezes o *hash* por divisão com 29 *buckets* obtém melhores resultados do que com 997? Porque a versão com produtório é melhor? Por que este problema nao apareceu quando usamos tamanho 29?**

   Primeiramente, fazer uma tabela *hash* com a função soma dos caracteres $mod(997)$ não é muito razoável pois a soma dos caracteres de uma palavra não costuma ultrapassar esse valor. Dito isto, palavras que possuem a mesma soma de caracteres ocupam o mesmo *bucket* independente de a função soma ser $mod(29)$ ou $mod(997)$, de modo que ocorre concentração de palavras em alguns *buckets* devido ao viés do caso teste ou até mesmo à lingua utilizada. No entanto, palavras que possuem somas pouco obtidas ficam mais "isolados" na função $mod(997)$, isto é, há uma grande quantidade de *buckets* com poucos elementos ou nenhum, principalmente os que representam somas maiores, pois as palavras não tem a tendência de ser consideravelmente grandes. Enquanto isso na função $mod(29)$ as palavras são mais bem distribuídas, pois se trata de um número pequeno, assim, a soma dos caracteres de uma palavra geralmente é bem superior ao 29 e seus primeiros múltiplos, ciclando mais vezes na função módulo e implicando mais certamente na maioria dos *buckets* bem preenchidos e descrevendo uma distribuição aproximadamente uniforme.
   A versão com produtório atinge resultados mais satisfatórios pois o produto dos caracteres tende ser bem superior a 997, de modo que não ocorre o que foi explanado acima. Somado a isso, a quantidade de *buckets* consideravelmente maior torna a tabela com função *hash* produto $mod(997)$ mais desejável.

8. ***Hash* por divisão é o mais comum, mas outra alternativa é *hash* de multiplicação. É uma alternativa viável? Porque *hashing* por divisão é mais comum?**

   Sim, conforme visto nos resultados, o *hashing* por multiplicação teve pouca entropia perdida, mostrando que é uma alternativa viável. No entanto, *hashing* por divisão é mais comum pois o método da multi-

plicação tem tempo de computação lento para o cálculo da chave.

9. **Qual a vantagem de *Closed hash* sobre *Open hash*, e quando escolheríamos *Closed hash* ao invés de *Open hash*?**

   As vantagens do *Closed hash* sobre o *Open hash* são questões de memória. Enquanto o *Closed hash* aloca os elementos linearmente em sua memória e ocupa um espaço exclusivamente relativo à tabela, o *Open hash* aloca elementos em outras estruturas de dados que ocupam mais espaço na memória. Além disso, se é previamente conhecido o número de chaves e há poucas colisões, é preferível utilizar *Closed hash* pois é possível alocar exatamente a memória desejada.

10. **Suponha que um atacante conhece exatamente qual é a sua função de *hash* (o código é aberto e o atacante tem acesso total ao código), e pretende gerar dados especificamente para atacar o seu sistema (da mesma forma que o arquivo $mod(30)$ ataca a função de *hash* por divisão com tamanho 30). Como podemos implementar a nossa função de *hash* de forma a impedir este tipo de ataque?**

    Para esse fim, é de interesse o uso de funções de *hash* universais:

    (a) Utilizar um tamanho m primo para a tabela de espalhamento;
    (b) Deve-se tratar as *strings* como números na base m (elas podem ser tratadas inicialmente como números na base 26 ou 52 e então feita uma transformação de base). Assim, decompondo cada chave k (*string*) em um número de $r+1$ dígitos $k = k_0, k_1, ..., k_r >, k_i \in \{0, 1, ..., m-1\}$;
    (c) Escolher aleatoriamente (no próprio código) $a =< a_0, a_1, ..., a_r >, a_i \in \{0, 1, ..., m-1\}$;
    (d) Utilizar a função *hash* $h_a(k) = \sum_{i=0}^{r} a_i.k_i \ \ mod(m)$.

    Dessa forma, não está explicito em seus código qual a função de *hash* e, assim, é mais difícil que o indivíduo consiga atacar a tabela.

3 Referências

1. Bargal Sarah, *Universal Hashing*. Disponível em: `http://cs-www.bu.edu/faculty/homer/537/talks/SarahAdelBargal_UniversalHashingnotes.pdf`. Acesso em 26 de abr. 2020

2. Souza Jairo, *Hashing*, Disponível em: `http://www.ufjf.br/jairo_souza/files/2012/11/4-Hashing-Fun%C3%A7%C3%B5es.pdf`. Acesso em 26 de abr. 2020

3. *Hash Tables: Open vs Closed adressing*, Disponível em: `https://programming.guide/hash-tables-open-vs-closed-addressing.html`. Acesso em 26 de abr. 2020

4. *Open and Closed Hashing*, Disponível em: `http://www.brainkart.com/article/Open-and-Closed-Hashing_8039/`. Acesso em 26 de abr. 2020