Relatório de CTC-12 / 2023

Laboratório 01 - Hash Tables

Aluno

Marcel Versiani e Silva

Turma COMP.25

Professor

Luiz Gustavo Bizarro Mirisola

Instituto Tecnológico de Aeronáutica - ITA

1

1.1: Por que necessitamos escolher uma boa função de hashing, e quais as consequências de escolher uma função ruim?

A função de hashing é responsável por mapear uma chave para um índice da Hash Table, e se ela não for bem escolhida, pode ocorrer a colisão das chaves, que prejudica o desempenho dessa estrutura de dados. Uma função satisfatória é aquela que satisfaz a premissa do hashing uniforme simples, isto é, cada chave tem a mesma probabilidade de ser armazenada em qualquer uma das m posições da tabela independentemente das posições das outras chaves já armazenadas. Se a função escolhida for ruim, algumas posições tornam-se mais prováveis, aumentando o número de colisões e também a complexidade do algoritmo de busca, que nos piores casos pode chegar a $O(n)$ e a estrutura se degenera numa lista ligada.

1.2: Por que há diferença significativa entre considerar apenas o 1o caractere ou a soma de todos?

Pelas figuras abaixo, é possível verificar que ao considerar apenas o primeiro caractere como argumento da função de hashing (azul), a perda de entropia é significantemente maior se comparada à soma de todos os caracteres da string como argumento (laranja), vide Figura 1. Além disso, na Figura 2 observa-se que as entropias do caso em laranja sempre são maiores se comparadas às de azul, isso significa que considerar todos os caracteres implica em uma distribuição de probabilidades mais uniforme, isto é, as chaves estão melhor distribuídas entre todas as buckets da tabela.

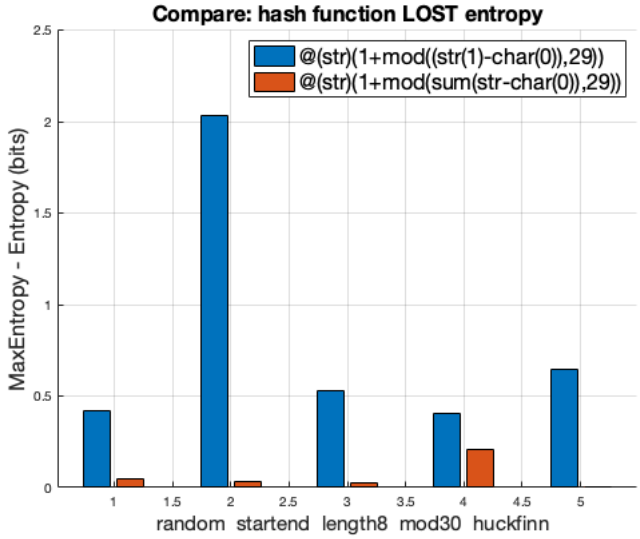

Figura 1: Perda de entropia para os dois casos analisados.

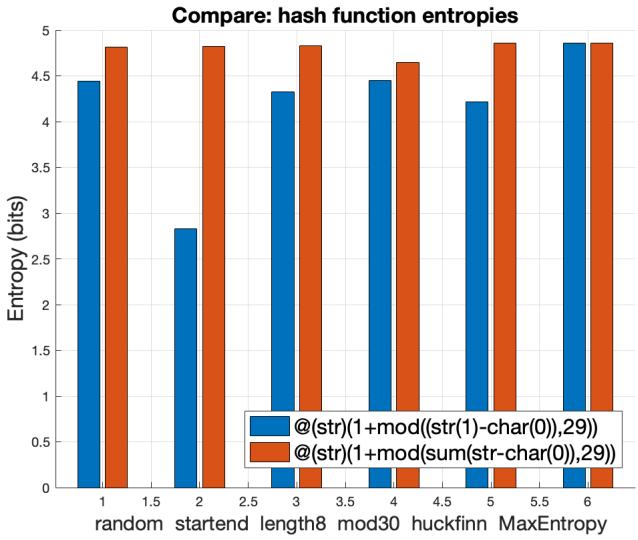

Figura 2: Entropia total dos dois casos analisados.

1.3: Por que um dataset apresentou resultados muito piores do que os outros, quando consideramos apenas o 1o caractere?

Pela Figura 3 abaixo, observa-se que o dataset *startend* obteve um desempenho bem pior na distribuição de chaves nas buckets se comparado com os outros, utilizando a função de hashing que considera apenas a primeira letra (azul). Isso se deve ao fato que esse dataset possui muitas palavras têm a primeira letra iguais, principalmente a letra c, e isso faz com que a distribuição entre a buckets esteja longe de uma distribuição uniforme.

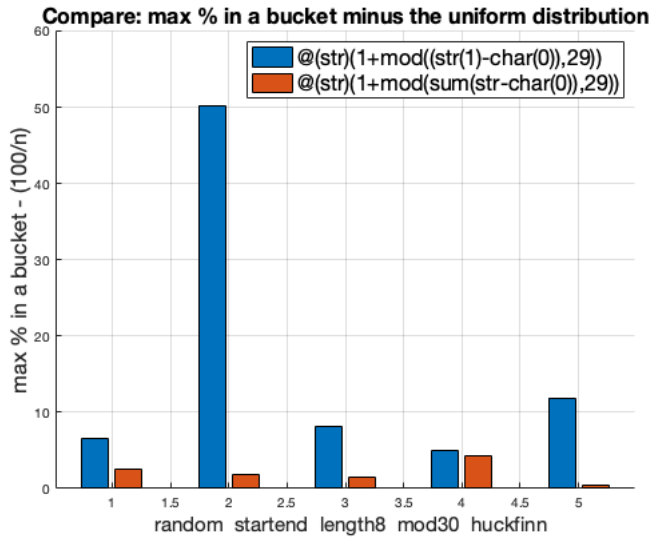

Figura 3: Comparação da fração máxima de ocupação na bucket subtraído da distribuição uniforme.

2

2.1: Com uma tabela de hash maior, o hash deveria ser mais fácil. Afinal temos mais posições na tabela para espalhar as strings. Usar Hash Table com tamanho

30 não deveria ser sempre melhor do que com tamanho 29? Porque não é este o resultado?

Na função de hashing utilizada, que usa o hash por divisão, a distribuição das chaves é mais uniforme quando utilizamos um número primo (azul e vermelho), isto é, as chaves estão mais bem distribuídas, o que leva a uma maior entropia. No caso da Figura 4 abaixo, utilizar um tamanho de 30 (amarelo e roxo) aumenta o número de colisões, pois é um número com vários divisores, o que aumenta a chance da chave ter divisores em comum com o tamanho da tabela, restringindo a quantidade de buckets para múltiplos desses divisores, gerando uma distribuição menos uniforme.

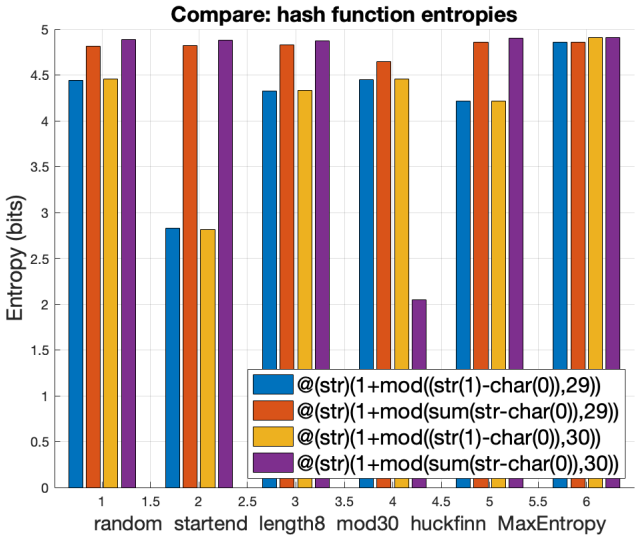

Figura 4: Comparação das entropias entre os casos analisados.

2.2: 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?

O tamanho primo evita o problema do compartilhamento de mesmos divisores comuns entre um número não primo e o tamanho da Hash Table, isto é, se usarmos um tamanho primo diminuímos as chances de haver divisores em comum com as chaves. Com isso, ele distribui os restos das divisões de maneira mais uniforme, minimizando a formação de padrões que podem estar contidos no dataset devido a fatores em comum, então algumas buckets não serão utilizadas com chaves que têm fator comum com o 30, o que forma padrões em outras buckets. Contudo, um tamanho não primo pode ser usado sem muitos problemas com a condição de não ter fatores não primos maiores do que 20, o que é satisfeito com o 30, por isso seu comportamento na maioria dos casos é satisfatório como observado na Figura 5.

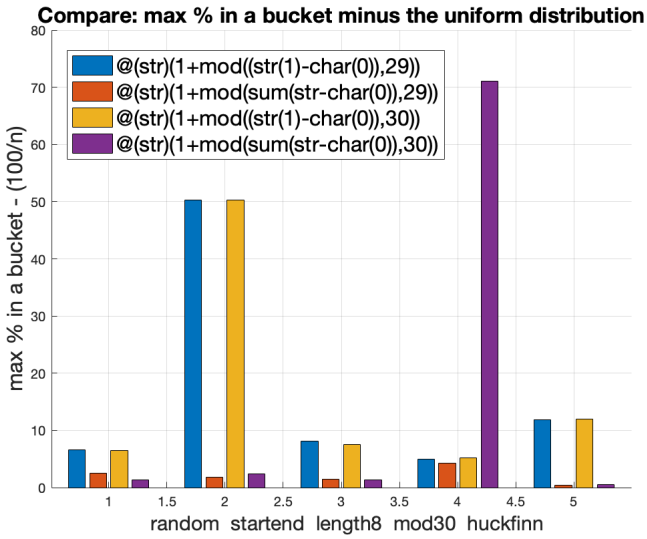

Figura 5: Comparação da fração máxima de ocupação na bucket subtraído da distribuição uniforme.

2.3: Note que o arquivo mod30 foi feito para atacar um hash por divisão de tabela de tamanho 30. Explique como esse ataque funciona: o que o atacante deve saber sobre o código de hash table a ser atacado, e como deve ser elaborado o arquivo de dados para o ataque.

Pela Figura 6 abaixo, percebemos que o ataque foi realizado por meio do sobrecarregamento da bucket de índice 4, ou seja, o dataset usado pelo atacante consistiu em strings que, em sua maioria, deixam resto 4 na divisão por 30, que é o tamanho da hash table. Com isso, o atacante deve saber como funciona a função de hashing da tabela e, dado que foi um hashing por divisão, ele também precisou saber o tamanho da tabela, e com isso elaborar um dataset para sobrecarregar uma determinada bucket. Desse modo ele praticamente degenerou a estrutura numa lista ligada.

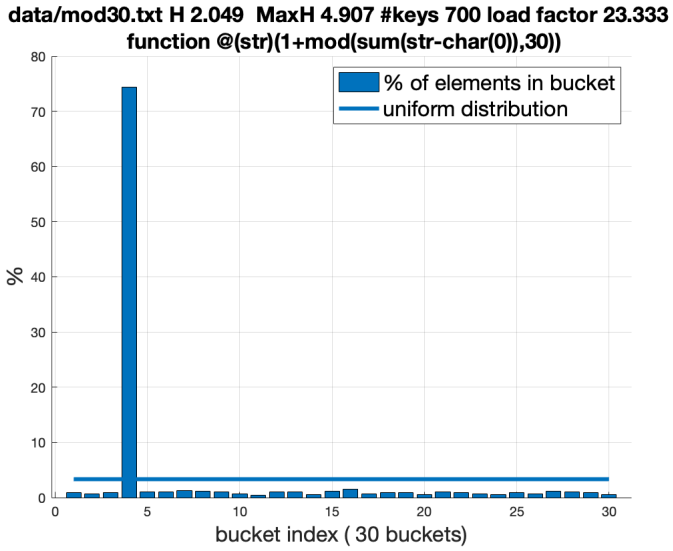

Figura 6: Distribuição das chaves por bucket, em fração.

3

3.1: Com tamanho 997 (primo) para a tabela de hash ao invés de 29, não deveria ser melhor? Afinal, temos 997 posições para espalhar números ao invés de 29. Porque às vezes o hash por divisão com 29 buckets apresenta uma tabela com distribuição mais próxima da uniforme do que com 997 buckets?

Analisando as Figuras 7 e 8 abaixo, percebe-se que há uma maior uniformidade na distribuição para a tabela de tamanho 29 se comparada à de tamanho 997 utilizando o arquivo *length8.txt*. Analisando o arquivo, ele contém várias palavras que possuem tamanhos iguais, no caso tamanho 8. Logo, ao realizar o hash por divisão, as somas de seus caracteres possuem valores relativamente próximos entre si mas da mesma ordem de grandeza de um número grande como o 997, porém para um número como o 29 essas somas não muito diferentes estão distantes de 29, então o resto da divisão fica mais uniformizado e distribui melhor as chaves.

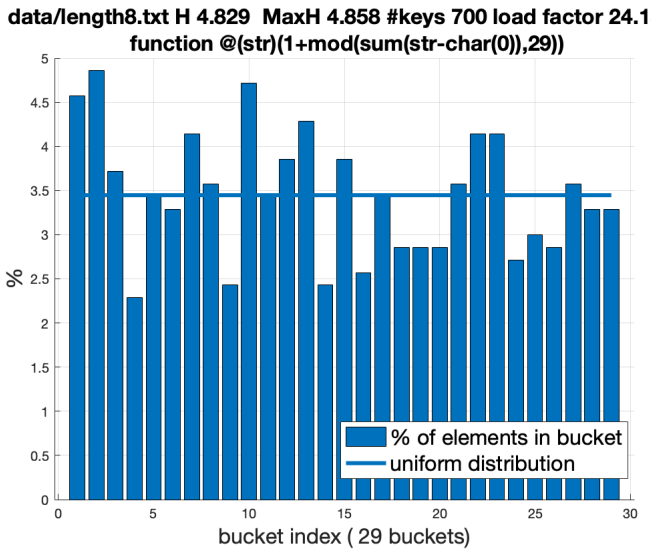

Figura 7: Distribuição para hash por divisão com tabela de tamanho 29 usando *length8.txt*.

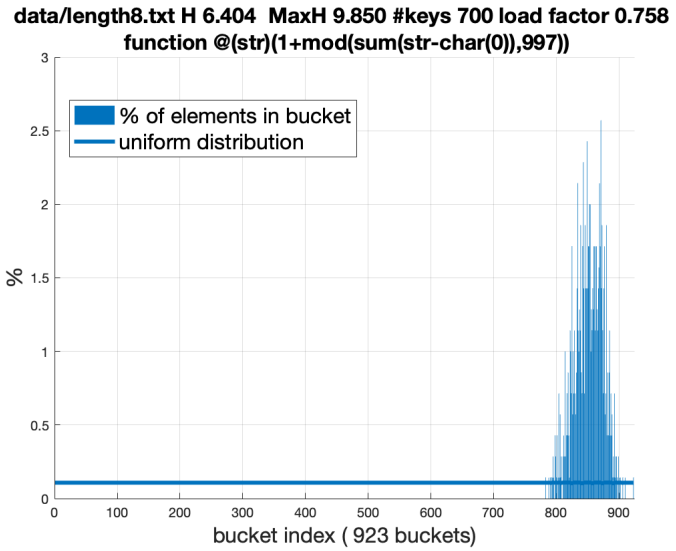

Figura 8: Distribuição para hash por divisão com tabela de tamanho 997 usando *length8.txt*.

3.2: Por que a versão com produtório (prodint) é melhor?

Quando utiliza-se o produtório do valor dos caracteres, esse número se torna bastante distante de 997 se comparado ao método da divisão, mesmo utilizando palavras de tamanho restrito a 8. Com isso, o resultado numericamente grande consegue ser bem distribuído ao utilizarmos o resto da divisão por 997, viabilizando o uso de uma tabela grande.

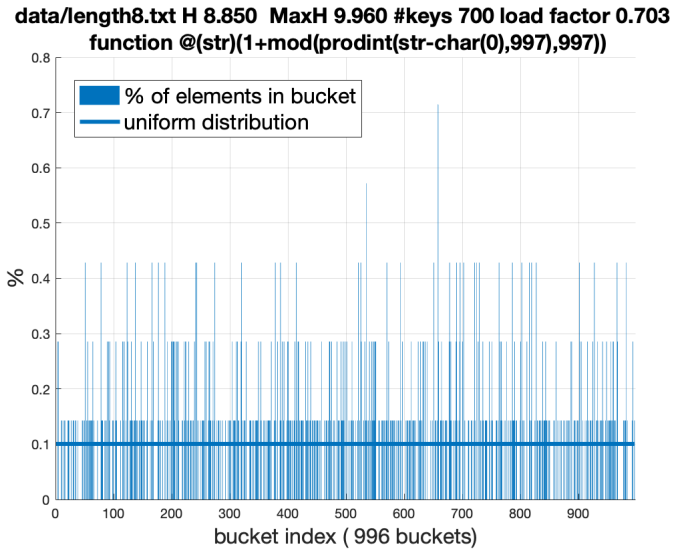

Figura 9: Distribuição para hash por produtório com tabela de tamanho 997 usando *length8.txt*.

3.3: Por que este problema não apareceu quando usamos tamanho 29?

Porque apesar do tamanho das palavras ser fixo em 8, a soma dos caracteres convertidas em número inteiro (que no nosso caso começa em 65 e termina em 122 utilizando a tabela *ASCII*), a qual não difere significativamente entre as palavras por terem tamanhos iguais, já é grande o suficiente se comparado ao número 29. Com isso, o resto da divisão pelo tamanho da tabela para cada palavra é uniforme o suficiente para termos bons resultados utilizando o hashing por divisão mesmo em casos específicos como o arquivo *length8.txt*, dado um tamanho de tabela suficientemente pequeno.

4

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

O hashing por multiplicação é feito multiplicando a chave por uma constante fracionária, multiplicar a parte fracionária deste valor pelo tamanho da tabela e utilizar os bits mais significativos da maneira que melhor se adequar para a utilização da Hash Table. É um método viável de hashing pois o tamanho da tabela pode não ser primo e isso não afetará criticamente o hashing, além de uniformizar melhor a distribuição se comparado ao método da divisão em alguns casos. Como pode-se observar na Figura 10, para uma tabela de tamanho 29 ela desempenhou tão bem (vermelho) quanto ao método da divisão (azul), sendo mais uniforme em alguns datasets e menos em outros. Contudo, ela acaba sendo menos comum por ser mais difícil de implementar, além de precisar escolher bem a constante fracionária para evitar colisões que aparecem com padrões de multiplicação e ser geralmente mais lento computacionalmente.

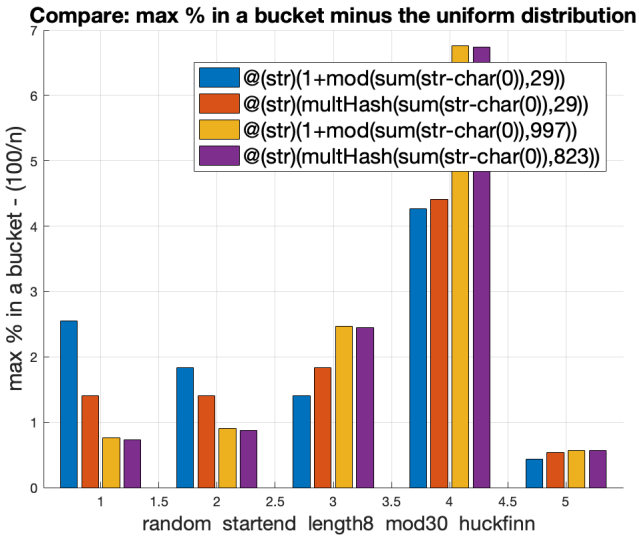

Figura 10: Comparação da fração máxima de ocupação na bucket subtraído da distribuição uniforme para os casos analisados.

5

Qual a vantagem de Closed Hash sobre OpenHash, e quando escolheríamos Closed Hash ao invés de Open Hash?

A principal vantagem do Closed Hash sobre o Open Hash é que o Closed Hash permite o armazenamento de mais itens na tabela hash do que o Open Hash sem afetar significativamente o tempo de busca dada uma boa função de hashing condizente com os dados a serem armazenados e o tamanho da tabela. Contudo, a desvantagem do Closed Hash é que ele pode consumir mais memória do que o Open Hash, pois cada posição pode armazenar mais de um item e uma estrutura de dados adicional é necessária para lidar com as colisões, como uma lista ligada que também armazena ponteiros. Em relação à escolha, o Closed Hash é preferível quando existe uma grande quantidade de itens para armazenar, mesmo com repetições, e a tabela precisa ser otimizada para o tempo de busca, mesmo que isso signifique um pouco mais de uso de memória. O Open Hash é preferível quando a memória é um recurso escasso e a quantidade de itens a serem armazenados é relativamente pequena para não ter que recalcular várias vezes o hash code para evitar as colisões.

6

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 mod30 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?

Podemos utilizar uma abordagem conhecida como Hash Universal, que é uma maneira eficaz de evitar esses tipos de ataques, escolhendo a função hash aleatoriamente de várias funções disponíveis, de um modo que seja independente das chaves que serão armazenadas. Nessa abordagem, no início da execução selecionamos a função hash aleatoriamente de uma classe de funções cuidadosamente projetada e essa aleatorização garante que nenhuma entrada isolada cairá sempre no comportamento do pior caso. Como selecionamos a função hash aleatoriamente, o algoritmo se comporta de modo diferente em cada execução ainda que a entrada seja a mesma,

garantindo um bom desempenho do caso médio para qualquer entrada.