Relatório de CES-12

Lab 01 - Hash

Aluno: Fernando de Moraes Rodrigues

Turma Comp-22

Professor Luiz Gustavo Bizarro Mirisola

Instituto Tecnológico de Aeronáutica – ITA

Perguntas

(1.1). Porque necessitamos escolher uma boa função de hashing, e quais as consequências de escolher uma função ruim?

Uma função de Hash é considerada satisfatória quando cada chave tem igual probabilidade de passar para qualquer das m posições por uma operação de hash, independentemente da posição que qualquer outra chave ocupou após o hash, ou seja, quando satisfaz a premissa do hashing uniforme simples.

Se escolhermos uma função de hash ruim, a escolha de algumas posições torna-se mais provável (como exemplificado na questão 4 desse relatório), aumentando o número de colisões e também a complexidade do tempo de busca.

(1.2). Porque há uma diferença significativa entre considerar apenas o 1º caractere ou a soma de todos?

Conforme pode ser visto na figura 1 a seguir, para o caso de considerarmos apenas o primeiro caractere (em azul) a perda de entropia é consideravelmente maior que no caso de considerarmos a soma de todos os caracteres (em laranja).

Com isso, temos que o segundo caso apresenta uma distribuição de probabilidades mais uniforme, o que implica em maior entropia, conforme pode ser visto na figura 2 a seguir.

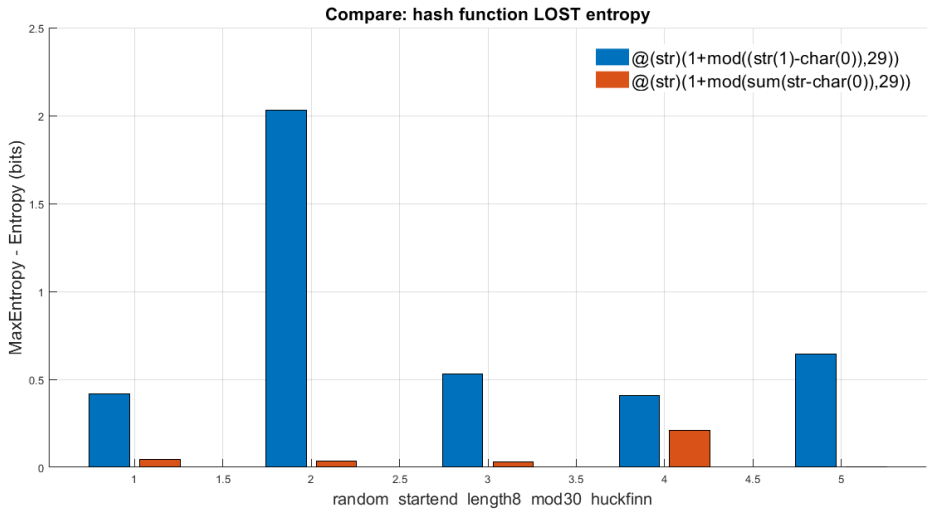

Figura 1. *Gráfico que representa a perda de entropia para os casos analisados.*

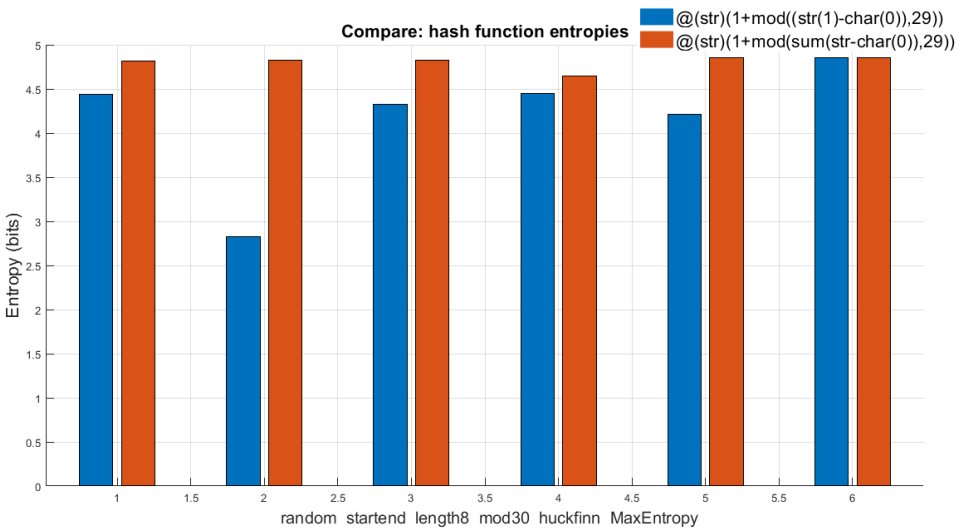

Figura 2. *Comparação das entropias dos casos analisados.*

(1.3). Porque um dataset apresentou resultados muito piores do que os outros, quando consideramos apenas o 1º caractere?

O dataset "startend" possui uma grande quantidade de elementos começados com a letra C, sendo, portanto, direcionadas para a mesma posição da tabela da função de Hash. Com isso, temos uma distribuição menos balanceada e menos entrópica conforme verificado nas figuras 1 e 2 acima e na figura 3 a seguir.

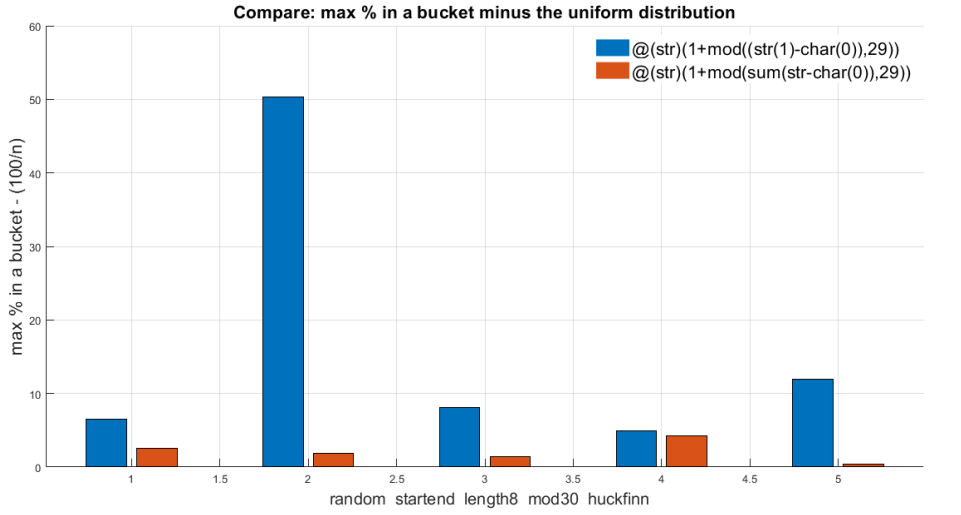

Figura 3. *Comparação entre a porcentagem máxima de ocupação e a distribuição uniforme.*

(2.1). 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? (Atenção: o arquivo mod30 não é o único resultado onde tamanho 30 é pior do que tamanho 29)

No método de hash por divisão, que foi o utilizado nessa análise, a distribuição dos dados na tabela tende a ser mais bem-sucedida se o tamanho for um número primo, como o 29, do que se o tamanho for um número com muitos divisores, como o 30, uma vez que os restos das divisões tendem a ser mais bem distribuídos caso o tamanho seja um número primo.

O comparativo das entropias das hashtable de tamanhos 29 e 30 pode ser visto na figura 4 a seguir.

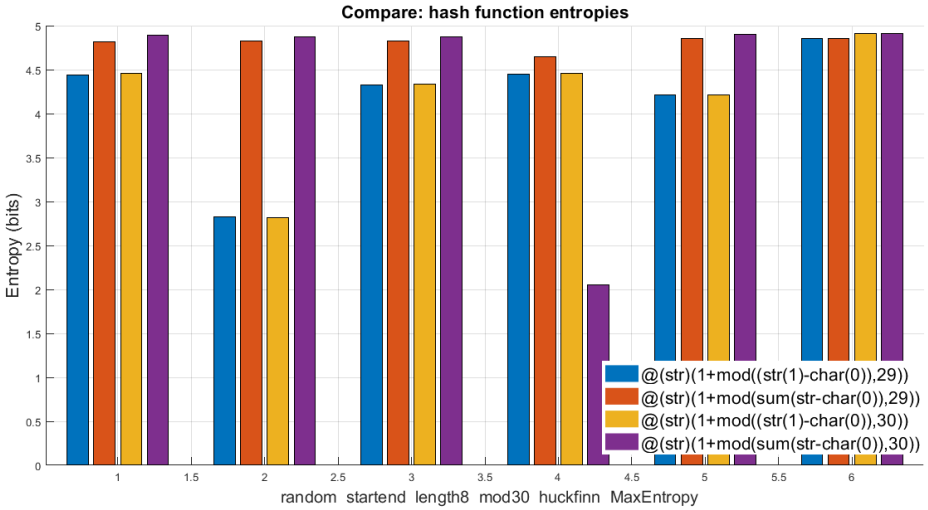

Figura 4. *Comparativo entre as entropias.*

(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?

Números primos tendem a distribuir os restos das divisões de maneira mais uniforme do que números não primos, minimizando a formação de padrões de distribuição, que podem ocorrer com os números não primos.

Divisores não primos podem trabalhar bem com a condição de que não tenham fatores não primos maiores do que 20 (Lum et al., 1971). Como o número 30 se encaixa nessas condições, seu comportamento comparado com o 29 é satisfatório.

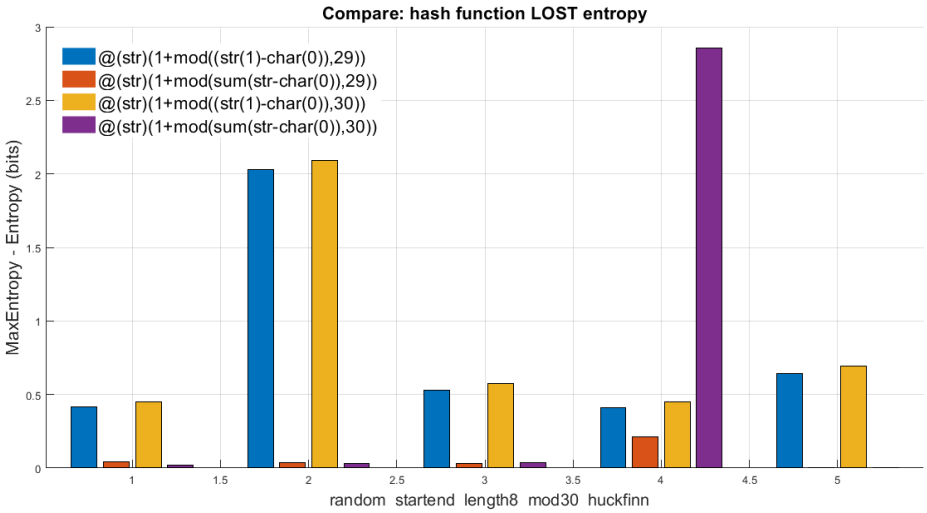

Figura 5. *Comparativo das perdas de entropias para tamanho 29 e 30.*

(2.3). Note que o arquivo mod30 foi feito para atacar um hash por divisão de tabela de tamanho 30. Como este ataque funciona? (Dica: plote a tabela de hash para a função correta e arquivo correto. Um exemplo de como usar o código está em checkhashfunc)

Conforme a figura 5 a seguir, o ataque consistiu em sobrecarregar certa posição da hashtable, adicionando diversos elementos que deixam resto 4 na divisão por 30.

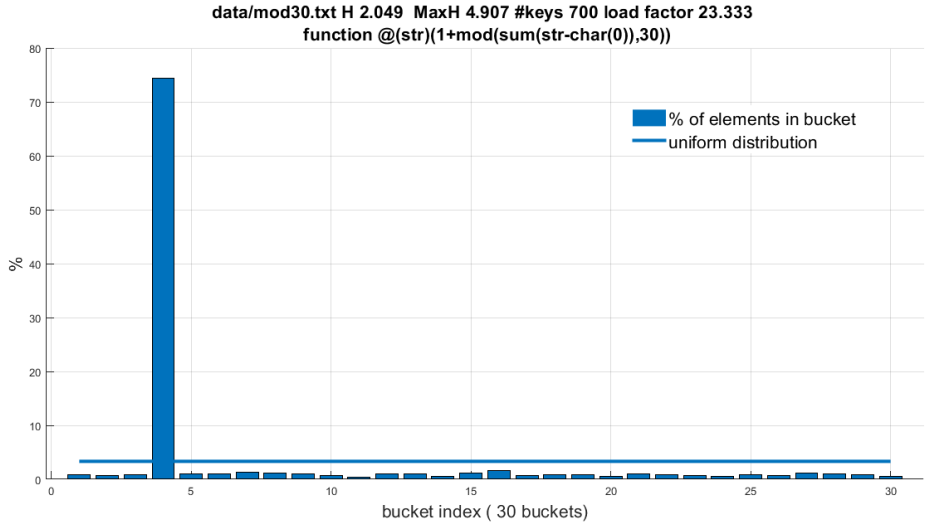

Figura 6. *Distribuição por bucket.*

(3). 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 (prodint) é melhor? Porque este problema não apareceu quando usamos tamanho 29?

Dica: plote a tabela de hash para as funções e arquivos relevantes para entender a causa do problema. Usar length8.txt e comparar com os outros deve ajudar a entender. Isto é um problema comum com hash por divisão.

Dica: prodint.m (verifiquem o código para entender) multiplica os valores de todos os caracteres, mas sem permitir perda de precisão decorrente de valores muito altos: a cada multiplicação os valores são limitados ao número de buckets usando mod.

Analisando as figuras 7 e 8 a seguir, percebemos que a tabela hash com tamanho 29 apresenta uma distribuição mais uniforme do que a com tamanho 997 para o arquivo "lenght8.txt". Analisando o conteúdo do arquivo, percebe-se que ele é formado por várias palavras de um mesmo tamanho não muito longo (8 letras). Sendo assim, devem apresentar somas com valores próximos entre si, mas distantes de 997 e mais próximas de 29, inviabilizando a distribuição baseada no resto da divisão por 997.

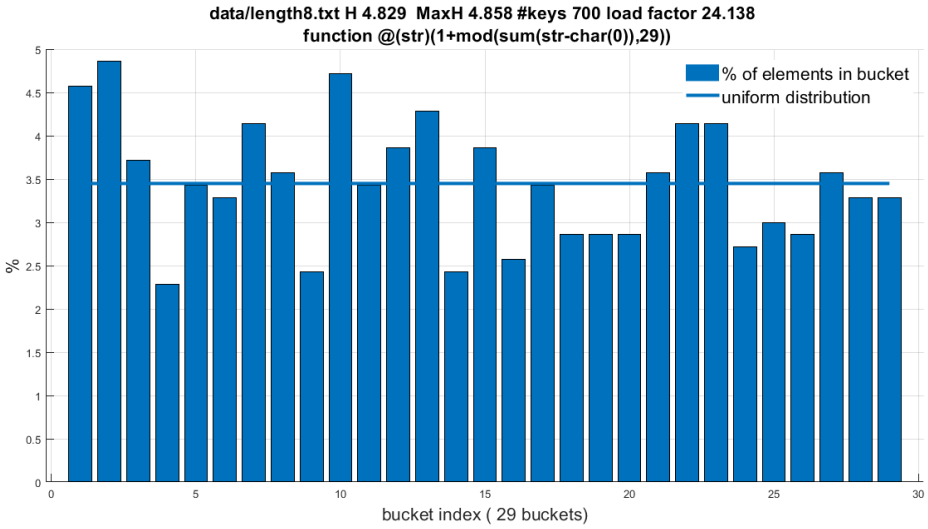

Figura 7. *Distribuição para hash por divisão de tamanho 29 para o arquivo "length8.txt".*

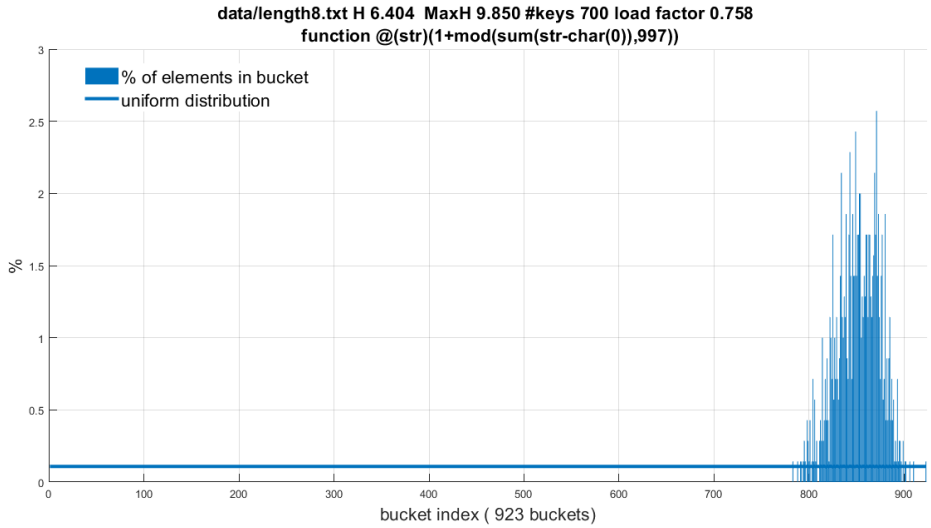

Figura 8. *Distribuição para hash por divisão de tamanho 997 para o arquivo "length8.txt".*

Quanto ao uso da função produtório, obteremos resultados numericamente maiores do que pelo método da soma, o que torna mais viável o uso da hashtable de tamanho 997, distribuindo melhor conforme visto na figura 9 a seguir.

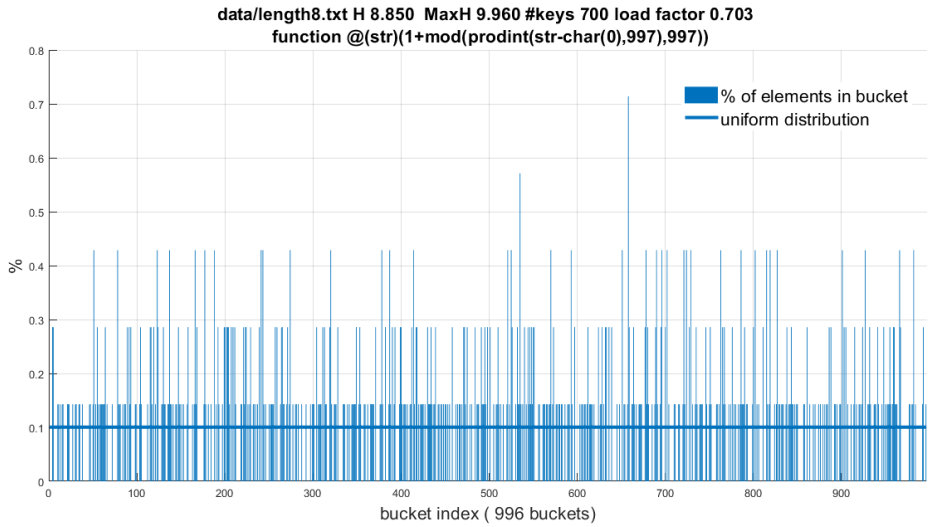

Figura 9. *Distribuição para hash por produtório de tamanho 997 para o arquivo "length8.txt".*

(4) Hash por divisão é o mais comum, mas outra alternativa é hash de multiplicação (NÃO É O MESMO QUE prodint.m). É uma alternativa viável? Porque hashing por divisão é mais comum?

Quando utilizamos o método de divisão (Função Hash: k mod (m)), em geral evitamos certos valores de m, por exemplo, potências de 2, pois, se $m = 2^p$, então h (k) será somente o grupo de p bits de ordem mais baixa de k. Outro caso problemático desse método ocorre se m é um número par, pois h (x) será sempre par quando x for par e

sempre ímpar quando x for ímpar. Desse modo, uma vez que esse método exige uma única operação de divisão, o hash por divisão é bastante rápido e, portanto, mais comum.

Por outro lado, o método de multiplicação apresenta como vantagem o fato de que o valor de m não é crítico. Em geral, nós o escolhemos de modo a ser uma potência de 2 ($m = 2^p$ para algum inteiro p). Entretanto, por ser um processo realizado em duas etapas (Primeiro, multiplicamos a chave k por uma constante A na faixa $0 < A < 1$ e extraímos a parte fracionária de kA. Em seguida, multiplicamos esse valor por m e tomamos o piso do resultado. Função Hash: $h(k) = \lfloor m (kA \bmod 1) \rfloor$), temos uma computação lenta para cálculo da chave.

(5). Qual a vantagem de Closed Hash sobre Open Hash, e quando escolheríamos Closed Hash ao invés de Open Hash? (Pesquise! É suficiente um dos pontos mais importantes)

A principal vantagem de Closed Hash sobre Open Hash é que no primeiro não é necessário calcular outro Hash Code, pois todos ficam na mesma entrada que é a lista encadeada, apesar de precisar de mais memória, pois cada colisão é necessária alocação de memória dinâmica.

Devemos escolher o Closed Hash se a tabela estiver muito cheia, pois a quantidade de colisões pode ser muito alta tornando a tabela inviável, uma vez que no Open Hash há a necessidade de recalcular o Hash Code (rehashing).

(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? Pesquise e explique apenas a ideia básica em poucas linhas (Dica: a estatística completa não é simples, mas a ideia básica é muito simples e se chama Universal Hash)

A abordagem conhecida como Hash Universal a única maneira eficaz de melhorar a situação, escolhendo a função hash aleatoriamente, de um modo que seja independente das chaves que realmente serão armazenadas. Além disso, tal técnica pode resultar em um desempenho demonstravelmente bom na média, não importando as chaves escolhidas pelo adversário.

No hashing universal, no início da execução selecionamos a função hash aleatoriamente de uma classe de funções cuidadosamente projetada. A aleatorização garante que nenhuma entrada isolada evocará sempre o comportamento do pior caso. Como selecionamos a função hash aleatoriamente, o algoritmo poderá se comportar de modo diferente em cada execução ainda que a entrada seja a mesma, garantindo um bom desempenho do caso médio para qualquer entrada.