CTC-12 - Projeto e Análise de Algoritmos

Lab 01 - Experimento com funções de *hash*

Aluno:

Ulisses Lopes da Silva

COMP 26

Professores:

Luiz Gustavo Bizarro Mirisola

22/03/2024

Instituto Tecnológico de Aeronáutica – ITA

Divisão de Ciência da Computação

Sumário

1 Questão 1

1.1

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

Resposta: O objetivo de uma estrutura de dados *hash* é tornar a busca por dados igual ou aproximadamente igual a O(1). Dessa forma, a escolha de uma estrutura *hash* adequada é fundamental pois, dependendo da quantidade de buckets, a busca na *hash table* pode ser muito custosa (ex.: muitos dados e poucos buckets, de modo que haverá muitas colisões ao se tentar inserir ou procurar um dado na tabela). Por outro lado, uma função de *hash* com o *load factor* muito baixo ($n << k$) pode se configurar num desperdício de memória, já que há pouquíssimos dados para uma tabela muito grande. Não obstante, o "fator da *hash table*"também é muito importante. Em geral, escolhe-se um número primo devido a este ter apenas 2 divisores, e, assim, há uma maior probabilidade de a estrutura retornar um espalhamento mais uniforme dos dados (considerando uma estrutura de *hash* por divisão, por exemplo).

1.2

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

Resposta: A função que considera apenas o primeiro caractere para a definição de qual *bucket* irá alocar a *string* tem uma tendência muito maior de gerar colisões: basta que várias *strings* iniciem com o mesmo caractere. Dessa forma, somar o código ASCII de todos os caracteres e dividir o valor obtido pelo fator tem uma probabilidade bem maior de garantir um espalhamento mais uniforme.

1.3

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

Resposta: Justamente pelo que foi descrito em 1.2. O *dataset* "startend.txt"possuía muitas palavras iniciadas com o mesmo caractere. Como a estrutura era *hash* por divisão, ao se dividir o código ASCII de várias palavras cuja letra inicial era a mesma, obtinha-se a mesma chave para o *bucket*, e, assim, várias *strings* ficaram no mesmo *bucket*, perdendo entropia da distribuição e gerando muitas colisões, fazendo com que o *hashing* não fosse eficiente.

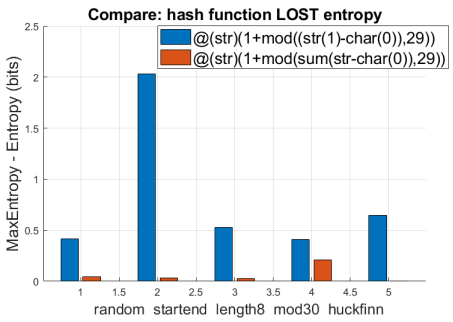

Figura 1: Resultados por *dataset*: entropia perdida

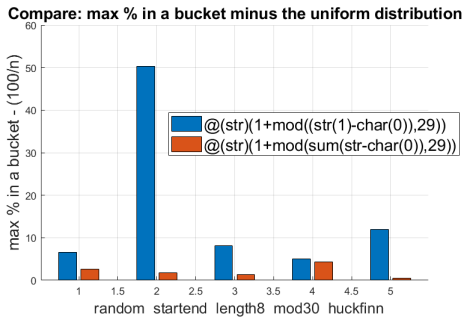

Figura 2: Resultados por *dataset*: lotação de *buckets*

2 Questão 2

2.1

Com uma tabela de *hash* maior, o *hashing* 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?
(Atenção: o arquivo "mod30.txt"não é o único resultado em que usar tamanho 30 é pior do que tamanho 29)

Resposta: A *Hash Table* com tamanho 30 é pior porque, ao usar o *hash* por divisão, dividimos o valor obtido na soma dos caracteres por 30 e utilizamos o resto dessa divisão como chave. Ocorre que o número 30 tem muitos divisores (a saber: 1, 2, 3, 5, 6, 10, 15, 30), ao contrário de um primo. Ou seja, existem muito mais chances de ocorrer restos iguais num *dataset* aleatório, o que culminaria em vários elementos no mesmo *bucket* e, consequentemente, várias colisões, diminuindo a entropia. Esse fato é ainda mais evidente com o *dataset* "mod30.txt", em que a maioria dos caracteres retorna um número divisível por 30, ou que, pelo menos, gera o mesmo resto.

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?

Resposta: Um número primo possui apenas dois divisores: o 1 e ele mesmo. Dessa forma, ao se efetuar a divisão por ele, dificilmente muitos números gerarão o mesmo resto. Em outras palavras, a distribuição vai tender a ser mais uniforme do que um número que possua muitos divisores. Por exemplo: ao alocarmos numa tabela *strings* cujos caracteres somem 120, 90, 60 e 30, todos irão para a mesma chave, já que $(n)mod30 \equiv 0$. Contudo, por 29, as chaves seriam, respectivamente, 4, 3, 2 e 1.

No exemplo, contudo, essa diferença se torna pequena devido ao *load factor* alto devido aos *datasets*. Por conterem quantidades muito maiores que 29, a tendência é ocorrerem muitas colisões (considerando que estamos supondo uma amostra bem distribuída), já que $n >> k$. Desse modo, o fato de 29 ser primo acaba não sendo tão relevante, apesar de ainda ser superior a um número com vários divisores.

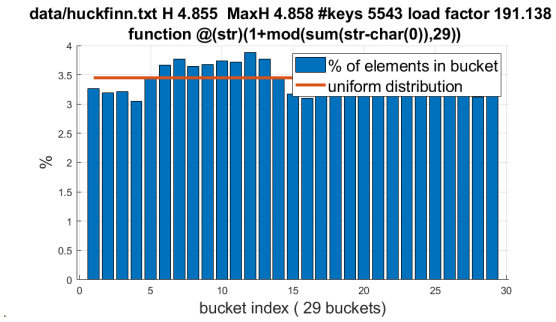

Figura 3: Resultados do *dataset* 'random.txt': mod29

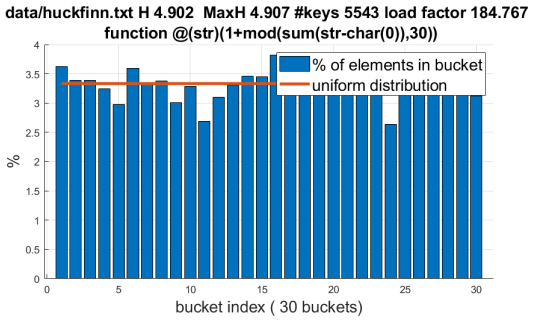

Figura 4: Resultados do *dataset* 'random.txt': mod30

2.3

Note que o arquivo "mod30.txt"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.
Dica: use "plothash.h"para plotar a ocupação da tabela de *hash* para a função correta e arquivo correto. Um exemplo de como usar o código está em em "checkhashfunc.m".

Resposta: Para atacar um *hash* por divisão, o atacante deve, primeiramente, saber a quantidade de *buckets* da tabela em que serão alocados os elementos. No caso em questão, 30 é o fator. Em seguida, ele deve saber qual será o cálculo realizado para se obter a *hash key*, que será dividida pelo fator 30 e retornará um resto, que será o índice do *bucket*. No caso em questão, o método de obtenção é a soma dos valores de cada caractere na tabela ASCII, constante do *dataset*. De posse dessas informações, o atacante pode gerar um *dataset* composto por dados que satisfaçam essas condições ou que gerem um resto específico. No arquivo do lab, por exemplo, ao se combinat a função *hash* de divisão por 30 junto com o *dataset* "mod30.txt", nota-se que a quase totalidade das palavras geram chaves que, divididas por 30, deixam resto 3 e são somadas a 1, tornando o *hash* ineficiente e com muitas colisões no *bucket* 4.

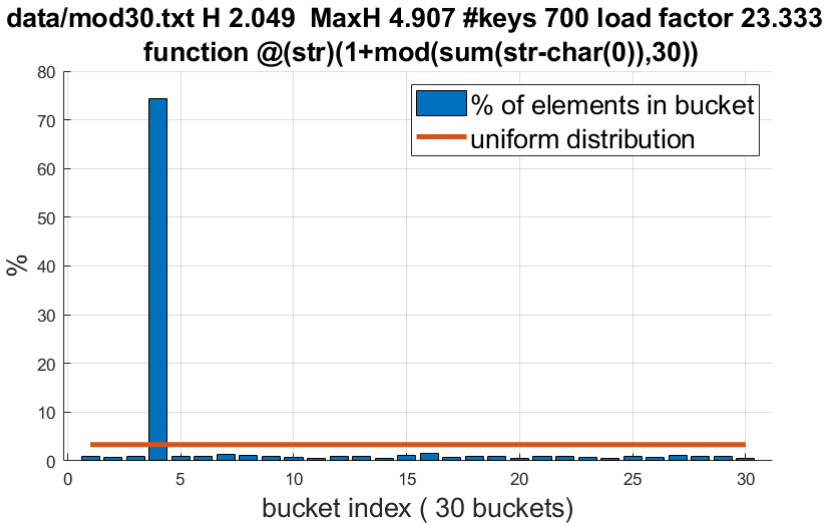

Figura 5: Função Hash de divisão por 30 com *dataset* 'mod30.txt'

3 Questão 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?

Resposta: O *hash* por divisão de 997 tendeu a se mostrar menos uniforme com os *datasets* "lenght8.txt"e "mod30.txt". Isso pode ter sido ocasionado pela forma de distribuição dos dados nesses arquivos, ou mesmo pela escolha dos dados nele presentes, gerando muitas colisões e uma concentração maior de elementos nos mesmos *buckets*. A escolha do fator é apenas uma parte da questão como um todo, de modo que ela sozinha não é a única responsável pela eficiência da distribuição. Por exemplo: se no conjunto de dados houver vários anagramas de strings, essas palavras podem ser alocadas no mesmo índice (em relação à função de estudo de que trata este relatório). não obstante, o fator de carga para 997, considerando o número de dados do arquivo (700 palavras), o fator de carga fica próximo de mais próximo de 0.7 a 0.8, o que, em tese, seria bom. Contudo, o fato de as palavras do arquivo length8 terem 8 caracteres faz as chaves se alternarem num range de, no mínimo, 520 ($AAAAAAAA = 65 * 8$), fato que pode concentrar as palavras na metade final dos *buckets*, já que $700 < 997$.

3.2

Porque a versão com produtório (prodint) é melhor?

Resposta: O *hash* por multiplicação se mostra mais eficiente em uniformizar o espalhamento porque, como há 997 espaços, há um melhor espalhamento utilizando a multiplicação do valor obtido na divisão. Sendo o fator de carga próximo de 1, há maior aleatoriedade na distribuição e, consequentemente, melhor espalhamento.

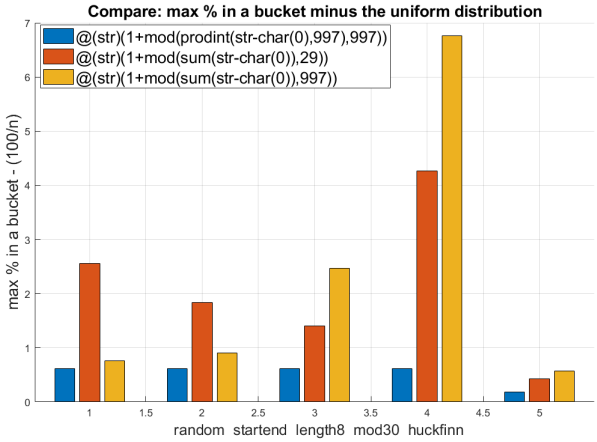

Figura 6: Função Hash de divisão. 997 *buckets*

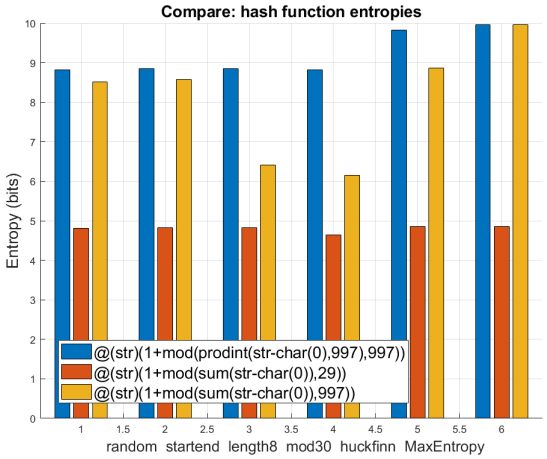

Figura 7: Função Hash com prodint

3.3

Porque esse problema não apareceu quando usamos tamanho 29?
Dica 1: plote a tabela de *hash* para as funções e arquivos relevantes para entender a causa do problema - não está visível apenas olhando a entropia. Usar o arquivo "length8.txt"e comparar com os outros deve ajudar a entender. Isto é um problema comum com *hash* por divisão.
Dica 2: 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.

Resposta: Porque o fator de carga, para o 29, era bastante alto, próximo do próprio número de 29, considerando os arquivos discrepantes (lenght8 e mod30). Dada a quantidade elevada de termos, a distribuição ao final tende a ser mais uniforme, mesmo no *hash* por divisão.

4 Questão 4

***Hash* por divisão é o mais comum, mas outra alternativa é *hash* de multiplicação (NÃO É O MESMO QUE prodint.m; verifiquem no livro-texto do Cormen). Esta é uma alternativa viável? Por que *hash* por divisão é mais comum?**

Resposta: Pelo resultado dos testes, não é tão viável, tendo em vista que a função ainda se mostra bastante não linear, comparado ao espalhamento uniforme. todavia, caso se admita alguma perda no tempo de busca, é possível usar esse método.

Não obstante, a função de *hash* por multiplicação funciona em duas etapas. Segundo Cormen (2009), uma vantagem do método de multiplicação é que o valor de m (que, em geral, é escolhido como sendo uma potência de 2) não é crítico. Contudo, o *hash* por divisão é mais comum provavelmente devido à sua fácil implementação e entendimento, além de ser, em geral, bastante rápido (Cormen, 2009).

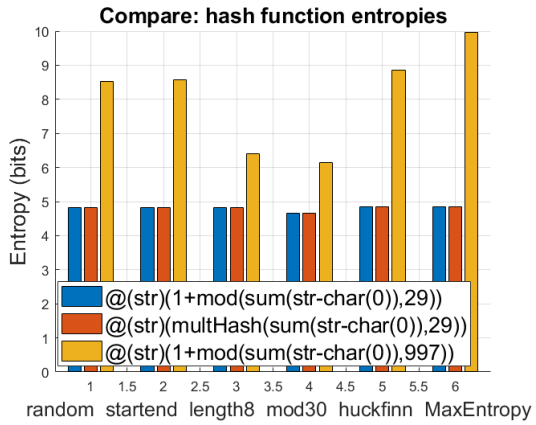

Figura 8: Função Hash com multiplicação - Entropia

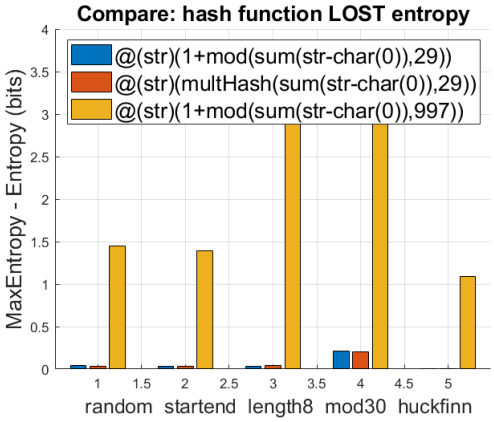

Figura 9: Função Hash com multiplicação - Entropia perdida

5 Questão 5

Qual é a vantagem de *Closed Hash* sobre *Open Hash*, e quando escolheríamos *Closed Hash* em vez de *Open Hash*? (Pesquise! É suficiente um dos pontos mais importantes).

Resposta: O *Closed Hash* nos dá um acesso direto aos dados da tabela, bem como não desperdiça memória com ponteiros adicionais para listas ligadas. O *Closed Hash* se torna uma opção melhor que o *Open Hash* nos casos em que não há movimentação dos dados da tabela (no sentido de que os dados serão prioritariamente, para consultas); em que se precise que o tempo de busca seja O(1) ou muito próximo disso; e em que se saiba que o fator de carga não seja tão próximo de 1, a fim de evitar muitas colisões.

6 Questão 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.txt"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*.

Resposta: Segundo Cormen (2009), a única maneira eficaz de melhorar a situação é escolher a função *hash* aleatoriamente, de um modo que seja independente das chaves que realmente serão armazenadas. Essa abordagem, que é o *Universal Hashing*, pode resultar em bom desempenho na média, não importando as chaves escolhidas pelo adversário. Em outras palavras, deve-se selecionar um conjunto de funções cuidadosamente projetadas e, ao longo da inserção e/ou procura dos dados, selecionar essas funções de modo aleatório. Dessa forma, conforme preveem os resultados estatísticos, a média deve representar uma distribuição relativamente uniforme.

7 Referências Bibliográficas

[1]. **CORMEN**, Thomas M. Algoritmos - Teoria e Prática. 3a edição, Editora ELSEVIER, Rio de Janeiro, RJ, 2012.