Bernardo Hoffmann da Silva
ITA COMP 24
Professor: Luiz Gustavo Bizarro Mirisola

(1.1) (pergunta mais simples e mais geral) porque necessitamos escolher uma boa função de hashing, e quais as consequências de escolher uma função ruim?

1.1) Precisamos escolher uma boa função de hash para evitarmos que muitos termos que iremos guardar na hashtable tenham o mesmo hash. Isso acarretaria em casos ruins em que a busca se daria apenas em uma lista ligada, e o tempo de busca seria O(n). Uma função hash ideal é aquela cujos diferentes possíveis resultados de hash são igualmente prováveis para todas as entradas da tabela.

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

1.2) Pois considerando apenas um caractere, a função hash irá operar apenas com o valor desse caractere. Isso gera uma limitação nas entradas a apenas o número de caracteres possíveis para iniciar as strings. Considerando todos os caracteres, há uma variedade muito maior de possibilidades, podendo-se assim aproveitar melhor o tamanho da tabela.

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

1.3) Pois neste dataset existiam muitas palavras diferentes que começavam com a mesma letra. Isso em um hash que considera todos os caracteres não apresentaria problema. Porém, considerando apenas o primeiro, todas essas palavras caem em um único bucket e geram uma lista de tamanho muito maior e resultados piores.

(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 deveriaser sempre melhor do que com tamanho 29? Porque não é este o resultado? (atenção: o arquivo mod30 não é o único resultado onde usar tamanho 30 é pior do que tamanho 29)

2.1) Tabelas com tamanho 30 não deveriam ser sempre melhores do que tabelas de tamanho 29. Isso ocorre porque um número primo como tamanho da tabela gera menos padrões de hashes para entradas não aleatórias, já que os hashes das entradas são melhor distribuídos na tabela. Como as entradas, mesmo aleatórias, não possuem probabilidade de hash perfeita, nem sempre um número maior de buckets gera um resultado melhor.

(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?

2.2) O número 30 é um número que possui muitos divisores, logo, a tabela hash pelo método da divisão acaba concentrando os números em determinados pontos da tabela, criando padrões para entradas que geram hashes múltiplos de fatores de 30. Já o número 29, por ser primo, não possui essa característica. Assim, os elementos podem ser melhor distribuídos na tabela. Para entradas aleatórias, essa característica não é tão evidenciada, porém, para entradas que seguem algum tipo de padrão, essa concentração de inputs em determinados pontos da tabela torna o resultado evidentemente pior. Em nosso exemplo a diferença não é muito grande justamente porque as entradas são aleatórias.

(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. (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)

2.3) O ataque funciona colocando muitos elementos em apenas um bucket, tornando a hashtable obsoleta. No caso do exemplo, mais de 70% de todos os elementos da entrada ficaram alocados no bucket 4. Assim, a hashtable precisa lidar com muitas colisões e o tempo de busca se torna linear. Para o ataque funcionar, o atacante deve saber qual a função hash, para selecionar entradas que geram a mesma chave e que, por consequência, ocupam o mesmo bucket.

(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?

3.1) Porque as entradas não possuem uma variedade tão grande de hashes quando comparadas ao tamanho da tabela. Quanto o range possível dos hashes das entradas não contempla um intervalo tão grande, não é vantajoso utilizar uma tabela tão maior. Por exemplo, se as palavras inseridas possuem hashes que variam em um intervalo de 100, apenas 100 dos 997 buckets serão utilizados.

(3.2) Porque a versão com produtório (prodint) é melhor?

3.2) Porque a versão com produtório acaba espalhando melhor as entradas dentro da tabela. É uma função de hash que gera hashes que contemplam intervalos maiores, assim, não concentra as entradas em determinadas posições da tabela.

(3.3) Porque este problema não apareceu quando usamos tamanho 29?

3.3) O problema não aparece na tabela de tamanho 29 pois o intervalo de hashes gerado pelas entradas é maior do que 29. Assim, a distribuição deles dentro da tabela se torna mais homogênea e não tão concentrada.

(4) hash por divisão é o mais comum, mas outra alternativa é hash de multiplicação (NÃO É O MESMO QUE prodint.m, verifiquem no Corben). É uma alternativa viável? porque hashing por divisão é mais comum?

4) O método de multiplicação é viável e possui a vantagem do valor do tamanho da hashtable não ser tão crítico assim. Porém, o valor constante A passa a possuir essa criticidade para o bom funcionamento da função. O método da divisão é mais comum pois é fácil de ser implementado, além de que muitos algoritmos envolvendo hashtables requerem uma tabela com um número primo de buckets, e também utilizam a divisão do hash por *m* em suas operações. Por isso, torna-se mais conveniente utilizar o método da divisão.

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

5) Normalmente, a utilização de Closed Hashing tem uma utilização de memória mais otimizada, já que percorrer a array é mais eficiente ao procurar um elemento. O custo dessa melhor utilização da memória de forma geral é a necessidade de uma função de hash que distribua bem os elementos, para não causar um overflow da parte de probing. Assim, em geral, a utilização da memória é mais otimizada e também a velocidade de procura na array é mais alta. Se torna melhor utilizar a Closed Hash ao ter um número menor de entradas, já que o custo de memória seria menor, pois a array necessária para armazenar os elementos seria menor.

(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?

6) Podemos aplicar universal hashing, que é um método que escolhe a função de hash aleatoriamente de uma forma independente das entradas que serão armazenadas. Nesse modo, escolhe-se um conjunto de funções que podem ser chamadas aleatoriamente, ao termos uma entrada a ser armazenada. Com isso, a hash gerada para cada input é feita pela função que é escolhida de forma aleatória. Logo, em execuções diferentes, até a mesma entrada pode gerar hashes diferentes, protegendo a hashtable de ataques. A aleatoriedade com que as funções são chamadas para cada entrada garante que não iremos cair no pior caso.