Relatório: Experimentos com funções de hash

Para entender os resultados dos experimentos e responder as perguntas, supõe-se pesquisar em bibliografia os conceitos e verificar o conteúdo de cada arquivo de dados (os nomes dos arquivos de dados são boas dicas).

(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?

R: Uma boa função de hashing é necessária para que haja uma distribuição mais uniforme das chaves nas entradas da tabela de *hash*. Uma função ruim, ou seja, que não faz uma boa distribuição, faz com que haja uma grande aglomeração em entradas específicas da tabela e as operações de inserção, busca e remoção demorem mais.

(1.2) porque ha uma diferenca significativa entre considerar apenas o 1o caractere ou a soma de todos?

R: Caso seja considerado somente o primeiro caractere, *strings* com o mesmo caracter inicial serão inseridos na mesma entrada na tabela *hash*. Desse modo, se a entrada de chaves for enviesada (por exemplo, apenas *strings* com letras iniciais sendo F ou G) haverá grande aglomeração de elementos na mesma entrada. Considerando a soma de todos caracteres, a entrada correspondente fica muito mais aleatória (um caso problemático nesse caso seria se as chaves de entrada fossem todas anagramas, mas isso já tem uma chance muito menor de acontecer).

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

R: Como explicado no item anterior, ao considerar apenas o primeiro caractere, as distribuições das palavras entre as entradas hash se torna muito enviesada, visto que há certas letras que são o caractere inicial de várias palavras (letra ‘e’, por exemplo), enquanto há letras que são o caractere inicial de pouquíssimas palavras (letra ‘y’, por exemplo).

(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)

R: Este não é o resultado pois 30 é um número com vários divisores, enquanto 29 é um valor primo.

(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?

R: Um número com vários divisores pode ser problemático pois é mais fácil atingir esse número (ou um múltiplo dele) com algarismos repetidos. Para o número 30, por exemplo, podemos ter o algarismo com valor 10 três vezes ou o algarismo com valor 6 dez vezes (nesse caso, atingir um múltiplo de 30 tem o mesmo valor de atingir 30), etc. No nosso exemplo, a diferença não é tão grande pois não há muitas *strings* com algarismos repetidos.

(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 em checkhashfunc)

R: Para esse arquivo, primeiro é feita a soma de todos os caracteres da string, então é tirado o *mod* dessa soma com o número 30, e então é somado 1 ao resultado.

(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 nao 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.

R: Com tamanho 997, de fato temos mais casas para espalhar as strings, mas também espera-se que haja um número muito menor de strings em cada entrada quando comparado com uma distribuição uniforme (afinal 1/997 é muito menor que 1/29). Desse modo, é difícil prever qual tamanho da *hash table* será mais eficaz. A versão com produtório é mais eficaz pois a multiplicação de valores inteiros produz números maiores e consequentemente mais dispersos (um exemplo: para chegar no número 35 com dois caracteres por soma há diversas combinações - 10+25, 23+12, 30+5, etc. - agora para chegar nesse valor com produto há bem

menos combinações - 35*1 e 7*5). Esse problema, porém, não aparecia quando o tamanho da tabela era 29, pois o fato do produtório gerar números grandes era anulado ao tirar o *mod* desse número com 29 (o *mod*, nesse caso, gera um número entre 0 e 28, enquanto para tamanho 997, o *mod* gera um valor entre 0 e 996).

(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?

R: No hash de multiplicação, a chave é multiplicada por um número irracional positivo menor que 1, é extraída a parte fracionária desse resultado, que é então multiplicado pelo tamanho da hash. A parte inteira desse resultado é a entrada hash correspondente à chave. Essa alternativa é viável mas não é tão comum pois requer a realização de mais cálculos computacionais (mais lento).

(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)

R: A vantagem de closed hash é que, pela limitação de elementos em cada bucket, as operações de busca tendem a ser O(1), além de que, pelo fato do tamanho de cada bucket ser fixo, a alocação de memória é mais rápida. Desse modo, é válido utilizar Closed Hash quando o número de chaves é igual (ou pouco maior) que o número de buckets.

(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 ***idéia básica em poucas linhas*** *(Dica: a estatística completa não é simples, mas a idéia básica é muito simples e se chama Universal Hash)*

R: Para impedir este tipo de ataque, uma solução é ter a disposição diversas funções de *hashing* e, para cada chave, atribuir uma das funções para o cálculo de *hash*. Desse modo, o atacante não saberá qual dos métodos de *hash* será utilizado para indexação das chaves.