Relatório CTC-12 Lab Hash

Aluno: Vinícius José de Menezes Pereira

1.

(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?

Uma boa função permitirá o acesso dos elementos em O(1). Uma função ruim levará à um acesso mais lento dos elementos, no pior caso em O(n). Além disso, levará à distribuição uniforme dos elementos.

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

Com a soma dos caracteres, é possível distribuir melhor os elementos ao longo do Hash, pois caso considerássemos apenas o primeiro caractere, as strings “a”,”abc” e “aifujcsduocs” estariam alocadas no mesmo bucket. Ao distribuirmos melhor os elementos no Hash, o acesso passa a ser mais rápido, O(1).

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

Com apenas um caracter, para datasets em que muitas palavras começam com o mesmo caracter, o desempenho tende a piorar muito, já que a distribuição tenderá a não ser 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? (atenção: o arquivo mod30 não é o único resultado onde usar tamanho 30 é pior do que tamanho 29)

Isso acontece pois, apesar de 30 ser maior, 29 é ligeiramente menor e ainda por cima é primo, o que melhora o desempenho do hash.

(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 número primo apresenta poucos divisores. Isso significa que ele terá maior desempenho quando utilizado, pois tende a aumentar a aleatoriedade da distribuição do hash. Considere, por exemplo, que se utiliza 30. Se colocarmos no hash strings com resultado da função que seja múltipla de um divisor de 30, por exemplo múltipla de 6, todos essas strings serão alocadas em espaços múltiplos de 6, gerando desbalanceamento do hash. Como o número primo tem poucos divisores, esse problema é significativamente amenizado.

(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)

Sabendo que o hash utiliza mod30, é possível que o atacante coloque na hash table strings cujo mod30 é o mesmo, como é o caso de acontecer com múltiplos de 2,3,5,6,10.. etc(os divisores de 30) periodicamente, fazendo com que o hash fique desbalanceado pelas múltiplas colisões, como já explicado no item anterior.

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?

Queremos uma distribuição uniforme. Um grande número de buckets ocupa mais espaço e para um pequeno conjunto de dados é ineficiente. A maioria dos buckets será inutilizada e as colisões não serão evitadas.

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

O prodint apresenta maior complexidade e consequentemente aleatoriedade em sua composição de ser. Este, portanto, consegue produzir uma distribuição mais uniforme ao longo da hash table, evitando colisões.

(3.3) 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 - 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.

29 é um número primo. Consequentemente, como já demonstrando, as funções de hash por divisão com números primos acabam por gerar uma boa distribuição de dados no hash.

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.

4.

(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?

O método da multiplicação em termos matemáticos é complicado de entender para um ser humano normal e apresenta um desempenho semelhante ao do método da divisão em termos práticos. Por ser mais simples e com bom desempenho, o método da divisão é geralmente escolhido.

5.

(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)

Dado um domínio bem conhecido, onde as colisões são raras, o Closed Hash pode ser mais vantajoso por apresentar espaço de alocação constante e controlado. Na OpenHash, o espaço é alocado pela lista ligada conforme a necessidade, o que pode custar muito na memória, apesar de ser mais flexível. Para um dataset menos controlado, o OpenHash tende a apresentar maior desempenho.

6.

(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)

Para evitar esse tipo de ataque, é necessário aplicar o universal hashing, que por uma série de funções de hash consegue adicionar aleatoriedade à função hash, fazendo com que o ataque seja neutralizado.