3º Trabalho Prático

Tiago Alves Macambira < tmacam@dcc.ufmg.br >

16 de Julho de 2003

1 Os algoritmos

A seguir, explicaremos sucintamente o funcionamento dos 3 algoritmos implementados.

1.1 Boyer-Moore-Horspool

Como o Cormen et al[1] dizem, o algoritmo de Boyer-Moore é muito semelhante ao de força bruta em sua estrutura. Contudo, uma diferença inicial marcante é o fato de que o algoritmo de BM verifica casamentos com a cadeia de trás para frente no padrão, ao contrário do de força bruta, que verifica casamentos entre o texto e a cadeia de frente para trás. Todavia, isso somente não é o responsável pela difenrença de complexidade entre os dois algoritmos. BH faz uso de heurísticas que permitem que muito do trabalho do algoritmo de força bruta seja evitado.

O algoritmo de Boyer-Moore-Horspoll, a mais importante simplificação de Boyer-More, consegue melhorar o desempelnho do algoritmo original evitando o tempo de escolha entre que heurística tomar. Seu grande trunfo está na função de avanço λ, definida como:

$$\lambda[x] = \min\{j | (j = m) \wedge ((1 \leq j < m) \vee (P[m - j] = x))\}$$

Sempre que no laço *while*interno não for encontrado um casamento, o algoritmo de BMH verifica qual é, naquele momento, o i-ésimemo caractere no texto, correspondente ao último caractere do padrão. Dado esse caractere, verifica-se qual será sua imagem na função λ. Essa informação diz quanto o padrão deveria andar para casar esse i-ésimo caractere no texto com um póssível caractere igual a este no padrão. Se este i-ésimo caractere nem sequer estiver presente no padrão, o algotitmo andará $m$caracteres.Vê-se facilmente então porquê o melhor caso desse algoritmo é $O(n/m)$. Esse também é o caso médio desse algoritmo, o que explica porque ele é tão atraente.

1.2 Shift-And

A idéia é simples: com um autômata finito determinítico (DFA) consegue-se em tempo $\Theta(n)$[1] verificar se existe ou não um casamento para o padrão Pno texto

Algorithm 1 BuscaForcaBruta(T,P)

```
n ← tamanho(T)
m ← tamanho(P)
for x ← 0..n − m do
  if P[1..m] = T[s + 1..s + m] then
    imprima "Casamento em",s
  end if
end for
```

Algorithm 2 AlgoritmoBMH

```
n ← tamanho(T)
m ← tamanho(P)
λ ←Função de avanço
i ← m
while i ≤ n do
  k ← i
  j ← m
  while ((j ≥ 0)^(T[k]) = P[j]) do
    k ← k − 1
    j ← k − 1
  end while
  if j < 0 then
    imprima "Casamento em",k + 1
  end if
  i = i + λ[T[i]]
end while
```

Algorithm 3 Shift-And Exato

$\lambda \leftarrow$Função de máscara de bits
$R \leftarrow 0^m$
for $i \leftarrow 0..n-m$ **do**
  $R \leftarrow ((R \ll 1) \wedge 10^{m-1}) \vee \lambda[T[i]]$
  if $R \vee 0^{m-1}1 \neq 0^m$ **then**
    imprima "Casamento em",$i-m+1$
  end if
end for

T. A construção de tal autômato, no entanto, pode ser bastante dispendiosa se o alfabeto Σ usado por T e P for grande.

Knuth, Morris e Pratt foram os primeiros a vir com um algoritmo eficiente para o problema de *string matching* e o fizeram utilizando idéias baseadas em autômatos. Nesse caso, em um 2DPDA (2-way Deterministic Pushdown Automata). Esse algoritmo, no entanto não é tão eficiente quanto o BMH e o que interessa aqui é falar do Shift-And. Cortemos o leriado e vamos falar logo dele.

O algoritmo Shift-And simula de forma bem interessante um NDFA (Non-Deterministic DFA) limitado para fazer o casamento de padrões em texto. Seu grande atrativo é que com pouquíssimas alterações é possível, com ele, fazer buscas aproximadas.

Ele utiliza o paralelismo de bits das palavras de um computador, de forma muito engenhosa. tanto para representar as transições possíveis entre estados como para representar o próprio estado do autômata (estados ativos). De maneira análoga ao BMH, existe um pré-processamento que deve ser feito com o padrão de busca P, a um custo $O(m)$. Este pré-processamento consiste em criar uma tabéla contendo a *máscara de bits* de cada caractere no padrão. Para um caractere i no padrão, a sua máscara $\lambda(i)$ é definida como:

$$\lambda(i) = b_m^i b_{m-1}^i ... b_1^i,$$

onde

$$b_p^j = \begin{cases} 1, & se \quad P[p] = i \\ 0 & se \quad P[p] \neq i \end{cases}$$

Uma fez feito isso, e tendo que $m \ll n$, o custo do algoritmo se resume a alimentar o *autômato com o texto*,$O(n)$. A notação usada no algoritmo segue a definida pelo professor: exponenciação significa repetição de um dado numero, por exemplo: $0^3 1^2 0^3 = 00011000$.

1.3 Shift-And Aproximado

Como dito na sub-seção anterior, o algoritmo shift-and pode ser modificado para incluir busca aproximada. Tal é feito acrescentando mais registradores (R_i, no algoritmo). O material fornecido pelo professor[2] explica essa parte em grande detalhe, de forma que julgo desnecessáio repetí-la aqui.

2 Análise de Desempenho

Os dados seguintes foram gerados através da script GERA_ESTATISTICAS.SH. Seu código foi gentilmente cedido por Erikson Freitas de Morais, e modificado por mim para assumir a forma atual. Os dados foram pré-processados pela script PARSE_ESTATISTICAS.PL de forma a poder gerar um conjunto de dados mais tratável.

Todos os testes foram executados na máquina TURMALINA. A medição do tempo foi feita usando-se o programa TIME e levando-se em conta apenas o tempo de usuário de cada processo lançado, de forma que os dados estão isentos de flutuações devidas a lentidões ocasiodados por problemas de I/O e etc.

2.1 Casamento Exato

Segue abaixo uma tabela contendo o tempo total que cada algoritmo levou para realizar a busca pelas palavras chaves em cada um dos arquivos de testes fornecidos.

tamanho do arquivo em kB	agrep exato	BMH	Shift-And Exato
2768	0,921	10,121	20,841
9912	2,781	34,721	73,291
19832	5,231	69,711	147,231
106952	26,241	375,551	794,071

Um primeiro dado que chama a atenção é quão lento é a minha implementação, mesmo do BMH, é do agrep. Isso deve ser causado pelo fato de que minha implementação lê e processa uma linha por vez, colocando um certo *overhead* a mais, já que a cada chamada da função um certo processo de inicialização é feita. Tem-se também o fato de que limita-se a procura a, em média, 80 caracteres por vez, ao invés de blocos de tamanho maiores.

Analisando apenas a minha implementação, já que não tem como comparar os dados do agrep com ela, vemos que o BMH mesmo estando limitado às mesmas condições do Shift-And, apresenta uma performance pelo menos 2 vezes melhor, aproximadamente.

Número de Caracteres	Padrão	Tempo
3	DCC	23,031
3	dia	23,991
3	New	23,561
4	UFMG	19,581
4	York	20,541
5	index	18,691
5	price	19,101
5	stock	18,451
6	branco	17,401
6	Canada	17,601
6	coffee	17,381
6	dollar	17,311
7	Gregory	16,341
7	Michael	16,741
7	Uberaba	16,481
8	Exchange	15,511
8	Treasury	15,461
9	Brazilian	15,071
9	Macunaima	15,041
10	Manacupuru	14,321
14	administration	13,941

Outro detalhe interessante é ver como o tamanho e a estrutura da chave de busca alteram os tempos de execução do BMH. Quanto maior a palavra, mais rápido fica o algoritmo. Isso é devido a possibilidade de dar saltos maiores no texto quanto maior for a palavra.

O Shift-And, ao contrário, nao apresenta comportamento semelhante (Shift-And exato, arquivo de 106M):

Número de Caracteres	Padrão de busca	Tempo
3	DCC	37,921
4	UFMG	38,601
14	administration	37,461

2.2 Casamento Aproximado

Vejamos agora uma tabela contendo o tempo médio que cada algoritmo utilizou para realizar, para cada valore de k possível, uma busca aproximada. Note que descartamos absurdos como busca aproximada com 3 erros para palavra de 3 caracteres (mutatis mutandis) porque isso casa com qualquer coisa e não é o nosso interesse ver esse caso. O caso médio será mostrado na tabela a seguir ao invés do somatório devido ao fato de que nem todas as instâncias de testes

possíveis para esse caso foram rodadas, devido ao alto tempo que esses testes levavam.

K (número de erros)	Agre Aprox.	Shift-And aprox
1	1,871	14,194
2	5,107	24,078
3	10,187	28,989
4	12,917	31,453
5	13,221	33,430
6	12,545	31,894

Algumas observações antes de comentarmos esses dados:

- O agrep não faz busca aproximada com mais de 8 caracteres
- Valores para os testes com $k \geq 7$não foram mostrados devido ao pequeno número de palavras com mais de 8 caracteres, para os quais faria sentido realizar procura com $k \geq 7$
- Como dito, não rodamos todas as possíveis instâncias de testes, em especial as que tomavam muito tempo.

Dessa forma, reduzimos a tabela para uma faixa mais conservadora dos dados.

O que podemos novamente observar é que minha implementação realmente ficou muito ruim. Tirando isso, podemos observer que a medida que $k$cresce, a diferença entre os tempos de execução para dois valores de $k$consecutivos decresce, claro sinal que quanto maior o k,maior é a probabilidade de casar o podrão mais facilmente, chegando a um ponto que a diferença volta a crescer denovo (valor absoluto): k fica tão grande que a facilidade de casar com qualquer coisa aumenta. Isso pode ser melhor observado olhandose para as linhas em que $4 \leq linha \leq 6$.

3 Versão eletrônica

A versão eletrônica desse relatório encontra-se em *http://www.dcc.ufmg.br/~tmacam/PAA/*, na pasta PAA3, bem com os relatórios dos trabalhos passados.

4 Referências

References

[1] "Introduction to Altorithms", Cormen et al, 1st Edition

[2] "Projeto de Algoritmos com implementações em Pascal e C". Ziviani, Nívio. 5a./6a. edição