Suffix Array

Universidade Federal de Minas Gerais - Instituto de Ciências Exatas
Curso de Pós-graduação em Ciência da Computação

Disciplina: Projeto e Análise de Algoritmos - Professor: Nívio Ziviani
Alunos: Braulio Alves Silva Lara & Guilherme Tavares de Assis

Resumo

Neste trabalho, é apresentada uma simples estrutura de dados, chamada *Suffix Array*, para buscas *on-line* de padrões de caracteres em textos. Sua lógica é muito parecida com a lógica da estrutura de dados *Pat Array*, diferenciando em apresentar estruturas adicionais que aumentam a sua eficiência com relação ao tempo de busca. Além disso, *Suffix Array* ocupa bem menos espaço que outras estruturas, como *Suffix Tree*, sendo bem competitivo com relação a construção e velocidade de busca.

1 Introdução

Segundo [5], o problema de **casamento de cadeias** ou **casamento de padrão** consiste, informalmente, no problema de se buscar todas as ocorrências de um padrão em um determinado texto. Tal problema pode ser formalizado da seguinte forma. O texto é um arranjo $T[1..n]$ de tamanho n $(t_0t_1t_2...t_{n-1})$ e o padrão é um arranjo $P[1..m]$ de tamanho $m \leq n$ $(p_0p_1p_2...p_{m-1})$. Os elementos de T e P são escolhidos de um alfabeto finito Σ de tamanho c. Fornecidas as cadeias P de comprimento $|P| = m$ e T de comprimento $|T| = n$, em que $n >> m$, o problema de casamento de padrão consiste em descobrir as ocorrências de P em T.

Existem muitas aplicações onde distintos padrões são pesquisados várias vezes em um texto. Assim, se faz necessário criar algoritmos rápidos para casamento de padrão e/ou construir estruturas de dados que promovam a realização de buscas rápidas. Baseado nesta última idéia, foi definida uma nova estrutura de dados denominada *Suffix Array*, que consiste basicamente em uma lista ordenada de todos os sufixos de um texto T, onde um sufixo de T que começa na posição i do texto é denotado por $T_i = t_it_{i+1}t_{i+2}...t_{n-1}$.

De acordo com [4], para se implementar o algoritmo de busca que se baseia no *Suffix Array*, são necessários três arranjos de valores inteiros: *Pos*, *Llcp* e *Rlcp*. *Pos* é um arranjo ordenado lexicograficamente dos sufixos de T, onde $Pos[k]$ corresponde a posição inicial do k-ésimo menor sufixo do conjunto de sufixos (T_0, T_1, ..., T_{n-1}); assim, tem-se que $A_{Pos[0]} < A_{Pos[1]} < ... < A_{Pos[n-1]}$. *Llcp* e *Rlcp* são arranjos responsáveis pelo armazenamento dos prefixos comuns mais longos ($lcp's$ - *longest common prefixes*) entre os sufixos de T; tais arranjos são explicados, detalhadamente, na seção 3 deste trabalho.

O restante deste trabalho está organizado como se segue. Na seção 2, é apresentado o processo de ordenação do arranjo *Pos*. Na seção 3, a construção dos arranjos *Llcp* e *Rlcp* é descrita. Na seção 4, é apresentado o algoritmo de busca que se baseia no *Suffix Array*, assim como é discutida a complexidade teórica de execução do mesmo. Na seção 5, são descritos os resultados experimentais obtidos a partir da implementação prática do algoritmo de busca do *Suffix Array* e de outros algoritmos de casamento de padrão. Na seção 6, uma conclusão sobre o assunto tratado neste trabalho é apresentada.

2 Ordenação do Arranjo Pos

Segundo [4], a ordenação do arranjo Pos é feita em $\lceil \log_2(n+1) \rceil$ estágios. Inicialmente, todos os sufixos encontram-se no mesmo bloco. É importante mencionar que apenas os apontadores dos sufixos são guardados e não os sufixos propriamente ditos. No primeiro estágio, os sufixos são separados em blocos de acordo com o primeiro caractere dos mesmos. Então, indutivamente, a cada estágio, novas partições são feitas nos blocos anteriores de acordo com o dobro de caracteres considerados na ordenação até o momento, gerando assim, novos pequenos blocos. Logo, no h-ésimo estágio, os sufixos de um mesmo bloco estão ordenados segundo a ordem ≤h[1]. Para uma maior simplicidade, são adicionados caracteres *branco* no final dos sufixos, para que todos tenham comprimento $n+1$.

Sejam T_i e T_j dois sufixos pertencentes ao mesmo bloco depois do h-ésimo estágio; assim, tem-se que T_i =h T_j. A partir daí, é preciso comparar T_i com T_j segundo os próximos h caracteres. Mas, os próximos h caracteres e T_i e T_j são exatamente os primeiros h caracteres de T_{i+h} e T_{j+h} respectivamente. Por suposição, já é conhecida a ordem relativa , segundo a relação ≤h, de T_{i+h} e T_{j+h}. Portanto, pode-se utilizar o conhecimento já adquirido, para completar o novo estágio eficientemente.

A idéia central do processo de ordenação do arranjo Pos é a seguinte. Começa-se com o primeiro bloco que contém os menores sufixos segundo a relação ≤h. Seja T_i o primeiro sufixo do primeiro bloco ($Pos[0] = i$) e considere T_{i-h} (se $i - h < 0$, então T_i é ignorado e o próximo sufixo (posição $Pos[1]$) é tomado, e assim sucessivamente). Já que T_i começa com a menor subcadeia h de caracteres, T_{i-h} deve ser a primeira em seus blocos $2h$. Logo, T_{i-h} é movido para o início do seu bloco e tal movimentação é marcada. Para todo bloco, é preciso conhecer o número de sufixos que já foram movidos para, assim, colocá-lo em ordem ≤2h. O algoritmo basicamente explora os sufixos como eles aparecem na ordem ≤h e, para cada T_i, move T_{i-h} (caso exista) para o próximo local avaliado em seu bloco h. O algoritmo completo, envolvendo as estruturas de dados temporárias e necessárias para a realização da ordenação, pode ser encontrado em [4]. Em um mesmo estágio, todos os passos da ordenação podem ser feitos em tempo $O(n)$, e já que existem $\lceil \log_2(n+1) \rceil$ estágios no máximo, a ordenação de Pos apresenta complexidade de tempo $O(n \log n)$.

3 Construção dos Arranjos $Llcp$ e $Rlcp$

Considere T_i e T_j dois sufixos de T. Além disso, considere que o arranjo Pos encontra-se ordenado. Seja $lcp(T_i, T_j)$ o comprimento do prefixo comum mais longo entre T_i e T_j. Assim, definindo L, R e M como sendo as posições mais a esquerda, mais a direita e a do meio de um determinado intervalo do arranjo Pos, pode-se estabelecer os seguintes arranjos de inteiros de tamanho $n-2$: $Llcp[M] = lcp(T_{pos[L]}, T_{pos[M]})$ e $Rlcp[M] = lcp(T_{pos[M]}, T_{pos[R]})$.

De acordo com [4], um eficiente algoritmo de busca baseado no *Suffix Array*, apresentado na seção 4, requer informação pré-computada sobre os $lcp's$ entre os sufixos que se encontram no meio, e aqueles que se encontram mais a esquerda e a direita de um intervalo do arranjo Pos. A idéia chave para se calcular os $lcp's$ entre os sufixos é a seguinte. Depois do primeiro estágio, os $lcp's$ entre sufixos em blocos adjacentes valem zero. Assumindo que, depois do estágio h, os

[1]Seja u^k o prefixo constituído dos primeiros k caracteres da cadeia u, se u contém mais de k caracteres, ou o próprio u, em caso contrário. A relação <k corresponde à ordem lexicográfica dos prefixos de k caracteres; assim, (u <k v) se e somente se ($u^k < v^k$). As relações ≤k, =k, >k e ≥k são definidas da mesma forma.

$lcp's$ entre sufixos em blocos adjacentes são conhecidos, no estágio $2h$, os $lcp's$ devem ser no mínimo h e no máximo $2h-1$, já que no estágio $2h$ os blocos são particionados de acordo com $2h$ caracteres. Assim, se os sufixos T_i e T_j estão no mesmo bloco h, mas estão em blocos $2h$ distintos, então $lcp(T_i, T_j) = h + lcp(T_{i+h}, T_{j+h})$. Portanto, tem-se que $lcp(T_{i+h}, T_{j+h}) < h$. O problema é que se tem apenas os $lcp's$ entre sufixos em blocos adjacentes, e T_{i+h} e T_{j+h} podem não estar em blocos adjacentes. Contudo, se $T_{Pos[i]}$ e $T_{Pos[j]}$, onde $i < j$, tem um lcp menor que h e Pos está em ordem $\leq h$, então o lcp entre tais sufixos é o mínimo dos $lcp's$ de todos os pares adjacentes de sufixos entre $Pos[i]$ e $Pos[j]$. Ou seja,

$$lcp(T_{Pos[i]}, T_{Pos[j]}) = min_{k \in [i,j-1]}(lcp(T_{Pos[k]}, T_{Pos[k+1]}))$$

Se fosse possível manter, depois do estágio h, informação sobre todos os $lcp's$ cujos valores são menores que h, a computação de $lcp's$ no estágio $2h$ seria direta da fórmula apresentada. Mas isto requer muito tempo de execução e espaço de memória. Usando uma árvore balanceada de altura de espaço $O(n)$ que registra o lcp mínimo sobre a coleção de intervalos do *Suffix Array*, pode-se determinar o lcp entre dois sufixos quaisquer em tempo $O(logn)$. Tal estrutura de dados, denominada árvore de intervalos, está descrita depois de umas proposições básicas.

Considere a seguinte definição: $Hgt[i] = lcp(T_{Pos[i-1]}, T_{Pos[i]})$, onde $1 \leq i \leq n-1$. $Hgt[i]$ é um arranjo de $n-1$ valores inteiros, que armazena os $lcp's$ entre sufixos adjacentes no arranjo Pos, e atinge seu valor correto no estágio h se e somente se $Hgt[i] < h$; em caso contrário, possui valor indefinido (especificamente $n+1$). Note que, se $Hgt[i] < h$, então $T_{Pos[i-1]}$ e $T_{Pos[i]}$ devem estar em blocos h distintos, já que blocos h contem sufixos com o mesmo prefixo de h caracteres.

[3] apresenta um algoritmo que calcula o arranjo Hgt em complexidade de tempo $O(n)$. Segundo tal algoritmo, não é necessário comparar todos os caracteres quando se computa o lcp entre um sufixo e seu sufixo adjacente no vetor Pos; para calcular todos os $lcp's$ entre sufixos adjacentes no Pos eficientemente, os sufixos de A_0 a A_{n-1} são examinados em ordem.

Os arranjos *Llcp* e *Rlcp* podem ser calculados, utilizando a árvore de intervalos. Tal árvore consiste de $2n-3$ nodos, cada um nomeado com um dos $2n-3$ pares (L, R) que podem ocorrer. A raiz da árvore é nomeada $(0, n-1)$ e os nodos seguintes são (L, M) e (M, R) para algum elemento do meio $M \in [1, n-2]$. Alternativamente, os $n-2$ nodos internos da árvore são (L_M, R_M) para cada elemento do meio M, e suas $n-1$ folhas são $(i-1, i)$ para $i \in [1, n-1]$ na ordem esquerda para a direita. Para cada nodo interno, $esquerda(L_M, R_M) = (L_M, M)$ e $direita(L_M, R_M) = (M, R_M)$. Logo, tem-se:

$$Height[(L, R)] = min(Hgt[k] : k \in [L+1, R]) = lcp(T_{Pos[L]}, T_{Pos[R]})$$

Portanto, $Llcp[M] = Height[(L_M, M)]$ e $Rlcp[M] = Height[(M, R_M)]$. Com esta árvore, os arranjos *Llcp* e *Rlcp* são diretamente avaliados sobre a conclusão da ordenação.

4 Algoritmo de Busca baseado no *Suffix Array*

O objetivo do algoritmo de busca é encontrar todas as ocorrências do padrão $P = p_0p_1p_2...p_{m-1}$ no texto $T = t_0t_1t_2...t_{n-1}$. Para tanto, sejam $L_P = min(k : P \leq m \ A_{Pos[k]}$ ou $k = n)$ e $R_P = max(k : A_{Pos[k]} \leq m \ P$ ou $k = -1)$. Já que Pos está em ordem $\leq$m, segue que P casa com $t_it_{i+1}...t_{i+m-1}$ se e somente se $i = Pos[k]$, para algum $k \in [L_P, R_P]$. Logo, se L_P e R_P podem ser encontrados rapidamente, então o número de casamentos é $(R_P - L_P + 1)$ e seus apontadores à esquerda são dados por $Pos[L_P]$, $Pos[L_{P+1}]$, ..., $Pos[R_P]$. Como Pos está em ordem ≤m, então uma simples pesquisa binária pode encontrar L_P e R_P usando $O(\log n)$ comparações de cadeias de tamanho no máximo m; assim, cada comparação requer $O(m)$ comparações de caracteres. Portanto, o arranjo Pos permite encontrar todas as instâncias de um padrão P no texto T no tempo $O(m \log n)$. O algoritmo de busca, para encontrar L_P, é o seguinte:

```
if (P ≤m A_Pos[0]) then
      L_P ← 0
else if (P >m A_Pos[n−1]) then
      L_P ← n
else {
      (L,R) ← (0,n-1)
      while (R-L > 1) do {
            M ← (L+R)/2
            if (P ≤m A_Pos[M]) then
                  R ← M
            else
                  L ← M
      }
      L_P ← R
}
```

O algoritmo apresentado é muito simples. Seu tempo de execução pode ser melhorado para $O(m + \log n)$, ao se utilizar dos arranjos *Llcp* e *Rlcp* descritos na seção 3. Para tanto, deve-se modificar o algoritmo, utilizando-se duas variáveis l e r, tal que $l = lcp(A_{Pos[L]}, P)$ e $r = lcp(P, A_{Pos[R]})$. Inicialmente, l recebe o tamanho do prefixo comum mais longo entre $A_{Pos[0]}$ e P, e r recebe o tamanho do prefixo comum mais longo entre $A_{Pos[n-1]}$ e P. Depois, a cada comparação de P com $A_{Pos[M]}$, l e r são adequadamente alterados. A partir de l e r, pode-se definir uma nova variável $h = min(l, r)$, como sendo o número mínimo de comparações ao se comparar $A[pos[M]]$ com P, pois ($A_{Pos[L]}$ =l P =r $A_{Pos[R]}$) implica ($A_{Pos[k]}$ =h P), para todo $k \in [L, R]$ incluindo M. Assim, o número de comparações é reduzido.

Considere o conjunto de todas as triplas (L, M, R) que podem surgir no processo de repetição interno da pesquisa binária. Existem exatamente $n-2$ triplas, cada uma com um único elemento do meio $M \in [1, n-2]$ e $0 \leq L < M < R \leq n-1$. Considere, agora, uma iteração do processo de pesquisa para a tripla específica (L, M, R) e, sem perda de generalidade, assume-se que $l \geq r$. Seja $h = min(l, r)$ e seja Δh a diferença entre o valor de h no começo e no final da iteração. Existem três casos a considerar, baseados se o $Llcp[M]$ é maior, igual ou menor que l, para se tomar as decisões corretas:

Caso 1 : $Llcp[M] > l \Rightarrow$ neste caso, $A_{Pos[M]}$ =(l+1) $A_{Pos[L]}$ ≠(l+1) P; portanto, P está no lado direito e l é alterado.

Caso 2 : $Llcp[M] = l \Rightarrow$ neste caso, os primeiros l caracteres de $Pos[M]$ e P são iguais; logo, deve-se comparar os caracteres a partir da $l + 1$-ésima posição até se encontrar um $l + j$-ésimo caractere tal que P ≠(l+j) $Pos[M]$. O caractere $l + j$-ésimo determina se L_P está no lado direito ou esquerdo. Em cada caso, l ou r deve ser alterado para $l + j$. Já que $l = h$ no começo do processo de repetição, este passo leva Δh+1 comparações de caracteres.

Caso 3 : $Llcp[M] < l \Rightarrow$ neste caso, já que P casa com l caracteres de L e menos que l caracteres de M, L_P está no lado esquerdo e r é alterado para $Llcp[M]$.

Portanto, o uso dos arranjos *Llcp* e *Rlcp* (o arranjo *Rlcp* é usado quando $l < r$) reduz o número de comparações para não mais que Δh+1, para cada iteração. Considerando todas as iterações, observa-se que $\Sigma\Delta$h $\leq m$; logo, o número total de comparações para se buscar um padrão é, no máximo, $m + \lceil \log_2(n-1) \rceil$ e, assim, o algoritmo possui complexidade de tempo $O(m + \log n)$ no pior caso. O algoritmo de busca melhorado, para encontrar L_P, é o seguinte:

```
l ← lcp(A_Pos[0], P)
r ← lcp(A_Pos[n−1], P)
if (l=m) or (P_l ≤m A_Pos[0]+l) then
        L_P ← 0
else if (P_r >m A_Pos[n−1]+r) then
        L_P ← n
else {
        (L,R) ← (0,n-1)
        while (R-L > 1) do {
              M ← (L+R)/2
              if (l ≥ r) then
                   if (Llcp [M] ≥ l) then
                            meio ← l + lcp(A_Pos[M]+l, P_l)
                   else
                            meio ← Llcp[M]
              else
                   if (Rlcp[M] ≥ r) then
                            meio ← r + lcp(A_Pos[M]+r, P_r)
                   else
                            meio ← Rlcp[M]
              if (meio=m) or (P_meio ≤m A_Pos[M]+meio) then
                            (R,r) ← (M,meio)
              else
                            (L,l) ← (M,meio)
        }
        L_P ← R
}
```

5 Experimentos Práticos

Esta seção apresenta os resultados de dois experimentos práticos envolvendo o algoritmo de busca do *Suffix Array*.

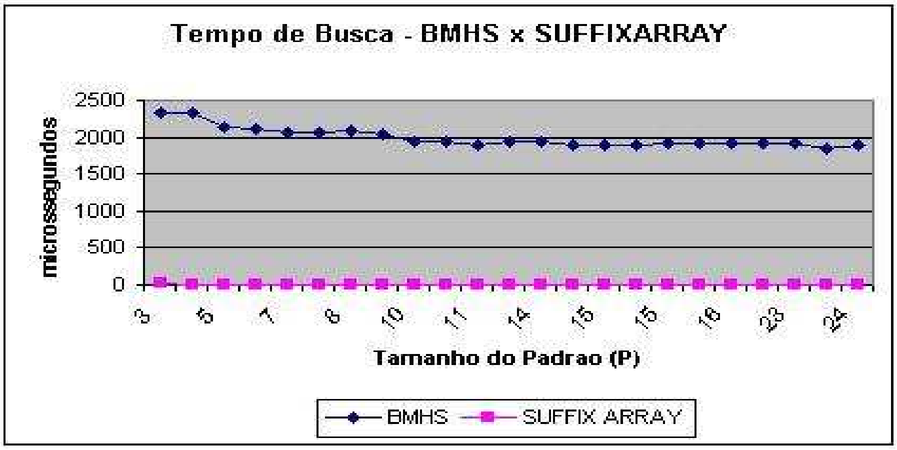
O primeiro experimento prático pode ser encontrado em [4], e consiste em uma comparação feita entre os algoritmos de busca que utilizam as estruturas de dados *Suffix Array* e *Suffix Tree*. Ambos algoritmos pertencem à mesma categoria para resolver o problema de casamento de cadeias: pré-processamento do texto. A tabela 1 apresenta um resumo conclusivo dos resultados obtidos.

Tabela 1: Comparação entre o *Suffix Array* e o *Suffix Tree*

	Suffix Array	*Suffix Tree*
Tempo de Construção	$O(n \log n)$ - para pequenos alfabetos $O(n)$ - necessita espaço adicional	$O(n)$
Complexidade de Tempo	$O(m + \log n)$ - bom para grandes alfabetos	$O(m \log \Sigma)$
Complexidade de Espaço	requer 2n inteiros (principal vantagem)	$O(n)$
Dependente do alfabeto	Não	Sim

O segundo experimento prático envolve a implementação dos algoritmos de busca *Suffix Array* e BMHS (*Boyer-Moore-Horspool-Sunday*), para fins de comparação de eficiência de busca. Tais algoritmos encontram-se em categorias distintas para resolver o problema de casamento de cadeias: no primeiro, como já mencionado, o texto é pré-processado e, no segundo, apenas o padrão é pré-processado. Os experimentos envolveram um texto com 100.000 caracteres e vários padrões de tamanho máximo igual a 30 caracteres. A Figura 1 apresenta os resultados obtidos, quanto ao tempo de busca, para cada padrão pesquisado no texto considerado.

Figura 1: Comparação entre o *Suffix Array* e o BMHS



Analisando a Figura 1, pode-se perceber que o tempo de busca do *Suffix Array* foi muito menor que o tempo de busca do BMHS. Esta grande diferença nos tempos de busca já era esperada, pois as complexidades teóricas de tempo referentes aos algoritmos *Suffix Array* e BMHS são, respectivamente, $O(\frac{n}{m})$ e $O(m + \log n)$.

6 Conclusão

O *Suffix Array* é uma estrutura de dados eficiente que permite buscas *on-line* rápidas. Seu uso é interessante em aplicações onde existe a necessidade da busca de diversos padrões em uma coleção grande e semi-estática. Dessa forma, é interessante a criação dessa estrutura, mesmo tendo um custo inicial significativo, pois viabiliza tempos de busca muito menores comparados com algoritmos que não fazem o pré-processamento do texto. No entanto, essa desvantagem inicial é compensada com o ganho de tempo após a execução de várias buscas de padrões.

A grande vantagem do *Suffix Array* é que ele ocupa bem menos espaço que outras estruturas, que também fazem o pré-processamento do texto, como por exemplo *Suffix Tree*, além de ser bem competitivo com relação ao custo de construção e à velocidade de busca.

Referências

[1] Apostolico, A. and Iliopoulos, C. *Parallel log-time construction of suffix trees*, CSD TR 632, Dept. of Computer Science, Purdue University, 1986.

[2] Gonnet, G.H. and Baeza-yates, R.A. and Snider, T. *New Indices for Text: Pat trees and Pat Arrays*, Information Retrieval: Data Structures and Algorithms, Prentice Hall, 1992.

[3] Kasai, Toru and Lee, Gunho et al. *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*, CPM 2001, LNCS 2089, pp.181-192, Springer-Verlag Berlin Heidelberg, 2001.

[4] Manber, Udi and Myers, Gene. *Suffix Arrays: A new method for on-line string searches*, First Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, January 1990, pp. 319-327.

[5] Ziviani, Nivio. *Projeto de Algoritmos: com implementações em PASCAL e C*, Pioneira Thomson Learning, São Paulo, 1994.