Um algoritmo para a construção de vetores de sufixo generalizados em memória externa

Aluno: Felipe Alves da Louza
Orientadora: Profa. Dra. Cristina Dutra de Aguiar Ciferri

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo, SP, Brasil

Defesa de Mestrado

17 de dezembro de 2013

Sumário

Introdução

Vetor de sufixo [Manber and Myers, 1990, Gonnet et al., 1992]:

- Estrutura de dados utilizada em problemas que envolvem cadeias de caracteres
- Muitos trabalhos têm sido propostos para construção de vetores de sufixo em memória externa *e.g.* [Dementiev et al., 2008, Bingmann et al., 2013]

Indexação de conjuntos de cadeias

- Para utilizar esses algoritmos é necessário concatenar todas as cadeias $T_i \in \mathcal{T}$ em $T = T_1\$_1 \dots T_k\$_k$, utilizando diferentes símbolos terminais $\$_i$
- Vetor de sufixo generalizado para conjuntos de cadeias

Limitação

- Esses algoritmos são direcionados para indexação de apenas uma cadeia

Introdução

Vetor de sufixo [Manber and Myers, 1990, Gonnet et al., 1992]:

- Estrutura de dados utilizada em problemas que envolvem cadeias de caracteres
- Muitos trabalhos têm sido propostos para construção de vetores de sufixo em memória externa *e.g.* [Dementiev et al., 2008, Bingmann et al., 2013]

Indexação de conjuntos de cadeias

- Para utilizar esses algoritmos é necessário concatenar todas as cadeias $T_i \in \mathcal{T}$ em $T = T_1\$_1 \ldots T_k\$_k$, utilizando diferentes símbolos terminais $\$_i$
- Vetor de sufixo generalizado para conjuntos de cadeias

Observações

- Essa abordagem limita o número de possíveis cadeias em $\mathcal{T}$
- Por exemplo, utilizando 1 byte para cada caractere, k é limitado por $256 - |\Sigma|$

Introdução

Vetor de sufixo [Manber and Myers, 1990, Gonnet et al., 1992]:

- Estrutura de dados utilizada em problemas que envolvem cadeias de caracteres
- Muitos trabalhos têm sido propostos para construção de vetores de sufixo em memória externa *e.g.* [Dementiev et al., 2008, Bingmann et al., 2013]

Indexação de conjuntos de cadeias

- Para utilizar esses algoritmos é necessário concatenar todas as cadeias $T_i \in \mathcal{T}$ em $T = T_1\$_1 \ldots T_k\$_k$, utilizando diferentes símbolos terminais $\$_i$
- Vetor de sufixo generalizado para conjuntos de cadeias

Contribuição

- Primeiro algoritmo para a construção de vetores de sufixo generalizados em memória externa
- Algoritmo eGSA [Louza et al., 2013]

Sumário

Fundamentos

Seja $T = T[1]T[2]\dots T[n-1]\,\$$ uma cadeia de tamanho n, $T[i] \in \Sigma$ e $\$ \notin \Sigma$

- $T[i,j] = T[i]\dots T[j]$, $1 \leq i \leq j \leq n$ é uma sub-cadeia de T
- $T[i,n]$ é um sufixo de T

Vetor de sufixo (SA):

- Vetor de inteiros $i \in [1,n]$
- Sufixos $T[i,n]$ ordenados lexicograficamente ($\$ < A < C < G < T$)

i	$SA[i]$	$suff(i)$
1	7	$
2	6	A$
3	4	AGA$
4	2	ATAGA$
5	5	GA$
6	1	GATAGA$
7	3	TAGA$

Figura : SA para $T = GATAGA\$$

Coluna $suff(i)$

$suff(i) = T[SA[i],n]$.

Busca por padrões

Busca binária $O(m \log n)$

Fundamentos

Estruturas auxiliares:

- Vetor de prefixo comum mais longo (LCP)
- Transformada de *Burrows-Wheeler* (BWT)
- Vetor de sufixo aumentado $ESA[i] = \langle SA[i], LCP[i], BWT[i] \rangle$

i	$ESA_1[i]$			$suff(i)$
	SA	LCP	BWT	
1	7	0	A	\$
2	6	0	G	A\$
3	4	1	T	AGA\$
4	2	1	G	ATAGA\$
5	5	0	A	GA\$
6	1	2	\$	GATAGA\$
7	3	0	A	TAGA\$

Figura : ESA para $T = GATAGA\$$

LCP

$LCP[i] = lcp(suff(i-1), suff(i))$ e $LCP[1] = 0$

BWT

$$BWT[i] = \begin{cases} T[SA[i]-1] & \text{se } SA[i] \neq 1 \\ \$ & \text{caso contrário} \end{cases}$$

Busca por padrões

Busca binária $O(m + \log n)$
FM-index [Ferragina and Manzini, 2000]
Árvore de sufixo [Abouelhoda et al., 2004]

Fundamentos

Vetor de sufixo generalizado (GSA):

- Seja $\mathcal{T} = \{T_1, \ldots, T_k\}$ um conjunto de k cadeias
- Vetor de pares de inteiros (i, j)
- Sufixos $T_i[j, n_i]$ ordenados lexicograficamente ($\$ < A < C < G < T$)

i	$GESA[i]$			$suff(i)$
	GSA	LCP	BWT	
1	(1,7)	0	A	\$
2	(2,7)	1	A	\$
3	(1,6)	0	G	A\$
4	(2,6)	1	G	A\$
5	(1,4)	1	T	AGA\$
6	(2,4)	3	G	AGA\$
7	(2,2)	3	T	AGAGA\$
8	(1,2)	1	G	ATAGA\$
9	(1,5)	0	A	GA\$
10	(2,5)	2	A	GA\$
11	(2,3)	2	A	GAGA\$
12	(1,1)	2	\$	GATAGA\$
13	(1,3)	0	A	TAGA\$
14	(2,1)	4	\$	TAGAGA\$

Figura : $GESA$ para $T_1 = GATAGA\$$ e $T_2 = TAGAGA\$$

Ordem lexicográfica $T_i[n_i, n_i] = \$$

$T_i[n_i, n_i] < T_j[n_j, n_j]$ se $i < j$

LCP e BWT

Estruturas auxiliares podem ser generalizadas

Busca por padrões

Busca binária $O(m + \log N)$
Sub-cadeia em comum mais longa
Identificação de repetições
...

Sumário

Algoritmo eGSA

eGSA: *External Generalized Enhanced Suffix Array Construction Algorithm*

- Baseado no algoritmo 2PMMS [Garcia-Molina et al., 1999]
- Entrada: um conjunto de k cadeias $\mathcal{T} = \{T_1, \ldots, T_k\}$
- Saída: $GESA = GSA + LCP + BWT$ para $\mathcal{T}$

Em resumo, eGSA funciona da seguinte forma:

- **Fase 1:** para cada $T_i \in \mathcal{T}$, construção em memória interna $\rightarrow SA_i$, LCP_i, e são armazenados em memória externa
- **Fase 2:** União dos vetores calculados obtendo $GESA$

Fase 1: Ordenação interna

Para cada $T_i \in \mathcal{T}$:

1. T_i é carregada em memória interna
2. Construção de SA_i e LCP_i utilizando qualquer algoritmo em memória interna (*e.g.* [Nong et al., 2011, Kasai et al., 2001, Fischer, 2011])
3. Vetores auxiliares: BWT_i e PRE_i
4. Escrever vetor composto em memória externa

Observações

- Caso não haja memória interna suficiente $\rightarrow$ construção em memória externa (*e.g.* [Bingmann et al., 2013]).
- No caso de muitas cadeias, $\mathcal{T}$ pode ser dividido em r partições $\mathcal{T}^1, \ldots, \mathcal{T}^r$, $\rightarrow$ construir um *GSA* para cada partição utilizando algum algoritmo em memória interna (*e.g.* [Shi, 1996]).

Fase 1: Ordenação interna

Para cada $T_i \in \mathcal{T}$:

1. T_i é carregada em memória interna
2. Construção de SA_i e LCP_i utilizando qualquer algoritmo em memória interna (*e.g.* [Nong et al., 2011, Kasai et al., 2001, Fischer, 2011])
3. Vetores auxiliares: BWT_i e PRE_i
4. Escrever vetor composto em memória externa

Vetores auxiliares

- $BWT_i[i] = T_i[SA_i[i] - 1]$ if $SA_i[i] \neq 1$ ou $BWT_i[i] = \$$ caso contrário
- $PRE_i[i]$ armazena o prefixo de $T_i[SA_i[i], n_i]$

Observação

- Utilizados para melhorar a segunda fase do algoritmo

Fase 1: Ordenação interna

Para cada $T_i \in \mathcal{T}$:

1. T_i é carregada em memória interna
2. Construção de SA_i e LCP_i utilizando qualquer algoritmo em memória interna (*e.g.* [Nong et al., 2011, Kasai et al., 2001, Fischer, 2011])
3. Vetores auxiliares: BWT_i e PRE_i
4. Escrever vetor composto em memória externa

Vetor composto

$R_i[j] = \langle SA_i[j], LCP_i[j], BWT_i[j], PRE_i[j] \rangle$

Fase 1: Ordenação interna

Vetor de prefixo de T_i, (PRE_i):

- PRE_i armazena o início de cada sufixo (de tamanho p) em SA_i
- $PRE_i[j] = T_i[SA_i[j], SA_i[j] + p]$ [Barsky et al., 2008]
- $PRE_i[j] = T_i[SA_i[j] + h_j, SA_i[j] + h_j + p]$, para $h_j = min(LCP_i[j], h_{j-1} + p)$

Figura : Exemplo para $T_1 = GATAGA\$$ e $p = 3$

j	$SA_1[j]$	$LCP_1[j]$	$PRE_1[j]$	$suff(j)$
1	7	0	\$\$\$	\$
2	6	0	A\$\$	A\$
3	4	1	AGA	AGA\$
4	2	1	ATA	ATAGA\$
5	5	0	GA\$	GA\$
6	1	2	GAT	GATAGA\$
7	3	0	TAG	TAGA\$

Limitação

Entretanto, a probabilidade de que dois valores consecutivos sejam iguais é alta, já que os sufixos $T_i[SA_i[j-1], n_i]$ e $T_i[SA_i[j], n_i]$ estão ordenados em SA_i.

Fase 1: Ordenação interna

Vetor de prefixo de T_i, (PRE_i):

- PRE_i armazena o início de cada sufixo (de tamanho p) em SA_i
- $PRE_i[j] = T_i[SA_i[j], SA_i[j] + p]$ [Barsky et al., 2008]
- $PRE_i[j] = T_i[SA_i[j] + h_j, SA_i[j] + h_j + p]$, para $h_j = min(LCP_i[j], h_{j-1} + p)$

Figura : Exemplo para $T_1 = GATAGA\$$ e $p = 3$

j	$SA_1[j]$	$LCP_1[j]$	$PRE_1[j]$	$suff(j)$
1	7	0	\$\$\$	\$
2	6	0	A\$\$	A\$
3	4	1	GA\$	AGA\$
4	2	1	TAG	ATAGA\$
5	5	0	GA\$	GA\$
6	1	2	TAG	GATAGA\$
7	3	0	TAG	TAGA\$

Melhoria [Sinha et al., 2008]

$PRE_i[j]$ armazena os p primeiros caracteres não comuns a $T_i[SA_i[j-1], n_i]$ e $T_i[SA_i[j], n_i]$ ou os que são comuns mas não foram armazenados anterioriormente

Fase 2: União externa

União de todos os vetores R_i construídos na primeira fase utilizando:

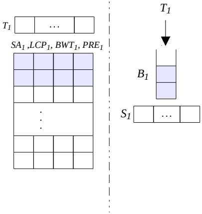
- Para cada $T_i \in \mathcal{T}$:
  - Buffer de partição $B_i \rightarrow$ blocos de $R_i = \langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - Buffer de cadeia $S_i \rightarrow$ sub-cadeias dos sufixos de T_i
- Árvore binária de comparação (*heap*), cada nó representa o elemento topo (sufixo) de cada B_i
- Buffer de saída $\rightarrow$ $GESA = GSA + LCP + BWT$

Quando o buffer de saída está cheio, ele é escrito em memória externa

Fase 2: União externa

União de todos os vetores R_i construídos na primeira fase utilizando:

- Para cada $T_i \in \mathcal{T}$:
  - Buffer de partição $B_i \to$ blocos de $R_i = \langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - Buffer de cadeia $S_i \to$ sub-cadeias dos sufixos de T_i
- Árvore binária de comparação (*heap*), cada nó representa o elemento topo (sufixo) de cada B_i
- Buffer de saída $\to$ $GESA = GSA + LCP + BWT$

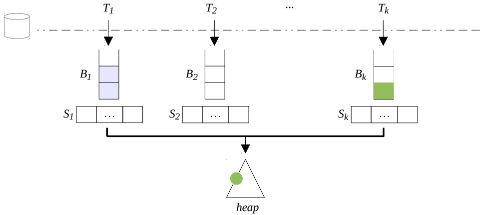


Quando o buffer de saída está cheio, ele é escrito em memória externa

Fase 2: União externa

União de todos os vetores R_i construídos na primeira fase utilizando:

- Para cada $T_i \in \mathcal{T}$:
  - Buffer de partição $B_i \rightarrow$ blocos de $R_i = \langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - Buffer de cadeia $S_i \rightarrow$ sub-cadeias dos sufixos de T_i
- Árvore binária de comparação (*heap*), cada nó representa o elemento topo (sufixo) de cada B_i
- Buffer de saída $\rightarrow$ $GESA = GSA + LCP + BWT$

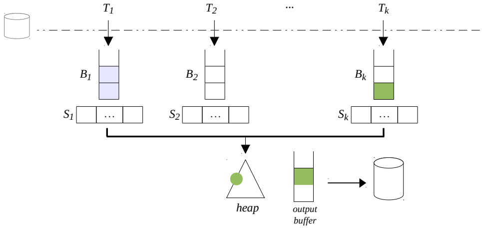


Quando o buffer de saída está cheio, ele é escrito em memória externa

Fase 2: União externa

União de todos os vetores R_i construídos na primeira fase utilizando:

- Para cada $T_i \in \mathcal{T}$:
  - Buffer de partição $B_i \rightarrow$ blocos de $R_i = \langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - Buffer de cadeia $S_i \rightarrow$ sub-cadeias dos sufixos de T_i
- Árvore binária de comparação (*heap*), cada nó representa o elemento topo (sufixo) de cada B_i
- Buffer de saída $\rightarrow$ $GESA = GSA + LCP + BWT$



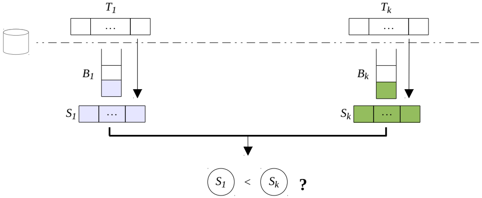
Quando o buffer de saída está cheio, ele é escrito em memória externa

Fase 2: União externa

A comparação entre os elementos na *heap* constitui a operação mais sensível nessa fase do algoritmo

Abordagem simples:

- Para cada comparação é necessário acessar T_i em memória externa
- Essas comparações podem exigir muitos acessos aleatórios à memória externa



Método melhorado para comparação de sufixos na *heap*:

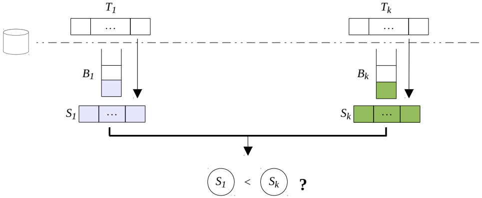
Para reduzir o número de acessos à memória externa, são propostas três estratégias: (i) montagem de prefixo; (ii) comparações de LCP; e (iii) indução de sufixos

Fase 2: União externa

A comparação entre os elementos na *heap* constitui a operação mais sensível nessa fase do algoritmo

Abordagem simples:

- Para cada comparação é necessário acessar T_i em memória externa
- Essas comparações podem exigir muitos acessos aleatórios à memória externa



Método melhorado para comparação de sufixos na *heap*:

Para reduzir o número de acessos à memória externa, são propostas três estratégias: (i) montagem de prefixo; (ii) comparações de LCP; e (iii) indução de sufixos

Fase 2: União externa

(i) Montagem de prefixo

- PRE_i é utilizado para carregar o início de $T_i[SA_i[j], n_i]$ em S_i
- Utilizando LCP_i e PRE_i podemos concatenar $(\cdot)$ $PRE_i[m]$ anteriores
  - $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$
  - $h_j = min(LCP_i[j], h_{j-1} + p),\ h_0 = 0$

j	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$T_1[SA[j], n_1]$
...	...	...	...	...	...
5	5	0	A	GA$	GA$
...	...	...	...	...	...

S_1						
	G	A	$	#	G	#

Exemplo

$j = 5,\ h_5 = 0$
$S_1 = GA\$\#$

Fase 2: União externa

(i) Montagem de prefixo

- PRE_i é utilizado para carregar o início de $T_i[SA_i[j], n_i]$ em S_i
- Utilizando LCP_i e PRE_i podemos concatenar $(\cdot)$ $PRE_i[m]$ anteriores
  - $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$
  - $h_j = min(LCP_i[j], h_{j-1} + p), \; h_0 = 0$

j	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$T_1[SA[j], n_1]$
...	...	...	...	...	...
5	5	0	A	GA\$	GA\$
6	1	2	\$	TAG	GATAG..

S_1	G	A	T	A	G	#

Exemplo

$j = 6, \; h_6 = min(LCP_i[6], h_5 + p) = min(2, 0 + 2) = 2$
$S_1 = S_1[1, 2] \cdot PRE_1[5] \cdot \# = GA \cdot TA \cdot \#$

Concatenação de $PRE_i[j]$ com os prefixos dos sufixos anteriores à $SA_i[j]$, armazenados em $PRE_i[m]$, para $m = 1, 2, \ldots, j - 1$.

Fase 2: União externa

(i) Montagem de prefixo

- PRE_i é utilizado para carregar o início de $T_i[SA_i[j], n_i]$ em S_i
- Utilizando LCP_i e PRE_i podemos concatenar $(\cdot)$ $PRE_i[m]$ anteriores
  - $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$
  - $h_j = min(LCP_i[j], h_{j-1} + p),\ h_0 = 0$

j	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$T_1[SA[j], n_1]$
...	...	...	...	...	...
5	5	0	A	GA$	GA$
6	1	2	$	TAG	GATAG..

S_1	G	A	T	A	G	#

Exemplo

$j = 6,\ h_6 = min(LCP_i[6], h_5 + p) = min(2, 0 + 2) = 2$
$S_1 = S_1[1,2] \cdot PRE_1[5] \cdot \# = GA \cdot TA \cdot \#$

Operação de I/O $\rightarrow$ apenas se a comparação envolver $h_j + p$ caracteres

Fase 2: União externa

(ii) Comparações de LCP

- Os valores de *lcp* podem ser utilizados para otimizar comparações de cadeias [Ng and Kakehi, 2008]

Lema 1:

Seja $S_1 < S_2$ e $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ então $lcp(S_2, S_k) \geq l$

Observação

Podemos iniciar a comparação de S_2 e S_k a partir de l

Fase 2: União externa

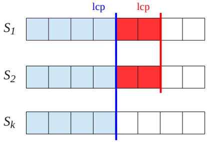
(ii) Comparações de LCP

- Os valores de *lcp* podem ser utilizados para otimizar comparações de cadeias [Ng and Kakehi, 2008]

Lema 1:

Seja $S_1 < S_2$ e $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ então $lcp(S_2, S_k) \geq l$



Observação

Podemos iniciar a comparação de S_2 e S_k a partir de l

Fase 2: União externa

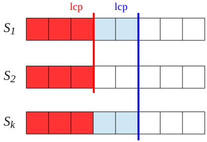
(ii) Comparações de LCP

- Os valores de *lcp* podem ser utilizados para otimizar comparações de cadeias [Ng and Kakehi, 2008]

Lema 1:

Seja $S_1 < S_2$ e $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ então $lcp(S_2, S_k) \geq l$



Observação

Podemos iniciar a comparação de S_2 e S_k a partir de l

Fase 2: União externa

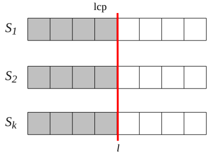
(ii) Comparações de LCP

- Os valores de *lcp* podem ser utilizados para otimizar comparações de cadeias [Ng and Kakehi, 2008]

Lema 1:

Seja $S_1 < S_2$ e $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ então $lcp(S_2, S_k) \geq l$



Observação

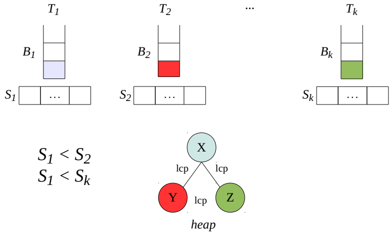
Podemos iniciar a comparação de S_2 e S_k a partir de l

Fase 2: União externa

(ii) Comparações de LCP

Sejam X, Y e Z nós na *heap* representando $B_1[i]$, $B_2[j]$ e $B_k[k]$

- Conforme X é removido da *heap*, $B_1[i]$ é movido para o *buffer* de saída
- X é substituído por outro nó W representando $B_1[i+1]$.
- A comparação de W com Y e Z pode utilizar o Lema 1



Exemplo

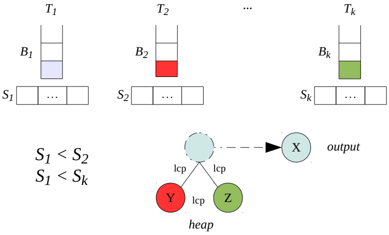
Se $lcp(X, W) > lcp(X, Y)$ e $lcp(X, W) > lcp(X, Z)$ então $W < Y$ e $W < Z$
$\rightarrow$ W é o próximo a ser removido da *heap* sem comparação de cadeias

Fase 2: União externa

(ii) Comparações de LCP

Sejam X, Y e Z nós na *heap* representando $B_1[i]$, $B_2[j]$ e $B_k[k]$

- Conforme X é removido da *heap*, $B_1[i]$ é movido para o *buffer* de saída
- X é substituído por outro nó W representando $B_1[i+1]$.
- A comparação de W com Y e Z pode utilizar o Lema 1



Exemplo

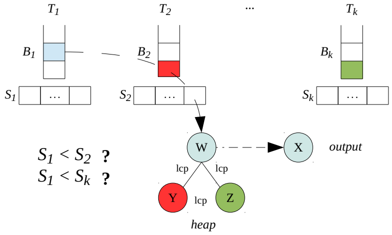
Se $lcp(X, W) > lcp(X, Y)$ e $lcp(X, W) > lcp(X, Z)$ então $W < Y$ e $W < Z$
$\rightarrow$ W é o próximo a ser removido da *heap* sem comparação de cadeias

Fase 2: União externa

(ii) Comparações de LCP

Sejam X, Y e Z nós na *heap* representando $B_1[i]$, $B_2[j]$ e $B_k[k]$

- Conforme X é removido da *heap*, $B_1[i]$ é movido para o *buffer* de saída
- X é substituído por outro nó W representando $B_1[i+1]$.
- A comparação de W com Y e Z pode utilizar o Lema 1



Exemplo

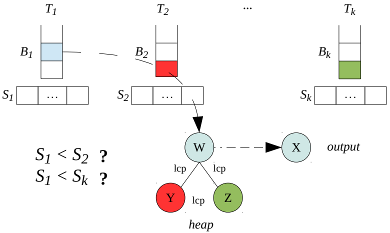
Se $lcp(X, W) > lcp(X, Y)$ e $lcp(X, W) > lcp(X, Z)$ então $W < Y$ e $W < Z$
$\rightarrow$ W é o próximo a ser removido da *heap* sem comparação de cadeias

Fase 2: União externa

(ii) Comparações de LCP

Sejam X, Y e Z nós na *heap* representando $B_1[i]$, $B_2[j]$ e $B_k[k]$

- Conforme X é removido da *heap*, $B_1[i]$ é movido para o *buffer* de saída
- X é substituído por outro nó W representando $B_1[i+1]$.
- A comparação de W com Y e Z pode utilizar o Lema 1



Exemplo

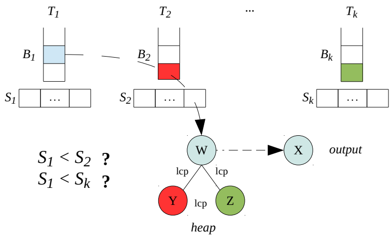
Se $lcp(X, W) > lcp(X, Y)$ e $lcp(X, W) > lcp(X, Z)$ então $W < Y$ e $W < Z$
$\rightarrow$ W é o próximo a ser removido da *heap* sem comparação de cadeias

Fase 2: União externa

(ii) Comparações de LCP

Sejam X, Y e Z nós na *heap* representando $B_1[i]$, $B_2[j]$ e $B_k[k]$

- Conforme X é removido da *heap*, $B_1[i]$ é movido para o *buffer* de saída
- X é substituído por outro nó W representando $B_1[i+1]$.
- A comparação de W com Y e Z pode utilizar o Lema 1



Exemplo

Se $lcp(X, W) > lcp(X, Y)$ e $lcp(X, W) > lcp(X, Z)$ então $W < Y$ e $W < Z$
$\rightarrow$ W é o próximo a ser removido da *heap* sem comparação de cadeias

Fase 2: União externa

(iii) Indução de sufixos

- Podemos determinar a ordem de sufixos não ordenados por meio dos sufixos já ordenados

Essa técnica tem sido empregada por diferentes algoritmos de construção em memória interna [Puglisi et al., 2007] e em memória externa [Bingmann et al., 2013]

Fase 2: União externa

(iii) Indução de sufixos

- Podemos determinar a ordem de sufixos não ordenados por meio dos sufixos já ordenados

Lema 2:

Seja Π o conjunto de todos os sufixos de $\mathcal{T}$

- Se $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ é o menor sufixo de Π
- Então $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ é o menor β-*sufixo* de $\Pi \setminus \{T_i[j, n_i]\}$

$$T_i[j, n_i] < T_a[h, n_a] < \cdots < T_i[j-1, n_i] = \alpha \cdot T_i[j, n_i] < \cdots < T_a[h-1, n_a] = \alpha \cdot T_a[h, n_a] < \ldots$$

β

Fase 2: União externa

(iii) Indução de sufixos

- Podemos determinar a ordem de sufixos não ordenados por meio dos sufixos já ordenados

Lema 2:

Seja Π o conjunto de todos os sufixos de $\mathcal{T}$

- Se $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ é o menor sufixo de Π
- Então $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ é o menor β-*sufixo* de $\Pi \setminus \{T_i[j, n_i]\}$

Indução:

- Remover $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ de Π
- Induzir $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ para a primeira posição disponível no β-*bucket*
- β-*bucket*: uma partição de SA que contém apenas β-*sufixos*

Note que se $\alpha > \beta$ o sufixo $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ já foi ordenado

Fase 2: União externa

(iii) Indução de sufixos

Utilizando o Lema 2 podemos induzir sufixos na *heap*:

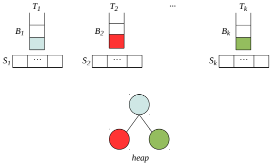
- Π é o conjunto de todos os sufixos não ordenados de $\mathcal{T}$ (restantes em B_i)
- Encontre o menor sufixo $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ *buffer* de saída
- Induza $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ se $\alpha < \beta$ (informação em $BWT_i[j]$)
- Quando o primeiro β-*sufixo* $T_i[j-1, n_i]$ for o menor na *heap*, F_β é lido, e outros sufixos são induzidos

Fase 2: União externa

(iii) Indução de sufixos

Utilizando o Lema 2 podemos induzir sufixos na *heap*:

- Π é o conjunto de todos os sufixos não ordenados de $\mathcal{T}$ (restantes em B_i)
- Encontre o menor sufixo $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ *buffer* de saída
- Induza $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ se $\alpha < \beta$ (informação em $BWT_i[j]$)
- Quando o primeiro β-*sufixo* $T_i[j-1, n_i]$ for o menor na *heap*, F_β é lido, e outros sufixos são induzidos

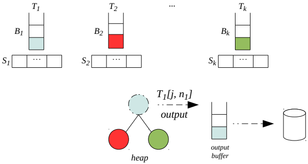


Fase 2: União externa

(iii) Indução de sufixos

Utilizando o Lema 2 podemos induzir sufixos na *heap*:

- $\mathit{\Pi}$ é o conjunto de todos os sufixos não ordenados de $\mathcal{T}$ (restantes em B_i)
- Encontre o menor sufixo $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i]$ → *buffer* de saída
- Induza $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ se $\alpha < \beta$ (informação em $BWT_i[j]$)
- Quando o primeiro β-*sufixo* $T_i[j-1, n_i]$ for o menor na *heap*, F_β é lido, e outros sufixos são induzidos

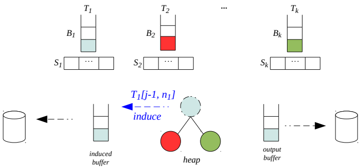


Fase 2: União externa

(iii) Indução de sufixos

Utilizando o Lema 2 podemos induzir sufixos na *heap*:

- $\mathit{\Pi}$ é o conjunto de todos os sufixos não ordenados de $\mathcal{T}$ (restantes em B_i)
- Encontre o menor sufixo $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ *buffer* de saída
- Induza $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ se $\alpha < \beta$ (informação em $BWT_i[j]$)
- Quando o primeiro β-*sufixo* $T_i[j-1, n_i]$ for o menor na *heap*, F_β é lido, e outros sufixos são induzidos



Buffer de sufixos induzidos I, composto por $|\Sigma|$ *buffers* I_β

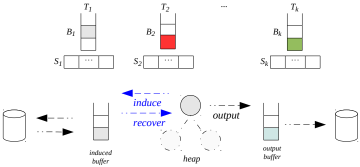
- Quando I_β está cheio, ele é escrito no arquivo F_β em memória externa

Fase 2: União externa

(iii) Indução de sufixos

Utilizando o Lema 2 podemos induzir sufixos na *heap*:

- Π é o conjunto de todos os sufixos não ordenados de $\mathcal{T}$ (restantes em B_i)
- Encontre o menor sufixo $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ *buffer* de saída
- Induza $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ se $\alpha < \beta$ (informação em $BWT_i[j]$)
- Quando o primeiro β-*sufixo* $T_i[j-1, n_i]$ for o menor na *heap*, F_β é lido, e outros sufixos são induzidos



Observação

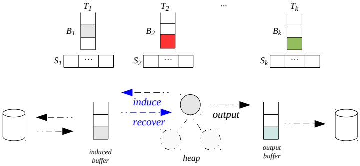
Sufixos induzidos também podem induzir

Fase 2: União externa

(iii) Indução de sufixos

Utilizando o Lema 2 podemos induzir sufixos na *heap*:

- Π é o conjunto de todos os sufixos não ordenados de $\mathcal{T}$ (restantes em B_i)
- Encontre o menor sufixo $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ *buffer* de saída
- Induza $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ se $\alpha < \beta$ (informação em $BWT_i[j]$)
- Quando o primeiro β-*sufixo* $T_i[j-1, n_i]$ for o menor na *heap*, F_β é lido, e outros sufixos são induzidos



Observação

Não é preciso compará-los novamente $\rightarrow$ seguir a ordem dos valores em F_α removendo os elementos de B_i

Sumário

Experimentos

O algoritmo eGSA foi analisado utilizando dados biológicos reais:

- Sequências de DNA (http://www.ensembl.org/)
- Proteínas (http://www.uniprot.org/)

Implementação:

- ANSI/C e compilado em GNU gcc, versão 4.6.3, com o parâmetro -O3
- Seu código fonte encontra-se disponível http://code.google.com/p/egsa/

Configuração:

- Linux Ubuntu 12.04/64 bits
- processador Intel Core i7 2,67 GHz, 8MB L2 cache
- 12 GB de memória interna
- disco SATA de 1 TB, 5900 RPM e 64MB cache

Testes de desempenho

Cadeias grandes: DNA

Foram gerados 5 conjuntos de testes (BDBs) utilizando os seguintes genomas:

- (1) *Homo sapiens*, (2) *Oryzias latipes*, (3) *Danio rerio*, (4) *Bos taurus*, (5) *Mus musculus* e (6) *Gallus gallus*

BDBs:

BDB	Genomas	nº cadeias	maior cadeia (MB)	*lcp-médio*	*lcp-máximo*	Total (GB)
D_1	2	24	29,37	19	4.073	0,54
D_2	6	30	186,14	17	5.476	0,92
D_3	3, 6	56	186,14	58	71.314	2,18
D_4	2, 3, 4	80	129,90	44	71.314	4,26
D_5	1, 4, 5, 6	105	226,69	59	168.246	8,50

Observações

- Os valores de *lcp-médio* e *lcp-máximo* indicam a dificuldade da ordenação
- Cada caractere ocupa 1 *byte* em memória

Testes de desempenho

Cadeias grandes: DNA

Foram gerados 5 conjuntos de testes (BDBs) utilizando os seguintes genomas:

- (1) *Homo sapiens*, (2) *Oryzias latipes*, (3) *Danio rerio*, (4) *Bos taurus*, (5) *Mus musculus* e (6) *Gallus gallus*

BDBs:

BDB	Genomas	nº cadeias	maior cadeia (MB)	*lcp-médio*	*lcp-máximo*	Total (GB)
D_1	2	24	29,37	19	4.073	0,54
D_2	6	30	186,14	17	5.476	0,92
D_3	3, 6	56	186,14	58	71.314	2,18
D_4	2, 3, 4	80	129,90	44	71.314	4,26
D_5	1, 4, 5, 6	105	226,69	59	168.246	8,50

Observações

- Os valores de *lcp-médio* e *lcp-máximo* indicam a dificuldade da ordenação
- Cada caractere ocupa 1 *byte* em memória

Testes de desempenho

Cadeias grandes: DNA

eGSA:

- Fase 1: algoritmo *inducing+sais-lite* [Fischer, 2011] para construir SA_i e LCP_i
- Tamanho de $p = 10$ para PRE_i
- Fase 2: os *buffers* S_i, B_i, saída e I foram configurados para utilizar 200 KB, 10 MB, 16 MB e 64 MB de memória interna

Comparação com o algoritmo eSAIS [Bingmann et al., 2013]:

- Estado da arte para construção de vetores de sufixo para uma única cadeia em memória externa

Limitação

- Entrada adaptada $\rightarrow$ todas as cadeias $T_i \in \mathcal{T}$ foram concatenadas em uma única cadeia $T = T_1\$_1 \cdot T_2\$_2 \cdot \ldots \cdot T_k\$_k$, de forma que $\$_i < \$_j$ se $i < j$ e $\$_i < \alpha, \forall \alpha \in \Sigma$.

Testes de desempenho

Cadeias grandes: DNA

eGSA:

- Fase 1: algoritmo *inducing+sais-lite* [Fischer, 2011] para construir SA_i e LCP_i
- Tamanho de $p = 10$ para PRE_i
- Fase 2: os *buffers* S_i, B_i, saída e I foram configurados para utilizar 200 KB, 10 MB, 16 MB e 64 MB de memória interna

Comparação com o algoritmo eSAIS [Bingmann et al., 2013]:

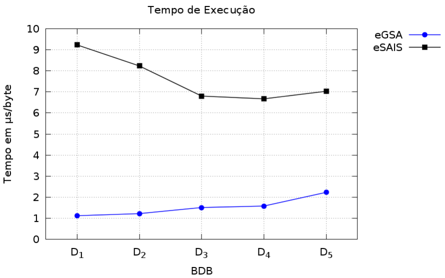
- Estado da arte para construção de vetores de sufixo para uma única cadeia em memória externa

Limitação

- Entrada adaptada $\rightarrow$ todas as cadeias $T_i \in \mathcal{T}$ foram concatenadas em uma única cadeia $T = T_1\$_1 \cdot T_2\$_2 \cdot \ldots \cdot T_k\$_k$, de forma que $\$_i < \$_j$ se $i < j$ e $\$_i < \alpha, \forall \alpha \in \Sigma$.

Testes de desempenho

Cadeias grandes: DNA

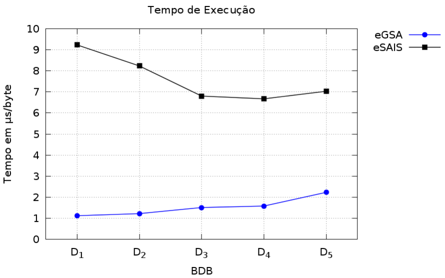


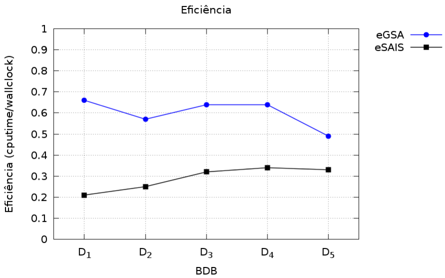
Tempo de execução

- eGSA foi muito mais rápido em todos os testes
- Tempo médio de 3,2 a 8,3 vezes menor que o eSAIS

Testes de desempenho

Cadeias grandes: DNA



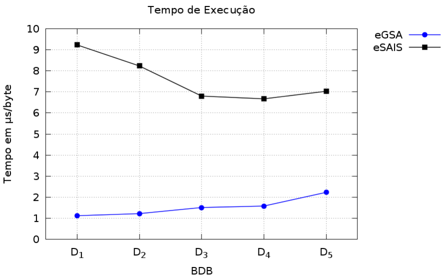


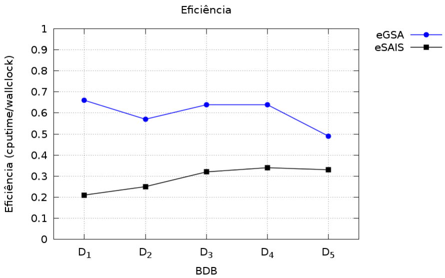
Eficiência

- Indica o proporção do *cputime* no tempo total

Testes de desempenho

Cadeias grandes: DNA



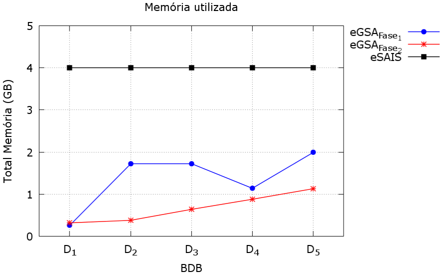


Eficiência

- Pode-se observar que a eficiência do eGSA diminui no BDB D_5
- Isso ocorre devido aos valores de *lcp-médio* e *lcp-máximo*, que aumentam 1,34 e 2,35 vezes do D_4 para o D_5

Testes de desempenho

Cadeias grandes: DNA

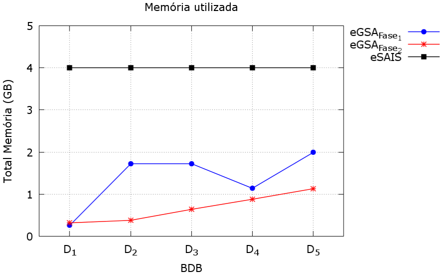


Memória interna do eSAIS

- Parâmetro utilizado pelo eSAIS no início da execução do algoritmo
- 4,0 GB

Testes de desempenho

Cadeias grandes: DNA



Memória interna do eGSA

- Fase 1: 1,99 GB para a construção do vetor de sufixo da maior cadeia do BDB D_5, a qual possui tamanho de 226,69 MB
- Fase 2: 1,1 GB para o BDB D_5

Testes de desempenho

Cadeias pequenas: proteínas

Foram gerados 5 conjuntos de testes (BDBs) de tamanhos entre 0,22 GB e 3,64 GB com 0,7 a 11 milhões de cadeias a partir do arquivo *uniprot_trembl.fasta*[1]

BDBs:

BDB	nº cadeias	menor (bp)	maior (bp)	média (bp)	*lcp-médio*	*lcp-máximo*	Total (GB)
A_1	0,7M	78	12.220	344	26	72	0,22
A_2	1,1M	78	12.220	378	42	72	0,38
A_3	2,3M	70	36.686	398	48	88	0,85
A_4	4,8M	62	36.686	403	63	107	1,80
A_5	11,2M	62	36.686	348	65	167	3,64

Observações

- Os valores de *lcp-médio* e *lcp-máximo* indicam a dificuldade da ordenação
- Cada caractere ocupa 1 *byte* em memória

[1] `http://www.uniprot.org/downloads`

Testes de desempenho

Cadeias pequenas: proteínas

Foram gerados 5 conjuntos de testes (BDBs) de tamanhos entre 0,22 GB e 3,64 GB com 0,7 a 11 milhões de cadeias a partir do arquivo *uniprot_trembl.fasta*[1]

BDBs:

BDB	nº cadeias	menor (bp)	maior (bp)	média (bp)	*lcp-médio*	*lcp-máximo*	Total (GB)
A_1	0,7M	78	12.220	344	26	72	0,22
A_2	1,1M	78	12.220	378	42	72	0,38
A_3	2,3M	70	36.686	398	48	88	0,85
A_4	4,8M	62	36.686	403	63	107	1,80
A_5	11,2M	62	36.686	348	65	167	3,64

Observações

- Os valores de *lcp-médio* e *lcp-máximo* indicam a dificuldade da ordenação
- Cada caractere ocupa 1 *byte* em memória

[1] `http://www.uniprot.org/downloads`

Testes de desempenho

Cadeias pequenas: proteínas

eGSA:

- Fase 1: $\mathcal{T}$ foi dividido em r partições de tamanhos iguais
  - SAIS [Nong et al., 2011] adaptado[2] para construção de *GSA*
  - Kasai [Kasai et al., 2001] para construção de *LCP*
- Tamanho de $p = 5$ para PRE_i
- Fase 2: os *buffers* S_i, B_i, D e I foram configurados para utilizar 200 *bytes*, 10 MB, 16 MB e 64 MB de memória interna

Comparação:

- No melhor de nosso conhecimento, não existem trabalhos correlatos que possam utilizados para conjuntos com muitas cadeias

[2] disponível em `https://github.com/jts/sga/blob/master/src/SuffixTools/`

Testes de desempenho

Cadeias pequenas: proteínas

eGSA:

- Fase 1: $\mathcal{T}$ foi dividido em r partições de tamanhos iguais
  - SAIS [Nong et al., 2011] adaptado[2] para construção de *GSA*
  - Kasai [Kasai et al., 2001] para construção de *LCP*
- Tamanho de $p = 5$ para PRE_i
- Fase 2: os *buffers* S_i, B_i, D e I foram configurados para utilizar 200 *bytes*, 10 MB, 16 MB e 64 MB de memória interna

Comparação:

- No melhor de nosso conhecimento, não existem trabalhos correlatos que possam utilizados para conjuntos com muitas cadeias

Observação

$R_i[j] = \langle GSA_i[j], LCP_i[j], BWT_i[j], PRE_i[j] \rangle$, isto é, GSA_i ao invés de SA_i

[2] disponível em `https://github.com/jts/sga/blob/master/src/SuffixTools/`

Testes de desempenho

Cadeias pequenas: proteínas

eGSA:

- Fase 1: $\mathcal{T}$ foi dividido em r partições de tamanhos iguais
  - SAIS [Nong et al., 2011] adaptado[2] para construção de *GSA*
  - Kasai [Kasai et al., 2001] para construção de *LCP*
- Tamanho de $p = 5$ para PRE_i
- Fase 2: os *buffers* S_i, B_i, D e I foram configurados para utilizar 200 *bytes*, 10 MB, 16 MB e 64 MB de memória interna

Comparação:

- No melhor de nosso conhecimento, não existem trabalhos correlatos que possam utilizados para conjuntos com muitas cadeias

Problema

A estratégia de adaptar a entrada não pode ser utilizada, devido ao número k de cadeias em $\mathcal{T}$ ser muito grande

[2] disponível em `https://github.com/jts/sga/blob/master/src/SuffixTools/`

Testes de desempenho

Cadeias pequenas: proteínas

eGSA:

- Fase 1: $\mathcal{T}$ foi dividido em r partições de tamanhos iguais
  - SAIS [Nong et al., 2011] adaptado[2] para construção de *GSA*
  - Kasai [Kasai et al., 2001] para construção de *LCP*
- Tamanho de $p = 5$ para PRE_i
- Fase 2: os *buffers* S_i, B_i, D e I foram configurados para utilizar 200 *bytes*, 10 MB, 16 MB e 64 MB de memória interna

Comparação:

- No melhor de nosso conhecimento, não existem trabalhos correlatos que possam utilizados para conjuntos com muitas cadeias

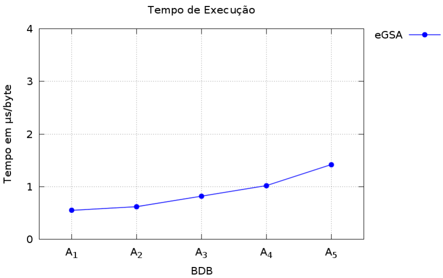
Observação

Os experimentos para conjuntos de cadeias pequenas consideram apenas o eGSA

[2] disponível em `https://github.com/jts/sga/blob/master/src/SuffixTools/`

Testes de desempenho

Cadeias pequenas: proteínas

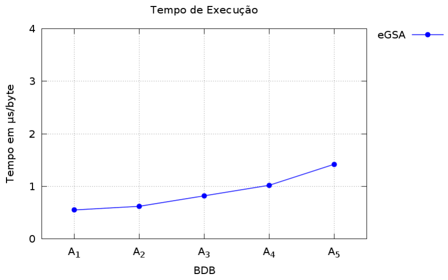


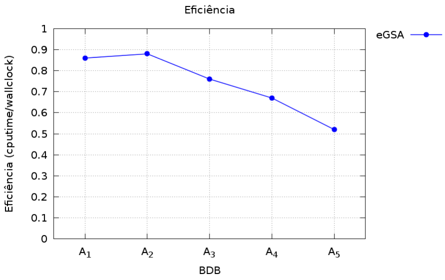
Tempo de execução

- Pode-se observar que os tempos são competitivos
- Além disso, eGSA é o único algoritmo que pode ser aplicado para esses BDBs

Testes de desempenho

Cadeias pequenas: proteínas



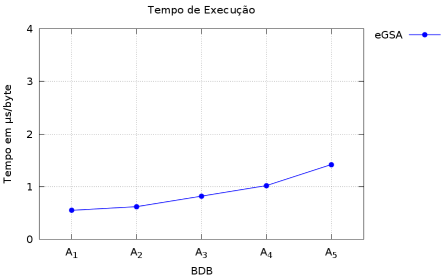


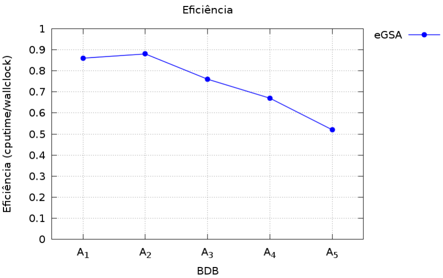
Eficiência

- Indica o proporção do *cputime* no tempo total

Testes de desempenho

Cadeias pequenas: proteínas



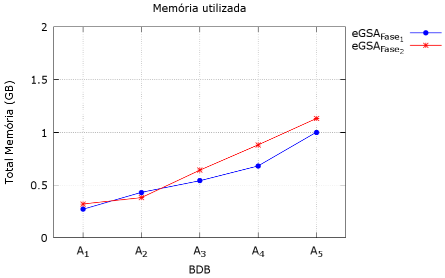


Eficiência

- Pode-se observar que a eficiência do eGSA diminui a partir do BDB A_2
- Isso ocorre devido aos valores de *lcp-médio* e *lcp-máximo* que aumentam nos BDBs A_2, A_3, A_4 e A_5

Testes de desempenho

Cadeias pequenas: proteínas



Memória interna do eGSA

- Fase 1: 1,03 GB para a construção da maior partição do BDB A_5, a qual possui tamanho de 85,89 MB
- Fase 2: 1,1 GB para o BDB A_5

Testes de desempenho

Características específicas do eGSA

- Tamanho para o parâmetro p do vetor de prefixos
- Sufixos induzidos
- Efeito de cada estratégia na *heap*
  - Montagem de prefixo
  - Comparações de *LCP*
  - Indução de sufixos

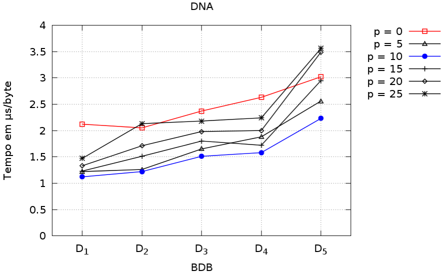
BDBs

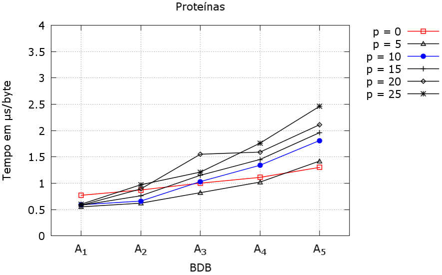
Testes com os BDBs de DNA e de proteínas

Testes de desempenho

Tamanho do vetor de prefixo

$p = 0, 5, 10, 15, 20, 25$





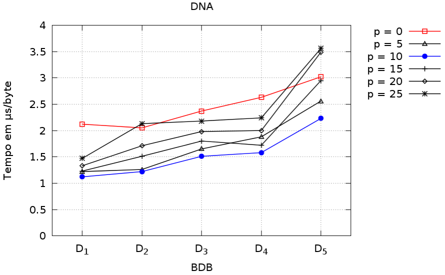
DNA e Proteínas

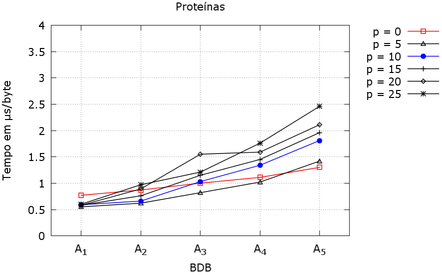
- DNA: $p = 10$ foi o melhor em todos os experimentos
- Proteínas: $p = 5$ foi o melhor em quase todos os experimentos

Testes de desempenho

Tamanho do vetor de prefixo

$p = 0, 5, 10, 15, 20, 25$





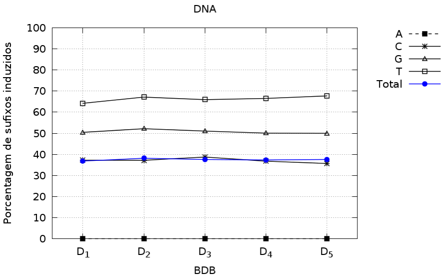
Conclusões

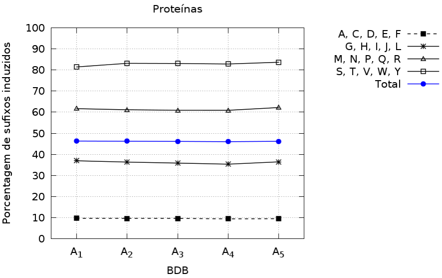
- Conforme p aumenta, o prefixo montado aumenta e melhora o desempenho do algoritmo até que B_i diminui fazendo com que o número de acessos à memória externa para carregar R_i tenha um impacto no desempenho

Testes de desempenho

Sufixos induzidos

Porcentagem de cada α-*sufixo*





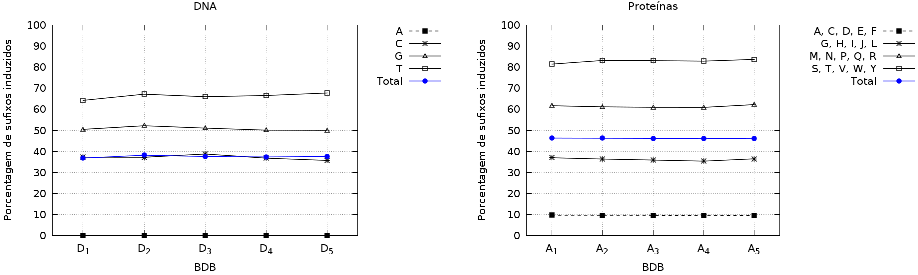
Observações

- A porcentagem dos sufixos que começam com "A" é muito pequena
- Esses sufixos podem ser induzidos apenas por sufixos que começam com $

Testes de desempenho

Sufixos induzidos

Porcentagem de cada α-*sufixo*



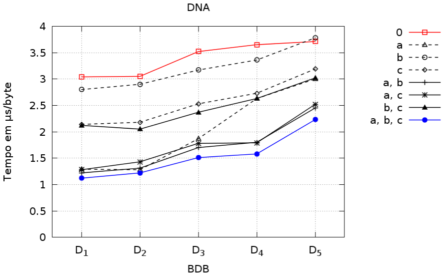
DNA e Proteínas, em média

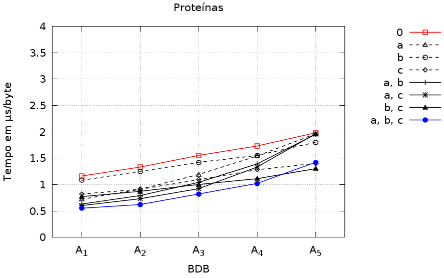
- DNA: $37,49\%$
- Proteínas: $46,18\%$

Testes de desempenho

Efeito de cada estratégia da *heap*

Foram desenvolvidas 8 versões do algoritmo eGSA:
(a) montagem de prefixo, (b) comparações de LCP e (c) indução de sufixos

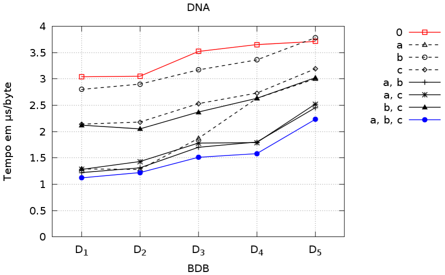


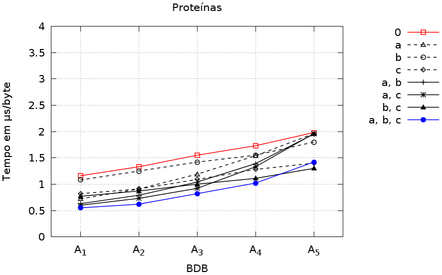


Testes de desempenho

Efeito de cada estratégia da *heap*

Foram desenvolvidas 8 versões do algoritmo eGSA:
(a) montagem de prefixo, (b) comparações de LCP e (c) indução de sufixos





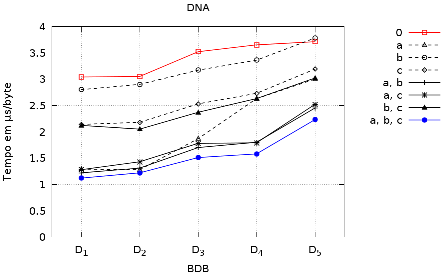
DNA e Proteínas

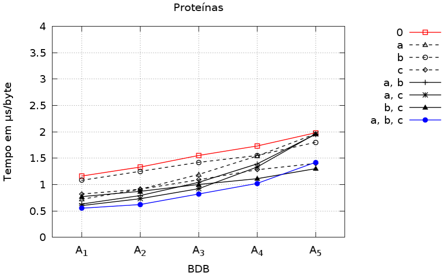
- DNA: a versão completa do algoritmo (a, b, c) foi a melhor em todos os casos
- Proteínas: apenas no BDB A_5, a versão completa não foi a melhor

Testes de desempenho

Efeito de cada estratégia da *heap*

Foram desenvolvidas 8 versões do algoritmo eGSA:
(a) montagem de prefixo, (b) comparações de LCP e (c) indução de sufixos





Conclusão

- Todas as estratégias, individualmente, melhoraram o desempenho do algoritmo
- DNA: a estratégia sem melhoria ainda é melhor do que o eSAIS

Sumário

Conclusões

Principais contribuições:

- Proposta do algoritmo eGSA para construção de vetores de sufixo generalizados aumentados em memória externa.
- Validação do algoritmo eGSA com conjuntos de cadeias pequenas e cadeias grandes, evidenciando um avanço no estado da arte.
- Revisão bibliográfica de algoritmos para construção de vetores de sufixo em memória externa.

Estágio BEPE (Apêndice)

- Montagem de genomas *de-novo* [Baets et al., 2012].
- No Apêndice é apresentado um algoritmo preliminar para a construção de grafos de cadeias utilizando vetores de sufixo generalizados

Conclusões

Principais contribuições:

- Proposta do algoritmo eGSA para construção de vetores de sufixo generalizados aumentados em memória externa.
- Validação do algoritmo eGSA com conjuntos de cadeias pequenas e cadeias grandes, evidenciando um avanço no estado da arte.
- Revisão bibliográfica de algoritmos para construção de vetores de sufixo em memória externa.

Estágio BEPE (Apêndice)

- Montagem de genomas *de-novo* [Baets et al., 2012].
- No Apêndice é apresentado um algoritmo preliminar para a construção de grafos de cadeias utilizando vetores de sufixo generalizados

Conclusões

Trabalhos Futuros:

- Adaptação do algoritmo eGSA para trabalhar com múltiplos discos
- Desenvolvimento de um método para particionar automaticamente o conjunto de cadeias na primeira fase no caso de conjuntos de cadeias pequenas
- Validação do algoritmo eGSA em outros domínios de dados

Publicações

1. **Louza, F. A.**, Telles, G. P. e Ciferri, C. D. A. . External Memory Generalized Suffix and LCP Arrays Construction. In: Proceedings of 24th Annual Symposium on Combinatorial Pattern Matching (CPM), 2013, Bad Herrenalb, Germany, p. 201-210.
2. **Louza, F. A.**, Hoffmann, S., Stadler, P. F., Telles, G. P. e Ciferri, C. D. A. . Suffix Arrays and Genome Assembly. In: Digital Proceedings of the 8th Brazilian Symposium on Bioinformatics (BSB) 2013, Recife, Brazil, p. 1-1.
3. **Louza, F. A.** e Ciferri, C. D. A.. Indexação de Dados Biológicos baseada em Vetores de Sufixo Generalizados para Disco. In: Anais do Workshop de Teses e Dissertações em Banco de Dados do 27º Simpósio Brasileiro de Banco de Dados (SBBD), 2012, São Paulo, Brasil. p. 87-92.
4. Chino, D. Y. T., **Louza, F. A.**, Traina, A. J. M., Ciferri, C. D. A. e Traina Jr., C.. Time Series Indexing Taking Advantage of the Generalized Suffix Tree. Journal of Information and Data Management - JIDM, v. 3, p. 101-109, 2012.

Final

Muito obrigado!

Dúvidas?

Sumário

Referências I

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004).
Replacing suffix trees with enhanced suffix arrays.
Journal of Discrete Algorithms, 2(1):53–86.

Baets, B. D., Fack, V., and Dawyndt, P. (2012).
Prospects and limitations of full-text index structures in genome analysis.
Nucleic Acids Research, pages 1–23.

Barsky, M., Stege, U., Thomo, A., and Upton, C. (2008).
A new method for indexing genomes using on-disk suffix trees.
Proc. CIKM, 236(1-2):649.

Bingmann, T., Fischer, J., and Osipov, V. (2013).
Inducing suffix and lcp arrays in external memory.
In *Proc. ALENEX*, pages 88–103.

Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. (2008).
Better external memory suffix array construction.
ACM J. of Experimental Algorithmics, 12.

Referências II

Ferragina, P. and Manzini, G. (2000).
Opportunistic Data Structures with Applications.
In *Proc. FOCS*, pages 390—-398.

Fischer, J. (2011).
Inducing the lcp-array.
In *Proc. Algorithms and Data Structures Symp.*, pages 374–385.

Garcia-Molina, H., Widom, J., and Ullman, J. D. (1999).
Database System Implementation.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Gonnet, G. H., Baeza-yates, R., and Snider, T. (1992).
New indices for text: PAT Trees and PAT arrays, pages 66–82.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001).
Linear-time longest-common-prefix computation in suffix arrays and its applications.
In *Proc. CPM*, pages 181–192.

Referências III

Louza, F. A., Telles, G. P., and Ciferri, C. D. A. (2013).
External Memory Generalized Suffix and LCP Arrays Construction.
In Fischer, J. and Sanders, P., editors, *Proc. CPM*, pages 201–210, Bad Herrenalb. Springer.

Manber, U. and Myers, E. W. (1990).
Suffix arrays: A new method for on-line string searches.
In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327.

Ng, W. and Kakehi, K. (2008).
Merging string sequences by longest common prefixes.
IPSJ Digital Courier, 4:69–78.

Nong, G., Zhang, S., and Chan, W. H. (2011).
Two efficient algorithms for linear time suffix array construction.
Computers, IEEE Transactions on, 60(10):1471–1484.

Referências IV

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (2007).
A taxonomy of suffix array construction algorithms.
ACM Computing Surveys, 39(2):1–31.

Shi, F. (1996).
Suffix arrays for multiple strings: A method for on-line multiple string searches.
In Jaffar, J. and Yap, R., editors, *Proc. ASIAN*, volume 1179 of *Lecture Notes in Computer Science*, pages 11–22. Springer Berlin / Heidelberg.

Sinha, R., Puglisi, S. J., Moffat, A., and Turpin, A. (2008).
Improving suffix array locality for fast pattern matching on disk.
In *Proc. ACM SIGMOD*, pages 661–672.

Extra:

(iii) Indução de sufixos

Os valores de *LCP* entre os sufixos induzidos também devem ser induzidos, desde que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*.

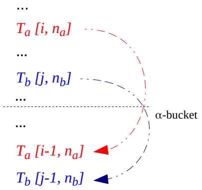
- $T_a[i, n_a]$ induz um α-*sufixo* e $T_b[j, n_b]$ induz o próximo α-*sufixo*
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$

Extra:

(iii) Indução de sufixos

Os valores de *LCP* entre os sufixos induzidos também devem ser induzidos, desde que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*.

- $T_a[i, n_a]$ induz um α-*sufixo* e $T_b[j, n_b]$ induz o próximo α-*sufixo*
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$

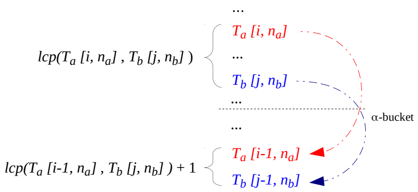


Extra:

(iii) Indução de sufixos

Os valores de *LCP* entre os sufixos induzidos também devem ser induzidos, desde que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*.

- $T_a[i, n_a]$ induz um α-*sufixo* e $T_b[j, n_b]$ induz o próximo α-*sufixo*
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$



Range minimum query(*rmq*):

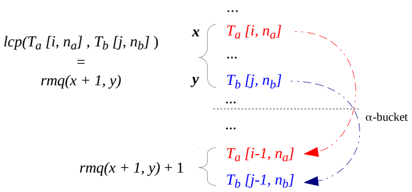
- $rmq_{LCP}(i, j) = \min_{i \leq k \leq j}\{LCP[k]\}$

Extra:

(iii) Indução de sufixos

Os valores de *LCP* entre os sufixos induzidos também devem ser induzidos, desde que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*.

- $T_a[i, n_a]$ induz um α-*sufixo* e $T_b[j, n_b]$ induz o próximo α-*sufixo*
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$



LCP e *rmq*:

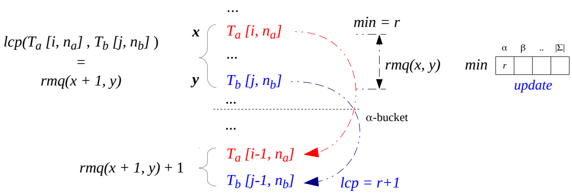
- $LCP(T_a[j, n_a], T_b[j, n_b]) = rmq_{LCP}(x+1, y)$
- É possível resolver a função $rmq_{LCP}(x, y)$ durante a indução dos sufixos

Extra:

(iii) Indução de sufixos

Os valores de *LCP* entre os sufixos induzidos também devem ser induzidos, desde que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*.

- $T_a[i, n_a]$ induz um α-*sufixo* e $T_b[j, n_b]$ induz o próximo α-*sufixo*
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1$.



Função $min[\alpha]$, $\forall \alpha \in \Sigma$

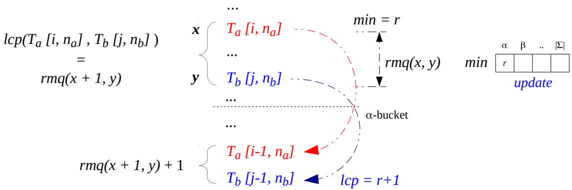
Quando um α-*sufixo* é induzido, $min[\alpha] \leftarrow \infty$, e $min[\alpha]$ é calculado até que o próximo α-*sufixo* seja induzido

Extra:

(iii) Indução de sufixos

Os valores de *LCP* entre os sufixos induzidos também devem ser induzidos, desde que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*.

- $T_a[i, n_a]$ induz um α-*sufixo* e $T_b[j, n_b]$ induz o próximo α-*sufixo*
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1$.



α-*bucket* e I_α

O valor de *LCP* deve ser induzido junto com o valor da cadeia em $I_\alpha = (cadeia, lcp)$

- $LCP = min[\alpha] + 1$ é induzido junto com $T_b[j-1, n_b]$ em I_α e $min[\alpha] \leftarrow max\_int$

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	GCC ...
j+1	b	1	\$	TAG	$GTAG$ ...

S_1	#	#	#	#	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor LCP generalizado, desde que os valores de LCP também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	GCC ...
j+1	b	1	\$	TAG	$GTAG$ ...

S_1						
	#	#	#	#	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor *LCP* generalizado, desde que os valores de *LCP* também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	*GCC* ...
j+1	b	1	\$	TAG	*GTAG* ...

S_1	#	T	A	G	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor *LCP* generalizado, desde que os valores de *LCP* também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	$GCC\ \ldots$
j+1	b	0	\$	GTA	$GTAG\ldots$

S_1	#	#	#	#	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor *LCP* generalizado, desde que os valores de *LCP* também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	*GCC* ...
j+1	b	0	\$	GTA	*GTAG* ...

S_1						
	G	T	A	#	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor *LCP* generalizado, desde que os valores de *LCP* também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	GCC ...
j+1	b	0	\$	GTA	$GTAG$ ...

S_1						
	G	T	A	#	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor *LCP* generalizado, desde que os valores de *LCP* também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Indução de sufixos

A montagem de prefixo deve considera os sufixos induzidos

Sejam dois α-*sufixos* consecutivos em SA_i, $SA_i[j] = a$ e $SA_i[j+1] = b$

- $T_i[a, n_i]$ é o último α-*sufixo* induzido
  - $T_i[a, n_i]$ será ignorado durante as comparações na *heap* (sem montagem)
  - $T_i[b, n_i]$ deve começar a montagem do início

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$suff(j)$
...	...	...	...	...	
j	a	0	A	GCC	*GCC* ...
j+1	b	1	\$	GTA	*GTAG* ...

S_1						
	G	T	A	#	#	erro

Solução:

- Se um sufixo será induzido ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- Esses valores não interferem na construção do vetor *LCP* generalizado, desde que os valores de *LCP* também são induzidos.
- O valor de $LCP[j+1]$ é sempre igual à 1, caso contrário, teria sido induzido

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

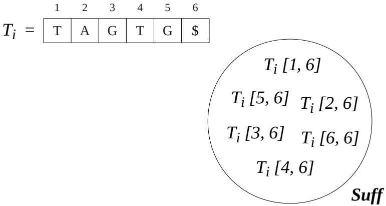
- $\mathit{\Pi}$ starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from $\mathit{\Pi}$
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

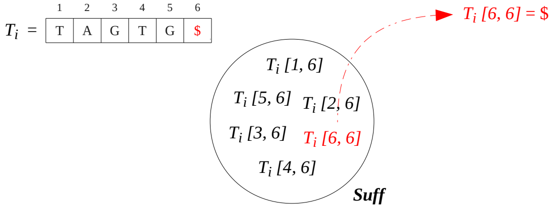


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- $\mathit{\Pi}$ starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from $\mathit{\Pi}$
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

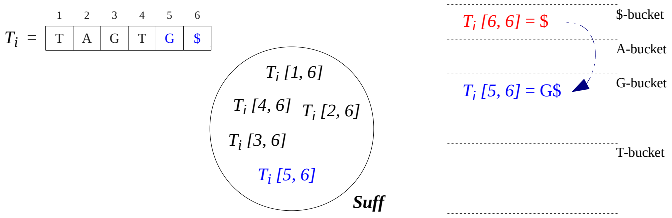


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



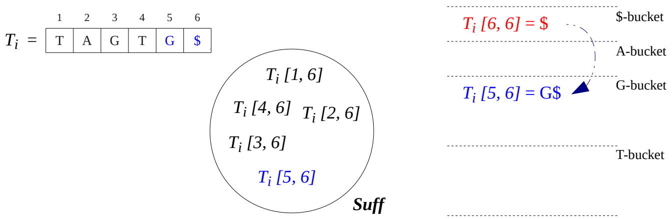
We define β-bucket is a partition of SA that contains only suffixes starting with β

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



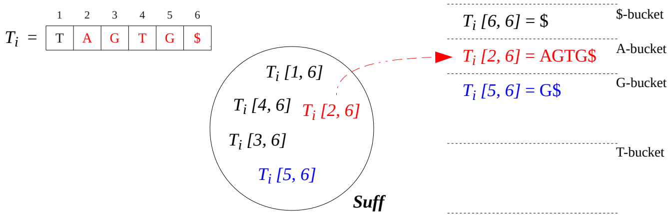
The induced suffixes $T_i[j-1, n_i] = \alpha \cdot T_i[j, n_i]$ cannot be removed from Π because they must induce suffixes $T_i[j-2, n_i]$ as well

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

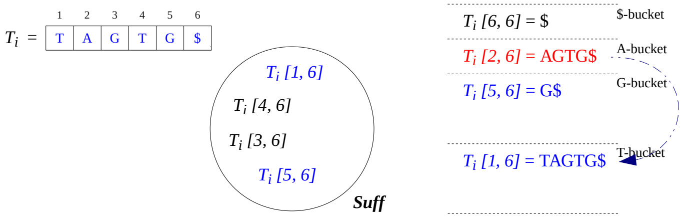


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- $\mathit{\Pi}$ starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from $\mathit{\Pi}$
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

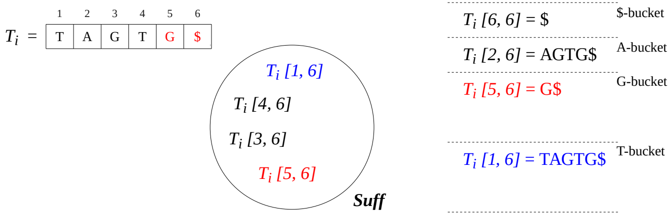


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



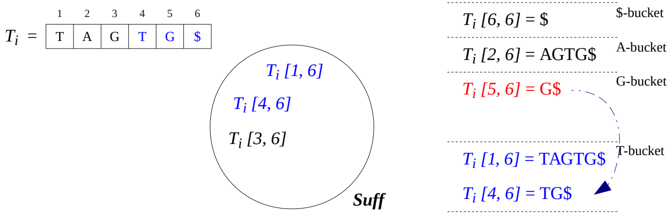
When we reach the β-bucket, as the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from Π find the first β-suffix $T_i[j-1, n_i]$ as the smallest suffix in Π, the β-bucket is read starting from the second element. As the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from Π

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



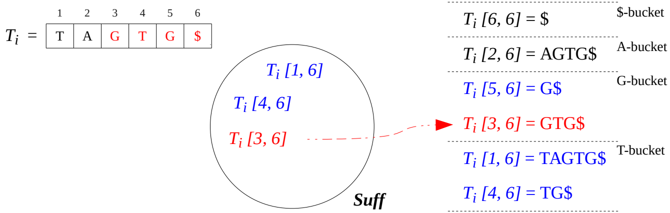
When we reach the β-bucket, as the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from Π find the first β-suffix $T_i[j-1, n_i]$ as the smallest suffix in Π, the β-bucket is read starting from the second element. As the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from Π

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- $\mathit{\Pi}$ starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from $\mathit{\Pi}$
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

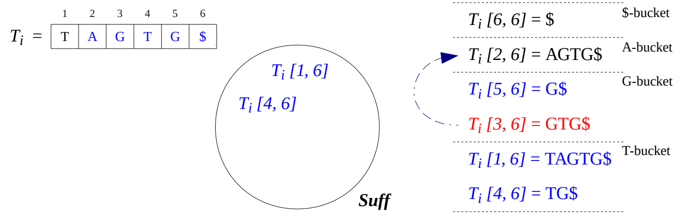


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



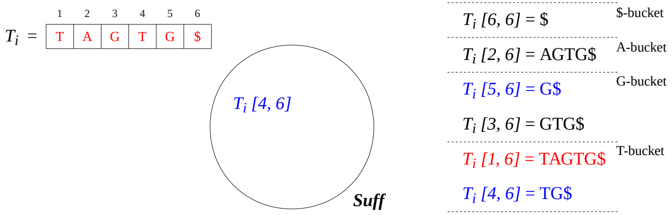
Note that if $\alpha > \beta$ the suffix $T_i[j-1, n_i]$ was already sorted

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- $\mathit{\Pi}$ starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from $\mathit{\Pi}$
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



Extra:

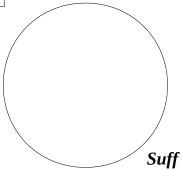
(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- $\mathit{\Pi}$ starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from $\mathit{\Pi}$
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

T_i =

1	2	3	4	5	6
T	A	G	T	G	$



$T_i\,[6, 6]$ = $ — $-bucket

$T_i\,[2, 6]$ = AGTG$ — A-bucket

$T_i\,[5, 6]$ = G$ — G-bucket

$T_i\,[3, 6]$ = GTG$

$T_i\,[1, 6]$ = TAGTG$ — T-bucket

$T_i\,[4, 6]$ = TG$

Extra:

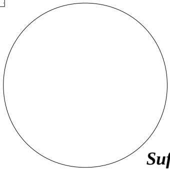
(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

T_i =

1	2	3	4	5	6
T	A	G	T	G	$



Suff

$-bucket
T_i [6, 6] = $

A-bucket
T_i [2, 6] = AGTG$

G-bucket
T_i [5, 6] = G$
T_i [3, 6] = GTG$

T-bucket
T_i [1, 6] = TAGTG$
T_i [4, 6] = TG$

Problem:

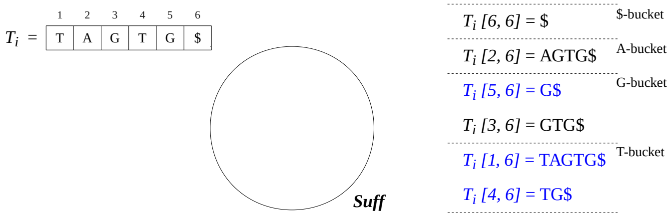
However, this approach is not efficient to sort a single string T_i, since it is always necessary to find the smallest suffix $T_i[j, n_i]$ in Π

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- Π starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from Π
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



Merging Sorting:

The smallest suffix is one of those nodes in the heap and can be determined efficiently