Estruturas de dados e algoritmos para buscas de padrões

Marco Alves de Alcantara

14 de Dezembro, 2022

Orientadora: Cristina Gomes Fernandes
IME USP

Roteiro

Roteiro

Contexto

- **Queremos buscar as ocorrências de palavras em um texto T**

Contexto

- **Queremos buscar as ocorrências de palavras em um texto T**
- **Estamos pensando em aplicações onde T é um texto bastante longo e fixo (imutável)**

Contexto

- Queremos buscar as ocorrências de palavras em um texto T
- Estamos pensando em aplicações onde T é um texto bastante longo e fixo (imutável)
- Algoritmos como KMP levam tempo $\mathcal{O}\left(|T|\right)$ por busca

Contexto

- Queremos buscar as ocorrências de palavras em um texto T
- Estamos pensando em aplicações onde T é um texto bastante longo e fixo (imutável)
- Algoritmos como KMP levam tempo $\mathcal{O}\left(|T|\right)$ por busca

- Queremos algo mais eficiente do que isso

Contexto

- **Queremos buscar as ocorrências de palavras em um texto T**
- **Estamos pensando em aplicações onde T é um texto bastante longo e fixo (imutável)**
- **Algoritmos como KMP levam tempo $\mathcal{O}\left(|T|\right)$ por busca**

- **Queremos algo mais eficiente do que isso**
  - Iremos pré-processar o texto T antes de realizar as buscas

Contexto

- **Queremos buscar as ocorrências de palavras em um texto T**
- **Estamos pensando em aplicações onde T é um texto bastante longo e fixo (imutável)**
- **Algoritmos como KMP levam tempo $\mathcal{O}\left(|T|\right)$ por busca**

- **Queremos algo mais eficiente do que isso**
  - Iremos pré-processar o texto T antes de realizar as buscas
    - » *o pré-processamento é relativamente lento (linear em $|T|$)*

Contexto

- **Queremos buscar as ocorrências de palavras em um texto T**
- **Estamos pensando em aplicações onde T é um texto bastante longo e fixo (imutável)**
- **Algoritmos como KMP levam tempo $\mathcal{O}\left(|T|\right)$ por busca**

- **Queremos algo mais eficiente do que isso**
  - Iremos pré-processar o texto T antes de realizar as buscas
    - » *o pré-processamento é relativamente lento (linear em $|T|$)*
    - » *mas é realizado uma única vez*
    - » *e permite agilizar **todas** as buscas por qualquer palavra P nesse texto*

Procedimento

1. **Pré-processamento:**
   1. Acrescentar um caractere especial \$ ao final de T

Procedimento

1. **Pré-processamento:**
   1. Acrescentar um caractere especial \$ ao final de T
   2. Construir o **vetor de sufixos** (VS) de T

Procedimento

1. **Pré-processamento:**
   1. Acrescentar um caractere especial \$ ao final de T
   2. Construir o **vetor de sufixos** (VS) de T
   3. Construir o **vetor LCP** de T a partir do VS

Procedimento

1. **Pré-processamento:**
   1. Acrescentar um caractere especial $ ao final de T
   2. Construir o **vetor de sufixos** (VS) de T
   3. Construir o **vetor LCP** de T a partir do VS
      - » ***e estender*** *o vetor LCP*

Procedimento

1. **Pré-processamento:**
   1. Acrescentar um caractere especial \$ ao final de T
   2. Construir o **vetor de sufixos** (VS) de T
   3. Construir o **vetor LCP** de T a partir do VS
      - » ***e estender*** *o vetor LCP*
2. **Em seguida, realizamos buscas binárias pelas palavras P no vetor de sufixos de T usando os LCPs estendidos**

Roteiro

Vetor de sufixos – definição

Uma estrutura de dados importante para as buscas é o vetor de sufixos:

Vetor de sufixos

- **Um vetor de sufixos é um vetor de índices $[0..|T|\text{-}1]$, onde cada índice representa um sufixo de T**
- **Se $VS[i] = k$, então $VS[i]$ representa $T[k..]$**
- **VS é ordenado lexicograficamente com base nos sufixos representados**

(https://en.wikipedia.org/wiki/Suffix_array)

Vetor de sufixos – exemplo

Exemplo de VS para o texto T = “abracadabra$”

$T[i..]$	i	VS$[i]$	T[VS$[i]..$]
abracadabra$	0	11	$
bracadabra$	1	10	a$
racadabra$	2	7	abra$
acadabra$	3	0	abracadabra$
cadabra$	4	3	acadabra$
adabra$	5	5	adabra$
dabra$	6	8	bra$
abra$	7	1	bracadabra$
bra$	8	4	cadabra$
ra$	9	6	dabra$
a$	10	9	ra$
$	11	2	racadabra$

Vetor de sufixos – ideia da construção

- **Utiliza-se um algoritmo recursivo de divisão e conquista**

Vetor de sufixos – ideia da construção

- **Utiliza-se um algoritmo recursivo de divisão e conquista**
  - O texto T é dividido em 3, alternando qual sufixo fica em qual parte

Vetor de sufixos – ideia da construção

- **Utiliza-se um algoritmo recursivo de divisão e conquista**
  - O texto T é dividido em 3, alternando qual sufixo fica em qual parte
  - É construído um texto auxiliar T_{12}, contendo os 3 primeiros caracteres de cada sufixo mod 1 e mod 2

Vetor de sufixos – ideia da construção

- **Utiliza-se um algoritmo recursivo de divisão e conquista**
  - O texto T é dividido em 3, alternando qual sufixo fica em qual parte
  - É construído um texto auxiliar T_{12}, contendo os 3 primeiros caracteres de cada sufixo mod 1 e mod 2
  - Cada tripla é ordenada com o RadixSort, e corresponde a um caractere num alfabeto maior

Vetor de sufixos – ideia da construção

- **Utiliza-se um algoritmo recursivo de divisão e conquista**
  - O texto T é dividido em 3, alternando qual sufixo fica em qual parte
  - É construído um texto auxiliar T_{12}, contendo os 3 primeiros caracteres de cada sufixo mod 1 e mod 2
  - Cada tripla é ordenada com o RadixSort, e corresponde a um caractere num alfabeto maior
  - Caso hajam triplas repetidas, é necessário construir o vetor de sufixos de T_{12} recursivamente, caso contrário, o RadixSort já nos dá esse vetor de sufixos

Vetor de sufixos – ideia da construção

- **Utiliza-se um algoritmo recursivo de divisão e conquista**
  - O texto T é dividido em 3, alternando qual sufixo fica em qual parte
  - É construído um texto auxiliar T_{12}, contendo os 3 primeiros caracteres de cada sufixo mod 1 e mod 2
  - Cada tripla é ordenada com o RadixSort, e corresponde a um caractere num alfabeto maior
  - Caso hajam triplas repetidas, é necessário construir o vetor de sufixos de T_{12} recursivamente, caso contrário, o RadixSort já nos dá esse vetor de sufixos
- **Depois do processo de divisão, fazemos a parte da conquista...**

Vetor de sufixos – ideia da construção – parte 2

- **Usamos VS $\left(T_1\right)$ para ordenar os sufixos de T_0 (obter VS $\left(T_0\right)$)**

Vetor de sufixos – ideia da construção – parte 2

- **Usamos VS $\left(T_1\right)$ para ordenar os sufixos de T_0 (obter VS $\left(T_0\right)$)**
- **Construímos certos pares e triplas formados por 1 (ou 2) caracteres de T_0, T_1 ou T_2 e com 1 rank de VS $\left(T_{12}\right)$**

Vetor de sufixos – ideia da construção – parte 2

- **Usamos VS (T_1) para ordenar os sufixos de T_0 (obter VS (T_0))**
- **Construímos certos pares e triplas formados por 1 (ou 2) caracteres de T_0, T_1 ou T_2 e com 1 rank de VS (T_{12})**
- **Fazemos um procedimento semelhante ao do algoritmo MergeSort para inicializar VS (T), comparando pares ou triplas elemento-por-elemento**

Vetor de sufixos – ideia da construção – parte 2

- **Usamos VS (T_1) para ordenar os sufixos de T_0 (obter VS (T_0))**
- **Construímos certos pares e triplas formados por 1 (ou 2) caracteres de T_0, T_1 ou T_2 e com 1 rank de VS (T_{12})**
- **Fazemos um procedimento semelhante ao do algoritmo MergeSort para inicializar VS (T), comparando pares ou triplas elemento-por-elemento**
- **Por fim, obtém-se VS (T) em tempo linear em $|T|$.**

(https://gist.github.com/markormesher/59b990fba09972b4737e7ed66912e044)
(http://www.cs.cmu.edu/ guyb/realworld/papersS04/KaSa03.pdf)

Roteiro

Vetor LCP – definição

Além do vetor de sufixos, para que seja possível fazer a busca eficientemente, é necessário também o vetor LCP do texto T:

Vetor LCP

- **O vetor LCP é um vetor de tamanho $|T|$ - 1, indexado a partir de 1, que mede o comprimento do prefixo comum mais longo entre dois sufixos de T**
- **Se LCP$[i] = k$, então os sufixos de índices VS$[i$ - 1$]$ e VS$[i]$ tem um prefixo comum mais longo de comprimento k**

(https://en.wikipedia.org/wiki/LCP_array)

Vetor LCP – exemplo

Vetor LCP para o texto T = “abracadabra$”

i	VS[i]	LCP[i]	T[VS[i]..]
0	11		$
1	10	0	a$
2	7	1	**a**bra$
3	0	4	**abra**cadabra$
4	3	1	**a**cadabra$
5	5	1	**a**dabra$
6	8	0	bra$
7	1	3	**bra**cadabra$
8	4	0	cadabra$
9	6	0	dabra$
10	9	0	ra$
11	2	2	**ra**cadabra$

Vetor LCP – algoritmo de construção

O vetor LCP pode ser construído a partir do VS com o Algoritmo de Kasai

Vetor LCP – algoritmo de construção

O vetor LCP pode ser construído a partir do VS com o Algoritmo de Kasai

```
1  i ← 0
2  enquanto i < |T| faça
3  |   rank[VS[i]] ← i
4  |   i ← i + 1
5  h, i ← 0
6  enquanto i < |T| faça
7  |   se rank[i] > 0 então
8  |   |   k ← VS[rank[i] - 1]
9  |   |   enquanto T[i + h] = T[k + h] faça
10 |   |   |   h ← h + 1
11 |   |   LCP[rank[i]] ← h
12 |   |   se h > 0 então
13 |   |   |   h ← h - 1
14 |   i ← i + 1
15 retorna LCP
```

Vetor LCP – extensão

- O vetor LCP por si só não ajuda muito nas buscas, já que ele só nos dá informações sobre dois sufixos consecutivos em VS

Vetor LCP – extensão

- **O vetor LCP por si só não ajuda muito nas buscas, já que ele só nos dá informações sobre dois sufixos consecutivos em VS**
  - É necessário **estender** o LCP para nos dar informações sobre um **intervalo** de sufixos

Vetor LCP – extensão

- **O vetor LCP por si só não ajuda muito nas buscas, já que ele só nos dá informações sobre dois sufixos consecutivos em VS**
  - É necessário **estender** o LCP para nos dar informações sobre um **intervalo** de sufixos
  - Mais precisamente, sobre os $\mathcal{O}\left(|T|\right)$ intervalos possíveis de aparecerem durante uma busca binária em VS

Vetor LCP – extensão

- **O vetor LCP por si só não ajuda muito nas buscas, já que ele só nos dá informações sobre dois sufixos consecutivos em VS**
  - É necessário **estender** o LCP para nos dar informações sobre um **intervalo** de sufixos
  - Mais precisamente, sobre os $\mathcal{O}\left(|T|\right)$ intervalos possíveis de aparecerem durante uma busca binária em VS
    - » *por isso, também iremos **pré-processar o vetor LCP**, simulando todos os caminhos possíveis em uma busca binária, e guardar as informações obtidas sobre os LCPs desses intervalos*

Vetor LCP – extensão

- **O vetor LCP por si só não ajuda muito nas buscas, já que ele só nos dá informações sobre dois sufixos consecutivos em VS**
  - É necessário **estender** o LCP para nos dar informações sobre um **intervalo** de sufixos
  - Mais precisamente, sobre os $\mathcal{O}\left(|T|\right)$ intervalos possíveis de aparecerem durante uma busca binária em VS
    - » *por isso, também iremos **pré-processar o vetor LCP**, simulando todos os caminhos possíveis em uma busca binária, e guardar as informações obtidas sobre os LCPs desses intervalos*
    - » *durante as buscas, essas informações poderão ser obtidas em tempo $\mathcal{O}(1)$, já que esse pré-processamento só depende de T*

Roteiro

Resumo da busca

- **Para buscar as ocorrências de uma palavra P, na verdade são necessárias duas buscas binárias**

Resumo da busca

- **Para buscar as ocorrências de uma palavra P, na verdade são necessárias duas buscas binárias**
  - Primeiro, busca-se o **predecessor** de P no vetor de sufixos, ou seja, o índice do maior sufixo de T que é lexicograficamente menor que P

Resumo da busca

- **Para buscar as ocorrências de uma palavra P, na verdade são necessárias duas buscas binárias**
  - Primeiro, busca-se o **predecessor** de P no vetor de sufixos, ou seja, o índice do maior sufixo de T que é lexicograficamente menor que P
  - O predecessor serve como um limitante inferior para um intervalo que contenha as ocorrências de P

Resumo da busca

- **Para buscar as ocorrências de uma palavra P, na verdade são necessárias duas buscas binárias**
  - Primeiro, busca-se o **predecessor** de P no vetor de sufixos, ou seja, o índice do maior sufixo de T que é lexicograficamente menor que P
  - O predecessor serve como um limitante inferior para um intervalo que contenha as ocorrências de P
  - Para o limitante superior, precisamos de um índice a partir do qual P não é mais um prefixo dos sufixos de VS

Resumo da busca

- **Para buscar as ocorrências de uma palavra P, na verdade são necessárias duas buscas binárias**
  - Primeiro, busca-se o **predecessor** de P no vetor de sufixos, ou seja, o índice do maior sufixo de T que é lexicograficamente menor que P
  - O predecessor serve como um limitante inferior para um intervalo que contenha as ocorrências de P
  - Para o limitante superior, precisamos de um índice a partir do qual P não é mais um prefixo dos sufixos de VS
    - » *podemos criar uma nova palavra Q, cujo predecessor é o limitante superior do intervalo*

Resumo da busca

- **Para buscar as ocorrências de uma palavra P, na verdade são necessárias duas buscas binárias**
  - Primeiro, busca-se o **predecessor** de P no vetor de sufixos, ou seja, o índice do maior sufixo de T que é lexicograficamente menor que P
  - O predecessor serve como um limitante inferior para um intervalo que contenha as ocorrências de P
  - Para o limitante superior, precisamos de um índice a partir do qual P não é mais um prefixo dos sufixos de VS
    - » *podemos criar uma nova palavra Q, cujo predecessor é o limitante superior do intervalo*
    - » *Q pode ser obtida, por exemplo, incrementando o último caractere de P*

Resumo da busca – exemplo

Suponha, ainda para T = “abracadabra$”, que estejamos procurando as ocorrências de P = “abra”.

Resumo da busca – exemplo

Suponha, ainda para T = "abracadabra$", que estejamos procurando as ocorrências de P = "abra".

$T[i..]$	i	VS[i]	T[VS[i]..]
abracadabra$	0	11	$
bracadabra$	1	10	a$
racadabra$	2	**7**	abra$
acadabra$	3	**0**	abracadabra$
cadabra$	4	3	acadabra$
adabra$	5	5	adabra$
dabra$	6	8	bra$
abra$	7	1	bracadabra$
bra$	8	4	cadabra$
ra$	9	6	dabra$
a$	10	9	ra$
$	11	2	racadabra$

Resumo da busca – exemplo

Suponha, ainda para T = “abracadabra$”, que estejamos procurando as ocorrências de P = “abra”.

$T[i..]$	i	VS[i]	T[VS[i]..]	
abracadabra$	0	11	$	
bracadabra$	**1**	10	**a$**	⟵ pred(P = “**ab**ra”) = **1**
racadabra$	2	**7**	abra$	
acadabra$	3	**0**	abracadabra$	
cadabra$	4	3	acadabra$	
adabra$	5	5	adabra$	
dabra$	6	8	bra$	
abra$	7	1	bracadabra$	
bra$	8	4	cadabra$	
ra$	9	6	dabra$	
a$	10	9	ra$	
$	11	2	racadabra$	

Resumo da busca – exemplo

Suponha, ainda para T = "abracadabra$", que estejamos procurando as ocorrências de P = "abra".

$T[i..]$	i	VS[i]	T[VS[i]..]	
abracadabra$	0	11	$	
bracadabra$	**1**	10	**a$**	⟵ pred(P = "**ab**ra") = **1**
racadabra$	2	**7**	abra$	
acadabra$	**3**	**0**	**abra**cadabra$	⟵ pred(Q = "**abrb**") = **3**
cadabra$	4	3	acadabra$	
adabra$	5	5	adabra$	
dabra$	6	8	bra$	
abra$	7	1	bracadabra$	
bra$	8	4	cadabra$	
ra$	9	6	dabra$	
a$	10	9	ra$	
$	11	2	racadabra$	

Resumo da busca – exemplo – conclusões

- P **ocorre em** T **se, e somente se,** $\textbf{pred}\left(P\right) \neq \textbf{pred}\left(Q\right)$

Resumo da busca – exemplo – conclusões

- P **ocorre em** T **se, e somente se, pred** $\left(P\right) \neq$ **pred** $\left(Q\right)$
- **O número de ocorrências é pred** $\left(Q\right)$ **- pred** $\left(P\right)$

Resumo da busca – exemplo – conclusões

- P **ocorre em** T **se, e somente se,** **pred** $\left(P\right) \neq$ **pred** $\left(Q\right)$
- **O número de ocorrências é** **pred** $\left(Q\right)$ **-** **pred** $\left(P\right)$
- **As posições onde** P **ocorre em** T **são os sufixos em** **VS**[**pred** $\left(P\right)$ + 1..**pred** $\left(Q\right)$]

Resumo da busca – exemplo – conclusões

- P **ocorre em** T **se, e somente se,** **pred** (P) $\neq$ **pred** (Q)
- **O número de ocorrências é** **pred** (Q) **-** **pred** (P)
- **As posições onde** P **ocorre em** T **são os sufixos em** **VS**[**pred** (P) + 1..**pred** (Q)]

T = "**a**bra cad**a**bra\$"
↑**0** ↑**7**

Busca – visão mais detalhada

- Na busca binária para encontrar o predecessor, além dos extremos L e R da busca, também guardamos índices l e r com o número de caracteres casados entre P e os sufixos dos extremos L e R

Busca – visão mais detalhada

- **Na busca binária para encontrar o predecessor, além dos extremos L e R da busca, também guardamos índices l e r com o número de caracteres casados entre P e os sufixos dos extremos L e R**
  - A finalidade desses índices l e r, bem como do pré-processamento realizado no vetor LCP, é minimizar o número de comparações diretas entre caracteres de P e T, a fim de agilizar a busca

Busca – visão mais detalhada – caso 0

Antes de fazer a busca binária, precisamos verificar se a palavra é maior que o último sufixo do VS. Se for, o predecessor já é o último sufixo.

Busca – visão mais detalhada – caso 0

Antes de fazer a busca binária, precisamos verificar se a palavra é maior que o último sufixo do VS. Se for, o predecessor já é o último sufixo.

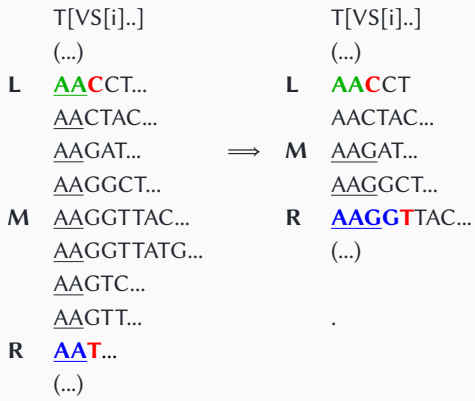
i	VS[i]	T[VS[i]..]	
0	11	$	
1	10	a$	
2	7	abra$	
3	0	abracadabra$	
4	3	acadabra$	
5	5	adabra$	
6	8	bra$	
7	1	bracadabra$	
8	4	cadabra$	
9	6	dabra$	
10	9	ra$	
11	**2**	**rac**adabra$	⟵ pred(P = "**rat**") = **2**

Busca – visão mais detalhada – caso 1

Se $l = r$, então é necessário comparar caracteres de P e T para decidir como prosseguir com a busca (exemplo para P = "AAGG")

Busca – visão mais detalhada – caso 1

Se **l** = **r**, então é necessário comparar caracteres de P e T para decidir como prosseguir com a busca (exemplo para P = "AAGG")



Busca – visão mais detalhada – caso 2

- **Se l $>$ r, há 3 sub-casos possíveis dependendo da relação entre l e o LCP do intervalo $[L, M]$**

Busca – visão mais detalhada – caso 2

- **Se l $>$ r, há 3 sub-casos possíveis dependendo da relação entre l e o LCP do intervalo $[L, M]$**
  - O caso 3 (**l** $<$ **r**) não será analisado aqui por ser simétrico ao caso 2 baseado na relação entre **r** e o LCP do intervalo $[M, R]$

Busca – visão mais detalhada – caso 2

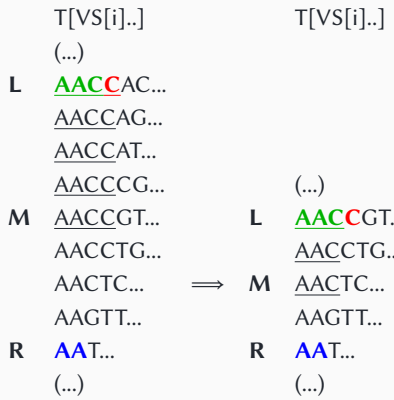
- **Se $\mathbf{l} > \mathbf{r}$, há 3 sub-casos possíveis dependendo da relação entre $\mathbf{l}$ e o LCP do intervalo $[L, M]$**
  - O caso 3 ($\mathbf{l} < \mathbf{r}$) não será analisado aqui por ser simétrico ao caso 2 baseado na relação entre $\mathbf{r}$ e o LCP do intervalo $[M, R]$
- **Iremos agora ver os 3 sub-casos do caso 2**

Busca – visão mais detalhada – sub-caso 2.1

Se $\mathbf{l} > \mathbf{r}$ e $\mathbf{l} < \underline{\text{LCP}[L, M]}$, é possível avançar a busca sem comparar caracteres (exemplo para P = "AACT")

Busca – visão mais detalhada – sub-caso 2.1

Se **l** $>$ **r** e **l** $<$ LCP$[L, M]$, é possível avançar a busca sem comparar caracteres (exemplo para P = "AACT")

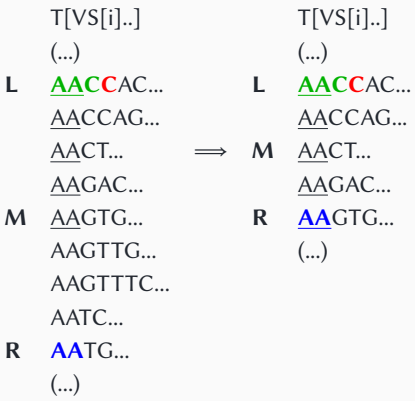


Busca – visão mais detalhada – sub-caso 2.2

Se $\mathbf{l} > \mathbf{r}$ e $\mathbf{l} > \underline{\text{LCP}[L, M]}$, é possível avançar a busca sem comparar caracteres (exemplo para P = "AACG")

Busca – visão mais detalhada – sub-caso 2.2

Se $\mathbf{l} > \mathbf{r}$ e $\mathbf{l} > \underline{\text{LCP}[L, M]}$, é possível avançar a busca sem comparar caracteres (exemplo para P = "AACG")

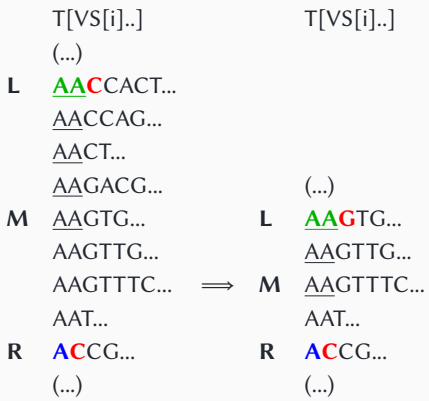


Busca – visão mais detalhada – sub-caso 2.3

Se $\mathbf{l} > \mathbf{r}$ e $\mathbf{l} = \underline{\text{LCP}[L, M]}$, é necessário comparar caracteres diretamente para avançar (exemplo para P = "AAT")

Busca – visão mais detalhada – sub-caso 2.3

Se $\mathbf{l} > \mathbf{r}$ e $\mathbf{l} = \underline{\text{LCP}[L, M]}$, é necessário comparar caracteres diretamente para avançar (exemplo para P = "AAT")



Busca – complexidade

- **Por meio de uma análise amortizada, é possível concluir que a busca pelo predecessor de uma palavra P leva tempo $\mathcal{O}\left(|P| + \log\left(|T|\right)\right)$**
- **Portanto, contar o número de ocorrências de P também leva tempo $\mathcal{O}\left(|P| + \log\left(|T|\right)\right)$**

Busca – complexidade

- **Por meio de uma análise amortizada, é possível concluir que a busca pelo predecessor de uma palavra P leva tempo $\mathcal{O}\left(|P| + \log\left(|T|\right)\right)$**
- **Portanto, contar o número de ocorrências de P também leva tempo $\mathcal{O}\left(|P| + \log\left(|T|\right)\right)$**
- **Encontrar as ocorrências leva tempo $\mathcal{O}\left(|P| + \log\left(|T|\right) + x\right)$, onde x é o número de ocorrências de P**

Roteiro

Árvore de sufixos – introdução

- **Uma alternativa a realizar as buscas no vetor de sufixos é buscar usando a árvore de sufixos de T**

Árvore de sufixos – introdução

- **Uma alternativa a realizar as buscas no vetor de sufixos é buscar usando a árvore de sufixos de T**
  - Uma desvantagem de usar árvore de sufixos é que elas gastam mais espaço

Árvore de sufixos – introdução

- **Uma alternativa a realizar as buscas no vetor de sufixos é buscar usando a árvore de sufixos de T**
  - Uma desvantagem de usar árvore de sufixos é que elas gastam mais espaço
  - Uma vantagem é que é feita uma única busca em vez de duas, e, além disso, ela é feita em tempo $\mathcal{O}\left(|P|\right)$

Árvore de sufixos – introdução

- **Uma alternativa a realizar as buscas no vetor de sufixos é buscar usando a árvore de sufixos de T**
  - Uma desvantagem de usar árvore de sufixos é que elas gastam mais espaço
  - Uma vantagem é que é feita uma única busca em vez de duas, e, além disso, ela é feita em tempo $\mathcal{O}\left(|P|\right)$
- **A árvore de sufixos pode ser obtida a partir do vetor de sufixos e LCP em tempo linear em $|T|$ por meio de uma árvore cartesiana**

Árvore de sufixos – introdução

- **Uma alternativa a realizar as buscas no vetor de sufixos é buscar usando a árvore de sufixos de T**
  - Uma desvantagem de usar árvore de sufixos é que elas gastam mais espaço
  - Uma vantagem é que é feita uma única busca em vez de duas, e, além disso, ela é feita em tempo $\mathcal{O}\left(|P|\right)$
- **A árvore de sufixos pode ser obtida a partir do vetor de sufixos e LCP em tempo linear em $|T|$ por meio de uma árvore cartesiana**
- **Alternativamente, pode ser construída direto de T, sem o vetor de sufixos, por meio do Algoritmo de Ukkonen**

Árvore de sufixos – definição

Árvore de sufixos

- **Uma árvore de sufixos de um texto T é uma árvore tal que:**
  - Cada nó pode ter até $|\Sigma|$ filhos, onde Σ é o alfabeto do texto;

Árvore de sufixos – definição

Árvore de sufixos

- **Uma árvore de sufixos de um texto T é uma árvore tal que:**
  - Cada nó pode ter até $|\Sigma|$ filhos, onde Σ é o alfabeto do texto;
  - Cada aresta entre um nó pai e filho contém um rótulo, que é um intervalo do texto T;

Árvore de sufixos – definição

Árvore de sufixos

- **Uma árvore de sufixos de um texto T é uma árvore tal que:**
  - Cada nó pode ter até $|\Sigma|$ filhos, onde Σ é o alfabeto do texto;
  - Cada aresta entre um nó pai e filho contém um rótulo, que é um intervalo do texto T;
  - Cada nó filho é indexado a partir do nó pai pelo primeiro caractere do rótulo da aresta entre eles;

Árvore de sufixos – definição

Árvore de sufixos

- **Uma árvore de sufixos de um texto T é uma árvore tal que:**
  - Cada nó pode ter até $|\Sigma|$ filhos, onde Σ é o alfabeto do texto;
  - Cada aresta entre um nó pai e filho contém um rótulo, que é um intervalo do texto T;
  - Cada nó filho é indexado a partir do nó pai pelo primeiro caractere do rótulo da aresta entre eles;
  - Os rótulo são o mais longo possível, isso é, se $T \neq$ "\$", não há nós com exatamente um filho (a árvore é comprimida);

Árvore de sufixos – definição

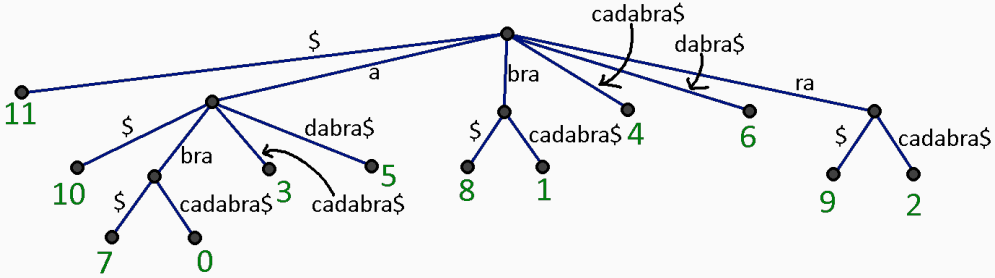
Árvore de sufixos

- **Uma árvore de sufixos de um texto T é uma árvore tal que:**
  - Cada nó pode ter até $|\Sigma|$ filhos, onde Σ é o alfabeto do texto;
  - Cada aresta entre um nó pai e filho contém um rótulo, que é um intervalo do texto T;
  - Cada nó filho é indexado a partir do nó pai pelo primeiro caractere do rótulo da aresta entre eles;
  - Os rótulo são o mais longo possível, isso é, se $T \neq$ "\$", não há nós com exatamente um filho (a árvore é comprimida);
  - Cada folha contém o índice de um sufixo de T, e todos os sufixos de T são representados por uma folha.

(https://en.wikipedia.org/wiki/Suffix_tree)

Árvore de sufixos – exemplo

Árvore de sufixos para T = “abracadabra$”



Os índices das folhas aparecem na mesma ordem que no vetor de sufixos

Árvore de sufixos – busca

- **A busca consiste em simplesmente encontrar o nó menos fundo n cujo caminho até a raiz contém P**

Árvore de sufixos – busca

- **A busca consiste em simplesmente encontrar o nó menos fundo n cujo caminho até a raiz contém P**
  - Se n não existir, então P não ocorre em T

Árvore de sufixos – busca

- **A busca consiste em simplesmente encontrar o nó menos fundo n cujo caminho até a raiz contém P**
  - Se n não existir, então P não ocorre em T
  - Se existir, então o número de ocorrências de P é igual ao número de folhas abaixo de n, e as ocorrências são os índices dessas folhas

Árvore de sufixos – busca

- **A busca consiste em simplesmente encontrar o nó menos fundo n cujo caminho até a raiz contém P**
  - Se n não existir, então P não ocorre em T
  - Se existir, então o número de ocorrências de P é igual ao número de folhas abaixo de n, e as ocorrências são os índices dessas folhas
- **Para encontrar n, apenas tente percorrer a árvore a partir do nó atual procurando o nó indexado pelo próximo caractere de P, e verifique se P casa com o rótulo da aresta entre eles**

Roteiro

Resultados – Complexidade assintótica de tempo

Complexidade	KMP	VS e LCP	Árv. suf.
Pré-processamento único	não tem	$\mathcal{O}\left(\lvert T\rvert\right)$	$\mathcal{O}\left(\lvert T\rvert\right)$
Pré-processamento por busca	$\mathcal{O}\left(\lvert P\rvert\right)$	não tem	não tem
Contar as ocorrências	$\mathcal{O}\left(\lvert T\rvert\right)$	$\mathcal{O}\left(\lvert P\rvert + \log\left(\lvert T\rvert\right)\right)$	$\mathcal{O}\left(\lvert P\rvert\right)$
Encontrar todas as ocorrências	$\mathcal{O}\left(\lvert T\rvert\right)$	$\mathcal{O}\left(\lvert P\rvert + \log\left(\lvert T\rvert\right) + x\right)$	$\mathcal{O}\left(\lvert P\rvert + x\right)$

Tabela 1: *Comparação da complexidade de tempo de diferentes algoritmos de busca, onde x denota o número de ocorrências de P em T.*

Buscas de Padrões em Textos

Perguntas