Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Definições do Projeto Final

da disciplina Projeto e Análise de Algoritmos

Alexandre M. Kaihara 18/0029690
Felipe X. B. Silva 18/0016326
Gabriel R. D. Macêdo 15/0126808
Guilherme C. Suzuki 18/0032518
Italo Franklin 13/0115428
Jaqueline G. Coelho 15/0131283
Pedro L. C. Rocha 18/0054635

1 Ambiente e ferramentas

A seguir serão descritas as ferramentas necessárias para a execução do projeto, além do ambiente no qual o algoritmo irá operar.

- Linguagens: Python 3 para webcrawler e pré-processamento; Javascript (Node.js), versão 14.18.1, para busca e servidor;
- *Webcrawler*: Script em Python utilizando a biblioteca BeautifulSoup.
- Servidor: NuxtJS, API de servidor *Web*, *MIT License*;

Note que todas ferramentas citadas podem ser utilizadas: A licensa do Apache HTTPD é do tipo GPLv3 e a do Scrapy é do tipo BSD.

2 Algoritmos utilizados

O algoritmo, em sua totalidade, consiste na junção de 3 etapas, executadas na respectiva ordem: indexação, busca e ranqueamento. Por isso, serão descritos os algoritmos a serem utilizados em cara etapa e, também, serão analisadas as complexidades de tempo de cada um deles.

2.1 Indexação

A abordagem usada é baseada na indexação inversa. Como será feita a busca de palavras em documentos, então os documentos serão indexados por suas palavras, retornando um subconjunto de documentos (com possíveis intersecções entre esses subconjuntos), em uma estrutura de dados de consulta rápida.

Algorithm 1 Algoritmo de indexação inversa simples

```
Docs := conjunto de páginas
function INDEX(Docs)
    Index ← ∅
    id ← 0
    for D ∈ Docs do
        for s ∈ D.palavras do
            if Index[s] = ∅ then
                Index[s].id ← id
                id ← id + 1
            end if
            Index[s] ← Index[s] ⊔ {D.id}
        end for
    end for
    return Index
end function
```

No algoritmo 1, $Index$ refere-se a uma estrutura de dados que indexa conjuntos de documentos dada uma palavra como índice, operação representada por $Index[string]$. $Index.id$ é um identificador que será usado em outra etapas. $\sqcup$ representa a operação de união disjunta entre conjuntos. $Docs$ é uma lista contendo todos documentos extraídos pelo $WebCrawler$ e cada documento D possui uma forma de acessar todas as suas palavras na lista $D.palavras$ e um id único $D.id$.

Para o algoritmo 1, o número de passos pode ser dado por:

$$C_{indexing} = \sum_{i=1}^{Docs.n} \sum_{j=1}^{Docs[i].n} C_{append} \tag{1}$$

Sendo C_{append} um custo da operação $Index[s] \leftarrow Index[s] \sqcup \{D.id\}$, no momento em que $Index$ possui tamanho n. Note que, a seguinte inequação é satisfeita:

$$\begin{aligned} \sum_{i=1}^{Docs.n} \sum_{j=1}^{Docs[i].n} C_{append} &\leq Docs.n \cdot \max_{D \in Docs} (D.n \cdot C_{append}^{\max}) \\ &= 1 \cdot (Docs.n \cdot C_{append}^{\max} \cdot \max_{D \in Docs} D.n) \end{aligned}$$

Por isso, temos que, para o pior caso, $C_{indexing} \in O(Docs.n \cdot C_{append}^{\max} \cdot \max D.n)$, para a constante 1. Ou seja, o limitante superior é o produto entre número de documentos analisados, o maior número de palavras que um documento pode apresentar e o custo máximo da operação C_{append}. As etapas posteriores requerem que cada lista contida em $Index$ esteja ordenada, o que pode ser feito ao usar uma árvore binária para permitir inserções de custo $\Theta(\log n)$. Dessa forma, a complexidade do algoritmo no pior caso deve ser da ordem de $O(Docs.n \cdot \log_2 N \cdot \max D.n)$, em que N é o tamanho da maior árvore contida em $Index$ após o final do algoritmo.

2.2 Pré-processamento das páginas

Para o algoritmo 3, será necessário processar cada documento conforme o algoritmo 2, o qual cria uma lista de identificadores para cada palavra do documento, consultando a estrutura $Index$ construída no algoritmo 1.

Algorithm 2 Algoritmo de pré-processamento das páginas

Docs := conjunto de páginas
function MAKE_ID_LIST(Docs)
    for $D \in Docs$ **do**
        $D.L \leftarrow \emptyset$ ▷ Lista de ids para o documento d
        for $s \in D$ **do**
            $D.L.append(Index[s])$ ▷ Adiciona id na lista
        end for
    end for
end function

O algoritmo possui complexidade semelhante ao do algoritmo 1, já que cada palavra de cada documento é explorada da mesma maneira. A diferença é que o custo da operação *append* é constante, mas ainda é preciso considera o custo da consulta $Index[s]$. Logo, a complexidade do pior caso é $O(Docs.n \cdot \log_2 N \cdot \max D.n)$, em que N é o tamanho da maior lista contida em $Index$.

2.3 Pesquisa

2.3.1 Palavras simples

A busca de uma palavra depende da estrutura de dados usada na etapa de indexação. Para evitar fazer pesquisas com strings, o que pode ser lento, podemos utilizar uma função de *hashing* que converta cada palavra para um número inteiro em um intervalo pré-definido da seguinte forma:

$$\begin{aligned} hash(s) &= (s[0] + s[1] \cdot p + s[2] \cdot p^2 + ... + s[m-1] \cdot p^{m-1}) \mod k \\ &= \sum_{i=0}^{m-1} s[i] \cdot p^i \mod k \end{aligned}$$

Em que p é um número primo próximo da quantidade de caracteres possíveis, m é o tamanho da string e k é o tamanho do intervalo. Esta abordagem, na forma mais básica, permite colisões entre palavras. Dizemos que uma colisão entre duas palavras A e B ocorre se $hash(A) = hash(B)$, e a chance disso ocorrer, se A e B são strings aleatórias, é dada por $\frac{1}{m}$.

Por padrão, a estrutura de objetos existente em Javascript utiliza tabelas de hash para fazer buscas eficientemente. Da forma que é utilizada, a busca nessa tabela possui complexidade $O(\textit{número máximo de palavras com o mesmo hash})$. Embora palavras possam ser escolhidas de forma que todas colidam entre si, esse não vai ser o caso no buscador desenvolvido. Sendo assim, na prática, as colisões serão raras, e a complexidade de busca do hash é então próxima de $O(1)$.

2.3.2 Pesquisa de palavras compostas

A busca de palavras compostas é feita por meio da busca da primeira palavra simples que compõe a entrada, o que fornece uma lista de documentos contendo tal palavra simples. Depois, cada documento é examinado a fim de determinar se a palavra composta existe no documento, utilizando uma versão adaptada do algoritmo de Knuth–Morris–Pratt.

Algorithm 3 Algoritmo de Knuth–Morris–Pratt

S := Lista de ids de strings a ser buscada
Text := Lista de ids de strings
function KMP(S,Text)
    $Positions \leftarrow \emptyset$
    $matches \leftarrow 0$
    $\Pi \leftarrow Preprocess(S)$
    for $i \in \{1, 2, ..., Text.n\}$ **do**
        while $matches > 0 \land S[matches + 1] \neq Text[i]$ **do**
            $matches \leftarrow \Pi[matches]$
        end while
        if $S[matches + 1] = Text[i]$ **then**
            $matches \leftarrow matches + 1$
        end if
        if matches = S.n **then**
            $Positions \leftarrow Positions \cup \{i - S.n + 1\}$
            $matches \leftarrow \Pi[matches]$
        end if
    end for
    return $Positions$
end function

O algoritmo 3 opera em complexidade $\Theta(Text.n + C_{preprocess})$, em que $C_{preprocess}$ é o custo de pré-processamento de Π, visto que o algoritmo explora cada palavra contida em $Text$ apenas uma vez.

Algorithm 4 Algoritmo de pré-processamento do KMP

```
S := Lista de ids de strings a ser processada
function PREPROCESS(S)
    Π ← {0}                                              ▷ Π[1] = 0
    length ← 0
    for matches ∈ {2, 3, ..., S.n} do
        while length > 0 ∧ S[length + 1] ≠ S[matches] do
            length ← Π[length]
        end while
        if S[length + 1] = S[matches] then
            length ← length + 1
        end if
        Π[matches] = length
    end for
end function
```

A complexidade dessa etapa (algoritmo 4) é da ordem de $\Theta(S.n)$, pois cada palavra contida em S é processada uma única vez em tempo constante. Portanto, a complexidade total do algoritmo (3,4) é da ordem de $\Theta(Text.n + S.n)$.

Note que, ao invés de procurar por encaixes a nível de caracteres, são procurados encaixes a nível de palavras (identificadas por números inteiros). Por isso, é necessário um modo de converter cada palavra para seu id correspondente, o que pode ser feito ao consultar a estrutura $Index$. Nesse caso, a complexidade do pior caso ao levar em conta o cálculo de cada id é $O(S.n \cdot C_{Index} + Text.n + S.n)$, em que C_{Index} é o custo da consulta do id de uma palavra à estrutura $Index$.

2.4 Operações entre conjuntos

Para pesquisas que usam expressões que descrevem um conjunto usando os operadores $\cup$ (União), $\cap$ (Intersecção) e $\setminus$ (Diferença). Cada uma delas pode ser realizada conforme os seguintes pseudocódigos.

2.4.1 União

Para suportar a operação OR entre dois termos de pesquisa, é suficiente fazer a união dos conjuntos de documentos de cada termo. Uma forma de fazer essa união está descrita no algoritmo 5. O algoritmo pressupõe que os conjuntos iniciais, A e B, estão ordenados, e adiciona sempre o menor dos elementos restantes para o conjunto de retorno, C. Quando os conjuntos A e B possuem um elemento em comum, apenas um deles é adicionado. Como o algoritmo consiste, essencialmente, na iteração nos conjuntos de entrada, sua complexidade é sempre $\Theta(A.n + B.n)$, onde $A.n$ é o tamanho do conjunto A e $B.n$ é o tamanho do conjunto B.

Algorithm 5 Algoritmo de União entre conjuntos

Require: $A[i] < A[i+1], \forall i \in \{1, 2, ..., A.n\}$
Require: $B[j] < B[j+1], \forall j \in \{1, 2, ..., B.n\}$
Ensure: $C[k] < C[k+1], \forall k \in \{1, 2, ..., C.n\}$
A,B := Lista de elementos
function UNION(A,B)
  $C \leftarrow \emptyset$
  $i, j \leftarrow 1$
  while $(i \leq A.n) \wedge (j \leq B.n)$ **do**
    if $A[i] = B[j]$ **then**
      $C \leftarrow C \cup \{A[i]\}$
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$
    else if $A[i] < B[j]$ **then**
      $C \leftarrow C \cup \{A[i]\}$
      $i \leftarrow i + 1$
    else
      $C \leftarrow C \cup \{B[j]\}$
      $j \leftarrow j + 1$
    end if
  end while
  while $i \leq A.n$ **do**
    $C \leftarrow C \cup \{A[i]\}$
    $i \leftarrow i + 1$
  end while
  while $j \leq B.n$ **do**
    $C \leftarrow C \cup \{B[j]\}$
    $j \leftarrow j + 1$
  end while
  return C
end function

2.4.2 Intersecção

O algoritmo para suportar a operação AND é similar ao algoritmo da operação OR. No entanto, aqui fazemos a interseção, mostrada no algoritmo 6, ao invés da união. Para isso, adicionamos um elemento para o conjunto C somente se ele é o menor elemento restante tanto no conjunto A quanto no conjunto B. No pior caso, é preciso passar por todos os elementos de ambos os conjuntos. Portanto, o tempo máximo de execução do algoritmo é da ordem de $\Theta(A.n + B.n)$.

Algorithm 6 Algoritmo de Intersecção entre conjuntos

Require: $A[i] < A[i+1], \forall i \in \{1, 2, ..., A.n\}$
Require: $B[j] < B[j+1], \forall j \in \{1, 2, ..., B.n\}$
Ensure: $C[k] < C[k+1], \forall k \in \{1, 2, ..., C.n\}$

```
A,B := Lista de elementos
function INTERSECTION(A,B)
    C ← ∅
    i, j ← 1
    while (i ≤ A.n) ∧ (j ≤ B.n) do
        if e := A[i] = B[j] then
            C ← C ∪ {e}
            i, j ← (i + 1), (j + 1)
        else if A[i] < B[j] then
            i ← i + 1
        else                                        ▷ A[i] > B[j]
            j ← j + 1
        end if
    end while
    return C
end function
```

2.4.3 Diferença

Para dar suporte à operação NOT, isto é, a exclusão de um termo dos resultados, primeiro é feita a pesquisa com os demais termos, e depois são retiradas as páginas em que o termo excluído aparece. Em outras palavras, seja A o conjunto de páginas resultantes da pesquisa sem levar o termo excluído em consideração, e seja B o conjunto de todas as páginas onde o termo excluído aparece, o resultado final da busca é dado por $A \setminus B$. Esta operação está descrita no algoritmo 7. Ele é similar aos algoritmos das operações AND e OR, mas ao contrário deles, o menor elemento restante só é adicionado ao conjunto C se ele é o menor elemento restante de A, mas não de B. Como os conjuntos estão em ordem crescente, é garantido que se ele não for o menor elemento restante de B, ele não está nesse conjunto. No pior caso, é preciso passar por todo o conjunto A e B, e assim a sua complexidade também é $\Theta(A.n + B.n)$.

Algorithm 7 Algoritmo de Diferença entre conjuntos

Require: $A[i] < A[i+1], \forall i \in \{1, 2, ..., A.n\}$
Require: $B[j] < B[j+1], \forall j \in \{1, 2, ..., B.n\}$
Ensure: $C[k] < C[k+1], \forall k \in \{1, 2, ..., C.n\}$
  A,B := Lista de elementos
function DIFFERENCE(A,B) ▷ $A \setminus B$
  $C \leftarrow \emptyset$
  $i, j \leftarrow 1$
  while $(i \leq A.n) \wedge (j \leq B.n)$ **do**
    if $A[i] = B[j]$ **then**
      $i, j \leftarrow (i+1), (j+1)$
    else if $A[i] < B[j]$ **then**
      $C \leftarrow C \cup \{A[i]\}$
      $i \leftarrow i + 1$
    else
      $j \leftarrow j + 1$
    end if
  end while
  while $i \leq A.n$ **do**
    $C \leftarrow C \cup \{A[i]\}$
    $i \leftarrow i + 1$
  end while
  return C
end function

2.5 Notação Polonesa Reversa

Para converter uma string de busca em um formato conveniente para as operações mencionadas anteriormente, foi utilizada notação polonesa reversa. A conversão se deu pelos seguintes processamentos (n é o tamanho da string):

1. Remoção de aspas sem correspondência ($O(n)$)
2. Remoção de parênteses sem correspondência ($O(n)$)
3. Adicionar aspas ao redor de todos os termos que já não as possuem i.e. unb => "unb" ($O(n)$)
4. Adicionar parênteses ao redor de todos os termos que já não os possuem i.e. "unb" => ("unb") ($O(n)$)
5. Conversão para notação polonesa ($O(n \cdot k)$, k é o número máximo de parênteses usados em um único termo)
6. Otimização da notação ($O(n)$)
7. Reversão da notação polonesa ($O(n)$)

2.6 Combinação de Intervalos

Para gerar o *preview* mostrado nos resultados, isto é, o trecho da página onde o termo pesquisado aparece, foi necessário por vezes fazer a combinação de intervalos. Como exemplo, se a *preview* do primeiro termo pesquisado é, em uma página, composta pelas palavras de índices 1 - 20, e a *preview* do segundo termo na mesma página é composta pelas palavras 11 - 30, é necessário fazer uma combinação dos dois e gerar o intervalo único 1 - 30. Essa combinação foi feita em $O(nlogn)$, onde n é o tamanho da lista de intervalos, com o algoritmo 8. O algoritmo utiliza a função sort, que é $O(nlogn)$, e então itera sobre a lista de intervalos, em complexidade $O(n)$.

Algorithm 8 Algoritmo de Diferença entre conjuntos

```
intervals := Lista de intervalos, cada elemento no formato [L, R]
upper_limit := Numero de palavras na pagina
function ADJUST_INTERVALS(Intervals, upper_limit)
    Intervals ← Intervals.sort()
    atual ← Intervals[1]
    Finais ← [ ]
    i ← 2
    while (i ≤ Intervals.n) do
        if atual[2] >= Intervals[i][1] then
            atual[2] ← max(atual[2], Intervals[i][2])
        else
            atual[1] ← max(0, atual[1])
            atual[2] ← min(upper_limit, atual[2])
            Finais.push(atual)
            atual ← Intervals[i]
        end if
    end while
    atual[1] ← max(0, atual[1])
    atual[2] ← min(upper_limit, atual[2])
    Finais.push(atual)
    return Finais
end function
```