Pontifícia Universidade Católica de Minas Gerais Instituto de Ciências Exatas e Informática

Nome: Gabriel Henrique Vieira de Oliveira, Vinicius Cezar Pereira Menezes
Data: 04/12/2025
Matéria: Algoritmos e Estruturas de Dados 3
Link Github: https://github.com/GabrielHenrique20/TP_AEDS3

FORMULÁRIO

1. Qual campo textual foi escolhido para aplicar os algoritmos de casamento de padrões? Por quê?

Para realizar o casamento de padrões, nessa atividade, optamos por escolher o atributo destino da entidade Viagens.

Observamos que é um texto puro, todos os registros apresentam ele nos .db, e o usuário naturalmente quer pesquisar viagens pelo local ou um termo/padrão em específico (o que vai ser bom para o menu de busca com KMP e Boyer-Moore).

2. Explique o funcionamento do KMP implementado.

Em nosso projeto, o KMP (um algoritmo de casamento de padrões usado para procurar um padrão P (o que o usuário digitou) dentro de um texto T (o campo destino das viagens em Viagem.java)), funciona da seguinte maneira:

A) Pré-processamento: Antes de procurar no texto, o KMP chama uma função do tipo computeLPSArray(), que pré-processa o padrão e monta o vetor LPS. Ele percorre o padrão com dois ponteiros (i avançando e len recuando quando há falha na leitura). O LPS em cada posição guarda o maior prefixo próprio que também é sufixo até ali; isso evita recomeçar do zero a cada mismatch.

B) Busca/search: Com o vetor LPS pronto, o KMP passa a comparar o padrão e o texto, onde:

- i percorre o texto (campo destino da viagem)
- j percorre o padrão selecionado pelo usuário

Alguns passos de ocasiões diferentes são aplicadas no código, sendo eles:

Caracteres iguais:

Se texto[i] == padrao[j], avança i++ e j++;

Se j chegar ao fim (j == m, onde m é o tamanho do padrão), significa que encontrou uma ocorrência:

Registra o índice da ocorrência (no seu código, isso vira um "achei essa viagem"), e usa j = lps[j - 1] para continuar a busca a partir do próximo possível casamento, sem voltar no texto;

‘ **Mismatch com j > 0:**

Se texto[i] != padrao[j] e j > 0, o KMP não anda para trás no texto;

Em vez de recuar i, ele ajusta só o padrão: j = lps[j - 1];

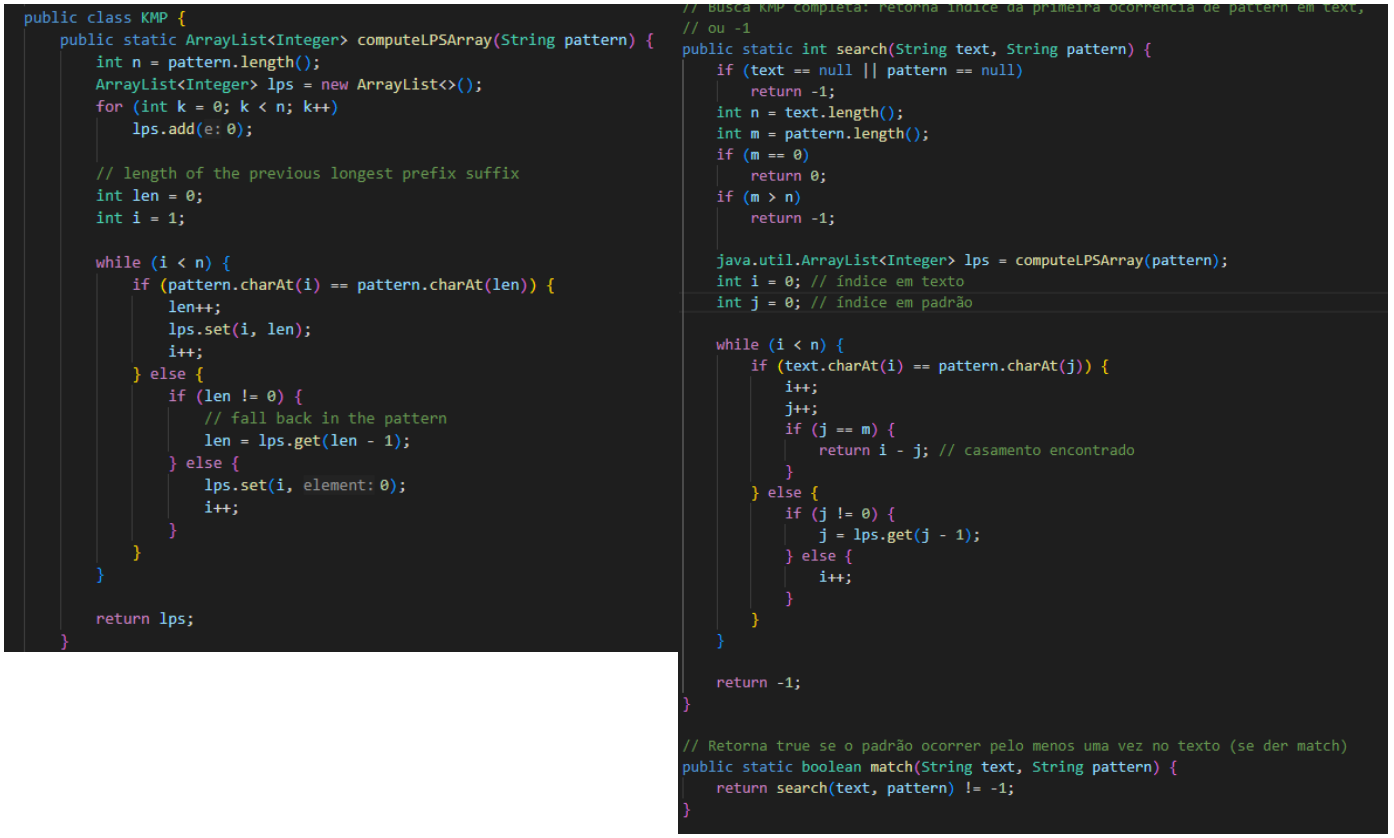
Ou seja, aproveita o que já sabe do padrão para tentar continuar de um ponto mais à frente.

Mismatch com j == 0:

Se texto[i] != padrao[0], não tem como reaproveitar nada do padrão.
O algoritmo simplesmente faz i++ e continua.

C) **Match:** método booleano que apenas aproveita o search e verifica se veio índice diferente de -1.

No geral, em nosso sistema, para cada viagem, o método que faz a busca pega viagem.getDestino() como texto e chama a função KMP passando o padrão. Se a função encontrar ao menos uma ocorrência, essa viagem vai para a lista de resultados do menu de busca.



```java
public class KMP {
    public static ArrayList<Integer> computeLPSArray(String pattern) {
        int n = pattern.length();
        ArrayList<Integer> lps = new ArrayList<>();
        for (int k = 0; k < n; k++)
            lps.add(e: 0);

        // length of the previous longest prefix suffix
        int len = 0;
        int i = 1;

        while (i < n) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps.set(i, len);
                i++;
            } else {
                if (len != 0) {
                    // fall back in the pattern
                    len = lps.get(len - 1);
                } else {
                    lps.set(i, element: 0);
                    i++;
                }
            }
        }

        return lps;
    }
```

```java
// Busca KMP completa: retorna índice da primeira ocorrência de pattern em text,
// ou -1
public static int search(String text, String pattern) {
    if (text == null || pattern == null)
        return -1;
    int n = text.length();
    int m = pattern.length();
    if (m == 0)
        return 0;
    if (m > n)
        return -1;

    java.util.ArrayList<Integer> lps = computeLPSArray(pattern);
    int i = 0; // índice em texto
    int j = 0; // índice em padrão

    while (i < n) {
        if (text.charAt(i) == pattern.charAt(j)) {
            i++;
            j++;
            if (j == m) {
                return i - j; // casamento encontrado
            }
        } else {
            if (j != 0) {
                j = lps.get(j - 1);
            } else {
                i++;
            }
        }
    }

    return -1;
}

// Retorna true se o padrão ocorrer pelo menos uma vez no texto (se der match)
public static boolean match(String text, String pattern) {
    return search(text, pattern) != -1;
}
```

3. Explique o funcionamento do Boyer–Moore implementado.

Em nosso projeto, o Boyer-Moore (um algoritmo de casamento de padrões com a versão baseada apenas na lógica do bad character, adaptada para funcionar bem com String em Java e com textos em português (acentos, etc.), e integrada ao menu de busca de viagens no atributo destino), foi implementado da seguinte maneira:

A) **Pré-processamento:** no método badCharHeuristic() constrói uma tabela badchar (de tamanho 65536), cobrindo todo o espaço Unicode (UTF-16), com o último índice de cada caractere do padrão (todas posições iniciam em -1). Isso permite, ao encontrar um mismatch e gerar um -1, saltar mais à frente. Essa tabela é o que permite decidir de quanto o padrão pode "andar" quando acontece um mismatch.

B) **Busca/indexOf:** com a tabela pronta, acontece o seguinte passo a passo:

O padrão é alinhado com o texto em uma posição s (inicialmente 0);

Compara-se do fim do padrão para o início (j = m - 1 até 0):

Enquanto pat[j] == txt[s + j], o algoritmo vai deslocando j–;

Se j chega a -1, significa que todos os caracteres bateram -> há casamento; o método retorna s (índice da primeira ocorrência).

Se houver mismatch em txt[s + j]:

O algoritmo consulta badchar[txt[s + j]] para saber a última posição desse caractere no padrão.

Ou seja, descola o padrão para a direita o máximo possível, sem pular nenhum casamento.

O texto nunca vai voltar para trás, apenas o padrão é deslocado e sempre para frente.

C) **Match:** método elaborado para facilitar a integração com nosso menu de busca, e assim foi criado um wrapper; indexOf devolve o índice da primeira ocorrência ou -1 e depois match devolve só true/false.

```java
// Versão que retorna o índice da primeira ocorrência de padrão em txt (ou -1)
public static int indexOf(String txt, String pat) {
    if (txt == null || pat == null)
        return -1;
    int n = txt.length();
    int m = pat.length();
    if (m == 0)
        return 0;
    if (m > n)
        return -1;

    int[] badchar = new int[NO_OF_CHARS];
    badCharHeuristic(pat.toCharArray(), m, badchar);

    int s = 0; // shift do padrão em relação ao texto
    while (s <= (n - m)) {
        int j = m - 1;

        // anda do fim do padrão para o início enquanto os caracteres casam
        while (j >= 0 && pat.charAt(j) == txt.charAt(s + j))
            j--;

        if (j < 0) {
            return s; // casamento encontrado
        } else {
            s += max(a: 1, j - badchar[txt.charAt(s + j)]);
        }
    }

    return -1;
}

// Retorna true se o padrão ocorrer pelo menos uma vez no texto
public static boolean match(String txt, String pat) {
    return indexOf(txt, pat) != -1;
}
```

```java
public class BoyerMoore {

    static int NO_OF_CHARS = 65536; // vai pegar o numero de caracteres Unicode (UTF-16)

    // A utility function to get maximum of two integers
    static int max(int a, int b) {
        return (a > b) ? a : b;
    }

    // The preprocessing function for Boyer Moore's
    // bad character heuristic
    static void badCharHeuristic(char[] str, int size,
            int badchar[]) {

        // Initialize all occurrences as -1
        for (int i = 0; i < NO_OF_CHARS; i++)
            badchar[i] = -1;

        // Fill the actual value of last occurrence
        // of a character (indices of table are ascii and
        // values are index of occurrence)
        for (int i = 0; i < size; i++)
            badchar[(int) str[i]] = i;
    }
/*
* A pattern searching function that uses Bad
* Character Heuristic of Boyer Moore Algorithm
*/
static void search(char txt[], char pat[]) {
    int m = pat.length;
    int n = txt.length;

    int badchar[] = new int[NO_OF_CHARS];

    /*
     * Fill the bad character array by calling
     * the preprocessing function badCharHeuristic()
     * for given pattern
     */
    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with
               // respect to text
    // there are n-m+1 potential alignments
    while (s <= (n - m)) {
        int j = m - 1;

        /*
         * Keep reducing index j of pattern while
         * characters of pattern and text are
         * matching at this shift s
         */
        while (j >= 0 && pat[j] == txt[s + j])
            j--;

        /*
         * If the pattern is present at current
         * shift, then index j will become -1 after
         * the above loop
         */
        if (j < 0) {
```

4. Descreva como integrou os algoritmos ao sistema.

Os algoritmos de casamento de padrões (KMP e Boyer–Moore) foram integrados ao sistema na camada de menus, reutilizando a estrutura já existente de CRUD de viagens. A integração foi feita em alguns pontos principais:

1º) EM **dao/Arquivo.java, no método readAll(),** ocorre uma varredura do .db inteiro, reconstruindo todos os registros ativos em memória. Serve de base para aplicar KMP/Boyer–Moore sem depender de índices.

```java
// Ler todos os registros .db válidos para listar e fazer a busca com o casamento de padroes
public java.util.ArrayList<T> readAll() throws Exception {
    java.util.ArrayList<T> lista = new java.util.ArrayList<>();
    arquivo.seek(TAM_CABECALHO);
    while (arquivo.getFilePointer() < arquivo.length()) {
        @SuppressWarnings("unused")
        long posicao = arquivo.getFilePointer();
        byte lapide = arquivo.readByte();
        short tamanho = arquivo.readShort();
        byte[] dados = new byte[tamanho];
        arquivo.read(dados);

        if (lapide == ' ') {
            T obj = construtor.newInstance();
            obj.fromByteArray(dados);
            lista.add(obj);
        }
    }
    return lista;
}
```

2º) EM **dao/ViagemDAO.java, no método listarTodas(),** existe wrapper que chama readAll() e devolve List<Viagem> pronta para filtrar por padrão.

```java
// listar todas as viagens (varrendo todos os arquivos de dados para o casamento de padroes)
public java.util.List<Viagem> listarTodas() throws Exception {
    java.util.ArrayList<Viagem> out = new java.util.ArrayList<>();
    for (Viagem v : arqViagem.readAll()) {
        if (v != null) {
            out.add(v);
        }
    }
    return out;
}
```

3º) EM **controller/MenuViagem.java**, o menu ganhou a opção 5 - Pesquisar por padrão (KMP / Boyer-Moore), de acordo com as requisições do professor. Nele, há a presença do método pesquisarPorDestino(), responsável por:

- Lê o padrão digitado e a escolha do algoritmo (1 = KMP, 2 = Boyer-Moore);
- Carrega todas as viagens via listarTodas();
- Coloca as variáveis destino e padrão em minúsculas;
- Aplica KMP.match ou BoyerMoore.match sobre o campo textual destino;
- Coleta e exibe apenas as viagens cujo destino contém o padrão; se nenhuma, mostra a mensagem “Nenhuma viagem encontrada cujo destino contenha o padrão informado”;

Assim, a interface de pesquisa usa os dois algoritmos implementados sobre o campo textual mais prático (destino de Viagem), cumprindo o requisito do menu “Pesquisar por padrão (KMP / BM)”.

5. Quais dificuldades encontrou na implementação dos dois algoritmos?

A principal dificuldade não foi tanto entender a teoria do KMP e do Boyer–Moore em si, mas encaixar esses algoritmos dentro da estrutura do nosso sistema sem quebrar o que já existia, como por exemplo:

Lidar com a camada de persistência (Arquivo.java e DAO):

No começo tentamos aproximar demais a busca da parte de arquivo, e isso gerou confusão. Depois vimos que o melhor era usar o DAO só para carregar as viagens e aplicar KMP/Boyer–Moore no campo destino;

Ajustes práticos no código:

Tivemos alguns problemas com índices, String e char[], normalização (maiúsculas/minúsculas) e casos especiais (padrão vazio, padrão maior que o texto) até os dois algoritmos funcionarem direitinho no fluxo do sistema;