Informe N° 12: Pattern Matching

Gianfranco Pupiales

2024-08-01

Tabla de Contenidos

1. Objetivos

- Distinguir los conceptos fundamentales del algoritmo de Fuerza Bruta.
- Distinguir los conceptos fundamentales del algoritmo Knuth-Morris-Pratt (KMP).

2. Introducción

En el contexto de cadenas de texto, el algoritmo de fuerza bruta busca una subcadena específica dentro de una cadena más grande comparando cada posición posible de la subcadena dentro de la cadena principal. Comienza alineando la subcadena con el primer carácter de la cadena principal y verifica carácter por carácter si coincide. Si hay una discrepancia, desplaza la subcadena una posición hacia la derecha y repite el proceso hasta que se encuentre una coincidencia completa o se haya revisado toda la cadena principal («Pattern Matching Algorithm in C - javatpoint — javatpoint.com»).

```
def brute_force(text, pattern):
    l1 = len(text)
    l2 = len(pattern)

    i = 0

    flag = False

    while i < l1:
        j = 0
        count = 0

        while j < l2:
            if i + j < l1 and text[i + j] == pattern[j]:
                count += 1
            else:
                break

            j += 1

        if count == l2:
            print("\nPattern occurs at index", i)
            flag = True

        i += 1

    if not flag:
        print("\nPattern is not at all present in the text")
```

El algoritmo Knuth-Morris-Pratt (KMP) es un eficiente método de búsqueda de subcadenas que mejora sobre el enfoque de fuerza bruta al preprocesar la subcadena a buscar, construyendo un arreglo de prefijos que permite evitar comparaciones redundantes. Al comparar la subcadena con la cadena principal, el algoritmo utiliza este arreglo para saltar posiciones cuando encuentra una discrepancia, basándose en la información de las coincidencias previas, en lugar de retroceder en la cadena principal («Implementación del algoritmo KMP: C, C++, Java y Python — techiedelight.com»).

```
def failure_function(P):
    m = len(P)
    valor_f = [0] * m
    k = 0
    j = 1
```

```
    while j < m:
        if P[j] == P[k]:
            valor_f[j] = k + 1
            j += 1
            k += 1
        elif k > 0:
            k = valor_f[k - 1]
        else:
            j += 1
    return valor_f


def find_kmp(T, P):
    n, m = len(T), len(P)

    if m == 0:
        return 0

    result_failure = failure_function(P)
    print('f(k) =', result_failure)

    j = 0
    k = 0

    while j < n:
        if T[j] == P[k]:
            if k == m - 1:
                return j - m + 1
            j += 1
            k += 1
        elif k > 0:
            k = result_failure[k - 1]
        else:
            j += 1
    return -1
```

3. Desarrollo

Haga los siguientes ejercicios de la forma manual, donde se aprecie cómo el patrón se desplaza y la prueba de escritorio (cambio de valores en variables) con la ejecución del algoritmo

correspondiente. Puede hacer las capturas de pantalla o scanning con las debidas resoluciones y añadir las imágenes a su archivo pdf de informe .

3.1 Enunciado ejercicio 1

1. Aplique el algoritmo de Brute Force para tareas de pattern matching en el lenguaje que usted prefiera. Pruebe con:

a) T = "ABDDCCABBCAABABC", P = "AB"

Ejercicio A Brute Force a mano

```
brute_force('ABDDCCABBCAABABC', 'AB')
```

```
Pattern occurs at index 0

Pattern occurs at index 6

Pattern occurs at index 11

Pattern occurs at index 13
```

b) T = "ABDDCCABBCAABAB", P = "ABA"

Ejercicio B Brute Force a mano

```
brute_force('ABDDCCABBCAABAB', 'ABA')
```

```
Pattern occurs at index 11
```

c) T = "ABDDCCABB", P = "CCC"

Ejercicio C Brute Force a mano

```
brute_force('ABDDCCABB', 'CCC')
```

```
Pattern is not at all present in the text
```

3.2 Enunciado ejercicio 2

Aplique el algoritmo Knuth-Morris-Pratt con la asistencia de la función "failure". Coloque dos ejemplos con coincidencia entre T y P y un ejemplo más sin coincidencia.

a) T = 'thisisatesttext', P = 'test'

Ejercicio A KMP a mano

```
print(find_kmp('thisisatesttext', 'test'))
```

```
f(k) = [0, 0, 0, 1]
7
```

b) T = 'ababcabcdaaddab', P = 'abcabcda'

Ejercicio B KMP a mano

```
print(find_kmp('ababcabcdaaddab', 'abcabcda'))
```

```
f(k) = [0, 0, 0, 1, 2, 3, 0, 1]
2
```

c) T = 'abcdabcaabd', P = 'ababc'

Ejercicio C KMP a mano

```
print(find_kmp('abcdabcaabd', 'ababc'))
```

```
f(k) = [0, 0, 1, 2, 0]
-1
```

4. Conclusiones

- El algoritmo de Fuerza Bruta, aunque sencillo y directo, puede resultar ineficiente para buscar subcadenas en textos extensos. Al comparar cada posición posible de la subcadena dentro del texto principal, su tiempo de ejecución puede ser considerablemente alto en el peor de los casos, especialmente cuando el patrón es pequeño y se repite frecuentemente en el texto. Si bien este método es fácil de implementar y comprender, no es adecuado para manejar grandes volúmenes de datos o cuando se requiere una alta eficiencia.
- El algoritmo KMP proporciona una solución más eficiente para la búsqueda de subcadenas al emplear un arreglo de prefijos que elimina comparaciones redundantes. Al preprocesar el patrón y construir este arreglo, el algoritmo puede saltar posiciones en el texto principal, disminuyendo notablemente la cantidad de comparaciones necesarias.

5. Declaración del Uso de IA

En este informe se utilizó IA para mejorar la coherencia, cohesión y sintaxis de las conclusiones.

6. Referencias Bibliográficas

«Implementación del algoritmo KMP: C, C++, Java y Python — techiedelight.com». https://www.techiedelight.com/es/implementation-kmp-algorithm-c-cpp-java/.

«Pattern Matching Algorithm in C - javatpoint — javatpoint.com». https://www.javatpoint.com/pattern-matching-algorithm-in-c#:~:text=Brute%20Force%20Pattern%20Matching%20is,the%20pattern%20in%20the%20text.