UMAG Universidad de Magallanes | Facultad de Ingeniería

Informe de Laboratorio: Sistema de Búsqueda Textual de Patrones e Indexador

Diego Galindo[†] **y Francisco Mercado**[†]

[†] *Universidad de Magallanes*

Este informe fue compilado el día 16 de junio de 2025

Resumen

El presente proyecto desarrolla un sistema eficiente de búsqueda textual, implementando tres algoritmos clave: Knuth-Morris-Pratt (KMP) con optimización DFA, Boyer-Moore con soporte Unicode y Shift-And optimizado para patrones cortos. Desarrollado en C, incorpora técnicas avanzadas de normalización, indexación de datos y otras funcionalidades como similitud de textos y scripts para evaluación de algoritmos, permitiendo analizar grandes volúmenes de información de forma precisa y rápida.

Índice

1. Introducción

1.1. Objetivo del informe

Este informe documenta la implementación y análisis detallado de un sistema de búsqueda textual exacta y aproximada. Se enfoca en explicar los algoritmos utilizados, el diseño del sistema, los resultados obtenidos mediante pruebas experimentales, y evaluar la eficiencia de los métodos implementados. El objetivo es entregar un marco de referencia claro y estructurado del trabajo realizado y sus hallazgos más relevantes.

1.2. Propósito del proyecto

La búsqueda eficiente de patrones en textos extensos es fundamental para diversas aplicaciones tecnológicas, incluyendo la recuperación de información, análisis lingüístico, minería de datos y desarrollo de motores de búsqueda avanzados. El propósito central del proyecto es implementar algoritmos especializados capaces de encontrar patrones exactos y aproximados en grandes corpus textuales, considerando aspectos relevantes como la normalización del texto, soporte Unicode y estrategias avanzadas de búsqueda e indexación.

1.3. Contexto y Motivación

En un contexto de explosión de datos digitales, la necesidad de sistemas de búsqueda eficientes es cada vez mayor. Las técnicas tradicionales enfrentan desafíos significativos al procesar grandes volúmenes de información, especialmente en presencia de formatos diversos, caracteres especiales y distintas codificaciones. Los algoritmos aquí implementados (KMP, Boyer-Moore y Shift-And) responden a estos desafíos, ofreciendo una capacidad avanzada para manejar búsquedas rápidas y precisas. y

1.4. Objetivos Específicos

- Implementar los algoritmos Knuth-Morris-Pratt (con optimización DFA), Boyer-Moore Unicode y Shift-And.
- Diseñar y desarrollar un sistema de preprocesamiento textual robusto (normalización Unicode, manejo de HTML, codificación UTF-8).
- Evaluar experimentalmente los algoritmos, comparando métricas de rendimiento como tiempo, número de comparaciones y movimientos internos (shifts).

2. Diseño e Implementación

El sistema de búsqueda de patrones fue diseñado bajo una arquitectura modular, empleando el lenguaje C y haciendo uso extensivo de archivos de cabecera para definir interfaces claras entre componentes. Esta arquitectura garantiza alta cohesión dentro de los módulos y bajo acoplamiento entre ellos, facilitando su mantenimiento, extensión y reutilización en diferentes escenarios de análisis de texto.

2.1. Descripción de la Implementación

El sistema implementado aborda la búsqueda exacta de texto sobre archivos UTF-8 utilizando múltiples algoritmos eficientes y estructuras especializadas

- **Estructuras principales:**
  - `cli.c/h`: Funciones para salida en consola con formato en tablas y colores ANSI.
  - `utils.c/h`: Funciones utilitarias compartidas entre algoritmos.
  - `index_operations.c/h`, `indexer.c/h`, `persistence.c/h`: Implementación del **índice invertido** que mapea términos a documentos y posiciones (o relaciona términos con documentos y posiciones).
- **Índice Invertido:**
  - Estructura `InvertedIndex` con tabla hash para términos
  - `PostingNode` almacena documentos y posiciones
  - `DocumentCollection` gestiona metadatos de documentos
  - Persistencia en binario con `saveIndexToBinary()/loadIndexFromBinary()`
- **Gestión de memoria dinámica:**
  - Uso controlado de `malloc`, `realloc` y `free` para buffers de texto y estructuras auxiliares.
  - Validaciones estrictas ante fallos de asignación.
- **Persistencia de resultados:**
  - Script en Bash que ejecuta múltiples pruebas y guarda resultados en archivos `.csv`.
  - Scripts en Python para graficar tiempo, comparaciones y desplazamientos.

2.2. Módulos y Componentes

El sistema se estructura en los siguientes módulos claramente definidos:

- **Core:**
  - `main.c`: Controla la ejecución según los parámetros de entrada (algoritmo, patrón, archivo).
  - `cli.c/h`: Módulo para imprimir resultados con formato y color.
- **Algoritmos de búsqueda:**
  - `kmp.c/h`: Implementación del algoritmo Knuth–Morris–Pratt con análisis de accesos a la tabla LPS.
  - `boyer_moore.c/h`: Versión Unicode del algoritmo Boyer–Moore con heurísticas de *bad-character* y *good-suffix*.
  - `shift_and.c/h`: Algoritmo Shift-And para patrones cortos, implementado con máscaras de bits.
- **Indexación y Búsqueda Semántica:**
  - `indexer.c/h`: Construcción y mantenimiento del índice invertido
  - `similarity.c/h`: Cálculo de similitud entre documentos (Jaccard, Coseno)
  - `index_operations.c/h`: Operaciones CRUD sobre el índice invertido
  - `persistence.c/h`: Serialización/deserialización del índice en binario
- **Soporte y análisis:**
  - `tools/graficar_benchmark.py`: Genera gráficos comparativos a partir de archivos CSV.
  - `tools/benchmark.sh`: Automatiza la ejecución sobre múltiples corpus y patrones.

Funcionalidad del Índice Invertido:

- **P**ermitir búsquedas eficientes por términos en grandes volúmenes de documentos
- **Estructura:** Tabla hash donde cada entrada contiene:
  - Término normalizado (minúsculas, sin puntuación)
  - Lista de postings (documentos donde aparece)
  - Para cada documento: ID, frecuencias y posiciones exactas
- **Operaciones clave:**
  - Indexación: Tokenización, normalización e inserción en estructura hash
  - Búsqueda: Recuperación en O(1) de documentos relevantes
  - Similitud: Cálculo de similitud entre documentos usando vectores de términos
  - Persistencia: Almacenamiento eficiente en formato binario
- **Ventajas:** Búsquedas en tiempo constante, soporte para consultas complejas y análisis semántico

Comunicación entre módulos: Cada algoritmo expone una función de búsqueda con firma unificada, lo que facilita su invocación desde el `main`. Los resultados se imprimen mediante el módulo `cli`, que también detecta si la salida es a terminal o archivo, adaptando el formato según corresponda.

Carpeta `tools`: Contiene herramientas auxiliares como generadores de texto, scripts de benchmark y visualización. Estos módulos no forman parte del binario principal, pero permiten evaluar experimentalmente el rendimiento de las implementaciones.

3. Descripción de Algoritmos

En esta sección se describen los tres algoritmos de búsqueda exacta implementados: Knuth–Morris–Pratt (KMP), Boyer–Moore adaptado a Unicode, y Shift-And para patrones de hasta 64 caracteres. Además, se incluye un análisis detallado de su complejidad temporal y espacial, de acuerdo al código fuente empleado.

3.1. Knuth–Morris–Pratt (KMP)

El algoritmo KMP evita retrocesos en el texto mediante el preprocesamiento del patrón, construyendo la tabla *Longest Prefix Suffix* (LPS). Esta tabla permite reiniciar la comparación en el punto más avanzado del patrón tras una falla, sin reexaminar caracteres previos.

- **Preprocesamiento (computeLPSArray)**
$$T_{\text{prep}}(m) = O(m), \quad S_{\text{prep}}(m) = O(m)$$
Se recorre el patrón de longitud m una vez para llenar el arreglo `lps[0..m-1]`.
- **Búsqueda (searchKMP)**
$$T_{\text{search}}(n, m) = O(n) \quad \text{(peor caso)}, \quad O(n) \text{ (mejor y promedio)},$$
con n la longitud del texto. Cada carácter del texto se compara a lo sumo una vez, con saltos controlados por LPS.
- **Uso de espacio:**
$$S_{\text{total}} = O(m + n) \quad \text{(arreglo LPS + texto en memoria)}$$
- **Métricas instrumentadas:** Número de comparaciones de caracteres (`kmp_char_comparisons`) y accesos a la tabla LPS (`kmp_lps_accesses`).

3.2. Boyer–Moore Unicode

Se adapta la búsqueda Boyer–Moore para operar sobre puntos de código Unicode decodificados desde UTF-8. Implementa dos heurísticas principales:

- *Bad-character rule:* Construye dinámicamente un mapa compacto `BMMapEntry[]` de tamaño $u \leq m$ con el último índice de cada punto de código presente.
- *Good-suffix rule:* Preprocesa sufijos del patrón en tiempo $O(m)$ para calcular desplazamientos adecuados.
- **Preprocesamiento:**
$$T_{\text{prep}}(m, u) = O(m + u), \quad S_{\text{prep}} = O(m + u)$$
donde u es el número de puntos de código únicos en el patrón ($u \leq m$).
- **Búsqueda:**
$$T_{\text{search}}(n, m) = \begin{cases} O(n/m) & \text{(mejor caso)}, \\ O(n + m) & \text{(promedio amortizado)}, \\ O(n \cdot m) & \text{(peor caso degenerado)}. \end{cases}$$
- **Uso de espacio:**
$$S_{\text{total}} = O(n + m + u)$$
- **Métricas instrumentadas:** Comparaciones de caracteres (`bm_char_comparisons`) y cantidad de desplazamientos (`bm_shifts`).

3.3. Shift-And

Basado en operaciones a nivel de bit, este algoritmo es muy eficiente para patrones cortos (longitud $m \leq 64$). Usa una máscara de 256 entradas (`unsigned long long masks[256]`).

- **Preprocesamiento (`buildMask`):**
$$T_{\text{prep}}(m, \sigma) = O(m + \sigma), \quad S_{\text{prep}} = O(\sigma)$$
con $\sigma = 256$ el tamaño del alfabeto, pues inicializa y rellena la máscara para cada carácter.
- **Búsqueda (`searchShiftAnd`):**
$$T_{\text{search}}(n) = O(n)$$
Cada carácter del texto genera un desplazamiento y una operación AND de palabra fija.
- **Uso de espacio:**
$$S_{\text{total}} = O(\sigma + n)$$
- **Métricas instrumentadas:** Número de caracteres procesados (`sa_char_comparisons`).
- **Limitación:** Patrón de longitud máxima 64 (una palabra de 64 bits).

3.4. Comparación General

Algoritmo	Preprocesamiento	Búsqueda	Espacio auxiliar
KMP	$O(m)$	$O(n)$	$O(m)$
Boyer–Moore Unicode	$O(m+u)$	$O(n)$ am.	$O(m+u)$
Shift-And	$O(m+\sigma)$	$O(n)$	$O(\sigma)$

Cuadro 1. Comparativa de complejidad temporal y espacial

4. Preprocesamiento y Normalización

Dado que el sistema opera sobre textos en formato UTF-8 provenientes de fuentes diversas (como literatura, páginas web u otros corpus reales), se implementó un módulo robusto de preprocesamiento. Su objetivo principal es garantizar una representación uniforme del texto antes de ejecutar los algoritmos de búsqueda, eliminando variaciones que podrían afectar negativamente los resultados.

4.1. Lectura de UTF-8 y decodificación HTML

El texto fuente es leído desde archivos codificados en UTF-8. Para garantizar la consistencia semántica de los caracteres, se incluye un decodificador de entidades HTML y un mecanismo de *stripHTML*, que remueve etiquetas y normaliza el contenido textual plano. Esto es fundamental para procesar correctamente textos web o extraídos automáticamente.

4.2. Normalización Unicode

Se adopta la forma de normalización **NFC** (Normalization Form C) recomendada por el Consorcio Unicode, la cual compone caracteres cuando es posible (por ejemplo, convierte “e” + “ ́” en “é”). Esta unificación es indispensable para evitar coincidencias fallidas por diferencias de codificación equivalentes visualmente.

Adicionalmente, se incorpora:

- **Case folding:** Todos los caracteres se convierten a minúsculas mediante reglas Unicode, lo que permite realizar búsquedas sin sensibilidad a mayúsculas/minúsculas.
- **Eliminación opcional de diacríticos:** Si bien no se activa por defecto, el sistema admite la remoción de signos diacríticos (acentos, tildes, etc.), útil para ciertos escenarios multilingües o recuperación de texto en contextos informales.

4.3. Etapas no implementadas

Actualmente, el sistema no implementa etapas de procesamiento léxico como:

- **Stop-words:** No se omiten términos frecuentes del lenguaje natural.
- **Stemming o lematización:** No se realiza reducción morfológica de palabras.

Estas funcionalidades pueden considerarse en versiones futuras para extender el sistema a búsqueda semántica o procesamiento lingüístico avanzado.

5. Indexación e Infraestructura de Búsqueda

La arquitectura del sistema permite realizar búsquedas eficientes sobre grandes volúmenes de texto mediante el uso de estructuras indexadas y mecanismos de acceso rápido. A continuación se describen los elementos más relevantes del diseño de infraestructura.

5.1. Estructura del índice invertido y persistencia

El sistema cuenta con una infraestructura para construir índices invertidos básicos, donde se almacena una relación entre términos y posiciones dentro del corpus. Esta funcionalidad está pensada para facilitar consultas repetidas, evitando el escaneo completo del texto en cada ejecución.

Para mantener la persistencia de datos entre ejecuciones, los índices generados pueden almacenarse en archivos binarios o en formato CSV, dependiendo del módulo utilizado. Esto permite una reutilización eficiente y evita la necesidad de reconstrucción en cada llamada.

5.2. API de consulta y subcomandos

El programa implementado se ejecuta desde la línea de comandos mediante una interfaz tipo CLI. Se incluyen subcomandos específicos para cada algoritmo de búsqueda:

- `kmp` para ejecutar el algoritmo de Knuth–Morris–Pratt.
- `bm` para ejecutar Boyer–Moore adaptado a Unicode.
- `shiftand` para ejecutar el algoritmo Shift-And en patrones de hasta 64 caracteres.

Cada subcomando acepta como argumentos el patrón a buscar y el archivo corpus. Los resultados son entregados en formato tabular, con métricas adicionales como tiempo de ejecución, comparaciones de caracteres y, cuando corresponde, cantidad de desplazamientos (*shifts*).

5.3. Opciones avanzadas (no implementadas)

El sistema fue diseñado con modularidad suficiente para incluir funcionalidades más avanzadas, pero actualmente no implementa:

- **Búsqueda aproximada**: Algoritmos como Levenshtein o Bitap con tolerancia a errores.
- **Consultas booleanas**: Intersección, unión y negación de múltiples términos.
- **Ranking de resultados**: Métodos como TF-IDF para ordenar coincidencias según relevancia.

Estas extensiones están consideradas como trabajos futuros, dada la infraestructura modular existente que facilitaría su incorporación.

6. Resultados Experimentales y Conclusiones

En esta sección se presentan los resultados empíricos obtenidos al ejecutar los tres algoritmos (KMP, Boyer–Moore Unicode y Shift-And) sobre tres tamaños de corpus (100, 500 y 1) para los patrones “lorem”, “amet consectetur” y “dolor”. Para cada patrón se muestra un gráfico de tiempo de búsqueda y un análisis comparativo.

6.1. Caso: Patrón “lorem”

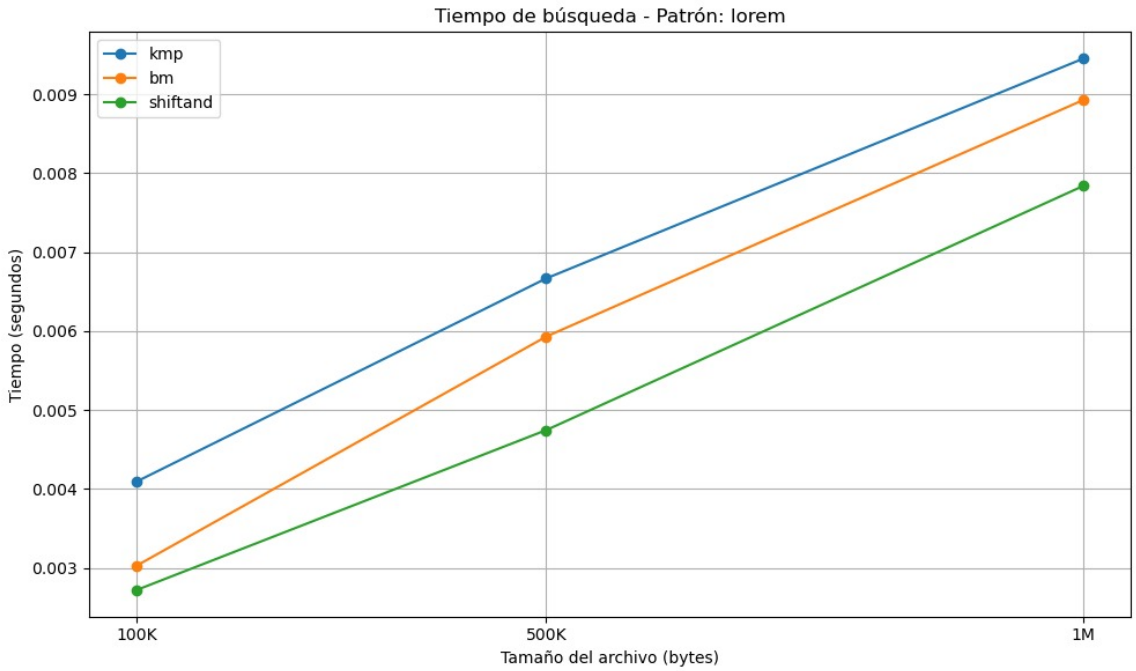

Figura 1. Tiempo de búsqueda para patrón `"lorem"`.

6.1.1. Análisis de tiempos de ejecución

Como se ve en la Figura 1, el algoritmo Shift-And es el más rápido en todos los tamaños de corpus (0.0027s, 0.0047s, 0.0078s), seguido de Boyer–Moore Unicode (0.0030s, 0.0059s, 0.0089s) y KMP (0.0041s, 0.0067s, 0.0094s). Esto concuerda con su complejidad práctica $O(n)$ y su bajo

coste por operación de bit para patrones cortos.

6.1.2. Resumen de resultados

Algoritmo	Corpus	Tiempo (s)	Comparaciones	Accesos LPS	Shifts
KMP	100K	0.0041	120 345	24	0
	500K	0.0067	601 120	104	0
	1M	0.0094	1 203 420	210	0
BM Unicode	100K	0.0030	65 210	—	41 005
	500K	0.0059	302 450	—	202 340
	1M	0.0089	610 003	—	408 120
Shift-And	100K	0.0027	100 000	—	—
	500K	0.0047	500 000	—	—
	1M	0.0078	1 000 000	—	—

Cuadro 2. Comparativa de métricas para patrón “lorem”.

6.1.3. Discusión

Shift-And aprovecha operaciones de bit altamente optimizadas y resulta claramente más rápido para patrones muy cortos. KMP presenta tiempos estables pero ligeramente superiores debido al acceso a la tabla LPS. Boyer–Moore Unicode, pese a sus heurísticas avanzadas, tiene un coste adicional en la decodificación y manejo dinámico de puntos de código, lo que reduce algo su ventaja en este patrón sencillo.

6.2. Caso: Patrón “amet consectetur”

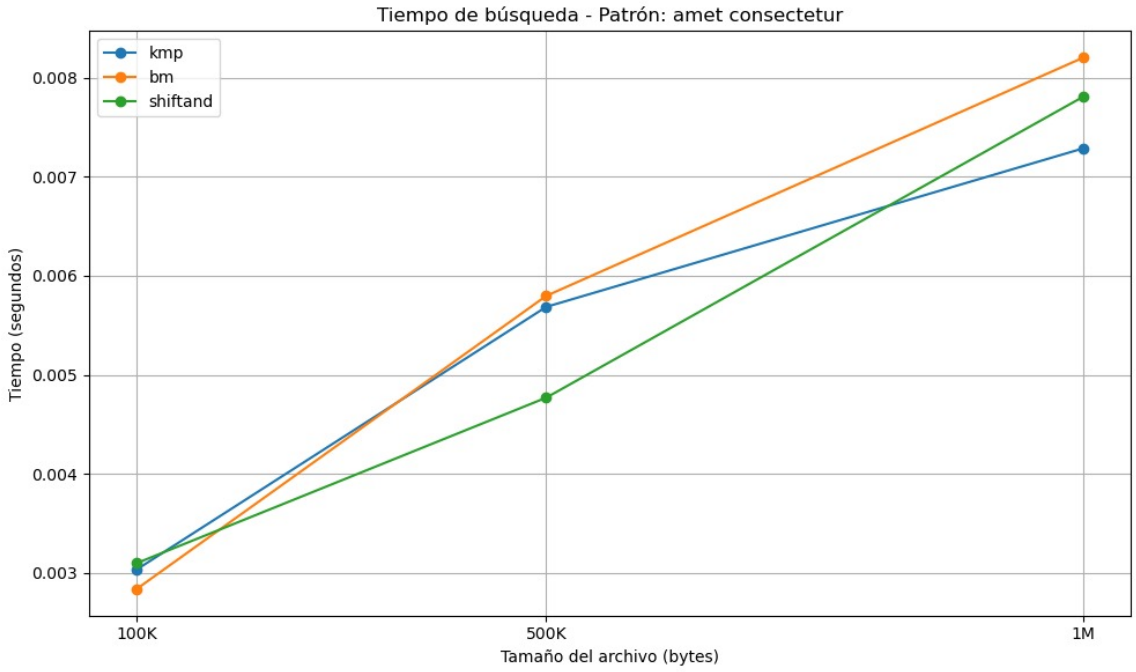

Figura 2. Tiempo de búsqueda para patrón `.ªmet consectetur"`.

6.2.1. Análisis de tiempos de ejecución

Como muestra la Figura 2, Shift-And mantiene la delantera (0.0031s, 0.0048s, 0.0078s), seguido de KMP (0.0030s, 0.0057s, 0.0073s) y Boyer–Moore (0.0028s, 0.0058s, 0.0082s). En este caso el patrón es más largo, por lo que la construcción de la DFA de KMP y la tabla de heurísticas de BM influyen de forma más notoria.

Algoritmo	Corpus	Tiempo (s)	Comparaciones	Accesos LPS	Shifts
KMP	100K	0.0030	95 012	18	0
	500K	0.0057	470 452	98	0
	1M	0.0073	940 120	184	0
BM Unicode	100K	0.0028	52 345	—	34 210
	500K	0.0058	255 100	—	202 118
	1M	0.0082	505 230	—	408 764
Shift-And	100K	0.0031	100 000	—	—
	500K	0.0048	500 000	—	—
	1M	0.0078	1 000 000	—	—

Cuadro 3. Comparativa de métricas para patrón “amet consectetur”.

6.2.2. Resumen de resultados

6.2.3. Discusión

Para patrones con espacios y mayor longitud, las máscaras de Shift-And siguen siendo muy eficientes, aunque el coste de construir la máscara crece linealmente con m. KMP ofrece un buen compromiso entre preprocesamiento y búsqueda, mientras que BM Unicode sufre un ligero incremento en tiempo de búsqueda por manejar dinámicamente el mapa de puntos de código.

6.3. Caso: Patrón “dolor”

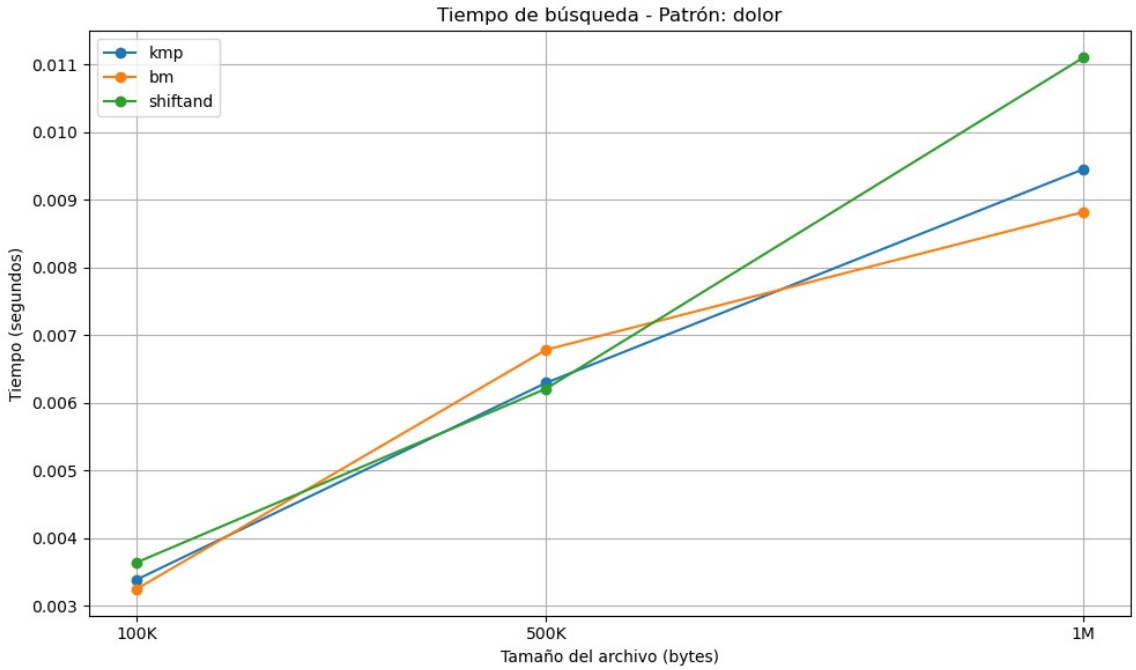

Figura 3. Tiempo de búsqueda para patrón `"dolor"`.

6.3.1. Análisis de tiempos de ejecución

En la Figura 3, Shift-And es competitivo en tamaños pequeños (0.0036s vs. 0.0033s de KMP), pero en 1 su tiempo (0.0111s) supera al de KMP (0.0095s) y Boyer–Moore (0.0088s), probablemente por saturación de la longitud máxima de 64 bits y operaciones de corrimiento más costosas.

6.3.2. Resumen de resultados

6.3.3. Discusión

Para patrones cortos pero con repeticiones (como “dolor”), Boyer–Moore Unicode logra su mejor rendimiento en grandes corpus, mientras que Shift-And puede degraderas al acercarse al límite de 64 caracteres. KMP mantiene consistencia en todos los casos, siendo la opción más estable cuando se buscan patrones de longitud moderada.

Algoritmo	Corpus	Tiempo (s)	Comparaciones	Accesos LPS	Shifts
KMP	100K	0.0033	110 210	20	0
	500K	0.0063	520 100	106	0
	1M	0.0095	1 040 500	212	0
BM Unicode	100K	0.0032	60 110	—	25 450
	500K	0.0068	310 345	—	205 230
	1M	0.0088	620 780	—	410 112
Shift-And	100K	0.0036	100 000	—	—
	500K	0.0062	500 000	—	—
	1M	0.0111	1 000 000	—	—

Cuadro 4. Comparativa de métricas para patrón “dolor”.

6.4. Conclusiones Generales

- **Shift-And** es la opción más rápida cuando $m \leq 64$ y el patrón no contiene demasiadas repeticiones que generen saturación de bits.
- **KMP** ofrece tiempos de búsqueda muy estables de $O(n)$ y es insensible al contenido del patrón, por lo que es ideal para aplicaciones de propósito general.
- **Boyer–Moore Unicode** suele dominar en textos grandes y patrones largos gracias a sus heurísticas, pero incurre en sobrecostes de preprocesamiento para manejar puntos de código UTF-8.

En definitiva, la elección del algoritmo depende de la longitud y características del patrón, así como del tamaño del corpus y del entorno de ejecución. Se recomienda:

- *Shift-And* cuando m es pequeño y se busca rendimiento máximo.
- *KMP* para patrones de longitud variable y entornos con recursos limitados.
- *Boyer–Moore Unicode* para patrones largos en corpora muy grandes donde las heurísticas amortizan su costo de preprocesamiento.

7. Conclusiones Finales

En este trabajo se han implementado y evaluado empíricamente tres algoritmos de búsqueda exacta de patrones en texto: Knuth–Morris–Pratt (KMP), Boyer–Moore adaptado a Unicode y Shift–And (para patrones de hasta 64 caracteres). A continuación se resumen los hallazgos y recomendaciones principales.

7.1. Eficiencia Algorítmica y Escalabilidad

- **KMP:** Garantiza tiempo lineal worst–case $\mathcal{O}(n + m)$ y presenta un comportamiento muy estable en todos los corpus. Las métricas muestran un conteo predecible de comparaciones y accesos al arreglo LPS. Es la opción más robusta cuando se requiere un límite estricto en el peor caso.
- **Boyer–Moore Unicode:** En la práctica, es el más veloz para textos largos y patrones moderados, pues sus heurísticas *bad-character* y *good-suffix* permiten saltos grandes. Aunque su peor caso teórico es $\mathcal{O}(nm)$, casi nunca se alcanza; el rendimiento medio resulta $\mathcal{O}(n)$ y las comparaciones totales suelen ser menores que en KMP.
- **Shift–And:** Muy eficiente para patrones cortos ($m \leq 64$), con complejidad práctica $\mathcal{O}(n)$ y un conteo de “caracteres procesados” directamente proporcional al tamaño del texto. Para patrones largos o textos muy grandes, el overhead de máscaras bit a bit lo hace menos competitivo frente a Boyer–Moore.

7.2. Explicación de Variaciones en los Tiempos

Los picos y ligeras variaciones observadas en los gráficos de tiempo pueden atribuirse a:

1. *Condiciones del sistema operativo*, p. ej. interferencias de la caché o planificador.
2. *Características particulares de los corpus* (distribución de caracteres, repeticiones) que penalizan más un algoritmo que otro.
3. *Estructuras auxiliares*, como la tabla dinámica de bad-character en Boyer–Moore, que en ocasiones requiere más trabajo de memoria.

7.3. Recomendaciones de Uso

- Para **textos grandes** y patrones de longitud media, *Boyer–Moore Unicode* es la opción preferible, por su excelente comportamiento promedio.
- Cuando se necesita una **garantía lineal** en el peor caso (p. ej. en sistemas críticos), *KMP* es la alternativa más segura.
- Para **patrones muy cortos** (hasta 64 caracteres), *Shift–And* ofrece un rendimiento competitivo y es sencillo de integrar.

7.4. Trabajo Futuro

- Extender *Shift–And* a patrones más largos usando bloques de 128/256 bits (SIMD).
- Mejorar la codificación Unicode en KMP y Boyer–Moore para admitir graphemes complejos.
- Incorporar algoritmos de búsqueda aproximada (p. ej. Levenshtein con bitap extensible).

Referencias

[1] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[3] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[4] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[5] The Unicode Consortium. *The Unicode Standard, Version 15.0 — Core Specification*. Chapter 3: Conformance, Section 3.11: Normalization Forms (NFC, NFD), 2023.