UNIVERSIDAD
Blas Pascal

“Proyecto Búsqueda-Compresión ”

Asignatura: ALGORITMOS Y ESTRUCTURAS DE DATOS II

Fecha: 24/09/25

Autores: RODRIGUEZ, RODRIGO EMMANUEL

ARMAS, FERNANDO GABRIEL

Profesor: Prof. GUTIERREZ, JULIO MARCELO

RESUMEN

El presente informe tiene carácter investigativo y se centra en el análisis de seis algoritmos y estructuras de datos fundamentales en las áreas de búsqueda de patrones y compresión de datos sin pérdida. Se estudian: Búsqueda por Fuerza Bruta, Knuth-Morris-Pratt (KMP), Rabin-Karp, Tries, Algoritmo de Huffman y Algoritmos de Ziv-Lempel (LZ77/LZ78).

Para cada uno se exponen su fundamento teórico, metodología, complejidad temporal y espacial, ventajas, desventajas y casos de uso reales. Además, se incluye una comparación estructurada, una conclusión integradora y recomendaciones prácticas para su aplicación en proyectos.

El objetivo de este trabajo es proporcionar al lector una visión completa que le permita comprender los principios, comparar alternativas y deducir cuál es más adecuada en diferentes escenarios de búsqueda y compresión.

1. Introducción

La búsqueda de patrones y la compresión de datos constituyen problemas centrales en la computación. Ambos tienen como objetivo optimizar recursos: el tiempo de ejecución, en el caso de la búsqueda, y el espacio de almacenamiento o el ancho de banda, en el caso de la compresión.

Históricamente, el primer enfoque fue la búsqueda por fuerza bruta, la cual, aunque simple y universal, es ineficiente en datos de gran tamaño. Con el tiempo surgieron algoritmos más refinados como Knuth-Morris-Pratt (KMP), que evita reprocesamientos innecesarios, y Rabin-Karp, que usa hashing para acelerar la búsqueda de múltiples patrones.

Por otra parte, el desafío de almacenar y transmitir información motivó la creación de técnicas de compresión como Huffman, que asigna códigos óptimos según frecuencias, y LZ77/LZ78, que detectan patrones repetidos en los datos y los reemplazan por referencias compactas.

El estudio de estos algoritmos no solo es relevante por su aplicación directa en motores de búsqueda, compresores, sistemas de archivos y protocolos de red, sino también porque sirven como base teórica para métodos más modernos como los algoritmos de compresión de texto en inteligencia artificial o los sistemas de indexación en bases de datos.

2. Algoritmos de Busqueda de Patrones

a. Búsqueda por Fuerza Bruta

La búsqueda por fuerza bruta es el algoritmo más simple y directo para localizar un patrón dentro de un texto. Constituye el punto de partida teórico para comprender los problemas de búsqueda de cadenas y sirve como base para algoritmos más eficientes como KMP o Rabin-Karp.

Concepto General

Dado un texto T de longitud n y un patrón P de longitud m, el algoritmo realiza una búsqueda exhaustiva de la siguiente manera:

- **Alineación del patrón:** se coloca P sobre T en cada posición posible, desde i = 0 hasta i = n - m.
- **Comparación carácter por carácter:** se verifica si P coincide con la subcadena de T en esa posición.
- **Reporte de coincidencia:** si los m caracteres coinciden, se registra la posición i como una coincidencia.
- **Desplazamiento:** si hay una discrepancia, el patrón se desplaza una posición a la derecha y se repite el proceso hasta recorrer todo el texto.

PseudoCódigo:

```
Entrada:
    Texto T[0..n-1], Patrón P[0..m-1]
Proceso:
    Para i ← 0 hasta n - m:
        j ← 0
        Mientras j < m y T[i + j] = P[j]:
            j ← j + 1
        Si j = m:
            Reportar "Coincidencia encontrada en posición i"
```

Este pseudocódigo muestra claramente que el patrón se compara desde cada desplazamiento posible, y en el momento en que se encuentra una coincidencia completa, se notifica la posición de inicio.

Ventajas

- **Extremadamente simple de implementar:** su lógica es directa y se puede codificar en pocas líneas.
- **Sin preprocesamiento ni estructuras extra:** no requiere tablas auxiliares ni hashing.
- **Versatilidad:** funciona con cualquier tipo de datos (texto, listas, secuencias binarias, ADN).

Desventajas

- **Ineficiencia en textos grandes:** su tiempo de ejecución es elevado para cadenas largas o patrones grandes.
- **Repite comparaciones innecesarias:** no aprovecha información de coincidencias parciales.
- **Poca escalabilidad:** no es adecuado para sistemas en tiempo real o búsqueda masiva de patrones

b. Algoritmo Knuth-Morris-Pratt (KMP)

El algoritmo Knuth–Morris–Pratt (KMP) es uno de los métodos más influyentes para la búsqueda de patrones en cadenas de texto. Fue desarrollado en 1977 por Donald Knuth, James H. Morris y Vaughan Pratt, y se convirtió en el primer algoritmo que logra realizar esta tarea en tiempo lineal O(n + m).

A diferencia de la búsqueda por fuerza bruta, KMP evita comparaciones redundantes utilizando un preprocesamiento del patrón para saber cuánto puede desplazarse en cada falla, sin volver a revisar caracteres ya comparados.

Concepto General

La idea principal de KMP es que, cuando ocurre un desajuste entre el patrón y el texto, no es necesario reiniciar la búsqueda desde cero:
Se aprovecha la información de prefijos y sufijos coincidentes para "saltar" posiciones en el texto.
Para lograr esto, se construye una tabla de fallos o arreglo de prefijos (también llamado lps por longest proper prefix which is also suffix), que indica para cada posición del patrón cuánto puede desplazarse en caso de un fallo.

Pseudocódigo:

```
Entrada:
    Texto T[0..n-1], Patrón P[0..m-1]

1. Construcción del arreglo lps[]
    lps[0] ← 0
    i ← 1, len ← 0
    mientras i < m:
        si P[i] = P[len]:
            len ← len + 1
            lps[i] ← len
            i ← i + 1
        sino si len ≠ 0:
            len ← lps[len - 1]
        sino:
            lps[i] ← 0
            i ← i + 1

2. Búsqueda
    i ← 0, j ← 0
    mientras i < n:
        si P[j] = T[i]:
            i ← i + 1
            j ← j + 1
        si j = m:
            reportar "Coincidencia en posición i - j"
            j ← lps[j - 1]
        sino si i < n y P[j] ≠ T[i]:
            si j ≠ 0:
                j ← lps[j - 1]
            sino:
                i ← i + 1
```

Este pseudocódigo muestra cómo el uso de lps[ ] evita reiniciar la búsqueda desde la posición siguiente en el texto, mejorando considerablemente la eficiencia.

Ventajas

- **Tiempo de ejecución lineal**: O(n + m), mucho más eficiente que fuerza bruta en el peor caso.

- **Evita comparaciones redundantes:** reutiliza información de coincidencias parciales.
- **Preprocesamiento eficiente:** el arreglo lps se calcula en tiempo lineal.

Desventajas

- **Mayor complejidad de implementación:** requiere construir la tabla lps, lo que puede ser confuso para principiantes.
- **Mayor uso de memoria:** necesita un arreglo auxiliar de longitud m.
- **No es el más rápido para textos muy cortos:** la sobrecarga del preprocesamiento no siempre compensa.

c. Algoritmo Rabin-Karp

El algoritmo Rabin–Karp es un método de búsqueda de patrones basado en hashing. Fue desarrollado por Richard M. Karp y Michael O. Rabin en 1987, y es especialmente útil cuando se necesita buscar múltiples patrones en un mismo texto o cuando la eficiencia es clave en búsquedas de gran escala.

Su principal idea es convertir el patrón y cada subcadena del texto en un valor hash y comparar estos valores en lugar de comparar carácter por carácter directamente. Si los hashes coinciden, entonces se realiza una verificación detallada de la subcadena para confirmar que no sea una colisión (falso positivo).

Concepto General

Dado un texto T de longitud n y un patrón P de longitud m, el algoritmo realiza los siguientes pasos:

- **Cálculo del hash del patrón:** se obtiene un valor numérico único (hash) para P.
- **Cálculo del hash inicial del texto:** se obtiene el hash de la primera subcadena de T de longitud m.
- **Desplazamiento con hash rodante:** se va "deslizando" el patrón por el texto y actualizando el hash de forma eficiente, sin volver a calcularlo desde cero.
- **Comparación de hashes:** si el hash de la subcadena coincide con el hash de P, se comparan los caracteres para confirmar.

- **Reporte de coincidencia:** si los caracteres coinciden, se registra la posición.

Pseudocódigo:

```
Entrada:
    Texto T[0..n-1], Patrón P[0..m-1]

1. Calcular hash_P ← hash(P)
2. Calcular hash_T ← hash(T[0..m-1])

3. Para i ← 0 hasta n - m:
    si hash_T = hash_P:
        si T[i..i+m-1] = P:
            Reportar "Coincidencia en posición i"
    si i < n - m:
        Actualizar hash_T usando hash rodante
```

El uso de hash rodante es la clave de la eficiencia, ya que permite calcular el hash de la subcadena siguiente en tiempo O(1) a partir del hash de la actual.

Ventajas

- **Eficiente para múltiples patrones:** se puede precomputar un hash para cada patrón y comparar contra el texto de una sola pasada.
- **Evita comparaciones carácter por carácter en la mayoría de los casos:** las comparaciones directas solo ocurren cuando los hashes coinciden.
- **Fácil de adaptar a distintos alfabetos o tipos de datos:** basta con definir una función hash apropiada.

Desventajas

- **Posibilidad de colisiones:** diferentes subcadenas pueden tener el mismo hash, lo que obliga a hacer comparaciones adicionales para verificar.
- **Rendimiento sensible a la calidad de la función hash:** si es mala, habrá muchas colisiones y el algoritmo se degrada a O(n·m).
- **Implementación más compleja que fuerza bruta:** requiere manejo de aritmética modular y cálculo eficiente de hashes.

d. Tries

Los Tries, también conocidos como árboles prefijo, son estructuras de datos especializadas en almacenar y buscar cadenas de manera eficiente. Fueron introducidos por René de la Briandais en 1959 y se utilizan ampliamente en autocompletado, correctores ortográficos y motores de búsqueda.

A diferencia de los algoritmos basados en comparación carácter por carácter (como Fuerza Bruta, KMP o Rabin-Karp), un trie organiza los patrones en un árbol jerárquico, donde cada nodo representa un carácter y cada camino desde la raíz hasta un nodo terminal representa una palabra completa.

Concepto General:

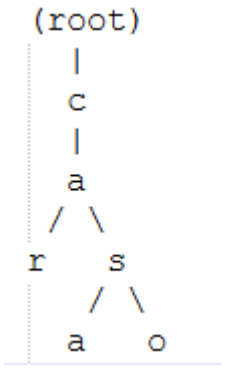
Dado un conjunto de palabras W = {w1, w2, ..., wn}, un trie almacena estas palabras de la siguiente manera:

- **Raíz vacía:** el trie comienza con un nodo raíz que no contiene ningún carácter.
- **Nodos por carácter:** cada nivel del árbol representa la posición de un carácter en la palabra.
- **Marcadores de final de palabra:** se utiliza un indicador para señalar que un camino desde la raíz hasta un nodo forma una palabra completa.
- **Inserción y búsqueda eficiente:** se recorren los nodos según los caracteres de la palabra, alcanzando complejidad O(L), donde L es la longitud de la palabra.

Ejemplo Visual

Supongamos que almacenamos las palabras: "car", "casa", "caso"

- Cada nodo representa un carácter.
- Los nodos terminales indican el final de una palabra.
- Para buscar "casa", se recorre c → a → s → a.



Pseudocódigo para Inserción

```
Función Insertar(trie, palabra):
    nodo ← trie.raíz
    Para cada carácter ch en palabra:
        si nodo.hijos[ch] no existe:
            nodo.hijos[ch] ← nuevo Nodo()
        nodo ← nodo.hijos[ch]
    nodo.finPalabra ← verdadero
```

Pseudocódigo para Búsqueda

```
Función Buscar(trie, palabra):
    nodo ← trie.raíz
    Para cada carácter ch en palabra:
        si nodo.hijos[ch] no existe:
            retornar falso
        nodo ← nodo.hijos[ch]
    retornar nodo.finPalabra
```

Ventajas

- **Búsqueda rápida por prefijos:** O(L), independiente del número total de palabras.
- **Autocompletado eficiente:** permite listar todas las palabras que comparten un prefijo común.
- **Soporta alfabetos grandes o conjuntos de caracteres variados:** adaptable a texto, secuencias binarias o ADN.

Desventajas

- **Alto consumo de memoria:** cada nodo puede tener muchos hijos, lo que aumenta el uso de memoria en grandes vocabularios.
- **Inserción y eliminación más complejas:** requiere recorrer varios niveles y actualizar referencias.
- **No siempre óptimo para textos pequeños o pocos patrones:** en esos casos, un array simple puede ser más eficiente.

3. Algoritmos de Compresión

a. Algoritmo de Huffman

El Algoritmo de Huffman es uno de los métodos de compresión sin pérdida más conocidos y utilizados. Fue desarrollado por David A. Huffman en 1952 y se basa en la idea de asignar códigos de longitud variable a los símbolos de un alfabeto, de manera que los símbolos más frecuentes tengan códigos más cortos, optimizando así el tamaño final de los datos.

Este algoritmo es ampliamente usado en formatos de compresión como ZIP, GZIP, MP3, JPEG y en protocolos de comunicación para reducir el tamaño de los mensajes.

Concepto General

Dado un conjunto de símbolos con sus frecuencias de aparición, el algoritmo construye un árbol binario de Huffman donde:

- Cada símbolo es una hoja en el árbol.
- La longitud del camino desde la raíz a cada hoja representa el número de bits de su código.
- Los símbolos más frecuentes se ubican más cerca de la raíz (códigos más cortos).
- Los símbolos menos frecuentes se ubican más lejos (códigos más largos).

Etapas del Algoritmo

1. **Contar frecuencias:** se calcula cuántas veces aparece cada símbolo.
2. **Crear nodos hoja:** cada símbolo se convierte en un nodo con peso igual a su frecuencia.
3. **Construir árbol:**
    - Se seleccionan los dos nodos de menor frecuencia.
    - Se unen en un nuevo nodo padre con peso igual a la suma de las frecuencias.
    - El proceso se repite hasta obtener un solo nodo raíz.
4. **Asignar códigos:**
    - A cada rama izquierda se asigna 0, a cada rama derecha 1.
    - El código de cada símbolo es la concatenación de bits en el camino desde la raíz hasta la hoja.

Ejemplo Paso a Paso

Supongamos el texto: "CASA"

1. Contar Frecuencias

Símbolo	Frecuencia
C	1
A	2

S	1

2. Crear Nodos Hoja

Cada símbolo se representa como un nodo con peso igual a su frecuencia:

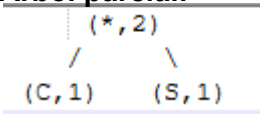
- Nodo(C,1)
- Nodo(A,2)
- Nodo(S,1)

3. Construcción del Árbol de Huffman

a. Paso 1: Seleccionar los dos nodos con menor frecuencia:

- C (1) y S (1)
- Crear nodo padre con peso 1+1=2

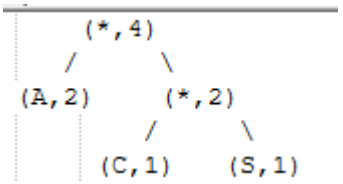
Árbol parcial:



b. **Paso 2:** Unir el nodo recién creado con el nodo A (2):

- Nodo padre con peso 2+2=4 → raíz final del árbol

Árbol completo:



4. Asignación de Códigos

a. Se asigna 0 a las ramas izquierdas y 1 a las derechas:

- **A** = 0
- **C** = 10
- **S** = 11

5. Codificación de "CASA"

Carácter	Código
C	10
A	0
S	11
A	0

Resultado final:
C A S A → 10 0 11 0 → 100110

6. **Comparación con Codificación Fija**

   Si usamos un código de longitud fija de 2 bits por símbolo (porque hay 3 símbolos), se necesitarían 4 × 2 = 8 bits.
   Con Huffman, usamos 6 bits, logrando una compresión de 25%.

Pseudocódigo

```
Función Huffman(Símbolos):
    crear cola prioritaria con nodos de cada símbolo y frecuencia
    mientras haya más de un nodo en la cola:
        x ← extraer nodo con menor frecuencia
        y ← extraer nodo con menor frecuencia
        z ← nuevo nodo con frecuencia = x.frecuencia + y.frecuencia
        z.izq ← x
        z.der ← y
        insertar z en la cola
    retornar nodo restante (raíz del árbol)
```

Ventajas

- **Compresión óptima:** genera códigos de longitud mínima para un conjunto de frecuencias dado.
- **Sin pérdida de información:** la decodificación es exacta, no se pierde contenido.
- **Amplia aplicación:** base de muchos algoritmos de compresión modernos.

Desventajas

- **Necesidad de almacenar el árbol:** para decodificar es necesario conocer la estructura de Huffman.
- **No es óptimo en flujos dinámicos:** si las frecuencias cambian constantemente, se debe reconstruir el árbol.
- **No siempre útil en datos aleatorios:** si todos los símbolos aparecen con igual probabilidad, no hay ganancia de compresión.

b. Algoritmos Ziv-Lempel (LZ77/LZ78)

i. Algoritmo LZ77

El LZ77 es el primero de la familia de algoritmos de compresión de Ziv-Lempel, publicado en 1977.

Su idea central es aprovechar las repeticiones de cadenas dentro de un texto reemplazando cada aparición repetida por una referencia a su posición previa.

Concepto General

LZ77 utiliza una ventana deslizante que divide el texto en dos regiones:

- **Buffer de búsqueda:** contiene los caracteres ya procesados.
- **Buffer de entrada:** contiene los caracteres que quedan por procesar.

El algoritmo busca la coincidencia más larga entre el prefijo del buffer de entrada y alguna subcadena dentro del buffer de búsqueda. Cuando la encuentra, emite una tupla:
(Distancia,Longitud,Siguiente_Caracter)

- **Distancia:** cuántos caracteres atrás comienza la coincidencia.
- **Longitud:** cuántos caracteres coinciden.
- **Siguiente_Caracter:** el carácter nuevo que sigue a la coincidencia.

Ejemplo Paso a Paso – LZ77

Texto: "**ABABABA**"

1. **Inicio:**
   Ventana vacía → no hay coincidencias.
   Salida: (0,0,'A') → escribir "A"

2. **Siguiente carácter:**
   No hay coincidencias previas para "B".
   Salida: (0,0,'B') → escribir "B"
3. **Desde la tercera posición:**
   Buffer de búsqueda = "AB", buffer de entrada comienza en "A".
   Se encuentra coincidencia de 2 caracteres: "AB".
   Salida: (2,2,'A') → mirar 2 atrás, copiar "AB", añadir "A".

   Resultado final:
   [(0,0,′A′),(0,0,′B′),(2,2,′A′)]
   Texto reconstruido: "**ABABABA**"

Ventajas de LZ77

- Compresión eficiente para textos con muchas repeticiones.
- No requiere un diccionario externo: las referencias apuntan al propio texto.
- Es la base de compresores modernos como Deflate (usado en ZIP, PNG, GZIP).

Desventajas de LZ77

- Si el texto no tiene repeticiones, el beneficio es mínimo.
- La búsqueda de coincidencias largas puede ser costosa en términos de tiempo.
- La ventana debe tener un tamaño adecuado: muy pequeña → baja compresión, muy grande → alto costo computacional.

ii. Algoritmo LZ78

El LZ78 fue publicado en 1978 como una mejora de LZ77.
En lugar de usar una ventana, construye un diccionario dinámico a medida que procesa el texto, asignando índices a las secuencias detectadas.

Concepto General

LZ78 procesa el texto carácter por carácter formando la cadena más larga que ya exista en el diccionario. Luego emite un par:

(Indice, Siguiente_Caracter)

- **Indice**: número de la secuencia ya conocida (0 si es nueva).
- **Siguiente_Caracter:** carácter que extiende la secuencia para formar una nueva.

Luego agrega la nueva cadena al diccionario.

Ejemplo Paso a Paso – LZ78

Texto: "**ABABABA**"

Paso	Subcadena Actual	Indice en Diccionario	Par Emitido	Diccionario Resultante
1	"A"	0	(0,'A')	{1:"A"}
2	"B"	0	(0,'B')	{1:"A", 2:"B"}
3	"A" + "B"	1	(1,'B')	{1:"A", 2:"B", 3:"AB"}
4	"A" + "A"	1	(1,'A')	{1:"A", 2:"B", 3:"AB", 4:"AA"}
5	"B" + "A"	2	(2,'A')	{1:"A", 2:"B", 3:"AB", 4:"AA", 5:"BA"}

Salida final:
[(0,′A′),(0,′B′),(1,′B′),(1,′A′),(2,′A′)]

Texto reconstruido: "**ABAABABA**"

Ventajas de LZ78

- No necesita una ventana fija: puede aprender patrones de cualquier longitud.
- Construye un diccionario reutilizable, ideal para grandes volúmenes de datos.
- Es la base de compresores como LZW (usado en GIF y TIFF).

Desventajas de LZ78

- Requiere almacenamiento extra para el diccionario.
- Puede consumir mucha memoria si el texto es largo y variado.

- Necesita reiniciar o limitar el tamaño del diccionario para evitar sobrecarga.

Resumen de Diferencias Clave

Característica	LZ77	LZ78
Estructura usada	Ventana deslizante	Diccionario dinámico
Salida	(distancia, longitud, símbolo)	(índice, símbolo)
Ventaja	No necesita diccionario externo	Aprende patrones de cualquier longitud
Desventaja	Menos eficiente si ventana pequeña	Diccionario puede crecer mucho
Uso actual	Base de ZIP, PNG, GZIP	Base de LZW, usado en GIF

4. Comparación

Algoritmo	Tipo	Casos de Uso Recomendados	Ventajas	Desventajas
Fuerza Bruta	Búsqueda de patrones	Textos cortos, casos didácticos	Extremadamente simple, sin estructuras auxiliares	Muy ineficiente en textos largos, no escala
Knuth-Morris-Pratt (KMP)	Búsqueda de patrones	Búsqueda en grandes textos donde se	Eficiente, evita comparaciones redundantes	Necesita preprocesamiento del patrón

		repite el patrón		
Rabin-Karp	Búsqueda de patrones	Búsqueda de múltiples patrones a la vez	Permite encontrar varios patrones de forma eficiente	Sensible a colisiones de hash
Tries	Estructura de datos para búsqueda	Autocompletado, correctores ortográficos, diccionarios	Búsqueda muy rápida, no depende de n	Alto consumo de memoria, requiere construcción previa
Huffman	Compresión de datos	Compresión óptima sin pérdidas (texto, imágenes, datos)	Genera códigos de longitud mínima, óptimo para frecuencias conocidas	Necesita conocer las frecuencias antes de codificar
LZ77	Compresión de datos	Compresión de datos con repetición local (ZIP, PNG)	No necesita diccionario externo, eficiente en textos repetitivos	Ineficiente si no hay repeticiones, depende de tamaño de ventana
LZ78	Compresión de datos	Compresión de grandes textos o datos con patrones complejos (GIF, TIFF vía LZW)	Aprende patrones de longitud arbitraria	Mayor consumo de memoria

Análisis Comparativo

Algoritmos de búsqueda (Fuerza Bruta, KMP, Rabin-Karp, Tries)

- Fuerza Bruta es el más simple pero el menos eficiente.
- KMP optimiza el proceso usando información de coincidencias parciales.
- Rabin-Karp destaca para búsquedas de múltiples patrones gracias a su hashing.
- Tries es el más rápido para consultas múltiples, pero consume más memoria.

Algoritmos de compresión (Huffman, LZ77, LZ78)

- Huffman es ideal para compresión estática (cuando conocemos la frecuencia de símbolos).
- LZ77 y LZ78 se adaptan dinámicamente al texto, siendo más versátiles.
- LZ77 funciona mejor con repeticiones cercanas (locales), mientras que LZ78 aprende patrones de cualquier longitud y los reutiliza.

5. Conclusión

Los algoritmos analizados muestran la evolución de la informática en dos dimensiones clave: búsqueda y compresión. Desde el método exhaustivo de fuerza bruta hasta soluciones inteligentes como LZ77, el progreso ha sido hacia más eficiencia y menor complejidad.

Para búsqueda, KMP es el algoritmo determinista más sólido, mientras que Rabin-Karp ofrece ventajas en búsqueda múltiple. Para compresión, Huffman sigue siendo óptimo en entornos estáticos, pero LZ77/LZ78 se imponen en aplicaciones dinámicas y de streaming.