# Algoritmos de búsqueda de subcadenas para encontrar semejanzas en cadenas numéricas


TRABAJO DE FIN DE GRADO DEL DOBLE GRADO EN INGENIERÍA INFORMÁTICA - MATEMÁTICAS

**Beatriz Coronado Sanz**

**Universidad Complutense de Madrid**

**Septiembre 2018**

Documento maquetado con TEXIS v.1.0.

Este documento está preparado para ser imprimido a doble cara.

# Algoritmos de búsqueda de subcadenas para encontrar semejanzas en cadenas numéricas

*Memoria que presenta para optar al título de Graduada en Ingeniería Informática y Matemáticas*

**Beatriz Coronado Sanz**

*Dirigida por la profesora*

**María Isabel Pita Andreu**

*Y el profesor*

**José Alberto Verdejo López**

**Universidad Complutense de Madrid**

**Septiembre 2018**

*A mis padres*

# Agradecimientos

Todo lo que empieza tiene que acabar, o al menos eso dicen. Hace 5 años empecé este doble grado y me parece increíble que llegue ya a su fin. Han sido años muy duros, con muchas alegrías pero también muchas tristezas. Ha habido asignaturas que parecían imposibles de sacar y otras que hubiera seguido dándolas un cuatrimestre más o dos solo para seguir aprendiendo.

Llegué sin saber nada de informática y me voy ahora con grandes conocimientos sobre ella. Muy feliz por haber descubierto una rama que no conocía y que seguramente sea clave en mi futuro. Mi amor por las matemáticas no ha cambiado pero ahora sé muchas más cosas y me voy con el deseo de seguir profundizando en la parte que se mezcla con la programación que tanto me gusta.

Pero aunque la universidad me ha dado muchos conocimientos, lo más importante que me llevo de estos años son las personas que he conocido y a las que quiero agradecer por todo lo que han hecho por mi.

En primer lugar, y para que no se me olvide, a todos los profesores que me han dado clase, ya sea en la facultad de Informática o en la de Matemáticas. Sin vosotros no habría aprendido nada nuevo y aunque hay profes y profes el balance que me llevo es positivo. Intentaré aplicar lo que me habéis enseñado todos estos años en mi vida aunque no prometo nada.

A mis amigas del colegio Alba e Iria que, aunque nos hemos visto menos, habéis seguido conmigo y me habéis apoyado cuando las cosas no salían del todo bien.

A mis compañeras de clase Elena y Patri que, aunque aún os queda un año para terminar, sé que en nada escribiréis una memoria como esta y acabaréis igual que yo. Sin vosotras el día a día en la universidad no habría sido lo mismo. Sobre todo sin ti Elena, eterna compañera de prácticas. No sé que voy a hacer ahora sin nuestras prácticas infinitas con mil líneas de comentarios.

A las mejores amigas que he podido conseguir gracias a mi deporte favorito, el fútbol. Reich, Lore, Alba me refiero a vosotras. Mi vida no sería la misma sin haberos conocido. Gracias por todo lo que habéis hecho por mi, por leerme y aconsejarme cuando he tenido algún problema. Nuestra vida va a cambiar a partir de ahora pero sé que nuestra amistad no.

Y por último, pero no menos importante, a mi familia. A mis padres por apoyarme en todo momento, confiar en mí y darme la libertad para tomar mis propias decisiones me equivoqué con ellas o no. Y a mis dos hermanos pequeños, intentaré devolveros todo lo que habéis hecho por mi estos años que estáis en la uni.

En definitiva, gracias a todos los que habéis formado parte de mi vida estos años. Se termina una etapa pero empieza una nueva y espero que sea igual de memorable que esta que he vivido.

# Resumen

El objetivo de este TFG es implementar distintos algoritmos de búsqueda de subcadenas en ciertas cadenas numéricas dadas. Los algoritmos se aplican a medidas de las variaciones que experimentan los diámetros de los troncos de los árboles a lo largo del día. Se implementan dos tipos de algoritmos: algoritmos generales de cadenas que buscan distintas características que se producen en el tronco a lo largo del tiempo, como por ejemplo el intervalo de tiempo más largo en el que el tronco se ha expandido sin comprimirse, o que comparan los intervalos en los que se producen estas características en varios árboles y algoritmos de búsqueda de subcadenas que buscan un cierto patrón en las cadenas que tenemos. Como los datos son muy variables buscamos la tendencia a crecer o decrecer de las medidas, permitiendo ciertas discrepancias. Los algoritmos implementados, por lo tanto, son en su mayor parte aproximados, permitiendo que algunos valores no se ajusten exactamente en las secuencias que se buscan.

Se implementan ambos tipos de algoritmos en una aplicación gráfica, en donde el usuario puede elegir qué tipo de algoritmo usar y las características con las que lo usa.

Al final se realiza un estudio de los tiempos de los algoritmos de búsqueda de subcadenas y se comentan los resultados más importantes que se obtienen. Primero se estudian los tiempos de los algoritmos aplicados a ejemplos de árboles reales y luego se crean varios árboles ficticios para estudiar ciertas características del algoritmo de Boyer Moore. Se estudia la mejora de los tiempos del algoritmo exacto de Boyer Moore frente al aproximado con patrones cíclicos y la igualdad cuando el tamaño del patrón es muy grande. Además se comprueba que el algoritmo Shift Add mantiene unos tiempos constantes respecto al tamaño del patrón y el número de discrepancias que se admiten.

**Palabras clave**: algoritmo de búsqueda de subcadenas, coincidencia aproximada, algoritmo Boyer Moore, algoritmo Shift Add.

# Abstract

The goal of this FDP (Final Degree Project) is to implement different string-searching algorithms in given numerical strings. The algorithms are applied as the diameter of the trunks of the trees changes along the day. Two types of algorithms are implemented: general algorithms of string that search different characteristics produced in the trunk as the time goes by, for example the largest interval of time in which the trunk has expanded without compressing, or compare the intervals in which this characteristics appear in some trees and string-searching algorithms which look for a certain pattern in the strings. Since the data changes a lot we search the tendency of growth or shrinking of the measurements letting some mismatches. Due to this the implemented algorithms are in the most part approximated, allowing that some values are not exactly adjusted in the strings.

Both algorithm are implemented in a graphical application where the user can choose the desirable algorithm and its characteristics.

At the end it is done a study of the execution time of the string-searching algorithms and the most relevant results are commented. Firstly it is studied the execution time of the algorithms applied to real trees and then virtual trees are created to study certain characteristics of the Boyer Moore algorithm. The improvement of the exact Boyer Moore algorithm times is compared to the approximated Boyer Moore algorithm with cyclic patterns and equality when there is a larger size of the pattern. Additionally it is checked that the Shift Add algorithm maintains constant execution times with respect to the size of the pattern and the number of mismatches admitted.

**Keywords**: string matching algorithm, approximate matching, Boyer Moore algorithm, Shift Add algorithm.

# Índice

**Agradecimientos** vii

**Resumen** ix

**Abstract** xi

**1. Introducción** 1

**2. Algoritmos** 3
2.1. Algoritmos generales de cadenas . . . 3
2.1.1. Búsqueda de intervalos de tamaño fijo que cumplen una propiedad . 3
2.1.2. Búsqueda del segmento máximo que cumple una propiedad . . . . . 4
2.2. Algoritmos de búsqueda de subcadenas . . . 6
2.2.1. Distancia de Hamming . . . 6
2.2.2. Algoritmo de Fuerza Bruta . . . 6
2.2.3. Algoritmo Boyer Moore . . . 7
2.2.4. Algoritmo Shift-Add . . . 14

**3. Aplicación** 19
3.1. Explicación del problema . . . 19
3.2. Descripción de la aplicación. Usuario . . . 20
3.2.1. Funciones de la aplicación . . . 21
3.2.2. Campos en la aplicación . . . 25
3.2.3. Ejemplos de uso de la aplicación . . . 26
3.3. Descripción de la aplicación. Implementación . . . 31
3.3.1. Entrada de Datos . . . 31
3.3.2. Modelo-Vista-Controlador . . . 33
3.3.3. Modelo de la aplicación . . . 34
3.3.4. Implementación de las funciones de la aplicación . . . 35
3.3.5. Vista de la aplicación . . . 40
3.3.6. Controlador de la aplicación . . . 43
3.3.7. Unión de los tres componentes . . . 44
3.3.8. Gráficas de los árboles . . . 46

**4. Análisis de tiempos** 49

4.1. Implementación . . . . . 49
4.1.1. Clase ControlTiempos . . . . . 49
4.2. Experimentos con árboles . . . . . 50
4.2.1. Tiempos para patrones de tamaño 3 . . . . . 50
4.2.2. Tiempos para patrones de tamaño 5 . . . . . 53
4.2.3. Tiempos para patrones de tamaño 10 . . . . . 55
4.2.4. Resumen comparativo de los tres tipos de patrones sobre los datos de un árbol . . . . . 57
4.3. Experimentos adicionales . . . . . 57
4.3.1. Patrones cíclicos . . . . . 57
4.3.2. Patrones con alfabetos grandes . . . . . 70

**5. Conclusiones** **79**

**Bibliografía** **81**

# Índice de figuras


2.1. Pseudo-código del algoritmo de Fuerza Bruta [3] . . . . . 7
2.2. Regla del sufijo bueno, el caracter $x$ de $T$ discrepa del caracter $y$ de $P$. Los caracteres $y$ y $z$ son distintos, por lo que $z$ igual coincide con $x$ . . . . . 10
2.3. Pseudo-código del cálculo de $L^{'}(i)$ [2] . . . . . 11
2.4. Pseudo-código del algoritmo Boyer Moore exacto [2] . . . . . 12
2.5. Pseudo-código del preprocesado del algoritmo Boyer Moore aproximado [8] 14
2.6. Pseudo-código del algoritmo Boyer Moore aproximado [8] . . . . . 15
2.7. Pseudo-código del preprocesado del algoritmo Shift-Add [6] . . . . . 17
2.8. Pseudo-código del algoritmo Shift-Add [6] . . . . . 18

3.1. Ventana de la aplicación . . . . . 20
3.2. Función *Data of a tree* . . . . . 27
3.3. Desplegable con los árboles posibles . . . . . 27
3.4. Resultados de la opción *Data of a tree* . . . . . 28
3.5. Función *Data comparison* . . . . . 28
3.6. Desplegable con los árboles posibles . . . . . 29
3.7. Resultados de la opción *Data comparison* . . . . . 29
3.8. Función *Growth of a tree trunk* . . . . . 30
3.9. Resultados de la opción *Growth of a tree trunk* . . . . . 30
3.10. Función *Patterns in a tree* . . . . . 31
3.11. Resultados de la opción *Patterns of a tree* . . . . . 31
3.12. Datos de los árboles en Excel . . . . . 32
3.13. Gráfica de nuestra aplicación con 3 árboles . . . . . 47

4.1. Gráfico con los tiempos de los algoritmos para el arbol *e16-A* con 0 discrepancias . . . . . 58
4.2. Gráfico con los tiempos de los algoritmos para el arbol e16-A con 0 discrepancias . . . . . 59
4.3. Gráfico con los tiempos de los algoritmos para el vector Nulo 1 . . . . . 62
4.4. Gráfico con los tiempos de los algoritmos para el vector Nulo 1 sin FB . . . 63
4.5. Gráfico con los tiempos de los algoritmos para el vector Alternado 1 . . . . 64
4.6. Gráfico con los tiempos de los algoritmos para el vector Alternado 1 sin FB 65
4.7. Gráfico con los tiempos de los algoritmos para el vector Alternado 1 . . . . 68
4.8. Gráfico con los tiempos de los algoritmos para el vector Alternado 1 sin FB 69

4.9. Gráfico con los tiempos de los algoritmos para el vector Nulo 1 . . . . . 70
4.10. Gráfico con los tiempos de los algoritmos para el vector Nulo 1 sin FB . . . 71
4.11. Gráfico con los tiempos de los algoritmos para patrones con muchas ocurrencias en distintos alfabetos . . . . . 75
4.12. Gráfico con los tiempos de los algoritmos para el patrón `50 51 52`... . . . 77

# Índice de Tablas


2.1. Regla de sufijo bueno para el patrón `1 2 4 1 2 3 1 2` . . . . . 9
2.2. Cálculo de $Z_{pos}$, $N_{pos}$, $L(pos)$, $L^{'}(pos)$ y $l^{'}(pos)$ para el patrón `2 1 2 4 1 2 4 1 2` . . . . . 11
2.3. Tabla $d_k$ para el patrón `1 -1 1 -1 1` . . . . . 13
2.4. Tabla $t$ para el patrón `1 -1 1 1` . . . . . 17

4.1. Tiempos para patrones de tamaño 3 en el árbol *e16-A* . . . . . 51
4.2. Tiempos para patrones de tamaño 3 en el árbol *e16-A* . . . . . 51
4.3. Tiempos para patrones de tamaño 3 en el árbol *e23-A* . . . . . 52
4.4. Tiempos para patrones de tamaño 3 en el árbol *e23-A* . . . . . 52
4.5. Tiempos para patrones de tamaño 5 en el árbol *e16-A* . . . . . 53
4.6. Tiempos para patrones de tamaño 5 en el árbol *e16-A* . . . . . 54
4.7. Tiempos para patrones de tamaño 10 en el árbol *e16-A* . . . . . 55
4.8. Tiempos para patrones de tamaño 10 en el árbol *e16-A* . . . . . 56
4.9. Tiempos para el patrón cíclico `0` . . . . . 60
4.10. Tiempos para el patrón cíclico `0` . . . . . 61
4.11. Tiempos para el patrón cíclico `2 -2` . . . . . 66
4.12. Tiempos para el patrón cíclico `2 -2` . . . . . 67
4.13. Tiempos para patrones anidados de la forma `10 -10` . . . . . 72
4.14. Tiempos para patrones anidados de la forma `20 -20` . . . . . 73
4.15. Tiempos para patrones anidados de la forma `100 -100` . . . . . 74
4.16. Tiempos para el patrón `50 51 52 53...` . . . . . 76

# Capítulo 1

# Introducción

La idea de este trabajo surge de la necesidad de informatizar y justificar rigurosamente los resultados de una serie de estudios realizados por un grupo de investigadores de la E.T.S. de Ingeniería de Montes, Forestal y del Medio Natural de la Universidad Politécnica de Madrid. En particular, estos investigadores miden el diámetro del tronco de una serie de árboles cada 30 segundos e intentan relacionar las oscilaciones que se obtienen en las medidas a lo largo del día con los fenómenos físicos a los que están sometidos los árboles. Ejemplos de estos fenómenos son la salida y puesta del sol, el momento en el que se riega al árbol o el momento en el que se abre la ventana del invernadero.

Algunos de los problemas de estos investigadores que vamos a solucionar son:

- Encontrar los intervalos en los que crece o decrece un árbol durante el día y compararlo con el crecimiento y decrecimiento de otros árboles.
- Buscar los momentos del día en los que un árbol discrepa en sus medidas frente a otro árbol.
- Estudiar el crecimiento del tronco de un árbol durante varios días y compararlo con otros árboles.
- Buscar pequeños momentos concretos en un árbol en los que se produce un fenómeno físico al mismo tiempo y contrastar si frente al mismo estímulo los árboles se comportan de la misma forma.

Para los primeros tres puntos usaremos distintos algoritmos generales de cadenas según la característica del árbol que querramos estudiar. Para el último punto estudiaremos distintos tipos de algoritmos para la búsqueda de subcadenas en un texto y aplicaremos alguno de los algoritmos estudiados a nuestro problema. Todos los algoritmos se aplicarán al conjunto de medidas obtenidas en los troncos de estos árboles durante un mes.

Además de todo esto, haremos una aplicación específica para permitir a los investigadores aplicar los algoritmos que hemos desarrollado y estudiado a los problemas que tienen de una manera más sencilla.

Por último, haremos un estudio de tiempos de los distintos algoritmos de búsqueda de subcadenas aplicados para averiguar cuál de ellos es el más efectivo para resolver nuestro problema.

La diferencia principal al aplicar los algoritmos de búsqueda de subcadenas clásicos a nuestro problema es la de tratar con datos numéricos en vez de con cadenas de caracteres. Nuestros datos son números reales, por lo que la primera medida es transformar el alfabeto infinito en un alfabeto finito manipulable. Para ello definimos distintas cotas que nos permiten transformar los números reales en números enteros y limitar estos a un intervalo manejable por los algoritmos que empleamos.

En cuanto al plan de trabajo, primero se han estudiando varios algoritmos y se han seleccionado los que se han aplicado en el estudio. Luego se ha implementado la aplicación con las funciones que se han considerado más importantes para resolver los distintos problemas. Y por último se han analizado los tiempos de los algoritmos de búsqueda de subcadenas para distintos tipos de árboles.

Los capítulos de esta memoria reflejan este plan de trabajo. El segundo capítulo cuenta las características más importantes de los algoritmos que se han implementado. El tercero describe los aspectos más importantes del código de la aplicación y ofrece una guía de uso para manejarla. El cuarto cuenta los distintos experimentos que se han realizado sobre el estudio y describe los resultados obtenidos. Se ha añadido un capítulo final para contar las conclusiones que se han sacado del trabajo realizado.

# Capítulo 2

# Algoritmos

Se van a diferenciar dos tipos de algoritmos: algoritmos generales de cadenas que buscan distintas características que se producen en el tronco a lo largo del tiempo o que comparan los intervalos en los que se producen estas características en varios árboles y algorimos de búsqueda de subcadenas que buscan un cierto patrón en las cadenas que tenemos.

## 2.1. Algoritmos generales de cadenas

### 2.1.1. Búsqueda de intervalos de tamaño fijo que cumplen una propiedad

Este algoritmo consiste en recorrer el vector de datos e ir buscando los intervalos en los que se cumple una determinada propiedad. Los intervalos que cumplen la condición se devuelven en un vector de índices.

Dado el tamaño de los intervalos que se buscan, primero se calcula el número de datos que cumplen la propiedad en una *ventana* inicial de ese tamaño y se guarda en una variable contador. Si los primeros valores forman un intervalo válido, se añade el resultado al vector de índices.

A continuación, se va tratando el vector elemento a elemento desde la *ventana* inicial. Para cada iteración:

- Se comprueba si el dato cumple la propiedad. Si el dato es válido se suma 1 al contador.
- Se elimina el primer elemento de la ventana. Si el dato era válido se resta 1 al contador.
- Se mira si el intervalo es correcto. Si lo es, se añade al vector de índices.

Una variedad de este algoritmo permite un número fijo de errores no consecutivos dentro de los intervalos que se devuelven, es decir, permite que un intervalo sea válido

aunque no todos los datos cumplan la propiedad, siempre que el número de datos erróneos sea menor que una cantidad fijada por el usuario y que no sean consecutivos.

Para hacer esto, se definen tres variables: *erroresCometidos* que lleva la suma de errores que se comenten en el intervalo, *erroresSeguidos* que lleva la suma de los errores consecutivos que hay en el intervalo y *error* que es un booleano que indica si el elemento anterior fue un error.

Lo primero que se hace es comprobar cuántos errores hay en la ventana inicial. Para ello, se comprueba la negación de la propiedad y se suma un error cada vez que la propiedad no se cumpla. Se utiliza la variable booleana *error* para calcular el número de errores seguidos que hay en el intervalo. Si se han producido menos errores que los permitidos y el número de errores seguidos es 0, el intervalo es válido y se añade al vector de índices.

A continuación, se va tratando el vector elemento a elemento desde esa primera ventana inicial. Para cada iteración:

- Se comprueba si el dato cumple o no la propiedad. Si el dato es válido se pone la variable *error* a falso. Si no lo es se suma 1 a la variable *erroresCometidos* y, si *error* es cierto, se suma 1 a la variable *erroresSeguidos*.
- Se elimina el primer elemento de la ventana. Si el dato era erróneo se resta 1 a la variable *erroresCometidos*. Si el dato anterior a este tampoco era correcto, se resta 1 a la variable *erroresSeguidos*.
- Se mira si en el intervalo se han producido menos errores que los permitidos y si no hay dos errores seguidos. Si pasa esto, el intervalo es válido y se añade al vector de índices.

Al mostrar los datos, los intervalos consecutivos se agrupan dando lugar a un único intervalo más largo para facilitar la lectura. Si se ha indicado un número de errores mayor que 0, la agrupación de los intervalos se realiza pero el intervalo mostrado puede no ser correcto debido a que contenga más errores. Los intervalos con el tamaño indicado por el usuario dentro de ese intervalo mayor serán los que cumplan la propiedad.

La complejidad del algoritmo es lineal respecto del número de datos del vector.

Este algoritmo se utiliza en la aplicación para encontrar los intervalos homogéneos dentro de uno o varios árboles y para encontrar los intervalos con discrepacias entre dos árboles.

### 2.1.2. Búsqueda del segmento máximo que cumple una propiedad

Este algoritmo consiste en recorrer el vector de datos y buscar el mayor intervalo en el que los datos cumplen una determinada propiedad. Se devuelven en un vector de índices todos los intervalos que tengan tamaño máximo.

Para implementar este algoritmo se definen dos variables: *cont* que lleva el número de elementos que cumplen la propiedad en el segmento que se está explorando y *contMax* que lleva el número de elementos que cumplen la propiedad en el intervalo máximo encontrado hasta el momento.

Los elementos del vector se recorren en un bucle. Si el dato cumple la propiedad se suma 1 al contador y si no la cumple se resetea a 0. En cada vuelta se comprueba si el número de elementos del segmento que se lleva hasta el momento es igual o mayor que el número de elementos del intervalo máximo que se ha encontrado. Si es igual se añade ese intervalo al vector de índices. Si es mayor se resetea el vector de índices y se añade el nuevo intervalo máximo encontrado.

Como en el algoritmo anterior, se puede permitir un número fijo de errores no consecutivos dentro del intervalo que se busca. En este caso, las variables *cont* y *contMax* se mantienen y además se definen otras dos variables: *erroresCometidos* que lleva el número de errores producidos en el último segmento y *erroresSeguidos* que es un booleano que indica si se han producido dos errores consecutivos o no. Además se tiene un TAD cola de errores con el índice de los datos erróneos que se han encontrado en el vector.

Para cada iteración tenemos un análisis de casos:

- Si el dato es correcto, se suma 1 al contador y se pone a falso la variable *erroresSeguidos* para indicar que el siguiente dato no puede hacer saltar ese error.
- Si el dato es erróneo pero el segmento aún puede ser válido, es decir, si el número de errores cometidos es menor que el permitido y la variable *erroresSeguidos* está a falso: se pone la variable *erroresSeguidos* a cierto porque el siguiente dato si puede hacer saltar ese error, se añade 1 al contador de errores cometidos y se añade el índice del error al vector de errores.
- Si el segmento no es aceptable porque se han cometido dos errores seguidos: se resetean todas las variables, es decir, se pone el contador y el número de errores cometidos a 0, el valor de *erroresSeguidos* a falso y se resetea el vector de errores.
- Si el segmento no es aceptable porque se ha superado el número de errores cometidos pero no han sido consecutivos: hay que recuperar el segmento desde el primer error cometido, ya que puede dar lugar a un segmento mayor. Tenemos que añadir el error encontrado al vector de errores, poner la variable *erroresSeguidos* a cierto, poner el contador a la diferencia entre el último error y el primero (será la longitud del nuevo segmento que tengamos) y borrar el primer error del vector de errores.

Tras este análisis de casos tenemos que mirar si el segmento que tenemos es mayor o igual que el que ya teníamos. Para ello miramos si el contador de los elementos correctos más el número de errores cometidos es mayor o igual que la variable *contMax*. Si es igual se añade el intervalo al vector de índices. Si es mayor se resetea el vector de índices y se añade el intervalo como el de mayor tamaño encontrado hasta el momento.

La complejidad del algoritmo es lineal respecto del número de datos del vector.

Este algoritmo se utiliza en la aplicación para encontrar el mayor intervalo homogéneo dentro de uno o varios árboles y para encontrar el mayor intervalo con discrepancias entre dos árboles.

## 2.2. Algoritmos de búsqueda de subcadenas

En general estos algoritmos consisten en buscar las apariciones de un cierto patrón $P$ en un texto $T$. En nuestro caso tanto el patrón como el texto son cadenas numéricas de numeros enteros, ya que los valores reales se han transformado en enteros antes de comenzar el algoritmo.

### 2.2.1. Distancia de Hamming

Los algoritmos que se consideran en el trabajo son aproximados, lo que significa que permiten que haya alguna discrepancia, es decir, permiten que haya algún elemento del patrón que no coincide con el correspondiente elemento del texto.

Se define *la distancia de Hamming* (Langmead [5]) para dos cadenas con la misma longitud como el mínimo número de sustituciones necesarias para convertir una cadena en la otra. Por ejemplo, si tenemos las cadenas `1 2 3 4` y `1 2 3 5`, la distancia de Hamming entre ellas es 1 porque si sustituímos el 4 de la cadena `1 2 3 4` por un 5 obtenemos la cadena `1 2 3 5`. Esta distancia es diferente de *la distancia de edición* (edit distance) en la que además de sustituciones se permiten inserciones y borrados.

En el trabajo se utiliza como medida la distancia de Hamming en lugar de la de edición porque las medidas están asociadas al momento en que se toman y las inserciones y borrados modificarían esta información. De esta forma, si nuestro patrón es `1 -1 1 -1` y permitimos un error, serían correctas las cadenas `c -1 1 -1`, `1 c 1 -1`, `1 -1 c -1` y `1 -1 1 c` siendo `c` cualquier número.

Matemáticamente, el problema de buscar un patrón con $k$-discrepancias puede definirse como encontrar todas las posiciones $1 \leq i \leq n - m + 1$ tal que la desigualdad $d_H(P, T[i..i+m-1]) \leq k$ se mantiene, donde $d_H(A, B)$ es la distancia de Hamming entre las cadenas $A$ y $B$, $n$ es la longitud del texto $T$ y $m$ es la longitud del patrón $P$.

### 2.2.2. Algoritmo de Fuerza Bruta

#### 2.2.2.1. Explicación

El algoritmo de Fuerza Bruta consiste en comparar carácter a carácter el patrón que se quiere buscar y la subcadena del texto que se considera en cada caso. Para ello chequea todas las subcadenas del texto de tamaño $m$. Si se encuentra una ocurrencia del patrón en el texto con menos discrepancias de las permitidas, se devuelve como resultado. Tras

cada comparación se mueve el índice del texto una posición a la derecha.

Puede optimizarse esta versión de forma que si se supera el número de discrepancias permitidas también se mueve el índice del texto una posición a la derecha.

Este algoritmo no requiere preprocesado ni espacio extra en memoria.

#### 2.2.2.2. Algoritmo

El pseudo-código para el algoritmo de Fuerza Bruta optimizado que compara un texto $T$ y un patrón $P$ con $k$ discrepancias se muestra en la figura 2.1.

**Input:** $T = T[1..n]$, $P = P[1..m]$, $k$
1: **for** $i \leftarrow 0$ **to** $n - m$ **do**
2: $neq \leftarrow 0$
3: **for** $j \leftarrow 1$ **to** $m$ **do**
4: **if** $P[j] \neq T[i + j]$ **then**
5: $neq \leftarrow neq + 1$
6: **if** $neq > k$ **then**
7: **break**
8: **if** $neq \leq k$ **then**
9: Report an occurrence $\mathrm{d_H}(P, T[i + 1..i + m]) \leq k$

Figura 2.1: Pseudo-código del algoritmo de Fuerza Bruta [3]

Si se eliminan las líneas 6 y 7, se obtiene el algoritmo de fuerza bruta sin la optimización.

#### 2.2.2.3. Complejidad

En el peor caso, el algoritmo de Fuerza Bruta tiene que comparar $m$ caracteres para las $n-m+1$ posibles subcadenas. Esto nos da un orden de complejidad de $O(m(n-m+1)) = O(nm)$. Sin embargo, en la práctica el algoritmo optimizado es bastante más rápido y tiene una complejidad media de $O(nk)$ siendo $k$ el número de discrepancias permitidas.

### 2.2.3. Algoritmo Boyer Moore

El algoritmo de Boyer Moore es un algoritmo clásico de búsqueda de subcadenas. Fue desarrollado por Bob Boyer y J Strother Moore en 1977. Para conseguir una complejidad lineal respecto al tamaño del texto en la búsqueda del patrón desplaza el patrón varias posiciones después de cada comparación en vez de una sola posición como hace el algoritmo de fuerza bruta. El algoritmo produce buenos resultados cuando el tamaño del patrón a buscar es grande.

#### 2.2.3.1. Explicación

El algoritmo exacto intenta encajar el patrón de derecha a izquierda en el trozo del texto considerado y usa dos heurísticas, en el caso de que se produzca una discrepancia entre el texto y el patrón, para el desplazamiento del patrón:

- *Regla de desplazamiento de carácter malo*: se desplaza el patrón a la derecha para hacer coincidir el carácter no coincidente del texto con una posición más a la izquierda del patrón en donde ese carácter coincide. Supongamos que la discrepancia se produce al comparar el carácter de la posición $k$ del texto con el carácter en la posición $j$ del patrón. Si la posición más a la derecha del carácter $T(k)$ se encuentra en el patrón a la izquierda de la posición $j$ desplazamos el patrón hasta hacer coincidir la aparición más a la derecha del carácter $T(k)$ en el patrón con la posición $k$ del texto. Si el carácter $T(k)$ se encuentra a la derecha de la posición $j$ del patrón desplazamos el patrón en una posición. Por último, si el carácter $T(k)$ no se encuentra en el patrón se desplaza el patrón hasta que la posición 0 del patrón esté alineada con la posición $k + 1$ del texto.

- *Regla de sufijo bueno*: se desplaza el patrón a la derecha para hacer coincidir el sufijo correcto del texto considerado con la siguiente ocurrencia de ese sufijo en el patrón. Además, el carácter incorrecto anterior al sufijo debe ser diferente al de la siguiente ocurrencia de ese sufijo en el patrón.

Para el patrón `1 2 4 1 2 3 1 2` en la regla de desplazamiento de carácter malo, si estamos comparando con la subcadena del texto $c_1c_2c_3c_4c_5c_6c_7c_8$ y el carácter $c_8$ del texto es un 1 entonces desplazamos el patrón una posición para alinear el 1 del texto con el 1 más a la derecha del patrón. Si el carácter $c_8$ del texto es un 3, desplazamos el patrón dos posiciones y si es un 4 se desplaza 5 posiciones. Si el carácter $c_8$ es un valor que no aparece en el patrón lo desplazamos 8 posiciones. En cambio, si la subcadena del texto que estamos comparando es $c_1c_2c_3c_4c_5c_6$ `1 2` y el carácter $c_6$ es un 1 solo se desplaza una posición porque el 1 está más a la derecha en el patrón. Lo mismo ocurre si el carácter $c_6$ es un 2, solo se desplaza un posición. Si el caracter $c_6$ es un 4, se desplaza 3 posiciones.

Para el mismo patrón y la regla de sufijo bueno, si no tenemos ningún sufijo correcto válido (0 posiciones correctas), se desplaza el patrón una posición. Si tenemos un sufijo correcto de una posición (2) se desplaza el patrón ocho posiciones (todo el patrón) porque no existe ningún 2 en el patrón que no vaya precedido de un 1. Si tenemos un sufijo correcto de dos posiciones (1 2) se desplaza el patrón 3 posiciones porque al sufijo 1 2 primero le precede un 3 y luego un 4, así que esa ocurrencia es válida. Si tenemos un sufijo correcto de tres posiciones (3 1 2), se desplaza el patrón 6 posiciones porque no existe ningún otro sufijo válido de tamaño tres en el patrón. Debido a este último caso, el resto de sufijos correctos de más de 3 posiciones también hacen que se desplace el patrón 6 posiciones a la derecha. El resumen de esta regla puede verse en la tabla 2.1.

En nuestro programa usaremos dos versiones del algoritmo: una versión del algoritmo de Boyer Moore para resolver el problema de búsqueda de patrones exacto en el que se

| **Tamaño sufijo** | **Sufijo correcto** | **Desplazamiento** |
|---|---|---|
| 0 | - | 1 |
| 1 | 2 | 8 |
| 2 | 1 2 | 3 |
| 3 | 3 1 2 | 6 |
| 4 | 2 3 1 2 | 6 |
| 5 | 1 2 3 1 2 | 6 |
| 6 | 4 1 2 3 1 2 | 6 |
| 7 | 2 4 1 2 3 1 2 | 6 |

Tabla 2.1: Regla de sufijo bueno para el patrón `1 2 4 1 2 3 1 2`

implementan las dos reglas y una versión basada en el algoritmo de Horspool (simplificación del algoritmo de Boyer Moore) para resolver el problema de búsqueda de patrones con $k$-discrepancias en el que únicamense se implementa la regla de desplazamiento de caracter malo.

#### 2.2.3.2. Algoritmo exacto (Gusfield [2])

Hay que generalizar la búsqueda de derecha a izquierda del patrón en el trozo de texto considerado y calcular el desplazamiento adecuado del patrón para cada heurística.

Para calcular el desplazamiento de la regla de carácter malo entendamos lo que queremos calcular. Supongamos que comparamos el patrón con una subcadena que discrepa en la posición $i$ del texto con la posición $j$ del patrón. Para calcular el desplazamiento tenemos que mirar la posición más a la derecha de la subcadena que coincide con el patrón y ver si desplazando el patrón hacía la derecha encontramos una coincidencia entre ese carácter y alguno de los del patrón. Si esta coincidencia ocurre desplazamos la subcadena tantas posiciones como hemos desplazado el patrón. Si no hay ninguna coincidencia desplazamos la subcadena $m$ posiciones, siendo $m$ la longitud del patrón. De esta forma nos saltamos la posición del texto que hemos estudiado.

Estudiando todos los valores posibles del texto y del patrón se puede construir un vector que nos indique cuántas posiciones hay que desplazar en la regla del carácter malo para cada número $x$. A este valor le llamamos $R(x)$.

Para calcular el desplazamiento de la regla de sufijo bueno entendamos lo que queremos calcular. Supongamos, dado el texto $T$ y el patrón $P$, que una subcadena $S$ de $T$ encaja con un sufijo de $P$ pero hay una discrepancia en la siguiente posición a la izquierda. Tenemos que encontrar, si existe, una subcadena $S'$ en $P$ que sea igual a $S$, que no sea un sufijo de $P$ y que el carácter a la izquierda de $S'$ en $P$ difiere del carácter a la izquierda de $S$ en $P$. En el diagrama de la figura 2.2 podemos ver gráficamente lo que acabamos de explicar.

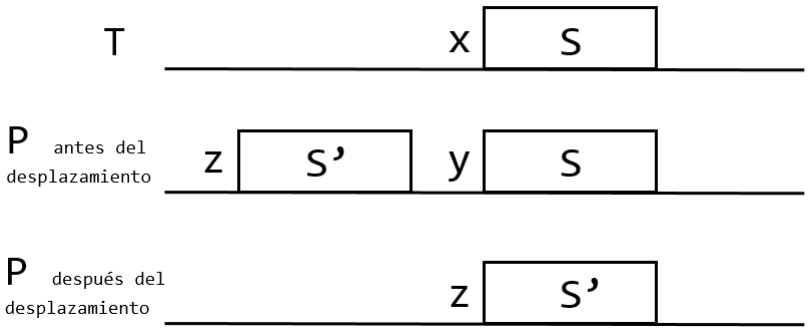


Figura 2.2: Regla del sufijo bueno, el caracter $x$ de $T$ discrepa del caracter $y$ de $P$. Los caracteres $y$ y $z$ son distintos, por lo que $z$ igual coincide con $x$

Si $S'$ existe, desplazamos $P$ a la derecha de tal forma que la subcadena $S'$ de $P$ coincida con la subcadena $S$ de $T$. Si $S'$ no existe, desplazamos a la derecha $P$ hasta pasar la subcadena $S$ del texto más la menor cantidad tal que un prefijo del patrón $P$ desplazado encaja con un sufijo de la subcadena $S$ de $T$. Si no encontramos esta cantidad, desplazamos $P$ $m$ posiciones a la derecha, esto es, pasamos la subcadena $S$ del texto.

Si se produce una ocurrencia de $P$ en el texto $T$, desplazamos $P$ con la menor cantidad tal que un prefijo del patrón $P$ desplazado encaja con un sufijo de la ocurrencia encontrada en $P$. Si no encontramos esta cantidad, desplazamos $P$ $m$ posiciones a la derecha, esto es, pasamos la ocurrencia encontrada en el texto.

Para la regla del sufijo bueno del algoritmo de Boyer Moore se definen varios conceptos:

- $Z_i(P)$: longitud de la mayor subcadena de $P$ que empieza en $i$ y coincide con un prefijo de $P$.
- $N_i(P)$: longitud del sufijo más largo de la subcadena $P[1..i]$ que es sufijo del patrón $P$.
- $L(i)$: la mayor posición estrictamente menor que la longitud del patrón $P$ tal que $P[i..n]$ coincide con un sufijo de $P[1..L(i)]$. Puede ser 0.
- $L'(i)$: la mayor posición estrictamente menor que la longitud del patrón $P$ tal que $P[i..n]$ coincide con un sufijo de $P[1..L(i)]$ y el carácter anterior de ese sufijo no es igual a $P(i-1)$. Puede ser 0.
- $l'(i)$: longitud del sufijo más largo de $P[i..n]$ que es prefijo de $P$.

Para el patrón `2 1 2 4 1 2 4 1 2` vemos en la tabla 2.2 cada uno de los valores para algunos de los conceptos definidos.

Se puede demostrar que $N$ es el inverso de $Z$, es decir, si se denota $inv(P)$ como el patrón inverso de $P$, entonces $N_i(P) = Z_{m-i+1}(inv(P))$. Como ejemplo, si $P$ es el patrón

| **Pos** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **Patrón** | **2** | **1** | **2** | **4** | **1** | **2** | **4** | **1** | **2** |
| $Z_{pos}$ | 8 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| $N_{pos}$ | 1 | 0 | 2 | 0 | 0 | 5 | 0 | 0 | 8 |
| $L(pos)$ | 0 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 |
| $L'(pos)$ | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 3 | 0 |
| $l'(pos)$ | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Tabla 2.2: Cálculo de $Z_{pos}$, $N_{pos}$, $L(pos)$, $L'(pos)$ y $l'(pos)$ para el patrón `2 1 2 4 1 2 4 1 2`

`2 1 2 4 1 2 4 1 2`, $inv(P)$ es el patrón `2 1 4 2 1 4 2 1 2` y $Z(inv(P))$ =`8 0 0 5 0 0 2 0 1`= $inv(N(P))$.

**Preprocesado** Para la regla de desplazamiento de carácter malo hay que determinar para cada número posible y para cada posición del patrón $i$ la mayor posición menor que $i$ de la ocurrencia de ese número en el patrón. Así que el preprocesado de esta parte consiste en construir una tabla que contenga para cada posición $i$ del patrón y cada número $x$, el número $s$ tal que la posición $i - s$ del patrón contenga el número $x$. Si no existe tal posición, el número $s$ se corresponde con la longitud del patrón (es decir, $m$). Matemáticamente: $d[i, x] = min\{s | s = m \ \vee \ (1 \leq s < m \ \wedge \ p_{i-s} = x)\}$. Como en nuestro algoritmo sólo queremos mirar la última posición del patrón, la tabla puede convertirse en un vector $R$ (sólo se rellena la fila $d[m, x]$). Al valor de cada número $x$ le llamamos $R(x)$

Para la regla de sufijo bueno hay que calcular $L'(i)$ y $l'(i)$ para cada posición $i$ del patrón $P$. Antes hay que calcular $Z_j(inv(P))$ por recursión y su inversa para obtener $N_j(P)$. El pseudo-código del cálculo de $L'(i)$ aparece en la figura 2.3.

**Z-based Boyer–Moore**

```
for i := 1 to n do L'(i) := 0;
for j := 1 to n − 1 do
    begin
    i := n − N_j(P) + 1;
    L'(i) := j;
    end;
```

Figura 2.3: Pseudo-código del cálculo de $L'(i)$ [2]

**Algoritmo** El pseudo-código del algoritmo Boyer Moore exacto se muestra en la figura 2.4. Una particularidad de este algoritmo es que necesita que el patrón buscado tenga al menos 2 elementos pues necesita el valor de $l'(2)$.

**The Boyer–Moore algorithm**

```
{Preprocessing stage}
    Given the pattern P,
    Compute L'(i) and l'(i) for each position i of P,
    and compute R(x) for each character x ∈ Σ.
{Search stage}
    k := n;
    while k ≤ m do
        begin
        i := n;
        h := k;
        while i > 0 and P(i) = T(h) do
            begin
            i := i − 1;
            h := h − 1;
            end;
        if i = 0 then
            begin
            report an occurrence of P in T ending at position k.
            k := k + n − l'(2);
            end
        else
            shift P (increase k) by the maximum amount determined by the
            (extended) bad character rule and the good suffix rule.
        end;
```

Figura 2.4: Pseudo-código del algoritmo Boyer Moore exacto [2]

**Complejidad (Gusfield [2])** En cuanto al preprocesado, el coste en memoria de la tabla es el número de carácteres en nuestro vocabulario (intervalo de números posibles). La complejidad del cálculo es del orden $O(m)$ siendo $m$ el tamaño del patrón $P$. El cálculo de $Z$, $N$, $L^{'}$ y $l^{'}$ es también del orden de $O(m)$. Así que, en total, la complejidad del procesado es del orden $O(m)$.

El algoritmo de Boyer Moore exacto tiene una complejidad del orden $O(n)$ para el peor caso (cuando el patrón no aparece en el texto) siendo $n$ el número de caracteres del texto.

#### 2.2.3.3. Algoritmo aproximado

La regla de desplazamiento de carácter malo produce en la práctica buenos resultados (Tarhio y Ukkonen [8]) y es fácilmente extensible para considerar discrepancias en el patrón. Por ello en el algoritmo aproximado solo se utiliza esta regla.

Hay que generalizar la búsqueda de derecha a izquierda del patrón en el trozo de texto y calcular el desplazamiento adecuado para cada carácter encontrado.

Para generalizar la búsqueda hay que escanear el patrón de derecha a izquierda en el trozo de texto hasta encontrar $k+1$ discrepancias (búsqueda infructuosa) o encontrar una cadena coincidente de como mucho $k$ discrepancias.

Para calcular el desplazamiento de carácter malo hay que entender qué queremos calcular. Supongamos que comparamos el patrón con una subcadena que discrepa por $k+1$ vez en la posición $i$ del texto y en la posición $j$ del patrón. Para calcular el desplazamiento tenemos que mirar las últimas $k+1$ posiciones de la subcadena coincidentes con el patrón y ver si desplazando el patrón hacía la derecha encontramos una coincidencia entre alguno de esos caracteres y alguno de los del patrón. Si esta coincidencia ocurre desplazamos la subcadena tantas posiciones como hemos desplazado el patrón. Si no hay ninguna coincidencia desplazamos la subcadena $m - k$ posiciones (valor para saltarnos la última posición estudiada).

**Preprocesado** El preprocesado del algoritmo para $k$-discrepancias consiste en construir una tabla que contenga para cada posición $i$ del patrón y cada número $n$ del vocabulario el número $s$ tal que la posición $i - s$ del patrón contenga el número $n$. Si no existe tal posición, el número $s$ se corresponde con la longitud del patrón (es decir, $m$). Matemáticamente: $d_k[i, n] = min\{s | s = m \vee (1 \leq s < m \wedge p_{i-s} = n)\}$. Como en nuestro algoritmo sólo queremos mirar las últimas $k + 1$ posiciones del patrón, podemos rellenar la tabla de $i = m - k$ hasta $i = m$. El pseudo-código del preprocesado se muestra en la figura 2.5.

Lo importante de este código es el tercer bucle, que calcula los valores de la tabla $d_k$ para los caracteres que pertenecen al patrón $P$. En definitiva, para cada carácter del patrón $p_i$ y para cada posición $j > i$, añade que $d_k[j, p_i] = j - i$.

Como ejemplo, si el patrón es `1 -1 1 -1 1`, vemos el resultado de la tabla $d_k$ en la figura 2.3.

| **c\i** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| **1** | 5 | 1 | 2 | 1 | 2 |
| **-1** | 5 | 5 | 1 | 2 | 1 |

Tabla 2.3: Tabla $d_k$ para el patrón `1 -1 1 -1 1`

**Algorithm 3.** Computation of table $d_k$.

```
1.  for a in Σ do ready[a] := m + 1;
2.  for a in Σ do
3.          for i := m downto m − k do
4.                  d_k[i, a] := m;
5.  for i := m − 1 downto 1 do begin
6.          for j := ready[p_i] − 1 downto max(i, m − k) do
7.                  d_k[j, p_i] := j − i;
8.          ready[p_i] := max(i, m − k) end
```

Figura 2.5: Pseudo-código del preprocesado del algoritmo Boyer Moore aproximado [8]

**Algoritmo** La idea del algoritmo es, para cada subcadena del texto de tamaño $m$, ir mirando de derecha a izquierda si la subcadena y el patrón coinciden. Si no lo hacen se suma 1 al número de discrepancias encontradas y, si se llega al final de la subcadena con un número menor de $k$ errores, se devuelve una ocurrencia en esa posición. Tras cada intento se desplaza la subcadena con el valor que permite hacer coincidir el carácter más a la derecha posible del patrón con un carácter de la subcadena estudiada. Si no existe ningún carácter, se desplaza $m - k$ (lo máximo que se puede desplazar si la subcadena y el patrón no tienen ningún carácter en común).

El pseudo-código del algoritmo Boyer Moore aproximado se encuentra en la figura 2.6.

**Complejidad (Tarhio y Ukkonen [8])** En cuanto al preprocesado, el coste en memoria de la tabla es de $(k + 1) \times c$ siendo $c$ el número de carácteres en nuestro vocabulario. La complejidad del algoritmo es del orden $O(kc)$ para los dos primeros bucles for. Para el tercero tenemos una complejidad del orden $O(m)$ para el bucle for más el tiempo de las actualizaciones de la tabla $d_k$. Como cada $d_k(j, p_i)$ se actualiza como mucho una vez (orden $O(kc)$)) tenemos un orden total de $O(m + kc)$.

La complejidad del algoritmo es del orden $O(mn)$ en el peor caso. En media, el orden de complejidad de este algoritmo es $O(nk \times (\frac{k}{c} + \frac{1}{m-k}))$.

### 2.2.4. Algoritmo Shift-Add

El algoritmo Shift-Add, también conocido como algoritmo Shift-Or o algoritmo Bitap, es un algoritmo de búsqueda de subcadenas aproximadas. Fue desarrollado en su versión exacta por Bálint Dömölki en 1964 y luego extendido por R. K. Shyamasundar en 1977. Fue reinventado como versión aproximada por Manber y Wu en 1991 basándose en el trabajo de Ricardo Baeza-Yates y Gaston Gonnet. Fue mejorado por Baeza-Yates y Navarro en 1996.

**Algorithm 4.** Approximate string matching with $k$ mismatches.

```
1.  compute table d_k from P with Algorithm 3;
2.  j := m;                                   {pattern ends at text position j}
3.  while j ≤ n do begin
4.      h := j; i := m; neq := 0;             {h scans the text, i the pattern}
5.      d := m − k;                           {initial value of the shift}
6.      while i > 0 and neq ≤ k do begin
7.          if i ≥ m − k then d := min(d, d_k[i, t_h]);
                                              {minimize over the component shifts}
8.          if t_h ≠ p_i then neq := neq + 1;
9.          i := i − 1; h := h − 1 end;       {proceed to the left}
10.     if neq ≤ k then report match at position j;
11.     j := j + d end                        {shift to the right}
```

Figura 2.6: Pseudo-código del algoritmo Boyer Moore aproximado [8]

La idea del algoritmo es precomputar un conjunto de máscaras de bits que contienen un bit para cada elemento del patrón. Tras esto, en la fase de búsqueda, se hace la mayor parte del trabajo con operaciones bit a bit. Este algoritmo funciona bien para patrones cortos (como mucho la longitud de palabra de la máquina en la que se use) y con alfabetos pequeños.

#### 2.2.4.1. Explicación

El algoritmo Shift-Add resuelve el problema de búsqueda de subcadenas con $k$-discrepancias usando vectores de bits. La idea principal es representar $m$ estados de la búsqueda (tantos como la longitud del patrón) en un vector y recorrer el texto de izquierda a derecha intentando llegar a estados en donde el patrón encaje con el texto.

Por ejemplo, dado el texto `1 1 -1 1 1 -1 1` y el patrón `1 -1 1 1` , debemos representar 4 estados, correspondientes a los cuatro dígitos del patrón. Cada estado lo representamos con 3 bits, ya que necesitamos guardar 5 posibles valores (0-4) correspondientes a las cuatro posibles discrepancias del patrón con el texto. Necesitamos por lo tanto 12 bits.

Cada posición del vector de estado lleva la cuenta de cuántas discrepancias se han detectado hasta esa posición en un alineamiento del patrón $P$ con un trozo del texto $T$. Así, si $S_i[j]$ es el vector de estados para la posición de texto $i$ y la posicion del patrón $j$, tenemos que $S_i[j] = d_H(T[i-j+1..i], P[1..j])$, es decir, la distancia de Hamming entre el prefijo de longitud $j$ del patrón y el sufijo del trozo de texto que termina en la posición $i$.

En nuestro ejemplo, si estamos comprobando la componente $i = 5$ del texto, el vector de bits será: `000 001 010 000`. Donde en el primer estado ($j = 1$) se guardan las 0 discrepancias entre el texto en la posición $i = 5$ y el patrón en la posición 1. El segundo estado ($j = 2$) guarda las discrepancias entre el texto en las posiciones [4 . . 5] y el patrón en las posiciones [1 . . 2]. Se observa que hay una discrepancia entre el dígito de la posición 5 del texto y el dígito en la posición 2 del patrón. El tercer estado guarda las dos discrepancias que hay entre el texto en las posiciones [3 . . 5] y el patrón en las posiciones [1 . . 3] y el último estado guarda las discrepancias entre las posiciones [2 . . 5] del texto y las posiciones [1 . . 4] del patrón.

De esta forma, $S_i[m] \leq k$ indica una ocurrencia del patrón $P$ en el trozo de texto $T[i - j + 1..i]$ donde como mucho hay $k$ discrepancias. Podemos comprobar que en el ejemplo se ha encontrado el patrón debido a que el último estado es `000`, lo que indica que no hay discrepancias entre las posiciones [2 . . 5] del texto y todas posiciones del patrón. En otras palabras, en las posiciones [2 . . 5] del texto se ha encontrado una ocurrencia del patrón.

Para pasar de un estado al siguiente se usan operaciones lógicas de bits como sumas y desplazamientos. En cada paso se actualizan simultáneamente las $m$ posiciones del vector.

La parte de búsqueda del algoritmo inicializa $S_0[j] = k + 1$ para todo $j = 1, ..., m$ y se procesa el texto de la primera a la última posición. Para cada posición $i = 1, ..., n$ del texto, se calcula el vector de estados $S_i$ aplicando la siguiente fórmula:

$$S_i[j] = \begin{cases} 0 & \text{si } j = 1 \\ S_{i-1}[j-1] & \text{en otro caso} \end{cases} + \begin{cases} 0 & \text{si } P[j] = T[i] \\ 1 & \text{si } P[j] \neq T[i] \end{cases} \tag{2.1}$$

Se observa que se aumenta en 1 el vector de estados en la posición $j$ si hay una discrepancia entre la posición $j$ del patrón y la posición del texto que toca procesar. Cada actualización del vector de estados introduce un nuevo alineamiento representado por $S_i[1]$ y desplaza el alineamiento más antiguo ($S_{i-1}[m]$). Tras actualizar el vector de estado, se comprueba el último valor para reportar una ocurrencia si $S_i[m] \leq k$.

En nuestro ejemplo, si consideramos que queremos un alineamiento exacto entre el texto y el patrón, inicializamos el vector de estados a `001 001 001 001`. De esta forma vemos que es imposible llegar a obtener un acierto en la búsqueda hasta haber actualizado el vector de estados 4 veces, tantas como números tiene nuestro patrón, ya que en las actualizaciones sólo se realizan sumas.

Para realizar la primera actualización con la primera posición del texto en todos los estados tenemos que desechar el último estado y añadir uno nuevo con valor `000`. Para el primer estado, además, tenemos que comprobar la igualdad entre la primera posición del texto y la primera posición del patrón. En nuestro ejemplo ambos valores son `1`, así que no se le suma nada y sigue valiendo `000`. Para los estados intermedios sumamos al estado

anterior la comprobación de si son iguales la primera posición del texto con la posición $j$-ésima del patrón. De esta forma, para el segundo estado, como la primera posición del texto es un `1` y la segunda del patrón es un `-1` no coinciden ambos números y el resultado del segundo estado será la suma del primero más uno, en otras palabras, `010`. Para el tercer y el cuarto estado si coinciden las posiciones entre el texto y el patrón pues la tercera y la cuarta posición del patrón es un `1` al igual que la primera posición del texto, así que el resultado de estos estados es el estado anterior sin sumarle nada, es decir, `001`. El vector de estados tras esta primera actualización es `000 010 001 001`.

#### 2.2.4.2. Implementación

Para implementar esto no necesitamos más que $\log_2(k+1)$ bits para representar el estado. Esto es debido a que cada estado debe tener al menos $b = \lceil \log_2(k+1) \rceil + 1$ bits ya que se necesitan $k+1$ discrepancias para determinar si un alineamiento no es correcto. Se precisa de un bit adicional para evitar los desbordamientos.

Para calcular más eficientemente la actualización del vector de estados, se define una tabla $t$ para cada número $c$ posible y cada posición $j$ que nos devuelve el valor del segundo término de $S_i[j]$ (un conjunto de $L$ bits). De esta forma, para un número $n$ tenemos:

$$t[c,j] = \begin{cases} 0^b & \text{si } P[j] = c \\ 0^{b-1}1 & \text{si } P[j] \neq c \end{cases} \tag{2.2}$$

En nuestro ejemplo, si suponemos que admitimos una discrepancia, necesitamos $\lceil \log_2(1+1) \rceil + 1 = 1 + 1 = 2$ bits. El alfabeto está constituido por los números `1` y `-1` y para rellenar la tabla tenemos que ir comparando cada número con las $j$ posiciones del patrón. Por ejemplo, para $j = 1$ tenemos que el valor del patrón vale `1` así que $t[1,1] = 00$ y $t[1,-1] = 01$. El resultado para nuestro ejemplo se encuentra en la tabla 2.4.

| **c\j** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| 1 | 00 | 01 | 00 | 00 |
| -1 | 01 | 00 | 01 | 01 |

Tabla 2.4: Tabla $t$ para el patrón `1 -1 1 1`

**Preprocesado** El pseudo-código del preprocesado se encuentra en la figura 2.7.

```
b = log(k+1)+1
OF_MASK = (10^(b-1))^|p| ; NEG_OF_MASK = ~OF_MASK
for h in α and j=1..|p|: t[h,j] = 0^(b-1)1
for j=1..|p|:            t[p[j],j] = 0^b
s = 0^(b(|p|+1)) ; o = 0^(b(|p|+1))
```

Figura 2.7: Pseudo-código del preprocesado del algoritmo Shift-Add [6]

La primera línea define la constante $b$ y la segunda las máscaras necesarias para el algoritmo. La tercera inicia la tabla $t$ para el alfabeto $\alpha$, que en nuestro caso es un conjunto de números, con el valor por defecto. La cuarta sobrescribe ese valor para los números que constituyen el patrón. La última línea inicia el vector de estados (variable $s$) y el vector que llevará los posibles desbordamientos (variable $o$).

Para nuestro ejemplo y suponiendo que admitimos una discrepancia ya habíamos calculado que $b = 2$. Por lo tanto, la línea 3 del algoritmo nos daría `t[1,:]=01 01 01 01` y `t[-1,:]=01 01 01 01`. La cuarta línea corregiría la tabla según los valores que toma el patrón y nos quedaría finalmente `t[1,:]=00 01 00 00` y `t[-1,:]=01 00 01 01` que, si nos fijamos, coincide con los valores calculados en la tabla 2.4.

**Algoritmo** El pseudo-código de la fase de búsqueda del algoritmo se encuentra en la figura 2.8.

```
for i=1..|x|:
  s = (s >> b) + t[x[i]]
  o = (o >> b) | (s & OF_MASK)
  s = s & NEG_OF_MASK
  if s[|p|] + o[|p|] <= k and i>=|p|:
    report i-|p|+1 as match
```

Figura 2.8: Pseudo-código del algoritmo Shift-Add [6]

Para cada posición nueva del texto se actualiza el vector de estados $s$ según la definición dada: se desplaza $b$ posiciones el vector de estados desechando el último estado y añadiendo uno nuevo nulo y se suma a este nuevo vector de estados el vector de la tabla $t$ correspondiente al número de la posición $i$-ésima del texto. Luego se divide esta suma en el vector de desbordamientos $o$ y el vector de estados $s$ sin la parte del desbordamiento. A continuación se comprueba si $S_i[m] + O_i[m] \leq k$ para reportar un posible ocurrencia del patrón $P$ en el texto con como mucho $k$ discrepancias.

#### 2.2.4.3. Complejidad (Hirvola [3])

En cuanto al preprocesado, la tercera línea tiene un coste del orden $O(\frac{|\alpha|mlog(k)}{b})$ y la cuarta un coste del orden $O(m)$, así que el total es del orden de su suma. El espacio de la tabla $t$ es del orden $O(\frac{|\alpha|mlog(k)}{b})$.

El coste del algoritmo es del orden $O(n)$ asumiendo que las operaciones de vectores de bits tienen tiempo constante. Para un patrón de longitud $m$ y un número de discrepancias $k$ tales que $m\log(k) > w$ siendo $w$ la longitud de la palabra en la máquina usada, el algoritmo tiene un coste $O(mn)$.

# Capítulo 3

# Aplicación

En este capítulo se va a describir la aplicación tanto a nivel de usuario como a nivel de implementación. En la parte de usuario primero se describen las funciones que hay en la aplicación y luego se incluye una pequeña guía de uso. En la parte de implementación se describe el patrón de diseño utilizado y luego los aspectos más relevantes de cada una de las clases. La aplicación puede descargarse en la siguiente dirección de GitHub: https://github.com/Bea1995/TFG

## 3.1. Explicación del problema

El problema que hay que resolver es el de encontrar semejanzas en las variaciones del grosor de los troncos de diversos árboles durante ciertos intervalos de tiempo para estudiar su reacción a diversos estímulos ambientales. Para ello, se recibe un fichero de texto con las medidas en micras de los troncos de varios árboles. Las medidas se toman cada 30 segundos con un sensor de voltaje y pueden convertirse a micras con una fórmula.

El fichero con el que se ha trabajado en este estudio incluye las medidas de 20 árboles durante 26 días. Esto supone 78832 medidas para cada árbol. Para minimizar el uso de la memoria, sólo se cargan en el programa los árboles necesarios para cada función. De esta forma, antes de aplicar cualquier función, se cargan los datos de los árboles que se necesitan.

Para cada árbol se tienen las fechas de inicio y fin y un dato del grosor del tronco cada 30 segundos. Lo primero que se hace es realizar una limpieza de los datos corrigiendo ciertos errores que se producen por fallos en el sensor. En particular, se quitan valores inválidos, se añaden fechas que faltan en las medidas y se eliminan grandes diferencias entre los valores. Una vez obtenida esta nueva colección de datos, se obtendrán otras dos colecciones asociadas de la siguiente manera:

- **Colección de datos normalizada**: se obtiene restando el primer elemento a todos los datos de la colección original. De esta forma, se normalizan los datos para que la primera medida se corresponda con el 0. Esta colección no modifica las relaciones de unos datos con otros, sino que mueve todos los datos a un punto de partida en dónde

es más fácil ver las relaciones que existen entre ellos. Esta colección se utilizará a la hora de representar las gráficas.

- **Colección de diferencias**: se obtiene restando a cada elemento de la colección de datos normalizada su elemento anterior. Se obtiene así una colección de datos con las diferencias que existen de unos datos a otros. Esta colección es la que se utiliza en la mayoría de los algoritmos, en los que se buscan relaciones en las diferencias que existen entre unos datos y otros.

La propuesta es aplicar los algoritmos que se han explicado en el Capítulo 1 sobre distintos conjuntos de árboles en distintos momentos del día y descubrir si existe algún paralelismo entre los datos. Se pone especial énfasis en los algoritmos de búsqueda de subcadenas porque son los que permiten estudiar más comportamientos para los distintos datos.

## 3.2. Descripción de la aplicación. Usuario

Para el usuario final, la aplicación se ve de la siguiente manera:

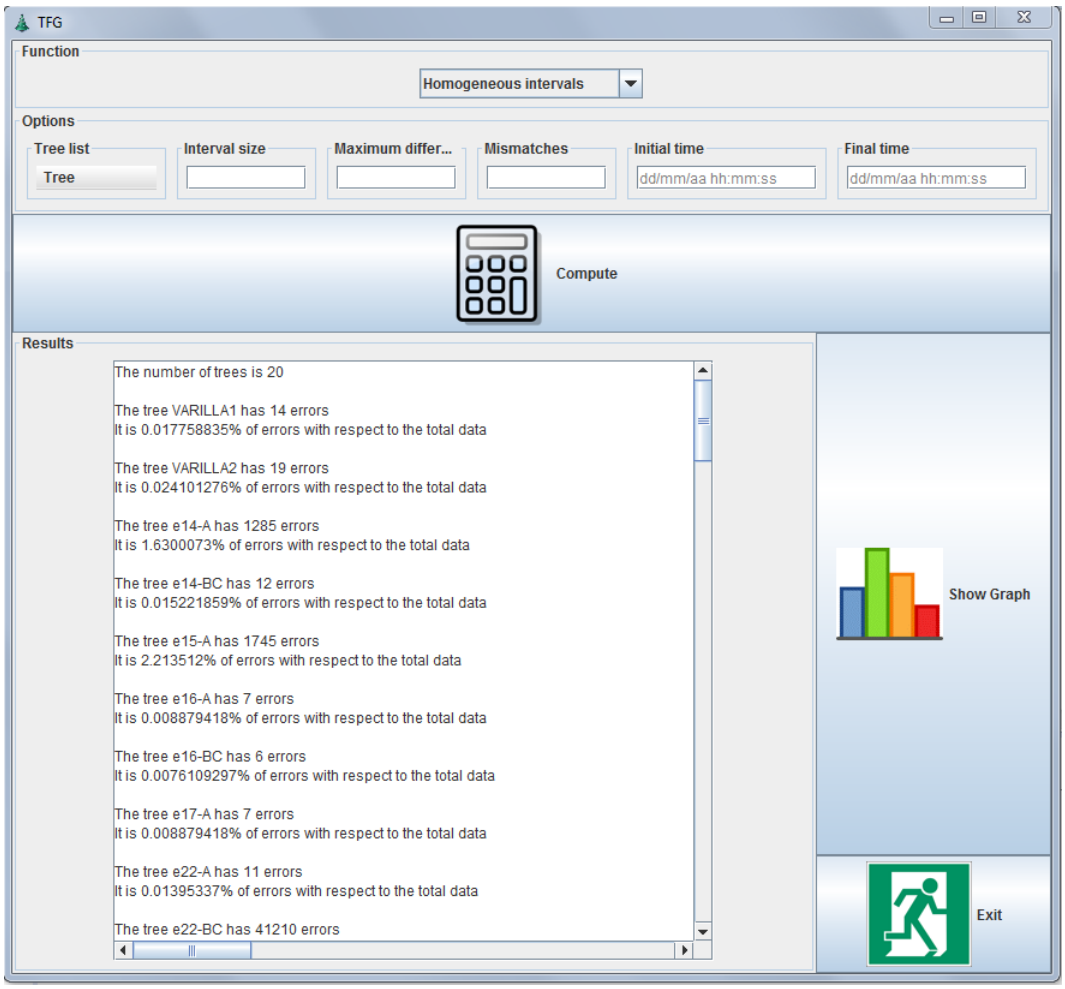


Figura 3.1: Ventana de la aplicación

En esta ventana, el usuario puede realizar varias acciones:

1. Elegir una opción del desplegable de funciones (campo *Function*) y rellenar los datos asociados a esa opción. Cuando se cambia de una función a otra, las opciones cambian automáticamente y muestran los datos asociados a la función escogida.
2. Dar al botón *Compute* una vez se han rellenado todos los datos. Si los datos son correctos, se muestra el resultado en el cuadro de texto. Si se ha producido algún fallo, se muestra en el cuadro de texto un mensaje de error.
3. Consultar los resultados obtenidos. En la pantalla inicial se muestra el número de datos corregidos de cada árbol (figura 3.1).
4. Dar al botón *Show Graphic* una vez se ha obtenido un resultado. Si los datos son correctos, se despliega una ventana diferente que muestra una gráfica con los datos de los árboles elegidos por el usuario en el intervalo de tiempo seleccionado. Si se ha producido algún fallo, no se muestra nada.
5. Dar al botón *Exit* y cerrar la aplicación.

### 3.2.1. Funciones de la aplicación

En esta sección se describe cada una de las funciones que ofrece la aplicación. Primero se explica lo que hace cada función y luego se describen los rasgos principales de su implementación. Se tiene especial énfasis en definir los distintos parámetros que necesita cada una de las funciones ya que se reciben del usuario final.

#### 3.2.1.1. Datos de un árbol

El objetivo de esta función es mostrar los datos de un árbol en un intervalo de tiempo elegido por el usuario. Se mostrarán los datos del vector del árbol, del vector normalizado y del vector de diferencias.

#### 3.2.1.2. Búsqueda de intervalos homogéneos en un árbol

Esta función consiste en buscar intervalos homogéneos en el vector de diferencias de un árbol entre dos fechas seleccionadas. Se considera que un intervalo es homogéneo cuando la diferencia de un dato al siguiente en valor absoluto es menor que una cantidad dada por el usuario.

Con esta función se pueden encontrar los intervalos de tiempo en donde la diferencia de una medida a la siguiente en el árbol no difiere en más de una cantidad. Para ello el usuario tiene que seleccionar el árbol sobre el que se quieren calcular los intervalos, indicar la cantidad máxima de una medida a otra y escribir las fechas de inicio y fin. Además tiene que indicar el tamaño del intervalo o si quiere que se encuentre el intervalo más largo en el conjunto de datos. También puede indicar el número de errores que permite en el intervalo, es decir, si admite que haya algún dato que no cumpla la función de homogeneidad que se ha definido pero aún así siga siendo un intervalo válido.

#### 3.2.1.3. Búsqueda de intervalos homogéneos en varios árboles

Para esta función se ampliará la condición de homogeneidad definida para poder aplicarla sobre varios árboles. Se ofrecen dos posibilidades:

1. Varios árboles cumplirán la condición de homogeneidad si la cumplen por separado en el mismo intervalo de tiempo.

2. Varios árboles cumplirán la condición de homogeneidad si la diferencia de sus valores en valor absoluto es menor que una cierta cantidad dada.

Con esta función se pueden encontrar los intervalos de tiempo en donde la diferencia de una medida a la siguiente no difiere en más de una cantidad en varios árboles a la vez. El usuario tiene que indicar los árboles sobre los que aplicar la función, las fechas de inicio y fin, el valor sobre el que aplicar la condición de homogeneidad y la forma de aplicarla (cuál de las dos versiones tener en cuenta).

Además, como en la función anterior, el usuario tiene que indicar si quiere buscar intervalos de un tamaño concreto o el intervalo más grande. También tiene que indicar el número de errores que permite que se produzcan en el intervalo. Como particularidad, no se puede buscar el intervalo más grande con errores para más de dos árboles debido a que la forma de calcular el resultado sería mediante ensayo y error y no usando los algoritmos ya explicados. El resto de posibilidades sí están implementadas.

#### 3.2.1.4. Búsqueda de intervalos con discrepancias en dos árboles

Esta función consiste en buscar intervalos entre dos árboles en donde se produce una discrepancia entre ellos a lo largo de todo el intervalo. Se define discrepancia como el momento en el que uno de los árboles toma un valor distinto en signo al del otro árbol considerando el caso constante como un signo distinto al positivo y el negativo. De esta forma, se buscarán intervalos en donde los dos árboles tienen valores discrepantes en sus medidas.

El usuario tiene que elegir los dos árboles en donde buscar los intervalos, las fechas de inicio y fin y el número de errores que se permiten en el intervalo. En este caso, un error se producirá cuando ya no se produzca una discrepancia, es decir, cuando el signo de las medidas de los dos árboles sea el mismo.

Del mismo modo al de las dos funciones anteriores, el usuario indicará si quiere buscar intervalos de un tamaño concreto o el intervalo más grande.

#### 3.2.1.5. Crecimiento del tronco en un árbol

Esta función tiene en cuenta que nuestros datos son los valores que va tomando el grosor del tronco de un árbol cada 30 segundos y su cometido es el de devolver cuánto ha

crecido cada árbol cada día que pasa.

El usuario tiene que indicar el árbol sobre el que quiere medir el crecimiento y las fechas de inicio y fin de esa búsqueda. Para este algoritmo sólo se tendrán en cuenta los días completos que hay entre ambas fechas. Como parámetro opcional, el usuario puede indicar si se devuelve el crecimiento de los días superiores a una cierta cantidad dada o el día en el que más creció el tronco.

#### 3.2.1.6. Comparación de los troncos en varios árboles

En esta función se compara el crecimiento de los troncos de dos árboles. Para ello, hay que definir cómo de parecidos son dos árboles respecto al crecimiento de sus troncos a lo largo de los días.

Dados los dos vectores con el crecimiento del tronco de los árboles ordenados de mayor a menor según lo que han crecido, la relación de semejanza que se considera es que la ordenación por días sea parecida entre ambos. Para ejemplificar esto, consideremos la siguiente situación:

Supongamos que nuestro intervalo de tiempo es de tres días y para el árbol 1 el orden del crecimiento es `[1,3,2]`, es decir, el día en el que más ha crecido el árbol ha sido el primero, luego el tercero y por último el segundo. Vemos a simple vista que si el orden del vector del árbol 2 es `[1,3,2]` también, esto tiene que ser más semejante que si es `[2,3,1]`. Lo que no sabemos es si es más semejante que el vector del árbol 2 valga `[2,3,1]` o que valga `[1,2,3]`.

Para solucionar esto calcularemos a cuántos días gana cada día en cada uno de los dos árboles y guardaremos el mínimo de estos dos valores. Luego sumaremos los mínimos de todos los días y dividiremos por el valor que resultaría si todos los días coincidieran en ambos árboles. De esta forma se obtendrá un valor de semejanza entre los dos árboles.

Retomando nuestro ejemplo, en el árbol 1 el día que se ha crecido más es el primero así que el día 1 gana a los otros dos días y toma un valor de 3. El tercer día gana al segundo así que toma un valor de 2 y el segundo día es en el que menos crece el árbol por lo que no gana a nadie (sólo se gana a sí mismo) así que su valor es de 1. El vector correspondiente de ganancia queda [3,1,2].

Si el árbol 2 tuviera un vector de `[2,3,1]` tendríamos que el segundo día gana a los otros 2 así que su valor sería de 3, el tercer día gana al primero y su valor sería 2 y el primer día es en el que menos crece el árbol así que su valor es 1. Su vector correspondiente de ganancia sería [1,3,2]. Del mismo modo, si el árbol 2 tuviera un vector de `[1,2,3]` obtendríamos que el primer día gana al resto y queda con un valor 3, el segundo día gana al tercero y queda con un valor de 2 y el tercero no gana a ningún día así que toma un valor de 1. El vector de ganancia queda [3,2,1].

Ahora comparamos el vector de ganancia del árbol 1 con el vector de ganancia del primer caso del árbol 2. Tenemos que encontrar el mínimo de cada día entre los dos vectores y sumar los vectores obtenidos. La operación sería $sum(min([3,1,2],[1,3,2])) = sum([1,1,2]) = 4$. Procedemos a hacer lo propio con el vector de ganancia del árbol 1 y el segundo caso del árbol 2. La operación en este caso es $sum(min([3,1,2],[3,2,1])) = sum([3,1,1]) = 5$.

Si los dos vectores fueran iguales, tendríamos un valor total de 6 (la fórmula de la progresión aritmética desde 0 hasta el número de días que tenemos). Como resultado final tenemos un valor de $\frac{4}{6}$ para el primer caso y de $\frac{5}{6}$ para el segundo. Así que el vector del segundo caso (`[1,2,3]`) es más semejante al vector del árbol 1 (`[1,3,2]`) que el vector del primer caso (`[2,3,1]`).

#### 3.2.1.7. Contracciones en un árbol

Para esta función se tiene en cuenta otra característica que se produce en los datos de los árboles. La contracción se define como la mayor diferencia que se produce entre el máximo de las medidas del tronco de un árbol por la mañana y el mínimo de las medidas del tronco a lo largo del día. Esto ocurre ya que en las horas de luz el árbol crece más que en las horas de oscuridad, así que la contracción sería la diferencia que se obtiene entre esos dos picos de crecimiento.

El usuario tiene que dar el árbol sobre el que se quieren medir las contracciones, las fechas de inicio y fin y, opcionalmente, indicar si quiere sólo el día cuando se produzca la mayor contracción o los días en donde la contracción supera una determinada cantidad.

#### 3.2.1.8. Comparación de contracciones en varios árboles

En esta función se comparan las contracciones de dos árboles. Se usa la misma definición de semejanza que la ya utilizada para comparar el crecimiento de dos árboles.

#### 3.2.1.9. Búsqueda de patrones en un árbol

Esta función consiste en buscar subcadenas en un árbol semejantes a un patrón dado con un número máximo de discrepancias entre dos fechas seleccionadas. Es la función más específica del trabajo ya que permite buscar intervalos muy concretos en las medidas del árbol.

Por ejemplo, si queremos encontrar intervalos de dos minutos (4 medidas) en los que el árbol crece de forma constante una unidad en cada medida respecto a la anterior tendríamos que buscar el patrón `1 1 1`. Cada número del patrón indica la diferencia que se produce con la medida anterior así que para intervalos crecientes tenemos que buscar patrones con todos los datos positivos y para intervalos decrecientes patrones con los datos negativos.

Debido a como se devuelven los resultados de esta función se puede usar la búsqueda

de patrones para encontrar intervalos más grandes en donde los árboles se comportan de la misma forma a lo largo del tiempo. Por ejemplo, si queremos buscar los intervalos en donde las medidas del árbol crecen en una unidad cada 30 segundos nos bastaría con buscar el patrón `1` y observar si en los resultados se devuelven intervalos consecutivos.

Para usar esta función el usuario tiene que seleccionar el árbol sobre el que quiere buscar el intervalo y escribir las fechas de inicio y fin. En cuanto a los datos del patrón debe escribir los números del patrón separados por espacios, el número de discrepancias que admite en las subcadenas encontradas y los datos de división y tolerancia que permiten transformar los datos reales de las medidas de un árbol a datos enteros.

El número que se indique en el campo de *Division* dividirá a todas las medidas del árbol y el número que se indique en el campo *Tolerance* (entre 0 y 0,5) aproximará el valor final al entero anterior o al entero superior obtenido del dato de la medida tras la división. Para ello obtendrá los decimales del número y si estos decimales son menores que el valor dado en la tolerancia el dato irá al entero anterior. Si la unidad menos los decimales son menores que la tolerancia el dato irá al entero superior. Si no ocurre ninguna de estas dos cosas el dato se considerará como erróneo. Cabe destacar que si la tolerancia vale 0,5 no pueden existir datos erróneos.

### 3.2.2. Campos en la aplicación

Según la opción escogida en el desplegable de las funciones, aparecerán unos datos a rellenar u otros. Los campos a rellenar comunes a todas las funciones son:

- **Tree number**: número de árboles que se van a tratar en el estudio. Debe ser un entero positivo.
- **Tree list**: lista desplegable con los identificadores de los árboles que se consideran en la aplicación. Se deben marcar tantos árboles como se vayan a estudiar. Debe coincidir el número de árboles marcados con el campo *Tree Number*. Si no aparece ese campo, sólo se puede marcar un árbol y en caso de pulsar más de uno, éste se reemplaza por el anterior.
- **Initial time**: fecha inicial de los datos de los árboles que se consideran en el estudio. Su formato debe ser `dd/mm/aa hh:mm:ss` con $ss = 00$ o 30.
- **Final time**: fecha final de los datos de los árboles que se consideran en el estudio. Su formato debe ser `dd/mm/aa hh:mm:ss` con $ss = 00$ o 30.

Para las funciones relacionadas con el cálculo de distintos intervalos en el árbol (condición de homogeneidad y discrepancias), se tienen los siguientes campos adicionales:

- **Interval length**: longitud del intervalo que se considera. Debe ser un entero mayor que 0 la palabra `INF`. En el caso de poner `INF` se devuelve el intervalo más grande que cumple la propiedad pedida. Si hay varios intervalos de longitud máxima, se devuelven todos ellos.

- **Maximum difference**: máximo valor que se permite de un dato al siguiente. Debe ser un real mayor que 0.

- **Mismatches**: máximo número de errores que se admiten en el estudio. Debe ser un entero mayor o igual que 0 o menor que el tamaño del intervalo entre 2.

- **Jumps (1/0)?**: etiqueta para indicar la condición de homogeneidad que se quiere usar. Poniendo un 1 se admiten las variaciones grandes de las medidas entre varios árboles y con un 0 no. Este campo sólo aparece en la función donde varios árboles tienen que cumplir la condición de homogeneidad.

Para el cálculo de patrones, se tienen los siguientes campos adicionales:

- **Pattern**: patrón que se quiere buscar. Un patrón válido está compuesto por números enteros separados por espacios.

- **Mismatches**: máximo número de errores que se admiten en el estudio. Debe ser un entero mayor o igual que 0 y menor que el tamaño del intervalo entre 2.

- **Division**: valor por el que se quieren dividir los datos del árbol a la hora de calcular sus patrones. Debe ser un real distinto de 0.

- **Tolerance**: tolerancia que hay en los datos del árbol para ajustarlos a un número entero o a otro. Debe ser un real mayor o igual que 0 y menor o igual que 0,5.

Para las funciones que calculan el crecimiento del tronco y de la contracción cada día, aparecen los siguientes campos adicionales:

- **Min trunk size**: tamaño mínimo del tronco que se permite en el árbol sobre el que se busca. Debe ser un real mayor que o igual a 0 o la palabra `INF`. En el caso de poner `INF` se devuelve el día con el tamaño del tronco más grande.

- **Min contraction size**: tamaño mínimo de la contracción que se permite en el árbol sobre el que se busca. Debe ser un real mayor o igual a 0 o la palabra `INF`. En el caso de poner `INF` se devuelve el día con la contracción más grande.

Si no se cumple alguna de las restricciones en los distintos campos, al darle al botón *Compute* se devuelve un error que informa sobre ese fallo concreto.

### 3.2.3. Ejemplos de uso de la aplicación

A continuación, se describen varios ejemplos de uso de la aplicación.

#### 3.2.3.1. Datos de un árbol

Si el usuario selecciona la función *Data of a tree*, la ventana de la aplicación se muestra de la siguiente manera:

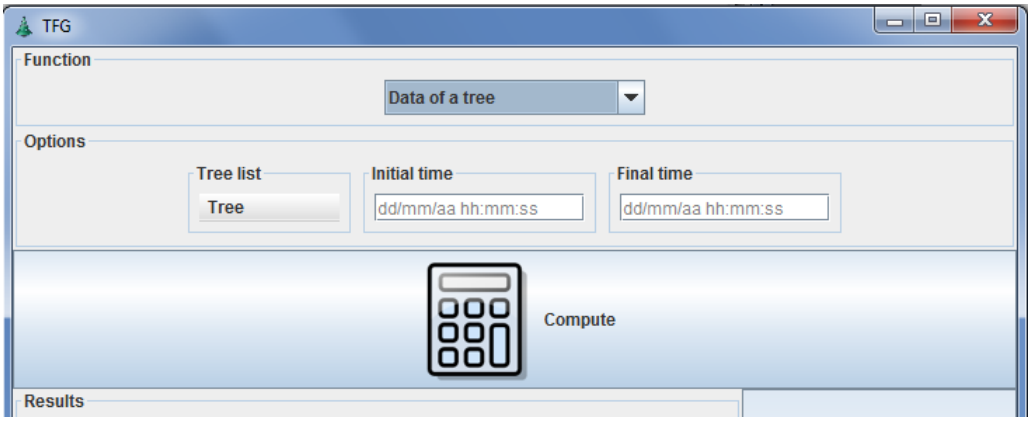


Figura 3.2: Función *Data of a tree*

Para mostrar los datos de un árbol, el usuario tiene que elegir el árbol que desea. Para ello, debe pulsar sobre el campo *Tree* obteniendo el siguiente desplegable:

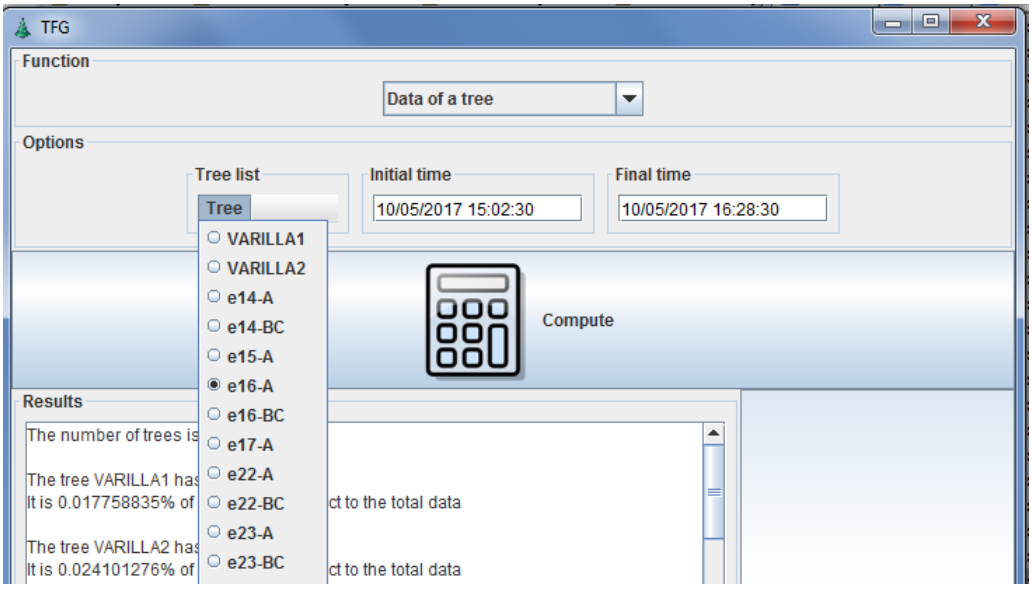


Figura 3.3: Desplegable con los árboles posibles

A continuación, tiene que rellenar los datos de las fechas de inicio y fin y pulsar sobre el botón *Compute*. Tras hacer esto, la ventana de la aplicación tendrá la siguiente forma:

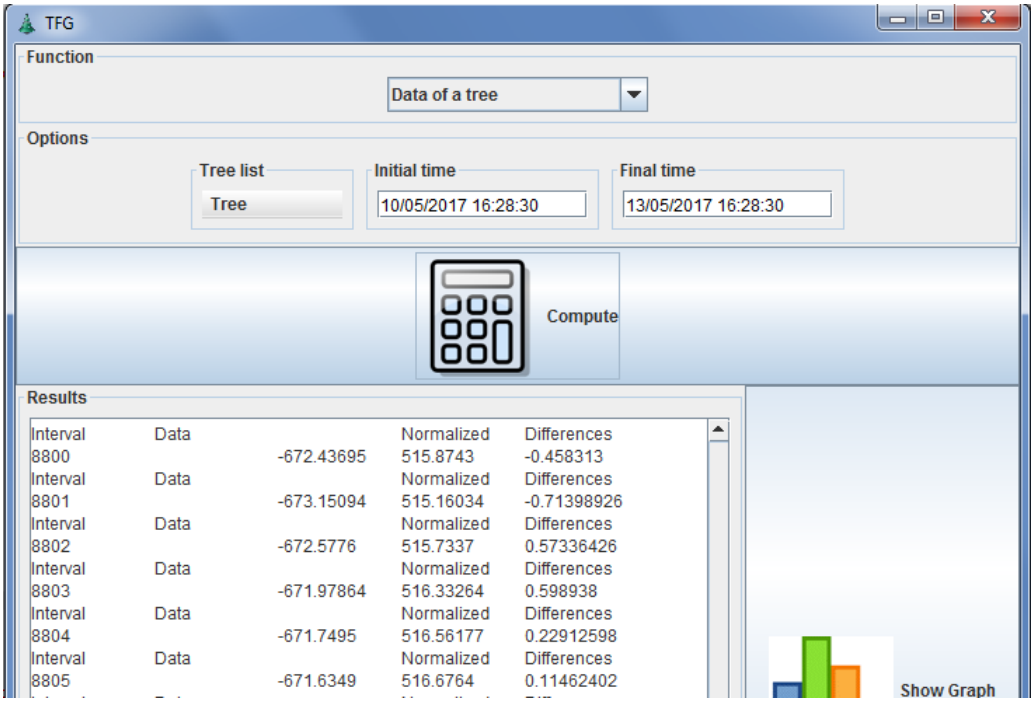


Figura 3.4: Resultados de la opción *Data of a tree*

#### 3.2.3.2. Condición de homogeneidad

Si el usuario selecciona la función *Data comparison*, la ventana de la aplicación se muestra de la siguiente manera:

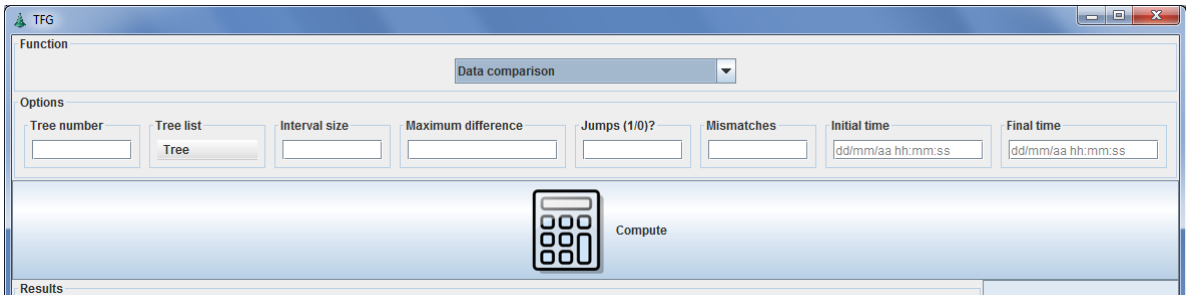


Figura 3.5: Función *Data comparison*

El usuario tiene que rellenar el campo *Tree number* con el número de árboles que desea comparar y elegir tantos árboles en el desplegable como el número que haya escogido. La ventana, por tanto, debe quedar de la siguiente forma:

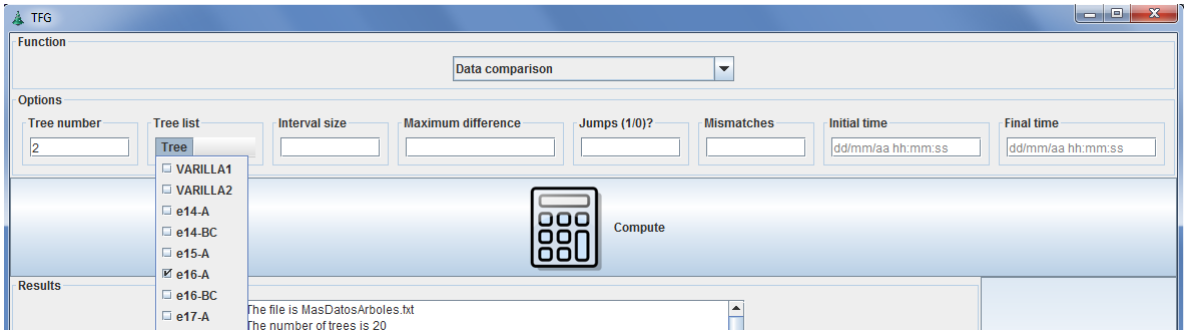


Figura 3.6: Desplegable con los árboles posibles

Tras esto, debe rellenar el resto de datos antes de darle al botón *Compute*. El campo *Interval size* se corresponde con el tamaño del intervalo que busca, el campo *Maximum difference* con la diferencia máxima que se permite entre un dato y el siguiente, el campo *Jumps* con la defínción de homogeneidad que quiere usar y el campo *Mismatches* con el número de errores que permite. Tras rellenar todos los campos y darle al botón *Compute*, el resultado de esta opción queda de la siguiente forma:

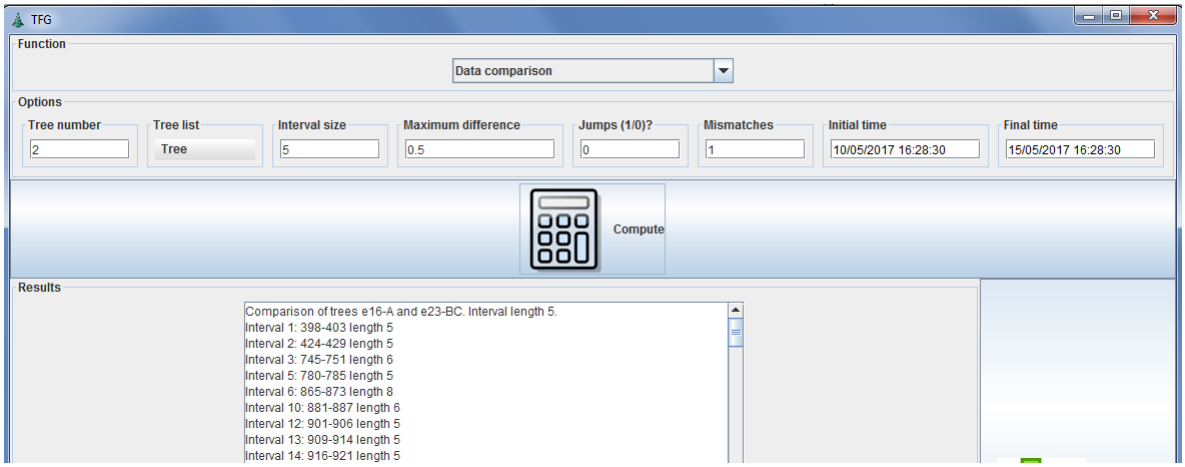


Figura 3.7: Resultados de la opción *Data comparison*

#### 3.2.3.3. Crecimiento del tronco de un árbol

Si el usuario selecciona la función *Growth of a tree trunk*, la ventana de la aplicación se muestra de la siguiente manera:

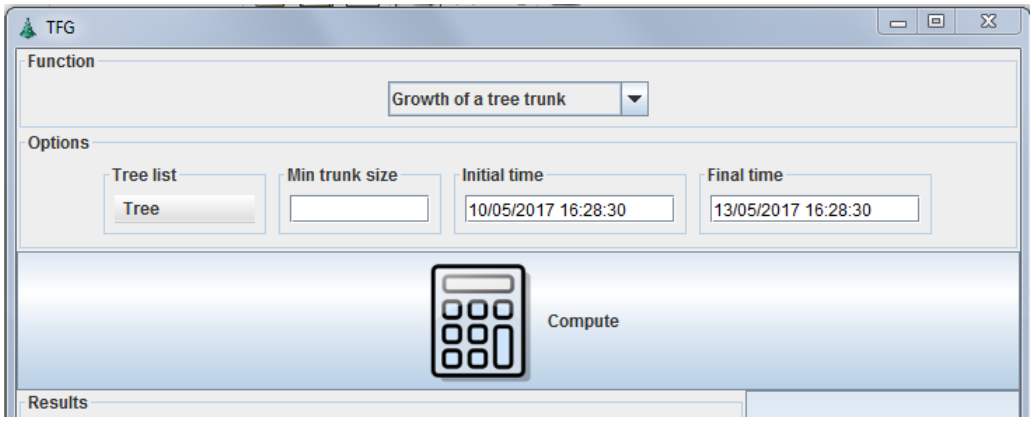


Figura 3.8: Función *Growth of a tree trunk*

El usuario tiene que elegir el árbol sobre el que realiza la búsqueda y rellenar los campos de *Initial time* y *Final time*. Además, deberá rellenar el campo *Min trunk size*, que se corresponde con el mínimo crecimiento del tronco que se permite en un día. Una opción válida en este campo es rellenarla con el valor `INF` para devolver el día en el que el árbol creció más. Si se elige este caso, el resultado de esta opción queda de la forma:

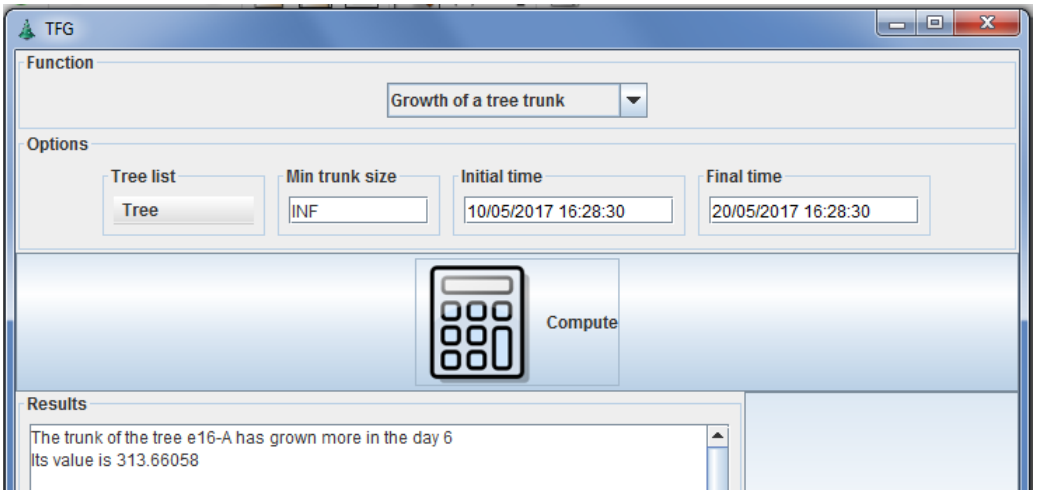


Figura 3.9: Resultados de la opción *Growth of a tree trunk*

#### 3.2.3.4. Patrones

Si el usuario selecciona la función *Patterns in a tree*, la ventana de la aplicación se muestra de la siguiente manera:

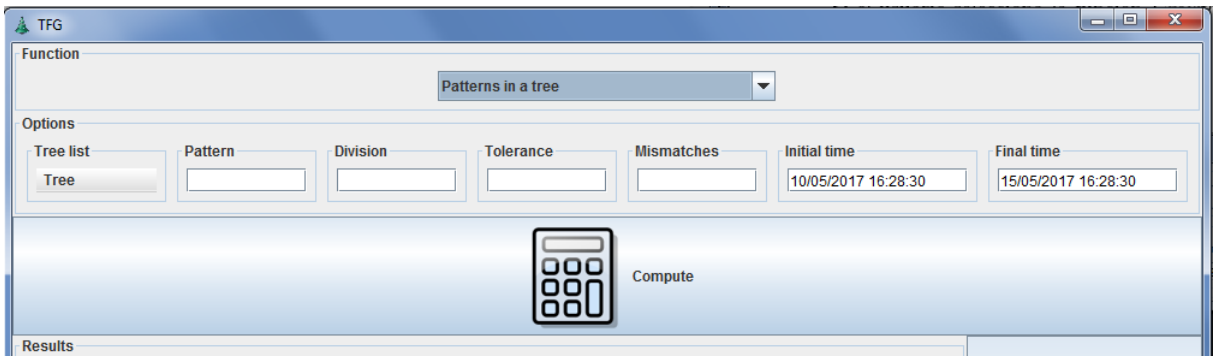


Figura 3.10: Función *Patterns in a tree*

El usuario tiene que elegir el árbol sobre el que realizar la búsqueda y rellenar los campos *Pattern* y *Mismatches* con el patrón que quiere y las discrepancias que admite. Ademas debe rellenar los campos *Initial time* y *Final time* de las fechas de inicio y fin de la búsqueda y los campos *Division* y *Tolerance* que se necesitan para adaptar el vector de medidas reales a uno de enteros. Tras rellenar todos los campos y dar al botón *Compute* el resultado queda:

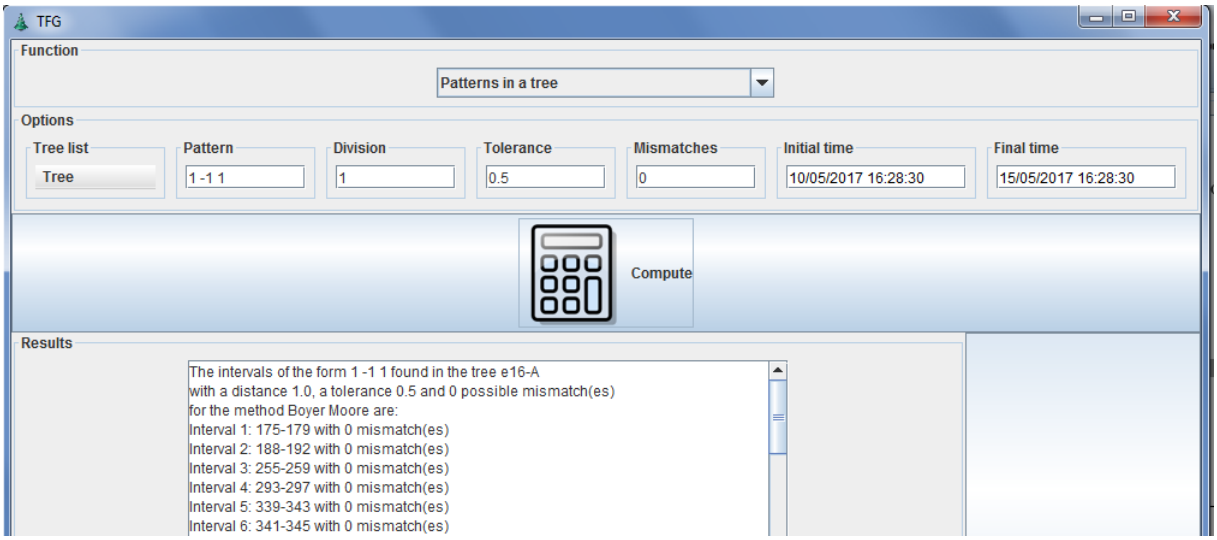


Figura 3.11: Resultados de la opción *Patterns of a tree*

## 3.3. Descripción de la aplicación. Implementación

### 3.3.1. Entrada de Datos

Los datos de la aplicación vienen de una hoja de Excel y tienen una forma como la de la siguiente imagen:

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | VARILLA1 | VARILLA2 | e14-A | e14-BC | e15-A | e15-BX | e16-A | e16-BC |
| 2 | 30/05/2017 0:00 | -2367,7111 | -312,4549 | 1565,97465 | 2166,09642 | 1307,42286 | 1861,18941 | 1191,78975 | 763,750078 |
| 3 | 30/05/2017 0:00 | -2366,5278 | -312,10118 | 1564,8198 | 2163,63635 | 1307,42286 | 1861,18941 | 1189,47559 | 763,164381 |
| 4 | 30/05/2017 0:01 | -2365,3446 | -312,4549 | 1565,97465 | 2163,63635 | 1307,42286 | 1861,18941 | 1189,47559 | 763,164381 |
| 5 | 30/05/2017 0:01 | -2366,766 | -312,05637 | 1564,56767 | 2162,92131 | 1306,24818 | 1860,69335 | 1190,71896 | 763,297939 |
| 6 | 30/05/2017 0:02 | -2366,5278 | -312,69072 | 1565,97465 | 2163,63635 | 1307,42286 | 1861,18941 | 1190,63267 | 763,750078 |
| 7 | 30/05/2017 0:02 | -2367,7111 | -312,33699 | 1565,97465 | 2162,40631 | 1307,42286 | 1862,36663 | 1189,47559 | 763,047241 |
| 8 | 30/05/2017 0:03 | -2366,766 | -312,05637 | 1564,56767 | 2161,69238 | 1307,42604 | 1860,69335 | 1190,71896 | 763,06387 |
| 9 | 30/05/2017 0:03 | -2366,5278 | -312,33699 | 1565,97465 | 2163,63635 | 1307,42286 | 1861,18941 | 1189,47559 | 763,047241 |
| 10 | 30/05/2017 0:04 | -2365,5838 | -312,64537 | 1565,72148 | 2162,92131 | 1307,42604 | 1861,86951 | 1190,71896 | 763,649042 |
| 11 | 30/05/2017 0:04 | -2367,7111 | -312,4549 | 1565,97465 | 2163,63635 | 1308,60178 | 1862,36663 | 1190,63267 | 764,218637 |
| 12 | 30/05/2017 0:05 | -2367,7111 | -312,10118 | 1565,97465 | 2163,63635 | 1308,60178 | 1863,54386 | 1189,47559 | 763,984358 |
| 13 | 30/05/2017 0:05 | -2367,4736 | -312,14607 | 1567,38417 | 2164,35267 | 1308,59966 | 1862,86465 | 1190,54623 | 764,672012 |
| 14 | 30/05/2017 0:06 | -2368,6579 | -312,61812 | 1567,38417 | 2165,58381 | 1308,59966 | 1862,86465 | 1190,54623 | 764,906502 |
| 15 | 30/05/2017 0:06 | -2366,5278 | -312,33699 | 1567,1295 | 2163,63635 | 1308,60178 | 1863,54386 | 1189,47559 | 763,632939 |
| 16 | 30/05/2017 0:07 | -2365,3446 | -311,86536 | 1564,8198 | 2162,40631 | 1308,60178 | 1861,18941 | 1189,47559 | 764,218637 |
| 17 | 30/05/2017 0:07 | -2368,6579 | -312,3821 | 1566,22828 | 2164,35267 | 1309,77964 | 1864,04293 | 1190,54623 | 764,437522 |
| 18 | 30/05/2017 0:08 | -2367,7111 | -312,33699 | 1567,1295 | 2163,63635 | 1308,60178 | 1862,36663 | 1191,78975 | 764,804335 |
| 19 | 30/05/2017 0:08 | -2367,4736 | -312,61812 | 1566,22828 | 2164,35267 | 1308,59966 | 1864,04293 | 1190,54623 | 764,672012 |
| 20 | 30/05/2017 0:09 | -2365,5838 | -312,17417 | 1564,56767 | 2164,15024 | 1308,6039 | 1863,04568 | 1190,71896 | 764,468282 |
| 21 | 30/05/2017 0:09 | -2365,3446 | -312,33699 | 1567,1295 | 2162,40631 | 1308,60178 | 1862,36663 | 1190,63267 | 764,335776 |
| 22 | 30/05/2017 0:10 | -2368,8944 | -312,4549 | 1567,1295 | 2163,63635 | 1308,60178 | 1863,54386 | 1191,78975 | 765,390033 |
| 23 | 30/05/2017 0:10 | -2365,3446 | -312,10118 | 1567,1295 | 2163,63635 | 1308,60178 | 1864,72108 | 1190,63267 | 764,218637 |
| 24 | 30/05/2017 0:11 | -2366,5278 | -312,33699 | 1567,1295 | 2166,09642 | 1308,60178 | 1863,54386 | 1191,78975 | 765,390033 |

Figura 3.12: Datos de los árboles en Excel

Para procesar estos datos se define la clase *FicheroMaestro*. Esta clase se encarga de, dada una entrada de datos como la anterior guardada en un fichero de texto separado por tabuladores, crear un fichero diferenciado para cada árbol que se puede utilizar en la aplicación.

El constructor de esta clase sólo necesita el nombre del fichero que contiene los datos. Su método principal se llama *arreglarFichero* y realiza el proceso que se ha descrito. Para ello, primero llama a un método auxiliar que intercambia todas las comas de los datos de los árboles por puntos (método *cambiarComasPorPuntos*). Este método es necesario para transformar el formato de Excel que utiliza comas en los números reales por el formato de Eclipse que utiliza puntos. No puede realizarse a mano debido al gran tamaño que tiene el fichero.

Tras esto se lee la primera línea del fichero, que contiene los nombres de todos los árboles, y se crea un fichero auxiliar para cada árbol que se rellena con los datos en bruto de ese árbol. Para ello se recorre el fichero de datos de principio a fin y para cada línea se escribe la fecha en todos los ficheros de árboles y el dato que le corresponde a cada árbol en esa línea. Tras realizar esto, se llama al método auxiliar *arreglarArbol* para cada uno de los ficheros que se han creado.

El método *arreglarArbol* lee los datos del fichero de un árbol en bruto y crea otro fichero con los datos del árbol corregidos, que son los que se usarán en la aplicación. Este método corrige tres posibles errores:

1. Añade las fechas que falten en los datos de los árboles debido a que el sensor no las ha tomado.

2. Corrige saltos muy grandes entre un dato y el siguiente debido a un fallo del sensor.

3. Elimina valores nulos en donde ha fallado el sensor.

Cuando se produce uno de estos errores en los datos de un árbol, se suma 1 a un contador de errores. Al finalizar, se crea un objeto de la clase *Errores* para cada árbol arreglado.

La clase *ListaArboles* es una clase auxiliar que contiene el nombre del árbol, el número de errores que se han producido y el porcentaje de errores respecto del total de los datos del árbol. Tiene una función *getValido* que devuelve si el árbol se puede utilizar para los cálculos de las distintas funciones o no. Se considera que un árbol no es válido cuando su porcentaje de errores es mayor que el 20 %. Aparte de este método, se define una función *compareTo* que compara el nombre del árbol con el nombre de otro árbol de otro objeto de la clase *ListaArboles*. Este método se usa para ordenar alfabéticamente distintos objetos en una lista de la clase *ListaArboles*.

### 3.3.2. Modelo-Vista-Controlador

Para la implementación del programa se usa MVC (Modelo-Vista-Controlador). El MVC es un patrón de arquitectura software que separa el manejo de los datos de su presentación al usuario y de la llamada *lógica de negocio*.

En MVC se distinguen tres componentes:

- **Modelo**: contiene los datos del dominio de la aplicación e implementa la funcionalidad que gestiona su comportamiento. Puede recibir consultas sobre su estado y solicitudes para cambiarlo. La implementación de esas solicitudes debe garantizar que las restricciones del dominio se siguen cumpliendo.

- **Vista**: consiste en una representación (visual) del modelo, por lo tanto es el componente que gestiona la salida gráfica y/o textual de la aplicación. La vista puede decidir destacar ciertos elementos del modelo e incluso ignorar y no presentar otros.

- **Controlador**: originalmente el controlador era el que interpretaba la entrada del usuario (en su forma más primitiva de pulsaciones de teclas y movimientos de ratón) y lo traducía a acciones sobre el modelo. Hoy por hoy el controlador realiza una tarea similar a la original pero sin lidiar normalmente con aspectos de tan bajo nivel. En ciertas instanciaciones del MVC, el controlador además de solicitar modificaciones al modelo también puede solicitarlas a las vistas.

Para aplicar el patrón MVC al problema se han seguido dos etapas:

1. Aplicación del patrón MVC a una interfaz textual sobre la consola.

2. Aplicación del patrón MVC a una interfaz gráfica sobre la que interactúa el usuario final.

Para cada una de estas etapas se ha construido un controlador y una ventana. La parte del modelo es la misma en ambas partes.

### 3.3.3. Modelo de la aplicación

En el modelo de la aplicación se encuentra toda la lógica de las distintas funciones explicadas en el Capítulo 2. Su clase principal es la clase *Programa*. La instancia de esta clase será la que se relacione con la vista y el controlador en el patrón MVC.

La clase *Programa* contiene los métodos necesarios para ir realizando las distintas acciones que permiten calcular los distintos resultados sobre los árboles que se tienen. Su constructor necesita el número de árboles que hay, las fechas de inicio y fin de los datos de los árboles, el nombre del fichero sobre el que se leen todos los árboles de la aplicación y el texto inicial que se muestra al inicio con todos los errores de los datos de los árboles.

Los principales métodos de la clase *Programa* son:

- `cargarArbol`: dado un nombre de árbol, su fecha de inicio y su fecha de final se construye un objeto de la clase *Arbol*. Los datos de ese árbol se leen en el fichero correspondiente (en el fichero asociado a ese árbol). Este método llama a los métodos auxiliares `calculo0` y `diferencias` para calcular el vector con los valores normalizados y el vector con las diferencias del árbol que se quiere añadir a la aplicación.

- `existeArbol`: dado un nombre de árbol, una fecha de inicio y una fecha de fin se comprueba si los intervalos de las fechas son correctos y después se llama al método `cargarArbol`. En caso de éxito se devuelve el árbol en cuestión y en otro caso se devuelve un valor nulo.

- `ejecutarFuncion`: dado un objeto *Funcion* se ejecuta su método `calculo` y se llama al método `onResultado` de todos los objetos de la lista de observadores de la clase *Programa*. Este método es general y funciona para cualquier tipo de *Funcion*.

Además de estos métodos, la clase *Programa* define la interfaz *Observer* con los métodos que tienen que implementar las clases que funcionen como vistas en la aplicación. Para indicar a estos observadores algún cambio en la clase se llaman a los métodos de la interfaz en los distintos métodos que de la clase *Programa*.

Para añadir las distintas funciones que se van a tener en cuenta en nuestro programa se define la clase abstracta *Funcion*. Esta clase tiene como atributos los árboles y las fechas, que son comunes a todas las funciones que se van a definir como hijas de esta clase, así como sus *getters* y *setters*. De esta forma se pueden definir algunos métodos generales que sirven para cualquier tipo de función. Por ejemplo se define el valor máximo o mínimo en los árboles considerados en un intervalo de fecha dados (esta función será usada para la gráfica en donde se muestran los datos de los árboles).

Los dos métodos más importantes de la clase abstracta *Funcion* son el método abstracto `calculo` y el método abstracto `toString`. Cada una de las clases hijas que se definan tendrán que implementar estos dos métodos obligatoriamente. El método `calculo` se refiere a la lógica que hay que realizar para cada función y el método `toString` al resultado

que se devuelve de cada función realizada. Además se definen algunos *getters* y *setters* como `getIndices`, `setIndices` y `setTam` que, aunque no tienen sentido para algunas de las funciones hijas, se necesitan definir en esta clase ya que es a la única a la que pueden acceder las vistas de la aplicación.

Otras clases que aparecen en el modelo de la aplicación son:

- **Arbol**: define los atributos, *getters* y *setters* necesarios para manejar un árbol en la aplicación. Cada árbol se caracteriza por su nombre, sus datos y por los intervalos de fecha de inicio y fin.
- **Fecha**: define una fecha a partir de un *String* con formato de fecha. Contiene todos los atributos, *getters* y *setters* necesarios para manejar una fecha, un método `toString` para mostrarla por pantalla y un método `mayor` que nos indica la relación de orden que existe entre dos fechas distintas.
- **Intervalo**: define una clase con un atributo entero y un atributo real, así como un método para comparar dos objetos de esa clase. Es utilizada para representar las discrepancias, las contracciones y los troncos de los árboles.
- **Resultado**: define una clase con dos enteros así como sus getters. Es utilizada para devolver los intervalos resultado en alguna de las funciones a las vistas por medio de su índice inicial y su tamaño.
- **Utils**: implementa distintos métodos auxiliares en distintas partes de la lógica de la aplicación. Se caracteriza por tener métodos estáticos, que pueden ser llamados en cualquier parte sin necesidad de crear una instancia de la clase. Dentro de esta clase hay distintos métodos para utilizar objetos de la clase *Fecha*: `medidasEntreFechas`, `medidasEnUnDia`, `diasHastaEseMes`, `FechaValida`... También se tienen los métodos `isInteger` e `isFloat` para determinar si un objeto es un número entero o un número real y un método para determinar si un *String* sigue el formato de un patrón o no (método `patronCorrecto`).

Además de clases, en el modelo de la aplicación hay dos enumerados:

- **EnumFuncion**: enumera las funciones que aparecen en la aplicación y define una función `toString` para mostrarlas por pantalla.
- **EnumPatron**: enumera los distintos tipos de funciones de patrones que aparecen en la aplicación y define una función `toString` para mostrarlos por pantalla.

### 3.3.4. Implementación de las funciones de la aplicación

Para cada una de las funciones se va a implementar una clase en Java que hereda de una clase abstracta principal (clase *Funcion*). De esta forma, todas las funciones van a funcionar igual exteriormente: tras obtener los valores de ciertos parámetros, se ejecuta la función (método `calculo`) y se desuelve su resultado (método `toString`). Esta forma de implementar las distintas funciones permite agruparlas más fácilmente a la hora de hacer la aplicación final, ya que sólo hay que diferenciar al principio, cuando se recaban los datos, y tras esto el proceso que se sigue es el mismo para todas ellas.

#### 3.3.4.1. Datos de un árbol

La implementación de esta opción se encuentra en la clase *FuncionDatos*. Su constructor recibe el árbol sobre el que se quieren mostrar los datos y las fechas de inicio y fin.

Esta opción tiene un método `calculo` nulo, que sólo se mantiene para que sea una clase análoga a la del resto de funciones. Su método `toString` devuelve los datos de los distintos vectores separados por tabuladores en una especie de tabla.

#### 3.3.4.2. Búsqueda de intervalos homogéneos en un árbol

La implementación de esta función se encuentra en la clase *FuncionIntervalo*. Su constructor recibe los distintos parámetros que necesita para funcionar:

- **Arbol**: objeto de la clase *Arbol* con el árbol sobre el que se va a realizar la función.
- **Ventana**: entero con la longitud del intervalo. Si su valor es `Integer.MAX_VALUE` quiere decir que el usuario ha marcado la opción de buscar el intervalo más largo que cumple la condición de homogeneidad.
- **ValorMax**: real con la cantidad que ha dado el usuario para definir la condición de homogeneidad.
- **Err**: número de errores que se permiten en el intervalo. El número de errores no puede superar la mitad de la longitud del intervalo.
- **Ini**: fecha de inicio del vector de datos sobre el que se busca.
- **Fin**: fecha de fin del vector de datos sobre el que se busca.

En cuanto al cálculo de la función, se va a implementar un método para cada una de las opciones posibles:

- `calculoIntervaloSinErrorExacto`: método que calcula los intervalos de longitud dada tal que los elementos no difieren en más de una cantidad. Utiliza el algoritmo de búsqueda de intervalos de tamaño fijo que cumplen una propiedad (sección 2.1.1).
- `calculoIntervaloConErrorExacto`: método que calcular los intervalos de longitud dada tal que los elementos no difieren en más de una cantidad permitiendo *err* errores. Utiliza la variante del algoritmo de búsqueda de intervalos de tamaño fijo que cumplen una propiedad que permite errores (sección 2.1.1).
- `calculoSegmentoMaximoSinError`: método que calcula el segmento de longitud maxima tal que los elementos no difieren en más de una cantidad dada. Utiliza el algoritmo del segmento máximo que cumple una propiedad (sección 2.1.2).
- `calculoSegmentoMaximoConError`: método que calcula el segmento de longitud máxima tal que los elementos no difieren en más de una cantidad dada permitiendo *err* errores. Utiliza la variante del algoritmo del segmento máximo que cumple una propiedad que permite errores (sección 2.1.2).

De esta forma, el método `calculo` elegirá usar una función u otra respecto de los parámetros `ventana` y `err`. El método de devolución de resultados `toString` realiza un bucle y devolverá todos los intervalos encontrados en el vector de datos del árbol seleccionado. Como particularidad, si varios intervalos se solapan, se agruparán en un intervalo de mayor tamaño.

#### 3.3.4.3. Búsqueda de intervalos homogéneos en varios árboles

La implementación de esta función se encuentra en la clase *FuncionComparar*. Su constructor recibe los mismo parámetros que la función anterior, más un parámetro adicional (`Saltos`) que es un booleano que indica si se quiere utilizar la primera versión (`false`) o la segunda (`true`) como condición de homogeneidad.

El cálculo de la función se implementa de forma parecida al de la función anterior, pero teniendo en cuenta cuál de las dos versiones de condición de homogeneidad se utiliza en cada función auxiliar. El método de devolución de resultados es exactamente igual que en el caso anterior.

La implementación de esta función se aplica sobre dos árboles y, para extenderla a un conjunto mayor, se llama una vez por cada pareja de árboles posible del conjunto. El resultado final será la intersección de todos los resultados encontrados.

Para poder devolver esta intersección, sólo permitida en la versión gráfica de la aplicación, la clase *MainWindow* definirá una serie de métodos para calcular la intersección de los índices de todos los árboles. Para poder realizar esta característica es necesario que la clase *Funcion* permita devolver los resultados encontrados en la función y permita cambiarlos si hace falta. Por esto se definen los métodos `getIndices`, `setIndices` y `setTam`. Los dos primeros devuelven y cambian el vector de índices de los intervalos resultado y el último cambia el tamaño del intervalo y se usa cuando estamos buscando el mayor intervalo entre muchos árboles.

Los métodos de la clase *MainWindow* que calculan esta intersección son:

- `interseccionFunciones`: calcula la intersección de los índices de varias funciones y los cambia para cada una de ellas.

- `maximoFunciones`: calcula el intervalo máximo de varias funciones en un vector de índices con los intervalos que pertenecen a la intersección.

- `interseccion`: devuelve la intersección entre dos vectores ordenados de manera creciente.

Además se define el método `escribirResultado` en la clase *MainWindow* para devolver en formato de texto el resultado de esta intersección.

#### 3.3.4.4. Búsqueda de intervalos con discrepancias en varios árboles

La implementación se encuentra en la clase *FuncionDiscrepancia*. Su constructor recibe los dos árboles, las fechas de inicio y fin, el número de errores y el parámetro asociado al intervalo que se quiere encontrar (`ventana`).

Su método `calculo` se realiza igual que en las dos funciones anteriores: se implementa una función auxiliar para cada uno de los cuatro problemas que se tiene que resolver y con un análisis de casos previo a partir de los parámetros `ventana`, `mayor` y `err` se elige a cuál de estas funciones llamar. En este caso hay que tener siempre en cuenta si estamos con dos medidas discrepantes entre sí o no.

El método de devolución de resultados es análogo a los dos anteriores, pero añadiendo el valor de la discrepancia hallada además del intervalo. Es decir, para cada intervalo encontrado, se devuelve su índice y la suma de los valores absolutos entre los datos discrepantes de los dos árboles, ya que esta suma acumulada nos da la idea de cómo de grande es la discrepancia entre ambos árboles.

#### 3.3.4.5. Crecimiento del tronco en un árbol

La implementación de esta función se encuentra en la clase *FuncionTronco* y su contructor recibe como parámetros el árbol sobre el que se quiere calcular, las fechas de inicio y fin y un real `cantidad` que se pone a `Float.MAX_VALUE` cuando el usuario ha indicado que quiere sólo el valor del día en el que más creció el árbol.

El método `calculo` consiste en encontrar la primera medida que se corresponde con un día nuevo (un valor de fecha terminado en 00:00) en el intervalo de tiempo dado e ir calculando cuánto crece el tronco cada día hasta llegar al final de un día (un valor de fecha terminado en 00:00) de un día posterior al intervalo de tiempo dado. Para esta función se utiliza el vector con los datos originales del árbol, no el vector de diferencias.

El método `toString` devuelve los valores del crecimiento de los troncos en los días que superen el valor del parámetro `cantidad`. Si `cantidad` vale `Float.MAX_VALUE`, sólo se devuelve el valor del día con mayor crecimiento.

#### 3.3.4.6. Comparación de los troncos en varios árboles

La implementación de esta función se encuentra en la clase *FuncionComparar Tronco* y sus parámetros son los dos árboles que se consideran y las fechas de inicio y fin.

El método `calculo` consiste en calcular los valores del crecimiento de los troncos para los dos árboles de la misma forma que se hacía para un solo tronco en la función anterior. El método `toString` calcula la relación de semejanza entre ambos troncos por medio del método auxiliar `parecidos`, que funciona de la misma forma que la explicación del ejemplo que se ha mostrado, y devuelve el resultado obtenido.

#### 3.3.4.7. Contracciones en un árbol

La implementación de esta función se encuentra en la clase *FuncionContraccion* y tiene como parámetros el árbol, las fechas de inicio y fin y el parámetro `cantidad` que funciona igual que en la función del crecimiento del tronco de un árbol.

El método `calculo` tiene el mismo mecanismo que la función en la que se calcula el crecimiento de los troncos: se busca el primer dato de un nuevo día y para cada día en nuestro intervalo de tiempos se calcula su contracción. La particularidad es que para el cálculo de la contracción se usa una función auxiliar en donde se busca el máximo de los datos hasta el mediodía y el mínimo de todo el día completo. Al igual que con el crecimiento del árbol, se usa el vector de datos originales del árbol y no el vector de diferencias.

El método `toString` es análogo al método `toString` del crecimiento del tronco de un árbol.

#### 3.3.4.8. Comparación de contracciones en varios árboles

La implementación se encuentra en la clase *FuncionCompararContraccion* y es completamente análoga a la función de comparar el crecimiento de dos árboles, pero calculando las contracciones en vez del crecimiento del tronco.

#### 3.3.4.9. Búsqueda de patrones en un árbol

La implementación de esta función se encuentra en la clase *FuncionPatron* y tiene como parámetros el árbol, el patrón que se quiere buscar, el número de discrepancias, las fechas de inicio y fin y los valores división y tolerancia para pasar las medidas del árbol de números reales a números enteros.

En esta clase tenemos todos los algoritmos para calcular búsquedas de subcadenas dentro de un vector. El método `calculo` manda que funcionen los algoritmos de Boyer Moore ya que son los que tienen mejores resultados para nuestra aplicación. El método `toString` devuelve en formato de texto los intervalos donde se ha encontrado el patrón buscado dentro del árbol.

Para poder implementar los algoritmos antes tenemos que pasar las medidas reales de los árboles a medidas enteras. Para ello se implementa el método `computoArbolPatron` que para cada medida real la convierte en una medida entera o en un error. Para realizar esto divide la medida por el valor dado en la variable `dist` y calcula los decimales que resultan tras quitarle la parte entera. Si estos decimales son menores que la variable `tol` se añade la parte entera del factor de la medida entre `dist`. Si la unidad menos estos decimales son menores que la variable `tol` se añade el siguiente número entero a la parte entera de media entre *dist* (según sea positiva o negativa). En otro caso la medida se convierte en un error que se representa con el valor `Float.MAX_VALUE`.

Otro método auxiliar clave de la clase *FuncionPatron* es `maxiVect` que calcula los

números máximo y mínimo del vector del árbol transformado por el vector anterior y del patrón. De esta forma podemos delimitar el alfabeto que utilizaremos en los algoritmos ya que hemos pasado del alfabeto infinito de números reales del vector de diferencias del árbol al alfabeto finito de números enteros en el intervalo definido por los números máximo y mínimo que hemos calculado.

A continuación describimos las características principales de las implementaciones de cada uno de los algoritmos de búsqueda de subcadenas:

- **Algoritmo de Fuerza Bruta**: implementado en los métodos `fuerzaBruta` y `fuerzaBrutaOpt`. Siguen el esquema dado en la figura 2.1 pero si el dato que se mira es erróneo (`Float.MAX_VALUE`) se suma uno al número de discrepancias encontradas.
- **Algoritmo de Boyer Moore exacto**: implementado en el método `BoyerMooreGoodSuffixRule`. Los métodos auxiliares `calculoZ`, `calculoL` y `calculol` calculan los valores de `Z`, `L` y `l'` para la regla de caracter bueno. Para la regla de caracter malo se calculan dos tablas: una para todos los valores positivos del alfabeto y otra para todos los negativos. Se sigue el pseudocódigo de la figura 2.5. Para la fase de búsqueda se sigue el pseudocódigo de la figura 2.4. Si el valor es un error se sale directamente del bucle y no se tiene en cuenta el valor del dato para el desplazamiento.
- **Algoritmo de Boyer Moore aproximado**: tenemos dos versiones de este algoritmo. Una implementada con matrices con el preprocesado basado en el pseudocódigo de la figura 2.5 que calcula la regla del caracter malo igual que en el caso exacto y otra implementada con tabla hash que no precisa de la función `maxiVect` para conseguir un alfabeto finito ya que sólo calcula el preprocesado para los valores del patrón y almacena un valor por defecto para el resto de números (incluido aquí el valor erróneo). En la fase de búsqueda similar al código de la figura 2.6 se tiene en cuenta si el valor es erróneo para aumentar el número de discrepancias y desplazar correctamente. La versión con tabla hash se implementa en el método `BoyerMoore` y la versión con matriz en el método `BoyerMooreMatriz`.
- **Algoritmo Shift-Add**: al igual que en el caso del Boyer Moore aproximado tenemos una versión con tabla hash y otra con matriz. En la versión con tabla hash solo se calcula el preprocesado para los valores del patrón y un valor por defecto que incluye el resto de valores. En la versión con matriz se calcula el preprocesado para todos los números posibles de nuestro alfabeto. Ambas versiones calculan la tabla `t` de forma similar a la figura 2.7. La fase de búsqueda primero calcula el valor del vector de estados para el dato que corresponde del vector (incluido cuando el dato es erróneo) y luego realiza la actualización de forma similar al código de la figura 2.8. La implementación de la versión con tabla hash se encuentra en el método `ShiftAdd` y la implementación de la versión con matriz en el método `ShiftAddMatriz`.

### 3.3.5. Vista de la aplicación

Para la parte de la vista en el patrón MCV se han creado dos clases principales: una vista para la parte de la consola y otra para la ventana de usuario. A continuación se

describen los aspectos más importantes de cada una de ellas.

#### 3.3.5.1. Vista de consola

La vista de la parte de consola está implementada en la clase *VistaDeConsola*. Esta clase implementa los métodos de la interfaz *Programa.Observer* que define la clase *Programa*. En su constructor se añade como observador de la clase *Programa*. Los distintos métodos que se implementan son:

- `onMostrar`: método que imprime por consola el menú con las distintas opciones que puede realizar el usuario. Normalmente este método es llamado al final del resto de métodos que implementa la clase *vistaDeConsola*.
- `onInic`: método de la interfaz *Programa.Observer*. Imprime por consola el texto que le pasan como parámetro y llama al método `onMostrar`.
- `onListar`: método de la interfaz *Programa.Observer*. Imprime por consola la lista de árboles que existen en el fichero que se le pasa como parámetro y llama al método `onMostrar`. Este método se usa para informar al usuario de los árboles disponibles que puede utilizar en el resto de funciones de la aplicación.
- `onError`: método de la interfaz *Programa.Observer*. Informa al usuario de un error y llama al método `onMostrar`.
- `onResultado`: método de la interfaz *Programa.Observer*. Devuelve al usuario el resultado de la función que ha escogido en el menú y la gráfica en una pestaña aparte con los datos de los árboles sobre los que se ha aplicado esa función en el intervalo de tiempo escogido. Llama también a la función `onMostrar`.
- `onMostrarGrafica`: método que inicializa los parámetros necesarios para mostrar la gráfica de uno o dos árboles en un intervalo de tiempo dado.

#### 3.3.5.2. Vista de ventana

La vista de la parte de la ventana está implementada en la clase *MainWindow*. Esta clase se encarga de construir la ventana gráfica e implementar los *listeners* necesarios para responder a las acciones que toma el usuario. Implementa los métodos de la interfaz *Programa.Observer* que define la clase *Programa* para devolver algunos resultados de forma gráfica.

El constructor de la clase *MainWindow* se encarga de añadir su instancia como observador de la clase *Programa*, de iniciar los botones que puede pulsar el usuario y de construir la ventana que se muestra al final a través de la unión de distintos objetos *JPanel*.

En cuanto a los componentes gráficos de Swing:

- Para los botones `Exit`, `Compute` y `Show Graph` se usa la clase *JButton*.

- Para las distintas etiquetas donde escribe el usuario se usa la clase *JTextField*.
- Para el cuadrado de texto de resultados se usa la clase *JTextArea*.
- Para la lista desplegable de las funciones disponibles se usa la clase *JComboBox*.
- Para la lista desplegable de árboles se usan las clases *JRadioButtonMenuItem* y *JCheckBoxMenuItem* según se pueda seleccionar sólo un árbol o más de un árbol, respectivamente.
- Para agrupar cada uno de los componentes se usa la clase *JPanel*.

En cuanto a los métodos de la clase *MainWindow*:

- `salir`, `calcular`, `sacarGrafica`: inicializan los botones `Exit`, `Compute` y `Show Graph`, respectivamente.
- `initDatos`: agrupa el panel de opciones.
- `initOpciones`: inicializa todos los componentes visuales de las opciones de las posibles funciones. Implementa el *listener* de la lista desplegable de opciones y añade unos paneles u otros según la opción elegida en esa lista. También implementa el *listener* del botón `Compute` según los valores que se encuentren en los paneles de opciones que se muestran (para ello tiene en cuenta el elemento escogido en la lista desplegable de funciones).

  Este *listener* consiste en mirar si los datos de la función escogida por el usuario son correctos. Si lo son se crea una función del tipo correspondiente y se le pide a *ControladorVentana* que la ejecute para poder mostrar los resultados al usuario. Si los datos no son correctos se devuelve un error.

  En este *listener* se calcula también la intersección de los resultados para la opción `Comparar` y se crean las gráficas correspondientes para los árboles y los resultados obtenidos que luego podrán verse si el usuario pulsa el botón `Show Graph`.
- `initResultados`: agrupa los paneles de resultados e implementa el *listener* del botón `Show Graph`.
- `calcResultados`: crea el panel que contiene el cuadrado de texto de resultados.
- `iniciarGrafica`: inicia la gráfica que va a mostrarse a continuación al usuario según los valores que ha escrito en los distintos campos de la función que quiere realizar.
- `agnadirGrafica`: añade a la gráfica creada los valores de un árbol entre las fechas de inicio y fin indicadas por el usuario.
- `agnadirTrozo`: añáde a la gráfica creada un intervalo resultado de la función que ha mandado calcular el usuario. Estos intervalos se superponen en negro encima de las gráficas de los árboles de forma que el usuario pueda saber que es un resultado de la función.

En cuanto a los métodos de la interfaz *Programa.Observer* implementados por la clase *MainWindow*:

- `onInic`: añade en el cuadro de texto de resultados el texto que le pasan como parámetro.
- `onListar`: no hace nada porque no tiene sentido en este tipo de vista.
- `onError`: añade en el cuadro de texto de resultado el texto de error.
- `onResultado`: añade en el cuadro de texto de resultados el texto que devuelve la función que se ha realizado.

### 3.3.6. Controlador de la aplicación

Para la parte del controlador en el patrón MCV se han creado dos clases: un controlador para la vista de la consola y un controlador para la vista de ventana del usuario. A continuación se describen los aspectos más importantes de cada una de ellas.

#### 3.3.6.1. Controlador de consola

El controlador de la consola está implementado en la clase *Controlador*. La función de este controlador es pedir acciones y datos al usuario a través de la consola y llamar a los métodos de la clase *Programa* necesarios para realizar esas acciones. La clase *Programa* será la que llame a las vistas por medio de su lista de observadores y se devolverán por consola los resultados al usuario.

Este controlador tendrá como atributos la instancia de la clase *Programa*, que representa al modelo, y la instancia de la clase *java.util.Scanner*, que se usa para leer datos por consola. Su método principal es `run`. Este método usa el método auxiliar `pedirDatos` para pedir toda la información necesaria al usuario para realizar una de las posibles funciones del programa.

El método `run` lee en un bucle la siguiente opción que quiere realizar el usuario y según la opción escogida realiza una determinada acción. De manera general, tras elegir una opción, se piden los datos necesarios al usuario para poder ejecutarla y luego, si estos son correctos, se llama al método `ejecutarFuncion` de la clase *Programa*. Una acción especial es la opción `Salir`. En este caso se llama a un método que pone el booleano que controla el bucle a cierto y se sale de la aplicación.

El método auxiliar `pedirDatos` diferencia la información que hay que pedir al usuario según una variable de tipo *EnumFuncion* que se inicializa en el método `run`. Para cada valor del enumerado se piden una serie de datos al usuario necesarios para realizar la acción que ha escogido en el menú.

#### 3.3.6.2. Controlador de ventana

El controlador de la ventana de usuario está implementado en la clase *ControladorVentana*. La función del controlador es relacionar la parte de las vistas con la parte del modelo y para ello es necesario que la instancia de la clase *Programa* sea un atributo de esta clase. En este caso, el controlador será un atributo de las clases que representen a las vistas, así que sus métodos estáran enfocados a devolver ciertos objetos o realizar ciertas funciones que necesita la vista pero que no puede pedir directamente al modelo de la aplicación.

Los métodos de la clase *ControladorVentana* son:

- `comprobarArbol`: dado un nombre de árbol, una fecha de inicio y una fecha de fin este método devuelve un objeto de la clase *Arbol* con los datos del árbol que se piden entre el intervalo de fechas seleccionado o un valor nulo que representa que se ha producido un error. La implementación del método consiste simplemente en llamar al método `existeArbol` de la clase *Programa*.

- `ejecutarFuncion`: dado un objeto de la clase *Funcion* se llama al método `ejecutarFuncion` de la clase *Programa* si el valor de la función no es nulo o se llama al método `error` de esta misma clase si lo es.

- `requestSalir`: este método cierra la ventana de la aplicación. Se llama cuando se pulsa el botón `Salir`.

- `getIni`: este método devuelve la fecha inicial de los datos de los árboles que hay en la aplicación.

### 3.3.7. Unión de los tres componentes

#### 3.3.7.1. Modelo con las vistas

Para relacionar las vistas con el modelo, la clase *Programa* define la interfaz interna *Programa.Observer* y extiende a la clase *Observable* con esa interfaz. La clase *Observable* es una clase genérica y su función consiste en añadir o eliminar instancias de objetos a una lista que tiene como atributo. La interfaz *Programa.Observar* define cuatro métodos: `onInic`, `onListar`, `onError` y `onResultado`.

Al extender a la clase *Observable*, la clase *Programa* obtiene la lista de objetos como atributo y, al hacerlo por medio de la interfaz interna *Programa.Observer*, hace que todos los objetos de esa lista tengan que implementar esa interfaz. De esta forma, todos los objetos de la lista implementan los métodos de la interfaz *Programa.Observer* ya mencionados y, por medio de la lista de objetos, la clase *Programa* puede llamar a los métodos definidos por estos objetos cuando ocurren ciertos estados. En particular, el método `onInic` se llama en el reset de la clase *Programa*, el método `onListar` en el método `listaArboles`, el método `onError` cuando ocurre un error (método `error`) y el método `onResultado` en

el método `ejecutarFuncion` una vez se ha calculado el resultado de la función correspondiente.

Las vistas tienen que definirse como observadoras de la clase *Programa* y tienen que implementar los métodos de la interfaz *Programa.Observer* como mejor les convenga para responder a los cambios de estado que se van produciendo en la lógica de la aplicación. Cuando se produce un cambio de estado, la clase *Programa* llama al método correspondiente de todos los objetos de su lista de observadores y, de esta forma, se van produciendo los distintos cambios en las vistas.

#### 3.3.7.2. Controlador con el modelo

Para relacionar el controlador con el modelo, las clases que representan al controlador (*Controlador* y *ControladorVentana*) tienen como atributo la instancia de la clase *Programa*. Las clases del controlador se encargan de llamar a los métodos de la clase *Programa* para ir realizando las diferentes acciones de la aplicación.

#### 3.3.7.3. Vista con el controlador

En la parte de consola la vista y el controlador no tienen ninguna relación. En la parte de ventana, la clase que representa la vista (*MainWindow*) tiene como atributo a la clase que representa el controlador (*ControladorVentana*). En este caso, la vista necesita al controlador para que haga de intermediario con la instancia de la clase del modelo (*Programa*) ya que necesita algunos datos adicionales para implementar los *listeners* de sus distintos componentes.

#### 3.3.7.4. Clase Main

La clase *Main* está implementada de forma que puede ejecutarse a través de cualquier consola que pueda ejecutar un programa de Java. Para iniciar el programa de esta forma hay que especificar los siguientes argumentos:

1. **Tipo de interfaz**: si quieres que la interfaz sea por consola (console) o por ventana (window). El argumentos es `java tfg.main.Main -u console/window`.
2. **Fichero**: el nombre del fichero con los datos de los árboles. El argumento es `java tfg.main.Main -f nombreFichero.txt`.

Si se inicia con una interfaz como eclipse sólo hay que cambiar la variable booleana `consola` a `true` o `false` según se quiera y la variable `fichero` con el nombre del fichero que contiene los datos de los árboles.

En ambos casos, el programa lo primero que hace es arreglar el fichero por medio de la clase *FicheroMaestro* para pasar de tener un fichero a tener un fichero por árbol. A continuación genera un `String` con los datos de los errores de los árboles del fichero e inicia una instancia de la clase *Programa*.

Si se elige la opción de consola se inicia un objeto de la clase *Scanner*, un objeto de la clase *Controlador* y un objeto de la clase *VistaDeConsola*. Para poner la aplicación en funcionamiento se llama al método `run` de la clase *Controlador*.

Si se elige la opción de consola se inicia un objeto de la clase *ControladorVentana* y un objeto de la clase *MainWindow*. En este caso no hace falta hacer nada más para que la aplicación comience.

En ambos casos se llama al método `reset` de la clase *Programa* para que las vistas se inicien mostrando el número de errores que hay en cada árbol.

### 3.3.8. Gráficas de los árboles

Para poder mostrar las gráficas de los árboles se ha hecho uso de una librería llamada *Java Plotter* de código OpenSource, implementada por el profesor Olly Oechsle de la Universidad de Essex.

Se ha modificado esta librería permitiendo así poder pintar varias gráficas de árboles a la vez con más de un color para poder diferenciar los árboles representados (no soportado en la librería original). Además, se ha creado la clase *MuestraArbol* para representar como función continua los datos de un árbol en un intervalo de tiempo dado. Para ello se ha usado repetidamente la ecuación de la recta que pasa por dos puntos a partir del vector de datos normalizados de un árbol. De esta forma, pasamos de un vector de datos finito a una función de datos continuos.

En la figura 3.13 podemos ver un ejemplo de una de las gráficas de nuestra aplicación. Observamos que se han pintado 3 árboles distintos. Debajo de la imagen tenemos una leyenda con el color que le corresponde a cada árbol. Podemos elegir que sección de la gráfica queremos observar cambiando las coordenadas de los ejes y dando al botón `Update` o podemos guardar la imagen con el botón `Save`. Las líneas negras que van desde el eje hasta las gráficas de los árboles indican que en esos puntos se ha encontrado un intervalo resultado de la función que ha mandado calcular el usuario.

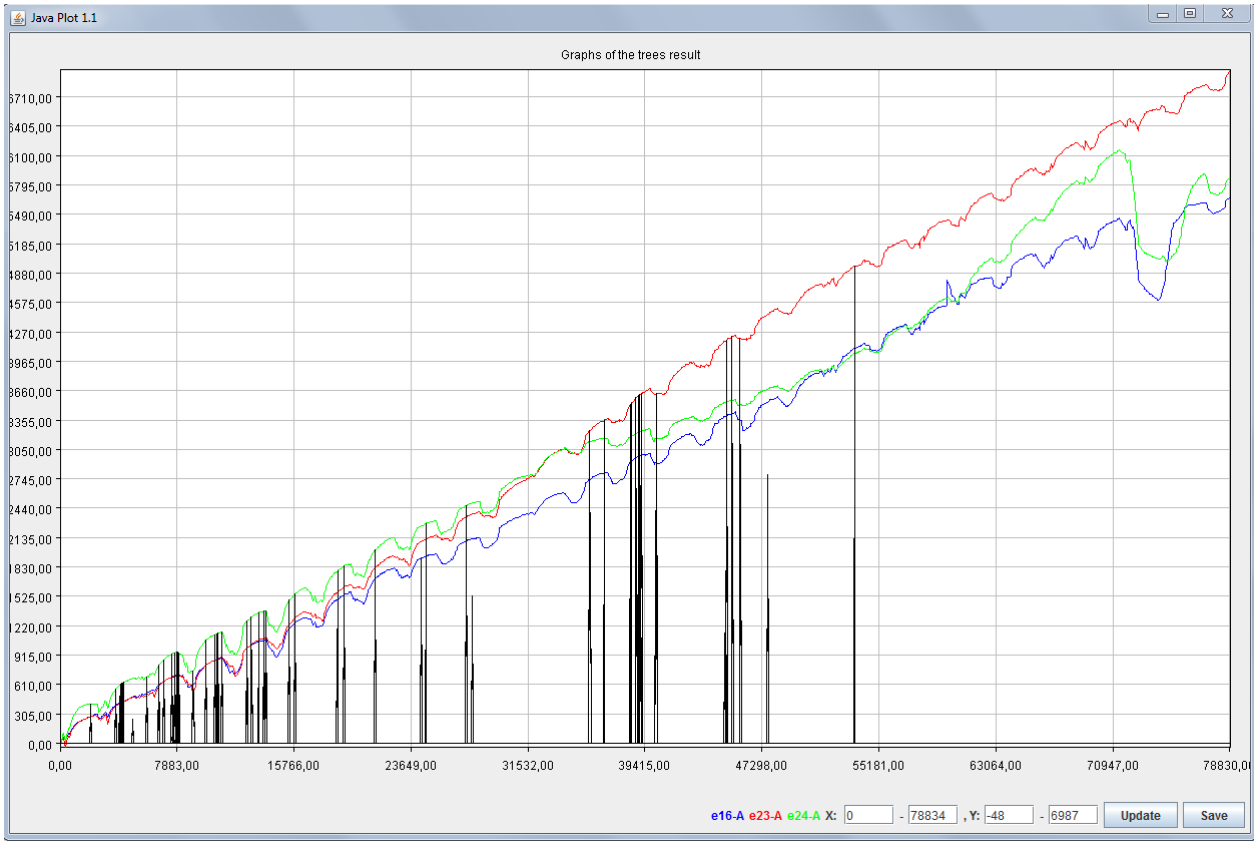


Figura 3.13: Gráfica de nuestra aplicación con 3 árboles

# Capítulo 4

# Análisis de tiempos

Se va a realizar un estudio de los tiempos de ejecución de los diferentes algoritmos de búsqueda de subcadenas para contrastar empíricamente la eficiencia de cada uno para nuestro problema.

## 4.1. Implementación

Se han implementado una serie de facilidades en la aplicación que nos permiten medir los tiempos de ejecución de los algoritmos. Se define el flag *tiempos* en la clase *Main* para indicar que se quiere realizar un análisis de los tiempos de los algoritmos de subcadenas. Si este flag se desactiva, la aplicación funciona igual que antes: si el flag *consola* está activado se inicia la vista de consola y si no lo está se inicia la vista de ventana. El comando para ejecutar el análisis de tiempos en consola es `java tfg.main.Main -u times`.

Para realizar el análisis de los tiempos hay que indicar una fecha de inicio, una fecha de fin, el árbol sobre el que se quiere realizar el análisis, el patrón que se quiere buscar, la división y la tolerancia para aproximar los datos del árbol a los del patrón y el número de discrepancias que se admiten. Además, hay que indicar cuántas repeticiones queremos que se hagan para cada algoritmo. Tras iniciar todos los datos, se crea una instancia de la clase *ControlTiempos* a la que le pasamos todos los datos como parámetro y se llama a su método principal (*tiempos*) que devuelve por pantalla el tiempo que ha tardado cada uno de los algoritmos que se consideran.

### 4.1.1. Clase ControlTiempos

La clase *ControlTiempos* tiene un constructor, que inicia los atributos con los parámetros dados por el usuario, y el método principal *tiempos*, que distingue entre los tres algoritmos que se han implementado (Fuerza Bruta, Boyer Moore y Shift Add) y llama a una función auxiliar para cada uno de ellos.

Para cada algoritmo implementado se realiza un bucle en el que se ejecuta tantas veces el algoritmo con los datos proporcionados como repeticiones haya pedido el usuario. Tras realizar esto se divide el tiempo total que se ha tardado en realizar el bucle por el número de

repeticiones y se devuelve al usuario el tiempo promedio de ese algoritmo en milisegundos.

Para el algoritmo de fuerza bruta se realiza un bucle distinto para la versión optimizada y la versión sin optimizar. La versión sin optimizar nos da una cota superior de los tiempos del resto de algoritmos porque es la peor forma de realizar la búsqueda.

Para los algoritmos Boyer Moore y Shift Add se realiza un bucle distinto para cada versión implementada (versión con matriz y versión con tabla hash) y se devuelve el tiempo que se tarda en cada caso teniendo en cuenta el tiempo con él preprocesado o sin él. El algoritmo de Shift Add solo se ejecuta si el tamaño del patrón y el número de discrepancias indicadas están dentro de los límites en donde se puede aplicar el algoritmo.

Por último, si el número de discrepancias es igual a 0 y el tamaño del patrón mayor que 1, se ejecuta otro bucle con la versión *Good Suffix Rule* del algoritmo Boyer Moore para ver si compensa su implementación respecto del algoritmo Boyer Moore que utiliza solo la regla de desplazamiento de carácter malo.

## 4.2. Experimentos con árboles

Vamos a considerar distintos patrones y distinto número de discrepancias para cada patrón. Primero estudiamos los tiempos en el caso de un patrón de tamaño pequeño (3 números), luego con uno un poco más grande (5 números) y por último con un patrón de tamaño 10.

Para cada tamaño de patrón generamos una tabla con el patrón que estamos considerando, el número de discrepancias que se piden, el número de ocurrencias encontradas en el árbol y los tiempos para todas las versiones de los algoritmos que consideramos. Para los algoritmos Shift-Add y Boyer Moore tenemos dos tiempos: el tiempo del preprocesado del algoritmo y el tiempo de la fase de búsqueda.

### 4.2.1. Tiempos para patrones de tamaño 3

Consideramos los patrones `0 0 0`, `1 1 1` y `1 -1 1` con cero y una discrepancia para dos árboles distintos: el *e16-A* y el *e23-A*, ambos con 78832 medidas. Los resultados se encuentran en las tablas 4.1, 4.2, 4.3 y 4.4.

Lo primero que observamos es que cuantas más ocurrencias del patrón existen en los datos del árbol, más tiempo tardan en ejecutarse los algoritmos. Por lo tanto, los tiempos son mayores cuando se busca el patrón con alguna discrepancia.

Para el algoritmo de fuerza bruta tenemos que la versión optimizada es bastante mejor que la versión sin optimizar y que esta mejora es inversamente proporcional al número de ocurrencias encontradas. Esto es porque cuantas menos ocurrencias haya en el árbol, menos veces es necesario comparar todo el patrón en el algoritmo optimizado. Observamos que se mejora entre un $7 - 9\,\%$ cuando hay más de 40000 ocurrencias. Con más de 2000

| Patrón | Disc | Ocurrencias encontradas | FB | FB opt | ShAdd (hash) | | ShAdd (matriz) | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Prep | Alg | Prep | Alg |
| 0\|0\|0 | 0 | 19800 | 10326 | 8711 | 22 | 8383 | 14 | 3492 |
| 0\|0\|0 | 1 | 41529 | 14206 | 13167 | 22 | 9073 | 14 | 6064 |
| 1\|1\|1 | 0 | 890 | 6497 | 3928 | 18 | 5717 | 12 | 1864 |
| 1\|1\|1 | 1 | 7726 | 7799 | 6798 | 22 | 6958 | 14 | 3242 |
| 1\|-1\|1 | 0 | 1307 | 6961 | 3899 | 24 | 5820 | 469 | 1882 |
| 1\|-1\|1 | 1 | 8732 | 8287 | 7422 | 24 | 7277 | 13 | 2441 |

Tabla 4.1: Tiempos para patrones de tamaño 3 en el árbol *e16-A*

| Patrón | Disc | Ocurrencias encontradas | BM (hash) | | BM (matriz) | | BM (exacto) | |
|---|---|---|---|---|---|---|---|---|
| | | | Prep | Alg | Prep | Alg | Prep | Alg |
| 0\|0\|0 | 0 | 19800 | 15 | 9298 | 12 | 2689 | 20 | 2743 |
| 0\|0\|0 | 1 | 41529 | 20 | 20890 | 12 | 5177 | - | - |
| 1\|1\|1 | 0 | 890 | 15 | 3595 | 12 | 1145 | 17 | 1129 |
| 1\|1\|1 | 1 | 7726 | 20 | 10979 | 13 | 3267 | - | - |
| 1\|-1\|1 | 0 | 1307 | 16 | 3984 | 45 | 1239 | 17 | 1187 |
| 1\|-1\|1 | 1 | 8732 | 20 | 9816 | 13 | 3315 | - | - |

Tabla 4.2: Tiempos para patrones de tamaño 3 en el árbol *e16-A*

ocurrencias del patrón en la cadena numérica, el algoritmo optimizado es entre un 10 % y un 20 % mejor, mientras que con menos de 2000 ocurrencias es entre un 60 % y un 80 % mejor.

Para el algoritmo Boyer Moore tenemos que las versiones exacta y con matriz son más eficientes que la versión con la tabla hash. Los tiempos de preprocesado son muy parecidos en las tres versiones siendo siempre superior el de la versión exacta al de la versión aproximada con matriz. Esto es debido a que en ambas se deben calcular las tablas de la regla de carácter malo pero en la exacta también se calculan los valores necesarios para la regla del sufijo bueno. Tanto la versión exacta como la versión aproximada con matriz son algo más del 200 % mejores que la versión aproximada utilizando la tabla hash. Esto es debido a que los accesos a las tablas hash en Java son muy costosos.

Los tantos por ciento de mejora entre los algoritmos de Boyer Moore no dependen del número de ocurrencias puesto que encontramos que la versión con matriz es un 214 % mejor que la de la tabla hash con 890 ocurrencias, un 85 % mejor con 20782 ocurrencias pero un 303 % mejor con 41529. Entre las versiones con matriz y exacta, quitando el dato anómalo del árbol *e23-A* (tabla 4.4) de que con 20782 ocurrencias la versión exacta es un 95 % mejor, obtenemos en media que la versión exacta es un 2 % mejor que la versión con matriz. El dato quitado corresponde con el patrón `0 0 0`, cuyas ocurrencias se van solapando ya que se corresponde con el patrón cíclico `0` que suele producirse asiduamente

| Patrón | Disc | Ocurrencias encontradas | FB | FB opt | ShAdd (hash) | | ShAdd (matriz) | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Prep | Alg | Prep | Alg |
| 0\|0\|0 | 0 | 20782 | 10279 | 8921 | 22 | 9306 | 7 | 2645 |
| 0\|0\|0 | 1 | 42798 | 14232 | 13065 | 21 | 9155 | 8 | 5303 |
| 1\|1\|1 | 0 | 857 | 6509 | 3657 | 22 | 6050 | 7 | 1892 |
| 1\|1\|1 | 1 | 7297 | 7978 | 6987 | 22 | 6875 | 7 | 2132 |
| 1\|-1\|1 | 0 | 1050 | 6656 | 3948 | 21 | 5766 | 7 | 1869 |
| 1\|-1\|1 | 1 | 7831 | 7863 | 6747 | 24 | 6873 | 7 | 3020 |

Tabla 4.3: Tiempos para patrones de tamaño 3 en el árbol *e23-A*

| Patrón | Disc | Ocurrencias encontradas | BM (hash) | | BM (matriz) | | BM (exacto) | |
|---|---|---|---|---|---|---|---|---|
| | | | Prep | Alg | Prep | Alg | Prep | Alg |
| 0\|0\|0 | 0 | 20782 | 18 | 9360 | 7 | 5061 | 14 | 2592 |
| 0\|0\|0 | 1 | 42798 | 20 | 21010 | 8 | 5303 | - | - |
| 1\|1\|1 | 0 | 857 | 13 | 2993 | 7 | 1183 | 11 | 1025 |
| 1\|1\|1 | 1 | 7297 | 19 | 10696 | 7 | 3189 | - | - |
| 1\|-1\|1 | 0 | 1050 | 17 | 3809 | 6 | 1153 | 12 | 1288 |
| 1\|-1\|1 | 1 | 7831 | 19 | 9153 | 7 | 3020 | - | - |

Tabla 4.4: Tiempos para patrones de tamaño 3 en el árbol *e23-A*

en los árboles. Para el estudio de estos patrones realizaremos un experimento posterior.

Para el algoritmo Shift-Add observamos de nuevo que la versión con tabla hash es mucho peor que la versión con matriz. En este caso el preprocesado también es superior en la versión con tabla hash. Los tantos por ciento de mejora entre los algoritmos de Shift-Add no parecen depender de las ocurrencias, aunque cuando hay más de 40000 ocurrencias la mejora desciende drásticamente: pasa del 189 % de mejora en media al 61 %. Teniendo en cuenta que hay 78832 medidas en total, podemos decir que este descenso de la mejora es debido a que es necesario realizar comparaciones en casi todos los elementos del árbol.

En cuanto a comparaciones entre algoritmos, las versiones de Boyer Moore parecen ser mejores que las versiones de Shift-Add para menos de 1500 ocurrencias. En particular un 38 % mejor con tabla hash y un 59 % mejor con matriz. Para más de 7000 ocurrencias, la versión con tabla hash del algoritmo Shift-Add es considerablemente mejor que la versión con tabla hash del algoritmo Boyer Moore. En particular, es un 67 % mejor. Las versiones con matriz para más de 7000 ocurrencias son parecidas siendo en algunos casos superior uno y en otros el otro. En media, para más de 7000 ocurrencias, queda que el algoritmo Shift-Add con matriz es mejor que el de Boyer Moore con matriz en un 16 %. Como conclusión podemos decir que el comportamiento para patrones de tamaño igual a 3 es similar en los dos algoritmos.

Respecto al algoritmo de fuerza bruta optimizado no parece haber una relación entre el número de ocurrencias y los tantos por ciento de mejora si se compara con el resto de los algoritmos, así que pasamos a indicar la media para cada par de algoritmos. Respecto a la versión con tabla hash del algoritmo Shift-Add, el algoritmo de fuerza bruta es mejor en un $1,4\,\%$. Respecto a la versión con matriz de este mismo algoritmo comprobamos que es bastante mejor que el algoritmo de fuerza bruta optimizado ya que en todos los casos se ejecuta en menos tiempo. En media, la versión con matriz del algoritmo Shift-Add es un $144\,\%$ mejor que el algoritmo de fuerza bruta. Para la versión con tabla hash del algoritmo de Boyer Moore tenemos que el algoritmo de fuerza bruta se ejecuta en menos tiempo para menos de 1100 ocurrencias pero tarda más para un número mayor de estas. En media, el algoritmo de fuerza bruta optimizado es un $12,5\,\%$ mejor. Por último, la versión con matriz del algoritmo de Boyer Moore siempre es mejor que el algoritmo de fuerza bruta optimizado, cuantitativamente es un $165\,\%$ mejor.

Observando los tiempos de las tablas de los dos árboles vemos que hay una similitud entre ellos, así que para valores del tamaño de patrón mayores solo mostramos los tiempos de los algoritmos en un árbol.

### 4.2.2. Tiempos para patrones de tamaño 5

En este caso consideramos los patrones `0 0 0 0 0` y `1 -1 1 -1 1` con cero, una y dos discrepancias para el árbol *e16-A* y 78832 medidas. Los resultados se encuentran en las tablas 4.5 y 4.6.

Observamos otra vez que los tiempos para los algoritmos son mayores cuantas más ocurrencias se encuentren del patrón en el árbol. En este caso si se observa que la mejora del algoritmo de fuerza bruta optimizado respecto al algoritmo sin optimizar va disminuyendo a medida que aumentan las ocurrencias encontradas ya que es del $189\,\%$ para 170 ocurrencias pero, si quitamos este valor, la media de mejora es del $37\,\%$.

| **Patrón** | **Disc** | **Ocurrencias encontradas** | **FB** | **FB opt** | **ShAdd (hash)** | | **ShAdd (matriz)** | |
|---|---|---|---|---|---|---|---|---|
| | | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| 0\|0\|0\|0\|0 | 0 | 12311 | 13757 | 8168 | 22 | 7448 | 14 | 2741 |
| 0\|0\|0\|0\|0 | 1 | 25498 | 14287 | 12372 | 25 | 8936 | 15 | 3653 |
| 0\|0\|0\|0\|0 | 2 | 41583 | 18346 | 16913 | 22 | 11434 | 15 | 3983 |
| 1\|-1\|1\|-1\|1 | 0 | 170 | 10642 | 3679 | 25 | 5734 | 14 | 1752 |
| 1\|-1\|1\|-1\|1 | 1 | 958 | 10533 | 6116 | 23 | 6121 | 13 | 1847 |
| 1\|-1\|1\|-1\|1 | 2 | 5585 | 11860 | 9518 | 23 | 6710 | 14 | 2027 |

Tabla 4.5: Tiempos para patrones de tamaño 5 en el árbol *e16-A*

En el algoritmo Shift-Add observamos que la versión con matriz es muy superior a la versión con tabla hash y la mejora es del orden del $229\,\%$ para menos de 10000 ocurren-

| Patrón | Disc | BM (hash) | | BM (matriz) | | BM (exacto) | |
|---|---|---|---|---|---|---|---|
| | | Prep | Alg | Prep | Alg | Prep | Alg |
| 0\|0\|0\|0\|0 | 0 | 17 | 8239 | 13 | 2987 | 22 | 2062 |
| 0\|0\|0\|0\|0 | 1 | 19 | 17199 | 13 | 5030 | - | - |
| 0\|0\|0\|0\|0 | 2 | 23 | 27491 | 14 | 9595 | - | - |
| 1\|-1\|1\|-1\|1 | 0 | 15 | 2556 | 29 | 919 | 19 | 810 |
| 1\|-1\|1\|-1\|1 | 1 | 18 | 7175 | 12 | 2110 | - | - |
| 1\|-1\|1\|-1\|1 | 2 | 23 | 13320 | 13 | 3900 | - | - |

Tabla 4.6: Tiempos para patrones de tamaño 5 en el árbol *e16-A*

cias y del orden del 167 % para más de 10000. Observamos que la diferencia se mantiene respecto a los patrones de tamaño 3.

En el algoritmo Boyer Moore observamos que la versión con matriz es superior a la de tabla hash y que no parece depender del número de ocurrencias encontradas ni del número de discrepancias propuesto. En media la versión con matriz es un 210 % mejor. Entre la versión con matriz y la versión exacta empezamos a ver que la versión exacta es superior y si depende del número de ocurrencias pasando de un 13 % para 170 a un 45 % para 12311. Podemos deducir que la aplicación de la regla del sufijo bueno que se aplica en el algoritmo exacto pero no en el algoritmo aproximado es mas efectiva cuando los patrones son más largos.

Comparando los algoritmos de Boyer Moore y Shift-Add tenemos que en la versión con tabla hash el algoritmo de Boyer Moore es un 55 % mejor en el primer caso con 170 ocurrencias. Para el resto de casos la versión del algoritmo Shift-Add es un $71,52$ % mejor en media llegando a alcanzar una mejora del 140 % cuando hay más de 40000 ocurrencias. Lo mismo ocurre para la versión con matriz, en el primer caso el algoritmo de Boyer Moore es un 47 % mejor, pero para el resto el Shift-Add es superior con un $58,2$ % y, al igual que en el caso anterior, para más de 400000 la mejora es del 140 %. En ambos casos comprobamos que la mejora aumenta a medida que aumentamos el número de discrepancias. Además, para un número de ocurrencias pequeño el algoritmo de Boyer Moore consigue superar al de Shift-Add cuando no admitimos que haya discrepancias.

Respecto al algoritmo de fuerza bruta optimizado, la versión con tabla hash del algoritmo Booyer Moore es peor en todos los casos menos en el primero donde es superior por un 43 %, para el resto es mejor el de fuerza bruta en un 22 %. Respecto a su versión con matriz es mejor el algoritmo de Booyer Moore siempre, siendo superior en un 300 % en el primer caso. Para el resto de casos la mejoría es del 145 % y desciende considerablemente para más de 400000 ocurrencias siendo sólo mejor por un 76 % cuando para el resto siempre ronda el 150 % de mejora. Como en un caso anterior, esto se explica debido a que hay 78832 medidas, por lo que en la mayoría de los casos no se producen grandes desplazamientos en el algoritmo de Boyer Moore.

En cuanto a fuerza bruta optimizado con Shift-Add con tabla hash tenemos que para 170 ocurrencias es mejor el algoritmo de fuerza bruta con un 36 % de mejora. Para el resto de casos el algoritmo Shift-Add es mejor con una ganancia del 27 %. Para la versión con matriz, el algoritmo Shift-Add siempre es mejor teniendo una media de ganancia del 243 %. Vemos en este caso que a medida que aumenta el número de discrepancias el algoritmo Shift-Add funciona mejor.

### 4.2.3. Tiempos para patrones de tamaño 10

En este caso consideramos el patrón constantemente 0 y el patrón alternando 1's y -1's con un máximo de 4 discrepancias para el árbol *e16-A* y 78832 medidas. Los resultados se encuentran en las tablas 4.7 y 4.8.

En este caso se observa claramente que a mayor número de discrepancias, mayor número de ocurrencias encontradas y mayores son los tiempos que tardan en ejecutar los distintos algoritmos. Todos los tiempos para el algoritmo de fuerza bruta superan los 20000 milisegundos.

| **Patrón** | **Disc** | **Ocurrencias encontradas** | **FB** | **FB opt** | **ShAdd (hash)** | | **ShAdd (matriz)** | |
|---|---|---|---|---|---|---|---|---|
| | | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 0 | 5715 | 21959 | 8236 | 24 | 6336 | 16 | 2045 |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 1 | 12092 | 23581 | 13329 | 26 | 7598 | 18 | 2223 |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 2 | 19016 | 24146 | 17961 | - | - | - | - |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 3 | 27201 | 24895 | 20306 | - | - | - | - |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 4 | 36520 | 28028 | 26045 | - | - | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 0 | 0 | 20193 | 3634 | 26 | 5680 | 15 | 1815 |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 1 | 17 | 20374 | 6008 | 28 | 5565 | 16 | 1772 |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 2 | 61 | 20202 | 8305 | - | - | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 3 | 230 | 20450 | 10745 | - | - | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 4 | 909 | 20420 | 13478 | - | - | - | - |

Tabla 4.7: Tiempos para patrones de tamaño 10 en el árbol *e16-A*

En este caso se observa muy claramente que el algoritmo optimizado disminuye su mejora sobre el de fuerza bruta a medida que aumenta el número de discrepancias y de ocurrencias encontradas. El número de ocurrencias determina la cuantía de la mejora inicial y, a partir de ahi, a mayor número de discrepancias menor es la mejora. Las discrepancias influyen más que las ocurrencias ya que para 909 ocurrencias y 4 discrepancias la mejora es del 51 % y para 5715 ocurrencias sin ninguna discrepancia la mejora es del 166 %. Más o menos la diferencia es que a una discrepancia más, la mejora se divide a la mitad. Como ejemplo, para el caso del patrón nulo tenemos la sucesión de mejoras $455\,\% - 239\,\% - 143\,\% - 90\,\% - 51\,\%$.

| Patrón | Disc | BM (hash) | | BM (matriz) | | BM (exacto) | |
|---|---|---|---|---|---|---|---|
| | | Prep | Alg | Prep | Alg | Prep | Alg |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 0 | 18 | 6523 | 13 | 2215 | 27 | 1799 |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 1 | 22 | 15655 | 13 | 4727 | - | - |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 2 | 24 | 25375 | 15 | 8286 | - | - |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 3 | 34 | 35575 | 14 | 11290 | - | - |
| 0\|0\|0\|0\|0\|0\|0\|0\|0\|0 | 4 | 30 | 42573 | 17 | 15861 | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 0 | 16 | 1580 | 16 | 562 | 25 | 488 |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 1 | 20 | 4255 | 22 | 1408 | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 2 | 22 | 8468 | 13 | 2687 | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 3 | 27 | 14073 | 13 | 4609 | - | - |
| 1\|-1\|1\|-1\|1\|-1\|1\|-1\|1\|-1 | 4 | 28 | 20577 | 14 | 6659 | - | - |

Tabla 4.8: Tiempos para patrones de tamaño 10 en el árbol *e16-A*

Para el algoritmo Shift-Add tenemos que la mejora de la versión con tabla hash respecto de la versión con matriz ronda el 219 %. Además observamos que los tiempos de ejecución son parecidos a los mostrados para tamaños del patrón inferiores. Esto se corresponde a lo estudiado en [3] ya que para un tamaño del patrón relativamente grande el tiempo ha permanecido invariable respecto al resto de patrones menores en el árbol sobre el que hemos buscado.

En cuanto al algoritmo Boyer Moore, la mejora de la versión con matriz frente a la versión con tabla hash se mantiene y es del 202 % en media. La versión exacta respecto a la versión con matriz es mejor con una ganancia del 19 % y respecto a la versión con tabla hash en un 247 %. Esto se corresponde con lo estudiado en [8] ya que para alfabetos grandes como el nuestro y patrones largos el algoritmo exacto es mejor que el aproximado.

Si comparamos los algoritmos Boyer Moore y Shift-Add comprobamos que para un número de ocurrencias pequeño (menos de 20) el algoritmo de Boyer Moore es mejor en ambas versiones y con valores parecidos: ganancia del 48 % para la versión con tabla hash y del 54 % para la versión con matriz. Para un número de ocurrencias alto es mejor el algoritmo Shift-Add siendo superior por un 110 % en ambas versiones para 12092 ocurrencias. Para 5715 ocurrencias sin discrepancias los valores son prácticamente los mismos en ambos algoritmos para las dos versiones. Comprobamos que para 0 discrepancias el algoritmo Boyer Moore exacto es mejor que el Shift-Add y la mejora va disminuyendo a medida que aumenta el número de ocurrencias encontradas.

En cuanto al algoritmo de fuerza bruta optimizado respecto al algoritmo Shift-Add encontramos que a medida que sube el número de ocurrencias el algoritmo Shift-Add es mejor en ambas versiones que el algoritmo de fuerza bruta. En el caso de tabla hash pasamos de que gane el algoritmo de fuerza bruta con un 36 % con 0 ocurrencias a que gane el algoritmo Shift-Add con un 75 % cuando tenemos 12092. En la versión con matriz el algoritmo Shift-Add siempre mejora con una proporción del doble al algoritmo de fuerza

bruta y esta mejora va aumentando hasta llegar a sixtuplicar el tiempo con 12092 ocurrencias (la mejora es del 500 %).

Por último, para los algoritmos de fuerza bruta optimizado y Boyer Moore observamos que el porcentaje de mejora depende del número de discrepancias considerado y del número de ocurrencias encontradas. Cuando no se admiten discrepancias el algoritmo Boyer Moore es mejor que el de fuerza bruta aunque el número de ocurrencias sea alto. Así tenemos que para 5715 ocurrencias la versión con tabla hash es un 26 % mejor y la versión con matriz un 271 % mejor. Para 0 ocurrencias las mejoras son mucho mayores en ambos casos y tenemos que la versión con tabla hash es un 130 % mejor que la de fuerza bruta y la versión con matriz un 546 % mejor. A partir de estos valores las mejoras van disminuyendo a medida que aumenta el número de discrepancias y van aparenciendo más ocurrencias. Para la versión con tabla hash el algoritmo de fuerza bruta pasa a ser mejor cuando hay dos discrepancias y pocas ocurrencias o cuando hay una discrepancia pero hay muchas ocurrencias. Las mejoras en cuanto al número de discrepancias suelen ser parecidas y no pasan del 40 % para las 4 discrepancias consideradas. En el caso de la matriz el algoritmo de Boyer Moore siempre es mejor que el de fuerza bruta pero la diferencia entre ambos algoritmos disminuye en un factor de división del $1,5$ aproximadamente por cada discrepancia que se añade.

### 4.2.4. Resumen comparativo de los tres tipos de patrones sobre los datos de un árbol

En las dos gráficas 4.1 y 4.2 podemos observar las diferencias de tiempo de los distintos algoritmos para los patrones nulo y cíclico `1 -1` con 0 discrepancias en el árbol *e16-A* para los tres tamaños de patrón estudiados. Las ocurrencias encontradas en el caso nulo pasan de 19800 para tamaño 3 a 5715 para tamaño 10. En el patrón cíclico `1 -1` pasamos de 1307 ocurrencias con tamaño 3 a ninguna con tamaño 10. Observamos que la cota superior dada por el algoritmo de fuerza bruta es muy superior a medida que aumenta el tamaño del patrón. El siguiente algoritmo que peor se comporta es la versión con tabla hash del algoritmo Shift-Add pero en este caso el algoritmo permanece constante en sus tiempos a medida que aumentamos el tamaño del patrón. Tras esto tenemos los algoritmos de fuerza bruta optimizado y la versión de tabla hash del algoritmo Boyer Moore, que mejora cuando aumenta el tamaño del patrón superando considerablemente al de fuerza bruta para tamaño 10 y alcanzado a la versión de matriz del algoritmo Shift-Add que ha permanecido también constante en todos los casos. Por último, observamos que las versiones exacta y de matriz de Boyer Moore son las más rápidas y que su diferencia para estos casos es ínfima.

## 4.3. Experimentos adicionales

### 4.3.1. Patrones cíclicos

Llamamos patrón cíclico a aquel patrón que se repite varias veces en la colección de diferencias de un árbol. De esta forma, cuando nos referimos el patrón cíclico `1 -1` de

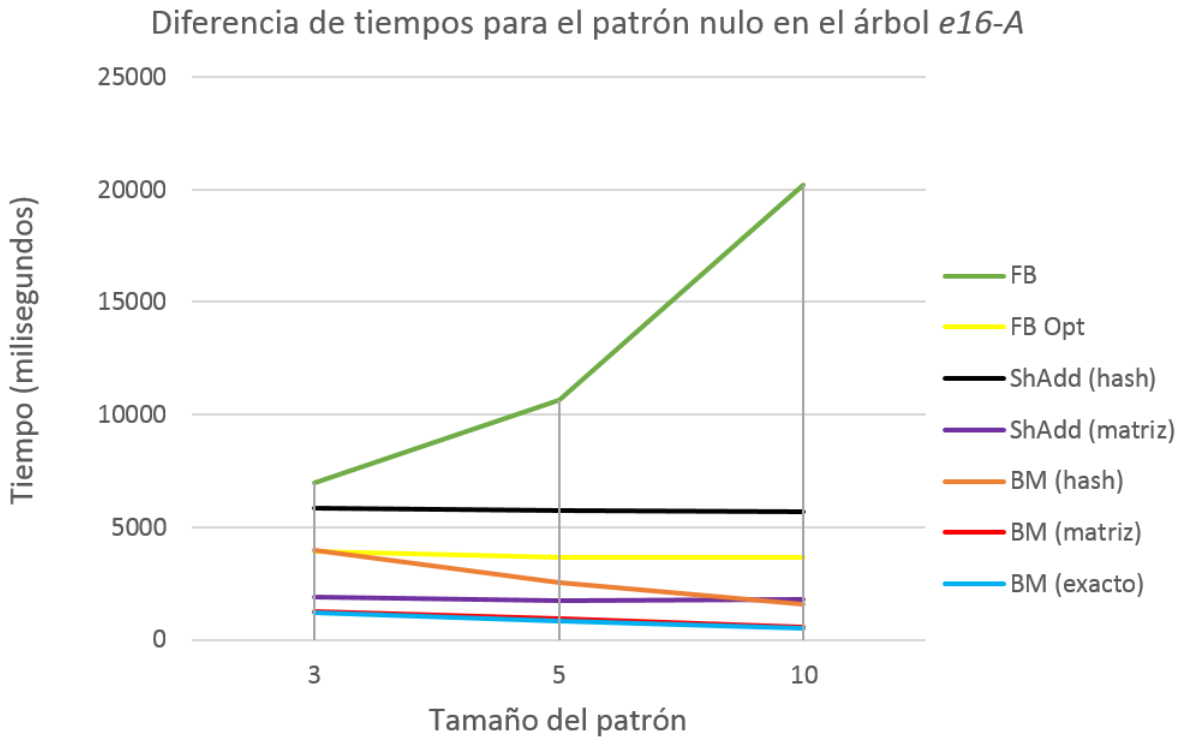


Figura 4.1: Gráfico con los tiempos de los algoritmos para el arbol *e16-A* con 0 discrepancias

tamaño 6, queremos decir que estamos estudiando el patrón `1 -1 1 -1 1 -1`.

En el estudio anterior observamos que con patrones cíclicos el algoritmo de Boyer Moore exacto obtiene mejores resultados que el algoritmo aproximado. Hemos realizado un experimento adicional para poder cuantificar esta mejora con mayor precisión. Para ello hemos creado varios ficheros ficticios que nos ayudarán a recabar los datos.

Para el estudio creamos un excel con 4 árboles ficticios con 78832 medidas y seguimos todos los pasos para hacer que funcionen en nuestra aplicación. El contenido de los 4 árboles creados es el siguiente:

- **Nulo 1**: árbol con todos los valores a 0. Este árbol nos permite estudiar el patrón cíclico `0`.
- **Nulo 2**: árbol con todos los valores a 0 pero con 10 valores discrepantes cada 1000 medidas. Este árbol nos permite estudiar el patrón cíclico `0` y la diferencia que supone tener menos ocurrencias que con el árbol *Nulo 1*.
- **Alternado 1**: árbol con valores 1's y -1's de forma alternada. De esta forma la colección de diferencias del árbol está compuesta por los valores 2 y $-2$ y podemos estudiar el patrón cíclico `2 -2`.
- **Alternado 2**: árbol con valores 1's y -1's de forma alternada pero con 10 valores

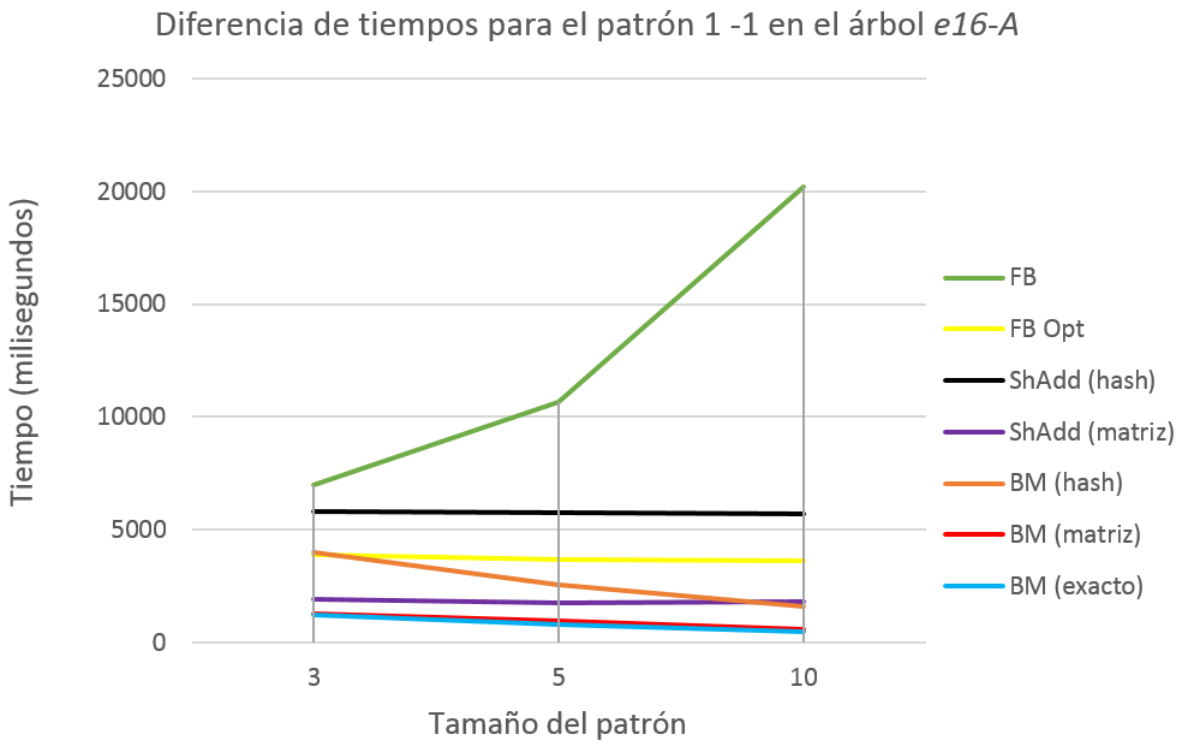


Figura 4.2: Gráfico con los tiempos de los algoritmos para el arbol e16-A con 0 discrepancias

discrepantes cada 1000 medidas. Estudiamos el patrón cíclico `2 -2` y la diferencia que supone tener menos ocurrencias que con el árbol *Alternado 1*.

Los valores discrepantes en los vectores *Nulo 2* y *Alternado 2* se corresponden con diferencias de tamaño 1 y con diferencias de tamaño 2 que se van alternado cada 1000 datos entre sí.

Para realizar el experimento pasamos a todos los árboles distintos tamaños de los patrones cíclicos `0` y `2 -2` pudiéndo estudiar el caso de tener el máximo número de ocurrencias y ninguna ocurrencia para cada árbol.

Para cada patrón se ha creado una tabla con los cuatro árboles y los tiempos para cada algoritmo para distintos tamaños del patrón cíclico. Además, tenemos varios diagramas auxiliares para ver la diferencia de tiempos de forma gráfica tanto cuando hay muchas ocurrencias en el vector como cuando no hay ninguna. En este segundo caso se utiliza un diagrama auxiliar quitando los algoritmos de fuerza bruta porque en comparación con el resto de algoritmos la diferencia de tiempo es muy considerable.

#### 4.3.1.1. Patrón cíclico `0`

Los resultados del estudio del patrón cíclico `0` se encuentran en las tablas 4.9 y 4.10 y en los diagramas 4.3, 4.4, 4.5 y 4.6. Las diferencias entre los vectores *Nulo 1* y *Nulo 2*

frente a *Alternado 1* y *Alternado 2* es que los tiempos de ejecución son mucho menores para los dos últimos debido a que como no se encuentran ocurrencias se producen todos los desplazamientos posibles.

Para el caso de los vectores *Nulo 1* y *Nulo 2* tenemos unas 78833 ocurrencias en el primer vector y unas 77821 en el segundo. Observamos que todos los tiempos aumentan a medida que pedimos buscar el patrón original repetido varias veces.

| **Árbol** | **Tamaño Patrón** | **FB** | **FB Opt** | **ShAdd (hash)** | | **ShAdd (matriz)** | |
|---|---|---|---|---|---|---|---|
| | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| **Nulo 1** | 3 | 48420 | 46359 | 22 | 13847 | 5 | 5723 |
| | 6 | 71616 | 70183 | 24 | 13922 | 4 | 8933 |
| | 12 | 119632 | 114183 | 27 | 16595 | 4 | 9369 |
| | 24 | 213630 | 210181 | - | - | - | - |
| **Nulo 2** | 3 | 48003 | 42988 | 23 | 12829 | 5 | 8388 |
| | 6 | 70643 | 65469 | 25 | 13016 | 6 | 8264 |
| | 12 | 116835 | 110161 | 28 | 14774 | 7 | 9282 |
| | 24 | 206808 | 199685 | - | - | - | - |
| **Alt 1** | 3 | 29871 | 16182 | 20 | 5151 | 3 | 1653 |
| | 6 | 53554 | 16121 | 23 | 5194 | 2 | 1657 |
| | 12 | 100490 | 16014 | 23 | 5244 | 3 | 1638 |
| | 24 | - | - | - | - | - | - |
| **Alt 2** | 3 | 29574 | 16123 | 21 | 5133 | 3 | 1718 |
| | 6 | 52995 | 15995 | 21 | 5287 | 3 | 1719 |
| | 12 | 99900 | 15858 | 23 | 5230 | 5 | 1731 |
| | 24 | - | - | - | - | - | - |

Tabla 4.9: Tiempos para el patrón cíclico `0`

En los algoritmos de fuerza bruta la mejora del algoritmo optimizado es del 2,75 % para *Nulo 1* y del 6,75 % para *Nulo 2*. En este segundo caso observamos que la mejora disminuye a medida que aumentamos el tamaño del patrón.

Para los algoritmos Shift-Add, la versión con matriz es mejor que la de tabla hash siempre sin tener en cuenta el tamaño del patrón. Para el vector *Nulo 1* la mejora es en torno al 91 % y para *Nulo 2* en torno al 56 %.

En cuanto a los algoritmos de Boyer Moore, tenemos que la versión de matriz siempre supera a la de tabla hash y su mejora va aumentando a medida que aumentamos el tamaño del patrón. Tenemos en media un valor de mejora del 154 % para ambos vectores. En cuando a la versión exacta y la versión con matriz, normalmente la versión exacta es mejor que la versión con matriz y en media la mejora es del 10 % para el vector *Nulo 1* y del 23 % para el vector Nulo 2.

| Árbol | Tamaño Patrón | BM (hash) | | BM (matriz) | | BM (exacto) | |
|---|---|---|---|---|---|---|---|
| | | Prep | Alg | Prep | Alg | Prep | Alg |
| **Nulo 1** | 3 | 20 | 24260 | 5 | 10893 | 12 | 14113 |
| | 6 | 20 | 33140 | 6 | 14170 | 17 | 12058 |
| | 12 | 30 | 55964 | 6 | 21034 | 23 | 15798 |
| | 24 | 30 | 97601 | 6 | 33520 | 30 | 29076 |
| **Nulo 2** | 3 | 24 | 24810 | 5 | 11300 | 15 | 9918 |
| | 6 | 24 | 35611 | 6 | 14881 | 17 | 12642 |
| | 12 | 29 | 55971 | 7 | 21064 | 23 | 15778 |
| | 24 | 31 | 99538 | 9 | 33162 | 31 | 25388 |
| **Alt 1** | 3 | 13 | 2180 | 3 | 823 | 8 | 780 |
| | 6 | 13 | 1496 | 3 | 520 | 10 | 439 |
| | 12 | 13 | 755 | 2 | 284 | 16 | 259 |
| | 24 | - | - | - | - | - | - |
| **Alt 2** | 3 | 15 | 2232 | 3 | 852 | 9 | 773 |
| | 6 | 13 | 1473 | 3 | 502 | 11 | 441 |
| | 12 | 14 | 704 | 4 | 267 | 18 | 273 |
| | 24 | - | - | - | - | - | - |

Tabla 4.10: Tiempos para el patrón cíclico `0`

Si comparamos los algoritmos Boyer Moore y Shift-Add obtenemos que las versiones del algoritmo Shift-Add son mejores que las del Boyer Moore y van aumentando a medida que aumentamos el tamaño del patrón en un factor cercano al $1,8$ en la mayoría de los casos. Esto significa que cuando doblamos el tamaño del patrón la mejora de los algoritmos Shift-Add frente a los de Boyer Moore es cercana a doblarla también. Cuantitativamente, la versión con tabla hash tiene valores más altos de mejora que la versión con matriz del algoritmo Shift-Add (84 % y 62 %, respectivamente para tamaño de patrón igual a 3).

Cuando comparamos el algoritmo de fuerza bruta optimizado con los algoritmos de Shift-Add observamos que los tiempos son muchos mejores para el algoritmo Shift-Add y que la mejora cuando doblamos el tamaño del patrón se multiplica por un factor del $1,6$ en la mayoría de los casos. Para la versión con tabla hash la mejora empieza siendo del 235 % y para la versión con matriz del 560 %.

Si comparamos el algoritmo de fuerza bruta optimizado con los algoritmos de Boyer Moore observamos que los tiempos son mucho mejores para este segundo caso. Para la versión con tabla hash la mejora ronda el 96 % siempre aunque se incrementa un poco cada vez que doblamos el tamaño del patrón. Para las otras dos versiones la mejora va aumentando cada vez que doblamos el patrón por un factor del 1,2 y empieza con una mejora en media del 291 % para un patrón de tamaño 3.

En cuanto a los vectores *Alternado 1* y *Alternado 2* no tenemos ninguna ocurrencia en ninguno de los dos vectores. Observamos que, los tiempos aumentan para los algoritmos

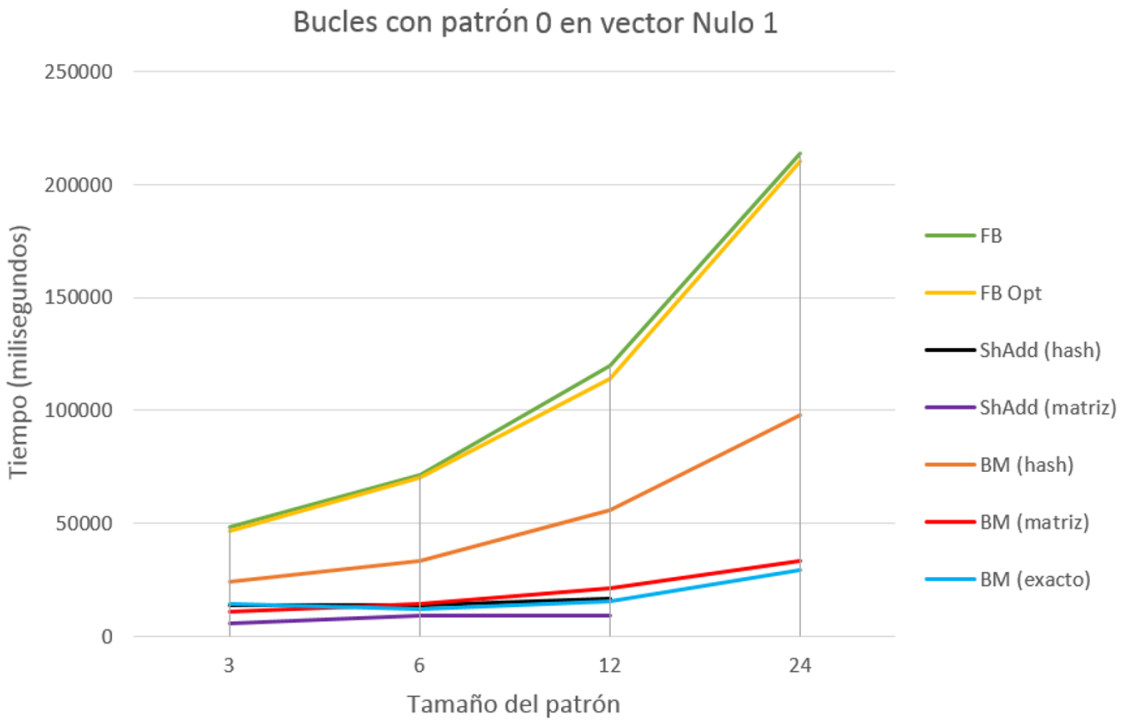


Figura 4.3: Gráfico con los tiempos de los algoritmos para el vector Nulo 1

de fuerza bruta y Shift-Add a medida que pedimos buscar el patrón original repetido varias veces pero disminuyen para el algoritmo Boyer Moore.

En los algoritmos de fuerza bruta la mejora de la versión optimizada aumenta con un factor mayor que el doble a medida que aumenta el tamaño del patrón y pasa del 84 % para tamaño 3 al 527 % para tamaño 12.

Los algoritmos Shift-Add tienen la particularidad de que la mejora de la versión con matriz se mantiene frente a la versión con tabla hash sin importar el tamaño del patrón y ronda en media el 208 %.

Para los algoritmos Boyer Moore observamos que siempre son mejores los algoritmos exacto y con matriz frente a la versión con tabla hash siendo la mejora del 172 % en media para la versión con matriz para los dos vectores y del 198 % para la versión exacta. Por transitividad, el algoritmo exacto es algo mejor que el aproximado en ambos casos pero la diferencia es muy pequeña y la mejora no supera el 10 % en media en ninguno de los dos vectores.

En este caso, cuando comparamos los algoritmos Shift-Add y Boyer Moore nos damos cuenta que las versiones que ganan en todos los casos son las del Boyer Moore. Al doblar el tamaño del patrón, la diferencia de mejora supera el factor 2 y llega a multiplicarse por un factor del $3,8$ en algún caso. Cuantitativamente, ambas versiones empiezan con una mejora por encima del 100 % para los dos vectores y llegan a alcanzar mejoras entre el

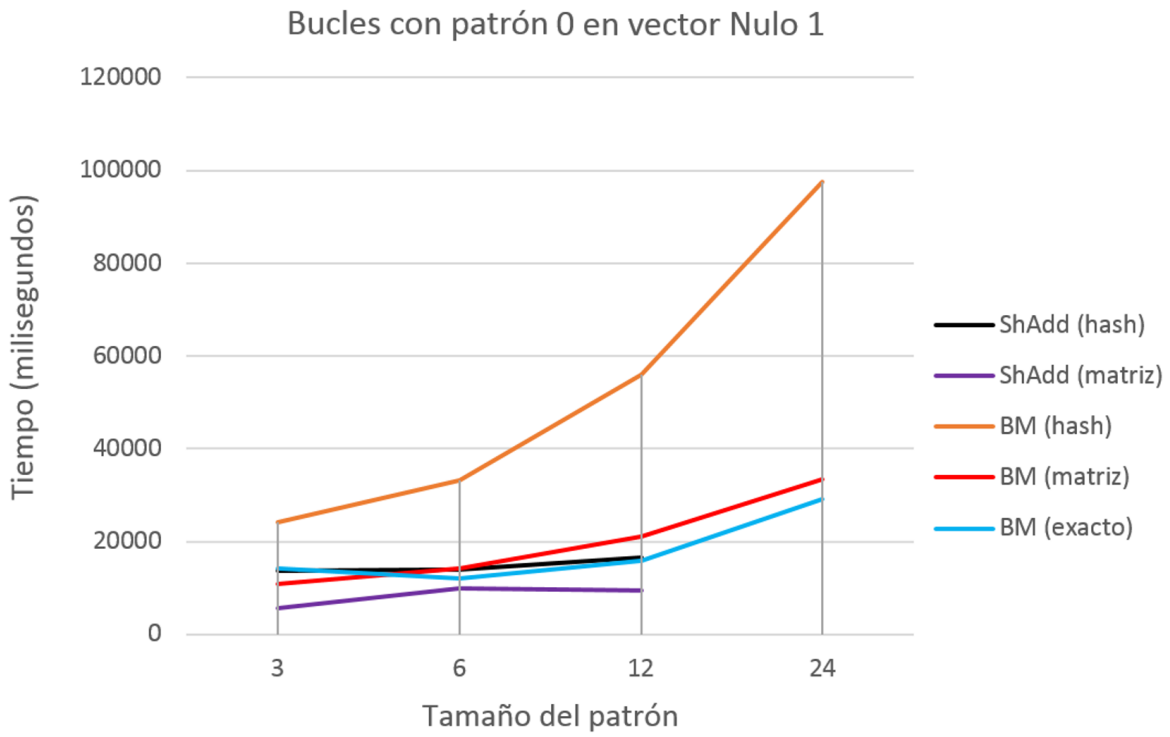


Figura 4.4: Gráfico con los tiempos de los algoritmos para el vector Nulo 1 sin FB

475 % y el 925 % cuando el tamaño del patrón es 12.

Si comparamos el algoritmo de fuerza bruta optimizado con las versiones del Shift-Add obtenemos que para la versión de tabla hash la mejora es del 208 % en favor del algoritmo Shift-Add. Este valor se multiplica por 4 cuando comparamos con la versión con matriz rondando un valor de mejora del 845 % sin importar el tamaño del patrón considerado.

Por último, si comparamos el algoritmo de fuerza bruta optimizado con las versiones del algoritmo Boyer Moore obtenemos que la mejora va incrementándose a medida que aumentamos el tamaño del patrón y ronda la multiplicación por factores entre el $1,5$ y el $2,2$. En este caso la diferencia en tiempo entre ambos algoritmos es enorme. Para la versión con tabla hash pasamos de una mejora en torno al 632 % a una en torno al 2086 %. Para las versiones con matriz y exacta esta diferencia es aún mayor y pasamos de una mejora en torno al 1909 % a una en torno al 5817 %.

Podemos sacar como conclusión final que el algoritmo Shift-Add suele tener siempre tiempos parecidos sin importar el tamaño del patrón a buscar y cuantas menos ocurrencias se encuentren menor es el tiempo que tarda en ejecutarse. Por otro lado, el algoritmo Boyer Moore es bastante malo cuando hay demasiadas ocurrencias en el patrón pero cuando no hay casi ninguna el tiempo que tarda en ejecutarse disminuye a medida que aumentamos el tamaño del patrón y alcanza unos valores muy bajos respecto al resto de algoritmos.

En cuanto a los diagramas, observamos en el diagrama 4.3 que para el vector *Nulo*

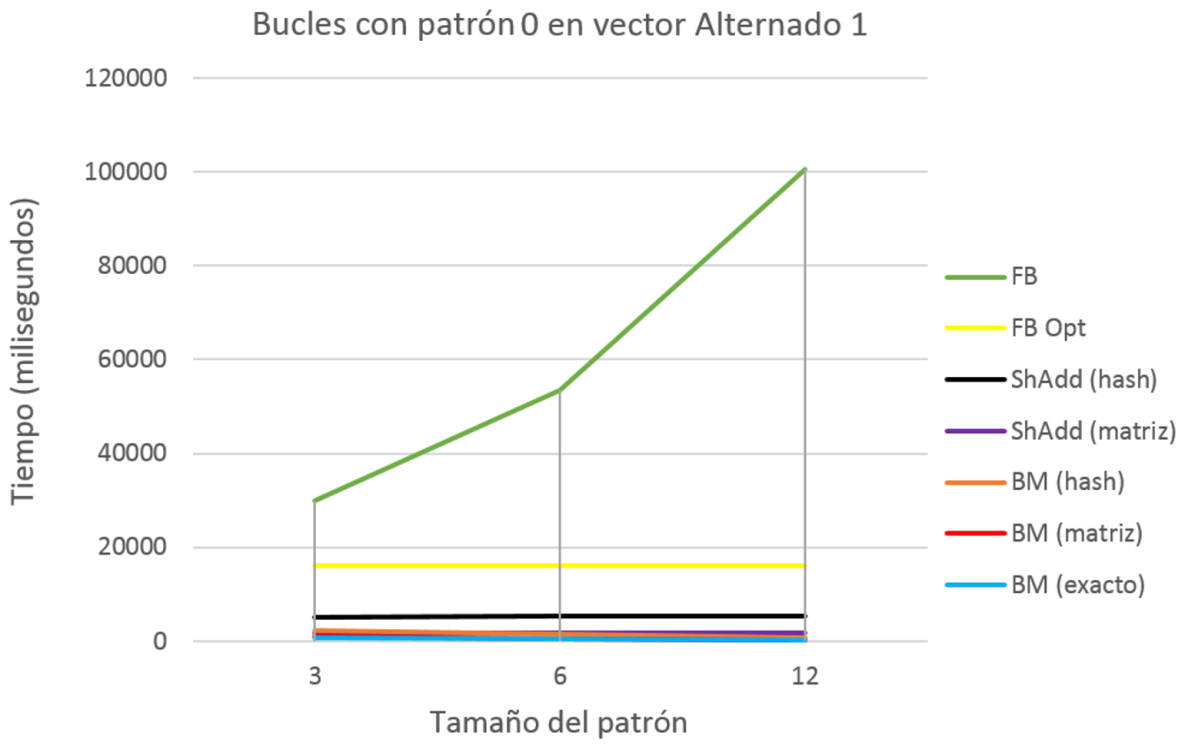


Figura 4.5: Gráfico con los tiempos de los algoritmos para el vector Alternado 1

*1* donde casi todos son ocurrencias los algoritmos de fuerza bruta aumentan cuadráticamente a medida que aumenta el tamaño del patrón.

En el diagrama 4.4 vemos que el algoritmo de Booyer Moore con tabla hash sigue un incremento constante que ronda un factor que va del $1,14$ al $1,03$ cada vez que aumentamos en 1 el tamaño del patrón. El factor de multiplicación baja en cada incremento del patrón. En cuanto al resto de algoritmos, observamos que el algoritmo Shift-Add con tabla hash es equiparable a las versiones exacta y con matriz del algoritmo Boyer Moore y que por debajo de todos estos algoritmos se encuentra la versión con matriz del algoritmo Shift-Add. Las versiones de Boyer Moore aumentan un poco cuando crece el tamaño del patrón pero las versiones del Shift-Add tienen un valor practicamente constante para cualquier tamaño del patrón.

Para el vector *Alternado 1* observamos primero en el diagrama 4.5 que la cota superior dada por el algoritmo de fuerza bruta es muy costosa en tiempo y hace parecer que el resto de algoritmos no varían a medida que aumenta el tamaño del patrón.

En el diagrama 4.6 donde hemos quitado los dos algoritmos de fuerza bruta observamos la verdadera tendencia de los algoritmos cuando no hay practicamente ocurrencias en el vector. El algoritmo más costoso es la versión de tabla hash del algoritmo Shift-Add. Luego tenemos un poco más abajo las versiones con tabla hash del algoritmo de Boyer Moore y la versión con matriz del algoritmo Shift-Add y por último las dos versiones que faltan

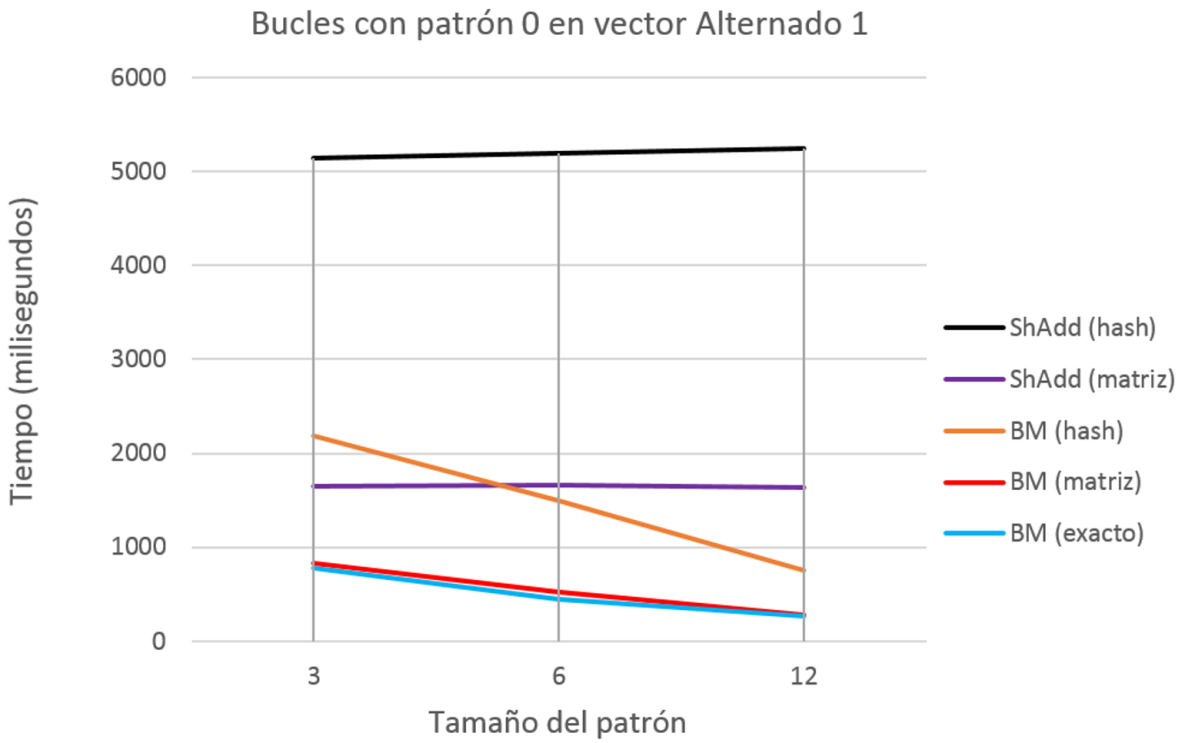


Figura 4.6: Gráfico con los tiempos de los algoritmos para el vector Alternado 1 sin FB

del algoritmo Boyer Moore. Observamos otra vez que los algoritmos Shift-Add siguen una linea recta a medida que aumenta el tamaño del patrón. En este caso, las versiones de Boyer Moore disminuyen su valor a medida que aumenta el tamaño del patrón dando como cierto el resultado teórico de que para un texto con un número razonable de ocurrencias a medida que aumentamos el tamaño del patrón este algoritmo va a funcionar mejor que el resto.

#### 4.3.1.2. Patrón cíclico `2 -2`

Los resultados del estudio del patrón cíclico `2 -2` se encuentran en las tablas 4.11 y 4.12 y en los diagramas 4.7, 4.8, 4.9 y 4.10. Con este patrón estudiamos el caso en el que hay muchas ocurrencias pero no el máximo posible ya que sólo pueden repetirse cada 2 posiciones. Debido a esto tenemos unas 39410 ocurrencias para el vector *Alternado 1* y unas 38860 para *Alternado 2*. Como en el estudio anterior, los vectores que no están destinados para tener la subcadena buscada no tienen ninguna ocurrencia. Volvemos a observar que, por lo general, los tiempos aumentan al aumentar el tamaño del patrón buscado.

Esta vez empezamos analizando los vectores *Alternado 1* y *Alternado 2* al ser los que nos aportan más datos para nuestro experimento.

El algoritmo de fuerza bruta es una cota superior para el resto de algoritmos. Su ver-

| Árbol | Tamaño Patrón | FB | FB Opt | ShAdd (hash) | | ShAdd (matriz) | |
|---|---|---|---|---|---|---|---|
| | | | | Prep | Alg | Prep | Alg |
| **Nulo 1** | 3 | 29623 | 15927 | 23 | 4964 | 3 | 1645 |
| | 6 | 52594 | 15966 | 23 | 4943 | 3 | 1643 |
| | 12 | 99234 | 16044 | 27 | 4990 | 2 | 1631 |
| | 24 | - | - | - | - | - | - |
| **Nulo 2** | 3 | 28768 | 15424 | 23 | 4958 | 3 | 1621 |
| | 6 | 52046 | 15066 | 24 | 4925 | 4 | 1644 |
| | 12 | 97679 | 15376 | 26 | 5080 | 4 | 1645 |
| | 24 | - | - | - | - | - | - |
| **Alt 1** | 3 | 38479 | 25842 | 25 | 9767 | 5 | 3617 |
| | 6 | 62026 | 36882 | 25 | 9783 | 5 | 3656 |
| | 12 | 107895 | 60381 | 28 | 9732 | 4 | 3645 |
| | 24 | 205162 | 109330 | - | - | - | - |
| **Alt 2** | 3 | 39340 | 26626 | 23 | 9737 | 5 | 3717 |
| | 6 | 62359 | 43273 | 27 | 9847 | 5 | 3714 |
| | 12 | 109618 | 65983 | 34 | 10485 | 7 | 3823 |
| | 24 | 202622 | 106293 | - | - | - | - |

Tabla 4.11: Tiempos para el patrón cíclico `2 -2`

sión optimizada mejora este algoritmo en un 50 % para el tamaño del patrón igual a 3 y a medida que se dobla este valor esta mejora va multiplicandose por un factor entre 1,1 y 1,5 llegando a ser superior con una mejora del 90 % para el tamaño del patrón igual a 24.

Entre las versiones Shift-Add observamos que sus valores permanecen relativamente constantes en el tiempo aunque aumentan un poco a medida que doblamos el tamaño del patrón. La mejora de la versión con matriz respecto a la de Shift-Add permanece constante con un valor en media del 172 %.

Para las versiones de los algoritmos de Boyer Moore tenemos que las versiones exacta y con matriz superan a la versión con tabla hash y que esta mejora se incrementa con un factor del $1,2$ cada vez que doblamos el tamaño del patrón. Se pasa de una mejora del 195 % a una del 254 % para la versión con matriz y de una mejora del 243 % a una del 308 % para la versión exacta. Observamos que en todos los casos el algoritmo exacto es mejor que el aproximado con matriz y, cuantitativamente, la mejora va del 13 % al 17 % dependiendo del caso. Con este resultado demostramos por tanto que el algoritmo exacto es mejor que el aproximado cuando los patrones son cíclicos.

Si comparamos versiones de distintos algoritmos tenemos que la versión con tabla hash del algoritmo Shift-Add es superior en tiempos a la de Boyer Moore. Esta mejora se cuatriplica cuando pasamos de un patrón de tamaño de 3 a uno de tamaño 6 y se dobla cuando pasamos de uno de tamaño 6 a uno de tamaño 12. Cuantitativamente pasamos de una mejora del 20 % en media a una del 210 %. En las versiones con matriz ocurre lo

| Árbol | Tamaño Patrón | BM (hash) | | BM (matriz) | | BM (exacto) | |
|---|---|---|---|---|---|---|---|
| | | Prep | Alg | Prep | Alg | Prep | Alg |
| **Nulo 1** | 3 | 15 | 2300 | 2 | 839 | 7 | 804 |
| | 6 | 14 | 1441 | 2 | 516 | 10 | 446 |
| | 12 | 14 | 824 | 2 | 248 | 15 | 259 |
| | 24 | - | - | - | - | - | - |
| **Nulo 2** | 3 | 16 | 3777 | 3 | 2179 | 10 | 2227 |
| | 6 | 15 | 1427 | 3 | 536 | 12 | 426 |
| | 12 | 14 | 822 | 4 | 359 | 17 | 275 |
| | 24 | - | - | - | - | - | - |
| **Alt 1** | 3 | 18 | 11750 | 4 | 3943 | 12 | 3363 |
| | 6 | 21 | 18075 | 4 | 5596 | 16 | 4780 |
| | 12 | 24 | 30128 | 6 | 8899 | 21 | 7759 |
| | 24 | 30 | 53850 | 5 | 15285 | 31 | 13242 |
| **Alt 2** | 3 | 23 | 12045 | 6 | 4113 | 16 | 3564 |
| | 6 | 19 | 18932 | 5 | 5614 | 16 | 4932 |
| | 12 | 22 | 32341 | 5 | 8589 | 23 | 7579 |
| | 24 | 29 | 53303 | 7 | 14967 | 29 | 12978 |

Tabla 4.12: Tiempos para el patrón cíclico `2 -2`

mismo y se multiplica la mejora primero por 5 y luego por $2,5$ pero en este caso la mejora es bastante menor que la que teníamos en la versión con tabla hash. Cuantitativamente pasamos de una mejora en media del 10 % para tamaño 3 a una del 130 % para tamaño 12.

En cuanto al algoritmo de fuerza bruta optimizado frente a las versiones del algoritmo Shift-Add observamos que para ambas versiones la diferencia se va aumentando en un factor entre $1,5$ y $1,8$ a medida que se dobla el tamaño del patrón. Esto es así debido a que el algoritmo de fuerza bruta tarda considerablemente más en cada caso mientras que los algoritmos Shift-Add tienen unos tiempos constantes para cualquier tamaño del patrón. Cuantitativamente la versión con tabla hash pasa de una mejora del 170 % a una del 525 % y la versión con matriz de una mejora del 615 % a una del 1590 %.

Por último, si comparamos el algoritmo de fuerza bruta optimizado frente a las versiones de Boyer Moore observamos que la diferencia de mejora se va manteniendo a medida que doblamos el tamaño del patrón. Para la versión con tabla hash tenemos una mejora del 110 % en media, para la versión con matriz tenemos una mejora del 600 % y para la versión exacta tenemos una mejora del 707 %.

En cuando a los vectores *Nulo 1* y *Nulo 2* volvemos a ratificar lo ya estudiado para el patrón `0` en el experimento anterior. Las comparaciones entre versiones de algoritmos quedan de la siguiente forma:

- El algoritmo de fuerza bruta optimizado es superior al de fuerza bruta en un 85 % en

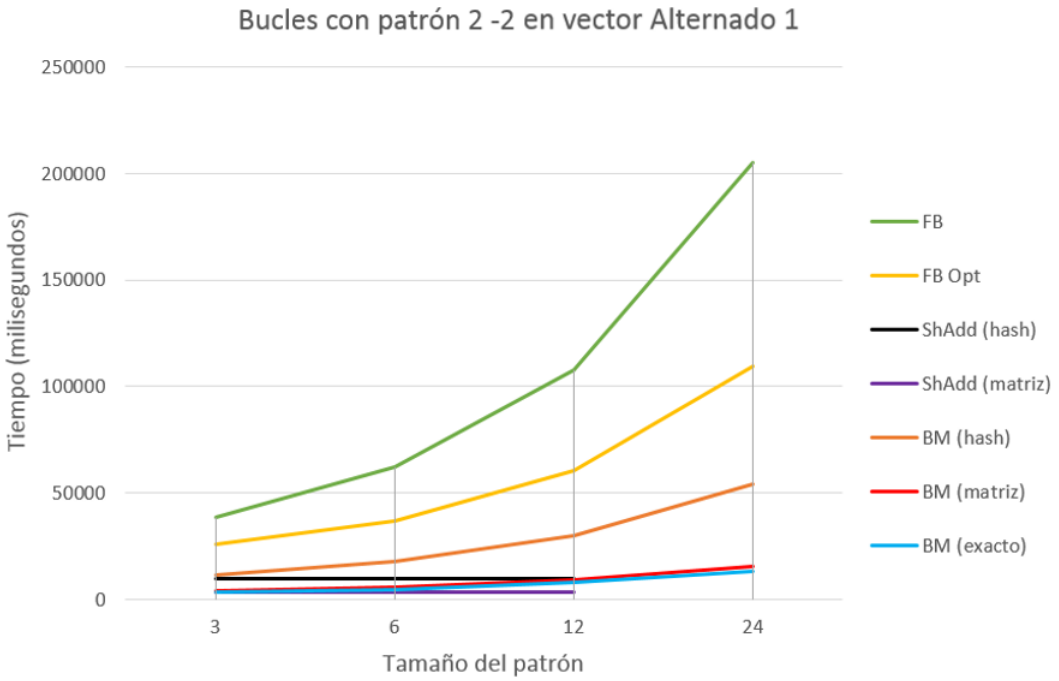


Figura 4.7: Gráfico con los tiempos de los algoritmos para el vector Alternado 1

media que aumenta en un factor entre 2, 2 y 2, 8 cada vez que doblamos el tamaño del patrón.

- La versión de matriz del algoritmo Shift-Add es superior a la versión con tabla hash en un 203 % en media y su mejora no depende del tamaño del patrón.
- Las versiones exacta y con matriz mejoran a la versión con tabla hash en todos los casos y suelen rondar un incremento superior al 200 %. Esta diferencia solo se incumple en el caso del vector *Nulo 2* para un tamaño del patrón igual a 3. En este caso la mejora es inferior y ronda el 70 %. Esta disminución posiblemente se deba a que cada 2000 datos hay 10 datos que se corresponden con diferencias de valor 2, así que en estas medidas los algoritmos de Boyer Moore no pueden desplazar los datos. Con tamaños del patrón superiores se elimina este problema.
- La versión exacta del algoritmo de Boyer Moore es ligeramente mejor a la versión con matriz. En muchos casos no hay practicamente diferencias de tiempo entre ambos algoritmos pero en los casos en los que difiere esta diferencia puede llegar a alcanzar un incremento del 30 % a favor del algoritmo de Boyer Moore exacto.

Las comparaciones entre distintos algoritmos quedan de la siguiente forma:

- Entre las versiones con tabla hash, el algoritmo de Boyer Moore siempre es superior al de Shift Add. Quitando el caso anómalo del algoritmo de Boyer Moore ya explicado (vector *Nulo 2* para tamaño de patrón igual a 3) la mejora se dobla cada vez que doblamos el tamaño del patrón. De esta forma empezamos con un valor de mejora igual al 115 % con tamaño 3 (del 31 % en el caso anómalo) y terminamos con un valor superior al 500 % de mejora.

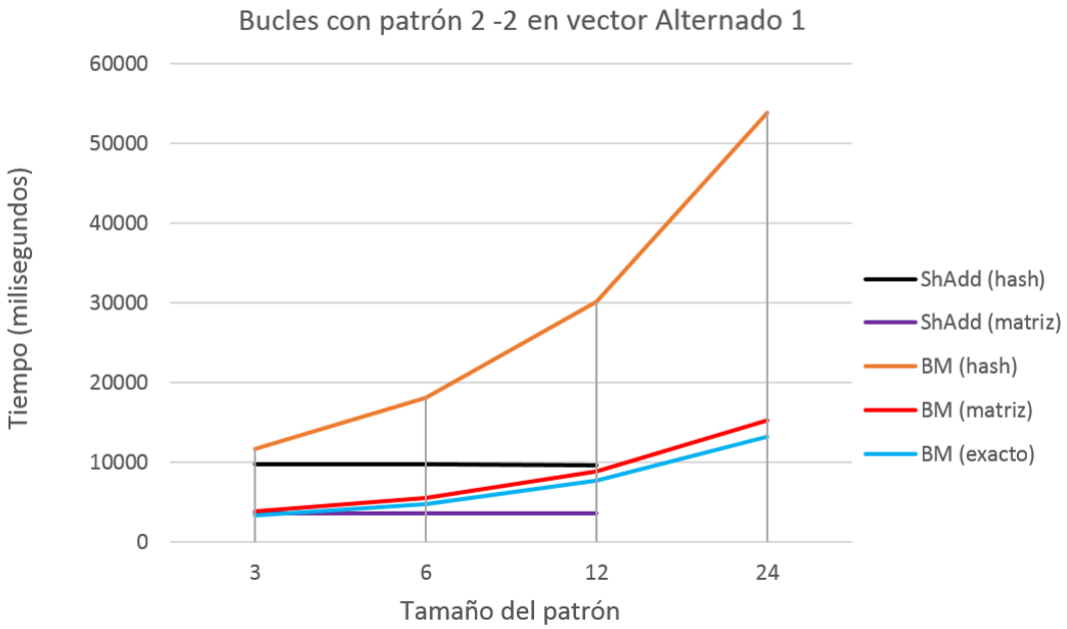


Figura 4.8: Gráfico con los tiempos de los algoritmos para el vector Alternado 1 sin FB

- En las versiones con matriz ocurre lo mismo y el algoritmo de Boyer Moore es superior al de Shift-Add en todos los casos menos en el anómalo donde la mejora del Shift-Add se corresponde con un valor del 25 %. La mejora vuelve a multiplicarse por un valor entre $1,7$ y $2,2$ cada vez que doblamos el tamaño del patrón. En el caso del vector *Nulo 1* pasa del 96 % al 517 % y en el *Nulo 2* del $-25$ % al 358 %.

- En cuanto al algoritmo de fuerza bruta optimizado frente a las versiones del algoritmo Shift-Add vemos que el algoritmo Shift-Add siempre es superior pero como ambos algoritmos mantienen sus valores en el tiempo las mejoras que se obtienen son parecidas sin importar el tamaño del patrón. En el caso de la tabla hash tenemos una mejora del 213 % en media y en el caso de la matriz del 851 %.

- Las diferencias del algoritmo de fuerza bruta optimizado frente a los algoritmos de Boyer Moore se incrementan a medida que doblamos el tamaño del patrón. Esto es debido a que el algoritmo de fuerza bruta permanece constante pero los algoritmos de Boyer Moore disminuyen a medida que el patrón se hace más largo. Cuantitativamente la mejora se multiplica por un factor de $1,8$ cada vez que doblamos el tamaño del patrón. Para la versión de tabla hash el algoritmo empieza con una mejora del 450 % y termina con una del 1800 %. Para las otras dos versiones en el vector *Nulo 1* se empieza con una mejora del 1840 % y se termina con una del 6380 %. En el vector *Nulo 2* se empieza con una mejora del 600 % y se termina con una del 4800 % en media.

Si estudiamos los diagramas vemos en el 4.7 que los algoritmos de fuerza bruta, fuerza bruta optimizado y la versión de Boyer Moore con tabla hash se comportan del mismo modo pero con una diferencia de tiempos considerable entre cada uno de ellos. Observamos que el crecimiento a medida que se dobla el tamaño del patrón es cuadrático en

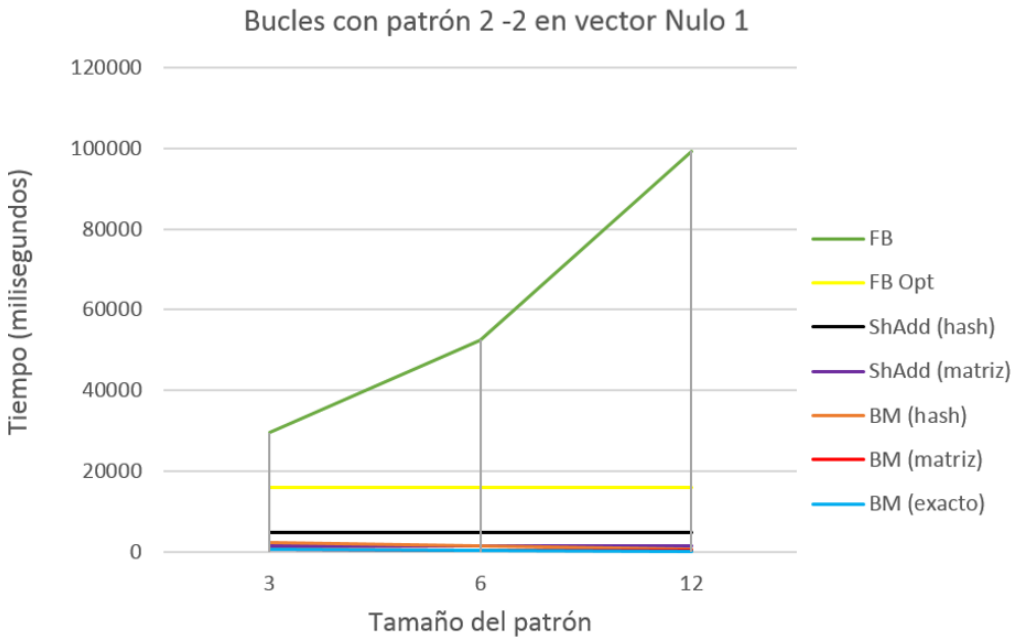


Figura 4.9: Gráfico con los tiempos de los algoritmos para el vector Nulo 1

los algoritmos de fuerza bruta. También vemos que el incremento que se produce en el algoritmo de Boyer Moore con tabla hash se multiplica por un factor entre el $1,13$ y el $1,04$ cada vez que aumentamos el tamaño del patrón en 1.

En el diagrama 4.8 observamos como las dos versiones de los algoritmos Shift-Add permanecen constantes cuando doblamos el tamaño del patrón. Además vemos que las versiones exacta y con matriz de Boyer Moore tienen sus tiempos entre las dos versiones del algoritmo Shift-Add y que a medida que se dobla el tamaño del patrón van aumentando pero de una forma más suave que la versión con tabla hash. También observamos como el algoritmo exacto es ligeramente mejor que el aproximado y que esta diferencia aumenta a medida que doblamos el tamaño del patrón.

En el diagrama 4.9 observamos que la diferencia entre la cota superior dada por el algoritmo de fuerza bruta es muy superior al tiempo que se obtiene en el resto de algoritmos. En el diagrama 4.10 observamos que los algoritmos de Shift-Add permanecen constantes a medida que se aumenta el tamaño del patrón mientras que los de Boyer Moore disminuyen a medida que se dobla el patrón. Observamos que para patrones largos y con pocas ocurrencias la versión con tabla hash del algoritmo de Boyer Moore es incluso mejor que la versión con matriz del algoritmo Shift-Add.

### 4.3.2. Patrones con alfabetos grandes

El estudio que hemos realizado en el apartado anterior usa árboles creados artificialmente con alfabetos pequeños. Queremos estudiar ahora si con un alfabeto más grande las diferencias obtenidas en la versión exacta del algoritmo de Boyer Moore y en la versión

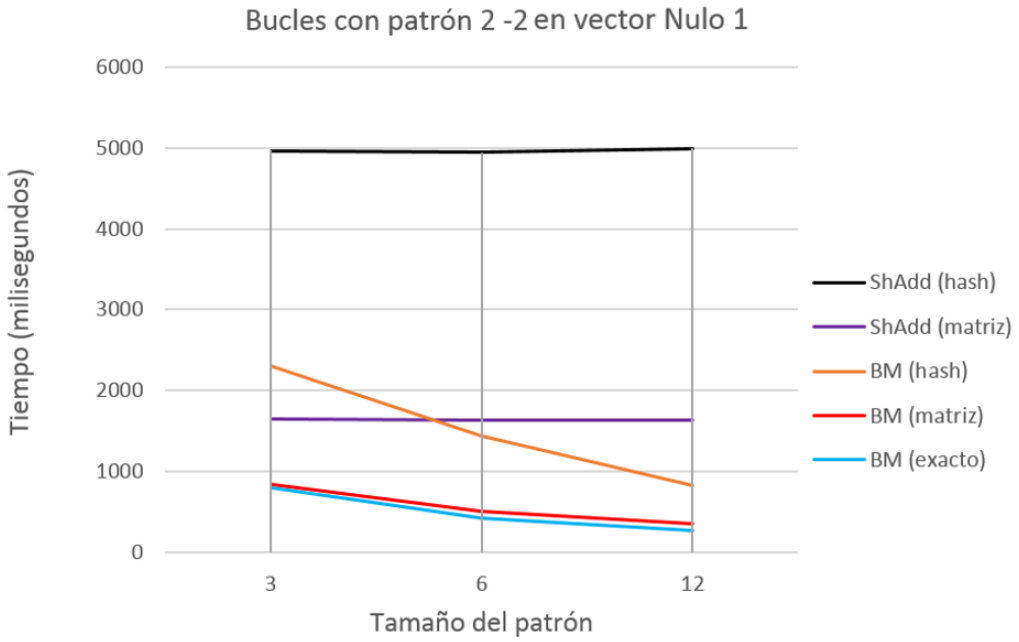


Figura 4.10: Gráfico con los tiempos de los algoritmos para el vector Nulo 1 sin FB

aproximada se mantienen. De esta forma podremos indicar si la diferencia entre la regla de carácter malo y la regla de sufijo bueno es lo suficientemente grande para tenerla en cuenta en la práctica.

Para poder estudiar esto añadimos al excel de experimentos otros 4 árboles ficticios con 78832 medidas y seguimos todos los pasos para hacer que funcionen en nuestra aplicación. El contenido de los 4 árboles es el siguiente:

- **Disperso 1**: árbol con los valores del vector *Alternado 2* multiplicados por 5. De esta forma la colección de diferencias del árbol está compuesta por los valores 10 y $-10$ y podemos estudiar alfabetos con al menos 20 números. Estudiamos el patrón cíclico `10 -10`.
- **Disperso 2**: árbol con los valores del vector *Alternado 2* multiplicados por 10. De esta forma la colección de diferencias del árbol está compuesta por los valores 20 y $-20$ y podemos estudiar alfabetos con al menos 40 números. Estudiamos el patrón cíclico `20 -20`.
- **Disperso 3**: árbol con los valores del vector *Alternado 2* multiplicados por 50. De esta forma la colección de diferencias del árbol está compuesta por los valores 100 y $-100$ y podemos estudiar alfabetos con al menos 200 números. Estudiamos el patrón cíclico `100 -100`.
- **Disperso 4**: árbol cuyos datos aumentan su diferencia en 1 respecto de un dato al siguiente. Este árbol está constituido por bloques de valores que empiezan con una diferencia de valor 1 y terminan con una de valor 98 aumentando la diferencia en cada dato. Con este árbol conseguimos un vector de diferencias con alfabeto de

tamaño 100 y con muchos datos diferentes en el árbol para hacer actuar las dos reglas del algoritmo de Boyer Moore. El patrón que buscaremos será la sucesión de números que empieza en el número 50 y aumenta de 1 en 1, es decir, el patrón `50 51 52...`.

Volvemos a estudiar los tiempos de ejecución con diferentes tamaños de los patrones que hemos considerado. Estudiamos el caso en el que tenemos el máximo número de ocurrencias y en el que no hay ninguna para cada árbol.

Se ha creado una tabla por patrón estudiado en donde aparecen los tiempos del algoritmo de fuerza bruta sin optimizar (la cota superior de los tiempos), el algoritmo aproximado de Boyer Moore con matriz y el algoritmo exacto de Boyer Moore. El árbol *Disperso 4* no aparece en los estudios de los patrones `10 -10` y `20 -20` debido a que el árbol *Disperso 3* cumple su misma función y, por tanto, no aporta nada nuevo al estudio.

#### 4.3.2.1. Patrón cíclico `10 -10`

Los resultados del patrón cíclico `10 -10` para distintos alfabetos se encuentran en la tabla 4.13. En este caso se produce una ocurrencia cada dos datos en el vector *Disperso 1* (alfabeto de tamaño 20) y no se producen ocurrencias en los vectores *Disperso 2* y *Disperso 3* (alfabetos de tamaño 40 y 200, respectivamente).

| **Árbol** | **Tamaño Patrón** | **Ocurrencias** | **FB** | **BM (matriz)** | | **BM (exacto)** | |
|---|---|---|---|---|---|---|---|
| | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| **Disperso 1** | 3 | 38865 | 43071 | 11 | 4149 | 23 | 3773 |
| | 6 | 38705 | 69545 | 13 | 5855 | 26 | 5096 |
| | 12 | 38462 | 118260 | 13 | 9174 | 41 | 8585 |
| **Disperso 2** | 3 | 0 | 33390 | 9 | 1010 | 16 | 905 |
| | 6 | 0 | 55698 | 15 | 600 | 20 | 580 |
| | 12 | 0 | 107978 | 11 | 353 | 24 | 281 |
| **Disperso 3** | 3 | 0 | 31816 | 15 | 931 | 24 | 905 |
| | 6 | 0 | 58601 | 15 | 499 | 28 | 637 |
| | 12 | 0 | 106701 | 18 | 303 | 33 | 271 |

Tabla 4.13: Tiempos para patrones anidados de la forma `10 -10`

La cota superior dada por el algoritmo de fuerza bruta va aumentando a medida que aumentamos el tamaño del patrón y tarda más cuántas más ocurrencias se producen. La mejora producida entre este algoritmo y los de Boyer Moore aumenta a medida que se dobla el tamaño del patrón. Con muchas ocurrencias (vector *Disperso 1*) el factor entre ambos tiempos es de 12. Sin ninguna ocurrencia el factor de multiplicación pasa de 34 para tamaño de patrón igual a 3 a 358 para tamaño del patrón igual a 12.

Si comparamos las versiones de los algoritmos de Boyer Moore obtenemos que para el alfabeto de tamaño 20 y muchas ocurrencias se comporta mejor la versión exacta con un

7 % de mejora en el peor caso (tamaño 12 del patrón) y con un 15 % en el mejor (tamaño 6 del patrón).

Para el alfabeto de tamaño 40 y sin ocurrencias obtenemos una mejora que fluctúa del $3,5\,\%$ al $25,6\,\%$ entre los distintos casos. Para el alfabeto de tamaño 200 sin ocurrencias llegamos a tener una mejora del 12 % para tamaño del patrón igual a 12 pero venimos de un empeoramiento del $21,6\,\%$ para tamaño del patrón igual a 6.

Necesitamos más datos para poder llegar a una conclusión entre ambas versiones del algoritmo.

#### 4.3.2.2. Patrón cíclico `20 -20`

Los resultados para el patrón cíclico `20 -20` para distintos alfabetos se encuentran en la tabla 4.14. Tenemos una ocurrencia cada dos datos en el vector *Disperso 2* (alfabeto de tamaño 40) y no se producen ocurrencias en los vectores *Disperso 1* y *Disperso 3* (alfabetos de tamaño 40 y 200, respectivamente). Ya no podemos tener vectores con alfabeto menor que 40 debido a que el patrón contiene los valores 20 y $-20$, que nos da un alfabeto de 40 números por defecto.

| **Árbol** | **Tamaño Patrón** | **Ocurrencias** | **FB** | **BM (matriz)** | | **BM (exacto)** | |
|---|---|---|---|---|---|---|---|
| | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| **Disperso 1** | 3 | 0 | 31480 | 6 | 870 | 14 | 850 |
| | 6 | 0 | 59440 | 7 | 573 | 21 | 542 |
| | 12 | 0 | 108899 | 7 | 350 | 20 | 262 |
| **Disperso 2** | 3 | 38865 | 42025 | 14 | 4115 | 23 | 3675 |
| | 6 | 38705 | 64744 | 15 | 5586 | 26 | 4844 |
| | 12 | 38462 | 118366 | 21 | 9356 | 47 | 8278 |
| **Disperso 3** | 3 | 0 | 32007 | 23 | 933 | 24 | 831 |
| | 6 | 0 | 53617 | 14 | 483 | 23 | 507 |
| | 12 | 0 | 106681 | 18 | 324 | 35 | 302 |

Tabla 4.14: Tiempos para patrones anidados de la forma `20 -20`

Volvemos a observar que la cota superior dada por el algoritmo de fuerza bruta aumenta a medida que doblamos el tamaño del patrón y que es mayor cuantas más ocurrencias se encuentren en el árbol. Las diferencias de este algoritmo con los de Boyer Moore son parecidas al caso anterior. Cuando se producen muchas ocurrencias, vector *Disperso 2*, los tiempos del algoritmo Boyer Moore tienen que multiplicarse por un factor de 12 para llegar a los de fuerza bruta. Sin ocurrencias, el factor es de 36 para tamaño de patrón igual a 3 y de 353 para tamaño de patrón igual a 12.

Comparando las dos versiones del algoritmo de Boyer Moore obtenemos que para el

vector *Disperso 2* con alfabeto de tamaño 40 la mejora del algoritmo exacto respecto al aproximado es en media del 13,42 % y no fluctúa mucho a medida que doblamos el tamaño del patrón. Para los casos sin ocurrencias tenemos para el vector *Disperso 1* una mejora que pasa del 2,35 % para tamaño del patrón igual a 3 a una del 33,58 % para tamaño del patrón igual a 12 a favor del algoritmo exacto. En el caso del vector *Disperso 3* con alfabeto de tamaño 200 tenemos una mejora del 12,27 % para tamaño del patrón igual a 3, un deterioro del 4,7 % para tamaño del patrón igual a 6 y una mejora del 7,3 % para tamaño del patrón igual a 12.

Empezamos a ver que cuánto más largo es el patrón mejor funciona el algoritmo exacto pero con alfabetos grandes las diferencias entre ambas versiones se asemejan más que con alfabetos pequeños.

#### 4.3.2.3. Patrón cíclico `100 -100`

Los resultados para el patrón cíclico `100 -100` para alfabeto de tamaño 200 se encuentran en la tabla 4.15. Tenemos una ocurrencia cada dos datos en el vector *Disperso 3* y no se producen ocurrencias en el resto de los vector.

| **Árbol** | **Tamaño Patrón** | **Ocurrencias** | **FB** | **BM (matriz)** | | **BM (exacto)** | |
|---|---|---|---|---|---|---|---|
| | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| **Disperso 1** | 3 | 0 | 31555 | 15 | 839 | 24 | 896 |
| | 6 | 0 | 58354 | 19 | 559 | 27 | 574 |
| | 12 | 0 | 107488 | 21 | 367 | 36 | 310 |
| **Disperso 2** | 3 | 0 | 31272 | 17 | 937 | 26 | 917 |
| | 6 | 0 | 56509 | 17 | 557 | 30 | 540 |
| | 12 | 0 | 106426 | 19 | 368 | 29 | 246 |
| **Disperso 3** | 3 | 38865 | 42122 | 18 | 4117 | 29 | 3593 |
| | 6 | 38705 | 69634 | 21 | 5104 | 36 | 5068 |
| | 12 | 38463 | 119984 | 29 | 10926 | 48 | 8157 |
| **Disperso 4** | 3 | 0 | 30858 | 15 | 848 | 19 | 768 |
| | 6 | 0 | 54805 | 25 | 494 | 26 | 547 |
| | 12 | 0 | 102334 | 47 | 353 | 30 | 238 |

Tabla 4.15: Tiempos para patrones anidados de la forma `100 -100`

La cota superior dada por el algoritmo de fuerza bruta vuelve a aumentar a medida que doblamos el tamaño del patrón y aumenta para el vector donde se producen las ocurrencias. En cuanto a factores de multiplicación de los tiempos respecto del algoritmo de Boyer Moore pasamos de un factor de 35 para tamaño del patrón igual a uno de 338 para tamaño del patrón igual a 12 en el caso de no tener ninguna ocurrencia. Para el vector *Disperso 3* con muchas ocurrencias se tiene un factor de multiplicación de 12 para pasar del tiempo de ejecución de los algoritmos de Boyer Moore al tiempo de ejecución

del algoritmo de fuerza bruta.

Si comparamos las versiones del algoritmo de Boyer Moore en el caso de no producirse ocurrencias vemos que los valores fluctúan y que algunas veces gana una versión y algunas otra. Observamos que cuando el tamaño del patrón es muy grande siempre gana la versión exacta y su porcentaje de mejora respecto a la versión aproximada llega a alcanzar el 49 %. Para tamaños del patrón menores los dos algoritmos fluctúan y sus tiempos son practicamente iguales en media.

En el caso del vector *Disperso 3* observamos que el algoritmo exacto es siempre mejor que el aproximado y que esa mejora para el tamaño del patrón igual a 12 llega al 34 %.

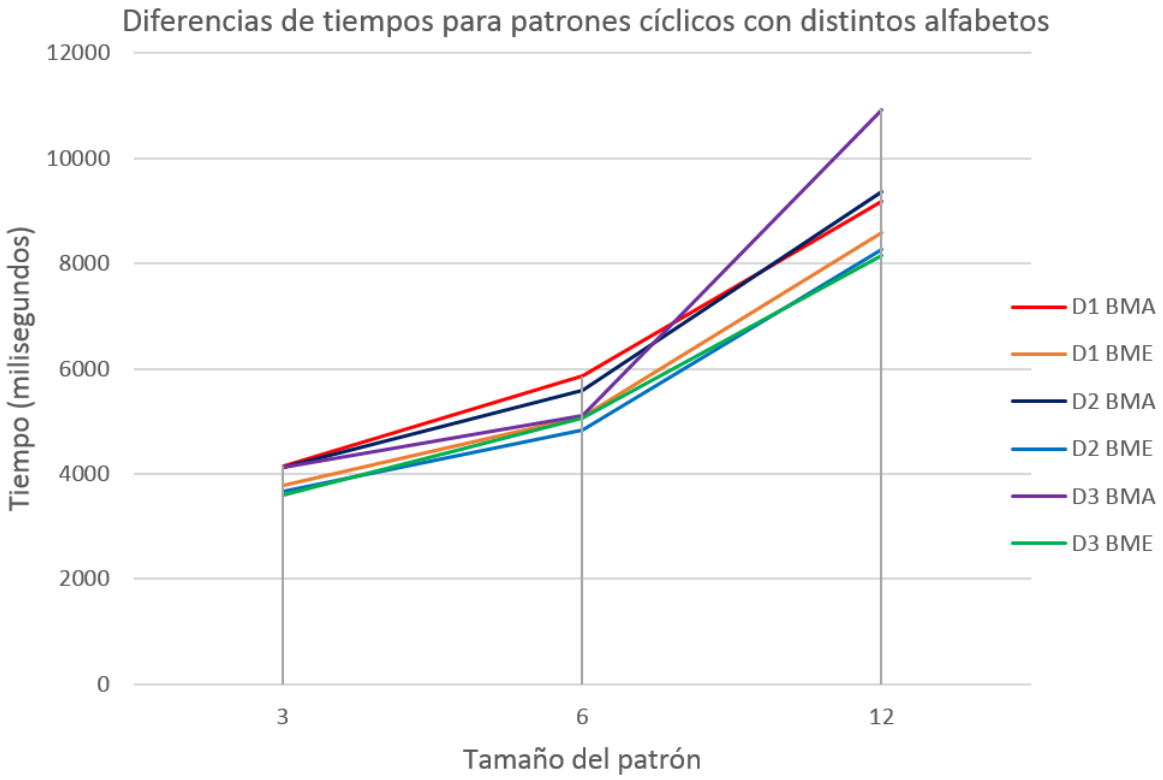


Figura 4.11: Gráfico con los tiempos de los algoritmos para patrones con muchas ocurrencias en distintos alfabetos

Las conclusiones de este experimento es que las diferencias entre las versiones exacta y aproximada del algoritmo de Boyer Moore para patrones cíclicos se aproximan a medida que aumenta el tamaño del alfabeto pero vuelven a diferenciarse si tenemos un patrón muy largo. En el diagrama 4.11 podemos ver los tiempos que se obtienen para las versiones exacta (BME) y aproximada (BMA) en los casos donde cada vector alcanza el mayor número de ocurrencias. Así podemos ver la diferencia que nos aporta el tamaño del alfabeto ya que el vector *Disperso 1* (D1) tiene un alfabeto de tamaño 20, el vector *Disperso 2* (D2) un alfabeto de tamaño 40 y el vector *Disperso 3* (D3) un alfabeto de tamaño 200.

Observamos que los tiempos para el algoritmo exacto son parecidos para cualquier alfabeto y son menores que para el algoritmo aproximado. Además, observamos que para un alfabeto mayor el algoritmo aproximado empieza a empeorar para patrones de tamaño superior a 6. Por último, también vemos que para tamaños del patrón largos los algoritmos exactos se posicionan todos en tiempos parecidos sin importar el tamaño del alfabeto mientras que para el caso aproximado el que menor tiempo tiene es el de menor tamaño del alfabeto y la diferencia es considerable a medida que aumentamos el tamaño de éste.

#### 4.3.2.4. Patrón `50 51 52...`

Los resultados para el patrón `50 51 52...` se encuentran en la tabla 4.16 y en el diagrama 4.12. En este caso estudiamos un patrón que no es cíclico para vectores con alfabeto mayor de 100 elementos. En todos los casos tenemos 788 ocurrencias para el vector *Disperso 4* y ninguna para el vector *Disperso 3*.

| **Árbol** | **Tamaño Patrón** | **Ocurrencias** | **FB** | **BM (matriz)** | | **BM (exacto)** | |
|---|---|---|---|---|---|---|---|
| | | | | **Prep** | **Alg** | **Prep** | **Alg** |
| **Disperso 3** | 3 | 0 | 30671 | 13 | 875 | 18 | 794 |
| | 6 | 0 | 56941 | 16 | 517 | 32 | 507 |
| | 12 | 0 | 103843 | 18 | 340 | 67 | 269 |
| **Disperso 4** | 3 | 788 | 30197 | 9 | 976 | 16 | 925 |
| | 6 | 788 | 56204 | 11 | 671 | 17 | 702 |
| | 12 | 788 | 106005 | 10 | 576 | 21 | 505 |

Tabla 4.16: Tiempos para el patrón `50 51 52 53...`

Vemos de nuevo que la cota superior dada por el algoritmo de fuerza bruta aumenta a medida que doblamos el tamaño del patrón. Para el caso sin ocurrencias pasamos una mejora del 3584 % a una del 34025 % a favor del algoritmo de Boyer Moore y para el caso con ocurrencias de una del 3079 % a una del 19597 %. Observamos que el algoritmo Boyer Moore funciona mejor cuántas menos ocurrencias aparezcan en el vector y que el tiempo va disminuyendo a medida que doblamos el tamaño del patrón.

En cuanto a las versiones de algoritmo Boyer Moore tenemos que para el caso sin ocurrencias siempre gana el algoritmo exacto y su mejora fluctúa entre el 2 % y el 25 %. Para el caso con ocurrencias a veces el algoritmo aproximado es mejor que el de Boyer Moore para tamaños del patrón pequeños pero cuando doblamos y alcanzamos un valor del tamaño del patrón de 12 vemos que gana el algoritmo exacto con un porcentaje del 14 % de mejora.

Vemos entonces que con alfabetos grandes las dos versiones del algoritmo Boyer Moore son muy parecidas en tiempos pero que el algoritmo exacto funciona mejor que el algoritmo aproximado para patrones largos. En el diagrama 4.12 observamos que los tiempos del vector *Disperso 3* sin ocurrencias son menores que los del vector *Disperso 4*. Además

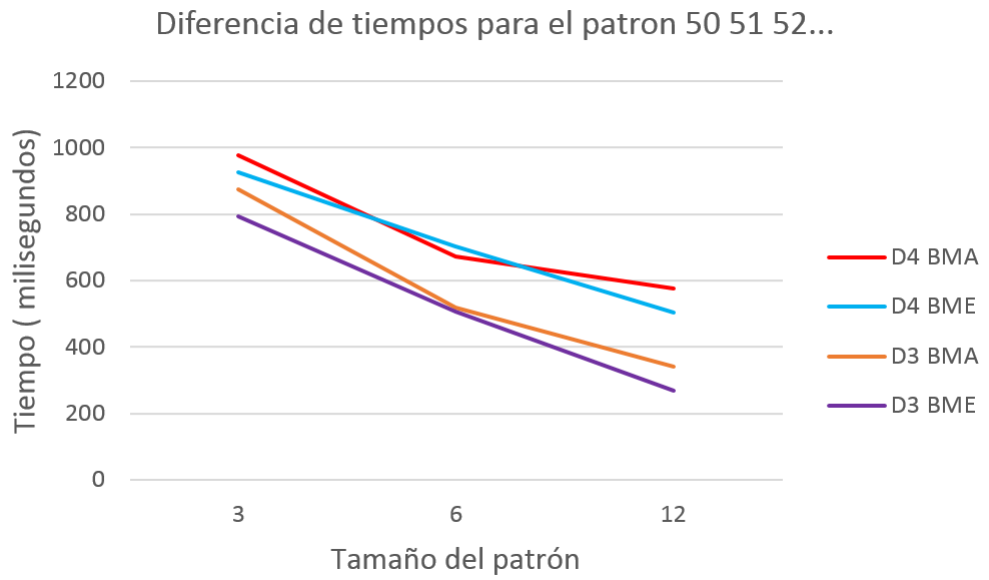


Figura 4.12: Gráfico con los tiempos de los algoritmos para el patrón `50 51 52...`

los tiempos del algoritmo exacto empiezan y terminan por debajo que los del algoritmo aproximado.

# Capítulo 5

# Conclusiones

Se ha desarrollado una aplicación que puede tratar las medidas del tronco de los árboles dadas por un archivo de excel y que devuelve distintos resultados interesantes sobre estas medidas a petición del usuario.

Como trabajo a realizar quedaría pendiente seguir añadiendo nuevas funciones que fueran útiles para el estudio de las medidas de los árboles. Además se podría realizar un estudio de cuáles de estas funciones son las más importantes en el mundo real para seguir trabajando sobre ellas añádiendo nuevas mejoras.

En cuanto al estudio de búsqueda de subcadenas, se ha realizado un estudio para patrones sencillos tanto en las medidas de los árboles como en otros vectores. Se ha podido constatar que el algoritmo Shift-Add tarda un tiempo razonable para cualquier problema estudiado y que el algoritmo Boyer Moore es razonablemente bueno en el caso general que contemplamos y cuando hay pocas ocurrencias en las medidas del patrón buscado.

En cuanto a resultados específicos hemos podido constatar que la regla del sufijo bueno del algoritmo Boyer Moore en búsquedas exactas siempre mejora algo al mero uso de la regla de desplazamiento de carácter malo. Y que esta mejora se acentúa si tratamos con patrones cíclicos que se dan bastantes veces en los vectores estudiados. Además, hemos estudiado que esta mejora se mantiene en alfabetos grandes cuando el patrón es largo y que, en la práctica, usar las dos reglas al final aporta más beneficios que solo usar la regla de desplazamiento de carácter malo. Por lo tanto, un posible nuevo estudio sería intentar adaptar esta regla al caso aproximado y ver cómo son sus nuevos tiempos.

Otros posibles trabajos que se podían realizar en cuanto al estudio de búsqueda de subcadenas sería añadir nuevos algoritmos a nuestro problema y ver cómo se comportan estos respecto a los que ya hemos estudiado. Además podríamos buscar otros casos en los patrones y estudiar los nuevos resultados ya que en este trabajo siempre hemos trabajado con patrones sencillos con valores numéricos muy bajos. Por ejemplo se podría intentar encontrar patrones de saltos muy grandes en las medidas y ver como se comportan los algoritmos implementados en esta nueva búsqueda.

# Bibliografía


[1] Charras, C. y Lecroq, T. *Handbook of Exact String Matching Algorithms*. College Publications, 2004. ISBN 978-0-9543006-4-7.

[2] Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-58519-8.

[3] Hirvola, T. *Bit parallel approximate string matching under Hamming distance*. Proyecto Fin de Carrera, Aalto University; School of Science, 2016.

[4] Al Khamaiseh, K. y Al Shagarin, S. A Survey of String Matching Algorithms. *Int. Journal of Engineering Research and Applications*, vol. 4(7 (Version2)), páginas 144–156, 2014. ISSN 2248-9622.

[5] Langmead, B. Approximate matching. 2014. Johns Hopkins whiting school of engineering. Dept. of Computer Science.

[6] Mailun, T. y Norgaard Storm Pedersen, C. Approximate matching, Lecture notes in String algorithms (Q4/2013). 2013. Dept. of Computer Science, Faculty of Science. Aarhus University.

[7] Salmela, L., Tarhio, J. y Kalsi, P. Approximate Boyer Moore String Matching for Small Alphabets. *Algorithmica*, vol. 58(3), páginas 591–609, 2010.

[8] Tarhio, J. y Ukkonen, E. Approximate Boyer Moore String Matching. *SIAM J. Comput.*, vol. 22(2), páginas 243–260, 1993. ISSN 0097-5397. Society for Industrial and Applied Mathematics; Philadelphia, PA, USA.