Trabajo de Fin de Máster
Máster Universitario en Ingeniería Informática

Búsqueda eficiente de hashes de similitud aproximada

Daniel Huici Meseguer

Director: Ricardo J. Rodríguez

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Junio de 2023
Curso 2022/2023

Agradecimientos

A Ricardo, por guiarme y brindarme esta oportunidad.
A mi familia, por confiar en mí.
A mis amigos, por acompañarme en este camino.

RESUMEN

En la actualidad, el manejo y análisis de grandes volúmenes de datos se ha convertido en un desafío clave en el campo del análisis de código dañino (*malware*). Los investigadores y profesionales de la seguridad informática necesitan herramientas eficientes y efectivas para identificar y analizar rápidamente las muestras de malware con el fin de tomar medidas adecuadas para proteger los sistemas. En este contexto, las técnicas de búsqueda aproximada, que permiten buscar una coincidencia en un valor próximo, combinadas con los algoritmos de similitud aproximada, que capturan características comunes entre archivos similares mediante la detección de pequeñas diferencias y variaciones en sus contenidos, se vuelven especialmente relevantes. Estas técnicas permiten realizar análisis rápidos y escalables en grandes conjuntos de datos, facilitando la detección y clasificación eficiente de nuevas variantes y familias de amenazas.

Por todo ello, en este trabajo se ha llevado a cabo el desarrollo en *Python* de un sistema de búsqueda de similitud aproximada basada en la estructura de datos *HNSW (Hierarchical Navigable Small World)*, capaz de realizar búsquedas sobre hashes de similitud aproximada. En el desarrollo de este sistema se ha hecho especial hincapié en la creación de una arquitectura robusta, que permita almacenar diversos tipos de datos, además de usar distintos algoritmos de similitud aproximada.

Con el fin de verificar el correcto funcionamiento y la eficiencia del sistema, se han definido una serie de métricas que muestran su comportamiento, combinando diversas configuraciones y con distintos tamaños de datos para comprobar su evolución y entender mejor su funcionamiento e impacto en el rendimiento. Este estudio permite escoger aquella configuración concreta que mantiene una buena precisión y reduce tiempos de búsqueda de forma considerable. Los resultados de la experimentación realizada muestran que el sistema obtenido cumple con el objetivo y las expectativas, obteniendo unos tiempos de ejecución en las funciones de búsqueda e inserción muy reducidos.

ABSTRACT

Currently, the handling and analysis of large volumes of data has become a key challenge in the field of malicious software (*malware*) analysis. Computer security researchers and professionals need efficient and effective tools to quickly identify and analyze malware samples in order to take appropriate measures to protect systems. In this context, approximate search techniques, which allow searching for a match in a close value, combined with approximate similarity matching algorithms, which capture common features between similar files by detecting small differences and variations in their contents, become especially relevant. These techniques enable fast and scalable analysis on large datasets, facilitating the efficient detection and classification of new variants and families of malware.

For all these reasons, in this work we have carried out the development in Python of an approximate similarity search system based on the *HNSW (Hierarchical Navigable Small World)* data structure, capable of performing searches on hashes of approximate similarity. In the development of this system, special emphasis has been placed on the creation of a robust architecture, which allows the storage of various types of data, in addition to using different approximate similarity algorithms.

In order to verify the correct operation and efficiency of the system, a series of metrics have been defined that show its behavior, combining various configurations and with different data sizes to check its evolution and better understand its operation and impact on performance. This study enables the selection of a specific configuration that balances precision and significantly reduces search times. The results of the experimentation carried out show that the system obtained meets the goal and expectations, obtaining very reduced execution times in the search and insertion functions.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Este trabajo surge de la necesidad de mejorar la eficiencia y precisión en el análisis de código dañino (*malware*) [4]. El mundo de la seguridad informática se enfrenta constantemente a nuevas amenazas y variantes de malware, lo que requiere soluciones más rápidas y efectivas para su detección y análisis [5].

La detección y reconocimiento de estas muestras son tradicionalmente realizados mediante el uso de firmas o hashes criptográficos (como *MD5* o *SHA-1*) [10], los cuales representan una entrada dada como una serie única de valores de bytes (*hash*). Sin embargo, estos enfoques presentan un problema conocido como el efecto avalancha [6]. Este efecto implica que cambios menores en el archivo, como la modificación de un solo bit, resultan en un hash completamente diferente. Como resultado, es necesario contar con métodos más flexibles y robustos que permitan identificar muestras similares de malware que pueden no ser idénticas pero sí similares.

Los algoritmos de similitud aproximada permiten detectar estas similitudes al calcular (normalmente) un hash que representa características clave de los datos y que puede ser utilizado para comparar y relacionar diferentes elementos [7,8]. Estos algoritmos permiten una detección más flexible y tolerante a pequeñas variaciones y, además, destacan por su eficiencia y su capacidad para manejar grandes volúmenes de datos de manera rápida y precisa.

En este contexto, también entran en juego las búsquedas aproximadas. En lugar de buscar una coincidencia exacta, las búsquedas aproximadas se centran en encontrar elementos que sean similares o se aproximen a una determinada consulta. La búsqueda aproximada se convierte así en una herramienta esencial para los investigadores y analistas de seguridad, proporcionando un enfoque más efectivo y versátil para hacer frente al siempre cambiante panorama del *malware*. La estructura de datos *HNSW (Hierarchical Navigable Small World)* [9] ofrece una estructura jerárquica para organizar y buscar eficientemente en grandes conjuntos de datos. La principal motivación detrás de la utilización de *HNSW* radica en su capacidad para reducir drásticamente los tiempos de búsqueda manteniendo altos niveles de precisión.

1.2. Objetivo

El objetivo de este Trabajo de Fin de Máster es la implementación de un sistema basado en la estructura de datos *HNSW* que permita la búsqueda eficiente de hashes de similitud aproximada. Como conjunto de datos se dispone de una base de datos conformada por hashes de programas legítimos, recogidos de una serie de sistemas operativos de Windows. Como caso de uso se ha considerado un usuario interesado en conocer si existe un hash determinado en la base de datos o consultar aquellos hashes que tengan una determinada puntuación de similitud respecto a un hash dado. Dado que una comparación 1:1 es inviable por rendimiento (la base de datos dispone de 2.804.630 registros), se pretende implementar y evaluar diversas configuraciones de *HNSW*, con objeto de encontrar la mejor configuración para el caso de uso particular estudiado.

1.3. Estructura del documento

Este documento se encuentra dividido en 6 capítulos y un anexo. En el Capítulo 2 se introducen los conceptos relativos a los algoritmos de similitud aproximada y a la estructura *HNSW*, explicando brevemente su funcionamiento. En el Capítulo 3 se hace una descripción más detallada del sistema desarrollado, entrando en detalles técnicos como la arquitectura del sistema y su funcionamiento, haciendo hincapié en los factores que más se han tenido en cuenta durante la etapa de desarrollo. En el Capítulo 4 se realiza una experimentación del sistema, poniendo a prueba su eficiencia y rendimiento con diferentes configuraciones y distintos tamaños del conjunto de datos, para realizar finalmente un análisis de los resultados. En el Capítulo 5 se realiza un análisis de las aplicaciones existentes en la comunidad, describiendo sus características y sus limitaciones con respecto al sistema desarrollado. Finalmente, en el Capítulo 6 se describen las conclusiones del trabajo, se mencionan las complicaciones que han surgido durante el desarrollo y se definen algunos aspectos y áreas de mejora de cara al futuro.

Al final del documento se encuentra el Anexo A, donde se muestra de forma detallada cómo se ha distribuido el tiempo invertido en este trabajo.

Capítulo 2

Conceptos previos

En este capítulo se introducen algunos conceptos que son necesarios para la comprensión del trabajo desarrollado, introduciendo en primer lugar qué son los algoritmos de similitud aproximada y su funcionamiento. Después, se definen las estructuras de datos de búsqueda eficiente haciendo hincapié en la estructura sobre la que se ha realizado el sistema, describiendo sus características más importantes.

2.1. Algoritmos de similitud aproximada

Un *hash* se define como una función matemática que se utiliza para convertir una cantidad de datos de cualquier tamaño en un valor único [10]. El propósito general de los hashes es facilitar la comparación y la identificación de datos de manera eficiente: el hash se utiliza para *resumir los datos de entrada en una forma que sea fácil de comparar*, sin tener que comparar directamente los datos originales. En análisis forense digital, su propósito general es verificar la integridad de los datos [11]. Por ejemplo, en una descarga de software, el hash se utiliza para asegurarse de que el archivo descargado no se ha corrompido durante la transferencia: el usuario descarga el archivo original y un fichero adicional donde se encuentra su hash calculado, para luego utilizar la misma función hash con el fin de calcular su hash y verificar que el archivo descargado es idéntico al original.

Los *algoritmos de coincidencia por similitud aproximada* o *algoritmos de similitud aproximada* [12] son una técnica de hashing que se utiliza para identificar similitudes entre dos conjuntos de datos, en contraste con los algoritmos de hashing criptográficos, más centrados en la integridad de los datos. Los algoritmos de similitud aproximada se pueden basar en detalles del contenido del archivo (*bytewise*), su estructura y formato (sintáticos), o incluso su contexto (semánticos).

En lugar de producir un solo valor hash para los datos de entrada, los algoritmos de similitud aproximada normalmente generan múltiples valores hash que combinan (de alguna manera) y representan diferentes características del dato de entrada. A diferencia de los hashes criptográficos, estos valores hash se utilizan para comparar la similitud entre datos, en lugar de comparar directamente los datos entre sí.

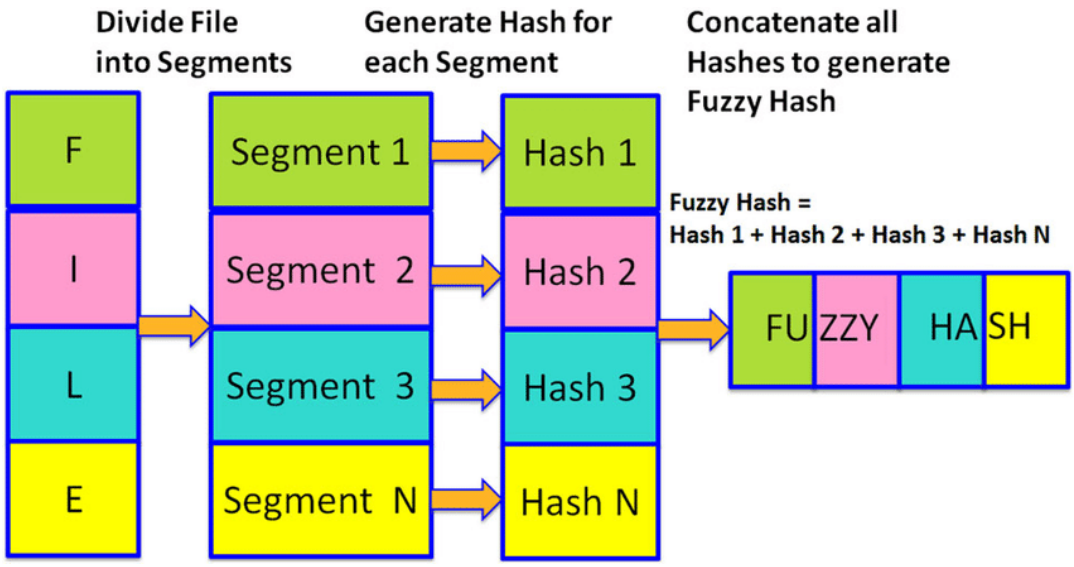

Figura 2.1: Proceso de generación de hash de tipo BBH [1]

Estos algoritmos de similitud aproximada pueden identificar archivos que tienen características similares, incluso si tienen diferentes nombres, tamaños o estructuras de archivo. En el campo de la seguridad informática, y la respuesta a incidentes en particular [11], esto es importante porque los cibercriminales a menudo hacen pequeñas modificaciones en el malware existente para evitar la detección por parte de los antivirus tradicionales. Así pues, estos algoritmos pueden ser capaces de detectar variantes de malware que hayan podido ser modificadas para evadir la detección [13]. Además, son muy eficientes cuando se trabaja con grandes volúmenes de datos. De nuevo, esto resulta de interés en el campo del análisis de malware, donde se deben analizar constantemente grandes cantidades de archivos cuya intención es desconocida.

En general, los algoritmos de similitud aproximada trabajan en dos fases diferentes: generación de hashes y comparación de hashes [7]. La generación de hashes incluye el procedimiento de extracción y codificación de características de la entrada binaria dada. Basándose en cómo las funciones de hash codifican estas características y calculan el hash final, se pueden categorizar los algoritmos existentes en los siguientes tipos [14]:

- **Context-triggered piecewise hashing (CTPH)**: este tipo de algoritmos dividen la secuencia de entrada en segmentos basados en la existencia de contextos especiales (llamados puntos de activación) dentro del objeto de datos. Un contexto (pequeña secuencia de entrada) se considera punto de activación si cumple con cierta propiedad (por ejemplo, una pequeña secuencia de bytes cuya suma de verificación es igual a un valor predefinido). Una función CTPH calcula un valor hash para los segmentos individuales y los concatena en un hash final.
- **Statistically-Improbable Features (SIF)**: estos algoritmos tratan de localizar

un conjunto de características compuestas por secuencias de bits poco habituales. De esta forma, la comparación entre hashes consiste en hallar similitudes entre estos conjuntos de características poco habituales.

- **Block-based rebuilding (BBR)**: este tipo de algoritmos hacen uso de datos externos a la propia entrada del algoritmo. Estos datos pueden obtenerse aleatoriamente o ser siempre los mismos. La forma de comparar hashes se realiza generalmente mediante el cálculo de distancia *Hamming*.
- **Blocked-based hashing (BBH)**: esta categoría engloba los algoritmos que generan un pequeño bloque del hash después de haber procesado una cierta cantidad de bytes de la entrada. El hash final resulta de la concatenación de todos los pequeños bloques generados. La Figura 2.1 muestra una idea abstracta de cómo funciona.
- **Locality-Sensitive Hashing (LSH)**: este algoritmo trata de reducir el inmenso espacio de posibles entradas a una cantidad discreta de posibilidades, maximizando la probabilidad de que dos entradas similares generen una salida casi idéntica.

Algunos de los algoritmos de similitud aproximada más conocidos son:

- **SSDEEP** [15]: está basado en la técnica de CTPH para la generación de hashes. Fue propuesto en el año 2006, siendo uno de los primeros en destacar dentro de los algoritmos de similitud aproximada, y fue ideado inicialmente para ayudar en la detección de spam. Es un algoritmo rápido y su capacidad de correlación depende en gran medida de la presencia de un gran fragmento de datos comunes, siendo además difícilmente escalable.
- **SDHASH** [8]: basado en la técnica BBH para la generación de hashes, fue propuesto en el año 2010. A diferencia del anterior, es más preciso y escalable.
- **TLSH** [16]: basado en la técnica LSH para generación de hashes, fue desarrollado por Trend Micro en el año 2013. Su uso está más centrado en el análisis de malware. Entre sus características, destaca su rapidez y su escalabilidad. Además, es significativamente más difícil de atacar o evadir que otros algoritmos, como los anteriormente mencionados.

2.2. Estructuras de datos para búsqueda eficiente y Hierarchical Navigable Small World (HNSW)

Con el crecimiento exponencial de los datos, se ha vuelto cada vez más desafiante y crucial encontrar formas eficientes de buscar similitudes y encontrar elementos relacionados en grandes volúmenes de información. En el ámbito de la búsqueda eficiente, se han explorado y desarrollado numerosas estructuras de datos a lo largo del tiempo. En este trabajo, se ha optado por utilizar *Hierarchical Navigable Small World* (*HNSW*) [9], una estructura de datos que proporciona un método de indexación y búsqueda de alta

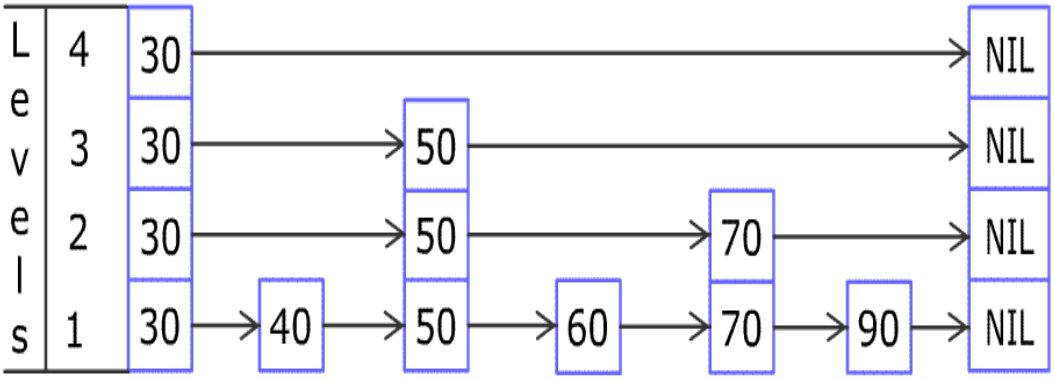

Figura 2.2: Ejemplo básico de una skip list [2]

eficiencia para conjuntos de datos de alta dimensionalidad [17–19]. Su funcionamiento se basa en la combinación de dos conceptos fundamentales en estructuras de datos: *skip lists* y *Navigable Small Worlds*.

Las *skip lists* son una estructura de datos eficiente y flexible utilizada para realizar búsquedas, inserciones y eliminaciones en listas enlazadas de manera rápida. Concretamente, el coste promedio de las operaciones es $\mathcal{O}(\log n)$ [20]. Sin embargo, esta estructura tiene como desventaja que no presenta un buen rendimiento con datos de alta dimensionalidad, y además es muy poco práctico equilibrar la *skip list* después de las operaciones de inserción y eliminación.

En la Figura 2.2 se muestra gráficamente un ejemplo de cómo se ve una *skip list* sencilla. La idea principal es proporcionar una forma eficiente de buscar elementos en una lista enlazada, evitando la necesidad de recorrer todos los elementos en orden secuencial. Esto se logra mediante la introducción de "saltos" o "niveles" adicionales en la estructura de la lista enlazada, lo que permite una búsqueda más rápida al saltar por encima de múltiples elementos en lugar de recorrer cada uno individualmente. Cada nodo contiene un valor y una serie de punteros que apuntan a otros nodos en diferentes niveles. El nivel más bajo corresponde a la lista enlazada original, donde los nodos están conectados siguiendo un cierto orden. Los niveles superiores contienen menos nodos y proporcionan saltos más grandes dentro de la estructura.

La clave para la eficiencia de las *skip lists* radica en cómo se generan y utilizan los niveles. Durante la inserción de un nuevo elemento, se decide aleatoriamente cuál va a ser su nivel. Los punteros en cada nivel apuntarán a los nodos que se deben saltar para encontrar rápidamente el lugar adecuado donde insertar el nuevo elemento. Esta técnica aleatoria asegura un buen rendimiento promedio y evita el caso peor de búsqueda lineal en la lista enlazada original.

La búsqueda se realiza de manera similar a la inserción. Comenzando desde el nivel superior, se sigue avanzando hasta que se encuentra un nodo cuyo valor es mayor o igual

al valor buscado. Entonces, se desciende al siguiente nivel y se repite el proceso hasta llegar al nivel más bajo, donde se realiza una búsqueda lineal para encontrar el elemento exacto (o el más cercano, en su defecto).

El concepto *Navigable Small World* (*NSW*) [21] fue introducido a lo largo de varios artículos entre 2011 y 2014 [19,22]. La idea fundamental radica en construir un grafo de proximidad que contenga enlaces tanto a corta como a larga distancia, lo que permite acotar los tiempos de búsqueda promedios a $\mathcal{O}(\log n \cdot \log k)$, donde cada vértice en el grafo se conecta (de promedio) con k otros vértices, a los que se denominan "vecinos", manteniendo una lista con todos ellos. En su conjunto, se conforma la estructura del grafo.

Al realizar una búsqueda en este grafo, se comienza en un punto de entrada predefinido. Desde este punto de entrada, se deberá identificar cuál de sus vecinos está más cerca de éste para desplazarse en ese sentido. Se repite el proceso de búsqueda, moviéndose de vértice en vértice e identificando aquellos vértices vecinos más cercanos. Eventualmente, no se encontrarán vértices más cercanos que el vértice actual, lo que significa que se ha llegado a un mínimo local, que será la condición de parada. Hay que destacar también que el enrutamiento en estos grafos puede dividirse en dos fases: en la fase *zoom out* se van recorriendo los vértices de menor grado hasta ir pasando por los vértices de mayor grado (fase *zoom in*). Este modo de enrutamiento puede causar que se llegue a un mínimo local falso durante la fase *zoom out*.

En *HNSW* se combinan las ideas anteriores para construir una estructura jerárquica capaz de optimizar la búsqueda y agilizar los saltos entre los niveles y los vecinos, mejorando la eficiencia y la precisión de la búsqueda en comparación con *NSW* en grandes conjuntos de datos.

HNSW está basada en la idea de crear una estructura jerárquica de grafos en la que cada nodo representa un objeto de datos y está conectado a un conjunto de otros nodos en diferentes niveles de la jerarquía. En el nivel superior de la jerarquía, los nodos están pobremente conectados, formando una estructura ligera, mientras que conforme se desciende en los niveles, los nodos están cada vez más conectados y forman grafos más robustos. Esto permite que la búsqueda se realice de manera eficiente y con alta precisión. Concretamente, el coste promedio de búsqueda es $\mathcal{O}(\log n)$ [9].

Durante la búsqueda, se ingresa a la capa superior a través del punto de entrada. En esta capa se encuentran nodos de bajo grado (lo que equivaldría a la fase de *zoom in* descrita para *NSW*). Se realiza un recorrido por el grafo, tal y como se realiza en *NSW*, para tratar de encontrar el mínimo local de dicha capa. Cuando se alcanza este mínimo local, se desplazará a ese mismo nodo, pero al situado en en la capa inferior. Desde este nodo, se vuelve a realizar otra búsqueda en ese nuevo grafo para encontrar otro mínimo local. Este proceso se repite hasta llegar a la capa 0.

A la hora de construir el grafo, cada uno de los nodos se va insertando uno a uno de forma iterativa. La capa máxima que ocupará un nodo vendrá determinado por un parámetro conocido como *multiplicador de nivel*. Según [9], el mejor rendimiento se logra cuando se minimiza la superposición de vecinos compartidos entre capas. Reducir este parámetro puede ayudar a minimizar la superposición (empujando más nodos a la capa

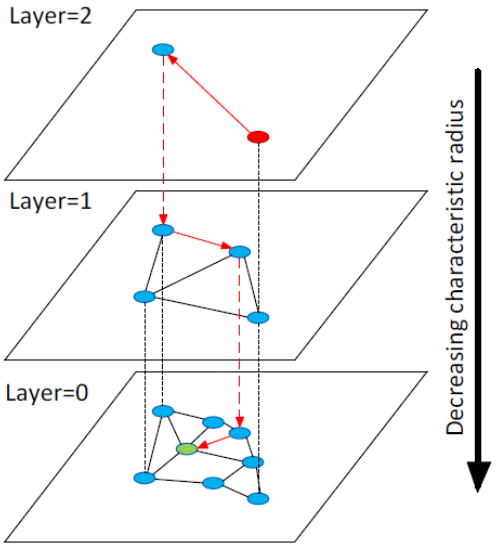

Figura 2.3: Ejemplo básico de una estructura HNSW [3]

0), pero esto aumenta, a su vez, el promedio de nodos recorridos durante la búsqueda. Por lo tanto, es necesario buscar un valor que equilibre ambos aspectos.

La construcción del grafo comienza en la capa superior. Después de ingresar al grafo, el algoritmo recorre las aristas, tratando de encontrar el vecino más cercano al nodo que se intenta insertar. Después de encontrar el mínimo local, se desciende a la siguiente capa (como la búsqueda). Este proceso se repite hasta alcanzar la capa de inserción obtenida según el multiplicador de nivel. Aquí comienza la segunda fase de construcción donde los vecinos más cercanos que vayan apareciendo a partir de esta capa serán vecinos candidatos que tendrá el nuevo nodo. El número de vecinos definitivo del nuevo nodo vendrá determinado por un parámetro preconfigurado de la estructura. Sabiendo cuáles serán los nuevos vecinos para este nuevo nodo en una determinada capa, se realizarán las conexiones bidireccionales entre ellos, quedando el nodo insertado en dicha capa de la estructura. Este proceso debe ser repetido hasta llegar a la capa 0.

Hay que tener en cuenta también que, tras varias iteraciones, es posible que existan nodos con un número muy elevado de conexiones, pudiendo esto impactar en el rendimiento de la estructura. Para minimizar este impacto, existen parámetros de control en la estructura que limitan el número de conexiones que puede tener un nodo, en cualquier nivel o en la capa 0. Todos los parámetros de configuración de la estructura se detallan más adelante (concretamente, en la Sección 3.3.1).

Capítulo 3

Sistema desarrollado

En este capítulo se describe en detalle el sistema desarrollado, comenzando con una visión arquitectural, la cual trata de facilitar su comprensión con ayuda de varios tipos de diagramas. También se comentan los aspectos importantes en cuanto a la tecnología utilizada y una descripción a nivel algorítmico de las funcionalidades del sistema.

3.1. Requisitos del sistema

Las necesidades expuestas anteriormente convergen en la creación de un sistema que sea capaz de hacer búsquedas eficientes usando la estructura de datos *HNSW*. Para tener claro lo que se procede a diseñar, se realiza una definición previa de requisitos funcionales y no funcionales.

- **Requisitos funcionales (RF):**

RF1. El sistema debe permitir la creación de una estructura *HNSW*, permitiendo al usuario definir los parámetros de configuración relevantes.

RF2. El sistema debe proporcionar la funcionalidad de inserción de nodos en una estructura *HNSW*.

RF3. El sistema debe permitir la realización de búsquedas de elementos similares a un nodo en función de un umbral de similitud.

RF4. El sistema debe ofrecer la posibilidad de realizar búsquedas de los elementos más cercanos a un objetivo (vecinos), permitiendo definir el número máximo de vecinos.

RF5. El sistema debe contar con la capacidad de guardar el modelo de la estructura *HNSW* en un almacenamiento secundario para su posterior uso.

RF6. El sistema debe contar con la capacidad de restaurar un modelo de la estructura *HNSW* desde un almacenamiento secundario.

- **Requisitos no funcionales (RNF):**

RNF1. El sistema debe ser capaz de soportar versatilidad de tipos de datos y procesar diferentes tipos de datos (no solo numéricos) que permitan definir distancias entre ellos.

RNF2. El sistema debe ser capaz de manejar grandes volúmenes de datos y realizar búsquedas eficientes en tiempos razonables, garantizando así un rendimiento óptimo, utilizando los recursos de manera eficiente.

En definitiva, el desarrollo de este sistema de búsqueda eficiente tiene como objetivo principal brindar a los usuarios una herramienta potente y versátil para abordar el desafío de buscar elementos similares en conjuntos de datos de alta dimensionalidad sobre una estructura de datos *HNSW* (véase Sección 2.2), que además cuente con una arquitectura correcta para un amplio soporte en cuanto a versatilidad de datos. Cumpliendo con los requisitos establecidos, se espera que el sistema sea una solución eficaz y eficiente para el caso de uso planteado inicialmente (véase Sección 1.2).

3.2. Arquitectura del sistema

Con el fin de entender mejor la arquitectura de la aplicación se han realizado diagramas *UML* (*Unified Modeling Language*) [23]. *UML* es un estándar que permite describir un sistema en forma de "plano". Se incluyen en él aquellos aspectos tales como funcionalidad del sistema, expresiones del lenguaje concreto, etc. Este estándar utiliza y asocia elementos de distintas formas con el fin de desarrollar diagramas que representen ciertos aspectos tanto estructurales como estáticos de un sistema en concreto, como la colaboración de los distintos objetos de este de forma específica.

La Figura 3.1 muestra el diagrama de clases para tener una visión global de todo el sistema, junto con todos sus componentes. La arquitectura está conformada por varias clases. La clase principal de la arquitectura es *HNSW*, que representa, tal y como dice su nombre, una estructura *HNSW*. Esta clase mantiene los parámetros de configuración y las operaciones para realizar operaciones de inserción y búsqueda usando la citada estructura. Para representar los nodos que conforman la estructura y tratar de abordar el reto de la versatilidad de datos, se ha decidido aplicar el patrón de software *Abstract Factory* [24]. Este patrón permite crear familias de objetos relacionados sin especificar sus clases concretas. En lugar de crear objetos directamente, se utiliza una interfaz o clase abstracta que define los métodos necesarios para crear los objetos de la familia. Luego, se implementan diferentes fábricas concretas que heredan de esta fábrica abstracta y se encargan de implementar los métodos para la creación de objetos específicos de cada familia.

En este caso, se dispone de una interfaz *Node*. En esta arquitectura, se otorga soporte a datos de tipo numérico y hashes de similitud aproximada. Para cada uno de estos tipos de nodo, se tienen sus correspondientes factorías *NumberNodeFactory* y *HashNodeFactory*, que a su vez heredan de la clase *NodeFactory*. Para crear un nuevo nodo, por tanto, habrá que hacer uso de estas factorías, pasando como parámetro el dato en cuestión. En caso de que el dato sea un hash, se deberá también especificar el algoritmo usado para

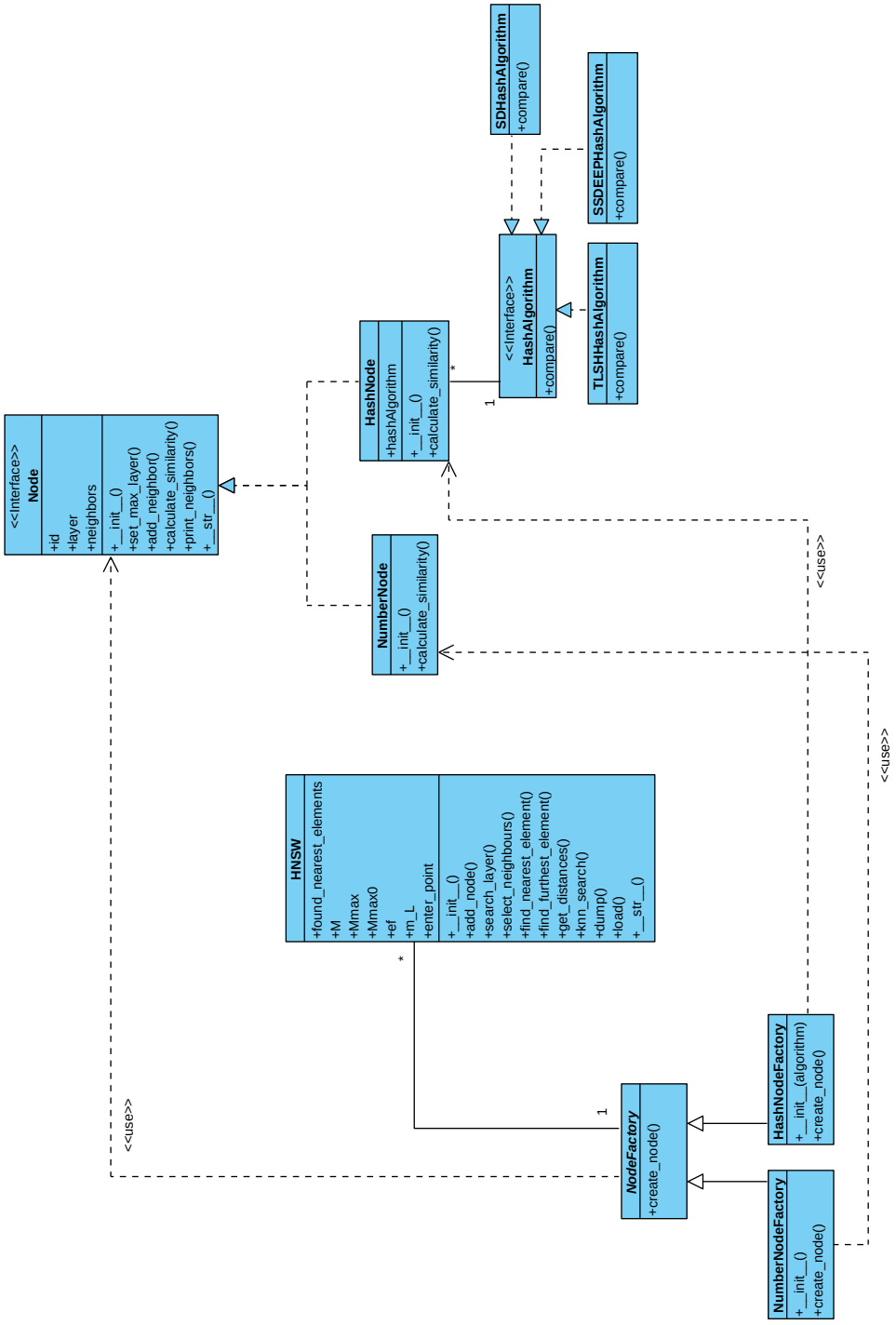

Figura 3.1: Diagrama de clases UML

generar dicho hash. Para cubrir los distintos algoritmos, se crea una interfaz llamada *HashAlgorithm* que será implementada por cada uno de los algoritmos a los que se quiera dar soporte en el sistema. De esta forma, no es necesario crear un tipo de nodo distinto por cada uno de los algoritmos de similitud aproximada que se tengan. Actualmente, el sistema ofrece soporte para tres tipos distintos de algoritmos de similitud aproximada: *SSDEEP*, *SDHASH* y *TLSH* (clases *SDHashAlgorithm*, *SSDEEPHashAlgorithm* y *TLSHHashAlgorithm*, respectivamente).

Gracias a esta estructura, se logra abordar los desafíos de la versatilidad de datos de una manera sencilla y modular, permitiendo que la arquitectura software del sistema sea lo suficientemente desacoplada como para dar soporte a la versatilidad de datos mencionada, sin la necesidad de modificar el núcleo de la infraestructura.

3.3. Funcionamiento y características

La elección de un lenguaje de programación adecuado para implementar algoritmos y estructuras de datos es crucial para garantizar un desarrollo eficiente y exitoso. En el caso de la implementación de *HNSW*, *Python* se destaca como una opción altamente beneficiosa ya que se trata de un lenguaje de programación ampliamente utilizado y reconocido por su enfoque en la legibilidad, facilidad de uso y rápido prototipado.

Estas características (legilibidad, facilidad de uso y rapidez de prototipado) han marcado fundamentalmente la elección de este lenguaje para el desarrollo. La estructura clara y expresiva de este lenguaje facilita la traducción de algoritmos complejos en un código limpio y conciso, lo que a su vez facilita el mantenimiento y la colaboración en el desarrollo del proyecto.

Otra ventaja significativa es la disponibilidad de una amplia gama de bibliotecas y módulos que simplifican el proceso de implementación de *HNSW*. Esto implica que algunas de las tecnologías de las que depende el desarrollo de este proyecto, como pueden ser los algoritmos de similitud aproximada, ya están actualmente implementados y pueden acoplarse al sistema desarrollado con facilidad.

La flexibilidad y versatilidad de *Python* también juegan un papel importante en la elección de este lenguaje. Es un lenguaje multiparadigma que admite programación orientada a objetos, funcional y procedural. Esto permite adaptar y extender fácilmente la implementación según los requisitos específicos del proyecto, pudiendo crear una arquitectura eficaz para abordar los retos de diseño que puedan surgir (algunos de ellos se han comentado previamente).

Si bien es cierto que *Python* no es conocido por ser el lenguaje más rápido en términos de rendimiento (pues es un lenguaje interpretado), ofrece también opciones para mejorar la velocidad de ejecución. La integración con extensiones como *CPython* permite compilar partes críticas del código en lenguaje de bajo nivel, lo que puede mejorar significativamente el rendimiento. Además, *Python* permite la integración con bibliotecas y módulos escritos en lenguajes de bajo nivel, como C++, lo que brinda la posibilidad de aprovechar implementaciones optimizadas cuando sea necesario. Este proyecto, sin embargo, no ha usado ninguna de estas posibilidades en el momento actual del desarrollo.

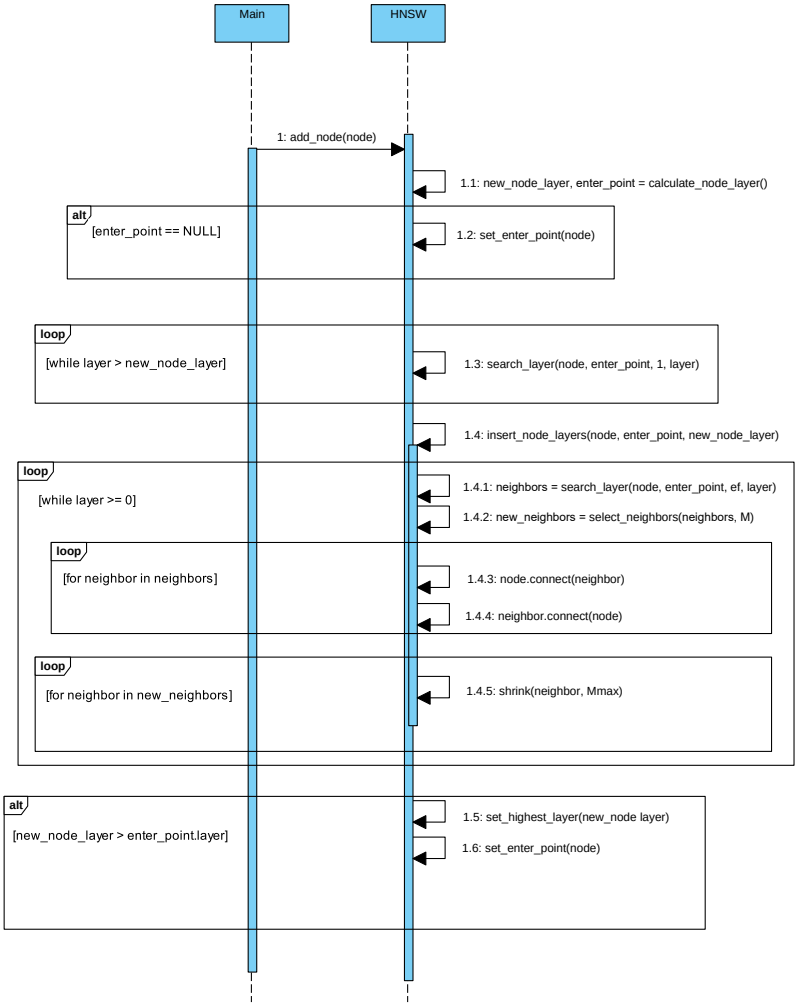

Figura 3.2: Diagrama de secuencia UML para inserción de un nodo

Como punto adicional, *Python* presume de tener una gran comunidad y un soporte activo, pudiendo así minimizar los problemas y complicaciones que puedan surgir durante el desarrollo.

3.3.1. Funcionamiento

En este apartado se explica de forma detallada el funcionamiento del sistema desarrollado. Con el fin de comprenderlo mejor, se han diseñado varios diagramas de secuencia UML para visualizar correctamente las funcionalidades principales del sistema.

Parametrización de la estructura

El primer paso fundamental para comenzar a usar *HNSW* es la creación y configuración de la estructura. La Figura 3.2 muestra un diagrama de secuencia UML sobre el proceso de inserción de un nodo en la estructura. Inicialmente, para permitir esta operación es necesario establecer cuatro parámetros de configuración que determinarán el comportamiento y desempeño de la estructura. Estos parámetros son:

- M: Controla el nivel de conexión entre los nodos en la estructura. Es el número máximo de vecinos que tendrá un nodo en el momento de ser insertado.
- ef: Determina la cantidad de nodos vecinos que se explorarán durante la búsqueda.
- $Mmax$: Establece el número máximo de vecinos que un nodo puede tener en todos los niveles de la estructura, excepto en la capa cero.
- $Mmax_0$: Establece el número máximo de vecinos que un nodo puede tener en la capa cero.

En definitiva, estos parámetros de configuración deberán ser seleccionados cautelosamente en función del modelo, pues tendrán un impacto en los tiempos de búsqueda y en la precisión. Se trata de encontrar aquellos valores para mantener un equilibrio entre tiempo y precisión, para así garantizar un rendimiento óptimo.

Algoritmo de inserción

Una vez que está creada la estructura, se puede comenzar añadir nodos en ella. Lo primero que se realiza, antes de comenzar con la inserción, es calcular el nivel máximo que ocupará el nodo (`calculate_node_layer()`). La fórmula utilizada para calcular este nivel se basa en la idea de lograr una distribución equilibrada de los nodos en los diferentes niveles de la estructura, de manera que queden la gran mayoría de ellos en la capa inferior y algunos de ellos suban a capas superiores. La fórmula utilizada es la siguiente:

$$\text{Nivel} = \lfloor -(logU)\, m_L \rfloor$$

El parámetro m_L se trata del multiplicador de nivel que se ha mencionado en la Sección 2.2. En [9] se define una *regla de oro* para calcular este parámetro, que a su vez depende del parámetro M. Concretamente:

$$m_L = \frac{1}{\ln M}$$

Un valor alto de M causará que la probabilidad de que un nodo ocupe un nivel alto sea menor, y viceversa, un valor bajo causará que el nodo tenga mayor probabilidad de subir a niveles superiores. Teniendo ya el nivel máximo que ocupará el nodo, se procede a insertarlo a partir del nivel máximo hasta la capa inferior.

Un punto importante a la hora de ir construyendo la estructura es definir el punto de entrada. Este quedará denotado por el primer nodo que ha ocupado el nivel más alto

de la estructura (`set_enter_point()`). Cuando se calcula el nivel de cierto nodo que se quiere insertar, este nodo será el punto de entrada si el nivel resultante está vacío.

Para continuar con la inserción, será necesario recorrer el grafo desde el punto de entrada e ir descendiendo por los niveles hasta llegar al nivel calculado por la formula anterior (`search_layer()`). Una vez se haya alcanzado este nivel, se realizará la inserción de este nodo en cada uno de los niveles hasta el nivel inferior (`insert_node_layers()`). Por cada una de las capas, se realiza una búsqueda de los ef nodos más cercanos al nuevo nodo, y de estos nodos se escogerán los M más cercanos (`select_neighbors()`). La cercanía viene determinada por el resultado del algoritmo de similitud aproximada que se esté utilizando. Con estos M nodos, se establecerán las conexiones bidireccionales, quedando el nodo insertado y conectado dentro de la estructura (`node.connect(neighbor)` y `neighbor.connect(node)`).

No hay que olvidarse de los parámetros $Mmax$ y $Mmax0$. Así pues, será necesario comprobar por cada uno de estos vecinos donde se ha establecido un nuevo enlace que no se hayan superado estos umbrales. En caso de que alguno de ellos haya superado algún umbral, se volverá a aplicar el algoritmo para seleccionar los vecinos, eliminándose los enlaces con aquellos vecinos más lejanos y así cumplir con el umbral (`shrink()`).

Algoritmo de búsqueda

Para realizar la búsqueda es necesario establecer el nodo de consulta y el tipo de búsqueda que se desea realizar. Se permiten dos tipos de búsqueda:

- **Basada en vecinos cercanos**: Dado un nodo de consulta, se devolverán aquellos K nodos más cercanos a este, siendo K un número dado como parámetro en la búsqueda.

- **Basada en umbral**: Dado un nodo de consulta, se devolverán todos aquellos vecinos que superen cierto umbral de similitud dado como parámetro.

La Figura 3.3 muestra un diagrama de secuencia UML para el proceso de búsqueda basado en vecinos cercanos. Esta búsqueda se comienza en la capa superior, donde se encuentra el punto de entrada. A partir de este nodo, se recorre el grafo en este primer nivel para encontrar el vecino más cercano ($ef = 1$; `search_layer()`).

En este proceso de búsqueda se comienzan definiendo tres listas. La primera de ellas (`visited_elements`) tendrá la función de ir acumulando todos aquellos nodos que ya hayan sido visitados para no quedarse en un bucle infinito de búsqueda. La segunda lista (`candidates`), que en realidad será una cola de prioridad, guardará aquellos nodos que deben ser recorridos durante la búsqueda, por orden de cercanía con el nodo consulta. Por último, se mantendrá una lista denominada `currently_found_nearest_neighbours`, que en definitiva mantendrá los nodos más cercanos encontrados hasta el momento. Esta cercanía se calcula en función del resultado de aplicar un algoritmo de similitud aproximada entre los nodos encontrados y el nodo a buscar.

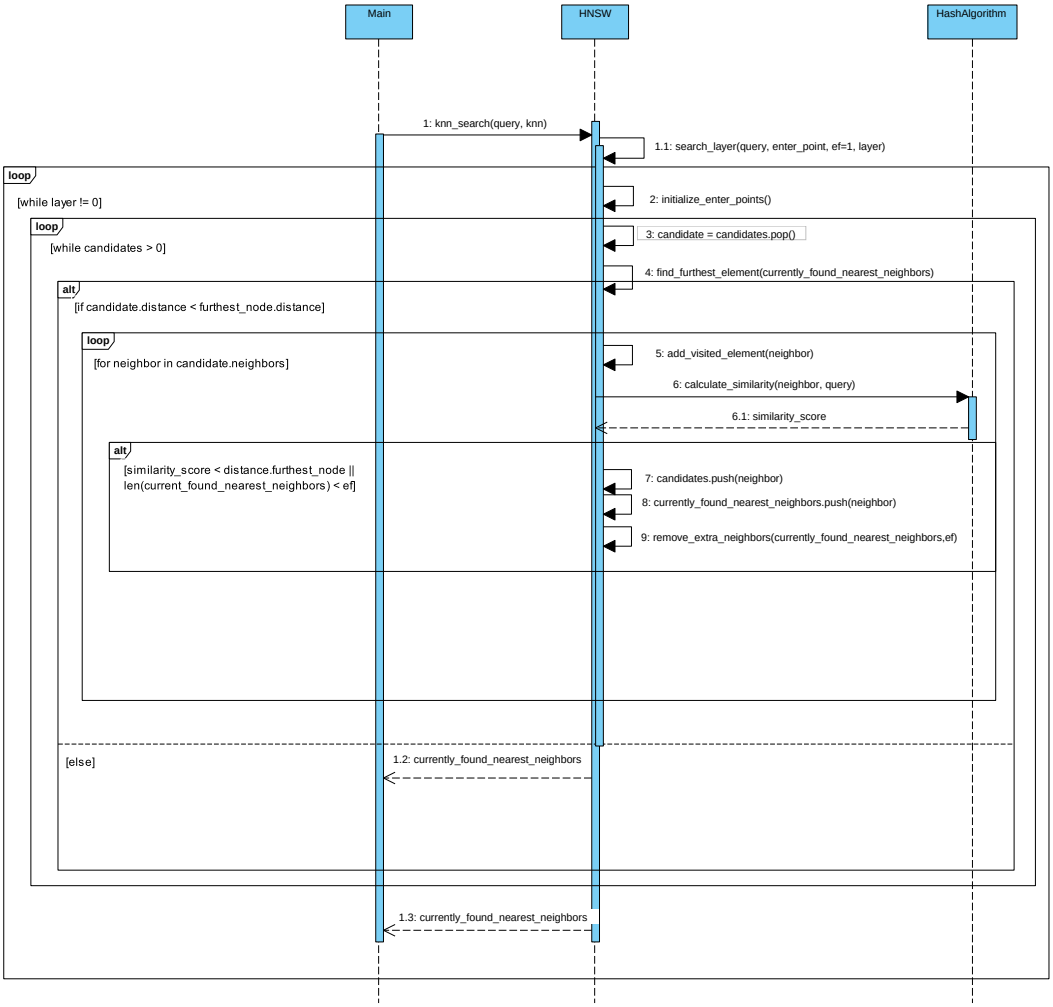

Figura 3.3: Diagrama de secuencia UML para búsqueda basada en vecinos cercanos

Lo primero que se hace es inicializar `candidates` con el punto de entrada de la estructura (`initialize_enter_point()`). A continuación, se inicia un bucle mientras haya elementos en esta cola. En cada iteración, se extraerá el primer nodo de esta cola (es decir, el vecino más cercano detectado hasta el momento) (`candidates.pop()`). Después, se consulta la distancia de este nodo con la distancia al nodo más lejano encontrado hasta el momento, disponible en `currently_found_nearest_neighbours` (`find_furthest_element(currently_found_nearest_neighbors)`). En caso de que la distancia sea mayor, implica que los candidatos restantes tienen una mayor distancia que los nodos cercanos que se han encontrado hasta el momento, por lo que terminaría la búsqueda.

En caso contrario, se van a recorrer todos los vecinos de este nodo. Por cada uno de los nodos, en primer lugar, se añade a `visited_elements` para evitar volver a visitarlo en el futuro (`add_visited_element(neighbor)`). A continuación, se calcula la distancia de este vecino con el nodo consulta (`calculate_similarity(neighbor, query)`). Si la longitud de `currently_found_nearest_neighbours` no ha superado todavía el umbral establecido por ef, o bien lo ha superado pero el último elemento de esta lista es más lejano que el vecino que se está consultando en esta iteración, este vecino será añadido tanto a esta lista (`currently_found_nearest_neighbours.push(neighbor)`) como a `candidates` (`candidates.push(neighbor)`) para ser visitado posteriormente. Todo esto se repite hasta que o bien `candidates` quede vacía o el primer nodo candidato esté más lejos que el último de `currently_found_nearest_neighbours`.

En cuanto al funcionamiento de la búsqueda basada en umbral, es muy similar a la ya vista. Su funcionamiento queda reflejado en el diagrama de secuencia UML de la Figura 3.4, con la única diferencia que la lista `currently_found_nearest_neighbours` no tiene un límite de nodos marcado por *ef*, sino que se añadirá en ella todo aquel nodo que supere cierto umbral de similitud.

Finalmente, para guardar y restaurar los modelos creados, se ha utilizado la biblioteca *pickle*, que permite volcar objetos a disco de una forma sencilla para luego poder recuperarlos y continuar con su operativa.

3.4. Disponibilidad del código

Todo lo desarrollado para este trabajo se ha publicado en un repositorio en GitHub, disponible en `https://github.com/reverseame/HNSW4hashes` y se encuentra liberado bajo licencia GNU/GPLv3.

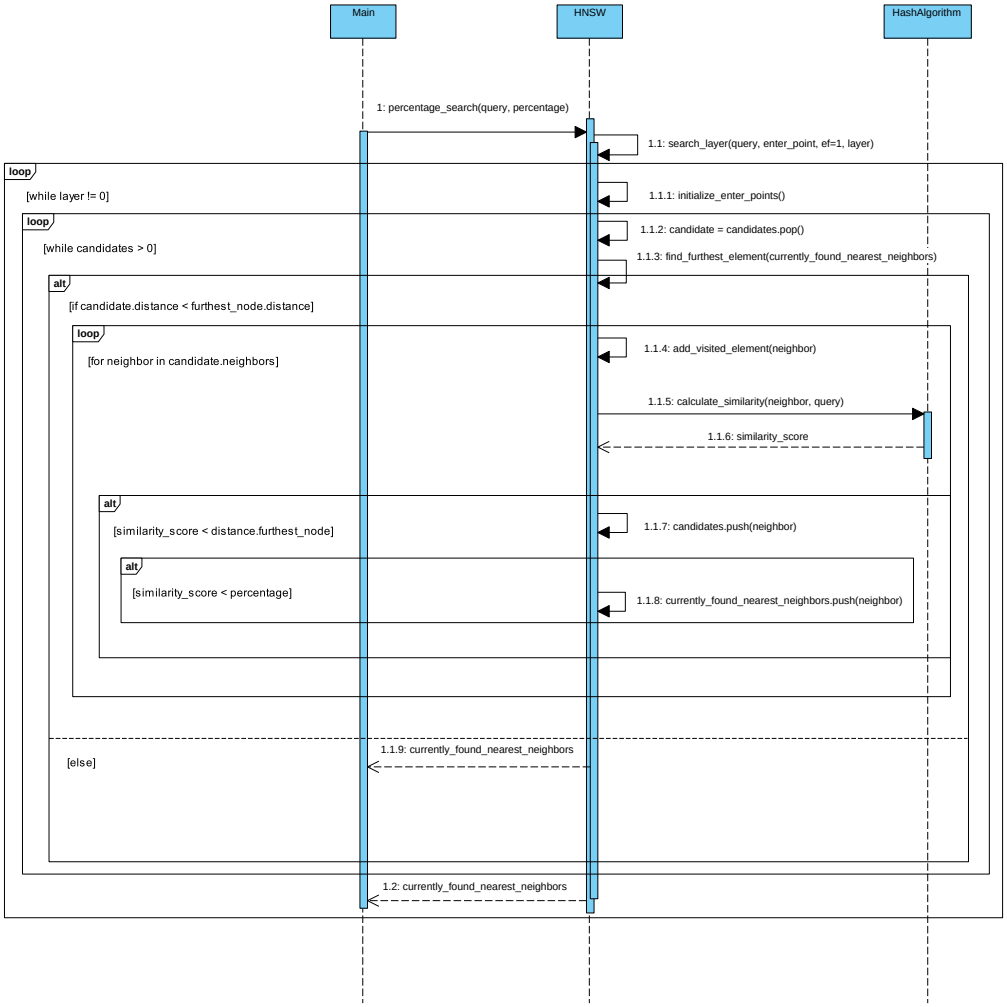

Figura 3.4: Diagrama de secuencia UML para búsqueda basada en umbral

Capítulo 4

Experimentación

Con el fin de comprobar el sistema desarrollado, se ha llevado cabo un análisis exhaustivo para evaluar su rendimiento y validar su correcto funcionamiento. Para lograr esto, se han realizado una serie de pruebas y experimentos que permitirán obtener métricas cuantitativas y realizar comparaciones relevantes. El propósito principal de esta etapa es llevar a cabo una evaluación del sistema, es decir, medir y analizar el desempeño en términos de tiempos de ejecución y precisión. Se recopilarán y analizarán los resultados obtenidos de cada prueba, utilizando gráficas para facilitar la interpretación de los resultados y la identificación de patrones o tendencias. Esto permitirá tomar decisiones informadas sobre posibles mejoras y optimizaciones que se puedan aplicar al sistema.

En este capítulo se hablará en primer lugar del entorno donde se han llevado a cabo las evaluaciones y se explicará en que consiste el conjunto de datos sobre el que se ha trabajado. Después, se definirán aquellas métricas que se han seleccionado para la correcta evaluación del sistema. Finalmente, se realizará un análisis de los resultados obtenidos y se discutirá el rendimiento del sistema.

4.1. Definición del entorno

El entorno seleccionado para realizar las pruebas consiste en un equipo propiedad de la Universidad de Zaragoza, con un procesador Intel(R) Core(TM) i7-10700 a 2.90GHz de 8 núcleos y 16 hilos. El sistema cuenta con 16GiB de RAM DDR4 a 3200MHz y ejecuta un sistema operativo Debian en su versión 11.

4.2. Base de datos

El conjunto de datos que se ha escogido para trabajar consta de una colección de aproximadamente 3 millones de páginas pertenecientes 13.000 módulos de ficheros de sistema de los sistemas operativos Windows 7, Windows 8.1, Windows Server 2012 R2, Windows Server 2016 y Windows Server 2019. Una *página* es la unidad de memoria virtual utilizada para gestionar el intercambio de datos entre la memoria principal y

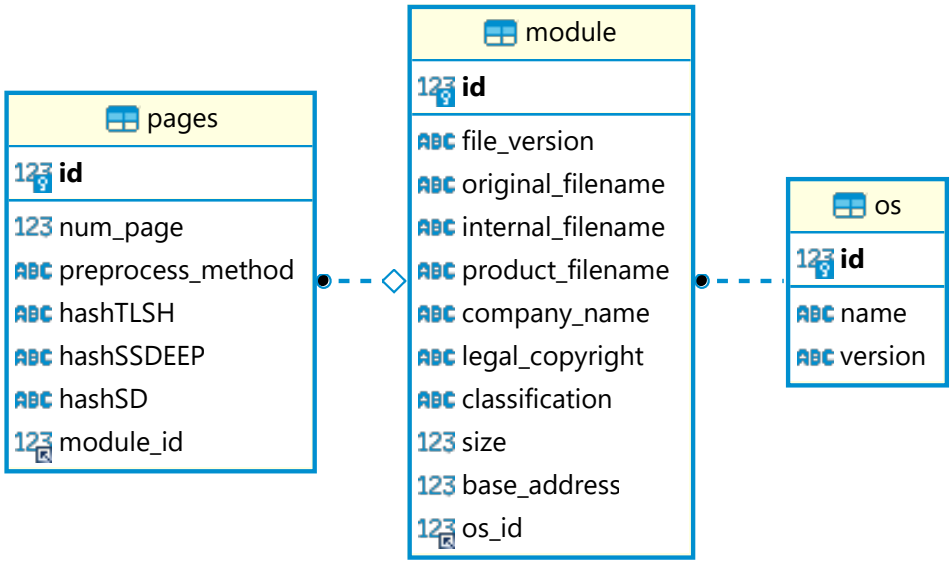

Figura 4.1: Diagrama Entidad-Relación de la base de datos

almacenamiento secundario [25]. En Windows, un *módulo* es un fichero binario (como una biblioteca de enlace dinámico o DLL) que contiene funciones, datos y recursos que pueden ser utilizados por otros programas y que se han cargado en memoria [26]. Por cada uno de estos módulos, existe información de su versión, del nombre del fichero, de la compañía desarrolladora, de su tamaño y de la dirección de memoria base donde se localiza en el volcado de memoria. Por cada una de las páginas, se guarda el hash de similitud aproximada en tres algoritmos distintos: *SSDEEP*, *SDHASH* y *TLSH*. Esta base de datos relacional se encuentra guardado en un sistema gestor *MySQL*.

Esta base de datos fue creada por un alumno de la Universidad de Valencia durante sus prácticas en empresa en el grupo de investigación DisCo de la Universidad de Zaragoza en 2021. Cabe destacar que inicialmente la base de datos presentaba una construcción errónea, donde no existían relaciones entre las tablas. Tras corregir estos errores, creando un diagrama Entidad-Relación apropiado y normalizando (véase la Figura 4.1), se consiguió un diseño más correcto, logrando además una optimización del espacio ocupado (concretamente, pasando de 4GB a 1.7GB). Esta base de datos sirve como una *lista de hashes permitidos* (o benignos) que pueden encontrarse durante la ejecución de programas en entornos Windows. Para la evaluación del sistema desarrollado se ha utilizado el hash TLSH.

4.3. Métricas y configuraciones HNSW

Se detallan a continuación las diversas métricas y configuraciones de parámetros de *HNSW* utilizadas para evaluar el desempeño y la eficacia del sistema implementado.

Estas métricas se seleccionaron cuidadosamente para capturar aspectos clave del sistema y brindar una evaluación completa de su funcionamiento.

En primer lugar, se realiza una evaluación de los tiempos de indexación. Es decir, se mide la cantidad de tiempo invertido en insertar nuevos nodos a un modelo *HNSW*. Con el fin de comprobar el comportamiento y la evolución de estas medidas, se combinan distintas configuraciones del modelo y distintos tamaños del conjunto de datos. Los distintos tamaños corresponden al 3 %, 10 %, 25 %, 50 %, 75 % y 100 %. Cabe destacar que el tamaño del 3 % se seleccionó especialmente por ser el máximo tamaño posible sobre el que se puede realizar comparaciones 1:1, necesarias para la comparación de rendimiento entre el sistema desarrollado y un algoritmo de búsqueda mediante fuerza bruta. El formato del nombre de las configuraciones usado en este trabajo viene determinado de la siguiente forma: $\text{CFG}_{M,ef,Mmax,Mmax_0}$.

Una vez se han creado y medido las estructuras *HNSW* con las distintas combinaciones de configuraciones y tamaños, se han medido los tiempos de búsqueda basada en vecinos cercanos. Para ello, por cada uno de los distintos tamaños, se obtendrán 100 hashes que no pertenezcan al conjunto de datos usado para la creación de la estructura *HNSW*. La elección de este valor se basa en la idea de que un tamaño de muestra suficientemente grande puede ofrecer una estimación confiable del rendimiento promedio del algoritmo en diferentes configuraciones y tamaños de datos.

También resulta de gran interés medir la precisión del algoritmo durante estas búsquedas, conociendo cómo va variando la media en función de las configuraciones y del tamaño de datos. Se realizarán comparaciones realizando búsquedas de estos 100 hashes escogidos con $K = 1$. De esta forma, se compararán las precisiones del nodo más cercano encontrado en cada configuración. Con el fin de tener una mejor idea del comportamiento del algoritmo, se realizarán búsquedas con $K = 10$ y se realizará una evaluación de la variación de la precisión media en función del valor de K.

Tal y como se comentó en la Sección 3.3, el algoritmo también soporta una búsqueda basada en umbral. Por tanto, también se ha evaluado su precisión comparando el número de nodos que superan un umbral de 60 en función de las configuraciones y el tamaño[1].

Por último se realizan medidas de precisión y tiempo basadas en el algoritmo de búsqueda mediante fuerza bruta con el fin de verificar las ventajas del sistema desarrollado sobre este tipo de búsqueda. Es decir, partiendo de un conjunto de datos usable por ambos algoritmos (3 % del conjunto de datos), se comparará cada uno de ellos con el resto. Se medirá el tiempo invertido en la búsqueda y se guardarán aquellos nodos que superen el umbral de 60. Posteriormente, se compararán los resultados obtenidos mediante este algoritmo de búsqueda por fuerza bruta con cada una de las diferentes configuraciones *HNSW*.

Finalmente, para comprender dentro del sistema desarrollado cuáles son aquellos parámetros de configuración de *HNSW* que más impactan en el rendimiento, se van a considerar nuevas combinaciones de configuración. De los cuatro parámetros de configuración, tres de ellos se mantendrán estáticos y el restante se irá duplicando (desde 4

[1] En el algoritmo TLSH utilizado, una similitud de 0 entre dos elementos significa la máxima coincidencia entre ellos.

hasta 64) para observar su impacto.

4.4. Análisis de resultados

El objetivo principal de este análisis es obtener una comprensión más profunda del desempeño y comportamiento del sistema desarrollado, así como identificar patrones, tendencias y posibles áreas de mejora, además de obtener la mejor configuración posible para el conjunto de datos que se presenta.

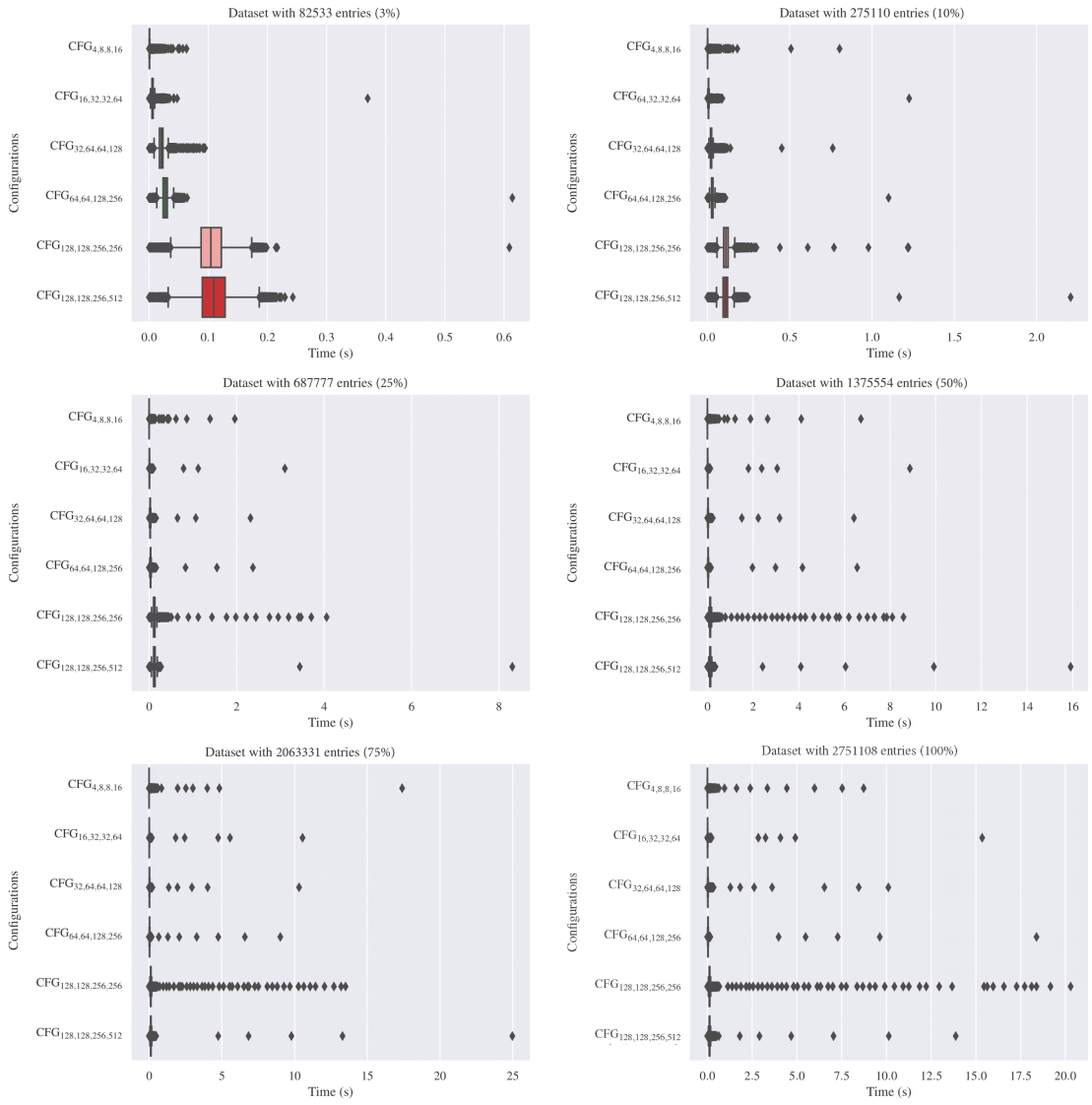

Figura 4.2: Tiempos de inserción para distintas configuraciones y tamaños del conjunto de datos.

En la Figura 4.2 se muestran los tiempos medios de inserción para cada subconjunto de datos considerado y las distintas configuraciones mencionadas como diagramas de cajas. Recuérdese que un diagrama de caja muestra la distribución de un conjunto de datos y se compone de una caja que abarca el rango entre el primer y tercer cuartil, una línea que indica la mediana y dos bigotes que muestran la variabilidad de los datos. Además, pueden incluir valores atípicos.

Se puede apreciar como a partir de la configuración $CFG_{128,128,256,256}$ se incrementa el tiempo medio de inserción de forma significativa con respecto al resto de configuraciones, incrementando los tiempos medios de inserción y el número de valores atípicos existentes, sobre todo en los conjuntos de datos con tamaños más altos.

La Figura 4.3 muestra los tiempos medios de búsqueda basada en vecinos cercanos. Se puede apreciar un comportamiento similar para las configuraciones más altas ($CFG_{128,128,256,256}$ y $CFG_{128,128,256,512}$). No obstante, apenas existe diferencia de tiempos entre los diferentes tamaños del conjunto de datos.

La Figura 4.4 muestra las precisiones medias para búqueda basada en vecinos cercanos con $K = 10$. Como se puede apreciar, los valores medios de precisión se estabilizan. Configuraciones posteriores no suponen una mejora notable a partir de la tercera configuración ($CFG_{32,64,64,128}$). Se puede observar también como esta tendencia es todavía más pronunciada conforme se incrementa el tamaño del conjunto de datos. Sorprendentemente, en el caso de contar con todo el conjunto de datos, la configuración $CFG_{4,8,8,16}$ muestra una variabilidad en los resultados no esperados (comparados con el resto de configuraciones). Unos valores de M y $Mmax$ muy bajos que pueden causar que muchos nodos se queden aislados y sin conexiones, causando una pérdida considerable de precisión. Un comportamiento similar se observa cuando se evalúan los tiempos para para cada uno de los valores K desde 1 hasta 10, como se muestra en la Figura 4.5. En este caso, al contemplar el conjunto de datos entero, se aprecia como la precisión de los vecinos disminuye a partir del séptimo vecino ($k = 7$).

La Figuras 4.6 y 4.9 muestran respectivamente el impacto que tienen en el rendimiento los parámetros ef y $Mmax0$. Se puede concluir que estos son los parámetros que más impacto tienen en cuanto a tiempos de búsqueda, pudiendo llegar a suponer un incremento exponencial en cuanto al tiempo, pero sin que suponga un incremento en la precisión en la misma proporción. La Figura 4.7 representa el impacto que tiene M sobre el rendimiento del sistema. Se puede apreciar como los tiempos de búsqueda disminuyen si se aumenta este parámetro, pues implica que habrá un menor número de capas, y por tanto, un menor número de grafos a recorrer. Finalmente, la Figura 4.8 representa el impacto que tiene sobre el sistema la variable M. Se observa que es el parámetro que menos impacto tiene en cuanto al tiempo y la precisión.

Finalmente, se presentan los resultados de la comparativa del algoritmo de búsqueda con umbral entre la estructura *HNSW* y un algoritmo de búsqueda mediante fuerza bruta. En la Figura 4.10 se puede apreciar mejoras considerables en cuanto a los tiempos de búsqueda medios para todas las configuraciones estudiadas. En cuanto a las precisiones, se puede apreciar como las 3 últimas configuraciones se ajustan más a la primera. En la Figura 4.11, se manifiestan los tiempos totales de búsqueda, y se puede apreciar de

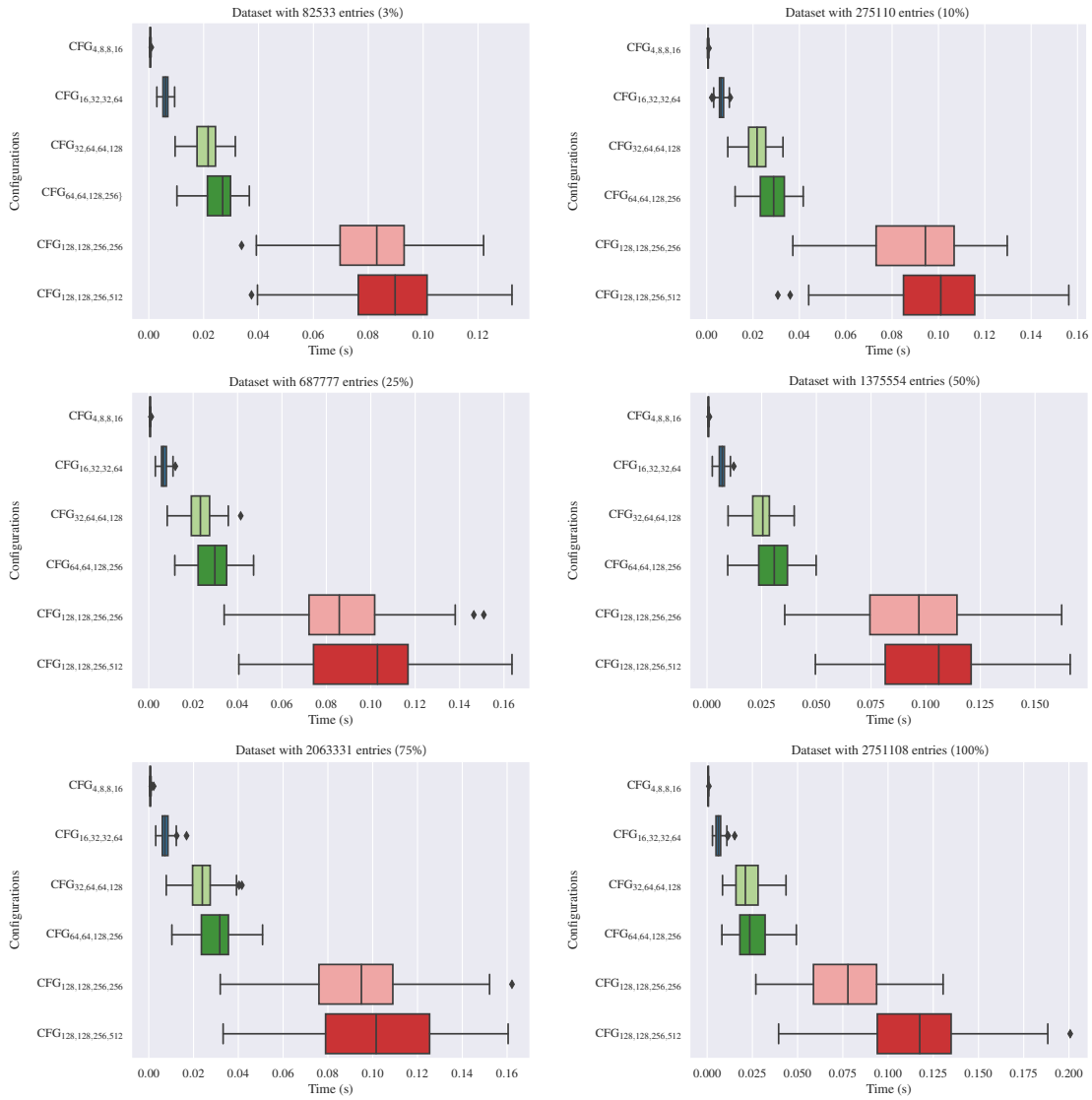

Figura 4.3: Tiempos de búsqueda basada en vecinos cercanos para distintas configuraciones y tamaños del conjunto de datos.

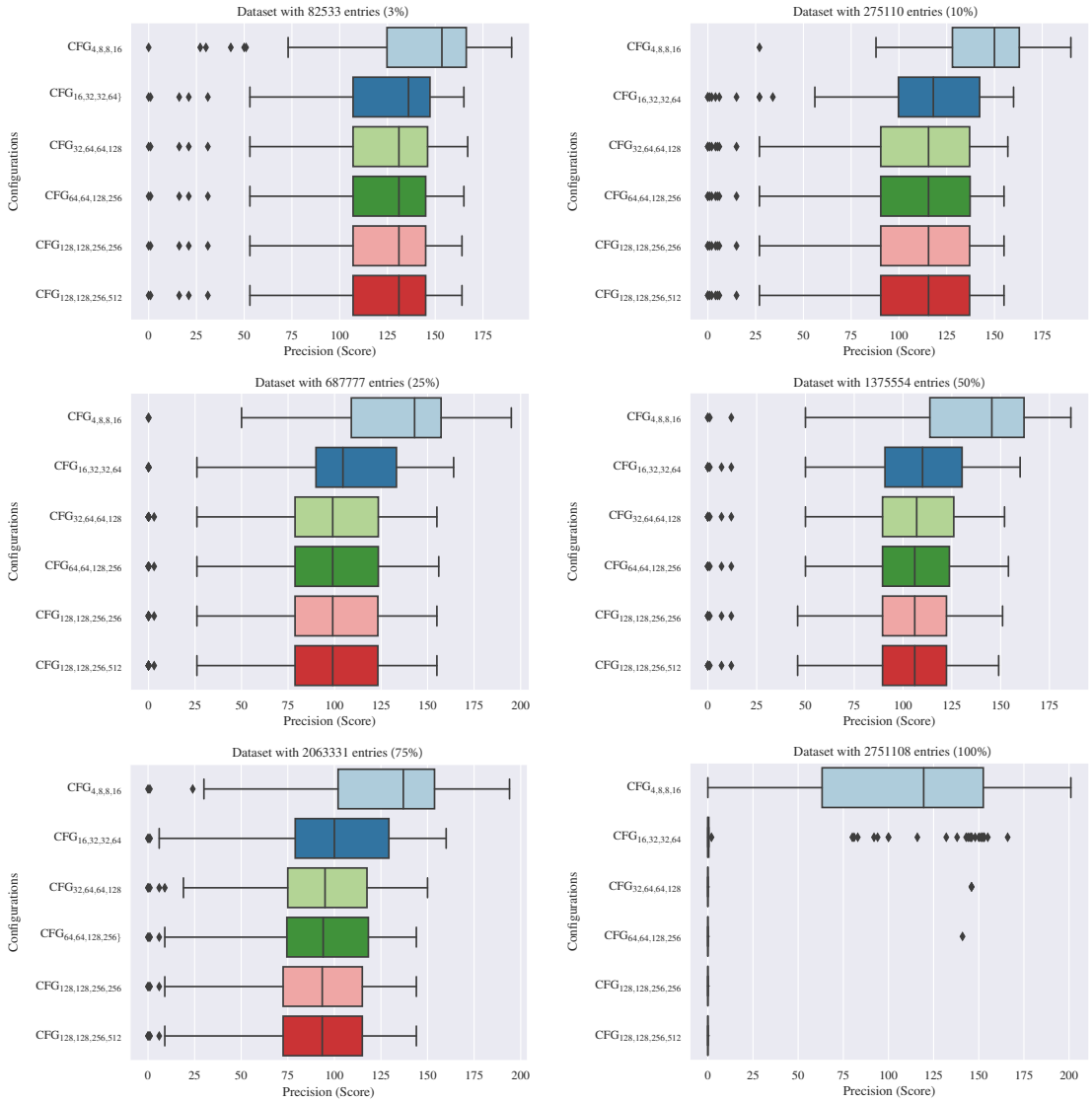

Figura 4.4: Precisión para la búsqueda de vecinos cercanos con $K = 1$ para distintas configuraciones y tamaños del conjunto de datos.

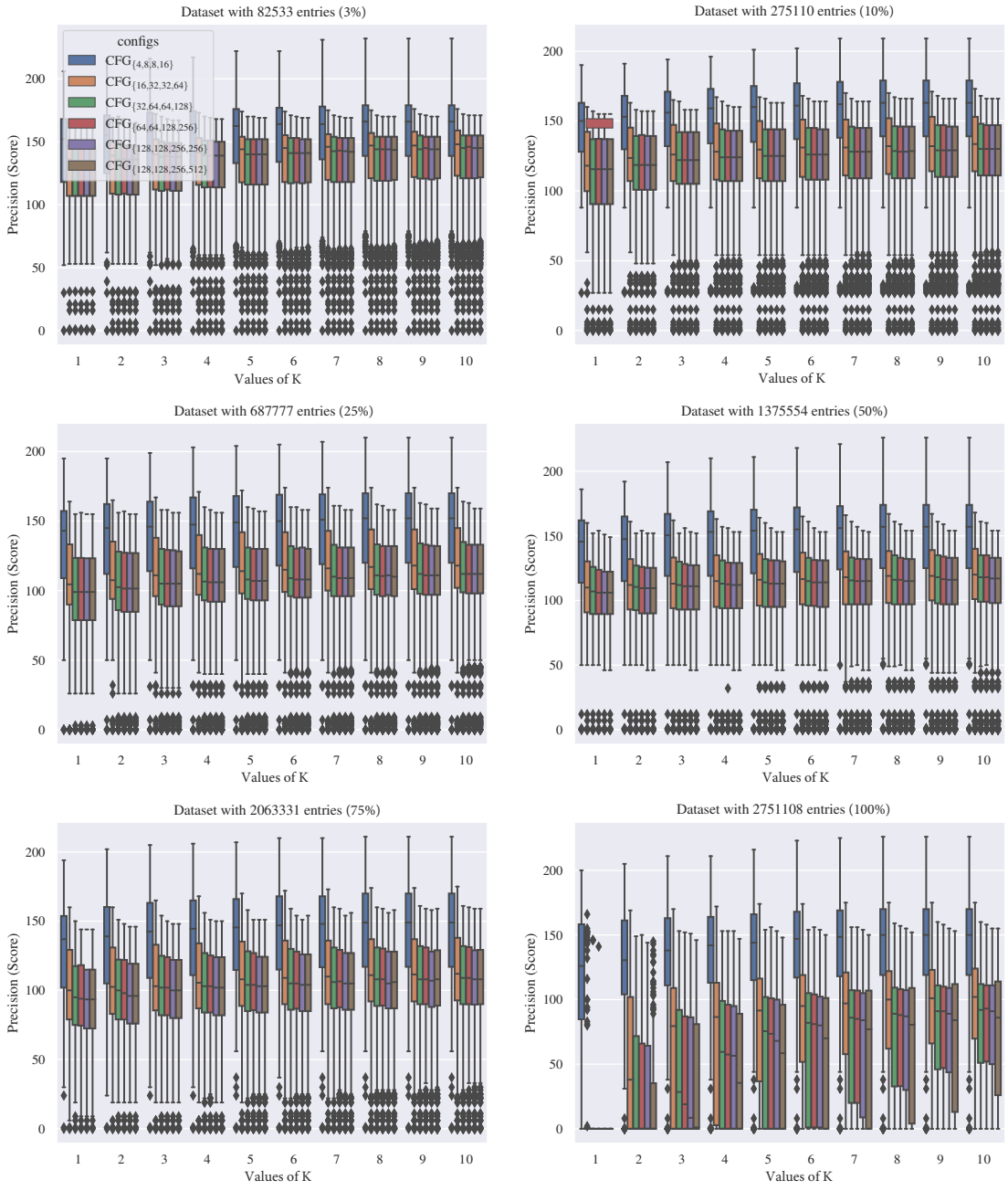

Figura 4.5: Precisión para la búsqueda de vecinos cercanos con $K = 10$ para distintas configuraciones y tamaños del conjunto de datos.

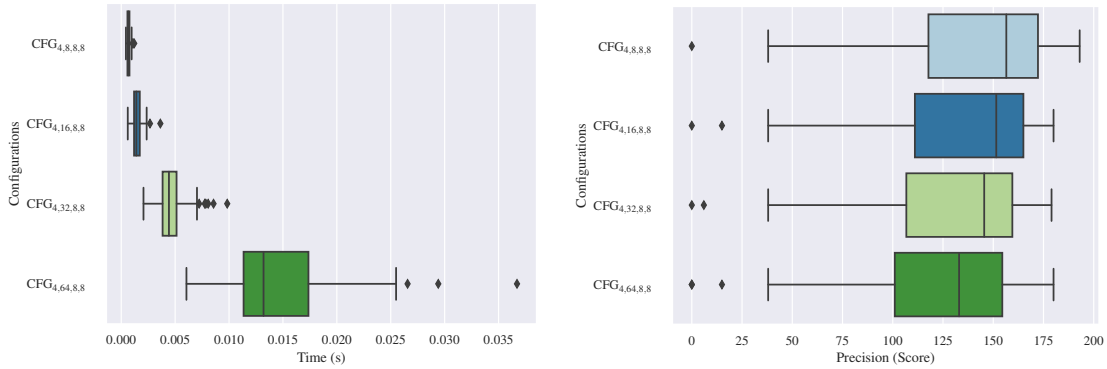

Figura 4.6: Variación del tiempo y la precisión en función de ef

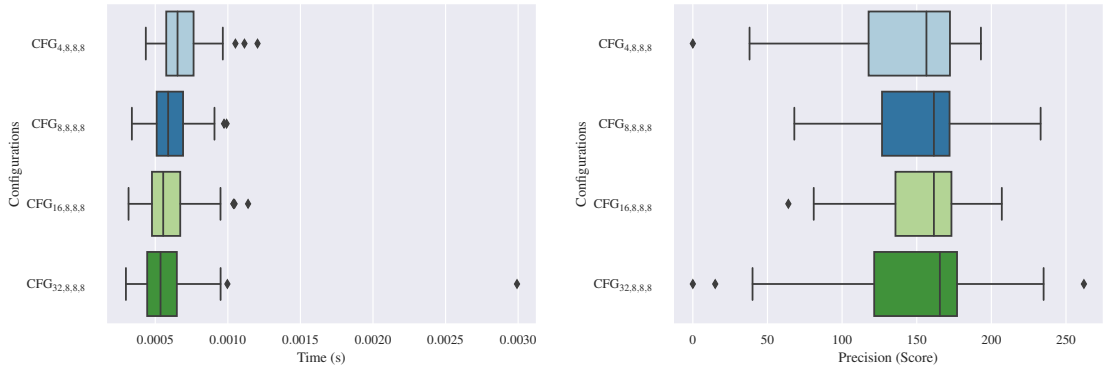

Figura 4.7: Variación del tiempo y la precisión en función de M

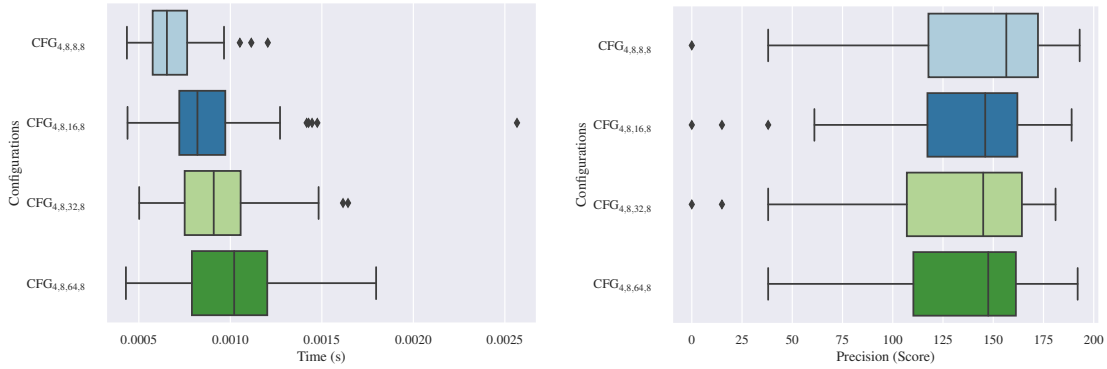

Figura 4.8: Variación del tiempo y la precisión en función de $Mmax$

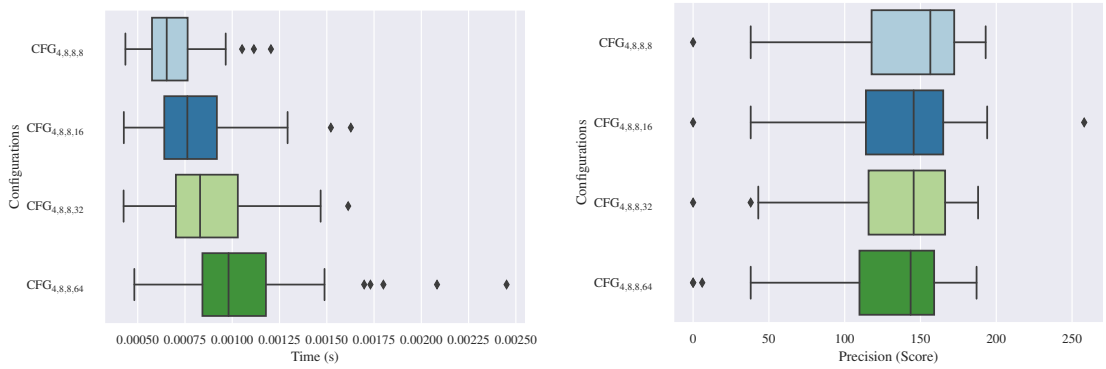

Figura 4.9: Variación del tiempo y la precisión en función de $Mmax_0$

forma más clara las importantes mejoras que tiene el uso de la estructura *HNSW*.

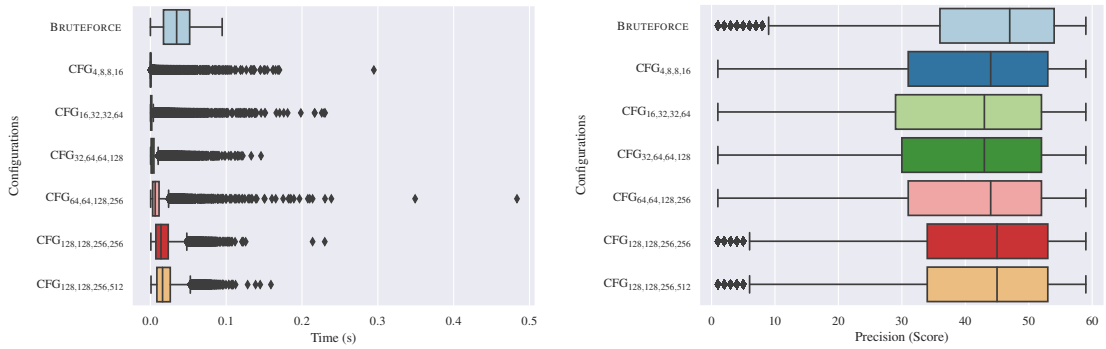

Figura 4.10: Tiempos y precisión de búsqueda mediante fuerza bruta versus búsqueda mediante estructura HNSW basada en umbral

El análisis realizado en este estudio proporciona evidencia sólida de que el sistema desarrollado tiene la capacidad de mejorar significativamente los tiempos de búsqueda sin comprometer la precisión de los resultados. En particular, se destaca que la configuración $CFG_{64,64,128,256}$ muestra un rendimiento óptimo en términos de precisión y tiempos de búsqueda. Esta configuración específica logra mantener un nivel aceptable de precisión y proporciona una notable mejora en los tiempos de ambos métodos de búsqueda soportados.

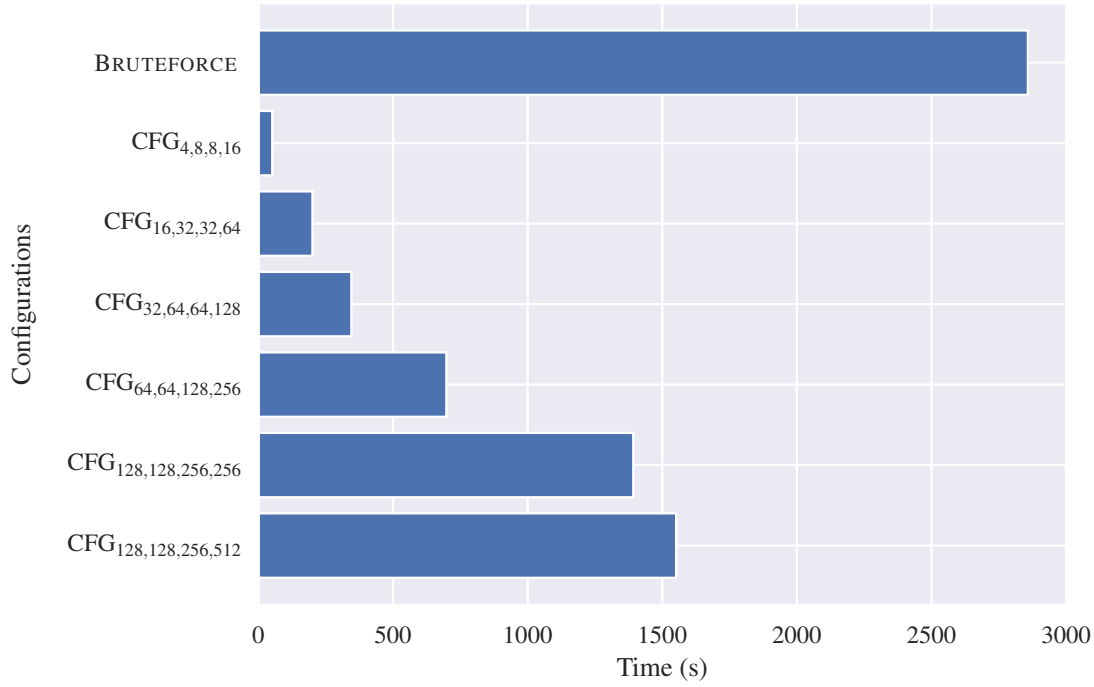

Figura 4.11: Tiempos totales de búsqueda mediante fuerza bruta versus búsqueda mediante estructura HNSW basada en umbral

Capítulo 5

Aplicaciones similares

Es importante destacar que la estructura *HNSW* ha despertado un gran interés en la comunidad de desarrolladores debido a su eficiencia y efectividad. A lo largo de los años, se han desarrollado diversas implementaciones de esta tecnología, sobre todo centradas en búsquedas vectoriales.

La más popular es *HNSWlib* [27], que se trata de una implementación del algoritmo en C++ con soporte Python a través de *bindings*. Destaca por ser una biblioteca ligera, con una arquitectura robusta, que permite definir cómo se calculan las distancias entre nodos de forma personalizada. Por otro lado, *Facebook AI Similarity Search* [28] es una implementación desarrollada por *Meta AI* sobre C++ que incluso cuenta con soporte para aceleración por GPU, e implementa la misma estructura de datos. Además de estas aplicaciones, existen otras basadas en las ya especificadas, que cuentan con algunos cambios mínimos. La comunidad también ha hecho esfuerzos en tratar de hacer llegar *HNSW* a otros lenguajes como Java, Go, Rust, Node.js, Ruby, .NET, etc.

La mayoría de estas aplicaciones cuentan con un desarrollo muy refinado. Concretamente, utilizan tecnología de programación multiproceso de memoria compartida (en particular, *OpenMP* [29]) para optimizar los tiempos de ejecución al máximo. No obstante, cuentan con la desventaja de poseer una arquitectura limitada: al estar orientadas expresamente para búsquedas vectoriales, donde los datos son numéricos, hace especialmente complicado su uso para datos que no numéricos (como los hashes de similitud aproximada). Paralelamente al desarrollo de este TFM ha aparecido [30], que proporciona la estructura *HNSW* en código *Python*. Sin embargo, presenta la misma limitación que las soluciones anteriores. Es por ello que se llegó a la conclusión de que era inviable adaptar una de estas soluciones como punto de partida para el sistema planteado en este trabajo.

A pesar de ello, estas aplicaciones han servido de gran inspiración y ayuda para desarrollar el sistema planteado ya que han proporcionado ideas innovadoras, enfoques eficientes y soluciones técnicas que han contribuido a mejorar y enriquecer la implementación para el sistema planteado en este trabajo. Al estudiar y analizar detenidamente estas soluciones, se han identificado elementos clave que han sido adaptados y aprovechados en el diseño y la implementación del sistema, permitiendo así aprovecharse de los

avances previos.

Gracias a todo ello, se ha podido desarrollar un sistema pionero en la comunidad, capaz de ofrecer una arquitectura abierta para extender el soporte de tipos de datos y con ello dar un paso más en el campo de las búsquedas aproximadas, abriendo camino a nuevas posibilidades que puedan surgir.

Capítulo 6

Conclusiones, problemas encontrados y trabajo a futuro

Se presentan a continuación los principales hallazgos y logros alcanzados en este trabajo de investigación. Además, se abordan los desafíos y obstáculos encontrados durante el desarrollo del proyecto, junto con las posibles líneas de trabajo futuro que podrían explorarse para continuar mejorando y ampliando el sistema propuesto.

6.1. Conclusiones

En este trabajo se ha abordado el desafío de mejorar la eficiencia en el análisis de código dañino en un contexto de grandes volúmenes de datos. Se ha demostrado la importancia de buscar soluciones más rápidas y efectivas para la detección y análisis, superando las limitaciones de los enfoques tradicionales, dando así también oportunidad de conocer y profundizar en los conocimientos dentro de este área.

Durante la implementación del sistema de búsqueda aproximada basado en la estructura *HNSW*, se ha hecho especial hincapié en mantener una arquitectura software robusta y se ha trabajado en optimizar su rendimiento al máximo, manteniendo siempre las buenas prácticas de desarrollo. Se ha llevado también a cabo una fase de experimentación donde se ha trabajado con una base de datos de páginas módulos y se han llevado a cabo diversas mediciones sobre tiempos de ejecución y precisión.

Este enfoque ha sido inspirado por el análisis de otras soluciones existentes. El énfasis establecido en la arquitectura ha resultado en una robustez capaz de soportad versatilidad en cuanto a los tipos de datos. En la fase de experimentación se ha podido comprobar el correcto funcionamiento y el buen rendimiento del sistema, así como los beneficios de utilizar esta solución frente a las tradicionales.

Con todo el tiempo y esfuerzo invertido en este trabajo, se ha logrado desarrollar toda una infraestructura con un sistema óptimo y único capaz de proponer una solución para el problema planteado. El sistema desarrollado es capaz de manejar grandes volúmenes de datos y mejorar los tiempos de búsqueda sin que suponga un gran impacto en la precisión, logrando así dar un paso más en el campo de las búsquedas aproximadas, aportando una

solución innovadora y demostrando su efectividad. En definitiva, este trabajo representa un paso adelante en la mejora de las técnicas de análisis de código dañino y contribuye al avance de la seguridad informática en un entorno de constante evolución y crecimiento de amenazas.

6.2. Problemas encontrados

El desarrollo de este trabajo ha sido largo y complicado. No obstante, su realización ha otorgado la oportunidad de adquirir grandes conocimientos sobre el mundo de las búsquedas aproximadas y algoritmos de similitud aproximada. Como se iba a trabajar sobre conceptos sobre los que no se tenía mucha experiencia, han aparecido ciertas dificultades que se han podido ir solventando con éxito durante el desarrollo del proyecto.

Entender la estructura de datos *HNSW* ha sido un desafío considerable, ya que su comprensión a partir de la documentación original mantenía un alto nivel de detalle técnico y formalismo matemático. Sin embargo, gracias a la existencia de artículos y recursos complementarios que simplificaban su explicación, se logró adquirir un entendimiento más claro y accesible de los conceptos subyacentes.

La fase de experimentación ha supuesto también un enorme reto. Encontrar las mejores técnicas para tratar de medir el rendimiento del sistema desarrollado y realizar un análisis correcto y sin sesgos ha tomado mucho tiempo. Se han tenido que tomar numerosos tipos de medidas y realizar muchas representaciones gráficas distintas para decidir cuáles podían ayudar más en el análisis, quedándose con las más interesantes y descartando aquellas que no tenían valor o que no se lograba obtener ninguna conclusión aparente.

A pesar de todos los desafíos y obstáculos encontrados a lo largo de este trabajo, éstas han sido superadas con éxito durante el desarrollo del proyecto.

6.3. Trabajo futuro

A pesar de que se ha conseguido desarrollar un sistema con bastante potencial, no deja de ser susceptible a algunas mejoras y es un nicho de investigación que puede ser ampliado. En primer lugar, pese haberse demostrado que *HNSW* hace un trabajo muy eficaz para el contexto de este problema, existen más algoritmos de búsqueda aproximada para ser estudiados y comparados con éste. Por ejemplo, *Product Quantization* [31] es un método de compresión que consiste en dividir los datos en subconjuntos y asigna un código a cada subconjunto para construir posteriormente un índice con estos códigos para realizar búsquedas eficientes en espacio comprimido.

Con el fin de mejorar el rendimiento de la aplicación, se propone una adaptación del código desarrollado a C++ o en su defecto, el uso de *CPython* para tratar de acelerar aquellas partes más críticas del algoritmo. Recuérdese que al ser *Python* un lenguaje interpretado puede suponer una pérdida en rendimiento. Una solución compilada podría suponer una mejora en cuanto al tiempo de ejecución.

Bibliografía

[1] Nitin Naik, Paul Jenkins, Nick Savage, Longzhi Yang, Tossapon Boongoen, Natthakan Iam-On, Kshirasagar Naik, and Jingping Song. Embedded YARA rules: strengthening YARA rules utilising fuzzy hashing and fuzzy rules for malware analysis. *Complex and Intelligent Systems*, 7:1–16, 11 2020.

[2] Amanda Lee and Travis Atkison. A Comparison of Fuzzy Hashes: Evaluation, Guidelines, and Future Suggestions. In *Proceedings of the SouthEast Conference*, ACM SE '17, page 18–25, New York, NY, USA, 2017. Association for Computing Machinery.

[3] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, jun 1990.

[4] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, February 2012.

[5] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. An Inside Look into the Practice of Malware Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 3053–3069, New York, NY, USA, 2021. Association for Computing Machinery.

[6] A. F. Webster and S. E. Tavares. On the Design of S-Boxes. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 523–534, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[7] Miguel Martín-Pérez, Ricardo J. Rodríguez, and Frank Breitinger. Bringing order to approximate matching: Classification and attacks on similarity digest algorithms. *Forensic Science International: Digital Investigation*, 36:301120, 2021.

[8] Frank Breitinger and Harald Baier. A fuzzy hashing approach based on random sequences and hamming distance. In *Proceedings of the 2012 Annual ADFSL Conference on Digital Forensics, Security and Law*, pages 89–100, 05 2012.

[9] Yu A. Malkov and D. A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, Apr 2020.

[10] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2015.

[11] Karen Kent, Suzanne Chevalier, Tim Grance, and Hung Dang. Guide to Integrating ForensicTechniques into Incident Response. Technical Report Special Publication 800-86, National Institute of Standards and Technology (NIST), August 2006.

[12] Frank Breitinger, Barbara Guttman, Michael McCarrin, Vassil Roussev, and Douglas White. Approximate Matching: Definition and Terminology. techreport NIST Special Publication 800-168, National Institute of Standards and Technology, May 2014.

[13] Vikram S. Harichandran, Frank Breitinger, and Ibrahim M. Baggili. Bytewise Approximate Matching: The Good, The Bad, and The Unknown. *J. Digit. Forensics Secur. Law*, 11:59–78, 2016.

[14] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Function representations for binary similarity. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2259–2273, 2022.

[15] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, 2006.

[16] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh – a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, 2013.

[17] Ponomarenko Alexander, Yury Malkov, L. Andrey, and Vladimir Krylov. Approximate Nearest Neighbor Search Small World Approach. In *Proceedings of the 2011 International Conference on Information and Communication Technologies & Applications*, 2011.

[18] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In Gonzalo Navarro and Vladimir Pestov, editors, *Similarity Search and Applications*, pages 132–147, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[19] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[20] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Average search and update costs in skip lists. *BIT Numerical Mathematics*, 32(2):316–332, June 1992.

[21] Marián Boguñá, Dmitri Krioukov, and K. C. Claffy. Navigability of complex networks. *Nature Physics*, 5(1):74–80, Nov 2008.

[22] Yury A. Malkov and Alexander Ponomarenko. Growing Homophilic Networks Are Natural Navigable Small Worlds. *PLOS ONE*, 11(6):e0158162, Jun 2016.

[23] OMG. *Unified Modelling Language: Superstructure.* Object Management Group, August 2011. Version 2.4, formal/11-08-05.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.

[25] Microsoft Docs. Memory Management. [Online; `https://docs.microsoft.com/en-us/windows/win32/memory/memory-management`], May 2018. Accessed on February 15, 2020.

[26] Microsoft Docs. Modules. [Online; `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/modules`], May 2017. Accessed on Jun 1, 2023.

[27] Yury Malkov. HNSWlib. [Online; `https://github.com/nmslib/hnswlib`], 2018. Accedido el 5 de junio de 2023.

[28] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[29] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP.* Morgan Kaufmann, 2001.

[30] brtholomy. Hierarchical Navigable Small World: a scalable nearest neighbor search. [Online; `https://github.com/brtholomy/hnsw`], 2023. Accedido el 5 de junio de 2023.

[31] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

Apéndice A

Horas de Trabajo

En la Figura A.1 se muestra el tiempo invertido en cada uno de los tipos de tarea. En total se ha trabajado un total de 604 horas entre 129 tareas. La Figura A.2 contiene un diagrama de Gantt que ilustra cómo se ha planificado este trabajo. Como se puede apreciar, comienza con un periodo de investigación, donde se adquieren conocimiento acerca del tema que se está tratando. Poco después, comienza una fase de diseño en la cual se estudia la arquitectura del sistema a desarrollar para después, comenzar con una fase de desarrollo donde se llevarán a cabo todos los objetivos. Esta fase se convierte en la más larga del sistema, seguida de una fase de experimentación. Finalmente, queda la elaboración de la memoria, aunque se ha utilizado el trabajo redactado durante el desarrollo de las otras fases.

Este trabajo se ha llevado a cabo como unas prácticas de empresa dentro del grupo DisCo, integrándose en las dinámicas de los miembros del grupo que trabajan en ciberseguridad, bajo la supervisión del tutor y director de este trabajo y financiado parcialmente por una beca de Trabajo de Fin de Máster del Instituto de Investigación en Ingeniería de Aragón (I3A).

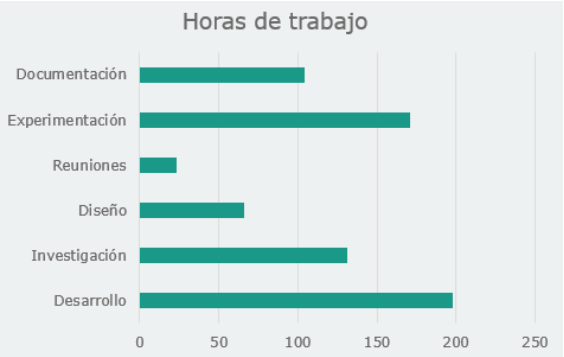

Figura A.1: Desglose de horas empleadas por tarea.

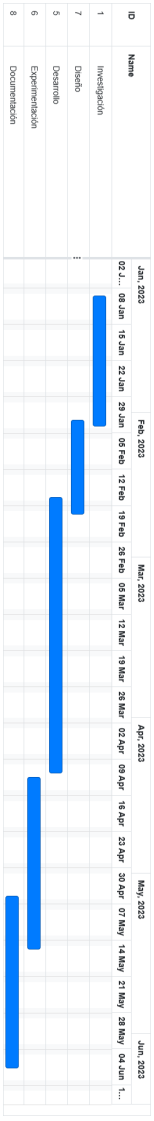

Figura A.2: Diagrama de Gantt.