UNIVERSIDAD AUTÓNOMA METROPOLITANA
UNIDAD IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍAS

Casa abierta al tiempo

ESQUEMA DE BÚSQUEDA IPv4/6 EFICIENTE EN MEMORIA USANDO CODIFICACIÓN DE VECTOR DE BITS Y PARTICIONAMIENTO DE PREFIJOS

Tesis que presenta
Fidel Ulises Sánchez Jiménez
Matricula: 2123803432

Para obtener el grado de
Doctor en Ciencias y Tecnologías de la Información

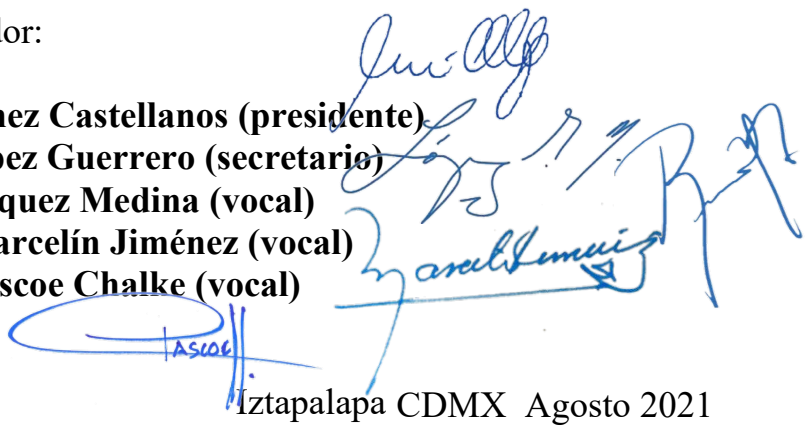
Directores:

Dr. Miguel Ángel Ruiz Sánchez

Dr. César Jalpa Villanueva

Jurado Calificador:

Dr. Javier Gómez Castellanos (presidente)
Dr. Miguel López Guerrero (secretario)
Dr. Rubén Vázquez Medina (vocal)
Dr. Ricardo Marcelín Jiménez (vocal)
Dr. Michael Pascoe Chalke (vocal)



Iztapalapa CDMX Agosto 2021

Resumen

ESQUEMA DE BÚSQUEDA IPv4/6 EFICIENTE EN MEMORIA USANDO CODIFICACIÓN DE VECTOR DE BITS Y PARTICIONAMIENTO DE PREFIJOS

por

Fidel Ulises Sánchez Jiménez

Doctor en Ciencias en Ciencias y Tecnologías de la Información

Longest Prefix Matching (LPM) address lookup speed in core routers has a significant impact on service quality in communication. The rapid growth of traffic on the Internet and the development of communications links working at speeds of several gigabits per second imply that LPM lookup is a challenge for routers on Internet. There are different LPM lookup schemes based on counts of bit values set to one of a bit-vector that encode routing information in structures of constant size that guarantee fast LPM lookups. However, these schemes have exponential memory complexity and then prefix partitioning schemes are used to save memory consumption, this by using LPM lookup in stages. This work contains two main contributions. The first one is a generalizad LPM lookup scheme based on precalculated counts of set bits in a bit-vector for which we obtain a memory cost function that is used to justify fully the choosed optimal parameter values used in the scheme. Our proposal guarantees a constant number of memory access in LPM lookup, maximum possible savings in memory, and memory independent costs of the prefix distribution in forwarding tables. The second is that, we propose a LPM lookup scheme based on a precalculated count of bit values set to one of a bit-vector where a LPM lookup has an instruction cost of two memory accesses and a prefix-partitioning scheme that optimize the available memory usage in LPM lookups in accordance with the prefix distribution in forwarding tables. The proposed prefix-partitioning is based on heuristic

search using genetic algorithms that allows a memory optimization in a short period of time. Our proposal can be implemented in general purpose hardware for IPv4/6 schemes.

El desempeño de los esquemas de búsqueda del prefijo de mayor coincidencia en dispositivos de encaminamiento globales tiene un impacto significativo en la calidad de servicio de las comunicaciones que se producen en Internet. El crecimiento acelerado del tráfico de Internet aunado al desarrollo de enlaces de comunicaciones que pueden operar con un ancho de banda de varios gigabit por segundo, implica un desafío para los esquemas de búsqueda del prefijo de mayor coincidencia. Existen diferentes esquemas de búsqueda del prefijo de mayor coincidencia que basan su idea principal en conteos precomputados de bits puestos en uno de un vector de bits que codifica la información de encaminamiento; dichos esquemas se caracterizan por tener una complejidad en memoria constante y garantizar un alto desempeño al realizar las búsquedas. Sin embargo la desventaja de esquemas clasificados en conteos precomputados es que tienen una complejidad en memoria exponencial por lo que son utilizados como algoritmos base dentro de esquemas que realizan la búsqueda en etapas; de esta manera se puede reducir drásticamente su consumo en memoria. En este trabajo se presentan dos aportaciones principales. El primer aporte consiste en una generalización de un esquema basado en el conteo precomputado de valores puestos en uno de un vector de bits, esta generalización es independiente a la distribución y número de prefijos en las tablas de encaminamiento; la razón de la generalización es para garantizar el mínimo consumo en memoria eligiendo y justificando los parámetros de operación más adecuados. La segunda aportación que se presenta en este documento consiste en un esquema general de particionaminento en etapas basado en la exploración heurística debido que se debe considerar el número y distribución de prefijos en la tabla. Cabe mencionar que en cada etapa se realiza una búsqueda utilizando cualquier esquema que utiliza conteos precomputados.

A ti Alejandra, a ti Avelina y a ti Fidel les dedico este mérito. Permíteme Dios utilizar lo aprendido para construir un mundo mejor.

Índice general

Índice de figuras

Índice de cuadros

Acknowledgments

Agradezco profundamente a

A los Doctores Miguel Ángel Ruiz Sánchez y César Jalpa Villanueva por dirigir este proyecto de investigación.

A los lectores de la tesis: Dr. Javier Gómez Castellanos, Dr. Miguel López Guerrero, Dr. Rubén Vázquez Medina, Dr. Ricardo Marcelń Jiménez y Dr. Michael Pascoe Chalke.

A los profesores del PCyTI por compartir sus conocimientos y experiencia con los alumnos de este posgrado.

A la Universidad Autónoma Metropolitana por ser mi hogar de estudios durante esta bonita experiencia.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por la beca recibida.

Capítulo 1

Introducción

El ser humano desde su origen tiene la necesidad intrínseca de comunicarse con otros seres humanos e históricamente ha tratado de lograr comunicarse rebasando sus propias capacidades biológicas; por ejemplo, la invención del teléfono permite sostener una comunicación entre dos personas separadas por una gran distancia. De esta misma manera, cuando se desarrollaron las primeras computadoras comerciales, no tardó mucho tiempo para que surgiera la necesidad de comunicarlas entre sí; de esta forma surgen las redes de computadoras y con ello, toda clase acuerdos, estándares, protocolos para lograr una comunicación eficaz.

En este capítulo nos planteamos explicar las ideas principales de los protocolos de comunicación que permiten a las computadoras comunicarse a través de Internet. Comenzaremos explicando el principio de comunicación entre computadoras conectadas a la red; específicamente se trabajará en el protocolo de comunicación TCP/IP, en donde, fijaremos especial atención a lo que se denomina el protocolo IP en el cual, es donde reside una de las funciones de interés del tema de investigación del presente documento y que es llamada reenvío de paquetes.

1.1. Redes de computadoras

Al igual que una comunicación entre seres humanos, comunicar computadoras, implica tener los componentes básicos: emisor, mensaje, medio de comunicación y receptor. Por

definición, las redes de computadoras son el conjunto de equipos conectados por medio de cables, señales, ondas o cualquier otro método de transporte de datos, que comparten información recursos y servicios [1]. Internet es un ejemplo de la red que conecta millones de computadoras y dispositivos.

El modo de funcionamiento de las redes de computadoras está compuesto por hardware y software. El hardware se divide en dos grandes grupos: los equipos terminales o también llamados *hosts* como lo son laptops, computadoras, teléfonos, servidores web y los dispositivos de red como son conmutadores, puntos de acceso y dispositivos de encaminamiento. Las aplicaciones o software de los *hosts* son las que establecen una comunicación mediante el hardware. Sin embargo, para lograr establecer una comunicación también son necesarios estándares y protocolos de comunicación que permitan que todos los dispositivos puedan pertenecer y acceder a la red de manera equitativa y ordenada.

Internet utiliza los protocolos del modelo a capas TCP/IP (Protocolo de control de transmisión y Protocolo de Internet por sus siglas en inglés). Un modelo de capas permite hacer más fácil el diseño de protocolos y ofrece ventajas como la actualización de hardware o software en las diferentes capas sin la necesidad de modificar a las demás. A continuación explicamos brevemente en que consiste la arquitectura de protocolos de este modelo ya que, en una de sus capas se encuentra situado el problema que abordamos en este proyecto de investigación, por lo tanto, es necesario contextualizar la importancia del mismo en todo el proceso que se desarrolla cuando se produce una comunicación entre computadoras.

1.2. Arquitectura de protocolos: El modelo TCP/IP

El modelo de capas TCP/IP [1] es un estándar que establece el conjunto de protocolos utilizados por Internet para la conexión de computadoras o *hosts.* Este modelo es referenciado por sus dos protocolos más importantes: TCP (Protocolo de Control de Transmisión) e IP (Protocolo de Internet). Sin embargo, dicho conjunto de protocolos de Internet se constituye por más de cien protocolos entre los que se encuentran: FTP (Protocolo de Transferencia de Archivos), HTTP (Protocolo de Transferencia de Hipertexto), POP (Protocolo de Oficina Postal). Este modelo establece cuatro capas (ver figura 1.1): Capa de acceso al medio, encargada de controlar los dispositivos y los medios que

Figura 1.1: Modelo TCP/IP.

forman la red. Capa Internet que se encarga de determinar la mejor ruta dentro de la red. Capa de transporte, encargada de admitir las comunicaciones entre distintos dispositivos y distintas redes y Capa de aplicación que se encarga de mostrar los datos al usuario, así como de controlar la codificación de los mismos.

1.3. Protocolo IP

El protocolo de Internet o también llamado IP por sus siglas en inglés *Internet Protocol* está clasificado en la capa de red del modelo TCP/IP y su objetivo fundamental consiste en envío de paquetes a través de las redes físicas mediante un modelo no orientado a conexión.

La información que se produce en una comunicación se divide en paquetes y se envía a través de la red física para ser entregada a su destino sin configuraciones adicionales

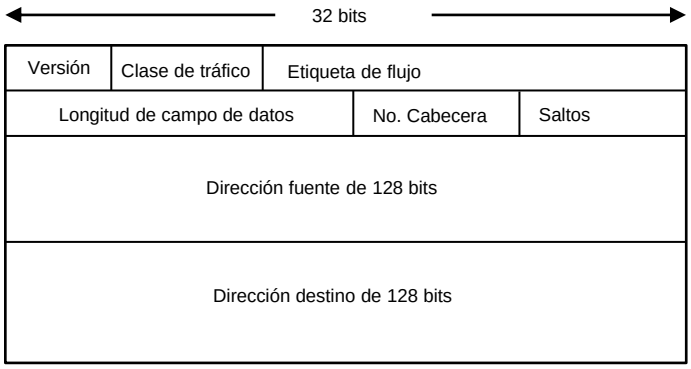


Figura 1.2: Cabecera del paquete IPv6.

más que etiquetas conocidas como direcciones IP que identifican a la fuente y al destino. Dichas direcciones son etiquetas numéricas que identifican, de manera lógica y jerárquica, a una computadora o *host* dentro de la red de Internet, la cual es un valor binario de 32 o 128 bits (IPv4 [2] e IPv6 [3], respectivamente).

En el modelo IP, el encaminamiento se realiza mediante las decisiones locales de cada nodo en la ruta del paquete. Esto significa que el paquete o datagrama que es enviado a la red no tiene garantía de llegar a su destino (no fiable). Sin embargo, el protocolo IP está diseñado para ofrecer el servicio del "*mejor esfuerzo*" lo que hace referencia a que el protocolo hará lo mejor posible para entregar el paquete a su destino. El paquete tiene un tamaño máximo conocido y una cabecera con campos que el protocolo IP necesita para poderlo enviar a su destino. En la figura 1.2 se muestra la estructura de una cabecera del datagrama de la versión 6 de IP mientras que en la figura 1.3 se muestra la estructura del datagrama de la versión 4 de IP. Se puede notar que en ambas versiones el paquete tiene las direcciones de fuente y destino (32 y 128 bits según la versión de su protocolo).

1.4. Encaminamiento

Un enrutador o dispositivo de encaminamiento es un elemento de hardware que sirve para la interconexión de redes de computadoras, que opera en la capa de red y que

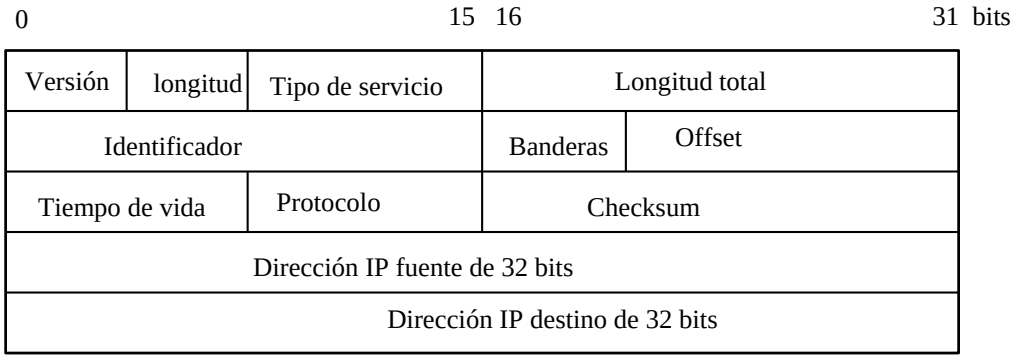


Figura 1.3: Cabecera del paquete IPv4.

permite el encaminamiento de paquetes entre redes y determina la ruta que debe tomar la información. Un enrutador tiene varias interfaces de red (tarjetas de red) y se conecta con otros enrutadores o con *hosts* terminales. Entre los procesos elementales que un dispositivo de encaminamiento realiza estan: el proceso distribuido encargado de recolectar información de las redes que el dispositivo de encaminamiento puede alcanzar y el proceso de la reexpedición de un paquete, mismo que comienza cuando llega un paquete al él y es cuando se busca el mejor enlace de salida para encaminar el paquete.

Las direcciones IP se asignan de forma jerárquica a los *hosts* de tal manera que una red pertenezca a otra red de mayor jerarquía. Los bits más significativos de una dirección se llaman prefijos de red y denotan a las redes. Un prefijo corto denota una red de mayor tamaño y un prefijo largo denota una red más pequeña. En la figura 1.4 se muestra un ejemplo de una red jerarquizada con base en los prefijos de red con direcciones IPv4; note que los dispositivos de encaminamiento deben decidir la mejor ruta de la información para que los *hosts* puedan comunicarse entre ellos, el enrutador con prefijo de 8 bits (184.0.0.0/8) puede alcanzar más redes que el enrutadores con prefijo de 24 bits (por ejemplo: 148.150.10.0/24). Si el *host* 148.15.48.17 desea establecer una comunicación con el *host* 148.200.13.7 se debe encaminar la información hasta el enrutador de mayor jerarquía. Las redes se denotan como una dirección IP seguida de la longitud del prefijo (tamaño de la red) y por supuesto, los *hosts* se denotan como direcciones IP. El número binario

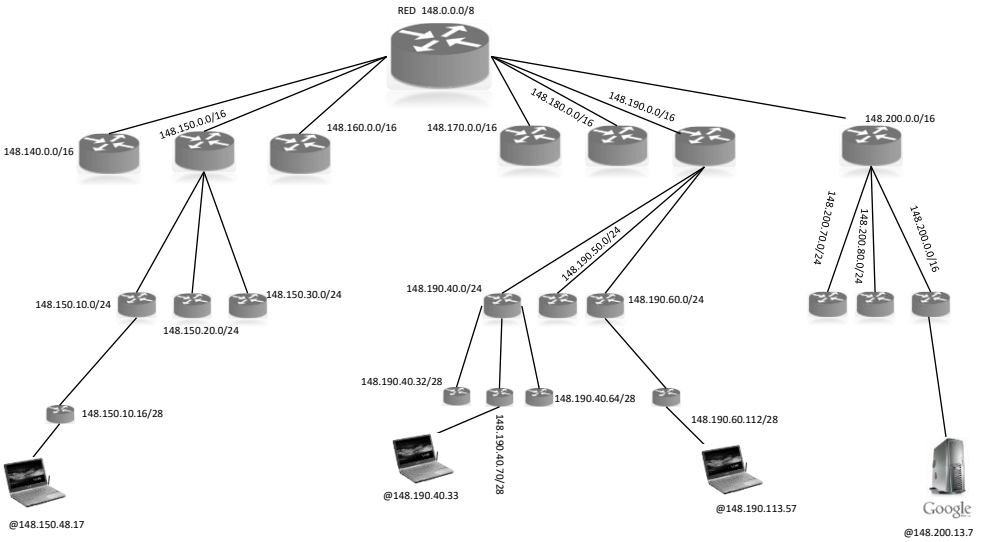


Figura 1.4: Ejemplo de la jerarquía de la red con direcciones IPv4.

que conforma las direcciones IP se divide en 4 octetos donde cada uno de ellos puede tener un rango decimal ente 0 y 255. Cuando se establece una comunicación entre dos *hosts*, los dispositivos de encaminamiento se encargan de resolver la posible ubicación de destino reenviando los paquetes producidos en la comunicación a redes de menor o mayor jerarquía. Los dispositivos de encaminamiento forman subredes hacia abajo mientras que hacia arriba son parte de una red de mayor tamaño. Esto significa que una red puede pertenecer a una red de mayor tamaño y a su vez puede contener redes de menor tamaño.

En el proceso de encaminamiento, el dispositivo de encaminamiento guarda la información de las redes que tiene al alcance mediante el uso de una estructura de datos llamada tabla de encaminamineto. La información de encaminamiento está representada por los prefijos de red ligados a los enlaces de salida del dispositivo de encaminamineto. La tabla generalmente se almacena en un enrutador en forma de un archivo de base de datos. Cuando los paquetes deben ser enviados desde un *host* a otro en la red, los dispositivos de encaminamiento utilizan dicha tabla con el fin de encontrar la mejor ruta para la transferencia de paquetes. La construcción de la tabla se realiza mediante un proceso distribuido en el cual participan los enrutadores que conforman la red para determinar el estado de

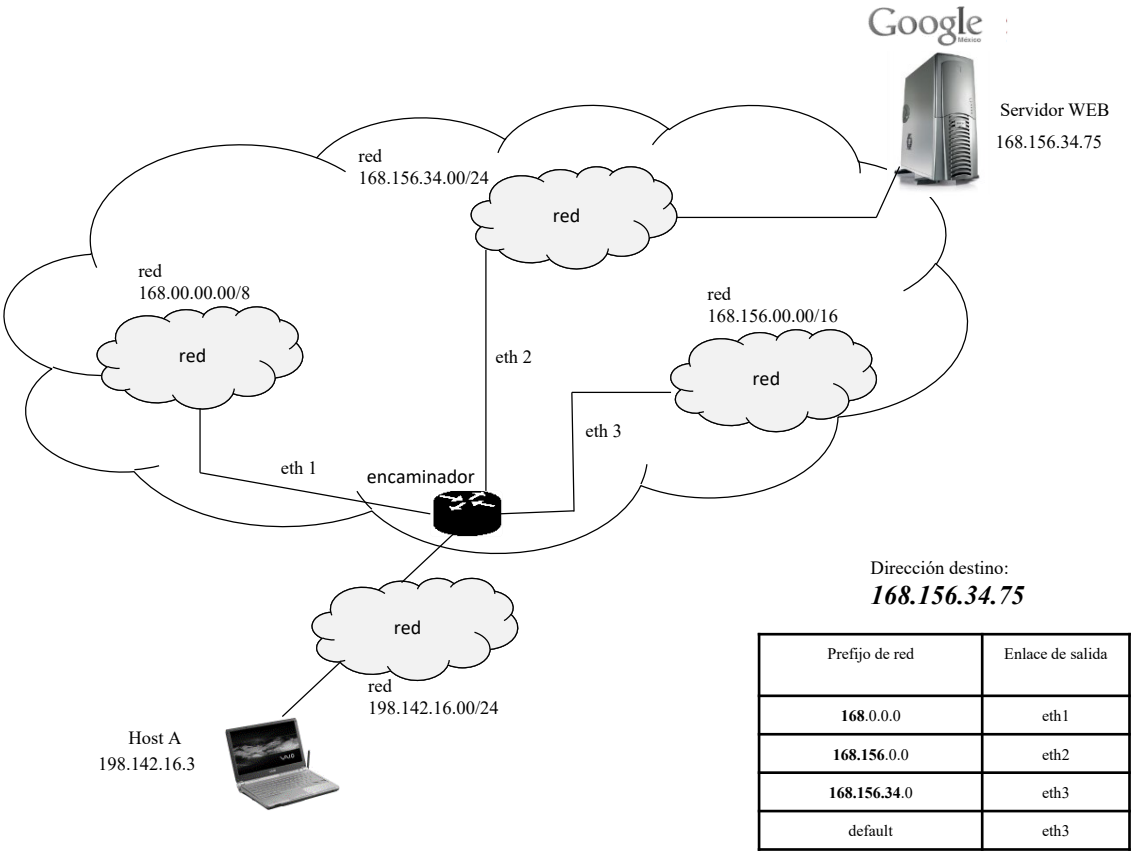
la misma y así determinar las redes alcanzables de cada dispositivo de encaminamiento. También se puede actualizar alguna entrada (prefijo) de la tabla de forma manual para informar algún cambio específico en la red.

Como ejemplo se presenta el caso en que una computadora desea hacer una petición de búsqueda en algún servidor web. En la figura 1.5 se muestra una computadora (*hos*t A) y el servidor de búsqueda (Google). Ambas entidades se encuentran en extremos de la red. Al interior de Internet se encuentran redes de menor tamaño definidas por los prefijos de red. El *host* A no tiene comunicación directa con el servidor de Google por lo que envía su petición al enrutador con que accede a la red. El enrutador debe de reexpedir el paquete por algunos de sus tres enlaces, así que hace uso de la dirección destino (168.156.34.75) y de la tabla de encaminamiento para determinar el prefijo de mayor coincidencia con la IP destino. En el ejemplo se proponen direcciones IPv4 en notación decimal y es posible visualizar que la red 168.156.34.00/24 es la que coincide en mayor número de bits con la IP destino, de esta forma, es la que puede acceder al destino.

1.5. Longest Prefix Matching

De forma específica, los dispositivos de encaminamiento tienen como finalidad reexpedir los paquetes de información que llegan a ellos para acercarlos a su destino con el uso de la tabla de encaminamiento. Dicha tabla contiene prefijos de red junto con los datos acerca del siguiente dispositivo (*Next Hop Information*, NHI) que se encuentra en la ruta correspondiente. Para realizar la labor de encaminamiento, el dispositivo busca en su tabla el prefijo que tenga el mayor número de bits coincidentes con la dirección IP de destino del paquete que está reenviando. Al proceso de búsqueda del prefijo con mayor coincidencia se le denomina búsqueda LPM (*Longest Prefix Matching*) o simplemente búsqueda IP [4]. Este proceso es determinante en la calidad de las comunicaciones entre computadoras ya que la latencia de una búsqueda LPM está estrechamente relacionada con la rapidez de reexpedición de los paquetes que arriban al dispositivo de encaminamiento.

El crecimiento de Internet representa un gran desafío para los algoritmos de búsqueda LPM, ya que no solamente hay un aumento constante del tamaño de las tablas de encaminamiento, sino que ahora se cuenta con enlaces de comunicación de hasta 100 Gbps



Prefijo de red	Enlace de salida
168.0.0.0	eth1
168.156.0.0	eth2
168.156.34.0	eth3
default	eth3

Figura 1.5: Ejemplo de la reexpedición de paquetes con direcciones IPv4.

[5]. Además, la aparición del protocolo IPv6, cuya adopción se está dando rápidamente, ha extendido el tamaño de las direcciones de 32 bits, utilizado por IPv4, a 128 bits. Los dispositivos de encaminamiento deben realizar varias decenas de millones de búsquedas LPM por cada segundo (*Million Lookups Per Second*, MLPS), para así evitar cuellos de botella en los enlaces de alta velocidad. La exigencia de velocidad requiere que los algoritmos de búsqueda LPM realicen un uso eficiente de la memoria. Por un lado, está el hecho de que al aumentar el tamaño de las direcciones y prefijos aumentará también el tamaño de las tablas de encaminamiento. Por otra parte, se tiene que, en los equipos modernos, la latencia de las operaciones aritmético-lógicas es despreciable en comparación con la latencia de los accesos a memoria [6]. Así que el factor determinante en el desempeño es la cantidad de accesos a memoria más que las operaciones aritmético-lógicas. Por otro lado, las tecnologías de las memorias son más costosas mientras menores son las latencias por lo cual la cantidad de memoria de alta velocidad disponible para los algoritmos es bastante limitada. El problema que se aborda en la presente investigación consiste en optimizar el tiempo de búsqueda del prefijo de mayor coincidencia considerando el consumo de memoria requerida por el esquema propuesto. Se presentan dos aportes que en capítulos posteriores se detallan.

1.6. Organización del documento

Este documento tiene la siguiente estructura: el capítulo 2 contiene la presentación del problema de investigación y su delimitación. En el capítulo 3 presentamos el estado del arte del problema que abordamos en términos de las investigaciones que poseen alguna similitud con nuestra propuesta. En el capítulo 4 se hace una descripción detallada de la solución y el modelo propuesto. En el capítulo 5 se muestran los resultados obtenidos después de la implementación de la posible solución descrita en el capítulo anterior. En el capítulo 6 hacemos el análisis de los resultados obtenidos en retrospectiva con los objetivos y pregunta de investigación planteados en el capítulo 2. Finalmente, en el capítulo 7, concluimos respondiendo a la pregunta tentativa de la solución planteada en la hipótesis.

Capítulo 2

Planteamiento del problema

Al término del capítulo anterior se abordó el proceso que realiza un dispositivo de encaminamiento para decidir la mejor ruta local por donde será reexpedido un paquete que tiene una dirección destino determinada. En este capítulo se describe el proceso de búsqueda LPM (*Longest Prefix Matching*) como problema central de la presente investigación y planteamos los elementos que se relacionan entre sí para dar contexto y forma a la investigación.

2.1. Planteamiento

El proceso de búsqueda LPM (*Longest Prefix Matching*) consiste específicamente en encontrar la coincidencia más larga entre los bits más significativos de una dirección IP y los prefijos dentro de la estructura conocida como tabla de encaminamiento. Las direcciones IP tienen una longitud de 32 o 128 bits (IPv4 e IPv6) mientras que los prefijos en la tabla tienen una longitud desde 0 bits hasta 32 o 128 bits según la versión del protocolo IP. Esto implica que, el número y la longitud de prefijos en una tabla de encaminamiento son un factor determinante para el desempeño de un algoritmo de búsqueda LPM; además de que, se debe considerar que las tablas de encaminamiento son regularmente actualizadas en periodos no más largos que pocos segundos. De esta manera, la calidad del servicio en las comunicaciones dentro de Internet dependerá entre otros factores de la rapidez en que se realice dicha búsqueda.

2.1.1. Objetivo especifico

El objetivo general de esta propuesta de investigación consiste en el planteamiento de un esquema de búsqueda LPM de baja latencia de ejecución que considere las versiones de direccionamiento IPv4 e IPv6. Entenderemos que la latencia de ejecución de una búsqueda LPM dependerá del tiempo de actualización de la tabla y principalmente dependerá de la cantidad de búsquedas que deba realizar un dispositivo de encaminamiento para no convertirse en un cuello de botella. Partiremos de uno de los esquemas base que conoceremos más adelante como el esquema de *Luela*, con ello, nos planteamos realizar una mejora e incluso proponer un algoritmo general de búsqueda de valores de bits dentro de un arreglo (arreglo que contenga la información de la tabla de encaminamiento). A partir de esto, nos proponemos constituir un esquema de búsqueda LPM optimizado en memoria a partir de soluciones obtenidas con exploración heurística; con ello, presentar una mejora a propuestas que utilizan algoritmos deterministas. Finalmente consideraremos que actualmente nos encontramos en un proceso de adopción de IPv6 por lo que también nos planteamos obtener, utilizar y procesar tablas IPv6 reales/sintéticas que simulen escenarios post-migración de IPv6.

2.1.2. Preguntas de investigación

¿Será posible generalizar el algoritmo de búsqueda LPM que utiliza el esquema de *Lulea* para segmentos de direcciones IP de cualquier longitud y ademas, maximizar sus cualidades? ¿Cómo podremos medir la adopción del protocolo de Internet versión 6 en la actualidad? En relación a lo anterior ¿Es posible generar un esquema que divida la búsqueda LPM en etapas, tanto en IPv4 como en IPv6, con el fin de reducir el tamaño total de memoria requerida del esquema con respecto a realizar la búsqueda en una sola etapa? En todo caso ¿Existen y se pueden encontrar los parámetros de operación de este esquema que optimicen los requerimientos de memoria? ¿Puede la exploración heurística brindar una solución eficiente en términos de calidad de la solución y tiempo para obtenerla?

2.1.3. Objetivos de la investigación

1. Estudiar las principales características de los algoritmos de búsqueda LPM de los esquemas clásicos de la literatura para determinar las ventajas y desventajas de los mismos en términos de eficiencia en requerimientos de memoria e instrucciones. Esto significa realizar el análisis para determinar el tipo de estructuras son utilizadas en la búsqueda LPM, su tamaño y por supuesto el número de accesos a ella.

2. Estudiar el esquema de búsqueda LPM para IPv4 conocido como *Lulea* [8] que se caracteriza por su alto desempeño en el uso de memoria y su baja latencia de ejecución.

3. Estudiar el estado actual de la adopción de IPv6, revisando bases de datos públicas y estudios científicos para determinar la pertinencia de un estudio de búsquedas LPM en este protocolo.

4. Obtener, procesar e interpretar tablas de encaminamiento públicas de dispositivos de encaminamiento globales para establecerlas como punto de referencia de análisis de desempeño para la propuesta que prestaremos.

5. Utilizando como referencia el esquema de *Lulea*; se propone realizar una posible generalización que establezca un algoritmo de búsqueda LPM para segmentos de direcciones IP de cualquier tamaño, con el objeto de utilizarlo en esquemas completos de búsqueda LPM tanto en IPv4 así como en IPv6.

6. Proponer un esquema general de búsqueda LPM basado en etapas tanto para IPv4 como para IPv6, tal que, en cada etapa, se optimice la búsqueda LPM del segmento de dirección IP correspondiente. Esto con el objeto de reducir el tamaño de memoria requerida por el esquema con respecto a la búsqueda en una sola etapa.

7. Utilizar técnicas de inteligencia artificial para determinar los parámetros de operación que maximicen el desempeño del esquema de búsqueda basado en etapas, tales como, el número y longitud de segmentos de direcciones IP en cada etapa.

8. Evaluar nuestro esquema en situaciones futuras donde la migración de IPv6 se haya completado, esto con ayuda tablas de encaminamiento sintéticas.

2.1.4. Justificación

La búsqueda LPM en tablas de encaminamiento IPv4/6 es un nicho de posibilidades de investigación por los problemas que aún están presentes en IPv4 y por nuevos problemas que surgen en IPv6. Las causas que actualmente consideramos tienen mayor relevancia y justifican la presente investigación son las siguientes:

Mayor ancho de banda en enlaces de comunicación. El estándar IEEE 802.3 [5] considera enlaces de 100Gb/s con lo cual un dispositivo de encaminamiento conectado a un enlace, debe realizar hasta 250 millones de búsquedas LPM en 1 segundo. Ésta es una tarea que representa un reto tanto en IPv4 como IPv6. Además, existen líneas de investigación para aumentar el ancho de banda de los enlaces de comunicaciones.

Tablas de encaminamiento con más prefijos y de mayor longitud. La forma de agregación de subredes y crecimiento de la Internet en IPv4 ha causado un aumento de hasta el doble de prefijos por año en las tablas de encaminamiento y en IPv6 se estima que el número de prefijos también presente el mismo comportamiento de crecimiento [9]. Tomando en cuenta que los prefijos en IPv6 pueden tener un longitud de hasta 128 bits, el espacio de búsqueda dentro de las tablas de encaminamiento aumenta severamente en comparación con IPv4.

Huecos en el cuerpo de conocimiento de esquemas LPM para los protocolos IPv4/6. Si bien aún existen interrogantes en las propuestas de esquemas de búsqueda LPM en IPv4, en la versión 6 del protocolo de Internet se suman interrogantes como la distribución final de los prefijos de red dentro de las tablas de encaminamiento. De esta manera, se tiene la necesidad de estudio de propuestas de esquemas de búsqueda LPM que puedan adaptarse a la gran mayoría de cambios que pueda tener la red de Internet, ofreciendo el mismo desempeño.

Tecnologías de almacenamiento. El tiempo de ejecución de una operación aritmético lógica es despreciable en comparación a la latencia de acceso a una memoria. Así que el factor determinante en el desempeño de un algoritmo de búsqueda LPM es la cantidad

de accesos a memoria más que, las operaciones aritmético-lógicas. Las tecnologías de las memorias son más costosas mientras menores son las latencias, por lo cual, la cantidad de memoria de alta velocidad disponible para los algoritmos es muy limitada.

2.1.5. Viabilidad

La presente investigación es factible de realizar ya que es posible responder afirmativa o negativamente la pregunta de investigación de acuerdo con los elementos técnicos descritos a continuación que rodean a la problemática.

1. Es posible evaluar el desempeño del esquema de *Lulea* y estudiar los principios básicos de su funcionamiento.

2. Es viable hacer un análisis de los estudios acerca de la migración al protocolo IPv6 y es posible obtener información de ruteo de bases públicas globales.

3. Es posible realizar exploración heúristica para resolver problemas combinatorios, por lo que podemos hacer una evaluación de desempeño en el contexto de la problemática de la búsqueda del prefijo de mayor coincidencia.

2.2. Delimitación del tema

Se propondrá un esquema de búsqueda LPM para las versiones 4 y 6 del protocolo de Internet. El esquema permitirá ofrecer una solución de búsqueda LPM para la reexpedición de paquetes en tablas de encaminamiento actuales y también sintéticas en el caso de IPv6 (generadas pseudo-aleatoriamente para emular Internet después de la migración a dicho protocolo), tal que, su eficacia radique en la optimización en memoria cuidando se tenga una baja latencia de ejecución. Se utilizará como base el algoritmo de búsqueda LPM propuesto por *Lulea*, con el cual, pretendemos presentar una generalización de búsqueda LPM para segmentos de direcciones IP de cualquier tamaño, esto con el objeto de proponer un esquema de búsqueda LPM en etapas generalizado para IPv4/6 que también este basado en la optimización en memoria. Se hará uso de técnicas de exploración heurística

para poder determinar la división apropiada en etapas la búsqueda con latencias acordes con la actualización de las tablas de encaminamiento.

2.3. Hipótesis general

Es posible proponer un esquema de búsqueda LPM para las versiones 4 y 6 del protocolo de Internet de baja latencia de ejecución, mediante la optimización de memoria utilizada por las estructuras de datos que ocupa dicho esquema, de tal forma que dichas estructuras puedan ser contenidas en memorias de rápido acceso. El esquema tendrá dos aportaciones innovadoras; la primera consistirá en la propuesta de un algoritmo de búsqueda LPM dividido en etapas dinámicas basadas en la distribución de prefijos de red, mientras que, la segunda aportación será la generalización de búsquedas LPM en segmentos de direcciones de cualquier tamaño para las etapas del algoritmo antes enunciado; la base de dicha generalización será el esquema de *Lulea*. Los parámetros de operación del algoritmo de búsqueda LPM serán establecidos mediante exploración heurística con el fin de obtener la máxima optimización en requerimientos de memoria, al mismo tiempo que, la ejecución del algoritmo de exploración heurística sea lo suficientemente eficiente como para realizarse antes que la información de la tabla de encaminamiento sea actualizada.

Capítulo 3

Estado del arte

En este capítulo presentamos el estado del arte de las propuestas y estudios que motivan la presente investigación y que son referencia base de la importancia de nuestra contribución. Abordamos tres ejes transversales que nos permiten ubicar la investigación en un contexto de desarrollo actual con respecto a otras propuestas relacionadas al problema de búsqueda LPM. En un primer eje, presentamos el protocolo IP versión 6, su propósito, migración y topología inicial de la red que utiliza este protocolo. Como segundo eje situamos nuestra atención en el desarrollo actual de memorias de almacenamiento de baja latencia de acceso, esto porque, como mencionamos en el primer capítulo, la latencia de accesos y capacidad de almacenamiento influyen directamente en el rendimiento de un algoritmo de búsqueda LPM. Finalmente, como tercer eje presentamos los trabajos relacionados con el problema de búsqueda LPM.

3.1. IPv6 en la actualidad

Las direcciones IP en la versión 4 son etiquetas binarias de 32 bits de longitud, esto implica que, este protocolo posibilita alrededor de 4 mil millones de *hosts* conectados al mismo tiempo. Sin embargo, como se muestra en la figura 3.1, el número de usuarios y dispositivos que se han incorporado a la red desde el año 2003 hasta la actualidad superan el límite de dispositivos que posibilita IPv4 desde el año 2011, esto según las estadísticas de Google [7]. De la figura también se observa que desde 2016 tanto el número de dispositivos

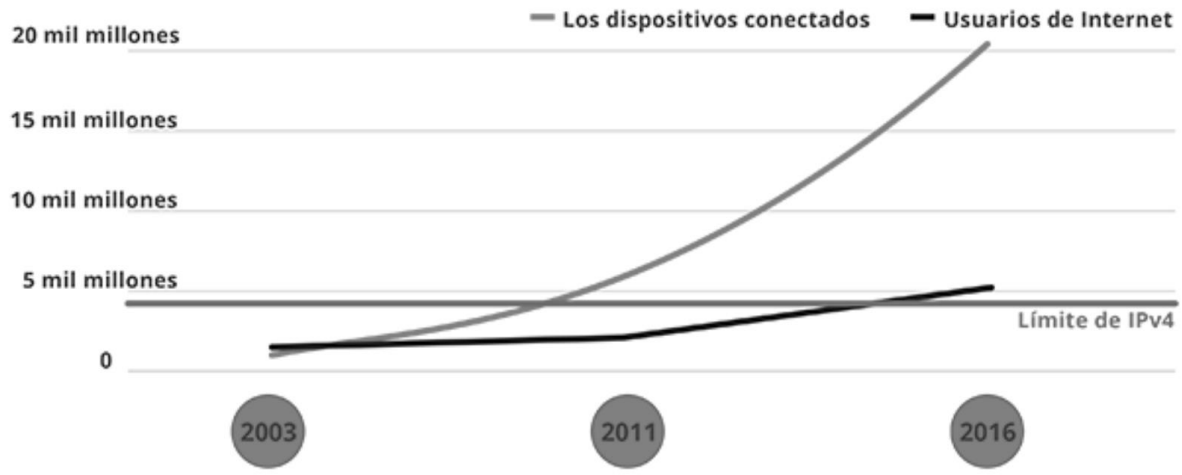


Figura 3.1: Dispositivos conectados y usuarios de Internet con IPv4.

como de usuarios ya ha superado el límite de IPv4.

El protocolo IPv6 definido en el RFC 2640 establece que direcciones IP «con una longitud de 128 bits» resuelve el problema de crecimiento de la red de Internet. Desde su origen en 1998 se ha hablado de una migración para adoptar a IPv6 y desechar a IPv4. Dicha migración consiste en que, los proveedores de servicios Internet o ISPs (por sus siglas en inglés *Internet Service Provide*r) y sus usuarios utilicen la tecnología tanto de software como de hardware que posibilite el uso de IPv6 sin problemas de conexión. En la migración de IPv4 a IPv6 han habido fechas importantes como el Día Mundial de IPv6 (en inglés: *World IPv6 Day*) que fue una prueba técnica y un evento publicitario que tuvo lugar en 2011 para promover el despliegue público de IPv6. Fue patrocinado y organizado por la *Internet Society* y varios proveedores de contenidos. Tras el éxito del Día Mundial de IPv6, la *Internet Society* se llevó a cabo el Lanzamiento Mundial de IPv6 [10] (en inglés: *World IPv6 Launch*) el 6 de junio de 2012. Esta vez el objetivo no era realizar pruebas, sino que los participantes desplegaran IPv6 de forma permanente en sus productos y servicios. Como resultado ISPs globales, así como los principales proveedores de servicios web permanecen prestando y mejorando su calidad de servicio con IPv6.

Desde los años 90 del siglo pasado las computadoras de propósito general tienen soporte

para el uso de IPv6 y los proveedores de software ofrecen actualizaciones constantes para su conectividad, lo que implica una migración avanzada por parte de los usuarios, en cambio, para medir la adopción de IPv6 de prestadores de servicios web podemos utilizar el *Ranking Alexa* que es una subsidiaria de la compañía *Amazon.com*. Es conocida por operar el sitio *webalexa.com* que provee información acerca de la cantidad de visitas que recibe un sitio web y los clasifica en un *ranking*. *Alexa* recoge información de los usuarios que tienen instalado *Alexa Toolbar*, lo cual le permite generar estadísticas acerca de la cantidad de visitas y de los enlaces relacionados. El cuadro 3.1 muestra los diez sitios con mayor número de accesos a nivel global. Los diez primeros sitios en el *ranking* reciben más del 99 % de visitas a nivel mundial de los cuales *Google* ocupa el primer lugar con más 90 % de visitas. Todos estos sitios antes mencionados tienen activado de forma permanente en sus servicios a IPv6, con ello podemos decir que, los usuarios de la red, en el esquema cliente servidor, están en una etapa avanzada de la migración a la red IPv6. Nosotros pudimos validar la disponibilidad de los sitios web ofrecen servicios IPv6 de la Tabla 3.1 utilizando túneles *teredo,* puesto que hasta la fecha en México, los ISP a los que tenemos acceso solo permiten el direccionamiento de IPv4.

La figura 3.2 muestra la disponibilidad de la conectividad de IPv6 entre usuarios de *Google* [12]. El gráfico muestra el porcentaje de usuarios que acceden a *Google* mediante la red IPv6 ya sea de forma nativa (el ISP local del usuario le proporciona una dirección IPv6) o mediante túneles *Teredo* [11] (se utiliza la red IPv4 como acceso a la red IPv6). Las estadísticas de la figura muestran que la tendencia de visitas crece de forma exponencial año con año.

La migración de IPv4 a IPv6 también puede ser medida con el número y cobertura de proveedores de Internet que ofrecen conectividad a la red utilizando este protocolo. Los ISPs globales (AFRINIC, APNIC, ARIN, LACNIC y RIPE) reconocen a *Google* [12] como uno de los mejores aportadores de estadísticas en el estado de la migración a IPv6. La prueba de conectividad a la red IPv6 es sencilla: acceder a *google.ipv6*. Por ejemplo en la actualidad Estados Unidos de América tiene un porcentaje de adopción de 45.22 % de IPv6. En otras palabras el 45.2 % de usuarios de *Google* en este país pueden conectarse a sus servicios IPv6, lo que implica que sus proveedores de servicios de Internet locales son capaces de brindarles una dirección IPv6 a sus usuarios, en contraste con México que

Cuadro 3.1: Ranking Alexa.

Ranking Alexa	Sitio Web	Protocolos en uso
1	Google.com	IPv4/6
2	Youtube.com	IPv4/6
3	Tmall.com	IPv4/6
4	Baidu.com	IPv4/6
5	Qq.com	IPv4/6
6	Suhu.com	IPv4/6
7	Facebook.org	IPv4/6
8	Taobao.com	IPv4/6
9	360.com	IPv4/6
10	Twitter.com	IPv4/6

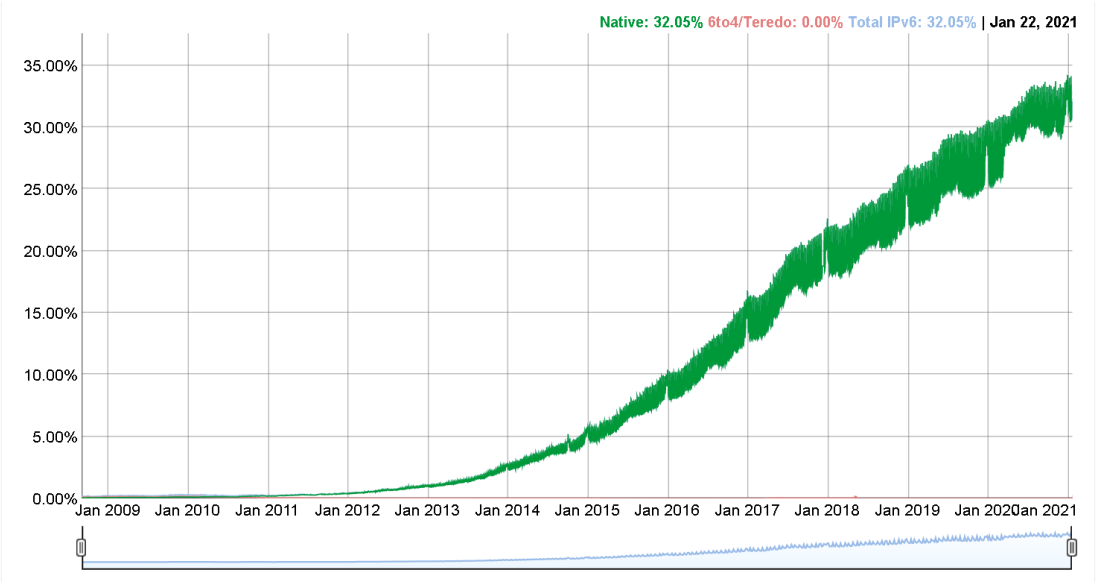


Figura 3.2: Usuarios IPv6 de *Google*.

tiene un avance 39.38 % de usuarios de Google puede conectarse a sus servicios IPv6. Sin embargo, países desarrollados como Rusia tienen hasta hoy un 8.37 % de adopción. Las estadísticas muestran que año con año estos porcentajes se duplican, lo que significa que siguiendo esta tendencia en muy pocos años se completará la migración a IPv6.

Podemos asumir que la adopción del protocolo de comunicaciones IPv6 visto desde el cliente tiene un crecimiento exponencial, mientras que, proveedores de servicios mayormente utilizados tienen habilitados de forma permanente sus servicios a este nuevo protocolo. Existen diferentes estudios de la adopción, migración y expectativas de IPv6 como los vistos en [49] donde se habla de los mecanismos de transición a este; o bien, estudios que hacen especificaciones de su uso como los vistos en [50]. Sin embargo, los ISP globales recomiendan usar las medidas en tiempo real de proveedores de servicios mundialmente reconocidos.

3.1.1. Tablas de encaminamiento IPv6

El RFC 2373 propuesto en el año de 1998 para IPv6, establece prefijos *unicast* de 128 bits con un prefijo común "001", esto implica que un octavo de todas las direcciones en IPv6 tienen el mismo prefijo (puesto que quedan 125 bits para generar subredes). El RFC 3513 en 2003 substituyó y dejó obsoleto al RFC 2373. Uno de los cambios más importantes en este RFC es darle todo el espacio de direccionamiento a los usuarios (no hay ningún prefijo en común) asignando solo algunas direcciones para otros usos específicos de la red.

IANA por sus siglas en inglés *Internet Assigned Numbers Authority* es la institución que entre otras actividades coordina a nivel global los sistemas de direcciones del Protocolo de Internet, asigna bloques de direcciones directamente a ISP globales de cobertura continental (AFRINIC, APNIC, ARIN, LACNIC y RIPE) para que ellos a su vez repartan direcciones IP a proveedores de servicios de menor jerarquía. Desde la definición del protocolo de Internet versión 6, IANA asignó bloques de direcciones a los ISP globales [1] dando a todos ellos el prefijo de "0010". Para cuando se desechó la idea del prefijo común "001" para direcciones *unicast*, una gran cantidad de bloques de direcciones ya habían

[1] http://www.iana.org/assignments/ipv6-unicast-address-assignments/ipv6-unicast-address-assignments.xhtml

sido asignados con el prefijo común 0010. El tamaño de los bloques de direcciones IPv6 asignados son tan grandes que muchas propuestas de búsqueda LPM utilizan la idea de un prefijo común con el fin de reducir el tiempo de ejecución de una búsqueda, sin embargo, experimentalmente nosotros encontramos que en la actualidad existen direcciones y prefijos sin el prefijo común 001, por lo que cualquier optimización con base en prefijos comunes debe ser desechada.

La información de encaminamiento utilizada por un enrutador para la reexpedición de paquetes está contenida en su tabla de prefijos de red, dicha tabla sufre modificaciones cada determinado tiempo para actualizar el estado de la red. En la presente investigación es necesario el uso de tablas de encaminamiento globales para demostrar la eficacia de nuestra propuesta. La universidad de Oregon tiene un proyecto llamado *University of Oregon Route Views Project* en el cual, parte del proyecto es la colección de una base de datos de libre acceso de la tabla de encaminamiento IPv6 del enrutador AS6447 [13] desde su implementación hasta ahora. Dicha base de datos hace copias integras de la tabla de encaminamiento en intervalos de 15 minutos, junto con todas las actualizaciones que se le realizaron en dicho periodo de tiempo. Es importante tomar en cuenta que la precisión en el registro de ocurrencias de una actualización es de un segundo. La propuesta de investigación por lo tanto, considerará un segundo como el tiempo mínimo entre actualizaciones; hay otras métricas que de igual forma toman como la precisión entre actualizaciones de un segundo [14].

3.2. Tecnologías de almacenamiento

Una búsqueda LPM exige una rápida ejecución que se depende directamente de las latencias de acceso a las memorias que guardan las estructuras de datos que se utilizan en dichas búsquedas. En la actualidad la capacidad de almacenamiento ha dejado de ser un problema en retrospectiva de tan solo unos años atrás. Sin embargo, la latencia de acceso a una memoria puede ser una gran problemática. Mientras menor es la latencia de una memoria, mayor es el costo en la elaboración de la misma. Esto implica que en la actualidad el tamaño de las memorias de rápido acceso tan solo están en el orden de los megabytes. Existen tecnologías de memorias de acceso aleatorio como DDR5 [15] que

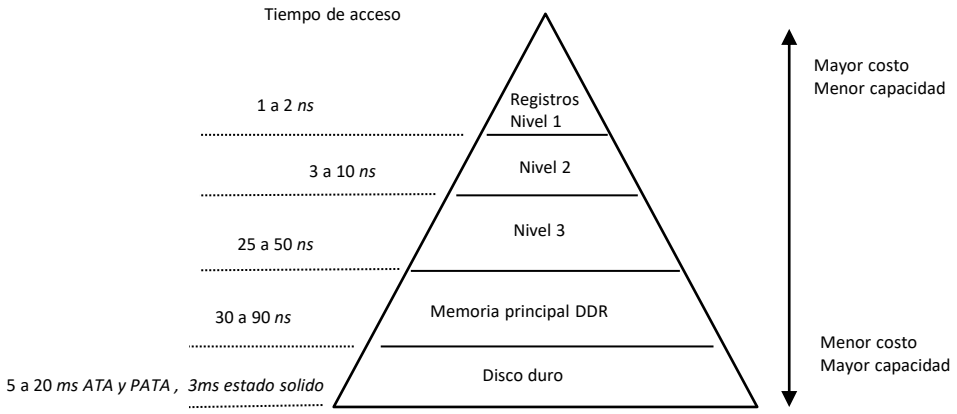


Figura 3.3: Latencias de acceso a memorias. «John Hennessy – David Patterson Arquitectura de Computadores – Un enfoque cuantitativo (1a edición, capítulo 8)».

tiene una frecuencia de reloj muy grande en comparación con tecnologías predecesoras, además de un ancho de bus de 128 bits y hasta 16 canales paralelos. Esta tecnología es un gran avance para la rapidez del procesamiento de la información. Sin embargo, la latencia de acceso a sus circuitos sigue estando muy por encima en comparación de las memoras de rápido acceso *cache*. La figura 3.3 muestra los rangos de latencias y capacidades de memorias actuales reportados en [16]. Usualmente los tres primeros niveles se encuentran dentro del procesador. A la fecha en los procesadores comerciales las memorias de primer nivel tienen una capacidad de no más de 48KB por núcleo de procesamiento.

Existe un constante desarrollo en tecnologías de almacenamiento. Sin embargo, las latencias de acceso a la información siempre están sujetas por la jerarquía de la figura 3.3. Al menos en la actualidad no se puede presentar un esquema de búsqueda LPM que se desvincule de la capacidad de almacenamiento de las memorias y las latencias de acceso que la caracteriza. De esta manera, la propuesta presentada considera dichas limitaciones.

3.3. Trabajo relacionado

A continuación, presentamos las investigaciones que son modelos de base para nuestro esquema y que tienen alguna similitud con nuestra propuesta. Inicialmente explicamos una estructura de datos nombrada *trie* binario y que sirve para representar prefijos de

red en las tablas de encaminamiento mediante una estructura de árbol. Exponemos en qué consiste el *trie* binario porque por medio de él es más sencillo comprender la idea principal de nuestra propuesta así como los fundamentos de los trabajos relacionados con ella.

Presentamos principalmente como referente al esquema que se propone en [8] conocido como *Lulea*, destacando su idea central de búsqueda LPM así como sus principales cualidades. En seguida también presentamos esquemas clásicos de búsqueda que poseen similitudes con este esquema. Finalmente explicamos una solución para realizar la búsqueda LPM en etapas, propuesto por [18] que utiliza el método de exploración exhaustiva de todas las posibles formas de segmentar una dirección IP y realizar búsquedas LPM en cada segmento, esto para encontrar la combinación con menor requerimiento de memoria.

3.3.1. *Trie* binario

Una estructura para representar los prefijos de red contenidos en una tabla de encaminamiento (e incluso hacer búsquedas LPM) es el *trie* binario [30]. Los nodos se asocian con los prefijos que llevan hasta ellos recorriendo el *trie* desde la raíz considerando, uno tras otro, el valor de los bits que forman al prefijo: si el bit es un cero, se desciende al hijo de la izquierda y si es un uno, hacia el de la derecha. De alguna manera se debe identificar a los nodos que representan a los prefijos de la tabla de encaminamiento de los que no y que solamente están en el trayecto hacia un nodo que sí representa algún prefijo.

Diversos esquemas de búsqueda LPM utilizan la estructura de *trie* o variaciones de él, al menos para explicar las ideas en las que fundamentan sus principios base de búsqueda LPM [31][32][33] [34]. Una de las variaciones que es más recurrente al representar prefijos de red en el *trie* binario es, segmentar el *trie* en niveles para realizar búsquedas LPM en etapas en cada uno de sus niveles. La partición del *trie* facilita la reducción de la memoria necesaria y permite por consiguiente mejorar el desempeño al realizar búsquedas. La figura 3.4 (a) ilustra, como ejemplo, un fragmento de una tabla de encaminamiento con prefijos cuya máxima longitud es de 4 bits. En la figura 3.4 (b) se puede observar el *trie* binario correspondiente a dicho fragmento, también observe que S1 y S5 son los *subtries* donde se encuentran los prefijos de longitud mayor a 4 bits. En general los prefijos de red no son

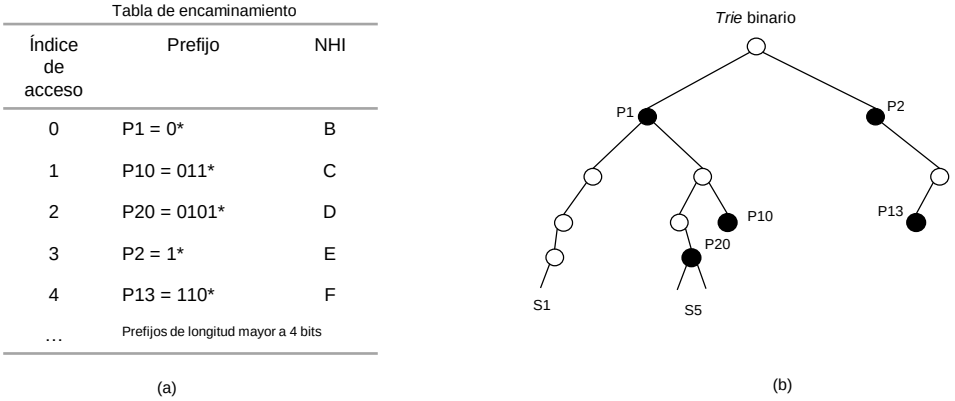


Tabla de encaminamiento

Índice de acceso	Prefijo	NHI
0	P1 = 0*	B
1	P10 = 011*	C
2	P20 = 0101*	D
3	P2 = 1*	E
4	P13 = 110*	F
…	Prefijos de longitud mayor a 4 bits	

(a)

Figura 3.4: (a) Tabla de encaminamiento. (b) Prefijos en el *trie* binario

disjuntos, esto significa que un prefijo puede ser prefijo de otro, tal como se muestra en el ejemplo de la figura 3.4 (b), donde el prefijo P2 está contenido en el prefijo P13. En un *trie* binario todos los descendientes de un nodo tienen como prefijo común al prefijo asociado con dicho nodo, esto puede observarse en la figura 3.4 (b). Si en un recorrido del *trie,* para determinar el LPM, el último nodo visitado representa un prefijo, éste será el resultado de la búsqueda; pero si el último nodo visitado no representa un prefijo, el resultado de la búsqueda será el prefijo visitado más recientemente antes de terminar el recorrido. Por lo tanto, se debe mantener un registro del prefijo más reciente por el que se ha pasado. Cabe mencionar que, en el ejemplo de la figura, los prefijos representados en el *trie* tienen una máxima longitud de 4 bits y que esto implica que, el segmento correspondiente de la dirección IP destino del paquete que será reexpedido tiene una longitud de 4 bits.

3.3.2. Búsqueda LPM en el *trie* binario

La búsqueda LPM o IP es un tema de interés en la actualidad tanto para IPv4 como para IPv6. El estudio presentado en [35] hace una inspección detallada de los mecanismos de búsqueda LPM más representativos hasta la fecha de su publicación (2012). En dicho trabajo se puede observar que, la búsqueda en la que se utiliza el *trie* binario es citada en la mayoría de las propuestas teniendo en cuenta que, la complejidad en memoria o bien en instrucciones es proporcional a la longitud los prefijos de red y a la distribución de los

mismos. Sin embargo, si el *trie* se codifica o se comprime, ello puede representar disminuir la complejidad de la búsqueda LPM inclusive hasta en una instrucción.

La propuesta de árbol de búsqueda heredado múltiple O también nombrado MIST por sus siglas en inglés [36]; es un esquema de búsqueda LPM únicamente para IPv4 donde se considera la longitud del prefijo para realizar una partición en etapas donde en cada una de ellas se realiza una búsqueda en un *trie* binario. Este trabajo presenta una mejora al esquema clásico a segmentar la búsqueda LPM de acuerdo a múltiples arboles segmentados en diferentes profundidades.

Optimización de acuerdo al longitud y densidad del prefijo Existen propuestas como las descritas en [37] y en [38], que de acuerdo a la longitud y densidad (distribución) de prefijos en la tabla se pueden agrupar diferentes *tries* de búsqueda donde, en cada uno de ellos se puede eliminar la redundancia de prefijos (un prefijo dentro de otro prefijo). Esto permite por un lado realizar búsqueda en IPv6 y al mismo tiempo tener una ganancia significativa en su complejidad de instrucciones.

Árbol de búsqueda equilibrada En esta propuesta se intenta disminuir la cantidad de instrucciones de actualización del *trie* binario [39]. La optimización que se presenta funciona principalmente en IPv6 y existe un equilibrio de complejidad en instrucciones de búsqueda LPM dentro del *trie* binario y la actualización del mismo.

***Trie* codificado en SDRAM** Esta propuesta tiene la peculiaridad de funcionar en una memoria de acceso aleatorio SDRAM y mapear en ella el *trie* binario [40]. La búsqueda LPM es segmentada en etapas en cada una de ellas y los *tries* correspondientes son codificados. En este trabajo operan tres ideas básicas: uso de SDARAM, codificación del *trie* y finalmente la segmentación en etapas.

***Trie* y tablas Hash sobrepuestas** Esta propuesta contiene la innovación de sobreponer dos tipos de estructuras: el *trie* binario y una tabla de búsqueda *hash* [51]. Esta optimización se logra construyendo arboles multi bit (un nodo represa a varios *subtries*),

logrando así, una ganancia significativa comparada la búsqueda mediante recorridos del *trie* binario.

***Trie* de decisión multi paso** De la figura 3.4 puede notar que un prefijo puede estar contenido dentro de otro prefijo; esto implica que la búsqueda debe continuar, cuando se encuentra un prefijo y esté aún tiene nodos desentiendes. En la propuesta [52] se realiza una serie de búsquedas paralelas sobre cada recorrido para considerar todos los posibles prefijos y de esta manera compararlos con la dirección IP obteniendo el de mayor coincidencia.

Trie* utilizando filtros *Bloom Al igual que la propuesta presentada en [52] se considera que un prefijo puede estar dentro de otro prefijo. Sin embargo, en la propuesta realizada por [53] se utilizan filtros *bloom* para discriminar cual de todos los prefijos en un recorrido del *trie* binario. Un filtro *bloom* dependiendo de sus parámetros puede arrojar un falso positivo como resultado de la coincidencia más larga y su precisión está determinada por la complejidad en instrucciones del filtro.

3.3.3. Búsqueda LPM con memoria TCAM

La tecnología TCAM consiste en memorias de contenido direccionable que pueden encontrase en ternas. Estas memorias pueden simular en hardware el *trie* binario y se puede aprovechar la baja latencia de estas para mejorar el desempeño del algoritmo de búsqueda; siendo su principal desventaja el consumo en energía de operación.

La propuesta presentada en [42] consiste codificar el *trie* binario para después simularlo en la memoria TCAM y así, acceder con mayor eficiencia a su contenido reduciendo el número de accesos a memoria y por consiguiente a su consumo en energía.

Otra manera de usar las TCAM para búsquedas LPM se presenta en [43], en donde, se clasifican a los prefijos de las mismas longitudes y se hace una búsqueda por clase. La principal desventaja de esta propuesta es que las longitudes de prefijos en tablas IPv4/6 no están distribuidas uniformemente con lo cual hay pocas clases de longitudes de prefijos.

3.3.4. Algoritmos basados en conteos de bits

Las búsquedas del prefijo de mayor coincidencia en el *trie* binario se realizan mediante recorridos. El recorrido se efectúa utilizando el valor de los bits de la dirección IP de destino del paquete. El esquema de representación y búsqueda de prefijos mediante el *trie* binario es bastante sencillo. Sin embargo, tiene como principal desventaja que su desempeño en tiempo de ejecución es proporcional a la profundidad del *trie*, es decir, a la longitud de los prefijos. Además de que, la memoria necesaria es proporcional al número de prefijos en la tabla.

Existen propuestas que utilizan el *trie* binario como un mecanismo para generar un arreglo o arreglos de bits que contienen codificada la información de encaminamiento. Una vez construido dicho arreglo u arreglos, las búsquedas LPM se efectúan mediante el acceso a los contenidos de ellos, donde en el mejor de los casos el resultado de la búsqueda se puede completar en un solo acceso memoria. A continuación explicamos la idea principal de tres esquemas base de codificación del *trie* binario que son referentes para propuestas actuales; seguido de ellos, presentamos las ideas de propuestas actuales que son adecuaciones y/o mejoras de los algoritmos base.

3.3.4.1. Algoritmo de la universidad de *Lulea*

El algoritmo considera hasta tres etapas de búsqueda en IPv4 [8], de tal suerte que, en la primera se representa la información de encaminamiento en las ramas de un árbol binario completo de profundidad 16, cualquier nodo tiene ambos hijos: izquierdo y derecho, o bien, apuntadores a la siguiente etapa en caso de que fuera una hoja. Un recorrido desde la raíz hasta una hoja define a un prefijo de red (no hay prefijos en nodos intermedios). Cuando se modifica al árbol para que sea completo, se tiene que modificar la tabla de encaminamiento y en este caso debe estar ordenada de acuerdo con los prefijos que se encuentran de izquierda a derecha del árbol.

La principal contribución de este algoritmo se encuentra en la primera etapa de búsqueda, donde, los autores justifican una profundidad de 16 en el árbol debido a que, al momento de elaborar su propuesta más del 90 % de prefijos tenían una longitud máxima de 16 bits. Del árbol se obtiene un vector de bits de 2^{16} entradas que contiene la infor-

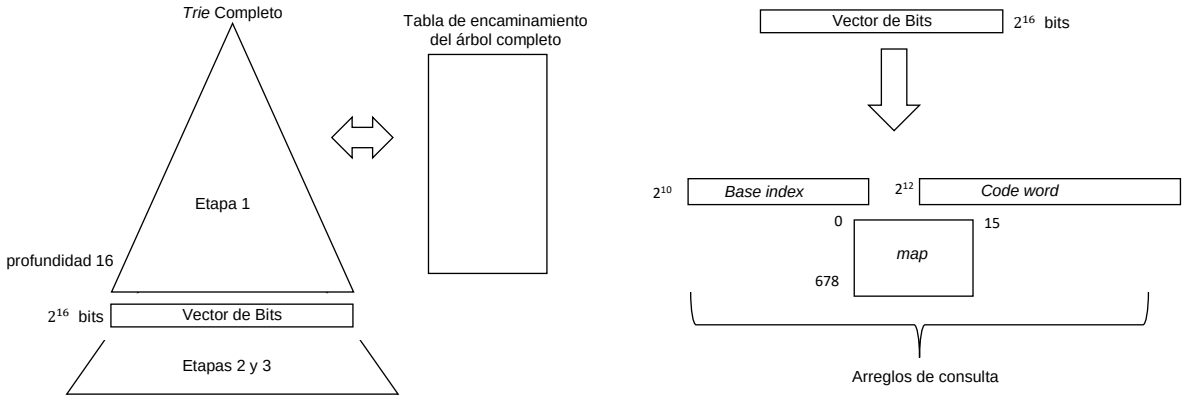


Figura 3.5: Esquema de *Lulea*

mación de ruteo en un arreglo plano. Finalmente del vector de bits se construyen tres arreglos de consulta: *base index*, *code word* y *map*. La figura 3.5 muestra el proceso de construcción del esquema de *Lulea*.

Para obtener el prefijo más largo se utiliza una dirección IP como índice de los tres arreglos construidos a partir del vector de bits. Primero se obtienen los 16 bits más significativos de la dirección IP para realizar la búsqueda en la primera etapa; luego, desde los bits más a menos significativos restantes se obtienen segmentos de 10, 12 y 16 bits que son usados como llaves en los arreglos. La adición del contenido dichos arreglos es a su vez, el índice del prefijo de mayor longitud en la tabla de encaminamiento. El proceso de búsqueda se puede observar en la figura 3.6.

Entre las principales ventajas de esta propuesta destaca el hecho de que en la primera etapa el número de accesos a memoria es constante y no depende del tamaño de la tabla de encaminamiento ni de la distribución de prefijos de red. También está el hecho de que el tamaño de los tres arreglos que codifican la información de encaminamiento es de tan solo unos pocos kilobytes y que además, la estructura de árbol binario completo desaparece una vez realizada la codificación. Lo anterior implica que la búsqueda se realiza en memorias de rápido acceso y con accesos constantes, haciendo esta propuesta una de las principales referencias para esquemas actuales.

Entre sus desventajas encontramos que este esquema fue creado para IPv4 (direcciones

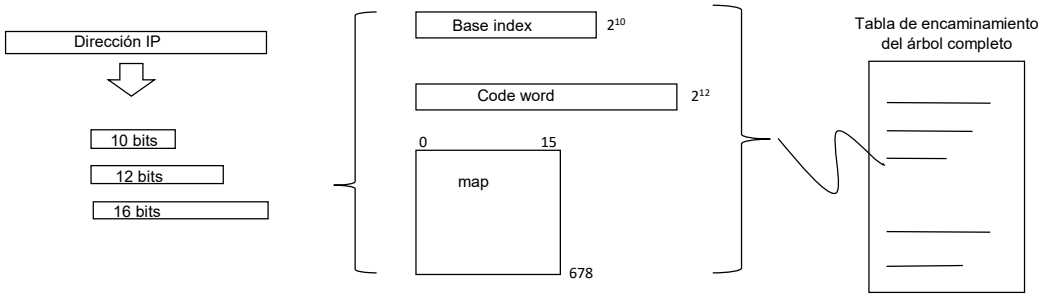


Figura 3.6: Búsqueda LPM en la primera etapa del esquema de *lulea*

de 32 bits) y obtiene prefijos de a lo más 16 bits. Para prefijos de longitud mayor a 16 bits se puede aplicar el algoritmo hasta dos veces más para completar dicha búsqueda (nivel dos y tres de búsqueda), no se se puede escalar directamente a IPV6 y sus parámetros como la profundidad del árbol y las propiedades de sus tres arreglos del primer nivel no están justificados en términos de desempeño o tamaño optimizado.

Para el 2021 con más de 2000 descargas y más de 300 citas solo en la revista donde fue publicado; este estudio sigue siendo una referencia base siendo citado en todas las propuestas que codifican y/o comprimen el *trie* binario en vectores de bits.

3.3.4.2. Treebitmap

Este esquema al igual que el de *lulea*, utiliza el *trie* binario como estructura auxiliar para construir vectores de bits de consulta que han de servir para encontrar el prefijo de mayor coincidencia[17]. En esta propuesta se realiza una búsqueda completa en n etapas, en donde cada etapa está constituida por él, o los, *tries* correspondientes, aunado a apuntadores a cada *trie* de cada etapa. A diferencia del esquema de *lulea*, el *treebitmap* no modifica la estructura del *trie*, lo que implica que los prefijos pueden estar en nodos intermedios o bien que un nodo no tenga ambos hijos; sin embargo, la tabla de encaminamiento está ordenada de acuerdo con el orden de los prefijos en el *trie* por profundidad en el mismo. Existen dos arreglos de consulta el primero que codifica el *trie* binario y el

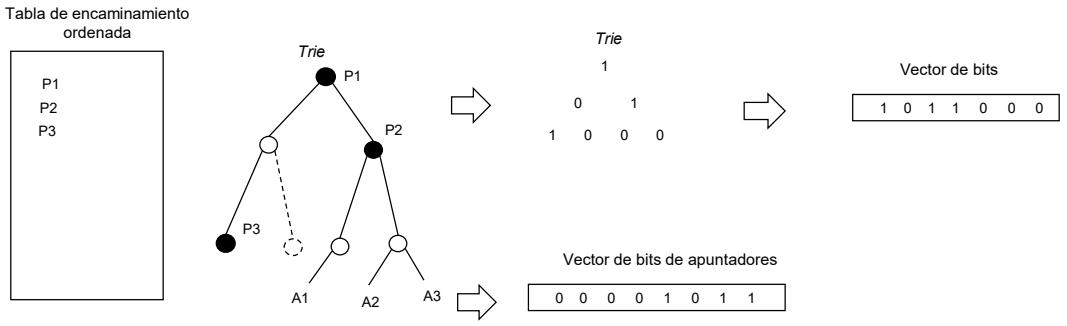


Figura 3.7: Esquema del Treebitmap

segundo que codifica los apuntadores a la siguiente etapa si es que hubiera alguna; después de ello, la búsqueda de prefijo de mayor coincidencia consiste en tomar el segmento de dirección IP correspondiente a la etapa de búsqueda y acceder a los vectores de bits para determinar el indice del NHI en la tabla, o continuar en la siguiente etapa. La figura 3.7 muestra un ejemplo de dicho esquema en una etapa cualquiera. El *subtrie* tiene una profundidad de 3 lo que lo posibilita a contener 7 prefijos y hasta 8 apuntadores a otros *subtries* de etapas siguientes; sin embargo, el ejemplo tiene tres prefijos que a su vez están ordenados en la tabla de encaminamiento. El primer arreglo de consulta tiene un tamaño de 7 bits y se obtiene con marcadores de la posición de cada prefijo en cada nivel del *trie*; el segundo arreglo de consulta es de tamaño 8 bits y también se obtiene con los marcadores de las posiciones de los *tries* de niveles inferiores.

Entre sus ventajas destaca su escalabilidad a IPv6, esto debido a la generalidad de sus etapas. Entre sus desventajas destaca un consumo alto de memoria en comparación con el esquema de *lulea* ya que es un esquema diseñado para operar sobre memora RAM; además, la profundidad de los *tries* en cada etapa no está justificada en términos de eficiencia en memoria o rapidez en una búsqueda LPM.

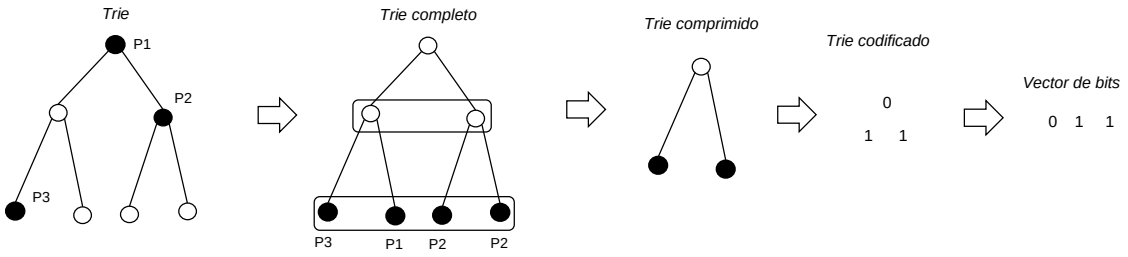


Figura 3.8: Trie multi bit

3.3.4.3. Tries multibit

Este mecanismo de búsqueda LPM es una variación de [17]; la diferencia es que el *trie* binario es comprimido para generar vectores de bits de menor tamaño y es utilizado por muchos esquemas actuales para lograr una eficiencia mayor en memoria [27]. La compresión del *trie* se logra al convertir al mismo en uno equivalente de menor profundidad para después codificarlo en un vector de bits. En la figura 3.8 se muestra un ejemplo de compresión del *trie* binario de profundidad 3 de la figura anterior. Al igual que el esquema de *lulea*, los prefijos en nodos intermedios son empujados hasta las hojas; después de ello, se construye un *trie* y una tabla de encaminamiento de menor tamaño (en este caso de profundidad 1) y finalmente se codifica en un vector de 3 bits, en comparación de los 7 bits utilizados en el ejemplo anterior.

La compresión es dinámica dependiendo de la distribución de prefijos en la tabla de encaminamiento. El esquema de compresión aumenta su eficacia en memoria; sin embargo, ya que la cantidad de memoria requerida para éste y esquemas anteriores crece de forma exponencial de acuerdo con la profundidad del *trie;* no es posible realizar la búsqueda en una sola etapa. En IPv6 es más evidente la necesidad de justificar adecuadamente la cantidad de las etapas y los parámetros de operación de cada una de ellas, ya que cualquier virtud de los esquemas anteriores se puede ver minimizada si no operan bajo parámetros optimizados.

3.3.4.4. Compresión del *trie* binario

Las tres propuestas que explicamos anteriormente en esta subsección son base de la compresión y/o codificación del *trie* binario. A lo largo del tiempo y hasta la actualidad existen variaciones, mejoras y por supuesto escalamiento a IPv6 a estos algoritmos. Ejemplo de ello es lo que se propone en [44] en donde el *trie* binario es comprimido en un vector de bits al igual que el esquema base de *tries* multibits [27] cuya diferencia radica en las profundidades de los *tries* y el grado de la compresión.

Dada la amplitud de los prefijos en IPv6 (longitud del prefijo y distribución de los mismos en la tabla de encaminamiento) la estructura del *trie* es diferente en cada tabla de encaminamiento y en cada actualización de la misma. De esta manera, se aborda el tema de la partición óptima del prefijo, partición óptima de la etapa y codificación óptima del *tri*e binario.

3.3.5. Esquemas de búsqueda LPM paralelos

Una manera de abordar el problema de búsquedas LPM es utilizar la programación paralela. El uso de GPU y núcleos permite hacer uso de estas técnicas. Sin embargo, cuando se divide la búsqueda LPM en etapas, una etapa no puede comenzar hasta terminar la etapa anterior, de esta manera se obtiene a lo más una optimización conocida como *pipeline.*

3.3.5.1. Búsqueda con GPU

Las tarjetas de gráficos contienen un gran número de microprocesadores de capacidad limitada donde su función consiste en mover o intercambiar *pixeles* en una pantalla. Una operación de búsqueda binaria en el *trie* no es más compleja que mover un *pixel*; de esta manera, los trabajos presentados en [46] y en [47] ofrecen una solución que garantiza una operación de búsqueda por GPU.

3.3.5.2. Búsqueda con *multicore*

Ya que las computadoras de propósito general poseen en su mayoría núcleos ya sea físicos o virtuales se puede utilizar programación paralela para realizar la búsqueda LPM; tal es el caso de [48] en donde, se segmenta la búsqueda en etapas y cada una de ellas es resuelta por un núcleo diferente.

3.3.6. Esquemas de segmentación en la búsqueda LPM

El esquema de particionamiento en etapas de la búsqueda LPM que se presenta en la propuesta [18] es una innovadora forma de abordar el ahorro en memoria mediante la adecuada elección de los tamaños de segmentos de direcciones IP en cada etapa de búsqueda LPM. La propuesta consiste en representar la tabla de encaminamiento en un *trie* de prefijos disjuntos seccionando dicho *trie* en niveles. Para generalizar el costo de un *subtrie*, se considera un costo proporcional al número de nodos que éste pueda tener, de esta forma, solo se presenta un posible costo que se considera constante sin tener que hablar de una codificación en particular. La generalidad más importante del esquema es que se considera un número variable de etapas en la búsqueda LPM mapeadas a los niveles de un *trie* binario, esto quiere decir que, cada etapa se puede localizar en cualquier profundidad del *trie* binario con un costo en particular, lo que implica, un problema combinatorio para determinar la localización apropiada de las etapas que garanticen una optimizacion en memoria requerida por seccionar la búsqueda LPM en etapas. Para determinar la mejor localización de los niveles, en esta propuesta se utiliza programación dinámica donde se reportan latencias de ejecución factibles al tiempo de actualización de la información de encaminamiento de las tablas usadas. También es importante mencionar que este esquema se puede utilizar en IPv4 e IPv6 y que en en IPv6 los costos en memoria requerida no pueden ser contenidos en memoria de rápido acceso, sin embargo, se propone el uso de memorias DDR5 [15], de baja latencia de acceso para eficientar la búsqueda LPM.

Una adaptación importante al esquema descrito anteriormente se muestra en [45] en donde no solo se habla de etapas que minimicen costos de acuerdo a la distribución de prefijos en la tabla de búsqueda LPM sino que también, se aborda el tema de una compresión que pueda ajustarse de la misma manera que las etapas. En este estudio

la compresión consiste en la aplicabilidad del almacenamiento en caché de reglas y la compresión con pérdida para crear clasificadores. La innovación consiste en encontrar los clasificadores que poseen menor pérdida y menor consumo en memoria. Para lograr este cometido se utiliza programación dinámica pera reducir el tiempo de ejecución y permitir las búsquedas LPM entre actualizaciones de la tabla.

En ambas propuestas expuestas se da solución con un algoritmo determinista y aunque, se obtenga una solución que garantice ser la mejor para cada tabla de encaminamiento, dicha tabla se actualizara en un tiempo no mayor a 1 segundo por lo que es cuestionable si es necesario una solución optimizada o una solución exacta.

Capítulo 4

Propuesta de esquema de búsqueda LPM IPv4/6

En este capítulo presentamos nuestra propuesta de esquema para realizar búsquedas LPM tanto en IPv4 así como en IPv6. Empezaremos explicando el principio de búsqueda del prefijo de mayor coincidencia basado en el conteo de valores de un vector de bits, constituido a partir del *trie* binario; explicamos su construcción, tamaño y características. Después presentamos una de nuestras principales aportaciones que es, la generalización de la distribución de conteos precalculados del vector de bits en múltiples arreglos contadores. La idea proviene de un caso particular el cual es el esquema de *Lulea.* Con esta generalización podemos determinar parámetros que optimicen el desempeño en la búsqueda LPM tanto en requerimentos de memoria como en número de accesos a la misma.

Debido a que los esquemas de búsqueda LPM basados en el *trie* binario aumentan de forma exponencial su consumo en memoria mientras mayor sea su profundidad; es necesario realizar la búsqueda en etapas. En la última sección del capítulo presentamos nuestra segunda innovación que es, un esquema de particionamiento eficiente del *trie* binario basado en la exploración heurística. El esquema minimiza los requerimientos de memoria en base a la distribución de prefijos y es capaz de soportar la rapidez de las actualizaciones de las tablas de encaminamiento.

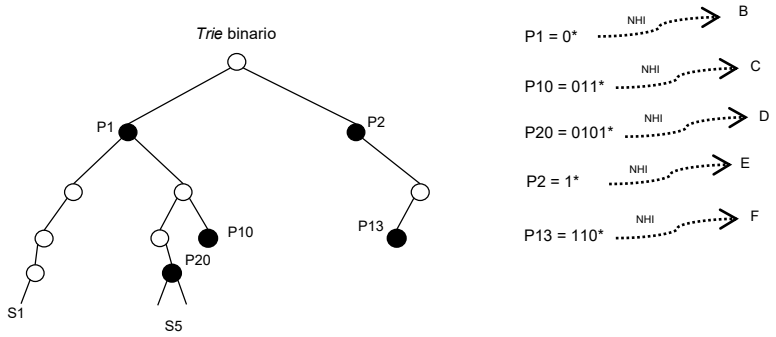


Figura 4.1: Tabla de encaminamiento y prefijos en el *trie* binario

4.1. Base de la búsqueda LPM

Comenzamos utilizando el ejemplo de un fragmento de una tabla de encaminamiento. El fragmento corresponde a la parte de la tabla que abarca a los prefijos cuya máxima longitud es de 4 bits. La figura 4.1 muestra la tabla y el *trie* correspondiente; se puede observar que los prefijos de red se definen mediante recorridos en el *trie*, además de establecer apuntadores a otros *tries* de niveles inferiores correspondientes a prefijos de longitud mayor a 4 bits como se indica en la figura.

En seguida presentamos un tipo particular de codificación del *trie* que facilita posteriormente realizar búsquedas LPM en ella. Resaltamos las ventajas, desventajas y viabilidad de utilizar la codificación presentada.

4.1.1. Vector de bits

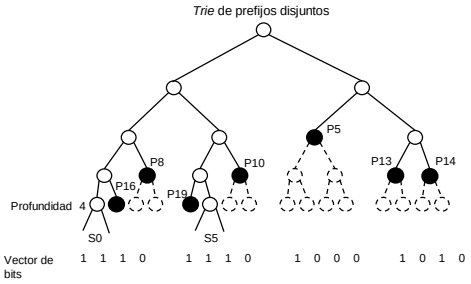
La búsqueda LPM utilizando recorridos en el *trie* se dificulta debido a que, un prefijo de red puede ubicarse hasta una hoja o bien, en un nodo intermedio. Lo que significa que durante un recorrido, si se ha encontrado un posible prefijo, se debe guardar un registro del prefijo obtenido y continuar el recorrido (la búsqueda) ya que aún no existe la posibilidad de encontrar un prefijo de mayor longitud.

Si todos los prefijos fueran disjuntos (ubicados todos en las hojas) no habría necesidad

Tabla de apuntadores/NHI

Índice de acceso	Prefijo/ Sub-trie	Apuntador o NHI
0	S0 = 0000*	AP_0
1	P16 = 0001*	B
2	P8 = 001*	B
3	P19 = 0100*	B
4	S5 = 0101*	AP_5
5	P10 = 011*	C
6	P5 = 10*	E
7	P13 = 110*	F
8	P14 = 111*	E

(a)



(b)

Figura 4.2: (a) Tabla extendida de prefijos disjuntos. (b) *Trie* binario de prefijos disjuntos y vector de bits correspondiente.

de mantener un registro del último prefijo visitado y el resultado de la búsqueda LPM se tendría siempre al final del recorrido debido a que todos los prefijos están en las hojas del *trie*. Esta es la ventaja más evidente pero las tablas con prefijos disjuntos se aprovechan en diversos esquemas de búsqueda LPM. Una manera de obtener prefijos disjuntos es utilizando la técnica conocida como *Leaf-Pushing* [29] que básicamente consiste en eliminar los prefijos ancestros de los nodos internos, remplazándolos por nuevos prefijos equivalentes ubicados en las hojas del *trie*. Cuando los prefijos de longitud menor de 4 bits, ilustrados en la figura 4.1(a), se trasladan a las hojas, utilizando *leaf-pushing*, el fragmento de tabla resultante es el que se muestra en la figura 4.2(a). En adelante nos referiremos a esta tabla como tabla extendida de prefijos disjuntos.

En vez de hacer recorridos, una búsqueda LPM en una tabla de prefijos disjuntos se puede reducir a determinar a qué intervalo de los cubiertos por cada prefijo corresponde la dirección IP. En otras palabras, esto significa que, cada prefijo en una hoja, es el prefijo de mayor longitud de un intervalo de direcciones IP que están dentro del *trie* correspondiente que se forma hasta la profundidad del mismo. Encontrar el intervalo correspondiente a una dirección IP se puede lograr con la ayuda de una estructura de datos llamada vector de bits. Un vector de bits es un arreglo cuyas entradas son bits y hay una entrada por cada dirección IP posible. Por ejemplo para IPv4 se requeriría un vector de 2^{32} entradas y un prefijo de l bits de longitud representa o abarca un intervalo de 2^{32-l} direcciones.

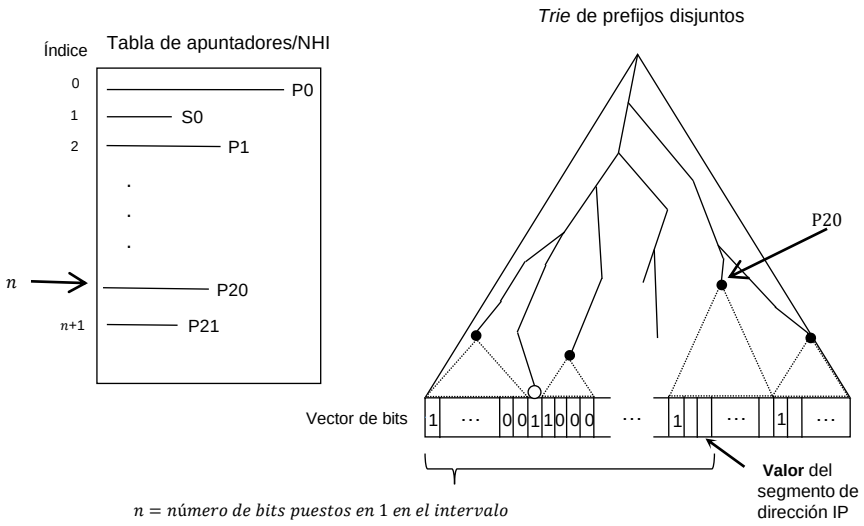


Figura 4.3: Relación entre vector de bits y tabla extendida de prefijos disjuntos para realizar una búsqueda LPM.

Para identificar los intervalos disjuntos en el vector de bits, se coloca un valor de 1 en la primera entrada del intervalo de direcciones que abarca un prefijo, en las entradas restantes se colocan valores de 0. La búsqueda LPM se realiza con el vector de bits y la tabla extendida de prefijos disjuntos en la que los prefijos siguen el mismo orden que los intervalos correspondientes en el vector de bits de izquierda a derecha. La figura 4.2(b) muestra el vector de bits correspondiente con la tabla de prefijos disjuntos de la figura 4.2(a) si las direcciones fueran de tan solo 4 bits (por el momento tratando a S0 y S5 como prefijos de red).

Para la búsqueda LPM primero se utiliza la dirección IP como índice en el vector de bits, luego se cuenta el número de bits puestos en 1 desde el extremo izquierdo del vector hasta la posición indexada por la dirección y para finalizar, el resultado de la cuenta de 1s se toma como índice en la tabla extendida para obtener la NHI que corresponde a la dirección (ver figura 4.3).

4.1.2. Búsqueda mediante un conteo precomputado

El esquema de conteos en el vector de bits es muy simple pero el tiempo de búsqueda, que es el tiempo en segundos necesario para realizar el conteo, depende de la latencia de acceso a memoria y del valor numérico de la dirección IP donde, en el peor de los casos serían necesarios 2^{32} accesos a memoria para el valor numérico más grande en el caso de IPv4. Para evitar tener que recontar los bits del vector puestos en 1 en cada nueva búsqueda, en vez del vector de bits, se puede utilizar un arreglo con las cuentas de 1 precomputadas y el valor numérico de la dirección ahora indexa este arreglo en vez de indexar el vector de bits (figura 4.4). Así con solamente 2 accesos a memoria, uno para el arreglo contador y otro para la tabla extendida se completa una búsqueda LPM (figura 4.5). La ecuación (4.1) muestra la función del costo en instrucciones f_i (accesos a memoria) para el esquema de búsqueda LPM que utiliza un arreglo contador.

$$f_i = 2 \tag{4.1}$$

Con este esquema, para un *trie* de profundidad K correspondiente a un segmento de dirección IP de longitud K, la función de costo en memoria necesaria en bits para guardar el arreglo contador además de la necesaria para almacenar la tabla extendida está definida en la ecuación (4.2). La obtención de la función es bastante sencilla; el arreglo contador tiene tantas entradas como posibles hojas tiene el *trie* completo y cada entrada tiene los bits necesarios para soportar la máxima cuenta de prefijos de dicho *trie*. Entonces para direcciones de 32 bits como en IPv4, $K = 32$, para el arreglo contador se necesitan 16 GBytes y para IPv6 esta cantidad crece de forma exponencial.

$$f_m = K \times 2^K \tag{4.2}$$

Cabe señalar que los costos efectivos tanto de acceso a memoria así como la cantidad de memoria dependen precisamente de las características de la memoria o niveles de memoria del dispositivo de encaminamiento donde se realiza la búsqueda LPM. Esto significa que, un acceso a memoria teórico puede significar uno o más accesos a memoria real debido al tamaño de los registros, ancho de bus, pipeline, etcétera.

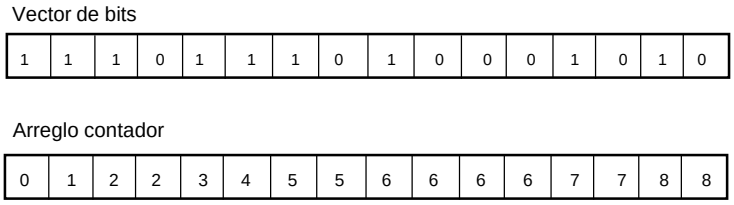


Figura 4.4: Vector de bits con su correspondiente arreglo contador.

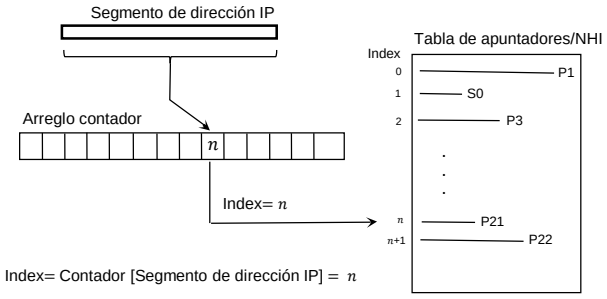


Figura 4.5: Algoritmo de búsqueda LPM usando un arreglo contador.

4.2. Propuesta de esquema general de búsqueda LPM

El esquema de *Lulea* [8] utiliza la misma noción del vector de bits presentado en la sección anterior para un *trie* binario de profundidad $K = 16$. Sin embargo, para ahorrar el consumo en memoria, ellos realizan la búsqueda LPM distribuyendo el conteo en tres conteos en lugar de uno. En este esquema hay un arreglo asociado a cada uno de los niveles 10, 12 y 16. El esquema de *Lulea* tiene un alto desempeño en requerimientos de memoria y tiempos ejecución. Sin embargo, no justifica el porqué de la elección de estos niveles y la profundidad del *trie*. Esta deficiencia nos ha motivado a buscar una generalización en el esquema para distribuir el conteo en varios arreglos contadores y en *tries* de cualquier profundidad con el fin de encontrar la mejor elección de niveles que minimice el consumo de memoria.

4.2.1. Búsqueda mediante varios conteos precalculados

Proponemos un esquema que generaliza al esquema de *Lulea*. En este esquema distribuimos el almacenamiento de las cuentas precalculadas de los bits 1 de un vector de bits en varios arreglos de manera jerarquizada. A continuación, explicamos nuestra propuesta.

Consideremos un *trie* de profundidad K que posee todos los posibles nodos en todos sus $K + 1$ niveles (siendo la raíz el nivel 0), a esta estructura en árbol nos referiremos en adelante como un *trie* completo. En este *trie* completo, ciertos nodos corresponden a los prefijos de la tabla de encaminamiento (que previamente han sido expandidos de tal forma que estos prefijos son disjuntos). Como se trata de un *trie* completo, este *trie* de profundidad K tiene hasta 2^K hojas (nodos del nivel K). A este *trie* de profundidad le vamos a hacer corresponder un vector de tantos bits como hojas tiene el *trie*; es decir, un vector de bits de tamaño 2^K.

Este vector de bits tendrá tantos bits puestos a 1 como prefijos tiene el *trie*. Para saber cuál bit en el vector de bits le corresponde a cada prefijo, procedemos de la siguiente manera. Tomemos a un nodo prefijo no ubicado en una hoja. A este prefijo le corresponde un subárbol, donde dicho prefijo es la raíz. Consideremos las hojas de dicho subárbol. Recordemos que existe una correspondencia uno a uno entre las hojas del *trie* y las en-

tradas del vector de bits. Al conjunto de las entradas del vector de bits correspondientes a las hojas del subárbol cuya raíz es el prefijo en cuestión le llamaremos la COBERTURA del prefijo. Así, cada prefijo tiene su cobertura correspondiente en el vector de bits. En la cobertura de cada prefijo pondremos a 1 el primer bit (el bit más a la izquierda) y los demás bits a 0. Como los prefijos son disjuntos sus coberturas no se traslapan. Con esto aseguramos que a cada prefijo le corresponde sólo un bit en el vector de bits y que este bit es el primero de su correspondiente cobertura.

Recordemos que nuestro *trie* completo tiene $K+1$ niveles $\{0,\ldots,K\}$. Dejemos por el momento el nivel 0 que corresponde a la raíz y consideremos el conjunto de niveles $T=\{1,2,\ldots,K\}$. Seleccionemos h de estos K niveles para definir el conjunto de niveles $L_h=\{l_1,l_2,...,l_i,...,l_h\}$, con $l_i \in T$ y $l_h = K$ fijo. Además $1 \leq h \leq K$. Definimos a $(l_i)_{1\leq i\leq h}$, como la sucesión de los elementos L_h ordenados del menor al mayor; es decir $(l_i)_{1\leq i\leq h} = l_1,l_2,...,l_h$ con $l_i \in L_h$, donde $\forall i \;\; l_{i-1} < l_i$, y ademas definimos a $l_0 = 0$ para considerar el nivel 0 en la raíz. A esta sucesión le llamaremos la ESCALA DE NIVELES.

Por cada elemento l_i de la escala de niveles $(l_i)_{1\leq i\leq h}$, definimos un arreglo A_i que almacenará las cuentas precalculadas. El número de entradas del arreglo A_i será igual al número de nodos en el correspondiente nivel l_i del *trie* completo; es decir, el arreglo A_i tendrá 2^{li} entradas. De esta forma, asociamos cada entrada del arreglo A_i con exactamente un nodo en el nivel l_i del *trie* completo (ver figura 4.6). Note que el arreglo A_h correspondiente al nivel $l_h = K$ tiene entonces tantas entradas como el vector de bits.

Considere un nodo cualquiera n en el nivel l_i del *trie* completo. Vamos a llamar ANCESTRO ESCALÓN de n al ancestro del nodo n en el nivel l_{i-1}. También vamos a definir la cobertura del nodo n como el conjunto de las entradas del vector de bits correspondientes a las hojas en el nivel l_{i+1} del subárbol cuya raíz es el nodo n. Es decir, aplicamos la misma definición de cobertura independientemente de que el nodo sea o no un prefijo. (ver figura 4.7). Además, como hay una correspondencia biunívoca entre los nodos del *trie* en el nivel l_i y las entradas del arreglo A_i, nos referiremos indistintamente a la cobertura del nodo l_i o bien a la cobertura de la entrada correspondiente en el arreglo A_i. Queriendo significar en ambos casos la misma cobertura.

Recordemos que el vector de bits tiene un bit puesto a 1 por cada prefijo existente y que dicho bit puesto a 1 está al inicio de la cobertura correspondiente a dicho prefijo.

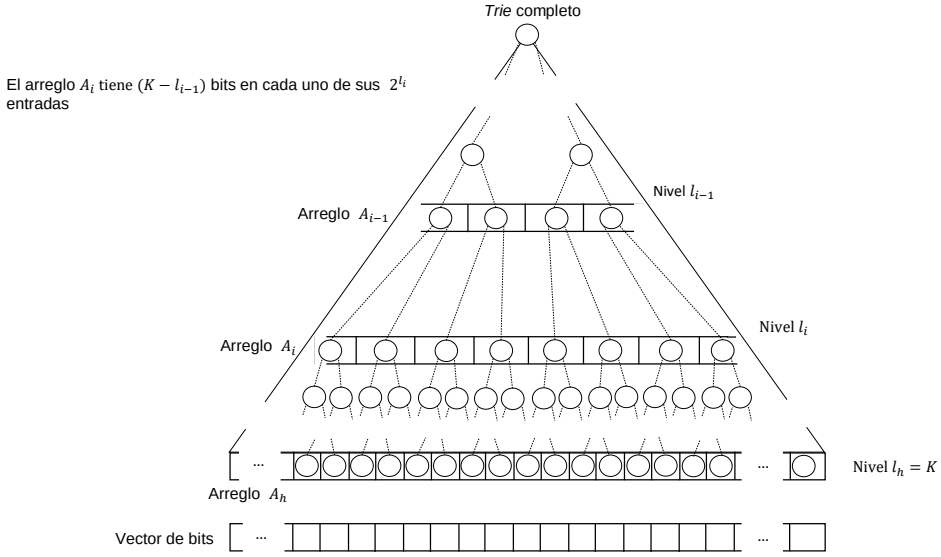


Figura 4.6: Relación entre el *trie* completo y los arreglos contadores.

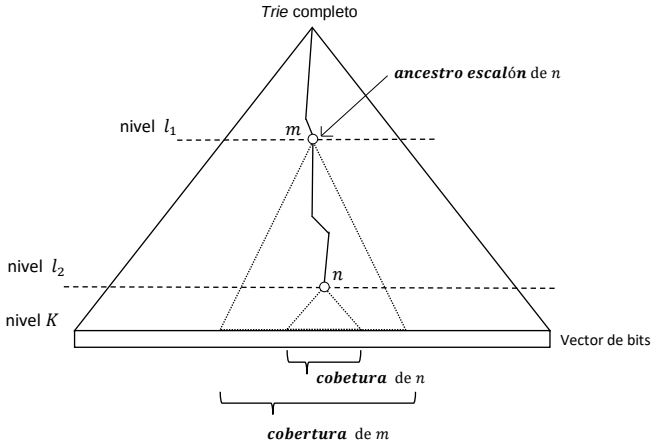


Figura 4.7: *cobertura* y *ancestro escalón* de un nodo.

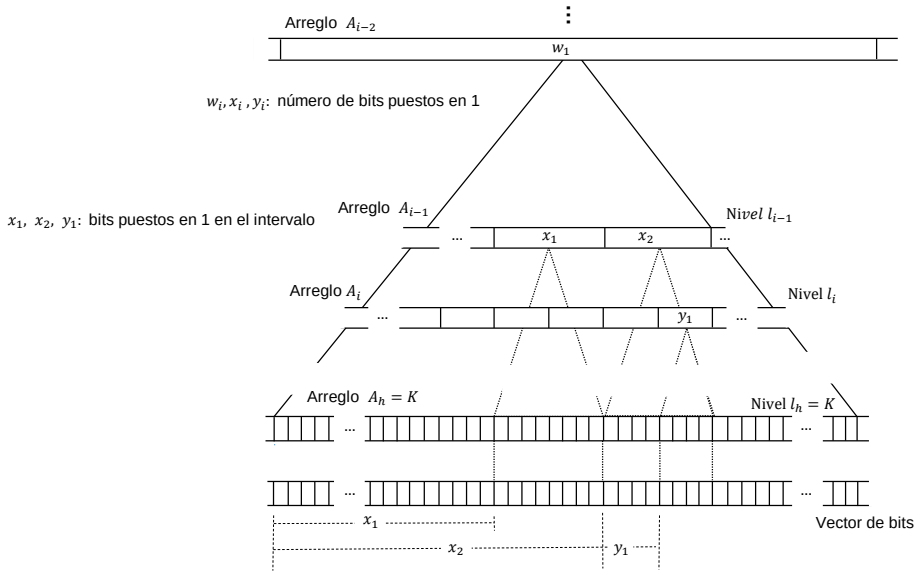


Figura 4.8: Contenidos de los arreglos contadores de acuerdo al vector de bits.

Además, como los prefijos son disjuntos sus coberturas correspondientes no se traslapan. Esto nos permite definir un orden de los prefijos, el mismo que tienen sus correspondientes coberturas en el vector de bits. A cada cobertura le podemos hacer corresponder un número que se usará como índice para acceder a una tabla con los prefijos y así obtener la información de encaminamiento correspondiente que ahí se encuentra. Este número se puede obtener simplemente contando el número de bits 1 desde el extremo izquierdo del vector de bits hasta la cobertura en cuestión.

Por otro lado, como el vector de bits tiene 2^K entradas y existen 2^K direcciones IP de K bits, podemos usar el valor de la dirección IP como índice para acceder a una y sólo una de las entradas del vector de bits. Y lo mismo sucede con las entradas del arreglo A_h puesto que hay una correspondencia biunívoca entre las entradas del vector de bits y las entradas del arreglo A_h.

Podemos deducir de lo anterior entonces que a cada cobertura de un prefijo le corresponde un intervalo continuo de direcciones IP y sabemos que sólo el primer bit de esta cobertura será 1. Además de que, para las direcciones IP comprendidas en dicha cobertura, el prefijo correspondiente a dicha cobertura será de hecho su prefijo de mayor coincidencia.

Así que encontrar el prefijo de mayor coincidencia para una dirección IP dada, se reduce a determinar el número de cobertura que le corresponde. Como ya dijimos, para determinar el número de cobertura buscado sólo necesitamos contar el número de bits puestos a 1 a la izquierda del bit que corresponde a la dirección IP en el vector de bits. Y para evitar el conteo repetitivo cada vez que se tenga que hacer una búsqueda del prefijo de mayor coincidencia, precalcularemos esta cuenta. Además, no almacenaremos la cuenta total en un solo lugar sino que almacenaremos cuentas parciales que al ser sumadas nos darán la cuenta total. Las cuentas parciales las almacenaremos precisamente en los h arreglos A_i. La idea de almacenar cuentas parciales en lugar de una cuenta total es para hacer más eficiente el consumo de memoria, como se verá a continuación.

Sea n un nodo cualquiera en el nivel l_i del *trie* completo. Sea m el ancestro escalón de n y por lo tanto un nodo en el nivel nivel l_{i-1}. Sea C la cobertura de n y sea D la cobertura de m. Por lo tanto $C \subset D$.

Vamos a definir la CUENTA PRECEDENTE INTRACOBERTURA o bien la CPI de C, como la cuenta de bits 1 en el vector de bits que se encuentran a la izquierda de la cobertura C pero circunscribiéndose a la cobertura D. En nuestra propuesta, como ya mencionamos, el conteo de los bits puestos a 1 en el vector de bits lo distribuimos en cuentas parciales almacenadas en h arreglos A_i. Estas cuentas parciales son precisamente las CPI que acabamos de definir. Así que cada entrada n del arreglo A_i guardará la CPI correspondiente a la cobertura de n (ver figura 4.8). Como el tamaño de la cobertura para un nodo en un nivel cualquiera l_j es de 2^{K-l_j}, entonces para una entrada n en el arreglo A_i (que sabemos corresponde al nivel l_i), el máximo valor de la CPI de la cobertura de n es $2^{K-l_{i-1}}$; es decir, el tamaño de la cobertura del ancestro escalón de n. Y por lo tanto el tamaño de cada entrada del arreglo A_i debe ser de $K - l_{i-1}$ bits. Note que la máxima cuenta ocurriría sólo cuando la entrada n está en el arreglo A_h y por tanto su cobertura es de tamaño 1 y esta cobertura estuviera en el extremo derecho de la cobertura del ancestro escalón de n, y además la cobertura de dicho ancestro escalón estuviera completamente llena de bits puestos a 1.

Una vez que todas las entradas de los h arreglos A_i se han llenado con las correspondientes CPI, es claro que ya no necesitamos el vector de bits para realizar las búsquedas del prefijo de mayor coincidencia para una dirección IP dada. Para realizar dicha búsqueda

necesitamos solamente los h arreglos A_i, como mostramos a continuación.

En la figura 4.9 se puede observar un ejemplo de búsqueda del prefijo de mayor coincidencia utilizando las CPI distribuidas en los h arreglos A_i y asumiendo una dirección IP de K bits. Recordemos que básicamente lo que se requiere es contar el número de bits puestos a 1 en el vector de bits a la izquierda de la entrada en dicho vector, entrada que está indexada por el valor de la dirección IP en cuestión. Dicha cuenta se obtiene de la siguiente manera: accedemos en el arreglo A_h a la entrada indexada por el valor de la dirección IP. Tomamos la cuenta (CPI) que se encuentra ahí guardada. Luego vamos a la entrada correspondiente al ancestro escalón de la entrada que accedimos en el arreglo. Dicho ancestro escalón se encuentra en el arreglo A_{h-1}. Para hacer esto, se toman los l_{h-1} bits más significativos de la dirección IP para indexar el arreglo A_{h-1}. Tomamos la cuenta (CPI) almacenada ahí y la acumulamos a la cuenta que habíamos encontrado primeramente. Luego haremos los mismo con el ancestro escalón de esta última entrada, la del arreglo A_{h-1}. Para esto tomamos los l_{h-2} bits más significativos para obtener nuevamente la siguiente cuenta parcial (CPI). Este proceso continúa de la misma manera de ancestro escalón en ancestro escalón hasta llegar a la entrada en el arreglo A_1, adicionamos a la cuenta acumulada la última CPI correspondiente y así obtenemos finalmente la cuenta acumulada total. Esta cuenta acumulada total será entonces el resultado de haber sumado h CPI. Finalmente usamos el valor de esta cuenta acumulada para indexar la tabla de prefijos y así obtener el prefijo de mayor coincidencia o bien al apuntador a otro *subtrie*.

El objetivo de almacenar cuentas parciales CPI en h arreglos es optimizar el consumo de memoria. Para determinar cuántos niveles y qué elección de ellos minimiza la cantidad de memoria necesaria, haremos el siguiente análisis.

Sea $T = \{1, 2, \ldots, K\}$ el conjunto de los niveles elegibles de nuestro *trie* completo; es decir, todos los niveles excepto el de la raíz.

Sea la colección de $K - 1$ conjuntos $\{\Omega_2, \Omega_3, ..., \Omega_K\}$, donde Ω_h (para $2 \leq h \leq K$) contiene todos los subconjuntos que tienen exactamente h elementos de T. Por lo tanto $|\Omega_h| = \binom{K}{h}$. Si L_h denota un subconjunto contenido en Ω_h, definimos a $(l_i)_{1 \leq i \leq h}$ como la sucesión de los elementos de L_h ordenados del menor al mayor; es decir $(l_i)_{1 \leq i \leq h} = l_1, l_2, ..., l_h$ con $l_i \in L_h$, donde $\forall i \;\; l_{i-1} < l_i$, y definimos a $l_0 = 0$.

A cada sucesión $(l_i)_{1 \leq i \leq h}$ le corresponde un conjunto de h arreglos A_i donde almace-

namos las CPIs. La memoria necesaria en bits para almacenar este conjunto de arreglos está dada por la ecuación (4.3).

$$M((l_i)_{1\leq i\leq h}) = \sum_{i=1}^{h}[2^{l_i} \times (K - l_{i-1})] \tag{4.3}$$

$$con \quad 2 \leq h \leq K \quad y \quad l_0 = 0$$

En la ecuación vemos que el consumo de memoria depende del número de niveles elegidos h y de cuáles niveles se eligieron. En la ecuación se consideran los h arreglos A_i, 2^{l_i} corresponde al número de entradas de cada arreglo A_i, $K - l_{i-1}$ es el tamaño en bits de cada entrada del arreglo A_i. Nótese que el costo no depende ni de la distribución ni del número de prefijos en la tabla extendida.

Para un valor de K dado, nos interesa encontrar la sucesión a $(l_i)_{1\leq i\leq h}$ (con $h = 2, 3, ..., K$) que minimice el consumo de memoria requerida para almacenar los h arreglos A_i correspondientes. Denotemos con $(\mu_i)_{1\leq i\leq h}$ a tal sucesión. Entonces, tenemos que encontrar $(\mu_i)_{1\leq i\leq h}$ tal que:

$$M((\mu_i)_{1\leq i\leq h}) = \begin{array}{c} min \\ (l_i)_{1\leq i\leq h} \\ 2 \leq h \leq K \end{array} \sum_{i=1}^{h}[2^{l_i} \times (K - l_{i-1})]$$

Veamos cuántos términos hay en el argumento de la función min; es decir, cuántos términos suma habrá que calcular. Esto equivale a preguntarnos cuántos subconjuntos de $T = \{1, 2, \ldots, K\}$ existen, exceptuando el conjunto vacío y los K subconjuntos de un único elemento, pues $h = 2, 3, ..., K$. Se sabe que el número de subconjuntos de $T = \{1, 2, \ldots, K\}$ es igual a 2^K, luego entonces tenemos que realizar un total de $2^K - K - 1$ sumas para encontrar dicho mínimo y así poder determinar la sucesión $(\mu_i)_{1\leq i\leq h}$ buscada. En el siguiente capítulo mostramos los resultados obtenidos utilizando este análisis que acabamos de desarrollar. Es decir, K la vamos a variar desde 2 hasta un máximo de 32 que es el tamaño de las direcciones IPv4.

La ecuación (4.4) muestra la función en accesos a memoria necesarios para realizar

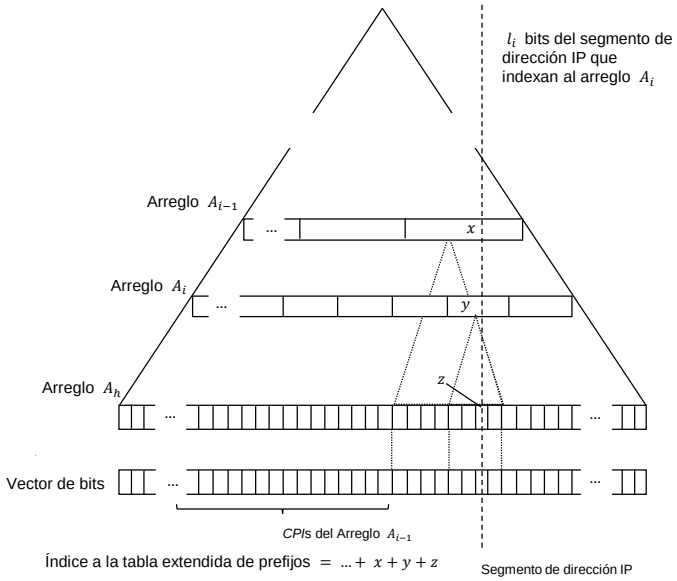


Figura 4.9: Búsqueda LPM utilizando arreglos contadores.

una búsqueda del prefijo de mayor coincidencia para una dirección IP de longitud K bits. En la ecuación los h accesos a memoria corresponden al hecho de que hay que leer una entrada por cada uno de los h arreglos A_i, y el uno es debido al acceso a la tabla extendida de prefijos disjuntos. Por lo tanto, la complejidad en accesos a memoria es constante.

$$f_i = h + 1 \tag{4.4}$$

4.2.2. Búsqueda mediante varios conteos precalculados y patrones de bits

Recordemos que conceptualmente los prefijos disjuntos de la tabla de encaminamiento están en algunos nodos de un *trie* de profundidad K y que dicho *trie* tiene 2^K hojas. Y tenemos, también conceptualmente, un vector de bits con tantas entradas como hojas tiene el *trie*; es decir, un vector de 2^K bits. Existe entonces una correspondencia biunívoca entre las hojas del *trie* y las entradas del vector de bits. Cada hoja, y por tanto cada entrada del vector de bits, representa una de las 2^K posibles direcciones IP.

Hemos definido al conjunto de las entradas del vector de bits correspondientes a las hojas del subárbol cuya raíz es un prefijo como la cobertura del prefijo. Y en general, a cada nodo del *trie* le corresponde una cobertura; la cobertura que está formada por las entradas del vector de bits correspondientes a las hojas del subárbol cuya raíz es dicho nodo.

Por otro lado, tenemos h arreglos A_i tal que, $i = 1, 2, ..., h$. Es decir, cada arreglo está asociado a uno de los h niveles seleccionados del *trie*. Y hay una correspondencia biunívoca entre las entradas del arreglo A_i y los nodos del nivel asociado l_i de dicho *trie*, donde $l_i \in \{1, 2, ..., K\}$. Así que hemos definido al conjunto de las entradas del vector de bits correspondientes a las hojas del subárbol cuya raíz es un nodo asociado a la entrada de un arreglo A_i como la cobertura de dicha entrada.

El arreglo A_h es el arreglo asociado al nivel l_h y sabemos que $l_h = K$. El nivel l_h es el último nivel del *trie*, es decir donde están las hojas del *trie*. Luego entonces, el arreglo A_h es el que tiene mayor número de entradas, a saber 2^Kentradas. Por lo tanto, este arreglo A_h contribuye en gran medida a la cantidad de memoria que se requiere.

De hecho el arreglo A_h no sólo es quien posee la mayor cantidad de entradas, también es el de mayor costo entre los demás arreglos contadores. Ambas afirmaciones las podemos demostrar de la siguiente manera. Recordemos que tenemos h arreglos contadores A_i tal que, $i = 1, 2, ..., h$ y que estos arreglos tienen una correspondencia unívoca con los niveles $(l_i)_{1 \leq i \leq h} = \{l_1, l_2, ..., l_h\}$; de esta forma, cada arreglo A_i tiene un tamaño que está dado por el número de entradas (2^{l_i}) y por el tamaño de las mismas $(K - l_{i-1})$; ya que $l_1 < l_2 < ... < l_{h-1} < l_h = K$ entonces $2^{l_1} < 2^{l_2} < ... < 2^{l_{h-1}} < 2^{l_h = K}$ con esto demostramos que el arreglo A_h tiene 2^K entradas y es quien tiene una mayor cantidad de ellas. Por otro lado, el tamaño mínimo que se pueda tener en las entradas el arreglo A_h, es de un bit, esto sucede cuando el nivel l_{h-1} es igual a $K - 1$, en otras palabras $(K - l_{i-1}) = (K - (K - 1)) = 1$, mientras que el tamaño máximo de las entradas corresponde al primer arreglo l_1 y tiene un tamaño de $(K - 0) = K$ bits; en consecuencia los tamaños mínimos y máximos de las entradas de los arreglos A_i oscilan entre 1 y K. Demos el menor tamaño a las entradas del arreglo A_h con $l_{h-1} = K - 1$, de esta forma las entradas del arreglo A_h son de 1 bit $2^K \times (K - (K - 1)) = 2^K \times (1) = 2^K$. Ahora vamos a conformar el arreglo A_{h-1} pero con un tamaño en cada una de sus entradas de

K bits y la mayor cantidad de entradas posibles después del arreglo A_h, esto es posible si $l_{h-1} = l_1 = (K-1)$; dicho arreglo entonces tiene un tamaño de $2^{l_{h-1}} \times (K) = 2^{K-1} \times (K) = 2^K$ bits que es igual al del arreglo A_h. Esto significa que el costo de A_h es menor A_{h-1} y solo se cumple la igualdad cuando $l_{h-1} = l_1 = K - 1$, en otro caso el arreglo A_h siempre tendrá el mayor número de entradas y el mayor costo en bits en comparación con los arreglos dados por $(l_i)_{1 \leq i \leq h}$.

En la sección anterior vimos cómo podemos encontrar para una dirección IP dada su prefijo de mayor coincidencia usando los h arreglos A_i. En esta sección veremos un nuevo esquema para encontrar el prefijo de mayor coincidencia. En este nuevo esquema vamos a deshacernos del extenso arreglo A_h y en su lugar usaremos una estructura de datos que requiere menos memoria. Recordemos que los valores de las entradas del vector de bits son función de la distribución de prefijos en el *trie*. Más específicamente, los bits 1 indican el inicio de la cobertura del prefijo correspondiente, y los demás bits de dicha cobertura son 0.

El arreglo A_h, al igual que los otros arreglos A_i, almacenan las correspondientes cuentas precedente intracobertura, CPI. Recalquemos el hecho de que la cuenta de bits 1 en una CPI se circunscribe a la cobertura del ancestro escalón correspondiente. Como el ancestro escalón de una entrada en A_h se encuentra en el arreglo A_{h-1}, entonces las CPI almacenadas en A_h se circunscriben a la cobertura de la entrada correspondiente del arreglo A_{h-1}.

Si observamos las CPI almacenadas en cada arreglo A_i veremos que forman secuencias crecientes. Y por cómo se han definido las CPI, estas secuencias están delimitadas por las coberturas del ancestro escalón correspondiente. Este ancestro escalón se encuentra en el arreglo A_{i-1}, como ya sabemos. Y por tanto en el arreglo A_i hay $2^{l_{i-1}}$ secuencias crecientes de CPI; es decir el mismo número que hay de entradas en el arreglo A_{i-1}. En otras palabras, a cada entrada del arreglo A_{i-1} le corresponde una secuencia de CPI almacenada en el arreglo A_i con $i = 2, 3, ..., h$. De manera equivalente podemos decir que a cada cobertura asociada a una entrada del arreglo A_{i-1} le corresponde una secuencia de CPIs almacenada en el arreglo A_i. En particular, nos vamos a interesar en las $2^{l_{h-1}}$ secuencias crecientes de CPI almacenadas en el arreglo A_h. Y de acuerdo a con lo que acabamos de decir, cada una de estas $2^{l_{h-1}}$ secuencias está asociada con una y sólo una

de las $2^{l_{h-1}}$ coberturas correspondientes a las entradas del arreglo A_{h-1}.

Cada una de las secuencias crecientes de CPI es función del patrón de bits de la correspondiente cobertura. Por ejemplo, si la cobertura fuera de 4 bits y tuviéramos el patrón 1011, la secuencia de CPI sería 1,1,2,3; en cambio si el patrón de bits fuera 1110 la secuencia sería 1,2,3,3. Notemos que el número de elementos de cada una de las secuencias almacenadas en el arreglo A_h es de $2^{K-l_{h-1}}$ el tamaño de las coberturas correspondientes; es decir, las asociadas a las entradas del arreglo A_{h-1}.

En general, habrá varias coberturas que tendrán la misma secuencia de CPI. En otras palabras, la misma secuencia de CPI estará almacenada varias veces en el arreglo A_{h-1}. Para evitar esta redundancia podemos usar la estrategia que describimos a continuación.

Almacenemos una sola copia de esta secuencia y hagámosla accesible para que se pueda consultar tantas veces como se requiera. Con esto haríamos que el arreglo A_h ya no fuera necesario.

Lógicamente, para que esta estrategia funcione tendremos que almacenar las secuencias de CPI de todos los patrones de bits posibles en las coberturas de las entradas del arreglo A_{h-1}. En principio, el número de patrones de bits diferentes sería $2^{2^{K-l_{h-1}}}$, ya que las coberturas de las entradas del arreglo A_{h-1} son de $2^{K-l_{h-1}}$ bits. Sin embargo, el hecho de que los patrones de bits de las coberturas están determinados por prefijos y en especial por prefijos disjuntos hace que no todos los $2^{2^{K-l_{h-1}}}$ patrones de bits sean posibles, como veremos más adelante.

En otras palabras, la idea es crear una estructura de datos diccionario con todos los patrones de bits posibles de longitud $2^{K-l_{h-1}}$ como llave y que al consultarlo podamos obtener como valor la secuencia de CPI y en particular la CPI asociada a una de las entradas del vector de bits; es decir, la entrada correspondiente a la dirección IP que nos ocupe en un momento dado para encontrar su prefijo de mayor coincidencia. Veamos ahora cuántas entradas tendrá nuestro diccionario; es decir, cuántos patrones de bits diferentes son posibles en una cobertura de $2^{K-l_{h-1}}$ bits. Para esto vamos primero a definir algunos conceptos que nos ayudarán a encontrar dichos patrones de bits posibles.

Sea V el conjunto de las entradas del vector de bits; es decir, $V = \{b_0, b_1, ...b_{2^K-1}\}$. Sea Δ_j el conjunto de todas las coberturas correspondientes a los nodos del *tr*ie en el nivel j, $j = 1, 2, ..., K$; es decir, $\Delta_j = \{\delta_0, \delta_1, ..., \delta_{2^j-1,}\}$. Notemos que Δ_j es una partición de V

ya que $V = \bigcup_{r=0}^{2^j-1} \delta_r$ y $\delta_s \bigcap \delta_t = \emptyset \ \forall \delta_s \neq \delta_t$.

Notemos también que cada cobertura Δ_{j-1} está formada por la unión de coberturas de Δ_j $j = 2, 3, ..., K$. O lo que es lo mismo, cada cobertura de Δ_j está contenida en una y sólo una cobertura de Δ_{j-1}. Se dice que Δ_j es un refinamiento de Δ_{j-1} y se denota por $\Delta_j \prec \Delta_{j-1}$. Como $j = 2, 3, ..., K$ tenemos en realidad refinamientos sucesivos; es decir, $\Delta_K \prec \Delta_{K-1} \prec \Delta_{K-2} \prec ... \prec \Delta_j \prec$.

Notemos que la cobertura c de un prefijo cualquiera de nuestro *trie* es necesariamente elemento de algún Δ_j; esto es $c \in \Delta_j$, para algún $j \in \{1, 2, ..., K\}$.

Recordemos que tenemos h arreglos A_i correspondientes a los niveles definidos por el conjunto $\{l_1, l_2, ...l_h\}$, con $l_i \in \{1, 2, ..., K\}$ y $l_h = K$ fijo; entonces Δ_{l_i}es el conjunto de todas las coberturas correspondientes a las entradas del arreglo A_i.

Ahora consideremos un conjunto cuyo tamaño sea igual a una potencia de 2, por ejemplo, el siguiente conjunto S de tamaño igual a $2^3 = 8$: $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Ahora consideremos los 2 subconjuntos de S formados uno por la mitad izquierda de los elementos de S y el otro por la mitad derecha de los elementos de S. Estos dos subconjuntos son de la mitad del tamaño de S, es decir $2^2 = 4$. Apliquemos este procedimiento de forma recursiva con los nuevos subconjuntos resultantes a cada paso, hasta obtener conjuntos de un único elemento. Los subconjuntos resultantes se muestran a continuación:

De tamaño $2^2 = 4$: $\{0, 1, 2, 3\}, \{4, 5, 6, 7\}$

De tamaño $2^1 = 2$: $\{0, 1\}, \{2, 3\}, \{4, 5\}, \{6, 7\}$

De tamaño $2^0 = 1$: $\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$

Encontremos las diferentes particiones posibles del conjunto S, que es de tamaño igual a una potencia de 2, con los subconjuntos que acabamos de obtener de tamaño igual también a una potencia de 2.

Para obtener las particiones posibles del conjunto S de tamaño 8, comencemos primero por ver cuántas particiones posibles hay para los subconjuntos de tamaño 1. Por ejemplo, para el subconjunto $\{2\}$ hay sólo una partición posible que es la trivial, es decir $\{2\}$. Es claro que para cualquier conjunto de tamaño 1, sólo hay una partición posible.

Ahora veamos cuántas particiones posibles hay para un subconjunto de tamaño 2. Por ejemplo, para el subconjunto $\{0, 1\}$ hay 2 particiones posibles: $\{\{0\}, \{1\}\}$ y la trivial $\{0, 1\}$. Y para el subconjunto $\{2, 3\}$ hay también 2 particiones posibles: $\{\{2\}, \{3\}\}$ y la

trivial $\{2,3\}$. Es claro también que para cualquier conjunto de tamaño 2, sólo hay dos particiones posibles.

Ahora veamos cuantas particiones posibles hay para los subconjuntos de tamaño 4. Por ejemplo $\{0,1,2,3\}$. Este subconjunto lo podemos ver como la unión de 2 subconjuntos de tamaño 2; es decir, $\{0,1,2,3\} = \{0,1\} \bigcup \{2,3\}$. Sabemos que cada uno de estos 2 subconjuntos tiene 2 particiones posibles, así que el número de particiones posibles del subconjunto de tamaño 4 sería la combinación de las diferentes particiones de las de tamaño 2; es decir $2 \times 2 = 4$. Y las 4 particiones son:

$\{\{0\},\{1\},\{2\},\{3\}\}$

$\{\{0\},\{1\},\{2,3\}\}$

$\{\{0,1\},\{2\},\{3\}\}$

$\{\{0,1\},\{2,3\}\}$

Y por último también tenemos que considerar la partición trivial, es decir:

$\{0,1,2,3\}$

Entonces el número total de particiones posibles para este subconjunto de tamaño 4 es 5. Y lo mismo sucede para el subconjunto $\{4,5,6,7\}$; es decir, el número de particiones posibles es 5 también. Es claro que para cualquier conjunto de tamaño 4, habrá 5 particiones posibles (con subconjuntos de tamaño igual a una potencia de 2).

Podemos ahora inferir entonces que el número de particiones para un subconjunto de tamaño 8 (que es la unión de 2 subconjuntos de tamaño 4) será igual a $5 \times 5 + 1 = 26$, donde el sumando 1 corresponde a la partición trivial.

En general, podemos obtener entonces el número de particiones posibles para un conjunto de tamaño 2^q (con $q = 0, 1, 2...$) cuyos subconjuntos sean de tamaño igual a una potencia de 2 y que se obtienen de forma recursiva al dividir cada subconjunto resultante en mitad izquierda y mitad derecha, mediante la siguiente relación de recurrencia: $p_q = (p_{q-1})^2 + 1$ con $p_0 = 1$. Donde, p_q es el número de posibles particiones para el conjunto de tamaño 2^q. De manera práctica y como se verá enseguida, el conjunto de tamaño 2^q en realidad el vector de bits de tamaño 2^Kbits y existen p_q patrones de bits de tamaño q bits. Veamos ahora cómo podemos aplicar este resultado en nuestro esquema para determinar los patrones de bits posibles en el vector de bits. Recordemos que tenemos refinamientos sucesivos; es decir, $\Delta_K \prec \Delta_{K-1} \prec \Delta_{K-2} \prec ... \prec \Delta_2 \prec \Delta_1$.

Sea $\prod$ el conjunto de todas las coberturas correspondientes a los prefijos de nuestro *trie* binario. Sea δ una cobertura correspondiente a una entrada del arreglo A_{h-1}; es decir, $\delta \in \Delta_{l_{h-1}}$. Y sea $\pi \in \prod$ la cobertura de un prefijo. Sabemos que π es también necesariamente elemento de algún Δ_j; esto es $\pi \in \Delta_j$, para algún $j \in \{1, 2, ...K\}$.

Tenemos dos casos posibles entre π y δ. Uno que $\pi \subset \delta$ y otro que $\delta \subset \pi$. Hay un tercer caso; es decir cuando $\delta \cap \pi \neq \emptyset$, pero este caso no nos interesa pues no tiene implicaciones en los patrones de bits posibles de la cobertura δ.

Analicemos el caso cuando $\delta \subset \pi$. Si esto sucede entonces $j < l_{h-1}$, además esto quiere decir que π contiene varias coberturas de $\Delta_{l_{h-1}}$, entre ellas δ. Y también podemos afirmar que no existe cobertura de prefijo que sea subconjunto de δ; es decir, no existe $p \in \prod$ tal que $p \subset \delta$. Esto es porque $\delta \cap \pi \neq \emptyset$ para todo $p \neq \delta$, ya que los prefijos son disjuntos; y como $p \subset \delta$ por lo tanto $\pi \cap \delta \neq \emptyset$ también y entonces $p \not\subset \delta$. Esto quiere decir que no contiene coberturas de prefijos y como cada cobertura de prefijo comienza con un bit 1, entonces el patrón de bits de δ en general estará constituido totalmente de bits 0. La única excepción es si δ es exactamente la primera cobertura de todas las coberturas de $\Delta_{l_{h-1}}$ contenidas en π; aquí el patrón de bits de δ sería un bit 1 seguido de los restantes bits 0. El bit 1 sería el correspondiente al comienzo de la cobertura del prefijo π. En conclusión, si $\delta \subset \pi$ donde $\delta \in \Delta_{l_{h-1}}$, $\pi \in \prod$ y $\pi \in \Delta_j$, entonces $j < l_{h-1}$ y los únicos dos patrones de bits posibles para la cobertura δ son: todos los bits 0, o bien el primer bit es 1 y los restantes bits 0.

Analicemos ahora el caso cuando $\pi \subset \delta$. Si esto sucede entonces $j \geq l_{h-1}$. Además, existirá al menos un bit 1 en el patrón de bits de la cobertura δ, el bit 1 correspondiente al inicio de la cobertura de prefijo π. Pero en general, la cobertura δ tendrá varios bits puestos a 1, además del de π. Es decir, generalmente habrá varias coberturas de prefijos que serán subconjuntos de δ. Veamos esto con más detalle. Como $\delta \in \Delta_{l_{h-1}}$, entonces la cobertura δ es un conjunto de $2^{K-l_{h-1}}$ bits. Esta cobertura δ la podemos particionar de varias maneras usando coberturas de las particiones $\Delta_K, \Delta_{K-1}, \Delta_{K-2}, ...\Delta_{l_{h-1}}$. Y todas estas coberturas son potenciales coberturas de prefijo que estarían incluidas en δ. Notemos que las coberturas de estas particiones son de tamaño igual a una potencia de 2. Las Δ_K son de tamaño 1, las Δ_{K-2} son de tamaño 2, las Δ_{K-4} son de tamaño 4, etc. Hasta llegar

a las de $\Delta_{l_{h-1}}$ que son de tamaño $2^{K-l_{h-1}}$.

Es claro ahora que podemos aplicar a nuestra cobertura δ el resultado de nuestro anterior análisis sobre las diferentes particiones posibles de un conjunto de tamaño igual a una potencia de 2.

Por ejemplo, ya vimos que el conjunto de tamaño 4 $\{0, 1, 2, 3\}$ tiene 5 particiones posibles:

$\{\{0\}, \{1\}, \{2\}, \{3\}\}$
$\{\{0\}, \{1\}, \{2, 3\}\}$
$\{\{0, 1\}, \{2\}, \{3\}\}$
$\{\{0, 1\}, \{2, 3\}\}$
$\{0, 1, 2, 3\}$

En nuestro caso diríamos que una cobertura de tamaño 4 puede estar formada por 4 coberturas de tamaño 1, o bien 2 coberturas de tamaño 1 y una de tamaño 2, o bien una cobertura de tamaño 2 y 2 de tamaño 1, o bien 2 coberturas de tamaño 2, o bien por una cobertura de tamaño 4. Por otro lado, sabemos que en una cobertura de prefijo el primer bit es 1 y los demás bits son 0. Así que una cobertura de prefijo de tamaño 1 se representa en el vector de bits por 1_b, una de tamaño 2 por 10_b, una de tamaño 4 por 1000_b, etc. También es claro que una cobertura de tamaño 4 puede estar formada por 2 coberturas de prefijo de tamaño 2, esto quedaría representado en el vector de bits por el patrón 1010. Si en cambio, estuviera formada por 4 coberturas de prefijo de tamaño 1, esto quedaría representado en el vector de bits por el patrón 1111. Así, a cada una de las posibles particiones le corresponde un patrón de bits en el vector de bits. En la siguiente tabla mostramos cómo quedarían representadas en el vector de bits cada una de las posibles particiones:

$\{\{0\}, \{1\}, \{2\}, \{3\}\}$	1111
$\{\{0\}, \{1\}, \{2, 3\}\}$	1110
$\{\{0, 1\}, \{2\}, \{3\}\}$	1011
$\{\{0, 1\}, \{2, \}\}$	1010
$\{0, 1, 2, 3\}$	1000

Notemos que el valor de los elementos de los conjuntos no interesa, lo único que cuenta es el tamaño de los subconjuntos y el número de ellos; es por esto que podemos aplicar

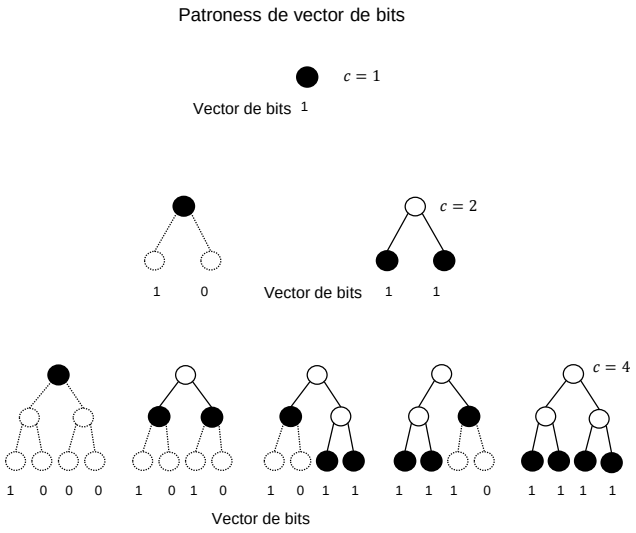


Figura 4.10: Patrones factibles del vector de bits

el mismo análisis a nuestras coberturas que son conjuntos de bits. Notemos también que aquí ningún patrón de bits comienza con 0 pues la representación de un subconjunto (cobertura) siempre comienza con 1; poner un 0 al inicio sería como incluir un subconjunto incompleto en la partición.

Podemos decir entonces que los patrones de bits posibles corresponden a las particiones posibles. Podemos obtener entonces el número de patrones de bits posibles para una cobertura de tamaño 2^q mediante la relación de recurrencia que ya habíamos obtenido para las particiones de un conjunto de tamaño igual a una potencia de 2: $p_q = (p_{q-1})^2 + 1$ con $p_0 = 1$.

Resumiendo, el número de patrones de bits posibles para una cobertura δ de tamaño 2^q son los dados por la relación de recurrencia anterior más el patrón de bits en donde todos los bits son 0; es decir, $p_q + 1$. Notemos que el patrón de bits con un bit 1 seguido de los restantes bits a 0 ya está incluido en la relación de recurrencia y puede significar 2 cosas: que $\delta = \pi$ o que $\delta \subset \pi$ y en este caso δ es la primera de todas las coberturas de $\Delta_{l_{h-1}}$contenidas en π (ver figura 4.10).

Veamos ahora cómo podemos deshacernos del extenso arreglo A_h.

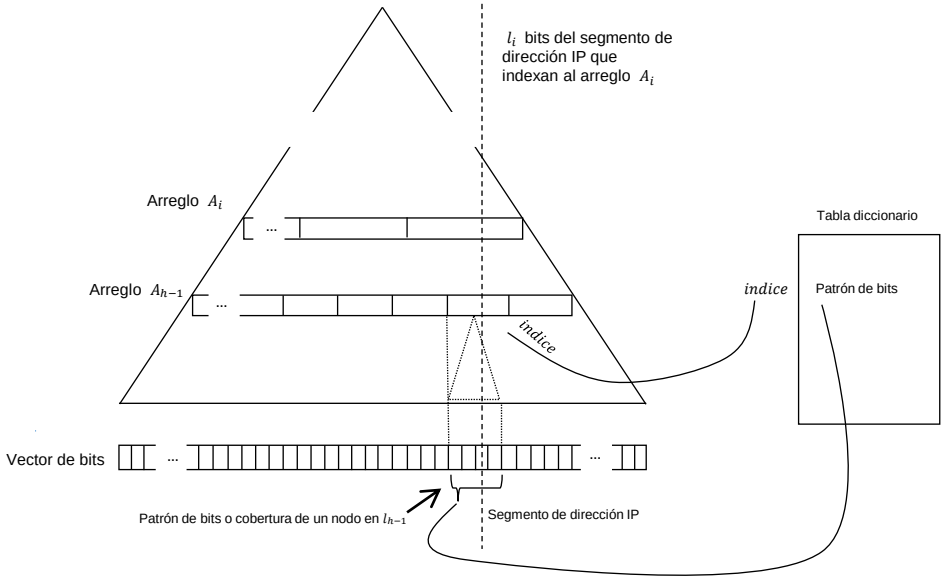


Figura 4.11: Apuntador a un patrón de bits.

Sea $S = |\delta|$ con $\delta \in \Delta_{l_{h-1}}$; es decir, S es el tamaño de la cobertura de cualquiera de las entradas del arreglo A_{h-1}. Sea $q = l_h - l_{h-1} = K - l_{h-1}$, entonces $S = 2^q$ con $1 \leq q \leq K-1$, pues $l_{h-1} \in \{1, 2, ..., k-1\}$.

Como habíamos dicho, aprovechando que no todos los patrones de bits son posibles, podemos substituir el extenso arreglo A_h por una tabla diccionario que contendrá las sucesiones de CPI de únicamente los patrones posibles de bits.

Esta tabla diccionario tendrá $p_q + 1$ renglones, donde $q = K - l_{h-1}$; ya que cada renglón de esta tabla corresponderá a uno de los posibles patrones de bits de las coberturas de $\Delta_{l_{h-1}}$. Y esta tabla diccionario tendrá $S = 2^q$ columnas, pues cada sucesión correspondiente a cada patrón de bits tiene CPI.

Por otro lado, cada entrada del arreglo A_{h-1} tendrá, además de su CPI, un índice que servirá de llave al renglón en esta tabla diccionario con la lista de S CPI asociada a la cobertura δ de dicha entrada; en lugar de que S entradas del arreglo A_h tuvieran estas CPI.

Como el número de renglones de la tabla diccionarios es $P_q + 1 = P_{K-l_{h-1}} + 1$, entonces este índice es de tamaño $\lceil log_2(p_{K-l_{h-1}} + 1) \rceil$bits.

En la figura 4.11 podemos observar la nueva estructura del arreglo A_{h-1} en donde, a diferencia de la figura 4.9, cada entrada tiene además de su CPI, un índice para acceder al renglón correspondiente al patrón de bits de su cobertura en la tabla diccionario. El esquema de *Lulea* utiliza una tabla que llaman "*map*table" que sería equivalente a nuestra tabla diccionario con un $q = 4$ y por tanto $s = 16$. En nuestro esquema generalizamos esta idea para cualquier q. La originalidad de nuestra propuesta es que determinamos formalmente y no empíricamente, como es el caso de *Lulea*, los parámetros para minimizar el consumo de memoria del esquema de búsqueda del prefijo de mayor coincidencia. Analicemos pues dicho consumo de memoria en nuestro nuevo esquema que incluye una tabla diccionario.

Recordemos que del conjunto de niveles de nuestro *trie* $T = \{1, 2, ..., K\}$, seleccionamos h de estos K niveles para definir el conjunto de niveles $L_h = \{l_1, l_2, ..., l_h\}$, con $l_i \in T$ y $l_h = K$ fijo, donde $2 \leq h \leq K$.

Recordemos también que definimos a la escala de niveles $(l_i)_{1 \leq i \leq h}$ como la sucesión de los elementos de L_h ordenados del menor al mayor; es decir $(l_i)_{1 \leq i \leq h} = l_1, l_2, ..., l_h$ con $l_i \in L_h$, donde $\forall i \;\; l_{i-1} < l_i$ con $l_0 = 0$. En nuestro anterior esquema, por cada elemento l_i de la escala de niveles $(l_i)_{1 \leq i \leq h}$, definimos un arreglo A_i que almacena las cuentas de bits 1 precalculadas CPI.

En el esquema que acabamos de presentar en esta sección, hay un arreglo menos. Es decir, ahora tenemos un arreglo A_i por cada elemento l_i de la sucesión $(l_i)_{1 \leq i \leq h}$; y en lugar del extenso arreglo A_h tenemos una tabla diccionario.

Calculemos ahora el costo en memoria de este nuevo esquema. Este costo en bits está dado por la ecuación (4.5), donde el subíndice D hace alusión a que este nuevo esquema usa una tabla diccionario. El primer sumando corresponde al costo de almacenar las CPI en los $h - 1$ arreglos A_i. El segundo sumando corresponde al costo de almacenar los índices en el arreglo A_{h-1} que apuntan a la tabla diccionario. Y finalmente el tercer sumando corresponde al costo de almacenar las secuencias de CPI en la tabla diccionario; una tabla $P_{K-l_{h-1}} + 1$ con renglones, $2^{K-l_{h-1}}$columnas y entradas de $K - l_{h-1}$ bits cada una donde se almacenan las CPI. Nótese que para cada uno de estos sumandos se indica entre paréntesis de qué es función cada uno.

$$M_D((l_i)_{1\leq i\leq h-1}) = m_A((l_i)_{1\leq i\leq h-1}) + m_{ix}(K, l_{h-1}) + m_d(K - L_{h-1})$$

$$m_A((l_i)_{1\leq i\leq h-1}) = \sum_{i=1}^{h-1}[2^{l_i} \times (K - l_{i-1})]$$

$$m_{ix}(K, l_{h-1}) = 2^{l_{h-1}}\lceil log_2[p_{K-l_{h-1}} + 1])\rceil$$

$$m_d(K - L_{h-1}) = (2^{K-l_{h-1}})(p_{K-l_{h-1}} + 1)(K - l_{h-1})$$

$$donde\ \ 2 \leq h \leq K\ \ y\ \ l_0 = 0\ \ y\ \ p_q = (p_{q-1})^2 + 1\ \ p_0 = 1 \tag{4.5}$$

Nuevamente, para un valor de K dado, nos interesa encontrar la sucesión $(l_i)_{1\leq i\leq h-1}$ (con $h = 2, 3, ..., K$) que minimice el consumo de memoria requerida para almacenar $h-1$ arreglos A_i y la tabla diccionario. Denotemos con $(\mu_i)_{1\leq i\leq h}$ a tal sucesión. Entonces, tenemos que encontrar $(\mu_i)_{1\leq i\leq h}$ tal que:

$$M_D((\mu_i)_{1\leq i\leq h}) = \min_{\substack{(l_i)_{1\leq i\leq h-1} \\ 2\leq h\leq K}} (m_A((l_i)_{1\leq i\leq h-1}) + m_{ix}(K, l_{h-1}) + m_d(K - L_{h-1}))$$

Nuevamente como en el esquema anterior, el número de casos que tenemos que evaluar para determinar dicho mínimo es $2^K - K - 1$; y así poder determinar la sucesión buscada $(\mu_i)_{1\leq i\leq h}$. En el capítulo 5 mostramos los resultados obtenidos utilizando este análisis que acabamos de desarrollar. Es decir, el parámetro K lo vamos a variar desde 2 hasta un máximo de 32 que es el tamaño de las direcciones IPv4.

Para tener una idea del ahorro en memoria de nuestro nuevo esquema podemos comparar la memoria que se necesita en el esquema de la sección anterior y en este nuevo para los arreglos A_h, A_{h-1} y la tabla diccionario.

El esquema anterior usaba $2^K(K - l_{h-1})$ bits para el arreglo A_h. Podemos compararlo con los dos últimos sumandos de la ecuación (4.5) si escogemos un valor de K y variamos l_{h-1}, y luego hacemos esto para diferentes valores de K. Por ejemplo, si tomamos los

valores que utilizó *Lulea* $K = 16$ con $L_3 = \{l_1, l_2, l_3\} = \{10, 12, 16\}$, el tamaño en bits del arreglo $A_h = 2^{16}(16 - 12) = 31.8$KB pero si lo substituimos por la tabla diccionario el costo de la misma es de $2^{16-12}(p_{16-12}+1)(16-12) = 5.2$KB con apuntadores en el arreglo A_{h-1} de tamaño $\lceil log_2[p_4 + 1])\rceil = \lceil log_2[678])\rceil = 10$ bits.

Es de suma importancia hacer notar que la tabla diccionario crece de forma cuadrática de acuerdo al distanciamiento entre K y l_{h-1}; si en el ejemplo anterior $l_{h-1} = l_2 = 11$ el tamaño de la tabla diccionario sería de 8.7MB. por consiguiente, la tabla diccionario no siempre es de menor tamaño que el arreglo A_h, de hecho si ella lo fuera para todo valor de K y $(l_i)_{1\leq i\leq h}$ podríamos realizar la búsqueda LPM utilizando solo a ella. De esta manera, una vez más recalcamos la importancia de elegir los parámetros adecuados que minimicen los costos en memoria de nuestro esquema.

La ecuación (4.6) muestra la función que da el número de accesos a memoria necesarios en este nuevo esquema para realizar una búsqueda del prefijo de mayor coincidencia para una dirección IP dada de longitud K. La ecuación consiste en $h - 1$ accesos a memoria correspondientes a los $h-1$ arreglos $A_i \quad i = 1, 2, ..., h-1$, un acceso a memoria a la tabla diccionario y finalmente un acceso a la tabla extendida de prefijos disjuntos.

$$I_D((l_i)_{1\leq i\leq h}) = h + 1 \tag{4.6}$$

La figura 4.12 muestra un ejemplo completo de la estructura de datos para la tabla de encaminamiento de la figura 4.2; en este ejemplo $h = 3$ y $L_3 = \{l_1, l_2, l_3\} = \{1, 2, 4\}$. Como $q = K - l_{h-1}$, entonces $q = 4 - 2 = 2$; y como $S = 2^q$, entonces $S = 4$.

La figura 4.13 muestra parte de un ejemplo más elaborado. Para este ejemplo se consideran los siguientes parámetros: $K = 8$, $L_4 = \{l_1, l_2, l_3, l_4\} = \{2, 4, 6, 8\}$ y por lo tanto $q = 8 - 6 = 2$; y como $S = 2^q$, entonces $S = 4$. Note que en este ejemplo y en el de la figura anterior se utiliza la misma tabla diccionario, ya que esta tabla sólo depende del valor de q y en ambos ejemplos $q = 2$.

La figura 4.14 muestra un ejemplo de búsqueda del prefijo de mayor coincidencia utilizando el ejemplo de la figura 4.12. Nótese las diferentes secciones en que se divide la dirección IP y cómo estas secciones son utilizadas para acceder a los arreglos contadores y a la tabla diccionario. En este ejemplo, el resultado de sumar los correspondientes CPI es

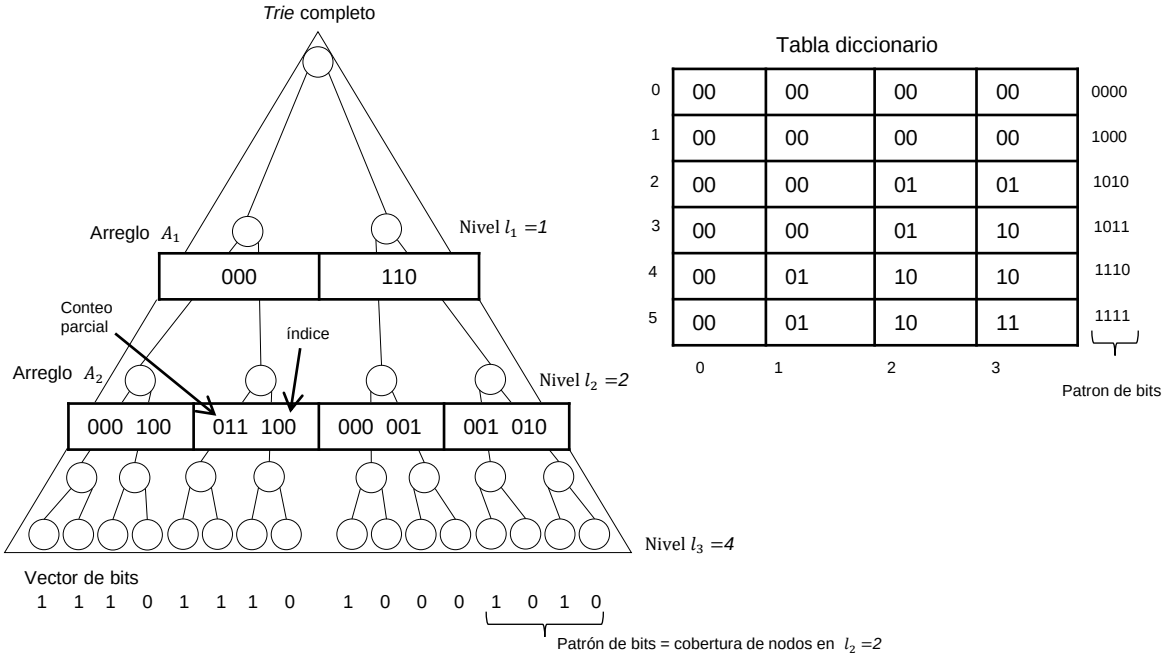


Figura 4.12: Estructuras de datos completa del ejemplo.

4. De esta forma, se puede ver en la figura 4.2 que el índice 4 corresponde al NHI llamado en el ejemplo AP_5.

4.2.3. Optimización en memoria de esquema general

El esquema general de conteos precalculados y tabla diccionario tiene como objetivo reducir el consumo en memoria comparado con el esquema que solo utiliza un arreglo contador. Recuerde que, si fijamos h conteos precalculados al vector de bits, éstos se asocian a niveles l_i del *trie* binario de profundidad K dados por un conjunto $(l_i)_{1\leq i\leq h} = l_1, l_2, ..., l_i, ..., l_h$. Este conjunto $(l_i)_{1\leq i\leq h}$ constituye una configuración de los niveles en el *trie*. Cada configuración tiene un costo en bits dado por la ecuación (4.3) o bien un costo dado por la ecuación (4.5) si se utiliza la tabla diccionario. Podemos analizar las ecuaciones para determinar cuál es la sucesión $(\mu_i)_{1\leq i\leq h}$ que minimiza el costo en memoria. El total de configuraciones pertenecen a un problema combinatorio con $\frac{(K-1)!}{(h-1)!(K-h)!}$ posibles combinaciones, siendo K la longitud de segmentos de direcciones IP

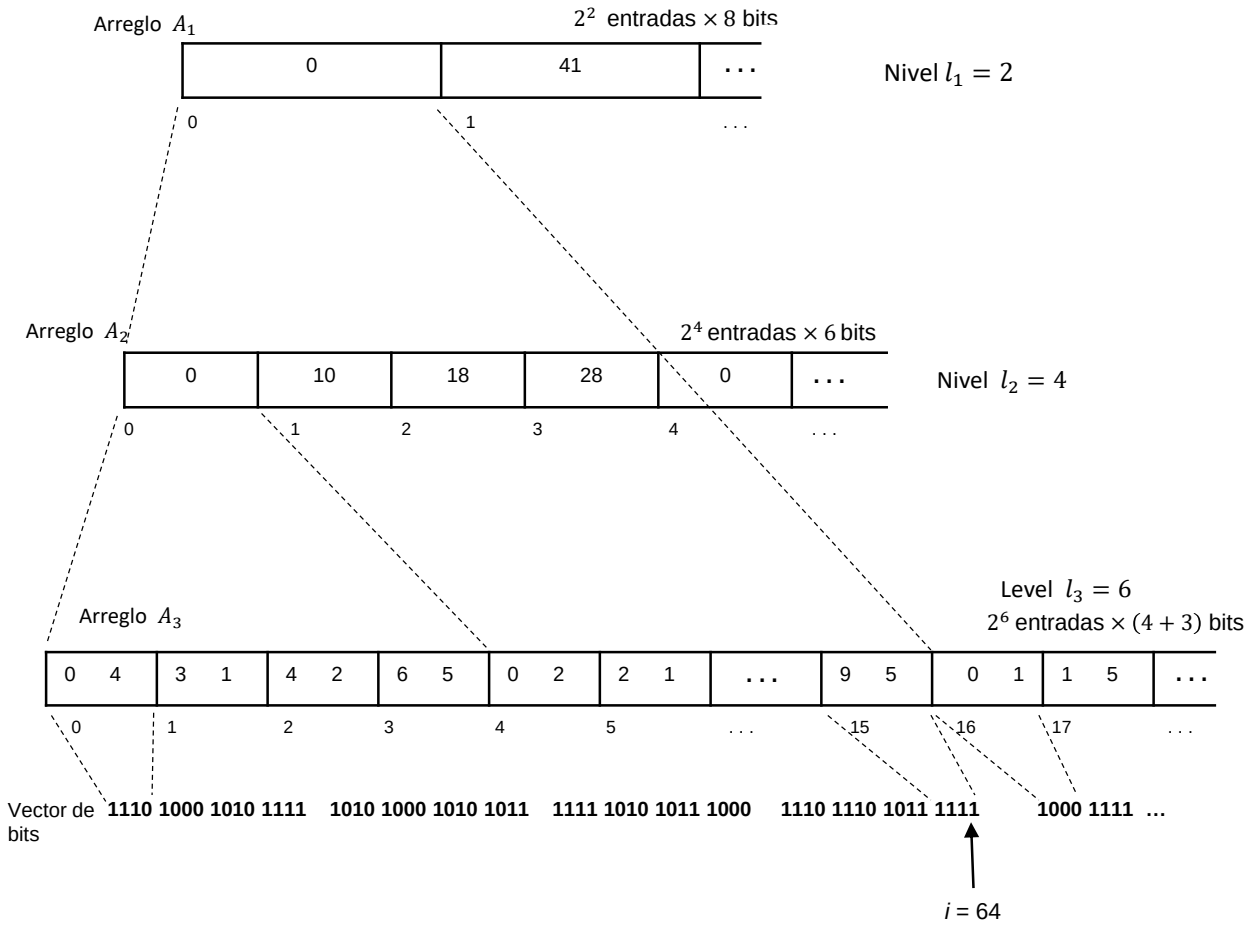


Figura 4.13: Arreglos contadores del ejemplo: $h = 4$, $K = 8$, y $L = \{l_1 = 2, l_2 = 4, l_3 = 6, l_4 = 8\}$.

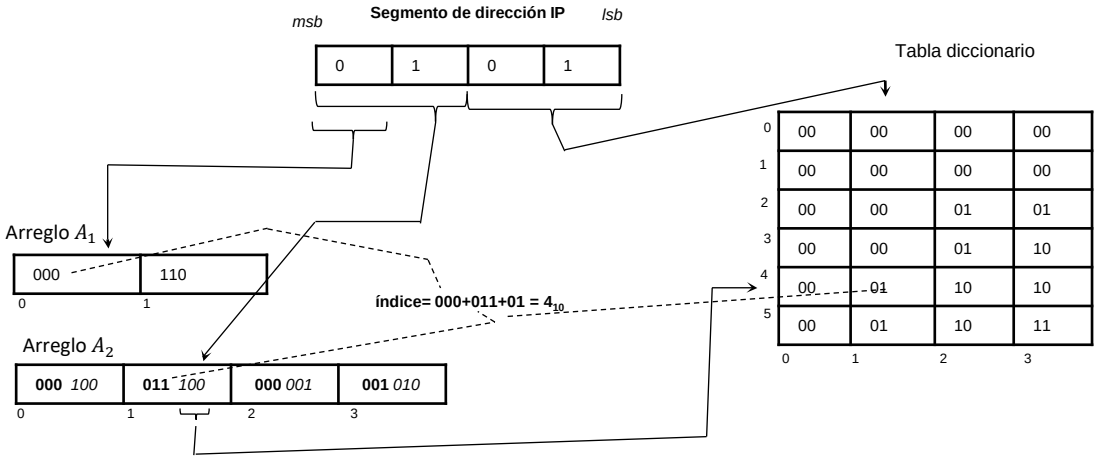


Figura 4.14: Ejemplo de búsqueda del prefijo de mayor coincidencia con el esquema que usa una tabla diccionario.

y h el número de conteos en el vector de bits.

Nosotros determinamos $(\mu_i)_{1\leq i\leq h}$ para todos los posibles valores de K y de h evaluando todas y cada una de las combinaciones. La ecuación (4.7) muestra las posibles combinaciones que tuvieron que ser comprobadas, donde el límite superior de la sumatoria es |IP| es 32 o 128 para los protocolos IPv4 o IPv6 respectivamente. Tenga en cuenta que la función en memoria dada por las ecuaciones 4.5 y 4.3 no dependen de la distribución ni número de prefijos en la tabla extendida, esto significa que, los valores en consumo de memoria son constantes y que por lo tanto, comprobar todas las posibles combinaciones de configuraciones de niveles, solamente se tiene que realizar una única vez para determinar que valores de $(l_i)_{1\leq i\leq h}$ son los que optimizan al máximo el consumo de memoria. Observe que dado que la ecuación es una doble sumatoria de una combinación sin repetición, el tiempo para calcular todas las posibles combinaciones de niveles se puede extender a horas o inclusive a días. Sin embargo, por las razones antes mencionadas, obtenemos por fuerza bruta los mínimos globales para una configuración de h conteos precalculados en

un *trie* de profundidad K.

$$\sum_{K=2}^{|IP|}\sum_{h=2}^{K}\frac{(K-1)!}{(h-1)!(K-h)!} \tag{4.7}$$

4.2.4. Algoritmos

4.2.4.1. Algoritmo de llenado de arreglos

Para llenar los arreglos contadores A_1,A_2,...,A_{h-1} se necesitan: el vector de bits, el parámetro K y el arreglo ordenado de los h niveles en el *trie* $(l_i)_{1\leq i\leq h} = \{l_1, l_2, ..., l_i, ..., l_h\}$. El algoritmo 4.1 muestra el procedimiento para llenar los $h-1$ arreglos contadores. El algoritmo no considera el llenado de la tabla diccionario ya que es una estructura de datos constante; es decir, sus valores nunca cambian aunque haya cambios en los prefijos en la tabla extendida de prefijos. Tenga en cuenta que, el *trie* no es necesario más que, como auxiliar para explicar nuestra idea.

El procedimiento es el siguiente: se obtiene la longitud de la cobertura de los nodos en el nivel l_{h-1}, esta longitud se utiliza para realizar un recorrido en el vector de bits en rondas de $2^{K-l_{h-1}}$ bits. En cada ronda se obtiene, en orden, a cada nodo (ancestro escalón de los nodos en $l_h = K$) en el nivel l_{h-1} (utilizando la variable i) y se cuenta el número de bits puestos en 1 su cobertura en cada ronda (Procedimiento *count_ones*). En este mismo paso se obtiene el apuntador correspondiente a la tabla diccionario (procedimiento *get_row*). El apuntador a la tabla diccionario se coloca en la entrada $A_{h-1}[node-1]$ tal que la variable *node* es la correspondencia con los ancestros escalones en el recorrido dado por las rondas de $2^{K-l_{h-1}}$ bits; simultáneamente desde el nivel l_{h-1} hasta el nivel l_1 (utilizando la variable de control Arr) se realizan un procedimiento para si es posible, actualizar los CPI en los arreglos correspondientes a los niveles ($A_{Arr}[node]$); si *node* y *node-1* en el nivel A_{Arr} significa que pertenecen a diferentes ancestros escalones (nivel l_{Arr-1}), entonces el conteo parcial de las CPI es actualizado con el número de bits puestos del ancestro escalón anterior más el conteo de unos en la ronda correspondiente ($pc = A_{Arr}[node-1] + pc$); seguido, la entrada del arreglo $A_{Arr}[node]$ obtienen el valor de cero y el algoritmo continuá actualizando pero ahora en el arreglo A_{Arr-1}; en otro caso, si *node* y *node*-1 están dentro de la misma cobertura de su ancestro escalón, el arreglo $A_{Arr}[node]$ obtiene el número

de bits puestos en 1 de la cobertura del ancestro escalón correspondiente al grupo de bits de la ronda que se está analizando ($A_{Arr}[node] = A_{Arr}[node - 1] + pc$). El algoritmo para llenar los $h - 1$ arreglos contadores tiene una complejidad de ejecución $O(2^K)$, esto significa que la complejidad equivale a realizar un solo recorrido bit a bit en el vector de bits.

Usamos la figura 4.13 como ejemplo para ilustrar el algoritmo 4.1, tal que, los segmentos de dirección IP son de 8 bits ($K = 8$) y el conjunto $(l_i)_{1 \leq i \leq 4} = \{l_1 = 2, l_2 = 4, l_3 = 6, l_4 = 8\}$ para distribuir los conteos en el vector de bits, en otras palabras los el *trie* binario es seccionado en los niveles 2, 4, 6 y 8. En el ejemplo hay tres arreglos contadores (A_1, A_2 y A_3) correspondientes a los niveles 2, 4 y 6, junto con sus respectivos tamaños. Para mostrar la idea principal de algoritmo explicamos la ronda número 16 de los grupos de 4 bits. Cuando la variable i obtiene el valor de 64 (64/4 = ronda 16), el conteo parcial obtiene el valor de 4 ($pc = 4$, patrón de bits 1111), el patrón de bits corresponde al quinto renglón de la tabla diccionario entonces el apuntador se coloca en $A_3[15]$ y comienza el proceso de actualización de los CPI que van desde el arreglo A_3 hasta el arreglo A_1. Justo en $i = 64$, tanto A_2 como A_1 tienen un ancestro escalón diferente al que tuviesen en $i = 63$, por lo tanto $A_3[16]$ y $A_2[4]$ se fija su valor en 0 mientras que $A_1[1]$ obtiene el total de bits puestos en 1 a la izquierda de su cobertura.

4.2.4.2. Algoritmo de búsqueda LPM

Los arreglos contadores, la tabla diccionario, la tabla extendida de prefijos disjuntos y el conjunto $(l_i)_{1 \leq i \leq h}$ es lo único que se necesita para realizar búsquedas LPM. El vector de bits, el *trie* de prefijos disjuntos y demás estructuras de datos son descartadas. El algoritmo 4.2 muestra el procedimiento general para obtener el índice del prefijo con mayor coincidencia en la tabla extendida de prefijos disjuntos. El segmento de dirección IP se divide en $h - 1$ partes desde el bit más significativo hasta el bit menos significativo de la dirección IP de acuerdo con los niveles l_i correspondientes. Cada parte de la dirección apunta a un arreglo A_i. Recuerde que el penúltimo arreglo l_{h-1} además de un conteo parcial, también contiene un apuntador a un patrón de bits en la tabla diccionario. Para acceder a una columna de la tabla diccionario se utilizan los $K - l_{h-1}$ bits menos

Algoritmo 4.1 Algoritmo de llenado de arreglos.

INPUT: Serie $(l_i)_{1 \leq i \leq h}$
OUTPUT: arreglos llenos

PROCEDIMIENTO llenar_arreglos ()
// se fijan los valores default en los arreglos
for (i = 1 to $h-1$) **do**
A_i [0] = 0
end for
// se obtiene el tamaño de las rondas
$size = 2^{K-l_{h-1}}$
// se recorre el bit-vector las rondas
for (i=$size$ to $2^K - size$ step $size$) **do**
// se obtiene el CPI
pc = count_ones (i-$size$, i-1)
// se obtiene el apuntador a la tabla diccionario
row = get_row(i-$size$, i-1)
set the matrix row index in A_{h-1}[(i/$size$)-1]
// se actualiza el arreglo
$Arr = h-1$
round_finished=FALSE
repeat
// se encuentra el nodo correspondiente del arreglo l_{Arr}
node =i/($2^{K-l_{Arr}}$)
// si node-1 & node tienen diferente ancestro escalon?
if (i mod $2^{K-l_{Arr-1}} == 0$) **then**
// se actualiza el CPI
pc=A_{Arr}[node−1]+pc
// se vasian los valores del arreglo
A_{Arr}[node]=0
// y se busca en el siguiente arreglo
$Arr--$
else
// el arreglo obtiene el valor de los CPIs
A_{Arr}[node] = A_{Arr}[node -1] + pc
round_finished=TRUE
end if
until (round_finished==TRUE)
end for
//finalmente se fija el indice a la tabla diccionario
row = get_row($2^K - size$, $2^K - 1$)
set the matrix row index in A_{h-1}[$2^K/size$]

Algoritmo 4.2 Algoritmo de búsqueda LPM.

```
INPUT: Segmento de direccion IP y la Serie (l_i)_{1≤i≤h}
OUTPUT: El indice del NHI

FUNCION busqueda_LPM()
NHI_index = 0
  // se busca en los contadores
for (i=1 to h − 1) do
    // se segmenta la direccion IP
  arr_index = l_i msb bits of the IP address
    // se obtiene el CPI
  NHI_index = NHI_index + A_i[arr_index]
  if (l_i== l_{h−1}) then
      // se obtiene el apuntador a la tabla diccionario
    row = pointer of A_{h−1}[arr_index]
  end if
end for
  // se obtiene el CPI de la tabla diccionario
NHI_index = NHI_index +
CM[row] [the lsb K − l_{h−1}bits of the IP address]
RETURN NHI_index
```

significativos de la dirección IP. Finalmente el índice del prefijo de mayor coincidencia y por lo tanto el resultado de la búsqueda LPM es la suma de los contenido de los arreglos contadores y de la tabla diccionario.

4.3. Particionamiento eficiente del *trie* binario

Se puede reducir la cantidad de memoria necesaria si la búsqueda completa LPM en IPv4/6 se realiza en varias etapas. Para esto, los prefijos del *trie* se empujan hasta las hojas (*leaf-pushing*), después el *trie* binario de prefijos se divide en varios niveles o profundidades, por cada nivel se obtiene un arreglo contador para cada *subtrie*. Cuando un *subtrie* contiene hojas que llevan a prefijos de mayor profundidad, en el vector de bits y en los arreglos contadores estas hojas se tratan de la misma manera que los prefijos reales pero se identifican con una bandera en la tabla extendida y en vez de la NHI hay un apuntador que dirige hacia el *subtrie* de la siguiente etapa que le corresponde. Entonces

la búsqueda LPM se efectúa dividiendo la dirección en varios segmentos y utilizando cada uno de esos segmentos para indexar los arreglos de contadores correspondientes a los *subtries* de cada etapa.

Es necesario hacer la especificación para diferenciar una búsqueda LPM en etapas y hacer conteos parciales en el vector de bits. Un conteo parcial se realiza en una serie de arreglos de conteos precalculados para realizar una búsqueda LPM en un segmento de dirección IP de longitud K, donde K puede ser igual a 32 (IPv4) e inclusive igual a 128 (IPv6); mientras que, la segmentación de una búsqueda LPM en etapas, consiste en dividir el problema en problemas más pequeños en cada etapa, donde la solución de cada problema más pequeño sirve como indicador para el siguiente problema de la siguiente etapa, con ello se busca disminuir los costos de las estructuras de datos utilizadas en cualquier esquema LPM.

Seccionaremos la búsqueda LPM en IPv4/6 en R número de etapas y en cada etapa vamos a utilizar el esquema de un solo conteo precomputado visto en la Sección 4.1.2 como esquema base de búsqueda LPM en segmentos de dirección IP de longitud K tal que, $K \leq R$. Usaremos este esquema de búsqueda LPM por su simpleza. Esta característica permite por un lado, entender de mejor manera la búsqueda LPM en etapas. Por el otro lado y más importante, podemos generalizar el particionamiento eficiente en etapas de la búsqueda LPM. La Fig. 4.15 ilustra las estructuras de datos del esquema completo de búsqueda LPM en IPv4 o IPv6 dividido en R etapas. Esencialmente una búsqueda LPM consiste en dividir a la dirección IP destino del paquete en R segmentos de longitud acorde con número y tamaño de cada etapa definida en el esquema. Cada segmento de dirección indexa a su respectivo arreglo contador de bits y este a su vez indexa a la tabla extendida de prefijos disjuntos y por ende al NHI correspondiente.

En la figura 4.15 se puede apreciar con detalle la forma y tamaño de los arreglos contadores y las tablas extendidas que se utilizarían en una búsqueda. Uno o varios arreglos contadores en la etapa i tienen un costo en bits dados por las ecuación (4.2) (4.3 y 4.5 en el caso que se utilicen esquemas optimizados en memoria); n_i es la longitud del *i-ésimo* segmento de dirección IP ($n_i = K_i - K_{i-1}$). Una tabla extendida en la etapa i contiene prefijos disjuntos que tienen una longitud de K_{i-1} e incluso K_i bits. Cada entrada de la tabla contiene un prefijo y un campo de 32 o 128 (IPv4 o bien IPv6) bits que identifican

al NHI o bien a un apuntador a la siguiente etapa de la búsqueda distinguidas por una bandera de un bit. La cantidad total de memoria requerida para codificar de esta manera la información de encaminamiento es la suma de la memoria necesaria para cada uno de los arreglos contadores junto con sus respectivas tablas extendidas. El costo en memoria depende entonces del número de etapas y de los niveles en el *trie* donde se ubica cada etapa. Determinar el número y ubicación de las etapas para minimizar el costo en memoria es un problema combinatorio (distinto al del problema de optimización de la sección 4.2 visto con anterioridad). Si un *trie* de profundidad K_T (32 o 128) se divide en R etapas de búsqueda $K_1, K_2, ..., K_R = K_T$, (tal que definimos $K_0 = 0$ y corresponde a la raíz) el número de combinaciones posibles es $\frac{K_T!}{(R-1)!\times(K_T-R-1)!}$ donde, R puede en principio tomar cualquier valor entre 1 y K_T. En el caso de IPv4, K_T tiene un valor de 32 mientras que en IPv6 K_T tiene un valor de 128. Entonces por ejemplo, si dividimos la búsqueda LPM en 10 etapas con $K_T = 128$ (IPv6), el número de combinaciones es de alrededor de 19 billones. Probar todas estas combinaciones con un algoritmo determinístico para encontrar la óptima, podría consumir mucho tiempo. En esta sección proponemos un proceso de exploración heurística mediante algoritmos genéticos, para determinar la ubicación de las etapas (los niveles) en el *trie* que optimicen la cantidad de memoria requerida para guardar los arreglos contadores y las tablas extendidas utilizadas en la búsqueda LPM. Note que en el problema de optimización de la sección anterior, la solución es invariante en el tiempo, debido a que, no depende de distribución de prefijos en la tabla de encaminamiento, sin embargo, en este caso, cada vez que la información de encaminamiento se actualiza se debe dar una nueva solución al problema de particionamiento.

4.3.1. Exploración heurística

Como se mencionó en la parte anterior, seccionar de forma optimizada el *trie* binario representa un nuevo problema. Sin embargo, a diferencia de la propuesta que hacemos para codificar un *subtrie*, el particionamiento del mismo depende en su totalidad de la distribución de los prefijos de red en la tabla de encaminamiento; y más aún, se tiene el problema de que la tabla se actualiza en periodos de tiempo muy pequeños, por lo que, para cada actualización se necesita una nueva optimización.

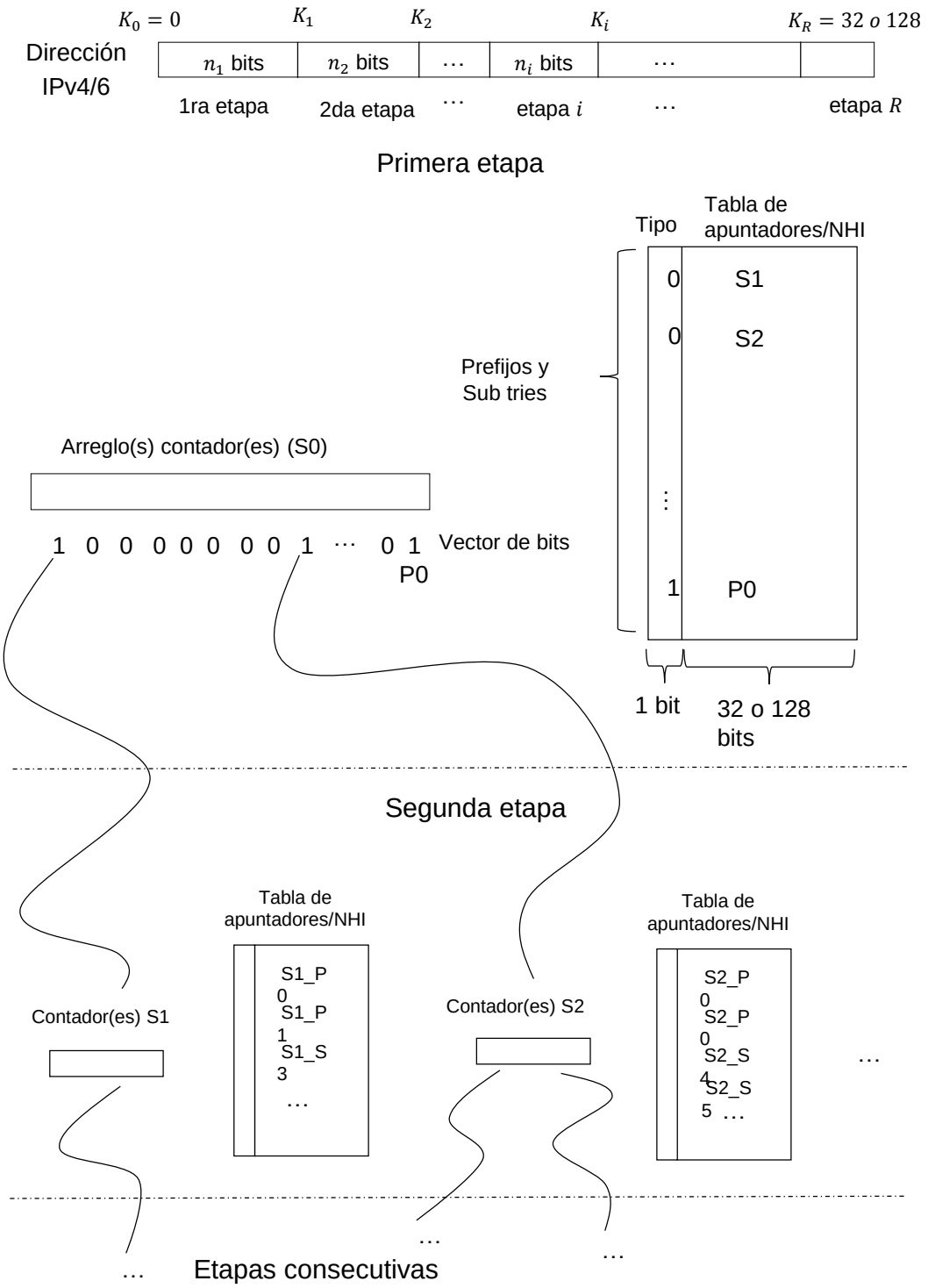


Figura 4.15: Esquema completo en etapas de búsqueda LPM.

Para este problema nosotros descartamos la idea de realizar una búsqueda exhaustiva, consideramos que aunque hay optimizaciones radicales en el desempeño de exploración determinística como lo presenta [18], el tiempo de ejecución es exponencialmente proporcional a la dimensión del problema. Tener la garantía de una solución óptima para particionar el *trie* binario puede no ser tan significativo si en cada unidad de tiempo, después de una actualización, se tenga que comenzar todo el proceso de búsqueda una vez más. En cambio, ahora proponemos realizar exploración heurística utilizando inteligencia artificial, proporcionando una buena solución (así como lo permita la técnica utilizada) que optimice la cantidad de memoria requerida en cada actualización de la tabla. Existen muchas técnicas de exploración heurística que podemos utilizar para este problema, como los son: recocido simulado, algoritmos genéticos, redes neuronales etc. Nosotros consideramos cada una de estas técnicas y determinamos que para este problema los algoritmos genéticos son una posible solución debido a la cantidad de memoria requerida, mapeo directo entre estructuras de datos y desempeño en instrucciones.

4.3.2. Algoritmos genéticos

Los procesos evolutivos ocurren cuando hay una población de individuos que son capaces de reproducirse y existe alguna diferencia entre ellos [19]. El ciclo de la evolución comienza cuando los individuos de una población con mejores capacidades de adaptación, sobreviven al proceso de selección natural, mientras que los individuos más débiles desaparecen. El ciclo continúa cuando parejas de individuos se reproducen heredando sus aptitudes a sus descendientes pero el mecanismo de mutación hace posible la existencia de descendientes con aptitudes diferentes a las de sus ancestros. Los descendientes, resultado de un ciclo de la evolución remplazan a sus ascendentes para volver a comenzar con un nuevo ciclo. Después de haberse repetido muchas veces el ciclo de la evolución, la población resultante tiene mejores cualidades de adaptación que sus ancestros originales.

Los algoritmos genéticos modelan a los procesos evolutivos y están basados en poblaciones cuyos elementos representan soluciones a distintos problemas de optimización que suelen ser difíciles de tratar [20]. El enfoque de algoritmos genéticos que se propone en este trabajo es adecuado por lo siguiente. Las tablas de encaminamiento IPv4/6 son ac-

tualizadas constantemente en intervalos pequeños de tiempo ya sea por cambios de NHI o por inserción-eliminación de prefijos de red. En una actualización, la distribución de prefijos en el *trie* cambia, pero no de forma radical puesto que la topología de Internet no cambia bruscamente de un momento a otro; esto significa que una solución que optimice la memoria requerida para la búsqueda LPM en un momento dado no será extremadamente diferente de la solución óptima después de la actualización. Los algoritmos genéticos permiten hacer uso de una serie de soluciones que en su momento fueron óptimas para usarlas como punto de partida después de que la tabla de encaminamiento haya sido actualizada. Por otro lado, como se verá en seguida, el mapeo entre los niveles del *trie* y una posible solución es directo, lo que implica una mayor eficiencia en instrucciones del algoritmo genético. Puesto que el costo en memoria por codificar un *subtrie* en un arreglo contador es constante, la función de aptitud del genético se reduce a una simple fórmula.

Desde el punto de vista biológico, un cromosoma es una estructura que contiene información genética de un individuo. La evolución es un proceso que opera sobre los cromosomas y no sobre los individuos que ellos codifican. Nosotros simplificamos a un cromosoma como una secuencia de bits que contiene una posible solución al problema de particionamiento. La figura 4.16(a) ilustra cómo un individuo o cromosoma consiste de un arreglo V_c de $K_T + 1$ bits de los cuales existen R bits con un valor de 1 que representan los niveles que forman las particiones del *trie* de profundidad K_T. Dicho de otra manera, un cromosoma es una posible solución válida (válida mas no necesariamente óptima) que se representa con un arreglo de bits, donde un bit puesto en uno en su entrada K_i representa una etapa de búsqueda en el nivel K_i del *trie* binario. El mapeo entre un cromosoma y los niveles en el *trie* es directo: los bits del cromosoma puestos en uno representan las etapas de búsqueda.

Una etapa está delimitada con un 1 en el nivel K_i y otro en el nivel K_{i-1} del *trie*. Ésta tiene un costo en memoria que es la memoria necesaria para los arreglos contadores correspondientes a los *subtries* generados en la profundidad K_{i-1} junto con sus tablas extendidas. Para el cálculo del costo en memoria de un cromosoma hacemos el uso de un arreglo A_p que contiene en la posición j la cuenta de los prefijos que existen en el nivel j y un arreglo A_s que tiene la cuenta del número de *subtries* que existen en cada nivel j (ver la figura 4.16(b)). Los arreglos A_p y A_s tienen tantas entradas como niveles en

tenga el *trie*. La ecuación (4.8) expresa el costo por partir el *trie* binario desde el nivel $K_i = r$ hasta el nivel $K_{i+1} = q$ utilizando los arreglos A_s y A_p. La ecuación consiste en el número de *subtries* generados en el nivel r ($A_s[r]$) y de cada uno de sus arreglos contadores de costo f_m dado por la ecuaciones 4.3 y 4.5. También se considera el costo de las tablas extendidas que consiste en la sumatoria de los prefijos entre los niveles r y q más el número de subtries que se generan en la siguiente etapa ($A_s[q]$) todo esto multiplicado por el tamaño del NHI o el apuntador (|IP|). Entonces el costo total de un cromosoma está dado por los $R + 1$ niveles en el *trie* que forman a las particiones y sus costos $C_{r,q}$ correspondientes.

$$c_{r,q} = (A_s[r] \times f_m + \left(\left(A_s[q] + \sum_{i=p}^{q} A_p[i] \right) \times |IP| \right) \tag{4.8}$$

La figura 4.17 muestra un ejemplo de un ciclo del proceso evolutivo. Suponemos una búsqueda de cinco etapas, la primera etapa comienza siempre en el nivel cero y la última termina siempre en el nivel K así que en cada cromosoma V_i deben existir cuatro entradas (además de la primera y la última) con valor 1. La figura 4.17(a) muestra la población aleatoria o bien previamente optimizada. La figura 4.17(b) ilustra el proceso de selección que se lleva a cabo mediante torneos. En este ejemplo se enfrentan el cromosoma V_1 contra el cromosoma V_2, en este caso el costo de V_1 es menor que el costo de V_2 por lo que el vencedor es V_1 y por lo tanto el contenido de V_1 se copia en V_2 obteniendo así el valor final de V_2. La figura 4.17(c) presenta la simulación de la reproducción sexual entre V_1 y V_3; se elige una posición de cruce aleatoria y se intercambian los valores a la izquierda de dicha posición de ambos cromosomas. Dado que cada cromosoma solo puede tener 4 bits con valor de 1, si al reproducir dos individuos uno adquiere más niveles y otro pierde niveles entonces se equilibran los valores de forma aleatoria. Finalmente la figura 4.17(d) muestra la simulación de la mutación que permite que individuos en una generación posean características únicas no heredas de sus ancestros. En este ejemplo, del cromosoma V_i se elige un bit puesto en 1 y un bit puesto en 0 de forma aleatoria y se intercambian.

Cabe mencionar que los ejemplos expuestos en este capítulo como los presentados en las figuras 4.14 y 4.15, para el caso de la generalización de los conteos precalculados,

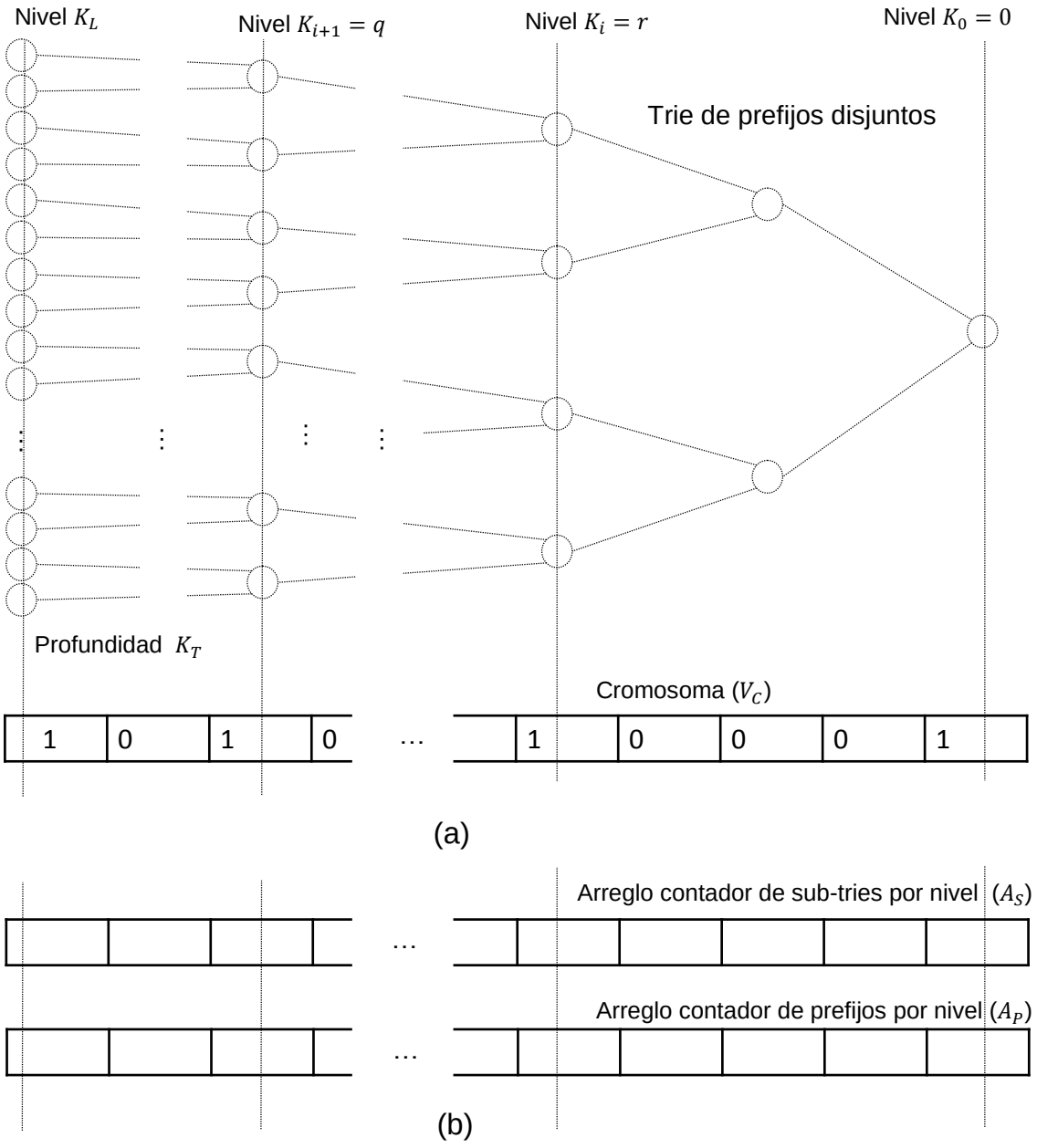
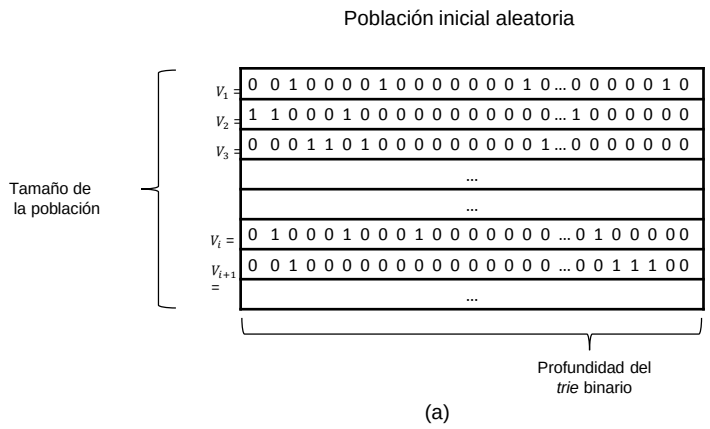
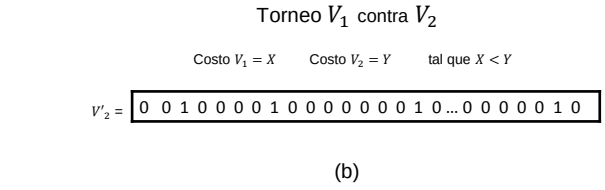


Figura 4.16: (a) Niveles en el *trie* codificados en un cromosoma. (b) Arreglos que guardan la distribución de prefijos y *subtries* por nivel.





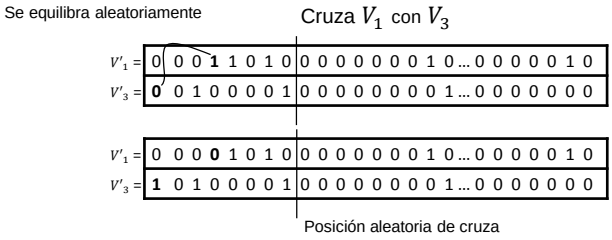


Figura 4.17: (a) Población. (b) Supervivencia del más fuerte mediante torneo. (c) Reproducción sexual. (d) Mutación.

y el ejemplo presentado en la figura 4.17, para el algoritmo genético, son meramente ilustrativos. Los niveles en el *trie* y el número de ellos son las variables que determinan las características de parámetros, arreglos contadores, estructuras de datos, consideraciones adicionales, etc. Sin embargo, el propósito de la generalización de conteos precalculados y partición del *trie* binario tiene como objeto reducir el consumo en memoria y conservar un alto desempeño en número de instrucciones. En consecuencia, en el siguiente capítulo se justifican los parámetros utilizados y por lo tanto se presentan costos y resultados.

Capítulo 5

Resultados

En este capítulo presentamos los resultados de nuestra propuesta así como el análisis de los mismos. En primer lugar presentamos todos los costos optimizados en memoria de la codificación del vector de bits en los esquemas de varios conteos precomputados utilizando o no patrones de bits. Hacemos el análisis de su rendimiento en memoria *vs* rendimiento en instrucciones; esto con el objeto de resaltar la flexibilidad de nuestra propuesta para poder ponderar los mejores parámetros de acuerdo al hardware utilizado. También hacemos el análisis de nuestro segundo aporte de particionamiento en etapas, realizamos experimentos con tablas globales de encaminamiento, considerando con ello, la distribución y número de prefijos.

Antes de comenzar la presentación de resultados es necesario precisar el uso de unidades y la base numérica de las operaciones aritméticas para el consumo de memoria de las estructuras de datos utilizadas en las diferentes partes del documento. Recordemos que un bit es la unidad mínima de información con valores de "0" o bien de"1" y que por definición 8 bits forman un byte. Para cuantificar unidades de almacenamiento utilizamos la base numérica binaria y no la decimal, esto significa por ejemplo que un número binario expresado en decimal es una potencia de su base dos (2^x tal que, $x \in \mathbb{N}$). Por ejemplo un millar de bytes es $2^{10} = 1024$ bytes y no 1000 bytes como se expresaría en base decimal. También se pueden utilizar los prefijos kilo, mega, giga para expresar cantidades grandes. Considere que en capítulo anterior los cálculos y ecuaciones utilizan la medida de bits, sin embargo, en este capítulo los resultados son expresados utilizando las consideraciones

antes descritas.

5.1. Esquema generalizado de conteos distribuidos de un vector de bits para búsquedas LPM

El esquema que proponemos generaliza la idea de conteos distribuidos en arreglos contadores de los valores en el vector de bits. Recuérdese que, todos los costos de memoria para los arreglos son independientes del tamaño de la tabla de encaminamiento y prefijos en ella. A continuación analizamos los costos de memoria y de accesos a la misma utilizando parámetros que puedan servir tanto para IPv4 o bien para IPv6.

5.1.1. Búsqueda mediante un conteo precomputado

Nosotros generalizamos las distribución de CPIs en arreglos contadores; sin embargo, es importante considerar el esquema simple de un solo conteo precomputado en el vector de bits y aunque, no podemos considerarlo como una novedad de nuestra propuesta, sí lo utilizamos como referencia de costos entre las variantes que presentamos.

El esquema de búsqueda LPM que utiliza un solo arreglo precomputado es naturalmente el que posee la mayor optimización en instrucciones de acceso a memoria ya que, con un solo acceso al arreglo contador es posible determinar la posición del NHI en la tabla de encaminamiento. Pero al mismo tiempo, es el más costoso en memoria, ya que según sea la profundidad del *trie* (denotado por K), el costo del arreglo contador sigue un orden exponencial ($K \times 2^K$ bits). El cuadro 5.1 muestra los costos de memoria en MB y el costo de accesos a memoria utilizando un solo arreglo contador de valores precomputados del vector de bits en el intervalo $K \in [16, 32]$ de profundidades del *trie* binario. Del cuadro se puede ver que, debido al costo en memoria de codificar un *trie* de profundidad $K = 16$, es totalmente factible realizar esta codificación en la memoria *cache* de una computadora actual. También, $K = 16$, es la profundidad que se utiliza en *lulea*. Para $K = 32$ el costo excede los 16GB, que bien puede ser contenida en una memoria RAM pero el costo para una profundidad superior no es factible por ahora.

Cuadro 5.1: Costo en memoria e instrucciones de acceso a memoria para *tries* en el intervalo de profundidades $[16, 32]$ utilizando un solo conteo precomputado.

Profundidad del *trie* K	Costo en MB	Instrucciones de acceso a memoria
16	0.125	1
17	0.265625	1
18	0.5625	1
19	1.1875	1
20	2.5	1
21	5.25	1
22	11	1
23	23	1
24	48	1
25	100	1
26	208	1
27	432	1
28	896	1
29	1856	1
30	3840	1
31	7936	1
32	16384	1

5.1.2. Búsqueda mediante varios conteos precomputados

El objetivo de realizar varios conteos precomputados en el vector de bits, es reducir el tamaño en memoria requerido en comparación de una búsqueda LPM utilizando un solo conteo precomputado. La función de costo en memoria dada por la ecuación (4.3) aunque, presenta una optimización en memoria utilizando varios conteos precomputados, sigue teniendo un comportamiento exponencial. Sin embargo, cada término de la sumatoria, es de menor tamaño que $K \times 2^K$ bits (tamaño de un solo conteo precomputado), lo que implica que se ha logrado una optimización en memoria. Recuerde que nuestro objetivo es encontrar los valores de $(l_i)_{1 \leq i \leq h} = l_1, l_2, ..., l_i, ..., l_h$ que garanticen el mínimo de memoria requerida y así, construir los arreglos contadores correspondientes, donde , h es el número de conteos precomputados y los valores $(l_i)_{1 \leq i \leq h}$ representan niveles en el *trie* que determinan las dimensiones de cada arreglo contador. Nótese que mientras más niveles hallá en $(l_i)_{1 \leq i \leq h}$, más operaciones de acceso a memoria se tendrán que realizar pero a cambio, se podría obtener una reducción en memoria.

Obtener el mínimo de la función dada por la ecuación (4.3) es un problema combinatorio de un *trie* de profundidad K con hasta $h = K$ arreglos contadores. En el peor de los casos se tendrán que comprobar por *fuerza bruta* cada una de las posibilidades y obtener así los valores $(l_i)_{1 \leq i \leq hi}$ que garanticen un mínimo consumo en memoria. Dado que, como hemos recalcado en secciones anteriores, el costo no depende de la distribución de prefijos en la tabla, entonces podemos realizar una búsqueda exhaustiva para encontrar dichos parámetros. Construimos un programa con un algoritmo recursivo para comprobar cada combinación posible basado en los argumentos de la sección 4.2.3 y obtuvimos los mínimos globales en pocos segundos hasta $K = 32$ (*tries* que cubren todo IPv4).

La cuadro 5.2 muestra una peculiaridad que encontramos en los *tries* de profundidad $K = 4, 8, 16$; estos *tries* tienen más de un mínimo global, lo que significa que con cualquiera de las configuraciones de $(l_i)_{1 \leq i \leq h}$ presentadas, se garantiza una máxima optimización en memoria requerida.

El cuadro 5.3 muestra los resultados donde obtuvimos el menor consumo en memoria para el intervalo de profundidades $K \in [16, 32]$. Para cada profundidad K se comprobaron todos los posibles números de conteos ($2 \leq h \leq K$) en todos los posibles niveles $(l_i)_{1 \leq i \leq h} =$

Cuadro 5.2: Mínimos globales repetidos utilizando conteos precomputados.

Profundidad del *trie*	Conjunto de niveles $L = (l_i)_{1 \leq i \leq h}$	Costo de codificación en MB
$K = 4$	$L = \{2, 4\}$	0.0000057MB
	$L = \{3, 4\}$	
	$L = \{1, 3, 4\}$	
	$L = \{2, 3, 4\}$	
$K = 8$	$L = \{3, 5, 7, 8\}$	0.0001029MB
	$L = \{2, 5, 7, 8\}$	
$K = 16$	$L = \{4, 7, 10, 13, 15, 16\}$	0.026MB
	$L = \{3, 7, 10, 13, 15, 16\}$	

$l_1, l_2, ..., l_h$. El cuadro incluye además de la profundidad correspondiente del *trie*, su costo en memoria en MB, número de accesos a memoria para realizar una búsqueda LPM y los niveles correspondientes a la optimización. Se puede notar que mientras mayor sea la profundidad, más conteos son necesarios. También note que, a diferencia del costo presentado en el cuadro 5.1 para $K = 32$ el costo se redujo bruscamente de 16 GB a 1.7 GB, en consecuencia se puede apreciar un ahorro bastante significativo.

5.1.3. Búsqueda mediante varios conteos precomputados y patrones de bits

Recordemos que el último arreglo contador de la búsqueda LPM que utiliza varios arreglos precomputados tiene 2^K entradas de $(K - l_{h-1})$ bits cada una de ellas y que, este arreglo es el que tiene el mayor costo entre los arreglos contadores que conforman la codificación. Recordemos también que en el capítulo anterior analizamos que en el vector de bits sólo aparecen ciertos patrones de longitud dada por la cobertura en el nivel l_{h-1}. De esta manera, quitamos el último arreglo contador y lo cambiamos por una tabla diccionario, agregando un apuntador a sus entradas en el penúltimo arreglo contador A_{h-1}.

En principio, mostramos los costos de la tabla diccionario y el apuntador a ella en el arreglo A_{h-1} a ella. El costo de la tabla diccionario es $(2^{K-l_{h-1}})(p_{K-l_{h-1}} + 1)(K - l_{h-1})$

Cuadro 5.3: Costo en memoria e instrucciones de acceso a memoria para *tries* en el intervalo de profundidades $[16, 32]$ utilizando conteos precomputados dados por $L = (l_i)_{1 \leq i \leq h}$.

Profundidad del *trie* K	Conjunto de niveles L	Costo en MB	Instrucciones de acceso a memoria
16	$L = \{3, 7, 10, 13, 15, 16\}$	0.026702881	6
17	$L = \{4, 8, 11, 14, 16, 17\}$	0.053407669	6
18	$L = \{5, 9, 12, 15, 17, 18\}$	0.106819153	6
19	$L = \{2, 6, 10, 13, 16, 18, 19\}$	0.213639736	7
20	$L = \{3, 7, 11, 14, 17, 19, 20\}$	0.427280426	7
21	$L = \{4, 8, 12, 15, 18, 20, 21\}$	0.854562759	7
22	$L = \{5, 9, 13, 16, 19, 21, 22\}$	1.709129333	7
23	$L = \{2, 6, 10, 14, 17, 20, 22, 23\}$	3.418262005	8
24	$L = \{3, 7, 11, 15, 18, 21, 23, 24\}$	6.836524963	8
25	$L = \{4, 8, 12, 16, 19, 22, 24, 25\}$	13.67305183	8
26	$L = \{5, 9, 13, 17, 20, 23, 25, 26\}$	27.34610748	8
27	$L = \{2, 6, 10, 14, 18, 21, 24, 26, 27\}$	54.69222021	9
28	$L = \{3, 7, 11, 15, 19, 22, 25, 27, 28\}$	109.3844414	9
29	$L = \{4, 8, 12, 16, 20, 23, 26, 28, 29\}$	218.7688847	9
30	$L = \{5, 9, 13, 17, 21, 24, 27, 29, 30\}$	437.5377731	9
31	$L = \{2, 6, 10, 14, 18, 22, 25, 28, 30, 31\}$	875.0755534	10
32	$L = \{2, 7, 11, 15, 19, 23, 26, 29, 31, 32\}$	1750.151108	10

Cuadro 5.4: Costos en memoria de la tabla diccionario y apuntador a ella *vs* longitud del patrón de bits seleccionado.

Longitud del patrón de bits	Tamaño en bits de la tabla diccionario	Tamaño en bits del apuntador a la tabla diccionario
2	6	2
4	48	3
8	648	5
16	43392	10
32	73332960	19
64	8.0×10^{13}	38
128	3.95×10^{13}	76
256	3.98×10^{13}	151
512	1.74×10^{13}	301
1024	1.47×10^{13}	602

tal que, $p_q = (p_{q-1})^2 + 1$ y $p = 1$ es la longitud de la cobertura en l_{h-1} y el costo del apuntador es $\lceil log_2[p_{K-l_{h-1}} + 1]) \rceil$. El cuadro 5.4 muestra dichos costos de memoria de la tabla diccionario y de su respectivo apuntador. Cada costo en cada renglón de la tabla corresponde a una longitud del patrón de bits y cada costo de la tabla diccionario crece de forma exponencial de acuerdo con la longitud del patrón de bits. Para patrones de bits de tamaño mayor a 32 bits se observa un comportamiento exponencial de costo en memoria, por lo tanto, utilizaremos patrones de 2, 4, 8 y hasta 16 bits para el siguiente análisis.

Dada la naturaleza del problema combinatorio de los conteos precomputados, agregar la tabla diccionario sigue conformando el mismo problema combinatorio. Sin embargo, las soluciones óptimas en memoria no necesariamente son las mismas. Nosotros construimos nuevamente el algoritmo recursivo de la seccion 4.2.3 que considera todas las posibilidades de niveles y calculamos sus respectivos costos basados en la ecuación (4.5).

El Cuadro 5.5 muestra una peculiaridad que encontramos en los *tries* de profundidad $K = 4, 8, 16$; estos *tries* tienen más de un mínimo global, lo que significa que con cualquiera de las configuraciones de $(l_i)_{1 \leq i \leq h}$ presentadas, se garantiza una máxima optimización en

Cuadro 5.5: Mínimos globales repetidos utilizando conteos precomputados y patrones de bits.

Profundidad del *trie*	Conjunto de niveles $L = (l_i)_{1 \leq i \leq h}$	Costo de codificación en MB
$K = 4$	$L = \{2, 3, 4\}$	0.00000643MB
	$L = \{1, 3, 4\}$	
$K = 8$	$L = \{3, 6, 8\}$	0.0000743MB
	$L = \{4, 6, 8\}$	
	$L = \{1, 4, 6, 8\}$	
	$L = \{2, 4, 6, 8\}$	
$K = 16$	$L = \{3, 6, 8, 16\}$	0.0121MB
	$L = \{4, 7, 10, 13, 16\}$	

memoria requerida. Note que esta misma peculiaridad la encontramos en los mismos niveles correspondientes a conteos precomputados sin la tabla diccionario. Sin embargo, los costos son diferentes y además menores.

El Cuadro 5.6 muestra los resultados para el intervalo de profundidades $K \in [16, 32]$ utilizando conteos precomputados y patrones de bits en los niveles dados por L. Para cada profundidad K se comprobaron todos los posibles números de conteos ($2 \leq h \leq K$) en todos los posibles niveles $(l_i)_{1 \leq i \leq h} = l_1, l_2, ..., l_h$. El Cuadro incluye ademas de la profundidad del *trie* binario, la longitud del patrón de bits, su costo en memoria, el número de instrucciones de acceso a la misma para una búsqueda LPM y los niveles correspondientes a la optimización. Se puede notar que mientras mayor sea la profundidad, más conteos precumputados son necesarios. Note también que, el costo de un solo conteo precomputado (Cuadro 5.1) con $K = 32$ es de 16 GB mientras que con el método de distribución de conteos y patrones de bits, el costo se reduce a 466 MB, de esta manera se puede apreciar la eficiencia entre los esquemas propuestos y por lo tanto la efectividad entre ellos.

Cuadro 5.6: Costo en memoria e instrucciones de acceso a memoria para *tries* en el intervalo de profundidades $[16, 32]$ utilizando conteos precomputados y patrones de bits dados por L.

Profundidad del *trie* K	Conjunto de niveles $L = (l_i)_{1 \leq i \leq h}$	Longitud del patrón de bits	Costo en MB	Instrucciones de acceso a memoria
16	$L = \{3, 7, 10, 13, 16\}$	8	0.01213169	5
17	$L = \{3, 7, 10, 13, 17\}$	16	0.02322483	5
18	$L = \{4, 8, 11, 14, 18\}$	16	0.04127884	5
19	$L = \{5, 9, 12, 15, 19\}$	16	0.07738876	5
20	$L = \{2, 6, 10, 13, 16, 20\}$	16	0.1496067	6
21	$L = \{3, 7, 11, 14, 17, 21\}$	16	0.29404163	6
22	$L = \{4, 8, 12, 15, 18, 22\}$	16	0.58291245	6
23	$L = \{5, 9, 13, 16, 19, 23\}$	16	1.16065598	6
24	$L = \{2, 6, 10, 14, 17, 20, 24\}$	16	2.31614304	7
25	$L = \{3, 7, 11, 15, 18, 21, 25\}$	16	4.6271143	7
26	$L = \{4, 8, 12, 16, 19, 22, 26\}$	16	9.24905777	7
27	$L = \{5, 9, 13, 17, 20, 23, 27\}$	16	18.4929466	7
28	$L = \{2, 6, 10, 14, 18, 21, 24, 28\}$	16	36.9807262	8
29	$L = \{2, 6, 10, 14, 18, 21, 24, 29\}$	32	66.0008569	8
30	$L = \{3, 7, 11, 15, 19, 22, 25, 30\}$	32	123.259745	8
31	$L = \{4, 8, 12, 16, 20, 23, 26, 31\}$	32	237.777521	8
32	$L = \{4, 9, 13, 17, 21, 24, 27, 32\}$	32	466.813076	8

5.1.4. Comparación de esquemas propuestos de búsqueda LPM

La codificación del vector de bits propuesta en esta investigación consiste en:

1. Un arreglo contador de conteos precomputados.
2. Varios conteos precomputados en donde se distribuye el conteo de CPIs del vector de bits.
3. Varios conteos precomputados de CPIs y patrones de bits en donde se distribuye el conteo del vector de bits.

Esencialmente la propuesta considera tres esquemas de búsqueda LPM diferentes. En el mismo orden, el primero es el que requiere un mínimo número de accesos a memoria pero la más costosa en memoria y el último lo contrario, la menos costosa en memoria pero la más costosa en instrucciones. La figura 5.1 muestra un gráfico comparativo en consumo en memoria de los tres esquemas mencionados con anterioridad. Cada punto en la gráfica corresponde a una profundidad del *trie* y su respectivo costo de codificación. Note que en el eje de las ordenadas el costo en bits tiene una representación logarítmica, de esta forma puede compararse el consumo de memoria de los tres esquemas. Note también que los tres esquemas tienen un crecimiento exponencial.

La figura 5.2 muestra los costos en instrucciones de acceso a memoria de los esquemas de búsqueda LPM con un solo arreglo de conteos precomputados, con más de un arreglo de conteos precomputados y también el esquema que considera patrones de bits. Puede observarse que utilizando un solo arreglo contador solo se necesita un acceso y que para los otros dos esquemas el costo crece de forma lineal. Note que a diferencia de la figura 5.1, el esquema de varios conteos precomputados que consideran patrones de bits tienen un menor costo en instrucciones de acceso a memoria en comparación con el esquema que no considera dichos patrones de bits. Este resultado es interesante puesto que, el esquema que requiere menos memoria, también es el que requiere menos accesos a memoria. Para ambas figuras solo presentamos una profundidad máxima de 32 puesto que el comportamiento no cambia para profundidades superiores y el costo se hace inoperable.

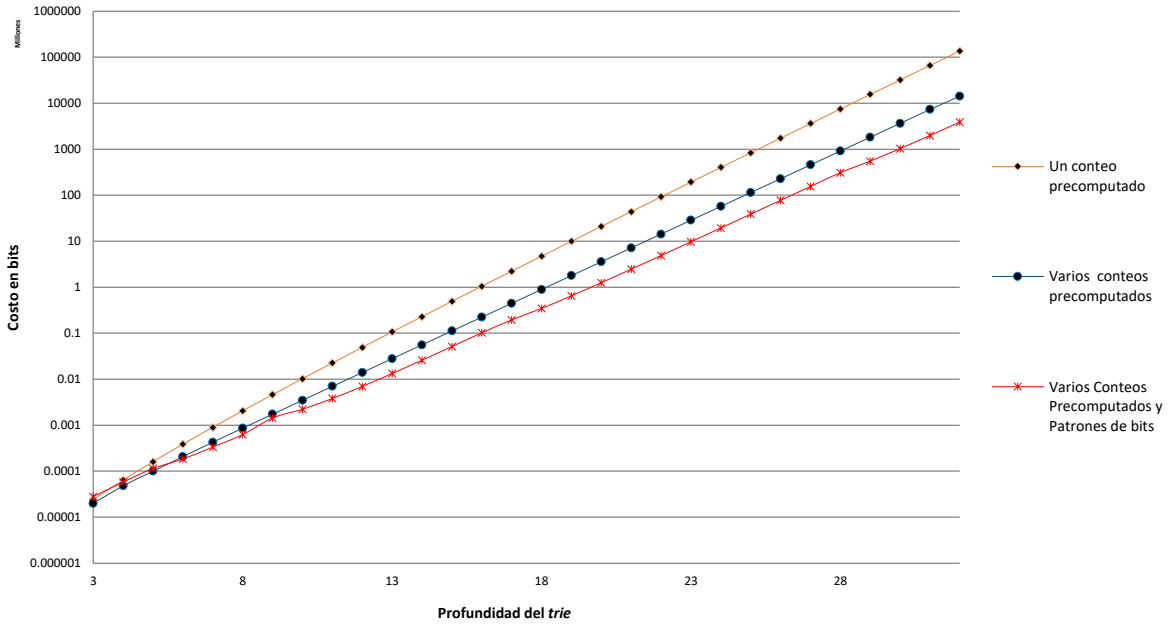


Figura 5.1: Costo en memoria en bits para los esquemas de búsqueda LPM con un conteo precomputado, varios conteos precumputados y patrones de bits. Los costos corresponden al intervalo de profundidades $[3, 32]$ del *trie* binario.

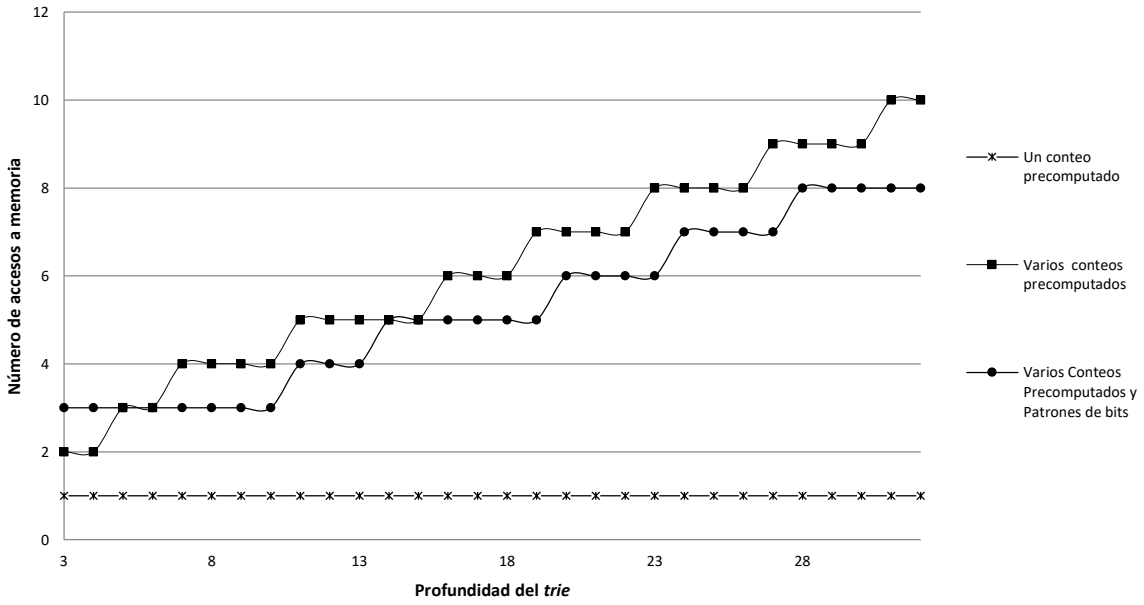


Figura 5.2: Costo en instrucciones de acceso a memoria para los esquemas de búsqueda LPM con un conteo precomputado, varios conteos precumputados y patrones de bits. Los costos corresponden al intervalo de profundidades $[3, 32]$ del *trie* binario.

5.1.5. Implementación del esquema general de búsqueda

En la sección anterior nos dimos a la tarea de encontrar los mínimos absolutos de costo en memoria para los esquemas que utilizan más de un arreglo de conteos precomputados del vector de bits. En esta sección nos cuestionamos si existe alguna diferencia substancial entre un mínimo local y un mínimo global. A continuación presentamos el análisis de parámetros involucrados en la codificación de un *trie* de una profundidad determinada; específicamente hacemos el análisis del número de conteos precomputados ya que en ese número recae la complejidad de instrucciones. Cabe mencionar que hicimos el análisis de los esquemas que consideran varios conteos precomputados y con el esquema que además considera patrones de bits, pero, debido al parecido solo presentamos los resultados del esquema que considera los patrones de bits puesto que en él hay una ganancia en memoria más significativa.

5.1.5.1. Análisis de parámetros

Tomamos un *trie* de profundidad 32 y de acuerdo al Cuadro 5.6 encontramos que, se necesitan 8 arreglos contadores y por lo tanto 8 accesos a memoria para determinar el LPM correspondiente a la dirección IP. Calculando los niveles que optimizan en memoria utilizando desde 2 hasta 32 arreglos contadores, encontramos los resultados de la figura 5.3. En dicha figura se puede observar el costo en MB por utilizar un solo arreglo contador y como ese costo disminuye al distribuir los conteos del vector de bits agregando un arreglo contador. En la imagen se puede observar que a medida que aumentan el número de conteos se reduce el costo; sin embargo, la mayor ganancia de ahorro en memoria se encuentra dentro los primeros números de conteos, de hecho empieza a elevarse después de 28 arreglos contadores.

Del análisis anterior vemos que aunque el óptimo global se obtiene con 8 arreglos contadores, el costo con un número menor de arreglos es bastante similar y se requeriría un número menor de accesos a memoria. Tener un menor número de accesos a memoria puede valer la pena por no tener el óptimo global de la cantidad de memoria. Este mismo análisis lo realizamos para todas las profundidades hasta 32 y observamos el mismo comportamiento. La figura 5.4 muestra una gráfica donde fijamos el número de conteos

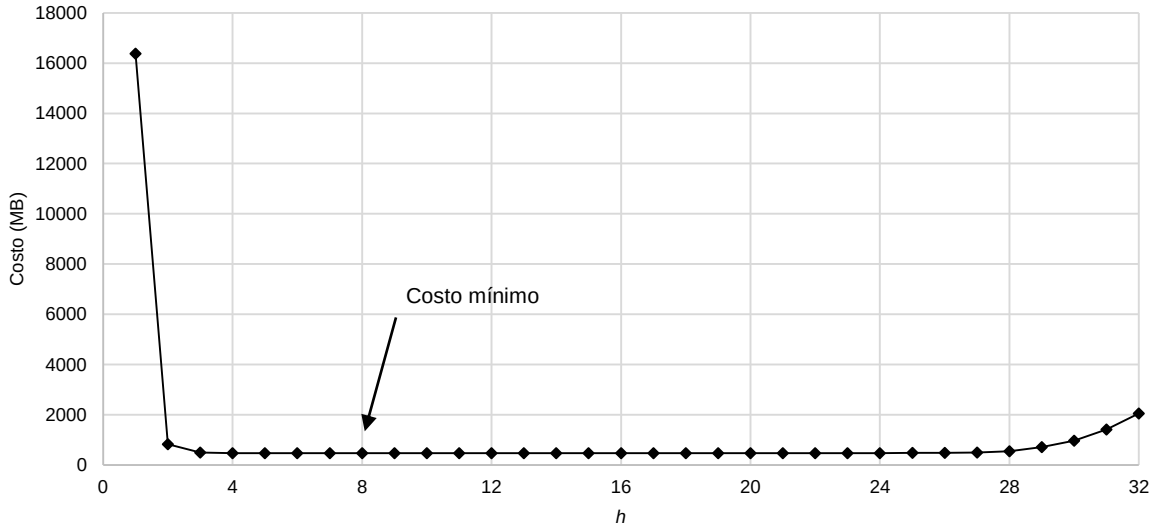


Figura 5.3: Costo en memoria en MB para todos los posibles valores de h (conteos precumputados) con un *trie* de profundidad $K = 32$.

en el vector de bits desde $h = 1$ hasta $h = 4$, optimizamos su costos y determinamos su comportamiento. En la gráfica puede verse que con $h = 1$ (un solo arreglo contador donde no se puede realizar ningún tipo de optimización) se tiene el costo más elevado, sin embargo, al agregar otro contador ($h = 2$) el costo disminuye para todos los *tries;* para $h = 4$ no hay ganancia significativa lo que nos hace pensar que no vale la pena rebasar los 4 arreglos contadores al codificar el vector de bits.

En el Cuadro 5.7 y 5.8 se muestran los resultados de la gráfica anterior, pero a diferencia de la gráfica, el cuadro muestra los valores correspondientes de l_i (niveles en los *tries*) y sus respectivos costos en MB. Por ejemplo, para $K = 32$ el costo óptimo es de 466MB (según el Cuadro 5.6) con $L = (l_i)_{1 \leq i \leq 8} = \{4, 9, 13, 17, 21, 24, 27, 32\}$ pero si limitamos el número de nivel a cuatro ($h = 4$) con $L = (l_i)_{1 \leq i \leq 4} = \{19,\ 24,\ 27,\ 32\}$ el costo es de 468MB; esto implica una diferencia poco significativa de 2MB en memoria pero en cambio una ganancia de 4 accesos a memoria por búsqueda LPM. Note que para algunas profundidades existe más de una solución de niveles $L = (l_i)_{1 \leq i \leq h}$.

Cuadro 5.7: Costos optimizados y sus niveles de acuerdo con 1, 2, 3, y 4 conteos precomputados fijos.

K	$h = 1$	$h = 2$	$h = 3$	$h = 4$
16	$L = \{16\}$	$L = \{12, 16\}$	$L = \{9, 13, 16\}$	$L = \{6, 10, 13, 16\}$
	0.125MB	0.017M	$L = \{10, 13, 16\}$	$L = \{7, 10, 13, 16\}$
			0.0123MB	0.0121MB
17	$L = \{17\}$	$L = \{13, 17\}$	$L = \{9, 13, 17\}$	$L = \{6, 10, 13, 17\}$
	0.265MB	0.031MB	0.0237MB	0.0232MB
18	$L = \{18\}$	$L = \{14, 18\}$	$L = \{10, 14, 18\}$	$L = \{7, 11, 14, 18\}$
	0.562MB	0.059MB	0.0425MB	0.0413MB
19	$L = \{19\}$	$L = \{15, 19\}$	$L = \{11, 15, 19\}$	$L = \{8, 12, 15, 19\}$
	1.187MB	0.118MB	0.0801MB	0.0775MB
20	$L = \{20\}$	$L = \{16, 20\}$	$L = \{12, 16, 20\}$	$L = \{9, 13, 16, 20\}$
	2.500MB	0.239MB	0.1555MB	0.1499MB
21	$L = \{21\}$	$L = \{17, 21\}$	$L = \{13, 17, 21\}$	$L = \{10, 14, 17, 21\}$
	5.250MB	0.489MB	0.3062MB	0.2948MB
22	$L = \{22\}$	$L = \{18, 22\}$	$L = \{14, 18, 22\}$	$L = \{11, 15, 18, 22\}$
	11.000MB	1.005MB	0.6106MB	0.5843MB
23	$L = \{23\}$	$L = \{19, 23\}$	$L = \{15, 19, 23\}$	$L = \{12, 16, 19, 23\}$
	23.000MB	2.067MB	1.2200MB	1.1648MB

Cuadro 5.8: Costos optimizados y sus niveles de acuerdo con 1, 2, 3, y 4 conteos pre-computados fijos.

K	$h = 1$	$h = 2$	$h = 3$	$h = 4$
24	$L = \{24\}$	$L = \{20,\ 24\}$	$L = \{16,\ 20,\ 24\}$	$L = \{13,\ 17,\ 20,\ 24\}$
	48.000MB	4.255MB	2.4426MB	2.3254MB
25	$L = \{25\}$	$L = \{21,\ 25\}$	$L = \{17,\ 21,\ 25\}$	$L = \{14,\ 18,\ 21,\ 25\}$
	100.00MB	8.755MB	4.8957MB	4.6477MB
26	$L = \{26\}$	$L = \{22,\ 26\}$	$L = \{18\ 22,\ 26\}$	$L = \{15,\ 19,\ 22,\ 26\}$
	208.00MB	18.005MB	9.8176MB	9.2942MB
27	$L = \{27\}$	$L = \{22,\ 27\}$	$L = \{19,\ 23,\ 27\}$	$L = \{16,\ 20,\ 23,\ 27\}$
	432.00MB	31.741MB	19.692MB	18.591MB
28	$L = \{28\}$	$L = \{23,\ 28\}$	$L = \{19,\ 23,\ 28\}$	$L = \{17,\ 21,\ 24,\ 28\}$
	896.00MB	55.741MB	38.491MB	37.192MB
29	$L = \{29\}$	$L = \{24,\ 29\}$	$L = \{20,\ 24,\ 29\}$	$L = \{17,\ 21,\ 24,\ 29\}$
	1856.0MB	104.74MB	68.366MB	66.195MB
30	$L = \{30\}$	$L = \{25,\ 30\}$	$L = \{21,\ 25,\ 30\}$	$L = \{18,\ 22,\ 25,\ 30\}$
	3840.0MB	204.74MB	128.24MB	123.67MB
31	$L = \{31\}$	$L = \{26,\ 31\}$	$L = \{22,\ 26,\ 31\}$	$L = \{19,\ 23,\ 26,\ 31\}$
	7936.0MB	408.74MB	248.24MB	238.67MB
32	$L = \{32\}$	$L = \{27,\ 32\}$	$L = \{22,\ 27,\ 32\}$	$L = \{19,\ 24,\ 27,\ 32\}$
	16384MB	824.74MB	$L = \{23,\ 27,\ 32\}$	$L = \{20,\ 24,\ 27,\ 32\}$
			488.74MB	468.74MB

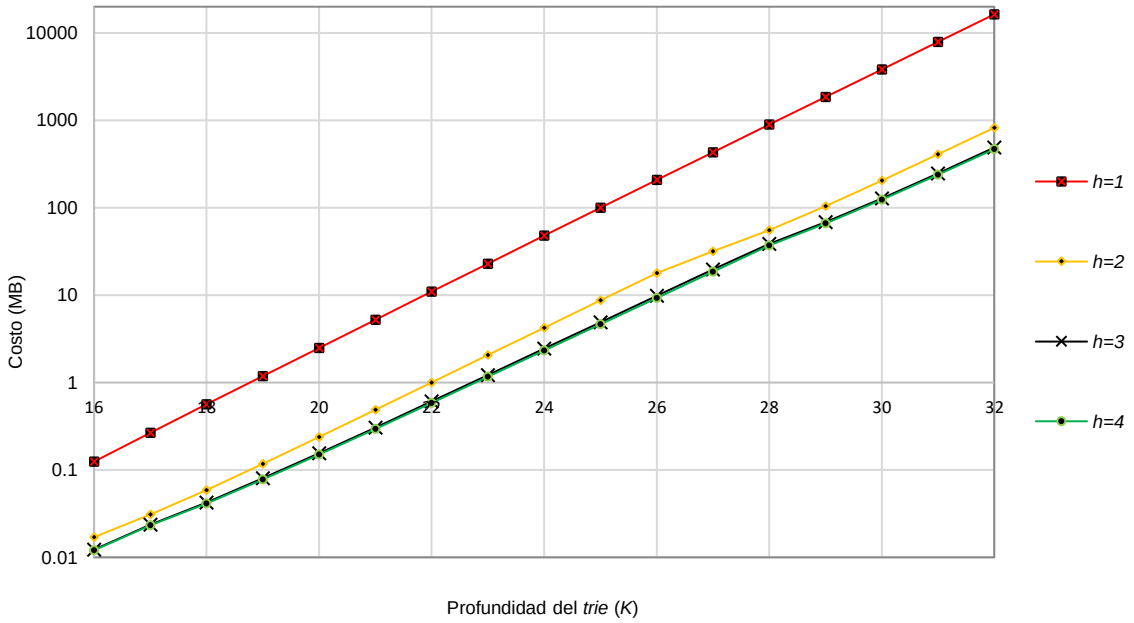


Figura 5.4: Costos en memoria en MB para 1, 2, 3, y 4 conteos precomputados y patrones de bits en intervalo de profundidades $[16, 32]$.

5.1.5.2. Implementación en tablas de encaminamiento reales IPv4

Como ejemplo de una implementación real tomamos tablas reales IPv4 de diferentes años y fechas y realizamos búsquedas LPM en dos etapas de acuerdo con la figura 4.14 del Capítulo 4. Se eligió que el *trie* de la primera etapa fuera de una profundidad $K = 24$ debido a que más del 99 % de prefijos tiene una profundidad menor o igual a 24 [21]. En consecuencia, la segunda etapa considera *subtries* de profundidad $K = 8$. Para realizar el experimento fijamos 1 y 2 conteos parciales ($h = 1$ y $h = 2$) en el vector de bits; a partir de ello usamos los parámetros de la tabla anterior. Utilizamos también una computadora de propósito general con las siguiente especificaciones: Intel core i5-3450 (cuatro núcleos de 3.1GHz), dos memorias cache de nivel 1 de 32KB por núcleo (latencia experimental de 1ns), una memoria cache de nivel 2 de 256KB por núcleo (latencia experimental de 3ns), una memoria de tercer nivel de 6MB (latencia experimental de 9ns) y finalmente una memoria principal SDRAM con una latencia de 72ns por acceso. Las latencias experimentales las obtuvimos con *LMbench - Tools for performance analysis* [22]. La latencia de

Cuadro 5.9: Resultados experimentales utilizando un arreglo precomputado en la primera etapa ($h = 1$).

Fecha de la Tabla de encaminamiento	Costo Total (MB)	Latencia en el Mejor Caso(ns)	Latencia en el Peor Caso (ns)
15/01/2021	50.98	[50.8,51.1]	[83.8,85.0]
20/03/2020	51.04	[51.0,51.9]	[83.2,84.2]
16/07/2015	50.19	[52.1,52.8]	[84.3,84.9]
01/01/2015	50.07	[52.1,52.5]	[84.2,84.7]
01/10/2007	49.33	[51.4,51.8]	[81.9,82.5]
26/10/2001	48.36	[51.0,51.7]	[79.5,80.1]

las memoria depende de su arquitectura pero también depende directamente de la pureza de los materiales con que está construida. Es por ello que existen herramientas que hacen pruebas de estrés a las memorias para determinar la latencia promedio de cada una de ellas.

Los Cuadros 5.9 y 5.10 muestran los costos totales de memoria y el número de accesos al realizar búsquedas LPM con uno y dos conteos precomputados respectivamente ($h = 1$ y $h = 2$). El costo total considera los arreglos contadores de *trie* de la primera etapa, los arreglos contadores correspondientes a los *subtries* de la segunda etapa y los prefijos en las tablas extendidas de encaminamiento. Presentamos el mejor y peor de los casos que corresponden a encontrar el LPM en la primera o en la segunda etapa respectivamente. Las latencias de mejor y peor caso se encuentran en un intervalo de confianza al 95 %, en otras palabras realizamos series de búsquedas LPM aleatorias que garanticen el intervalo de confianza. Puede notarse que las latencias de búsqueda dependen directamente del costo total en memoria y de las características de la PC. Con $h = 1$ se tienen costos de memoria promedio de 50MB por lo que se debe realizar accesos a memoria SDRAM pero con $h = 2$ se reduce bruscamente la latencia de búsqueda puesto que se ejecuta en memoria cache de nivel 3.

Podemos concluir con estos experimentos la veracidad de la afirmación de la relación entre latencias de acceso y capacidad de almacenamiento de acuerdo al tipo de memoria. De acuerdo a las características del equipo de cómputo utilizado esperábamos latencias

Cuadro 5.10: Resultados experimentales utilizando dos arreglos precomputados en la primera etapa ($h = 2$).

Fecha de la Tabla de encaminamiento	Costo Total (MB)	Latencia en el Mejor caso(ns)	Latencia en el Peor Caso (ns)
15/01/2021	6.680	[30.8,31.1]	[60.0, 60.5]
20/03/2020	6.742	[30.8,31.2]	[59.5, 60.2]
16/07/2015	6.445	[30.1,30.6]	[58.6, 59.3]
01/01/2015	6.333	[29.0,29.9]	[59.1,61.4]
01/10/2007	5.593	[24.8,25.6]	[41.6,42.3]
26/10/2001	4.624	[24.2,25.1]	[37.5,38.2]

de búsqueda LPM menores en el experimento donde hay más acceso a memoria. Lo cual comprobamos satisfactoriamente.

5.1.6. Análisis del esquema de búsqueda LPM y el esquema de *Lulea*

La tesis de este trabajo comenzó bajo el supuesto de que el esquema que propone *Lulea* pudiera generalizare a *tries* de mayor profundidad y justificar el número y tamaño de sus arreglos en función de un ahorro en memoria. Podemos considerar a *Lulea* como un caso particular de nuestra generalización siendo que utilizan un *trie* de profundidad 16 con dos arreglos que distribuyen un conteo de valores de un vector de bits (*base index* y *code word*) y un arreglo bidimensional de patrones de 16 bits (*maptable*). En otras palabras podemos decir que, $K = 16$ y que $(l_i)_{1 \leq i \leq 3} = \{l_1 = 10, l_2 = 12, l_3 = 16\}$ y una tabla diccionario que substituye al arreglo contador en la profundidad 16.

El Cuadro 5.11 muestra los parámetros del esquema de *Lulea* como una configuración posible de parámetros de la generalización propuesta. Puede observar de acuerdo a los resultados en la tabla 5.7 que, la serie $L = (l_i)_{1 \leq i \leq 3} = \{l_1 = 10, l_2 = 12, l_3 = 16\}$ no es la configuración donde existe un mayor ahorro de memoria. De esta manera se presentan en la tabla, los parámetros óptimos en memoria para el *trie* de profundidad $K = 16$ con la serie $L = (l_i)_{1 \leq i \leq 5} = \{l_1 = 3, l_2 = 7, l_3 = 10, l_4 = 13, l_5 = 16\}$, de tal suerte que se

Cuadro 5.11: Comparación en costo de operaciones y memoria entre el esquema de *Lulea* y nuestra propuesta optimizada en memoria.

Parámetros	*Lulea*	Parámetros optimizados
Niveles de arreglos contadores	$L = \{10, 12, 16\}$	$L = \{3, 7, 10, 13, 16\}$
Longitud patrones de bits	16	8
Costo (MB)	0.01493835MB	0.01213169MB
Accesos a memoria por búsqueda LPM	3	5

Cuadro 5.12: Comparación en costo de operaciones y memoria entre el esquema de *Lulea* y nuestra propuesta optimizada en memoria fijando $h = 3$.

Parámetros	*Lulea*	Parámetros optimizados
Niveles de arreglos contadores	$L = \{10, 12, 16\}$	$L = \{9, 13, 16\}$
Longitud patrones de bits	16	8
Costo (MB)	0.01493835MB	0.012768984MB
Accesos a memoria por búsqueda LPM	3	3

pueden comparar los consumos en memoria requerida y el número de accesos a memoria por búsqueda LPM (de tres accesos a memoria usando *Lulea* a 5 accesos utilizando nuestra propuesta).

Configurando nuestro esquema con $h = 2$, $K = 16$ y $L = (l_i)_{1 \leq i \leq 3} = \{l_1 = 9, l_2 = 13, l_3 = 16\}$ a fin de igualar el número de accesos a memoria *Lulea,* encontramos los resultados del Cuadro 5.12. Determinamos que aún así, nuestra propuesta ofrece un mayor ahorro de memoria eligiendo correctamente los valores de $(l_i)_{1 \leq i \leq 3}$. Destacamos el hecho que *Lulea* es un referente clásico de esquemas que codifican el *trie* binario, que nosotros mejoramos su rendimiento y más aún, convertimos todo su esquema en un caso particular del nuestro.

5.2. Esquema de particionamiento en etapas del *trie* binario

A diferencia del esquema de codificación de vector de bits en conteos precomputados que es totalmente independiente de la distribución y del número de prefijos en la tabla de encaminamiento, seccionar de la mejor manera la búsqueda LPM en etapas depende por completo de la distribución de los prefijos en la tabla. En el experimento de la sección 5.1.5.2 realizamos una búsqueda LPM en dos fases utilizando una tabla IPv4, el criterio para seleccionar la profundidad de las etapas fue concentrar la mayor cantidad de prefijos en el primer *trie* binario. En esta sección presentamos una metodología para optimizar el particionamiento en etapas utilizando los algoritmos evolutivos descritos en el capítulo anterior. Para validar el funcionamiento del algoritmo genético utilizamos las tablas de encaminamiento IPv4/6 del enrutador AS6447 [13] correspondiente al 20 de marzo de 2019 y al 15 de enero de 2021 y las actualizaciones correspondientes a esas fechas. Las actualizaciones de las tablas de encaminamiento están contenidas en una base de datos que contiene principalmente a los prefijos de red que serán insertados o eliminados aunados a la fecha y hora en que se produce la actualización. La fecha y hora de una actualización en la base de datos utilizada tiene precisión de un segundo.

5.2.1. Desempeño del algoritmo genético

El funcionamiento del algoritmo genético considera los costos de los *subtries* en los niveles donde existen particiones delimitadas en las etapas de búsqueda; cada *subtrie* es codificado bajo algún esquema, ya sea utilizando una de las técnicas antes descritas o cualquier otra técnica presentada en el estado del arte revisado en el Capítulo 2. Nosotros utilizaremos el costo de $K \times 2^K$ bits que es el mismo para un solo arreglo contador o bien el que considera todos los nodos posibles en el *trie*. Utilizaremos este costo ya que es el más elevado tanto en nuestra propuesta como en las propuestas vistas en el estado del arte, esto con el fin de demostrar que la optimización obtenida por distribuir los conteos del vector de bits, no afecta el desempeño del evolutivo y que, inclusive puede utilizarse en otros esquemas de codificación del *trie* binario.

El 20 de marzo de 2019 a cierta hora, la tabla de encaminamiento IPv4 contenía 742,313 prefijos. Después de aplicar *leaf-pushing* resultaron 1,082,558 prefijos disjuntos. En la tabla de encaminamiento IPv6 encontramos 62,096 prefijos, después de aplicar *leaf-pushing* resultaron 342,912 prefijos disjuntos. Mientras que para la fecha de 15 de enero de 2021, la tabla de encaminamiento IPv4 contenía 8026,970 prefijos, después de aplicar *leaf-pushing* resultaron 1,184,320 prefijos disjuntos. En la tabla de encaminamiento IPv6 encontramos 104,391 prefijos, después de aplicar *leaf-pushing* obtuvimos 387,979 prefijos disjuntos. Adicionalmente, en nuestra implementación, definimos a un periodo de actualización como el tiempo entre actualizaciones de la tabla de encaminamiento. En las bases de datos utilizadas, el periodo de actualizaciones es de un segundo. En este caso encontramos desde 0 hasta 3 millares de inserciones-eliminaciones de prefijos de red en un periodo de actualización.

Experimentalmente determinamos que, para configuraciones con más de 5 etapas de búsqueda, en la tabla de encaminamiento IPv4, no se tiene una ganancia significativa en ahorro de memoria. Esto porque mientras haya más niveles en el *trie*, existe una mayor cantidad de apuntadores en las tablas extendidas. Similarmente en IPv6, con más de 18 etapas de búsqueda no encontramos una ganancia significativa en ahorro de memoria requerida. Los experimentos realizados fueron hechos en tablas IPv4/6 de diferentes años hasta fechas actuales. Para mostrar la importancia de seleccionar los niveles adecuados en el *trie* binario para la tabla IPv4 de la fecha actual, proponemos un ejemplo con tres etapas de búsqueda LPM y contrastamos los costos en memoria obtenidos aplicando nuestro algoritmo: con los niveles determinado inmediatamente de la evolución y con los niveles obtenidos después de que el algoritmo ha convergido en una solución optimizada. En la primera iteración resultado el cromosoma {3,10,32} con el costo de 9GB y por el otro lado, el cromosoma de mayor aptitud después de la evolución que es {20,24,32} y tiene un costo total de 9.92MB. De igual forma, para demostrar la importancia de la posición de los niveles en el *trie* binario en la tabla IPv6 actual elegimos 18 niveles de búsqueda LPM. En este caso, obtuvimos que el costo en memoria es de 3TB para un cromosoma aleatorio en la primera iteración mientras que, el cromosoma de la solución optimizada tiene un costo de 6.7MB.

Las figuras 5.55.6(a) muestran un gráfico correspondiente al tiempo de convergencia

del algoritmo genético en 500 periodos de actualización de la tabla de encaminamiento IPv4 con 5 etapas de búsqueda. Las figuras 5.55.6(b) muestran los tiempos de convergencia del algoritmo genético en 500 periodos de actualización de la tablas IPv6 cuando la búsqueda LPM es dividida en 18 etapas. Como puede verse en las gráficas, el tiempo en que converge el algoritmo genético en cada periodo de tiempo no es constante debido a que por ejemplo puede haber un número diferente de inserciones-eliminaciones de prefijos en cada periodo de actualización. Nótese también que después de la primera vez que se obtiene una optimización en memoria requerida (el primer valor en tiempo de las gráficas) el algoritmo genético tiene una convergencia más rápida debido a que se parte de soluciones previamente optimizadas.

Actualmente se está migrando al protocolo IPv6 y se espera que el tamaño de las tablas para este protocolo tenga un crecimiento y tamaño parecido a las tablas de encaminamiento para IPv4. Para comprobar la eficiencia de nuestra propuesta en situaciones futuras utilizamos V6Gene [23] para crear una tabla IPv6 sintética con un mayor número de prefijos utilizando como semilla la tabla IPv6 real utilizada anteriormente. Inicialmente la tabla correspondiente al 2019 sintética tiene 98,231 prefijos, después de aplicar *leaf-pushing* obtuvimos 624,676 prefijos disjuntos; mientas que, la tabla sintética correspondiente al 2021 tiene 134,877 prefijos, después de aplicar *leaf-pushing* obtuvimos 803,443 prefijos disjuntos. Al igual que las tablas IPv6 reales, determinamos que con más de 18 etapas de búsqueda LPM ya no existe una mayor ganancia en memoria requerida. Las figuras 5.5 y 5.6(c) muestran los tiempos de convergencia del algoritmo genético en 500 periodos de actualización de la tablas IPv6 sintéticas. Note que dichos tiempos de convergencia tienen un comportamiento similar al de las tablas antes utilizadas. Para determinar con certeza la calidad de las soluciones del algoritmo genético se programó un algoritmo de fuerza bruta que genera todas las posibles soluciones de partir el *trie* binario en R niveles, (combinaciones sin repetición del capítulo anterior) calculando en cada una de ellas su costo. Nuestro algoritmo genético obtuvo la solución óptima en todas las simulaciones realizadas. Realizamos los experimentos anteriores utilizando tablas IPv4 real, IPv6 real e IPv6 sintética de distintos años, días y horas obteniendo resultados similares.

En general las gráficas de las figuras 5.5 y 5.6 muestran que la latencia del algoritmo genético es muy pequeña comparada con la precisión de los periodos de actualización

de las tablas utilizadas. Suponiendo casos donde existieran periodos de actualización con una mayor precisión, la capacidad de respuesta del algoritmo genético permite que la precisión de dichos periodos sea inclusive de decenas de milisegundos, precisando que, los algoritmos genéticos pueden ser optimizados en tiempo de ejecución si son programados en paradigmas de programación paralelos. En contraste, la propuesta de particionamiento de búsqueda exhaustiva de niveles óptimos hecha en [18] ofrece una gran optimización en tiempo de ejecución al precomputar de forma recursiva varias posibles soluciones en etapas, sin embargo, se tiene que considerar por lo menos una instrucción atómica para calcular el costo de cada combinación de niveles lo que significa que el tiempo de ejecución es directamente proporcional al número total de posibles combinaciones. En el caso de IPv6 ($K_T = 128$) dividida en $R = 18$ etapas de búsqueda se necesitarían varios billones de instrucciones para determinar la partición óptima cada vez que la tabla de encaminamiento sea actualizada. Lo cual tomaría un tiempo de ejecución del orden de segundos, lo que contrasta fuertemente con nuestra propuesta el Cuadro 5.13 muestra los rangos de costo en memoria requerida correspondiente a los periodos de actualización de las figuras 5.5 y 5.6. Los resultados presentados son los costos totales en memoria por contener tablas extendidas de prefijos y arreglos contadores y las tablas de encaminamiento. Las memorias actuales de rápido acceso superan los 50MB lo que implica que tanto los contadores como las tablas extendidas en IPv4/6 reales e IPv6 sintética pueden ser contenidos en ellas. La rapidez de una búsqueda LPM dependerá de la latencia de la memoria que contenga a los contadores y tablas extendidas. En el mejor de los casos el costo en instrucciones del esquema es dos accesos a memoria y en el peor de los casos es dos veces la cantidad de etapas en que se divida la búsqueda LPM.

5.2.2. Desempeño en memoria del esquema completo

Finalmente unificamos el esquema de particionamiento eficiente en etapas con el esquema de codificación del vector de bits mediante la distribución de conteos precalculados en arreglos contadores. Utilizamos las tablas de encaminamiento de la sección anterior y realizamos el mismo experimento obteniendo que los tiempos de convergencia del evolutivo son totalmente similares puesto que las ecuaciones de costo tienen exactamente el mismo

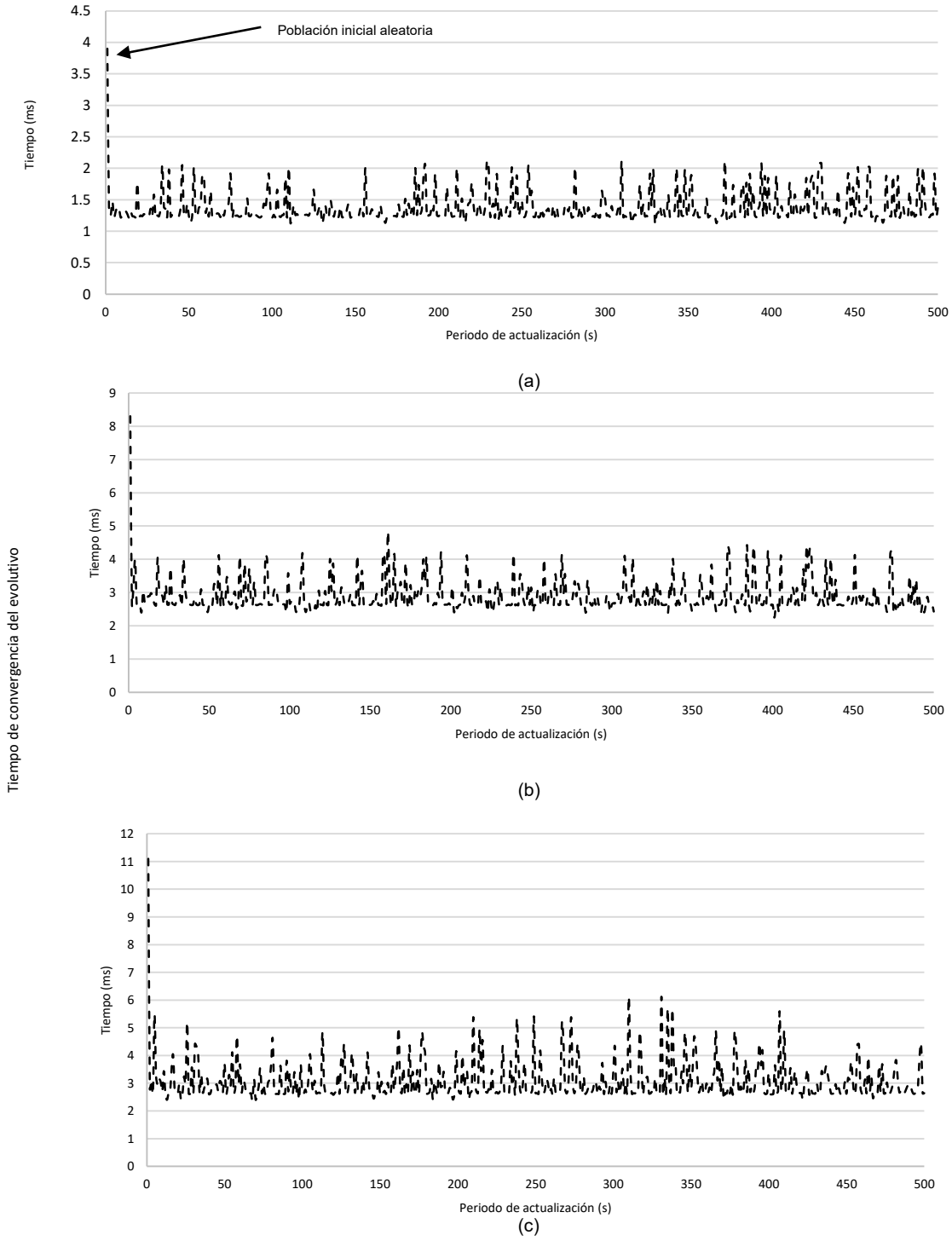


Figura 5.5: Tiempo que tarda en converger el algoritmo genético en cada actualización. (a) Tabla IPv4 con 5 etapas de búsqueda LPM. (b) Tabla IPv6 real con 18 etapas de búsqueda LPM (c) Tabla IPv6 sintética con 18 etapas de búsqueda LPM (fecha de 20 de marzo de 2019 .

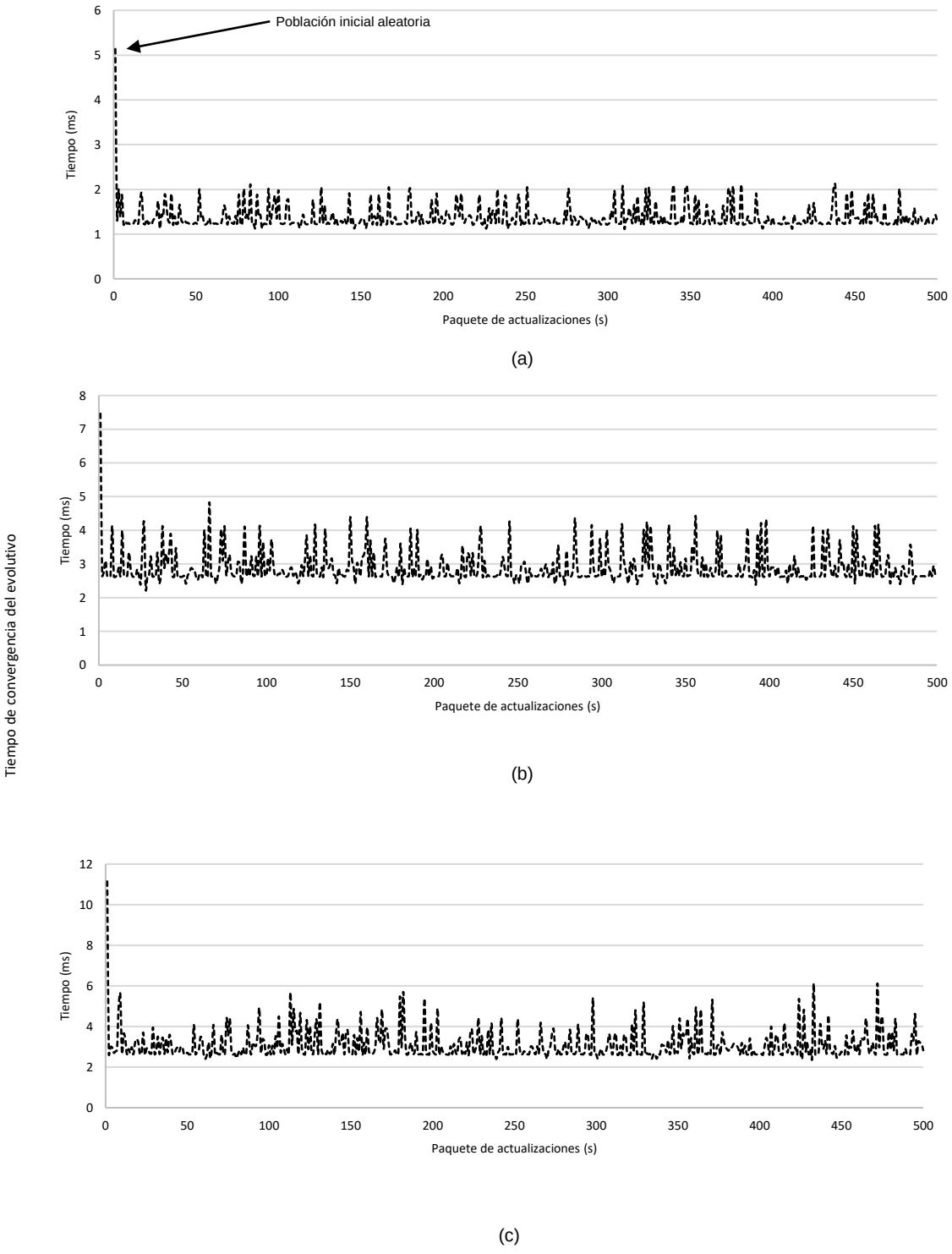


Figura 5.6: Tiempo que tarda en converger el algoritmo genético en cada actualización. (a) Tabla IPv4 con 5 etapas de búsqueda LPM. (b) Tabla IPv6 real con 18 etapas de búsqueda LPM (c) Tabla IPv6 sintética con 18 etapas de búsqueda LPM (fecha de 15 de enero de 2021.

Cuadro 5.13: Rangos de costos totales de memoria requerida para las tablas de encaminamiento utilizadas durante 500 segundos de actualización.

Tabla	Fecha	Costo mínimo (MB)	Costo Máximo (MB)
IPv4 Real	20/03/2019	5.5	5.7
IPv6 Real	20/03/2019	4.57	4.63
IPv6 Sintética	20/03/2019	22.1	22.5
IPv4 Real	15/01/2021	7.8	7.86
IPv6 Real	15/01/2021	7.93	8.2
IPv6 Sintética	15/01/2021	27.4	28.6

Cuadro 5.14: Rangos de costos totales de memoria requerida para las tablas de encaminamiento utilizadas durante 500 segundos de actualización.

Tabla	Fecha	Costo mínimo (MB)	Costo Máximo (MB)
IPv4 Real	20/03/2019	3.5	3.6
IPv6 Real	20/03/2019	3.23	3.52
IPv6 Sintética	20/03/2019	16.1	16.8
IPv4 Real	15/01/2021	5.76	6.21
IPv6 Real	15/01/2021	6.32	6.99
IPv6 Sintética	15/01/2021	18.1	18.94

comportamiento. Por otro lado mostramos en el Cuadro 5.14 los costos totales obtenidos, donde se observa una disminución consistente de costos con respecto al Cuadro 5.13 que solo considera un arreglo contador por *trie*.

5.3. Desempeño del esquema propuesto *vs* otras propuestas

En secciones anteriores hicimos un análisis de las propuestas que presentamos en el capítulo anterior; dicho análisis consistió en los posibles escenarios para implementar nuestra propuesta; esto porque, generalizamos uno de los mecanismos base codificación de vector de bits, dicho de otra manera, podemos elegir la configuración más adecuada

dependiendo de las características del dispositivo de encaminamiento (básicamente el tamaño de la memoria de rápido acceso *cache*).

El encaminamiento de un paquete se puede realizar de diferentes maneras como lo son: RIP (*Routing information protocol*, protocolo de información de encaminamiento); es un protocolo de encaminamiento interno, es decir para la parte interna de la red, la que no está conectada al backbone de Internet. OSPF (*Open shortest path first*, el camino más corto primero); OSPF se usa, como RIP, en la parte interna de las redes, su forma de funcionar es bastante sencilla. BGP (*Border gateway protocol*, protocolo de la puerta externa) BGP es un protocolo muy complejo que se usa en la interconexión de redes conectadas por un backbone de Internet. Existen propuestas para medir el desempeño del tipo encaminamiento que se utiliza como el que se presenta en [54]. Sin embargo, no la podemos usar ya que nuestro problema se encuentra en el modo BGP y depende en su totalidad de cada dispositivo de encaminamiento, dirección destino, y prefijos en la tabla.

De esta manera y por lo tanto, en esta sección comparamos algunas características de nuestra propuesta con características similares de las innovaciones presentadas en el capítulo 3; las características que consideramos relevantes en el desempeño de un esquema de búsqueda LPM son: el tipo de direccionamiento o que soportan IPv4, IPv6 o ambas. El tiempo de configuración de estructuras de datos necesitadas en la búsqueda. El desempeño en memoria y finalmente el número de búsquedas LPM que se pueden realizar en un periodo de tiempo.

5.3.1. Comparaciones por tipo de direccionamiento

Una de las formas más sencillas de comparar esquemas de búsqueda LPM puede ser el tipo de direccionamiento para el cual estén diseñados y operan de acuerdo a sus especificaciones reportadas. En el Cuadro 5.15 se muestran las propuestas que especifican con que versión de IP operan adecuadamente. Tenga en cuenta que, aunque la migración a IPv6 está en un punto avanzado, IPv4 sigue en uso y es igualmente necesario el mejoramiento de esquemas de búsqueda que operen en dicho protocolo.

Cuadro 5.15: Comparación por tipo de direccionamiento IP.

Referencia	Tipo	Direccionamiento
Propuesta de tesis	Codificación del *trie* y partición eficiente en etapas	IPv4/6
[36]	Árbol de búsqueda heredado múltiple (MIST)	IPv4
[37]	Optimizacion de acuerdo al longitud y densidad del prefijo	IPv4/6
[39]	Árbol de búsqueda equilibrada	IPv4/6
[40]	*Trie* codificado en SDRAM	IPv4/6
[47]	*Trie* codificado arquitectura CUDA	IPv4
[51]	*Hash Trie*	IPv4/6

5.3.2. Desempeño en tiempo de actualización

El desempeño en tiempo de actualización de estructuras de datos utilizadas en la búsqueda LPM puede interpretarse de dos maneras: el tiempo necesario para construir las estructuras de datos y el tiempo necesario para determinar cuáles son los parámetros adecuados de operación del esquema. A continuación realizamos ambas comparaciones con la literatura expuesta.

5.3.2.1. Tiempo para construir las estructuras de datos

Cada vez que se realiza una actualización en la tabla de encaminamiento las estructuras de datos utilizadas en la búsqueda sufren cambios, en el peor de los casos se deben reconstruir en su totalidad, tal es el caso de nuestra propuesta y demás propuestas que utilizan prefijos disjuntos. La principal aportación de [39] es reducir el tiempo de actualización de 1.33 a 6.88 que algoritmos que utilizan el *trie* original. Sin embargo, nuestra propuesta general de conteos precomputados tiene una complejidad obtenida en el Capítulo 4 de 2^K instrucciones (donde K es la profundidad del *trie*) para llenar todos los arreglos contadores, por lo que es viable construir los arreglos en cada actualización.

5.3.2.2. Tiempo para determinar los parámetros adecuados de operación del esquema

En la literatura encontramos tres propuestas que utilizan un mecanismo para determinar la mejor manera de expresar las estructuras utilizadas en una búsqueda LPM en termino de optimización en memoria. Dos de ellas utilizan programación dinámica y la otra es nuestra propuesta basada en exploración heurística. En [45] comprueban todas las posibilidades de compresión de un *trie,* mientras que en [18] se comprueban todas las posibles maneras de segmentar la búsqueda LPM. En ambos casos la programación dinámica ayuda a no repetir operaciones redundantes por comprobación. Sin embargo, en el mejor caso de deberá realizar una instrucción básica por comprobación. Si pensamos en un *trie* de profundidad K el cual se va a codificar en h segmentos o a dividir en h etapas. El número de comprobaciones corresponde a un problema combinatorio con $\frac{(K-1)!}{(h-1)!(K-h)!}$ posibles resultados. Nosotros propusimos un esquema de búsqueda mediante exploración heurística en la cual no es necesaria la comprobación de todas las posibilidades pero que tiene la desventaja intrínseca de no garantizar un mínimo global.

5.3.3. Desempeño en memoria

Puede decirse que la finalidad de un esquema de búsqueda LPM es reexpedir todos los paquetes que arriban a él sin que se produzca un cuello de botella. De esta manera, el número de paquetes que dé deben reexpedir es proporcional al ancho de banda de los enlaces de comunicación del dispositivo de encaminamiento. Sin embargo, recuerde que el desempeño de un algoritmo de búsqueda LMP está estrechamente relacionado con la latencia de la memoria donde son contenidas las estructuras de consulta.

Usualmente el tamaño total requerido en memoria depende del número y distribución de prefijos; por lo que sería difícil comparar resultados reportados en diferentes propuestas dado que las tablas de encaminamiento utilizadas no son iguales. Ademas que, existe una relación entre memoria e instrucciones de búsqueda; en el peor caso en consumo en memoria suele ser el mejor caso en instrucciones de búsqueda LPM y viceversa.

Las propuestas que utilizan como referencia el *trie* binario realizan una búsqueda parcial sobre un segmento de dirección IP para después consolidar etapas. Una manera de

Cuadro 5.16: Ahorro en memoria *vs trie* completo.

Referencia	Propuesta	Porcentaje de ahorro
Propuesta de tesis	Codificación del *trie*	55 %
[37]	Optimizacion de acuerdo al longitud y densidad del prefijo	87 %
[39]	Árbol de búsqueda equilibrada	del 4 % al 79 %
[42]	compresión del *trie* en una TCAM	50 %
[51]	*Hash Trie*	77 % IPv4 y 86 % IPv6

determinar el ahorro en memoria de algún esquema basado en el *trie* binario, es tomar todos los posibles nodos del *trie;* donde un nodo equivale a un bit y comparar dicho esquema. El Cuadro 5.16 muestra el ahorro reportado de 5 esquemas que utilizan el *trie* como referente de comparación. Nuestra propuesta presenta un ahorro del 55 %. Nosotros seleccionamos el *trie* de profundidad 32 y calculamos todos sos nodos, mismos que representan 1GB, tomamos después el costo del Cuadro 5.8 con $K = 32$ y $h = 4$ que es de 468.74MB obteniendo el ahorro presentado. Note que esta manera de homogeneizar diferentes esquemas puede darnos un panorama de la eficacia en memoria de diferentes propuesta con algún parecido. Sin embargo, solo nuestra propuesta y la que utiliza memorias TCAM [42] son independientes a la distribución y número de prefijos.

5.3.4. Desempeño de búsquedas por unidad de tiempo

El Cuadro 5.17 muestra el número de búsquedas realizadas por unidad de tiempo que documentan algunas de las propuestas presentadas en el estado del arte. Tenga en cuenta que, aunque pareciera que esta es la prueba más importante y sencilla de desempeño, depende del escenario en donde se hayan realizado. Por ejemplo, en [37] y en [40] no reportan el escenario de prueba. Sin embargo, ya que estos esquemas dependen en su totalidad de la distribución y número de prefijos, podemos inferir que, el desempeño se verá afectado por una tabla dada; además que, después de la adopción de IPv6 se esperan escenarios con grandes cantidades de prefijos y sobre todo distribuidos uniformemente en sus posibles valores y longitudes. También debe considerase que la rapidez de una

Cuadro 5.17: Búsquedas por unidad de tiempo.

Referencia	Propuesta	Millones de búsquedas por unidad de tiempo
Propuesta de tesis	Codificación del *trie*	40
[37]	Optimizacion de acuerdo al longitud y densidad del prefijo	588
[40]	*Trie* codificado en SDRAM	400
[46]	*Trie* codificado en arquitectura CUDA	5
[47]	*Trie* codificado en arquitectura multi core	1.3

búsqueda LPM depende del valor de la dirección IP destino del paquete. Dicho de otra manera, si por ejemplo realizamos búsquedas LPM según el *ranking alexa*, más del 99 % será sobre los dominios de apenas 10 proveedores de servicios en Internet. Por consiguiente, es justificable la diferencia entre los valores presentados en el cuadro.

Nuestra propuesta puede realizar alrededor de 40 millones de búsquedas LPM en un segundo dentro de un escenario IPv4 de dos etapas donde, en cada etapa son codificados los *subtries* binarios en arreglos contadores de tamaño constante independiente a la distribución de prefijos. También las búsquedas LPM se realizan con direcciones IP distribuidas uniformemente en sus posibles valores y los cálculos fueron realizados en el escenario de los Cuadros 5.9 y 5.10.

Las propuestas que realizan búsquedas paralelas en las etapas del *trie* aprovechando los *cores* de una tarjeta de video o bien a los núcleos de una computadora muestran un desempeño menor a los presentados. Sin embargo, presentan resultados con escenarios más descriptivos como el expuesto en [46] donde se presentan las características de la tarjeta GPU, una tabla IPv4 con 350K prefijos y un tamaño independiente a la distribución de prefijos en la codificación de los *subtries* binarios.

Capítulo 6

Análisis y discusión

El esquema de búsqueda LPM que proponemos considera dos optimizaciones importantes en el consumo de memoria; una de ellas consiste en seccionar la búsqueda IPv4/6 en etapas dinámicas seleccionadas con base en la distribución de prefijos por un algoritmo de exploración heurística. La otra optimización opera en cada etapa, en donde segmentos de prefijos de red en la tabla de encaminamiento están codificados en arreglos contadores cuyo tamaño es optimizado. En este capítulo realizamos la discusión de sus cualidades, trabajo pendiente, objetivos alcanzados, flexibilidad y posibles aplicaciones en otros problemas.

6.1. Esquema generalizado de conteos distribuidos de un vector de bits

Un resultado interesante en el que no se profundizó fue determinar la relación entre el número de conteos en el vector de bits y el ahorro de memoria en los mismos, esto significa que mientras más haya conteos precumputados en el vector de bits no implica una ganancia mayor ahorro en memoria por guardar dichos conteos, de hecho, para *tries* de cualquier profundidad incluso hasta 128, más de 4 conteos precomputados no representan una ganancia significativa en memoria. Para el problema de búsqueda LPM en IPv4/6 fijamos el número de conteos precumputados a un número menor a cuatro y esto nos

permitió tener una optimización adicional en instrucciones de acceso a memoria.

Otro aspecto en el que se puede profundizar es el siguiente. Para cualquier forma de codificación del vector de bits que proponemos, se utilizan uno o más arreglos contadores que llevan un conteo acumulativo del número de bits puestos en 1. Esto significa que es un conteo progresivo donde el menor valor posible forzosamente se encontrará en la primera entrada del arreglo, mientras que, el valor más grande posible solo puede estar en la última entrada. Dicho de otra manera, para la primera casilla sólo se puede contar hasta un 1, para la segunda hasta dos bits puestos en 1 y así sucesivamente en una casilla i sólo se pueden contar hasta i bits puestos en 1. Sin embargo, todas las casillas tienen la misma longitud que está determinada por el número total de posibles unos en la sección del vector de bits, por lo tanto existen casillas que tienen un número de bits mayor de los que se necesitan para contar. En la figura 6.1 se muestra un contador donde todos sus elementos tienen la misma longitud y una optimización donde cada casilla tiene el número exacto de bits para contar. Ésta es una forma de optimizar aún más el espacio en memoria de los algoritmos propuestos y se podría definir de nuevo a las funciones de costo en bits para encontrar una vez más los parámetros óptimos de operación.

El vector de bits se construye a partir de un *trie* binario de prefijos disjuntos y el *trie* a su vez se construye a través de una tabla de encaminamiento. Sin embargo, una vez construido el vector de bits se pueden hacer conteos distribuidos en él a través de arreglos contadores. Nosotros mapeamos los arreglos contadores a niveles del *trie* binario para determinar el número de entradas y tamaño de cada una de ellas pero, el *trie* es meramente ilustrativo, de hecho el Algoritmo 4.1 de llenado de arreglos no hace en ningún momento mención del *trie* binario.

Este resultado no fue considerado pero puede ser uno de los más importantes en este trabajo, dado cualquier vector de bits (ya sea que haya o no en él patrones de bits), nosotros pudimos establecer un mecanismo de conteos de bits puesto en 1 que puede ser ajustado para optimizar en memoria o instrucciones. Este resultado es muy interesante puesto que un vector de bits puede significar muchas cosas en distintas disciplinas, por ejemplo [28] donde un vector de bits representa eventos en redes cognitivas. Nuestra generalización es un nicho amplio de posibilidades y puede ser un componente importante en muchas optimizaciones que tengan que ver con el uso eficiente de memoria. Como

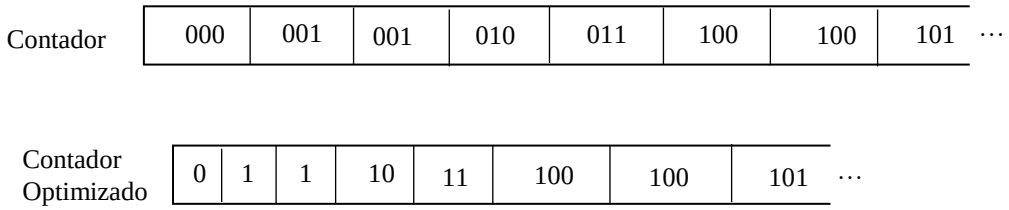


Figura 6.1: Ejemplo de optimización de un arreglo contador de 3 bits por casilla.

trabajo futuro proponemos buscar aplicaciones que usen conteos de vectores de bits e incorporar nuestra propuesta para evaluar su desempeño .

6.2. Esquema completo de búsquedas LPM

Nuestra propuesta considera dos problemas significativos de esquemas de búsquedas LPM que utilizan al *trie* binario: la codificación de un *subtrie* y la partición en etapas de la búsqueda LPM, ambas con el fin de obtener una mayor eficiencia en memoria requerida. Sin embargo, hay que considerar a todas las estructuras que se utilizan en el esquema, así como su consolidación y actualización. En seguida detallamos las consideraciones que son tema de investigación para representar, manejar y hacer operaciones en tablas de encaminamiento, *trie* binario, y arreglos contadores.

6.2.1. Tabla equivalente de prefijos disjuntos

El *trie* binario de prefijos disjuntos que utilizamos para la construcción del vector de bits es la estructura de datos que representa a la tabla equivalente de prefijos disjuntos. Cuando se da una actualización ya sea de inserción o eliminación de algún prefijo, la tabla equivalente puede ser actualizada en una o hasta en todas sus entradas en el peor de los casos, esto significa la reconstrucción de todo el *trie* binario. Por lo tanto es un problema de optimización abierto para nuestra propuesta y todas aquellas que utilicen *Leaf-Pushing*.

Cuando la tabla equivalente de prefijos disjuntos tiene alguna actualización, el número de valores puestos en 1 en el vector de bits cambia y con ello, el conteo distribuido en los arreglos de conteos precomputados también ha de cambiar. Al igual que en la tabla equivalente un conteo precumputado puede modificar una o todas las entradas de un arreglo contador. Es importante considerar como trabajo futuro brindar una optimización que considere las actualizaciones de los arreglos contadores.

6.2.2. Uso de estructuras auxiliares

A lo largo del texto hemos utilizado *tries* binarios de diferentes tipos: para representar prefijos, para representar prefijos disjuntos, para determinar el tamaño y características de arreglos contadores, inclusive para representar cromosomas en el algoritmo evolutivo. Sin embargo, recalcamos que todos los *tries* son ilustrativos, lo único que necesitamos para todo nuestro esquema son las tablas de prefijos y los arreglos contadores, inclusive el vector de bits es desechado una vez que se han llenado los arreglos de conteos precomputados. Es un nicho de investigación la creación, manejo e inclusive substitución del *trie* binario como forma de representación de los prefijos de red en la tabla de encaminamiento.

6.3. Análisis de metas alcanzadas

Tablas de encaminamiento reales Tanto para el esquema de distribución de conteos en un vector de bits, así como, para el esquema de búsqueda LPM en etapas, son necesarias tablas de encaminamiento IPv4/6. En el primer caso, utilizamos dichas tablas

para construir el vector de bits y para el segundo caso, son necesarias para considerar la distribución de los prefijos en cada etapa. Nosotros utilizamos tablas IPv4/6 del dispositivo de encaminamiento global [21]. Sin embargo, una tabla y sus actualizaciones es una serie de archivos binarios comprimidos; por lo que, para hacer uso de la misma, tuvimos que utilizar y modificar la herramienta de desarrollo que se brinda en el proyecto de la Universidad de Oregon. De esta manera, nosotros logramos obtener, procesar y estudiar tablas de encaminamiento reales de routers globales. Dado que la base de datos utilizada es un archivo detallado desde hace caso 20 años, resulta interesante analizar la evolución de las tablas de encaminamiento y el desempeño de propuestas de búsqueda LPM vistas en este documento.

Tablas IPv6 sintéticas Para obtener tablas IPv6 sintéticas utilizamos *V6Gene* [23]; con esto logramos simular escenarios después de la migración de IPv6, utilizamos y modificamos dicha herramienta logrando obtener inclusive escenarios que pondrían donde exista una gran dispersión de prefijos. El generador utilizado, toma una semilla (tabla IPv6 real) para generar un sintética .

Operaciones sobre el *trie* binario Para analizar la distribución de prefijos en tablas de encaminamiento IPv4/6 reales o sintéticas desarrollamos una librería que posibilita la conversión de tablas de encaminamiento en *tries* y *subtries* binarios. Las funciones de la librería van desde recorridos, inserción, eliminación, actualización etc.

Recuerde que en nuestra propuesta, el *trie* es utilizado como referencia para explicar adecuadamente el proceso de codificación del vector de bits en arreglos contadores; también lo utilizamos también para representar la segmentación en etapas. Sin embargo, nosotros de manera paralela construimos el *trie*, he hicimos en él las operaciones base para realizar recorridos, actualizaciones, hacer prefijos disjuntos etc. Es importante hacer notar que nuestra propuesta y las demás descritas en este documento parte del supuesto que la tabla de encaminamiento esta en memoria y que cada unidad de tiempo se actualiza de forma inmediata. Después de ello, cada propuesta configura sus propios tipos de datos para la búsqueda. Entonces el problema distribuido para obtener la a los prefijos y el problema local del procesamiento de actualización de la tabla es un reto que también

se presenta en la búsqueda LPM y una de las metas a futuro de este trabajo debe ser la propuesta estructuras de datos que representen a la tabla donde se puedan realizar actualizaciones y búsquedas LPM de forma optimizada.

Acceso a IPv6 Conseguimos acceder a sitios y servicios IPv6 utilizamos túneles *teredo* [11] por lo cual instalamos y configuramos la herramienta. La presente investigación se realizó en la CDMX teniendo como limitante el uso único de la red IPv4, con lo cual la experiencia en IPv6 fue un limitada. Un experimento sencillo pero significativo podría ser un *test* de la capacidad de respuesta de sitios web en ambos protocolos.

Prefijos disjuntos en tablas de encaminamiento Logramos simular el algoritmo de *leaf-pushing* [4] para obtener prefijos disjuntos. Sin embargo, algo que debe considerarse en las propuesta que utilizan dicho algoritmo, es que deben sumar su latencia de ejecución a la latencia de formación de estructuras de datos de búsqueda LPM. Dicho de otra manera, utilizar prefijos disjuntos ayuda a disminuir el tiempo de búsqueda pero hay que considerar que existe una latencia para conseguir dichos prefijos disjuntos.

En nuestro caso, la complejidad en instrucciones para obtener prefijos disjuntos y de ellos obtener el vector de bits es la misma que *leaf-pushing,* donde su principal desventaja es que cuando se realiza una actualización en la tabla de encaminamiento, se debe reconstruir los prefijos disjuntos y el vector de bits.

Esquema general de distribución de conteos Comprobamos la tesis de un algoritmo general de distribución de conteos precomputados del vector de bits. El producto de dicho algoritmo son los arreglos contadores de consulta optimizados en memoria. Realizamos experimentos para diferentes codificaciones y comprobamos que.

1. Nuestra propuesta es independiente a la distribución de prefijos e inclusive al vector de bits, lo que significa que es un esquema general de conteos de eventos en un vector de bits. Dicha afirmación se sostiene en las definiciones de las ecuaciones en memoria e instrucciones explicadas en el capítulo 4.

2. Que la optimización en memoria es parte fundamental en la búsqueda LPM; propusimos escenarios cuya latencia de búsqueda es menor, aún cuando su complejidad en

instrucciones es mayor en comparación con codificaciones optimizadas en instrucciones.

Patrones de bits en un vector Demostramos que en el caso de un vector de bits construido a partir de prefijos disjuntos, existen patrones posibles; nosotros caracterizamos dichos patrones y programamos el generador de patrones de bits posibles y el algoritmo de llenado de la tabla diccionario.

Comprobación de escenarios posibles de partición y codificación Programamos un algoritmo recursivo generador de combinaciones posibles que utilizamos para distribuir n conteos parciales en un *trie* de profundidad K. En cada combinación aplicamos las ecuaciones de costo de memoria del capítulo 4 para determinar el o los mínimos costos en memoria. Con ello logramos comprobar todas las posibilidades expresadas y demostramos que hay mínimos globales independientes a la distribución de prefijos e inclusive al los valores de cualquier vector de bits.

También programamos un generador de combinaciones posibles para el esquema de partición en etapas de la búsqueda LPM. Este generador considera los posibles costos y determina a fuerza bruta la mejor partición. Con ello logramos corroborar los resultados de nuestra propuesta de usar algoritmos evolutivos.

Búsquedas LPM en base a direcciones IP uniformemente distribuidas Aunque en Internet la mayoría de las consultas se realicen en una cantidad reducida de dominios, nosotros probamos nuestro esquema utilizando direcciones uniformemente distribuidas. Desarrollamos un simulador de eventos (direcciones IP aleatorias) con el fin de hacer pruebas de desempeño en diferentes *tries,* de diferentes profundidades, diferente número de arreglos contadores y parámetros; con ello demostramos la eficiencia de nuestro esquema y obtuvimos los resultados del capítulo anterior.

Exploración heurística Propusimos la técnica de algoritmos genéticos como mecanismo de exploración heurística. Sin embargo; exploramos otras técnicas como recocido simulado, donde obtuvimos latencias mayores a los periodos de actualización de la tabla

de encaminamiento. No podemos descartar otras técnicas similares pero demostramos su eficacia en comparación de esquemas de exploración determinista.

Esquema completo de búsqueda LPM Consolidamos un esquema completo de búsqueda LPM que va desde la tabla de prefijos disjuntos ya sea IPv4 o bien IPv6, hasta la búsqueda del prefijo de mayor coincidencia. Programamos el simulador completo que considera todo el proceso y todos los algoritmos antes mencionados: desde cargar una tabla IPv4/6 hasta realizar búsquedas LPM.

Publicaciones Finalmente escribimos y sometimos a publicación el trabajo Genetic Algorithm Based Prefix Partitioning for Memory-Efficient IPv4/6 Lookup [26].

6.4. Esquema de Lulea

El esquema de *Lulea* fue uno de los puntos de partida de esta investigación. Inicialmente nosotros nos cuestionamos el valor de sus parámetros ya que este esquema tiene un gran desempeño en memoria. Sin embargo, más allá de utilizar parcialmente su idea para aplicarla a IPv6, nosotros propusimos una generalización de distribución de conteos precumputados de cualquier vector de bits; de esta manera el esquema *Lulea* se convirtió en un caso particular de nuestra propuesta.

El esquema de *Lulea*, el *trebitmap* o los esquemas *multi-trie* son referencias para las propuestas actuales. En la literatura podemos observar que se realizan mejoras a estos utilizando hardware de propósito especifico o conceptos como filtros *bloom* y tablas *hash*. De esta manera, nosotros no hacemos una mejora conjunta con otras propuestas, notros proponemos replantear uno de los esquemas base y generalizarlo; esto significa que, gran parte de la propuestas que utilizan una codificación del *trie* binario pueden reportar mejoras en su desempeño si utilizan nuestra generalización. Nuestra propuesta posibilita una nueva discusión a los esquemas más actuales.

6.5. Consideraciones generales

Pudimos demostrar que los algoritmos evolutivos son una buena opción para determinar las características de las etapas que optimicen en memoria una búsqueda LPM. Nosotros utilizamos algoritmos evolutivos con una programación secuencial pero es posible optimizar el desempeño del algoritmo genético utilizando técnicas de programación paralela e inclusive distribuida. Al momento de realizar este trabajo no encontramos propuestas que incluyeran la búsqueda heurística o alguna técnica de inteligencia artificial utilizadas para optimizar en memoria alguno de los problemas contenidos en la búsqueda LPM. Debido a esto, se pueden analizar otras técnicas que mejoren aún más nuestra propuesta.

Para validar nuestra propuesta en IPv6 utilizamos tablas sintéticas creadas a partir de tablas reales; es necesario un análisis constante de tablas de encaminamiento mientras e incluso después de la migración a este nuevo protocolo.

Capítulo 7

Conclusiones

En este trabajo de tesis se abordó uno de los muchos problemas que existen al enviar información por una red de computadoras. Para el desarrollo del proyecto situamos el problema en la capa de red del modelo de TCP/IP. Estudiamos el proceso de encaminamiento de la información, y observamos el problema que debe resolver cada dispositivo de encaminamiento cuando arriba un paquete a él; ello realizando la denominada búsqueda LPM para reexpedir dicho paquete a una red que lo acerque a su destino. Estudiamos el estado de la adopción de IPv6 en la actualidad; obtuvimos, revisamos, e interpretamos tablas de encaminamiento IPv4/6 correspondientes a dispositivos de encaminamientos globales. También generamos tablas sintéticas IPv6 a partir de las reales para modelar escenarios después de la adopción de este protocolo.

Analizamos a detalle todas las características que hacen del esquema de *lulea* un esquema de alto desempeño en el uso de memoria y baja latencia en instrucciones de búsqueda LPM. A partir de ello, planteamos una generalización para realizar búsquedas LPM para segmentos de direcciones IP de cualquier tamaño. El esquema de búsqueda LPM propuesto garantiza una optimización en memoria y en instrucciones de acceso a la misma. Este esquema consiste en generar un vector de bits a partir de una tabla de prefijos disjuntos y distribuir el conteo de sus valores puestos en 1 en uno o más arreglos de conteos precomputados. La propuesta también considera conteos precomputados de patrones de bits válidos en el vector de bits. Entre las principales desventajas se encuentra la complejidad exponencial en la memoria requerida. Sin embargo, nuestro esquema generalizado se pue-

de adaptar de acuerdo al tamaño de la memoria y la latencia de acceso utilizadas en la implementación, esto significa que el rendimiento de nuestra propuesta se puede adaptar al desarrollo tecnológico de memorias. Ademas, el principio de funcionamiento de nuestro trabajo se puede utilizar en diferentes esquemas de búsqueda de IPv4/6 y puede implementarse en hardware de propósito general. Ya que el mecanismo de distribución de los valores del vector de bits en arreglos contadores no tienen relación alguna con las estructuras de la tabla de encaminamiento y el *trie* binario, nuestra propuesta es un mecanismo general optimizado en memoria de conteos de valores 1 de un vector de bits cualquiera.

La complejidad exponencial de búsquedas LPM utilizando conteos precomputados hace inviable realizar una búsqueda completa en IPv6. Nosotros proponemos realizar la búsqueda LPM en etapas, esto realizando una partición eficiente en memoria requerida de los segmentos de direcciones IP en cada etapa. Nuestra segunda contribución está basada en exploración heurística para determinar el tamaño de los segmentos de direcciones IP en cada etapa, mientras que otras soluciones utilizan métodos exhaustivos como la programación dinámica. La principal ventaja que encontramos en nuestra propuesta es la baja latencia y la alta calidad de soluciones de la exploración heurística, además de un mapeo directo entre estructuras de datos del esquema de búsqueda LPM y el algoritmo genético. Mientras que la desventaja evidente de la exploración heurística es no poder ofrecer una solución que garantice ser óptima. Sin embargo, nuestra propuesta demuestra ser eficiente en la optimización de memoria requerida en distintos modelos de direccionamiento IPv4 e IPv6, así como modelos sintéticos IPv6 con mayores requerimientos. La latencia del algoritmo genético es muy pequeña en comparación con la precisión de los periodos de actualización de la base de datos utilizada. Además, como trabajo futuro, se podría mejorar su desempeño de ejecución si su implementación se basara en programación paralela. Determinamos que cuando se utiliza una partición optimizada, las estructuras de datos utilizadas pueden ser contenidas en memorias de rápido acceso tanto para tablas IPv4 reales así como para tablas IPv6 reales y sintéticas. El escenario propuesto para validar nuestra propuesta fueron tablas de encaminamiento IPv6 e IPv4 actuales de un dispositivo de encaminamiento público global y en el caso de IPv6 creamos una tabla sintética para simular el futuro de IPv6.

Bibliografía

[1] Redes De Computadoras (7ª ed.), James F. Kurouse; Keith W.Ross, 2017.

[2] https://tools.ietf.org/html/rfc791, IETF Document Search, fecha de acceso 21/05/2021.

[3] https://tools.ietf.org/html/rfc2460, IETF Document Search, fecha de acceso 21/05/2021.

[4] Miguel Á. Ruiz-Sánchez. Survey and Taxonomy of IP Address Lookup Algorithms. IEEE Network, 15(2):87–23, abril-mayo 2001.

[5] IEEE 802.3 Ethernet Working Group, "IEEE P802.3ba 40 Gb/s and 100Gb/s Ethernet Task Force," 2010 [Online]. Available: http://www. ieee802.org/3/ba/.

[6] John Hennessy and David Patterson, Arquitectura de Computadores, Un enfoque cuantitativo. 1a edición, capítulo 8.

[7] https://www.google.com/intl/es/ipv6/index.html, Google IPv6, fecha de acceso 21/05/2021.

[8] Degermark, Mikael; Brodnik, Andrej; Carlsson, Svante; Pink, Stephen (1997), "Small forwarding tables for fast routing lookups", Proceedings of the ACM SIGCOMM 1997 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pp. 3-14.

[9] T. Bates, P. Smith, and G. Huston, "CIDR report," [Online]. Available: http://www.cidr-report.org/.

[10] http://www.worldipv6launch.org/, World IP/v6 Launch, fecha de acceso 21/05/2021.

[11] https://www.freenet6.net/, Prediksi Togel Singapura, fecha de acceso 21/05/2021.

[12] http://www.google.com/intl/es/ipv6/statistics.html, Google IPv6, fecha de acceso 21/05/2021.

[13] http://www.routeviews.org/, University of Oregon Route Views Project, fecha de acceso 21/05/2021.

[14] Available: http://www.cidr-report.org/, Geoff Huston, fecha de acceso 21/05/2021.

[15] https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram

[16] John Hennessy – David Patterson Arquitectura de Computadores – Un enfoque cuantitativo (1a edición, capítulo 8).

[17] Eatherton, W., Varghese, G., and Dittia, Z. 2004. Tree bitmap: hardware/software IP lookups with incremental updates. SIGCOMM Comput. Commun. Rev. 34, 2, Apr. 2004, 97-122.

[18]] Hoang Le and Viktor K. Prasanna; "Scalable Tree-Based Architectures for IPv4/v6 Lookup Using Prefix Partitioning", IEEE Transactions on Computers, Vol. 61, No. 7, July 2012.

[19] Darwin, C. On the Origin of Species by Meansof Natural Selectionor the Preservations of Favoured Racesin the Struggle for Life. 1859. London: John Murray.

[20] D. E. Goldberg. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1989. Genetic Algorithms in Search, Optimization and Machine Learning.

[21] http://bgp.potaroo.net/index-bgp.html, Geoff Huston, fecha de acceso 21/05/2021.

[22] Joshua Ruggiero, "Measuring Cache and Memory Latency and CPU to Memory Bandwidth", White paper, Intel Corporation, December 2008.

[23]] Kai Zheng. “V6Gene: a scalable IPv6 prefix generator for route lookup algorithm benchmark”. IEEE International Conference on. Pages 18-20. April 2006.

[24] Jesus Marco Vivas Ruiz, Carlos Silva Cardenas and Jose Luis Muñoz Tapia, “Implementation and Testing of IPv6 Transition Mechanisms”, IEEE 9th Latin-American Conference on Communications (LATINCOM), 2017.

[25] Jesus Marco Vivas Ruiz, Carlos Silva Cardenas and Jose Luis Muñoz Tapia, “Review of Approaches for the use of the Label Flow of IPv6 Header”, IEEE Latin America Transactions, Volume: 12, Issue: 8, Dec. 2014.

[26] F. U. Sánchez, M. A. Ruiz and C. Jalpa. Genetic Algorithm Based Prefix Partitioning for Memory-Efficient IPv4/6 Lookup. IEEE Latin America Transactions, Volume: 17, Issue: 11s, November. 2019.

[27] Bando, M. “FlashTrie: Beyond 100-Gb/s IP Route Lookup UsingHash-Based Prefix Compressed Trie”. Networking, IEEE/ACM Transactions on. Aug. 2012.

[28] C. Salgado, H. López, Cesar Hernandez, E. Rodriguez-Colina, “Multivariable algorithm for dynamic channel selection in cognitive radio networks,” in EURASIP Journal on Wireless Communications and Networking (EURASIP JWCN), Science Citation Index (JCR), ISSN: 1687-1499.

[29] V. Srinivasan and G. Varghese, “Fast Address Lookups Using Controlled Prefix Expansion,” ACM Trans. Computer Systems, vol. 17, pp. 1-40, 1999.

[30] Rowan Garnier; John Taylor (2009). Discrete Mathematics: Proofs, Structures and Applications, Third Edition. CRC Press. p. 620. ISBN 978-1-4398-1280-8.

[31] D. E. Taylor, J. S. Turner, J. W. Lockwood, and T. S. Sproull, “Scalable IP Lookup for Internet Routers,” IEEE J. Sel. Areas Commun., vol.21, no.4, pp.522–534, May 2003.

[32] R. Sangireddy and A. K. Somani, “High-Speed IP Routing with Binary Decision Diagrams Based Hardware Address Lookup Engine,” IEEE J. Sel. Areas Commun., vol.21, no.4, pp.513–521, Apr. 2003.

[33] W. Lu and S. Sahni, “Recursively Partitioned Static IP Router-Tables,” IEEE Trans. Comput., vol.59, no.12, pp.1683–1690, Dec. 2010.

[34] H. Lu, K. Kim and S. Sahni, “Prefix and Interval-Partitioned Dynamic IP Router-Tables,” IEEE Trans. Comput., vol.54, no.5, pp.545–557, May 2005.

[35] Hyesook Lim, Nara Lee. Survey and Proposal on Binary Search Algorithms for Longest Prefix Match. IEEE Communications Surveys & Tutorials. Vol 14, No 3, 2012.

[36] Cheng Hsu, Sun-Yuan Hsieh. Multi-Inherited Search Tree for Dynamic IP Router-Tables. IEEE Transactions on Computers. Vol 66, No 1, 2017.

[37] Thibaut Stimpfling. Normand Bélanger, J. M. Pierre Langlois, Yvon Savaria. SHIP: A Scalable High-Performance IPv6 Lookup Algorithm That Exploits Prefix. IEEE/ACM Transactions on Networking. Vol 27, No 4, 2019.

[38] Anand, Priyanka M. Mathikshara, T. Jayavignesh. An Efficient Mask Reduction Strategy to Optimize Storage and Computational Complexity in Routing Table Lookups. 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT).

[39] Tong Shen, Xian Yu, Gaogang Xie, Dafang Zhang. High-Performance IPv6 Lookup with Real-Time Updates Using Hierarchical-Balanced Search Tree. 2018 IEEE Global Communications Conference (GLOBECOM).

[40] Oğuzhan Erdem, Aydin Carus, Hoang Le. Value-Coded Trie Structure for High-Performance IPv6. The Computer Journal. Vol 58, No 2, 2015.

[42] Jhih-Yu Huang, Pi-Chung Wang. TCAM-Based IP Address Lookup Using Longest Suffix Split. IEEE/ACM Transactions on Networking. Vol 26, No 2, 2018.

[43] Wanli Zhang, Xiangyang Gong, Ye Tian, Jifan Tang. High Speed Route Lookup for Variable-Length IP Address. 2020 IEEE 28th International Conference on Network Protocols (ICNP).

[44] B. Indira, K. Valarmathi, D. Devaraj. A Trie based IP Lookup Approach for High Performance Router/Switch. 2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS).

[45] Ori Rottenstreich, János Tapolcai. Optimal Rule Caching and Lossy Compression for Longest Prefix Matching. IEEE/ACM Transactions on Networking. Vol 25, No 2, 2017.

[46] Yukito Ueno, Ryo Nakamura, Yohei Kuga, Hiroshi Esaki. Fast Longest Prefix Matching by Exploiting SIMD Instructions. IEEE Access, Vol 8, 2020.

[47] Hung-Mao Chu, Tsung-Hsien Li, Pi-Chung Wang. IP Address Lookup by Using GPU. IEEE Transactions on Emerging Topics in Computing. Vol 4, No 2, 2016.

[48] Ke Xu, Dafang Zhang, Yanbiao Li. Longest Name Prefix Match on Multi-Core Processor. 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS).

[49] Jesus Marco Vivas Ruiz, Carlos Silva Cardenas and Jose Luis Muñoz Tapia, "Implementation and Testing of IPv6 Transition Mechanisms", IEEE 9th Latin-American Conference on Communications (LATINCOM), 2017.

[50] Jesus Marco Vivas Ruiz, Carlos Silva Cardenas and Jose Luis Muñoz Tapia, "Review of Approaches for the use of the Label Flow of IPv6 Header", IEEE Latin America Transactions, Volume: 12, Issue: 8, Dec. 2014.

[51] Qiong Sun;Zhenqi_ng Li;Yan Ma. Overlapping Hash Trie: A Longest Ppefix First Search Scheme for IPv4/IPv6 Lookup. 2006 International Conference on Communication Technology.

[52] Jungwon Lee and Hyesook Lim. Multi-Stride Decision Trie for IP Address Lookup. IEIE Transactions on Smart Processing and Computing, vol. 5, no. 5, October 2016.

[53] Junghwan Kim, Myeong-Cheol Ko, Moon Sun Shin and Jinsoo Kim. A Novel Prefix Cache with Two-Level Bloom Filters in IP Address Lookup. Applied Sciences Vol. 10, Issue 20, 2020.

[54] J. Carroll Vargas, S. Salazar Fajardo, y E. J. Gómez, Propuesta de métrica para evaluar los protocolos de enrutamiento y direccionamiento IP, Avances, vol. 16, n.º 1, 2019.

[55] Sanchez, F. U., Ruiz Sanchez, M. A., & Jalpa Villanueva, C. (2019). Prefix Partitioning Based in Genetic Algorithms for Memory-Efficient IPv4/6 Lookup. IEEE Latin America Transactions, 17(11), 1823–1830.

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE DISERTACIÓN PÚBLICA

No. 00015
Matrícula: 2123803432

ESQUEMA DE BÚSQUEDA IPv4/6 EFICIENTE EN MEMORIA USANDO CODIFICACIÓN DE VECTOR DE BITS Y PARTICIONAMIENTO DE PREFIJOS

Con base en la Legislación de la Universidad Autónoma Metropolitana, en la Ciudad de México se presentaron a las 11:00 horas del día 4 del mes de agosto del año 2021 POR VÍA REMOTA ELECTRÓNICA, los suscritos miembros del jurado designado por la Comisión del Posgrado:

DR. JAVIER GOMEZ CASTELLANOS
DR. RICARDO MARCELIN JIMENEZ
DR. RUBEN VAZQUEZ MEDINA
DR. MICHAEL PASCOE CHALKE
DR. MIGUEL LOPEZ GUERRERO

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron a la presentación de la Disertación Pública cuya denominación aparece al margen, para la obtención del grado de:

DOCTOR EN CIENCIAS (CIENCIAS Y TECNOLOGIAS DE LA INFORMACION)

DE: FIDEL ULISES SANCHEZ JIMENEZ

y de acuerdo con el artículo 78 fracción IV del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:

Acto continuo, el presidente del jurado comunicó al interesado el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

FIDEL ULISES SANCHEZ JIMENEZ
ALUMNO

REVISÓ
MTRA. ROSALÍA SERRANO DE LA PAZ
DIRECTORA DE SISTEMAS ESCOLARES

DIRECTOR DE LA DIVISIÓN DE CBI	PRESIDENTE	VOCAL
DR. JESUS ALBERTO OCHOA TAPIA	DR. JAVIER GOMEZ CASTELLANOS	DR. RICARDO MARCELIN JIMENEZ
VOCAL	VOCAL	SECRETARIO
DR. RUBEN VAZQUEZ MEDINA	DR. MICHAEL PASCOE CHALKE	DR. MIGUEL LOPEZ GUERRERO

El presente documento cuenta con la firma –autógrafa, escaneada o digital, según corresponda- del funcionario universitario competente, que certifica que las firmas que aparecen en esta acta – Temporal, digital o dictamen- son auténticas y las mismas que usan los c.c. profesores mencionados en ella