Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA- IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍAS

MECANISMOS DE CODIFICACIÓN DE VECTOR DE BITS PARA BÚSQUEDAS EN TABLAS DE RUTEO IP

Tesis que presenta
Fidel Ulises Sánchez Jiménez

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

Asesores:

Dr. Miguel Ángel Ruiz Sánchez
Dr. César Jalpa Villanueva

Jurado Calificador:

Presidente: **Dr. Rubén Vázquez Medina**	**ESIME-Culhuacan**
Secretario: **Dr. Miguel López Guerrero**	**UAM-I**
Vocal: **Dr. Miguel Ángel Ruiz Sánchez**	**UAM-I**

México D.F. a 26 de julio de 2012

Resumen

Redes de computadoras como Internet han tenido una creciente popularidad y un número de usuarios que crece a cada día. En el origen de Internet (década de los sesenta) se enviaron 2 caracteres; hoy, 2 billones de personas navegan en la red, en promedio 1 millón de clientes por hora son atendidos y se proyecta que la información producida en el 2015 por los teléfonos móviles será de 6.3 Exabytes[1]. Toda esta información tendrá que ser descompuesta en paquetes definidos por el protocolo IP y enviada a su destino utilizando algoritmos de encaminamiento a través de la red. Determinar lo más rápido posible hacia dónde debe ser reexpedida la información en un dispositivo de encaminamiento o enrutador contribuirá de forma importante a prevenir la congestión de la red y por ende ofrecer un servicio de más alta calidad.

Para determinar a dónde se debe de enviar un paquete, un enrutador cuenta con una tabla de ruteo que básicamente es una estructura de datos que posee información de las redes que son alcanzables por él; esta estructura está constituida por prefijos de red asociados a un enlace de salida (dirección física) entre otras cosas. El enlace de salida asociado al prefijo más largo de una dirección IP destino será la mejor opción para reexpedir el paquete.

En el mundo de las redes de comunicaciones existen algoritmos para reexpedir paquetes en el enrutamiento jerárquico. Inspiramos inicialmente esta investigación en el algoritmo de búsqueda del prefijo más largo de la universidad de LULEA. En el cual, aparecen varios parámetros que intervienen en la codificación de la tabla. Los autores usan valores bien determinados para estos parámetros pero no es claro como escoger el valor más apropiado. En este trabajo de investigación se estudiará como estos parámetros influyen realmente en el grado de compresión que se obtiene y si es posible determinar valores que minimicen la complejidad del algoritmo en memoria e instrucciones.

[1] Proyección y estadísticas de ORACLE (www.oracle.com).

En este trabajo también se retoman los algoritmos representativos de búsqueda del prefijo más largo basados en la compresión del "Trie binario", esto para establecer un algoritmo que soporte la complejidad del tráfico y los cambios de topología de Internet y que minimice el número de instrucciones y el tamaño de las estructuras de datos que se utilicen; para ello se propone una técnica donde el número de instrucciones de búsqueda y el grado de compresión del "Trie" tienen una relación de proporción inversa; ofreciendo un esquema flexible donde los valores de los parámetros de operación pueden ser optimizados de acuerdo a la capacidad de procesamiento y memoria de cada enrutador.

Presentamos la relevancia de nuestra investigación en términos de las demandas que los algoritmos de ruteo deban poseer; así mismo, hacemos la propuesta de nuestro algoritmo el cual es evaluado y comparado matemáticamente con algunos trabajos relacionados. También hacemos una breve evaluación de desempeño mediante simulaciones acordes a las características actuales de los troncales de Internet.

Agradecimientos

Agradezco y dedico esta tesis a mi familia pues son el motor de mi vida; a mi madre Avelina Jiménez García y a mi hermana Alejandra Sánchez Jiménez. Seguiré una meta, cumpliré mis sueños. Sin embargo, no olvidaré quien soy y de dónde vengo.

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

A los sinodales Dr. Miguel López Guerrero, Dr. Rubén Vázquez Medina y Dr. Miguel Ángel Ruiz S.

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

A la Universidad Autónoma Metropolitana por ser mí casa de estudios durante esta bonita experiencia.

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

Contenido

Lista de Figuras

Lista de Tablas

Capítulo
1 Introducción

En este capítulo se presenta una visión general de las redes de computadoras y su modelado en capas con el fin de ubicar el problema de interés en lo que se conocerá como capa de red. Se identificará el problema general del encaminamiento de la información que circula a través de la red en función de la labor distribuida que efectúan los dispositivos de encaminamiento o enrutadores[2] reconociendo que cada enrutador tiene una perspéctiva distinta de la red pues en el interior de ella los enrutadores están situados en distintos puntos geográficos y por lo tanto tendrán una visión de la red bajo una perspectiva local.

Se presenta también a manera de ejemplos la observación del problema de interés para que en capítulos posteriores podamos hacer una delimitación del tema y seguir los pasos del método científico.

[2] En todo este trabajo se utilizará dispositivo de encaminamiento o enrutador de forma indistinta.

1.1 Redes de computadoras

Una red de computadoras es un conjunto de equipos informáticos conectados entre sí por un medio de transmisión (P. ej. aire, cable, fibra óptica) con la finalidad de compartir sus recursos (P. ej. Procesador y unidades de almacenamiento) e información y así poder ofrecer distintos servicios. Su definición es "*Conjunto de ordenadores o de equipos informáticos conectados entre sí que pueden intercambiar información*"[3].

El nacimiento de la redes parte de la necesidad de tener servicios que una computadora no es capaz de realizar sola. En 1940 se transmitieron caracteres por la red telefónica desde la Universidad de Darmouth, en Nuevo Hampshire a Nueva York concibiendo por primera vez el intercambio de información entre dos computadoras separadas. En la actualidad no sólo se comparte información de una computadora a otra sino que también se hace programación distribuida en la cual las computadoras en distintos puntos de la red trabajan de forma conjunta para resolver un problema. Los componentes básicos para poder conectar computadoras y establecer una red son:

Hardware para lograr conectar de forma física a las computadoras. Entre los elementos de hardware están los medios de transmisión (P. ej. cables de red o medios guiados para redes alámbricas o de luz infrarroja y radiofrecuencias para redes inalámbricas) y el hardware encargado de procesar las señales de los medios de transmisión para que sean entendidas por la computadora, esta tarea es realizada por las tarjetas de red. Cabe señalar que a cada tarjeta de red le es asignado un identificador único por su fabricante, conocido como dirección MAC (Medium Access Control), que consta de 48 bits (6 bytes). Dicho identificador permite direccionar el tráfico de datos de la red del emisor al receptor adecuado.

Software cuya función es ejecutar protocolos que permiten la comunicación entre distintas computadoras con distintas capacidades, arquitecturas y software interno (sistema operativo). Al igual que un equipo no puede trabajar sin un sistema operativo, una red de computadoras no puede funcionar sin protocolos de red.

[3] Definición de red de la Real Academia Española.

1.2 Modelo de capas

Existen diversos protocolos, estándares y modelos que determinan el funcionamiento general de las redes. Destacan el modelo OSI y el TCP/IP. Cada modelo estructura el funcionamiento de una red de manera distinta. El modelo OSI cuenta con siete capas definidas y con funciones diferenciadas. El modelo TCP/IP cuenta con cuatro capas diferenciadas pero que combinan las funciones existentes de las siete capas del modelo OSI.

El modelo a capas permite distinguir distintos problemas que se presentan en la red y atacarlos por separado, además ofrece la opción de efectuar cambios en alguna capa sin que las demás capas perciban dicho cambio. Por ejemplo, en Internet no importa que la transmisión de la información sea por vía alámbrica o inalámbrica (capa física), la aplicación del usuario no se verá afectada ni cambiará en nada.

En la figura 1-1 se muestran las capas de los modelos OSI y TCP/IP. Como se puede notar el modelo de TCP/IP reúne en la capa de Enlace las capas Física y Enlace del modelo OSI. La capa de Aplicación del modelo TCP/IP, para muchas aplicaciones, ofrece los servicios de Presentación y Sesión que el modelo OSI establece.

Modelo OSI

#	*Capa*
7	Aplicación
6	Presentación
5	Sesión
4	Transporte
3	Red
2	Enlace
1	Fisica

Modelo TCP/IP

#	*Capa*
4	Aplicación
3	Transporte
2	Red
1	Enlace / Física

Figura 1-1 Modelos OSI y TCP/IP

Cada capa debe realizar las funciones que le conciernen, en consecuencia cada capa debe de resolver problemas diferentes; en este trabajo es de interés tener bien definidas las funciones principales de la capa de red pues el problema de la reexpedición de un paquete que efectúa un enrutador está contemplado en esta capa.

La capa de red, tanto para el modelo OSI como para el modelo TCP/IP, cumple las mismas funciones. Esta capa proporciona conectividad y hace la selección de ruta entre dos sistemas terminales o *hosts* que pueden estar ubicados en redes geográficamente distintas. Su misión es conseguir que los datos lleguen desde su origen hasta su destino aunque no tengan conexión directa. Ofrece servicios a la capa superior (capa de transporte) y se apoya en la capa de enlace, es decir, utiliza sus funciones.

La capa de red provee principalmente los servicios de envío y enrutamiento de los datos (paquetes de datos) de un nodo a otro en la red, ésta es la capa más inferior en cuanto a manejo de transmisiones extremo a extremo.

Las principales características de la capa de red son:

- El propósito de esta capa es el de formar una interfaz entre los usuarios de una máquina y la red.
- Los servicios que se proveen deberán ser independientes de la tecnología de soporte.
- El diseño de la capa debe permitir conectar redes con diferentes tecnologías.
- Lo que a esta capa le interesa es un camino de comunicación de extremo a extremo de la red.

1.3 Protocolo IP y direcciones IP

Una dirección IP es una etiqueta numérica que identifica, de manera lógica y jerárquica, a un dispositivo (habitualmente una computadora) dentro de una red que utilice el protocolo IP (Internet Protocol) [1], que corresponde a la capa de red del modelo TCP/IP. Dicho número no se debe confundir con la dirección MAC que es un identificador de 48 bits para identificar de forma única a la tarjeta de red y sus valores no tienen nada que ver uno con el otro. La dirección IP puede cambiar muy a menudo por cambios en la red o porque el dispositivo encargado dentro de la red de asignar las direcciones IP, decida asignar otra IP (por ejemplo, con el protocolo DHCP[4]); a esta forma de asignación de dirección IP se denomina dirección IP dinámica (normalmente abreviado como IP dinámica).

Los sitios de Internet que por su naturaleza necesitan estar permanentemente conectados, generalmente tienen una dirección IP fija (comúnmente llamada IP fija o IP estática), esta no cambia con el tiempo. Por ejemplo, los servidores de páginas web necesariamente deben contar con una dirección IP fija o estática, ya que de esta forma se permite su localización en la red.

[4] Las especificaciones del protocolo DHCP (Protocolo de Configuración de *Host* Dinámica) se pueden encontrar en la RFC 1541.

A través de Internet los ordenadores se conectan entre sí mediante sus respectivas direcciones IP. Sin embargo, a los seres humanos nos es más cómodo utilizar otra notación más fácil de recordar, como los *nombres de dominio*[5]. La traducción de un nombre a una dirección IP se resuelve mediante los servidores de nombres de dominio DNS (servidor de nombres), que a su vez, facilita el trabajo en caso de cambio de dirección IP, ya que basta con actualizar la información en el servidor DNS y el resto de las personas no se enterarán ya que seguirán accediendo por el nombre de dominio.

En la actualidad existen dos tipos de direccionamiento IP; uno que especifica direcciones de longitud de 32 bits conocido como IPv4 y que permite tener hasta 2^{32} elementos en la red; el otro tipo de direccionamiento surge por la necesidad de tener una red de mayor tamaño, es por ello que las direcciones son de 128 bits y es conocido como IPv6.

Las direcciones se distribuyen de forma jerárquica de tal forma 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 1-2 se muestra un ejemplo de una red jerarquizada en base a los prefijos de red, nótese que el ejemplo es de una red con direcciones IPv4 y los usuarios de la red están en la parte inferior de la jerarquía; los enrutadores forman subredes hacia abajo y hacia arriba son parte de una red de mayor tamaño.

[5] Un sitio web que tenga su propio nombre de dominio es mucho más fácil de memorizar. Es mucho más sencillo para un usuario regresar a un sitio web cuyo nombre es fácil de recordar que a uno con un nombre especificado por un valor numerico.

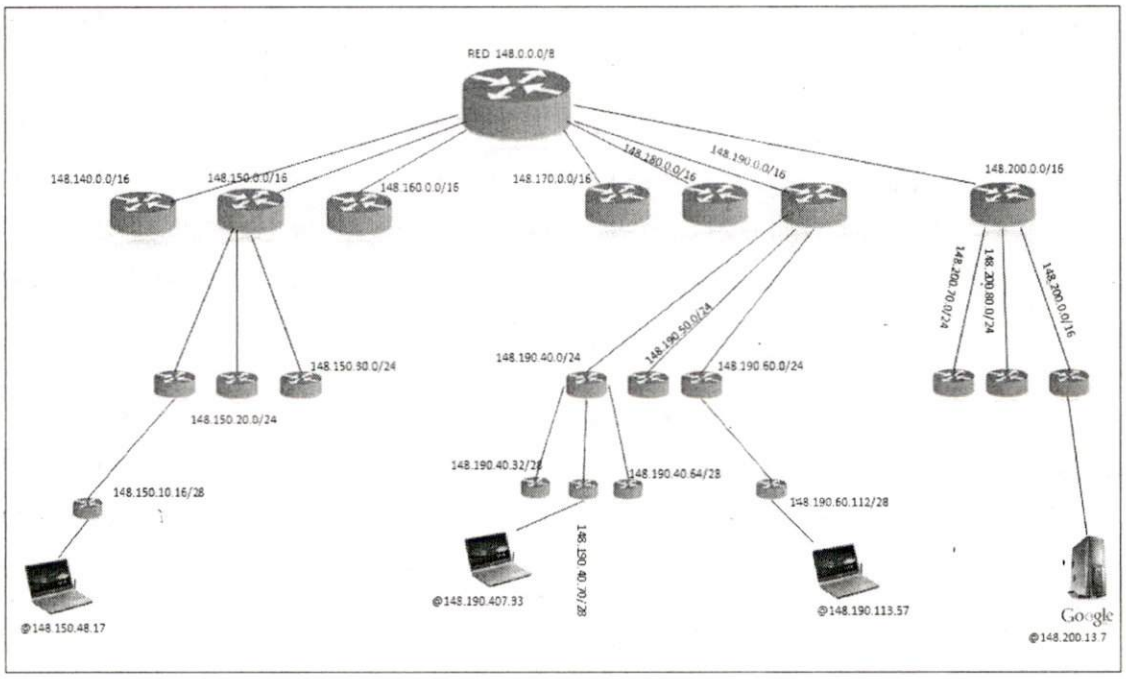


Figura 1-2 Red con jerarquía en base a los prefijos de red

El protocolo IP es parte de la capa de red del conjunto de protocolos TCP/IP. Es uno de los protocolos de Internet más importantes ya que permite el transporte de datagramas de IP (paquetes de datos), aunque sin garantizar su entrega.

El datagrama IP es la unidad de transferencia en las redes IP. Básicamente consiste en una cabecera IP y un campo de datos para protocolos superiores. El datagrama IP está encapsulado en la trama de nivel de enlace, que suele tener una longitud máxima (MTU, *Maximum Transfer Unit*), dependiendo del hardware de red usado. Para Ethernet, ésta es típicamente de 1500 bytes. En vez de limitar el datagrama a un tamaño máximo, IP puede tratar la fragmentación y el reensamblado de sus datagramas. Los fragmentos de datagramas tienen una cabecera, copiada básicamente del datagrama original, y de los datos que la siguen. Los fragmentos se tratan como datagramas normales mientras son transportados a su destino. Sin embargo, si uno de los fragmentos se pierde, todo el datagrama se considerará perdido, y los restantes fragmentos también se considerarán perdidos.

La cabecera del datagrama IP está formada por los campos que se muestran en la figura 1-3 correspondiente a IPv4 (direcciones de 32 bits). Nótese que el datagrama tiene un campo de dirección IP destino el cual es utilizado en la red para encaminar al datagrama hasta su destino; en este trabajo nos enfocaremos sólo en este campo ya que con él se toman decisiones de encaminamiento.

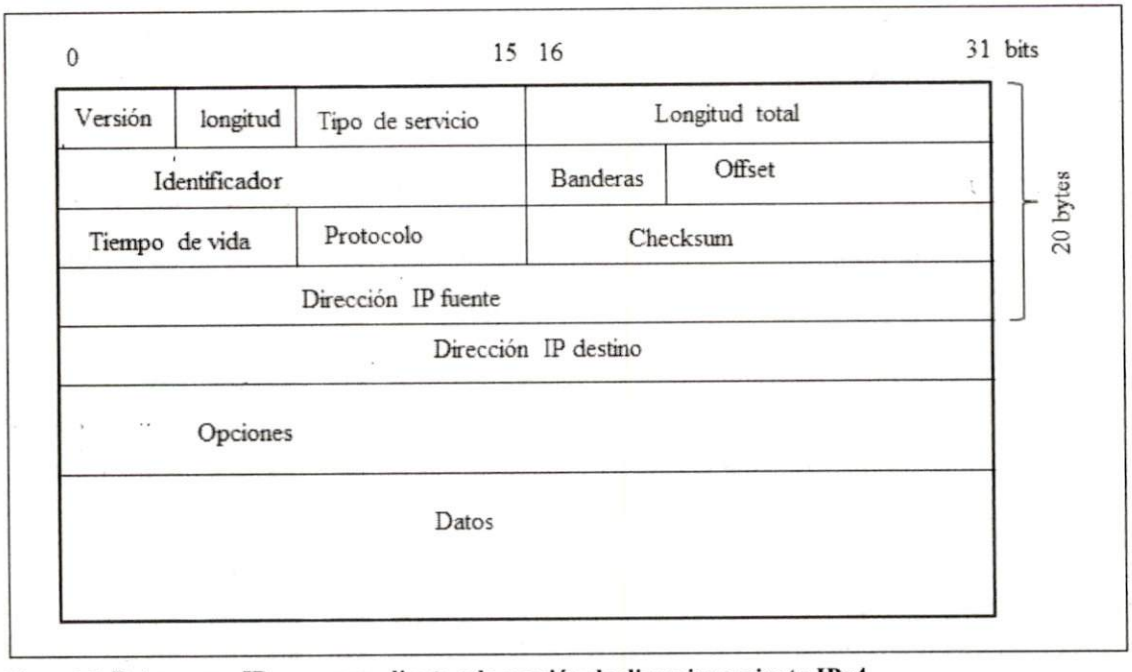


Figura 1-3 Datagrama IP correspondiente a la versión de direccionamiento IPv4

1.4 Enrutadores, ruteo y reenvió

Una red como Internet está compuesta por computadoras que pueden ser de dos tipos; en un primer tipo están definidas las computadoras de propósito general o *host* terminales las cuales están al extremo de la red y son las que la utilizan para enviar o recibir información a través de ella. El segundo tipo de computadoras está destinado a cumplir una labor específica. Estas computadoras se encuentran en el interior de la red y se caracterizan por estar dedicadas a encaminar los paquetes de información producidos por las computadoras del primer tipo; a estas computadoras se le conoce como dispositivos de encaminamiento o enrutadores.

Un **enrutador** en un dispositivo de hardware que sirve para la interconexión de redes de computadoras que opera en la capa de red y que permite asegurar el encaminamiento de paquetes entre redes o determinar 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 *host* terminales.

A continuación se hace la distinción del proceso de ruteo y el reenvió para poder distinguir con facilidad la tarea del encaminamiento de la información como proceso distribuido de los enrutadores en la red y el trabajo que hace cada uno de ellos de forma individual.

Ruteo. Es el proceso distribuido encargado de recolectar información de las redes que un dispositivo de encaminamiento puede alcanzar; los prefijos de red son los bits más significativos de una dirección IP que identifican a una red. La información que recolectan los enrutadores es precisamente la contenida en los prefijos de red que son la representación de las redes que están a su alcance; el prefijo tiene asociada una dirección física (MAC) que es el enlace de salida de los paquetes que circulan a través del enrutador.

Reenvío. El reenvío comienza cuando llega un paquete al enrutador; es el proceso en el cual el enrutador busca el mejor enlace de salida para encaminar el paquete. De la cabecera del paquete se obtiene la dirección destino y mediante una taba de rutas[6] de ruteo se realiza una búsqueda para encontrar el prefijo más largo para así determinar el enlace de salida correcto para reexpedir la información.

[6] En adelante a la tabla de rutas se le manejara de forma indistinta como tabla de ruteo, tabla de enrutamiento y tabla de prefijos.

1.5 Uso de tablas de ruteo

Una tabla de ruteo es una estructura de datos que almacena las rutas a los diferentes nodos en una red de computadoras; los nodos pueden ser cualquier tipo de dispositivo electrónico conectado a la red. La información de ruteo contenida en ésta representada por los prefijos de red ligados a una dirección física de salida. La tabla de ruteo generalmente se almacena en un enrutador en forma de un archivo de base de datos; cuando los datos deben ser enviados desde un nodo a otro de la red, se hace referencia a la tabla de ruteo con el fin de encontrar la mejor ruta para la transferencia de información.

La construcción de la tabla de ruteo se realiza mediante un proceso distribuido en el cual participan los enrutadores que conforman la red para determinar el estado de la topología y así determinar las redes alcanzables que cada dispositivo de encaminamiento percibe. 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 búsqueda en algún servidor. En la figura 1-4 se muestra una computadora (*host* A) y el servidor de búsqueda (Google). Ambas entidades se encuentran en el extremo de la red; al interior de la nube se encuentran redes de menor tamaño definidas por lo 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 la información por algunos de sus tres enlaces, así que hace uso de la dirección destino (168.156.34.75) y de la tabla de ruteo para determinar el prefijo más largo. En el ejemplo se proponen direcciones IPv4 en notación decimal y se asume una jerarquía de la red en base a los prefijos para hacer posible el encaminamiento de la información.

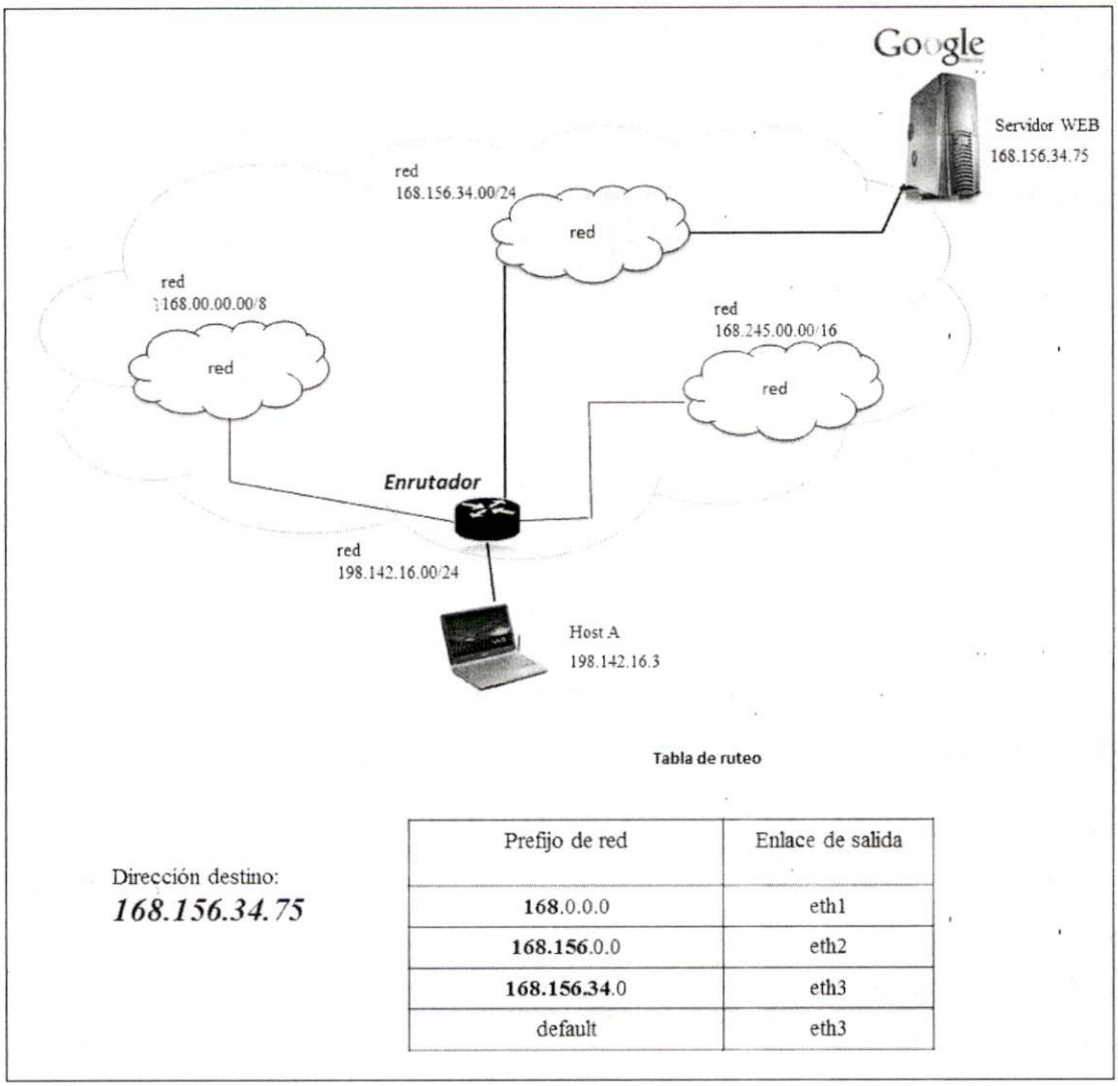


Figura 1-4 Reenvío de la información en base a los prefijos de red

El resto de esta tesis está organizada de la siguiente manera: En el capítulo dos se enuncia el problema de interés y la delimitación del mismo según el método científico. En el capítulo tres se presentan algunos de los algoritmos más representativos de búsqueda del prefijo más largo en tablas de ruteo IP con el propósito de dar a conocer las ideas principales de su funcionamiento, pero sobre todo, ofrecer al lector una idea del costo por utilizar cualquiera de estos algoritmos. En el capítulo cuatro se encuentra principalmente la contribución de esta tesis; se desarrolla un algoritmo de búsqueda del prefijo más largo mediante distintos mecanismos de codificación de vectores de bits, se presentan también un análisis matemático de su costo en memoria e instrucciones. En el capítulo cinco se presenta una breve comparación de nuestro algoritmo con un algoritmo en particular (algoritmo de LULEA). En el capítulo seis y siete se presentan las conclusiones y trabajo futuro para continuar con este tema de investigación.

Capitulo
2 Delimitación del problema

Después de presentar mediante ejemplos el problema de la reexpedición de paquetes en un enrutador. En este capítulo retomaremos y presentamos la definición formal del problema que nos interesa abordar junto con el razonamiento que nos ha llevado a distinguir la necesidad de una investigación más profunda del problema de la reexpedición de paquetes. Presentamos también la relevancia de nuestra investigación en términos de las demandas que requieren las redes de acuerdo a su crecimiento acelerado.

Finalmente presentamos los objetivos de trabajo basados en el análisis de los algoritmos de búsqueda del prefijo más largo basados en la compresión, ofreciendo por supuesto, la explicación del interés particular en este tipo de algoritmos.

2.1 Planteamiento del problema

El hecho de que dos o más computadoras conectadas a una red en distintos lugares establezcan una comunicación bajo cualquiera de sus aplicaciones o propósitos, trae consigo una serie de requerimientos que la red debe satisfacer para que esta comunicación se lleve se forma satisfactoria. Uno de los requerimientos de interés particular para este trabajo es el encaminamiento de la información a través de la reexpedición de paquetes. Un enrutador es quien recibe un paquete IP y tiene como objetivo reexpedirlo por uno de sus enlaces que conecta a otro enrutador, de tal forma que el paquete esté más cerca de su destino o en él. En consecuencia un enrutador debe de poseer información de las redes que están a su alcance para determinar bajo algún algoritmo cual es el enlace más adecuado para reexpedir los paquetes IP. La información que este dispositivo contiene acerca de la red se guarda en una tabla ruteo que contiene información de las redes a las que tiene acceso mediante los prefijos de red.

Inspirados en el algoritmo de búsqueda del prefijo más largo para la reexpedición de paquetes propuesto por la universidad de LULEA. En este trabajo retomamos la filosofía y parámetros de diseño que éste algoritmo propone, haciendo un análisis matemático para determinar y proponer un algoritmo con parámetros adecuados de operación para ser optimizado en su complejidad en memoria e instrucciones.

El problema de interés es el diseño y análisis de algoritmos de búsqueda del prefijo más largo basados en la compresión de las estructuras de datos que representan a la tabla de ruteo. Estos algoritmos tiene como finalidad encontrar el prefijo más largo dada una dirección IP destino lo más rápido posible; en otras palabras el paquete debe permanecer el menor tiempo posible en el enrutador antes de ser reexpedida para que siga el camino a su destino. Hay que hacer notar que el proceso de reexpedición de paquetes no contempla la recopilación de la información de la tabla de ruteo; dicha recopilación la realiza el enrutador pero por medio del proceso llamado **proceso de ruteo**.

2.2 Justificación

En la literatura se puede encontrar que los algoritmos de búsqueda del prefijo más largo intentan representar la información de ruteo de forma eficiente mediante estructuras de datos auxiliares. Las principales estructuras de datos utilizadas son árboles y tablas *hash*.

Una desventaja de estos algoritmos es que su complejidad en memoria para construir las estructuras de datos y su complejidad en instrucciones de acceso a memoria para obtener el prefijo más largo depende del número y longitud de los prefijos en la tabla de ruteo. Internet tiene un crecimiento muy rápido y con la llegada de IPv6 la longitud de los prefijos de red es mayor por lo que el desempeño de estos algoritmos decrece rápidamente.

El tráfico en las redes de comunicaciones es un fenómeno muy complejo que influye de manera importante en los algoritmos de reexpedición de paquetes (algoritmos de búsqueda del prefijo más largo), por lo tanto es necesario el uso de algoritmos que utilicen estructuras de datos que ocupen una cantidad de memoria independiente a la longitud y número de prefijos en la tabla de ruteo (solo depende de la longitud máxima de los prefijos) y que el acceso a la información de ruteo sea equitativa a cualquier dirección IP destino; esto quiere decir que el costo al acceso de la información de ruteo para cualquier dirección IP debe ser el mismo. Además se deben de tener algoritmos que minimicen dicha cantidad de memoria requerida para las estructuras de datos y que minimicen el número de accesos a memoria para la obtención del prefijo más largo. Si se minimizan el número de instrucciones de acceso a memoria por ende se reducirá el tiempo de búsqueda, por otro lado si las estructuras de datos utilizadas son lo suficientemente pequeñas para que sean contenidas en memoria principal o *cache* de una computadora, el acceso al contenido de las estructuras estará determinado por la frecuencia de operación del procesador; sin embargo, si las estructuras de datos ocupan mucho espacio y llegan a ser contenidas en el memorias como discos duros el acceso a ellas será más lento en comparación de la memoria *cache*.

2.3 Hipótesis

Existen mecanismos de compresión y codificación de las estructuras de datos que representan a los prefijos de una tabla de ruteo; tal que, minimizan la complejidad de un algoritmo de búsqueda del prefijo más largo en memoria o bien en instrucciones de acceso a memoria. Dichas minimizaciones dependen del grado de compresión o codificación en las estructuras de datos. Los mecanismos de compresión o codificación permiten construir estructuras de datos que con una complejidad en memoria $O(c)$, tal que c es constante para cualquier tamaño de la tabla de ruteo; además, el algoritmo para la obtención del prefijo más largo, minimiza el número de accesos a memoria.

2.4 Objetivos

Proponemos los siguientes objetivos que propician una solución de acuerdo a los requerimientos de una red que evoluciona y crece a cada momento.

2.4.1 Objetivo general

El objetivo general de este proyecto de investigación es consolidar un algoritmo de búsqueda del prefijo más largo que sea flexible en el valor de sus parámetros de operación para cambiar el grado de compresión y codificación de las estructuras de datos que utilice, en otras palabras, dado un valor especifico de sus parámetros; sea minimizado en su complejidad en memoria o instrucciones de búsqueda.

El algoritmo debe de poseer ciertas características para un conjunto de valores en sus parámetros de operación:

1. El tamaño de las estructuras de datos que utilice deben ser independientes al tamaño y la longitud de la tabla de ruteo (solo depende de la longitud máxima de los prefijos).

2. El tamaño de las estructuras de datos que utilice tienen que ser lo suficientemente pequeñas para estar contenidas en memoria principal.
3. El acceso a la información de ruteo en las estructuras de datos debe tener el mismo costo computacional para cualquier dirección IP destino.
4. El número de instrucciones de acceso a memoria para la búsqueda del prefijo más largo debe de ser constante y lo más mínimo posible.

2.4.2 Objetivos particulares

1. Inspirados en el algoritmo de la universidad de LULEA. Generalizar dicho algoritmo para construir un nuevo algoritmo que en base a los valores de sus parámetros, pueda controlarse la codificación y compresión para ajustar su complejidad en memoria e instrucciones.
2. Hacer un análisis de desempeño matemático a la solución propuesta para determinar el valor de los parámetros que minimicen el costo en memoria requerida para las estructuras de datos.
3. Hacer un análisis de desempeño matemático a la solución propuesta para determinar el valor de los parámetros que minimicen el costo de las instrucciones de búsqueda.
4. Comparar el modelo matemático de la propuesta con un algoritmo que sea de esta clase (LULEA). Los parámetros a comparar son los siguientes:
    - El tamaño requerido en memoria en bits para conformar las estructuras de compresión o codificación.
    - El número de instrucciones de accesos a memoria requeridos por el algoritmo para obtener el prefijo más largo de una dirección IP.
5. Aunque se especifica que el tráfico y la tabla de ruteo no es un factor que deba de influir en el desempeño de un algoritmo. Hacer simulaciones para un análisis de desempeño del algoritmo propuesto utilizando tablas de ruteo reales obtenidas de algún troncal de Internet utilizando direcciones IPv4 generadas de forma aleatoria con una distribución uniforme.

En este capítulo presentamos formalmente el problema dado que se hizo una observación de la problemática de interés en el capítulo 1. Justificamos la importancia de la investigación y planteamos una hipótesis la cual se intentará demostrar con el objetivo general de esta investigación.

Capítulo
3 Algoritmos representativos de búsqueda en tablas de ruteo

En este capítulo se presentan los principios básicos de operación de los algoritmos más representativos de búsqueda del prefijo más largo. El objetivo es presentar el modo en que operan dichos algoritmos y destacar sus cualidades.

Algunos algoritmos de búsqueda clásicos nos permitirán explicar de una manera más sencilla la forma de operación de nuestra propuesta que se presenta en el siguiente capítulo. Por ejemplo, el uso de estructuras auxiliares para representar la tabla de ruteo mediante el uso de árboles binarios. Se presentan con más detalle los algoritmos de búsqueda basados en la compresión de estructuras auxiliares que representan la tabla de ruteo. La propuesta bien puede ser un caso particular de los algoritmos basados en la compresión.

3.1 "Trie" binario

Una de las formas más sencillas de encontrar el prefijo más largo es utilizando el algoritmo de "Trie" binario [2]. En este algoritmo se representan a los prefijos que conforman a la tabla de ruteo como ramas[7] de un árbol binario.

El "Trie" o árbol binario se construye recorriendo bit a bit cada prefijo. Se comienza por el nodo raíz y se recorre cada prefijo bit a bit, si el bit tiene un valor de *cero* se agrega un nodo correspondiente al hijo izquierdo y si el bit tiene el valor de *uno* se agrega un nodo a la derecha correspondiente al hijo derecho. Entonces, se tiene que el árbol binario guarda la información de ruteo en sus nodos. En la figura 2-1 se puede observar una pequeña tabla de ruteo que contiene a los prefijos nombrados a, b, c, d, e, f y a la derecha la representación mediante un árbol binario. Nótese que los prefijos pueden ser hojas o nodos internos.

Tabla de ruteo

	Prefijo	*Enlace*
a	000	Eth1
b	001	Eth0
c	01	Eth0
d	011	Eth3
e	1	Eth3
f	111	Eth2
Raiz	----	Eth2

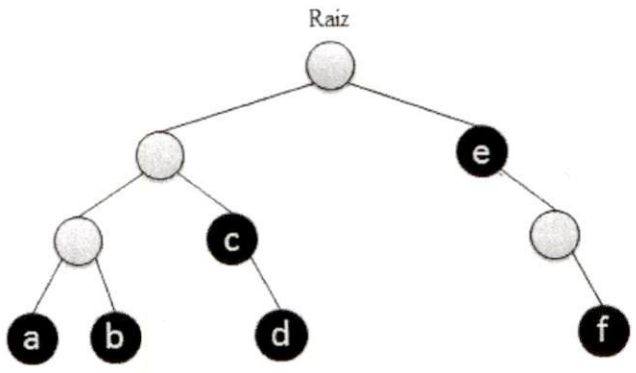


Figura 3-1 "Trie" binario

[7] Una rama en un árbol binario define una trayectoria desde la raíz hasta un nodo interno u hoja.

La búsqueda se efectúa mediante la dirección destino del paquete en cuestión. El árbol binario tiene una profundidad máxima igual a la longitud de la dirección IP. Además la dirección destino se debe recorrer bit a bit (comenzando con el más significativo) definiendo así una trayectoria en el árbol binario. El criterio de paro es el siguiente:

- Cuando se llegue a la máxima profundidad del árbol.
- Cuando se llegue a un bit de la dirección IP el cual ya no tenga una representación en el árbol binario (no existe un nodo para seguir definiendo la trayectoria).

En la figura 2-2 se puede ver un ejemplo en el cual la profundidad máxima del árbol es 3 y cómo una dirección IP de tres bits (010) define una trayectoria en el árbol. Nótese que al no encontrar un nodo para seguir con la trayectoria ,ya se ha encontrado el prefijo más largo correspondiente a la dirección IP destino (en este caso el prefijo más largo es el 01*).

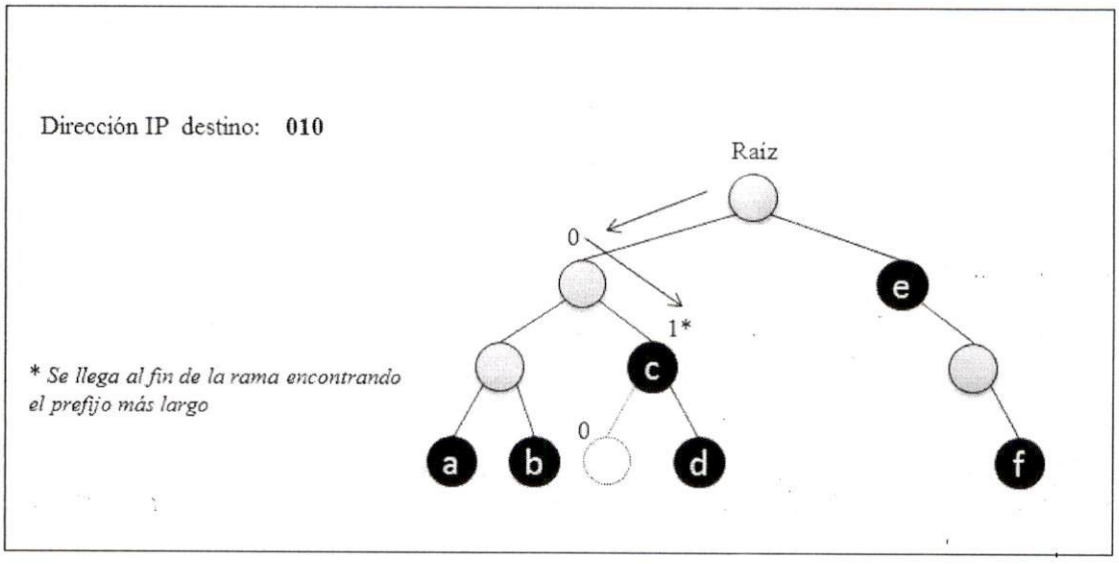


Figura 3-2 Búsqueda en el trie binario

El costo en memoria por realizar búsquedas es el costo total en nodos por tener la estructura de árbol binario. Por lo que si se hacen búsquedas en IPv4 se necesita tener en memoria un árbol de profundidad 32 con hasta 429,503,110 nodos.

3.2 Expansión controlada de prefijos

La tabla de ruteo puede contener prefijos de cualquier longitud. El algoritmo de "expansión controlada de prefijos" [3] expande a los prefijos hasta una longitud que permita hacer comparaciones en secciones de la dirección IP; de esta forma el algoritmo se efectúa en iteraciones, y en cada iteración se comparan secciones diferentes de bits. La expansión consiste en poner los bits que sean necesarios para completar la longitud de la sección de comparación. Los valores de los bits agregados deben de ser todas las combinaciones posibles que éstos tengan; por lo tanto, de un prefijo que será expandido se crearán más prefijos que cumplan la longitud de la sección establecida. Por ejemplo, en la figura 2-3 (izquierda) se muestra una tabla con prefijos de diferentes longitudes. Se establecen (para este ejemplo) secciones de comparación de 2, 4 y 7 bits, por lo tanto a los prefijos que tengan una longitud menor a dos se les agrega un bit; para prefijos de longitud mayor a dos pero menor que cuatro se les agrega también un bit, a los prefijos de longitud mayor a cuatro pero menor que siete bits se les agrega uno o dos bits según sea el caso. Nótese que de un prefijo de longitud menor a una sección establecida se derivan a más prefijos con las longitudes establecidas y todas las combinaciones de bits que se le agregan.

La búsqueda del prefijo más largo consiste en hacer búsquedas por sección (en cada sección se hace una iteración del algoritmo), de esta forma se busca primero en secciones con menor longitud, si el prefijo es más largo a la longitud de la sección entonces el algoritmo hace una iteración en la sección siguiente de mayor longitud y así sucesivamente hasta que termine.

Nombre	Prefijo original	Longitud
P1	0	1
P2	1000000	7
P3	010	3
P4	11	2
P5	10011	5

Nombre	Prefijo expandido	Longitud
P1	00	2
P1	01	
P4	11	
P3	0100	4
P3	0101	
P5	1001100	7
P5	1001101	
P5	1001110	
P5	1001111	
P2	1000000	

Figura 3-3 Ejemplo de expansión controlada del prefijo

3.3 Búsqueda multidirección multicolumna

Este algoritmo consiste en hacer un modelo de búsqueda exacta (que encuentra el único prefijo más largo) para uno que no lo es. Esto se logra añadiendo información de los intervalos de prefijos que un dispositivo de encaminamiento puede alcanzar [4]. Elaborando un pequeño ejemplo con direcciones de 8 bits, se tiene una tabla de ruteo que posee los prefijos 1, 101, 10101, 1010101; cada prefijo representa un intervalo de direcciones IP de longitud 8, en la figura 2-4.a se muestra el intervalo de direcciones que cubre cada prefijo. De los límites inferior y superior de direcciones IP que se encontraron se genera una tabla ordenándolos de menor a mayor (figura 2-4.b).

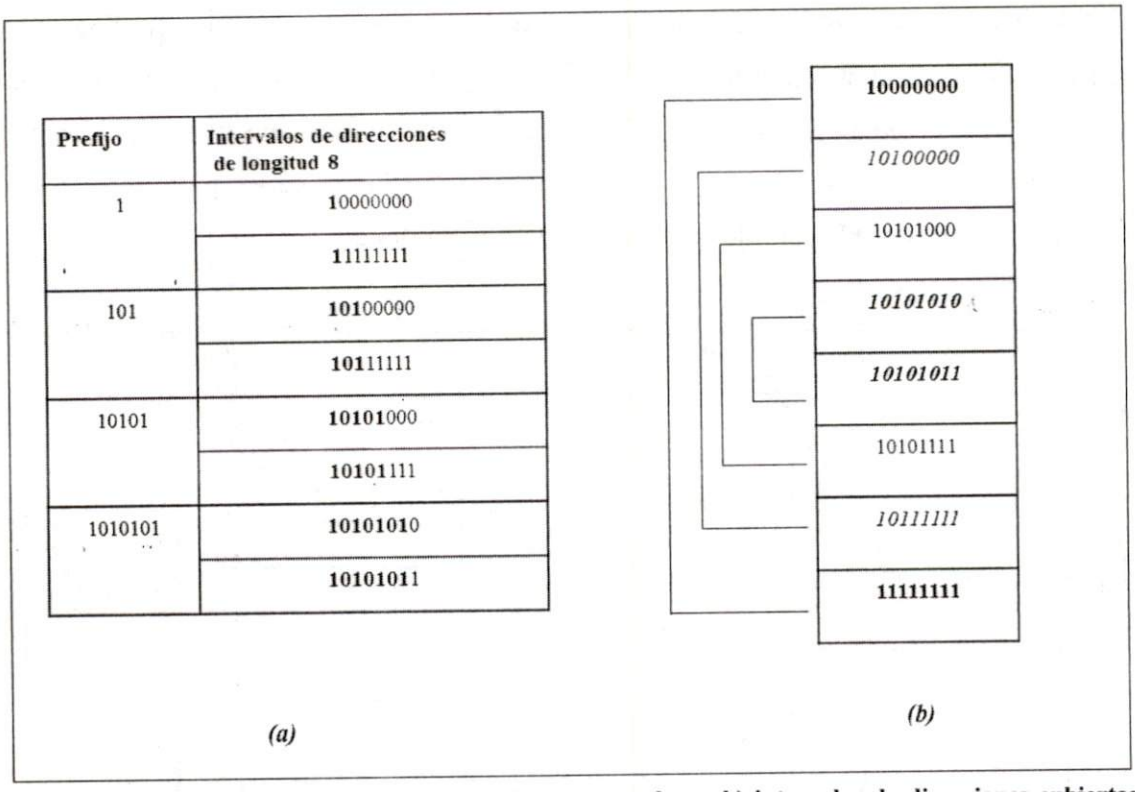


Figura 3-4 a) Prefijos y los intervalos de direcciones que cubren. b) intervalos de direcciones cubiertas por los prefijos a modo de tabla

Una vez conseguida la tabla de intervalos de direcciones cubiertas se pueden efectuar búsquedas en ésta pues la dirección IP destino estará en algún intervalo de direcciones de la tabla y este intervalo corresponde a un prefijo en la tabla de ruteo.

3.4 Búsqueda en tablas *hash*

Entre las estructuras de datos que representan a la información de ruteo se encuentra la búsqueda por tablas *hash* [5]. La dirección IP destino se considera una llave para buscar en la tabla de prefijos. La tabla de prefijos es ordenada por la longitud de los prefijos para que la búsqueda sea guiada por este parámetro. La implementación puede darse de diferentes formas: búsqueda lineal, búsqueda binaria y búsqueda asimétrica.

La **búsqueda lineal** consiste en un arreglo que contiene las longitudes de los prefijos y por supuesto un apuntador a la tabla que los contiene, de esta forma se puede hacer una búsqueda de acuerdo a las longitudes de los prefijos que se deseen encontrar. Por ejemplo en la figura 2-5 se muestra la tabla que contiene a las longitudes de los prefijos (3, 5, 7). Las entradas apuntan a los prefijos correspondientes a esas longitudes.

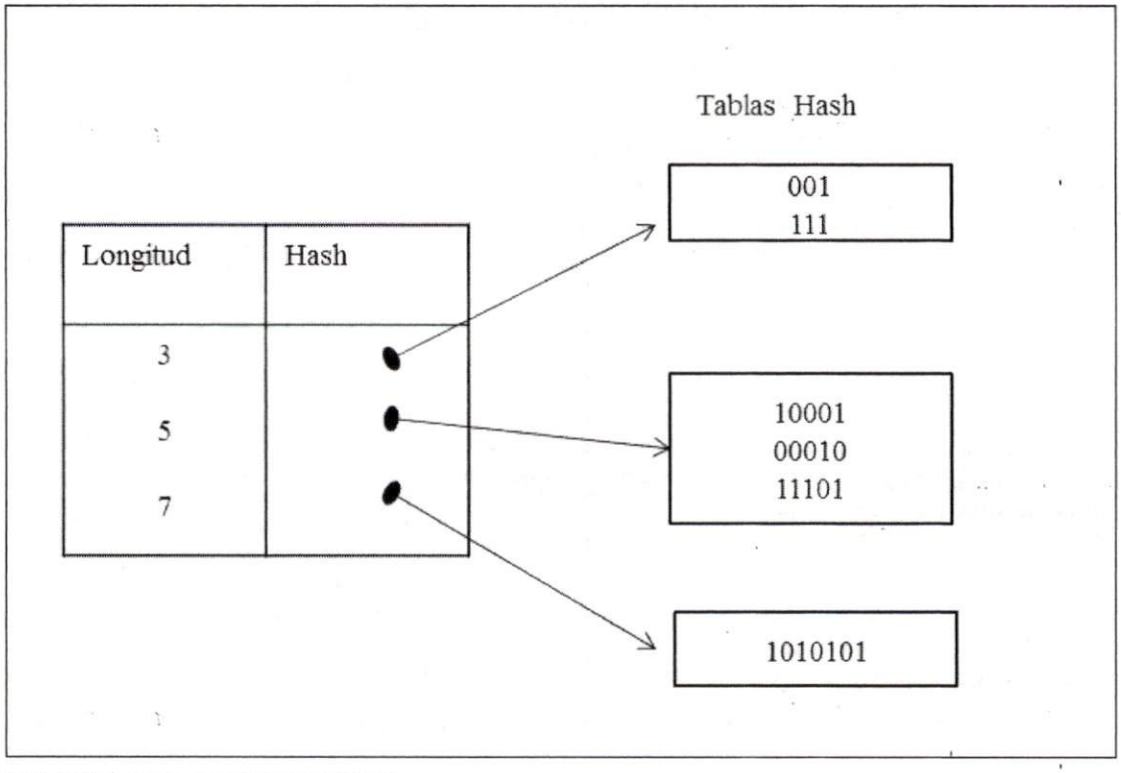


Figura 3-5 ejemplo de búsqueda lineal

Dado que se tiene un arreglo con las longitudes de los prefijos, la **búsqueda binaria** se efectúa en un arreglo que contiene las longitudes de los prefijos y las tablas *hash* a las que se apuntan. La condición es que en los elementos que contienen prefijos de longitud más grande se debe informar que existen elementos de menor longitud. Es por ello que dentro de la tabla existe un campo con una marca para determinar esta condición.

Otra forma de hacer la búsqueda en los elementos del arreglo es hacer una **búsqueda lineal asimétrica** en la que se utilizan estadísticas de las longitudes de los prefijos más utilizados para hacer la búsqueda sólo en determinadas entradas de las tablas de *hash*.

3.5 Búsquedas binarias

Este algoritmo de búsqueda binarias de Ju Hyoung [6] está inspirado en búsquedas en arboles binarios de Waldvogel [7]; la información de ruteo está contenida en las ramas de árbol binario definiendo prefijos desde la raíz del árbol hasta alguna hoja o nodo interno. El algoritmo separa a los nodos del árbol binario por nivel y los almacena en una tabla de *hash* para cada nivel.

La propuesta contempla hacer una mejora al algoritmo de Waldvogel mediante inspirado en Leaf-pushing [8] en donde un árbol binario representa a la tabla de ruteo de tal forma que se pueda definir a un prefijo desde la raíz del árbol hasta un nodo terminal u hoja; en otras palabras los nodos internos del árbol no constituyen a los prefijos. De esta forma solo se codifican prefijos por nivel en el árbol mediante tablas de *hash* haciendo más rápida la búsqueda.

Las principales características de este algoritmo es que el tamaño de las estructuras de datos utilizadas depende del tamaño de la tabla de ruteo y las instrucciones de búsqueda del prefijo más largo oscilan entre T_{nim} y T_{max} las cuales también son función del número y longitud de los prefijos en la tabla de ruteo. La ventaja de este algoritmo es que para tablas de ruteo reales (en el año de publicación: 2006) se necesitan de pocos accesos a memoria para obtener el prefijo más largo; a su vez, la desventaja es que el crecimiento acelerado de la red de Internet publica y para tablas de ruteo actuales y con más prefijos el desempeño en memoria e instrucciones es menor.

3.6 Búsquedas basadas en arboles binarios B

Este algoritmo propuesto por Yeim-Kuan [9] está basado en arboles binarios B; recordemos que en un árbol binario B los nodos son llamados páginas y que todas la paginas que son hojas están en el mismo nivel: así mismo las paginas están ordenadas según el valor de sus elementos de izquierda a derecha, de menor a mayor. En este algoritmo se propone el uso de una estructura de datos tipo árbol B el cual es llamado MMSPT (Multiway Most Specific Prefix Tree). La organización de los prefijos está dada por su valor binario en las claves del árbol B.

Este algoritmo tiene una complejidad en memoria dada por $O(n)$ para n prefijos en la tabla de ruteo. La instrucciones de búsqueda inserción y eliminación están dadas por la estructura de datos utilizada: $O(\log_m n)$, $O(m\log_m n)$, $O(m\log_m n)$, respectivamente donde m es el orden del árbol B.

Una vez más la ventaja de este algoritmo es que necesita una cantidad logarítmica de instrucciones para las instrucciones de búsqueda. Sin embargo; tanto la cantidad de memoria que se necesita para construir las estructuras de datos, así como las cantidad de instrucciones de búsqueda dependen del tamaño de la taba de ruteo y de la longitud de los prefijos que la constituyen.

3.7 Búsquedas Multi *hash*

Este algoritmo propuesto por Zhuo Huang [10] propone una manera de hacer uso de tablas de *hash* para hacer eficiente la representación de los prefijos en una tabla de ruteo. Se propone un algoritmo determinista con una sobrecarga de indexación pequeña para las entradas de la tabla *hash* y una distribución uniforme de los prefijos en las entradas de la tabla. También se reduce al mínimo el número de entradas a la tabla *hash* para tener un espacio de búsqueda más pequeño y eficiente.

En el algoritmo propuesto no se presenta una cota en memoria requerida o instrucciones necesarias para obtener el prefijo más largo, además de hacer una distribución de prefijos en base métricas de longitudes de prefijos existentes en tablas de ruteo de troncales de Internet; lo cual hace ver sus principales desventajas. Sin embargo, en comparación con algoritmos basados en tablas de *hash* simples se tiene un mejor rendimiento.

3.8 Algoritmos basados en la compresión

Entre otros usos, la compresión es un método que mejora la eficiencia de un algoritmo en términos de uso de memoria. Para el caso de las búsquedas del prefijo más largo, la compresión de las estructuras auxiliares que representan a la tabla de ruteo son codificadas en estructuras de datos más pequeñas [11]. La cualidad principal de estas estructuras es el tamaño que éstas tienen pues pueden ser guardas en memoria principal o *cache* de un enrutador. De esta forma los accesos a memoria para la búsqueda del prefijo más largo se realizan con la rapidez en que se tarda poner información en los registros del procesador y si las instrucciones de búsqueda y descompresión tienen una complejidad lineal, el procesamiento es aún más eficiente.

3.8.1 Algoritmo de la universidad de LULEA

Entre los algoritmos de búsqueda del prefijo más largo basados en la compresión, uno de los más representativos es el de la universidad de LULEA [12]. En este algoritmo se representa la información de ruteo en las ramas de un árbol binario completo. Del árbol se obtienen tres arreglos que constituyen la compresión y para obtener el prefijo más largo se utiliza una dirección IP como llave. El número de instrucciones para la descodificación es constante y no depende del tamaño de la tabla de ruteo. La estructura de árbol binario completo desaparece una vez realizada la compresión.

Este algoritmo fue creado para IPv4 (direcciones de 32 bits) y obtiene prefijos de a lo más 16 bits (a lo que se conoce como primer nivel de búsqueda); 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). Los beneficios de este algoritmo son:

- Para prefijos de longitud menor o igual a 16 bits se tienen estructuras de datos que constituyen la compresión lo suficientemente pequeñas para ser contenidas en memoria *cache* o principal.
- Las instrucciones de búsqueda del prefijo son exactamente tres extracciones de bits y dos aritméticas.

Las características de este algoritmo tanto en espacio requerido para las estructuras de datos que utiliza, como en instrucciones de descodificación hacen que la obtención del prefijo más largo sea muy rápida con respecto a los algoritmos anteriores; sin embargo, para longitudes de prefijos mayores a 16 bits decrece el desempeño del algoritmo.

3.8.2 Tree BitMap

Este algoritmo presenta un sistema de búsqueda del prefijo más largo utilizando una estructura compacta que codifica la tabla de ruteo representada por un árbol binario muy grande [13]. El objetivo es reducir el tamaño del "Trie" para que se puedan hacer búsquedas más rápidas en estructuras más pequeñas. El Tree BitMap [14] es un algoritmo basado en arboles binarios multibit (un nodo representa varios bits) que permite una rápida actualización del árbol y menos requisitos de almacenamiento; el diseño del mapa de bits se basa en las siguientes observaciones:

- Los nodos del árbol representan a varios bits del prefijo (nodos multibit) y tienen dos funciones; la primera es apuntar a los nodos hijos y la segunda es contener información de ruteo.
- El algoritmo está diseñado para que pueda ser contenido en memoria de acceso aleatorio RAM.

- En un ciclo de reloj se pueden procesar hasta 256 bits por lo que se considera que se pueden hacer comparaciones de más de un bit por nodo aprovechando la arquitectura.
- Para mejorar el tiempo de búsqueda se utiliza una sola estructura auxiliar de búsqueda a diferencia del algoritmo de LULEA [12] que utiliza tres.

3.8.3 Algoritmo de la universidad de Pisa

Este algoritmo fue desarrollado en la universidad de Pisa, Italia y está basado en la compresión [15]. La tabla de prefijos se llama T y el número de prefijos que puede contener es $|T|=2^{m}$ donde m es la longitud de la dirección IP. Para m= 32 (IPv4) la complejidad computacional en memoria es de $O(2^{m/2} + |T|)$ en el peor de los casos; lo que significa que la compresión dependerá del número y longitud de prefijos que tenga la tabla de ruteo.

El algoritmo efectúa la compresión de los prefijos de máxima longitud m (la longitud máxima de los prefijos está entre 1 y 32 bits) efectuando así una búsqueda parcial del prefijo si m es distinta de 32. Independientemente de la longitud máxima del prefijo se necesitan 3 operaciones de extracción de bits para determinar cuál de ellos es el más largo correspondiente a la dirección IP destino.

La compresión está dividida en dos etapas: la primera es la fase de **expansión** donde implícitamente se obtienen la interfaces (recuérdese que cada prefijo apunta a una interfaz física de salida) de salida para todas las direcciones posibles de tamaño m; la segunda etapa de **compresión** se fija un valor de $1 \leq k \leq m$ encontrando dos parámetros estadísticos de k.

En cuanto al desempeño del algoritmo los autores indican que si se cuenta con un procesador Pentium II con un ciclo de operación de 3.33 nano segundos por pulso de reloj; se pueden efectuar hasta 18 millones de búsquedas por segundo en una tabla de ruteo verdadera correspondiente al año de publicación (1999).

En este capítulo se abordaron algunos aspectos que introducen el razonamiento utilizado para hacer búsquedas del prefijo más largo. En el siguiente capítulo se utilizarán algunos aspectos e ideas principales de los algoritmos clásicos para concretar una propuesta que

minimice los problemas de instrucciones y memoria utilizada para hacer búsquedas en tablas de ruteo IP.

Capítulo 4 Propuesta de algoritmos de búsqueda en tablas de ruteo IP mediante vector de bits codificado

En este capítulo presentamos una propuesta para la realización de algoritmos de búsqueda del prefijo más largo que permite balancear el número de instrucciones y el tamaño de las estructuras de datos que se utilizan; se presenta un esquema flexible que optimiza el número de instrucciones o bien, el espacio requerido en memoria en bits, donde los valores de los parámetros de operación pueden ser optimizados de acuerdo a la capacidad y arquitectura de un enrutador. También se consideran las longitudes de los prefijos que componen a la tabla de ruteo como un factor que ayuda a determinar los parámetros adecuados de operación.

Para todos los algoritmos propuestos, la tabla de ruteo es representada por un árbol binario completo el cual se codifica en un vector de bits. El tipo de codificación del vector de bits determinará el grado de compresión del árbol binario completo.

Se presenta primero un esquema básico de codificación que optimiza el número de instrucciones del algoritmo de búsqueda del prefijo más largo; pero que requiere una cantidad de memoria que aumenta de forma exponencial conforme aumenta la longitud máxima de los prefijos a encontrar. Se proponen después otros algoritmos de búsqueda que *atenúan* este comportamiento, reduciendo el costo invertido en las estructuras de datos utilizadas. Reducir el costo en memoria tiene como consecuencia un incremento de instrucciones, por ello se presenta el algoritmo que ofrece la solución que optimiza el número instrucciones y el algoritmo que optimiza el tamaño en memoria requerido,

ofreciendo la posibilidad de balancear el costo del número de instrucciones y memoria según sean las necesidades y capacidades de un dispositivo de encaminamiento o enrutador.

4.1 Representación de una tabla de ruteo mediante un árbol binario

Una tabla de ruteo es básicamente una estructura de datos que posee información de las redes que son alcanzables por el enrutador que la contiene. Esta estructura está constituida por prefijos (información de redes alcanzables) ligados a un enlace de salida (dirección física) entre otras cosas. La forma de acceder a esta tabla es mediante un índice que nos sitúa en algún prefijo en particular.

El algoritmo de búsqueda del prefijo más largo necesitará a los prefijos con sus respectivos enlaces de salida y por supuesto una dirección destino para encontrar el prefijo más largo en la tabla. Además podemos ver a los prefijos como una cadena de bits de hasta 32 o 128 bits (IPv4 e IPv6 respectivamente). Un árbol binario puede representar a la tabla de ruteo, y una trayectoria en el árbol desde la raíz hasta una hoja o nodo puede definir un prefijo. La profundidad del árbol binario necesaria para representar a todos los prefijos debe ser igual a la longitud máxima de los prefijos en la tabla. Sin embargo, si la profundidad es menor a la longitud máxima de los prefijos sólo se podrán hacer búsquedas parciales del prefijo ya que no se podrá representar todo el prefijo en el árbol. Entonces, si la profundidad del árbol binario es 32 o 128 (IPv4 o IPv6, respectivamente) se podrán obtener todas la trayectorias que definen a los prefijos de redes como Internet.

Un árbol binario de profundidad k podrá contener todos los prefijos de longitud máxima k bits puesto que el árbol puede tener 2^k hojas y por lo tanto representará a 2^k direcciones IP de longitud k. En la figura 4-1 se muestran los prefijos en una tabla y su representación en el árbol binario. Nótese que la raíz está en la profundidad cero y desde ahí se definen

trayectorias que terminan en las hojas denotadas por un círculo negro[8] representando así un prefijo en el árbol. La raíz es el ancestro de todos y representa un prefijo, pues si ningún elemento de la tabla está contenido en la dirección IP destino se tiene que reexpedir el paquete por la salida por defecto (*default*) que es representada por la raíz del árbol binario.

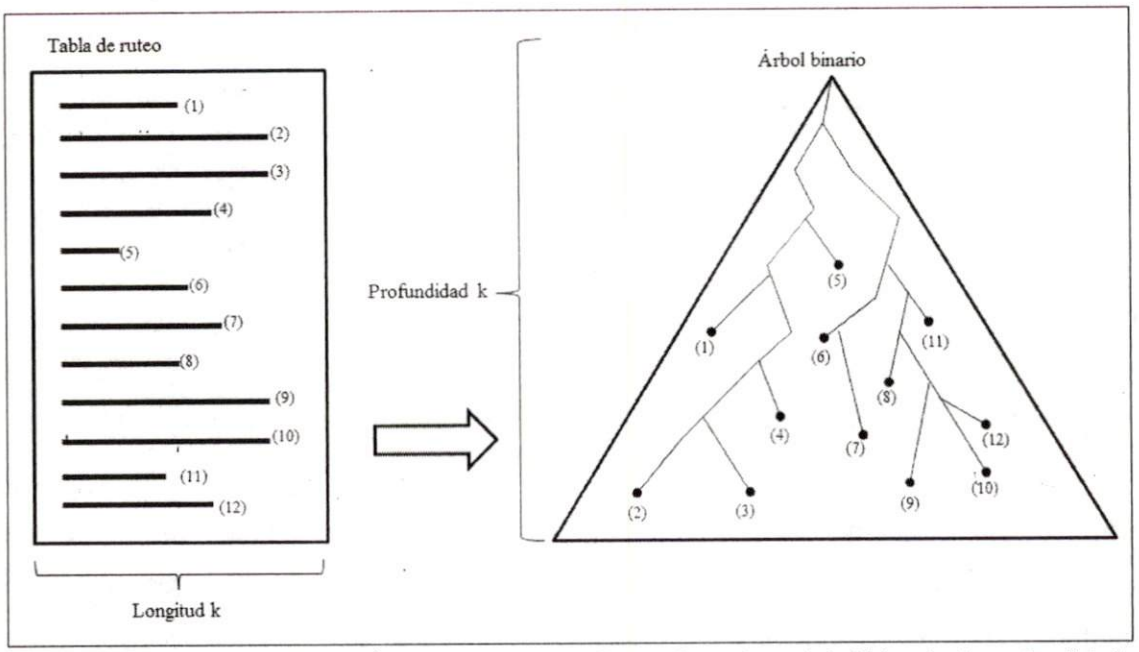


Figura 4-1 Representación de una tabla de prefijos, mediante el uso de un árbol binario de profundidad *k*. Cada prefijo define una trayectoria en el árbol binario

En la figura 4-2 se muestra un árbol binario de profundidad $k = 2$ el cual sólo está conformado por tres prefijos: la ***raíz***, el prefijo ***0*** y el prefijo ***11***. Nótese que las direcciones IP son de longitud 2 ya que sólo se permiten $2^k = 2^2 = 4$ direcciones IP y a cada una de ellas le corresponde un solo prefijo más largo.

Una tabla de ruteo contiene prefijos de longitud máxima k. Un prefijo de longitud p tal que $p \leq k$ está contenido en 2^{k-p} direcciones IP de longitud k. Por ejemplo, en la figura 4-2 se tienen tres prefijos: ***0***, ***default***, y ***11*** y se tienen direcciones IP de longitud k=2 (***00***, ***01***, ***10***, ***11***). El prefijo ***0***, es de longitud $p = 1$ y está contenido en $2^{2-1} = 2$ direcciones IP que son ***00*** y ***01***.

[8] En ilustraciones posteriores de árboles binarios, los círculos negros determinan el final de un prefijo, los círculos grises son nodos internos o subprefijos y los círculos punteados son nodos inexistentes que solo se muestran para poder visualizar de mejor manera la arquitectura del árbol binario.

Dirección IP	*Prefijo correspondiente*
00	0
01	0
10	*Default*
11	11

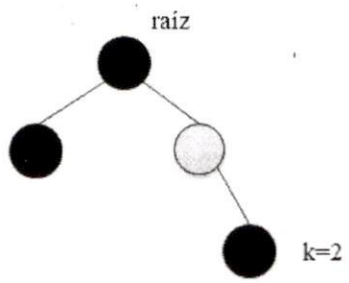


Figura 4-2 Árbol binario de profundidad 2 con los prefijos correspondientes a cada dirección IP

Un nodo en el árbol que se encuentra en la profundidad p representa a un intervalo de direcciones IP ya que está contenido en éstas. En la figura 4-3.a se puede observar que un prefijo en p cubre o representa 2^{k-p} direcciones IP. Sin embargo, existen direcciones IP que no están cubiertas por una hoja. En vez de ello están cubiertas por un prefijo en un nodo interno y además existen intervalos de direcciones que sólo están cubiertas por la raíz, la cual cubre a todas las direcciones IP.

Si se diera el caso especial en que todas las hojas en el árbol fueran prefijos y que todas la direcciones IP estuvieran cubiertas por alguna hoja (que no sea la raíz), entonces podríamos ordenar a los prefijos de acuerdo a un recorrido *en-orden*[9] del árbol binario y cada hoja donde termina un prefijo representaría un intervalo de direcciones IP ordenadas según su valor numérico de forma ascendente. En la figura 4-3.b. se puede ver que todos los prefijos cubren todas las direcciones IP de forma ordenada pues los prefijos cubren un intervalo de direcciones de acuerdo a su valor numérico.

[9] El recorrido *en-orden* que se efectúa toma en cuenta las hojas en el árbol de izquierda a derecha.

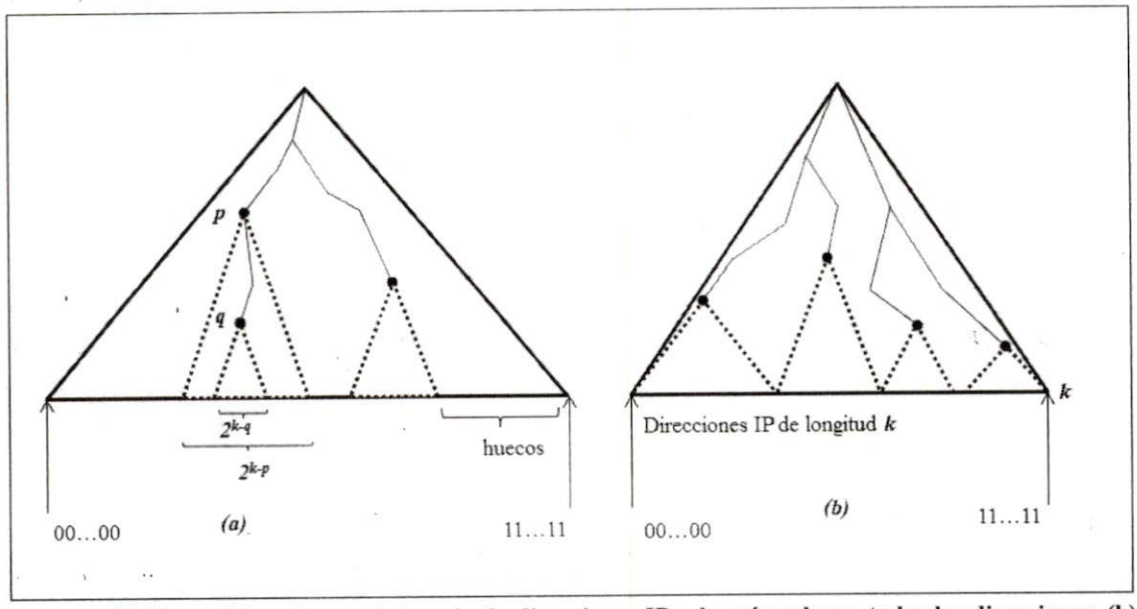


Figura 4-3 (a) Un prefijo cubre un intervalo de direcciones IP y la raíz cubre a todas las direcciones. (b) Si las hojas o prefijos cubren todas las direcciones IP, podemos agruparlos de acuerdo al valor numérico de las direcciones IP

El conjunto de prefijos de una tabla no necesariamente corresponde a un árbol como el de la figura 4-3.b. Sin embargo, es sencillo obtener este tipo de árbol; solamente se agregan nodos de tal forma que se forme un árbol binario completo. Un árbol binario completo según la teoría de graficas es aquel en que cualquiera de sus nodos tiene a dos hijos o a ninguno de ellos. Transformar un árbol binario a uno que sea completo repercute en la tabla de ruteo pues se están agregando elementos a ella, por lo que se debe construir una tabla equivalente que posea la misma información de ruteo que la original; pero que corresponda a un árbol binario completo. En la figura 4-4 se muestra una tabla de ruteo y su respectiva representación mediante un árbol binario, después se hace la transformación del árbol a uno que sea completo y se construye la nueva tabla de ruteo. Al construir la tabla de ruteo se hace un recorrido *en-orden* del árbol binario para encontrar todos los prefijos que este contiene; nótese que al hacer un recorrido *en-orden* de la hojas, los prefijos estarán ordenados de acuerdo a su valor numérico de longitud k (sí un prefijo es de longitud menor a k, se completa la longitud k agregando ***ceros***) de menor a mayor pues se recorre el árbol de izquierda a derecha.

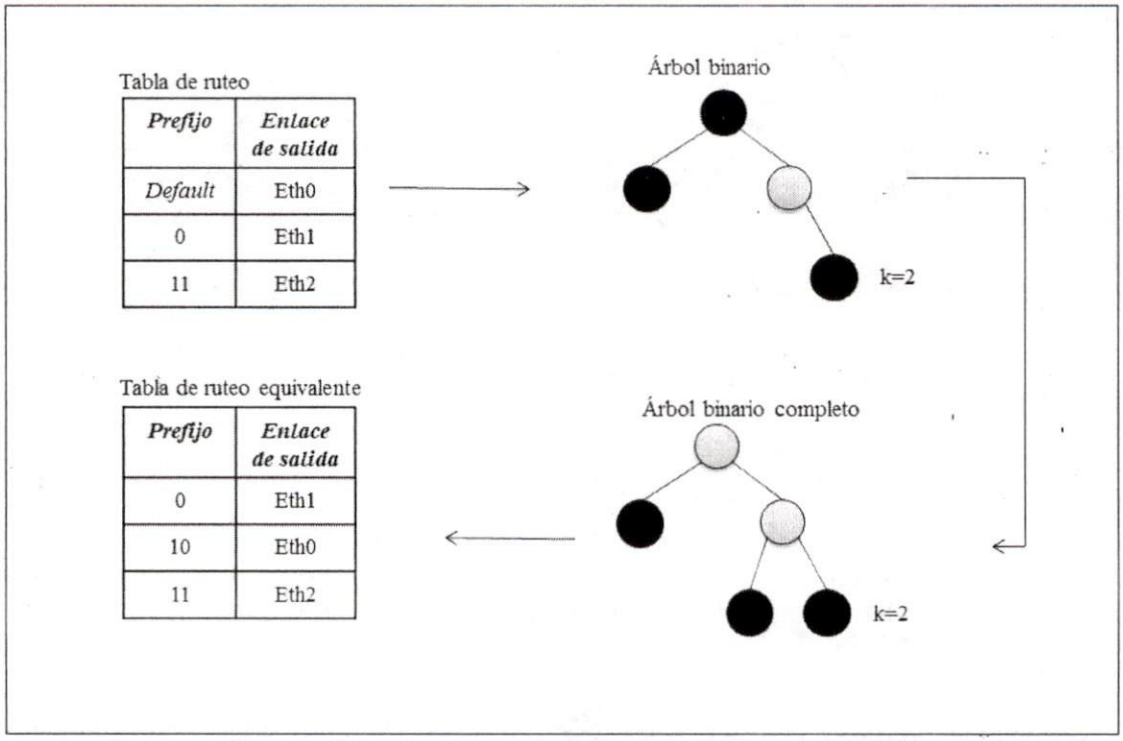


Figura 4-4 Trasformación a un árbol binario completo y su repercusión en la tabla de prefijos

En este trabajo, ordenar a los prefijos significa que se acomodarán en la tabla de ruteo de acuerdo a un recorrido *en-orden* del árbol binario donde se escriben las trayectorias desde la raíz hasta una hoja en cada entrada de la tabla. Otra forma de ordenar a los prefijos de longitud $p \leq k$ es mediante su valor numérico; a cada prefijo de longitud $p \leq k$ se le agregan $k-p$ ceros y se ordenan de menor a mayor según su valor numérico de forma ascendente.

4.2 Algoritmo de búsqueda directa mediante vector de bits

Partiendo del supuesto de que se tiene una tabla de ruteo ordenada y que corresponde a un árbol binario completo, se puede efectuar una búsqueda del prefijo más largo dada una dirección IP destino. La búsqueda consiste en considerar la dirección IP destino como un valor numérico; de esta forma una búsqueda consiste en recorrer el árbol de izquierda a derecha para encontrar el prefijo que representa un intervalo de direcciones IP en donde una de ellas es la IP destino.

4.2.1 Vector de bits

Cada prefijo ordenado representa a un intervalo de direcciones IP ordenadas de acuerdo a su valor numérico. Una dirección IP destino se encuentra cubierta por algún prefijo, entonces: *el número de prefijos que se encuentran a la izquierda de la dirección IP en el árbol es exactamente la posición del prefijo más largo en la tabla de ruteo*, por lo que las búsquedas consisten en hacer recorridos en *en-orden* del árbol, contando el número de hojas que existen a la izquierda de donde se forma una trayectoria dada por la dirección IP destino.

Recorrer el árbol binario para contar el número de hojas a la izquierda por cada dirección IP destino de un paquete que llega a un enrutador es ineficiente pues el árbol completo de profundidad k puede tener hasta 2^k prefijos. Además, los prefijos correspondientes a las direcciones IP de un valor numérico grande se encuentran a la derecha del árbol, por lo que serán los últimos donde se realice una búsqueda implicando que dependiendo del valor de la dirección IP será la rapidez de búsqueda. Por lo tanto, existirá inequidad al efectuar búsquedas.

Se necesita una estructura de datos que posea información de los prefijos y el número de direcciones IP cubiertas por ellos para hacer una búsqueda lineal en ella. La estructura que utilizamos es llamada **vector de bits** y es un arreglo de 2^k elementos de 1 bit cada uno de

ellos por lo que el costo en memoria para construirlo será de 2^k bits, donde k es la profundidad del árbol binario. En la figura 4-5 se pueden ver las hojas que representan prefijos y el intervalo de direcciones que representa cada uno de ellos.

Para dar valores al vector de bits se hace un recorrido *en- orden* del árbol binario completo; por cada hoja en la profundidad p se coloca un ***"1"*** en la primera posición disponible del vector de bits y 2^{k-p}-1 ceros en sus siguientes posiciones. El vector de bits se llena de izquierda a derecha, desde la primera casilla hasta la última. Recuérdese que cada casilla en el vector de bits representa una hoja de un árbol de profundidad k. De esta forma cuando se encuentre un ***1*** en el vector de bits se sabrá que existe un prefijo que representa un determinado grupo de direcciones IP hasta el siguiente ***1***. Nótese que la primera casilla del vector de bits siempre comienza con un ***1*** porque el árbol por lo menos tendrá a la raíz como prefijo por defecto.

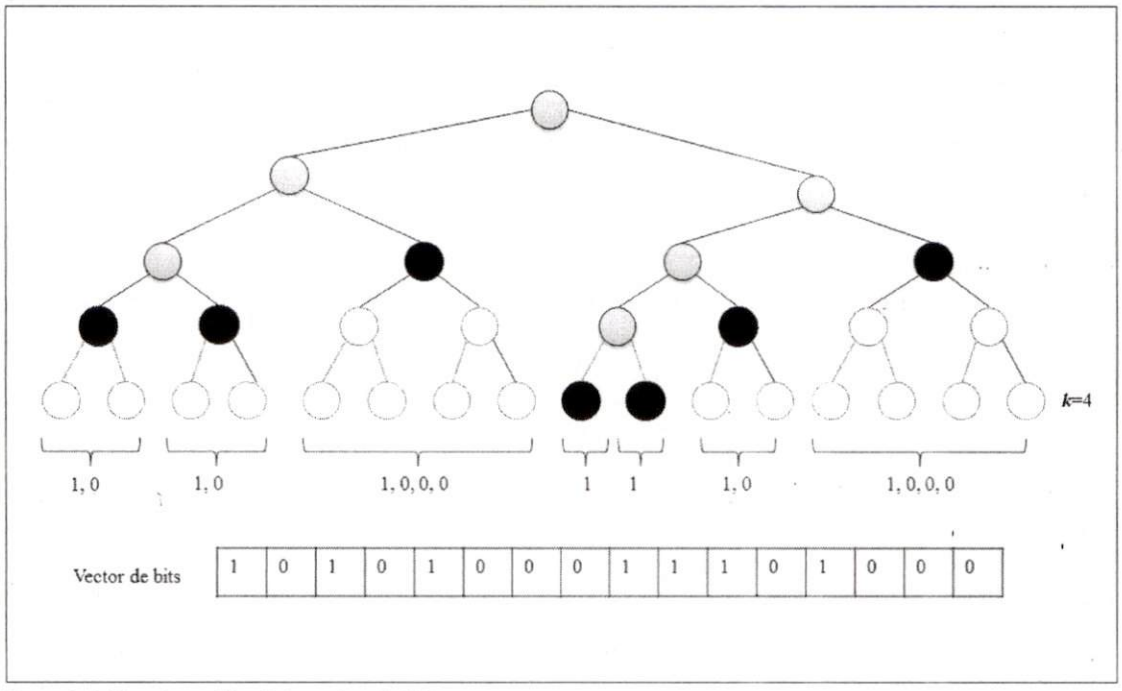


Figura 4-5 Construcción del vector de bits.

4.2.2 Búsqueda mediante un arreglo contador

Una vez definido y construido el vector de bits se pueden efectuar búsquedas mediante el conteo de bits *"1"* que se encuentran en éste. La búsqueda es simple, una dirección IP define una trayectoria en un árbol binario de profundidad k y su valor numérico define una posición en el vector de bits. Entonces, el índice del prefijo más largo correspondiente a la dirección IP será el *i-ésimo* prefijo en la tabla de prefijos ordenada que es exactamente igual al número de bits *"1"* a la izquierda de donde se localiza la dirección IP en el vector de bits. En la figura 4-6 se ilustra un ejemplo de la relación entre el vector de bits y la tabla de prefijos ordenada.

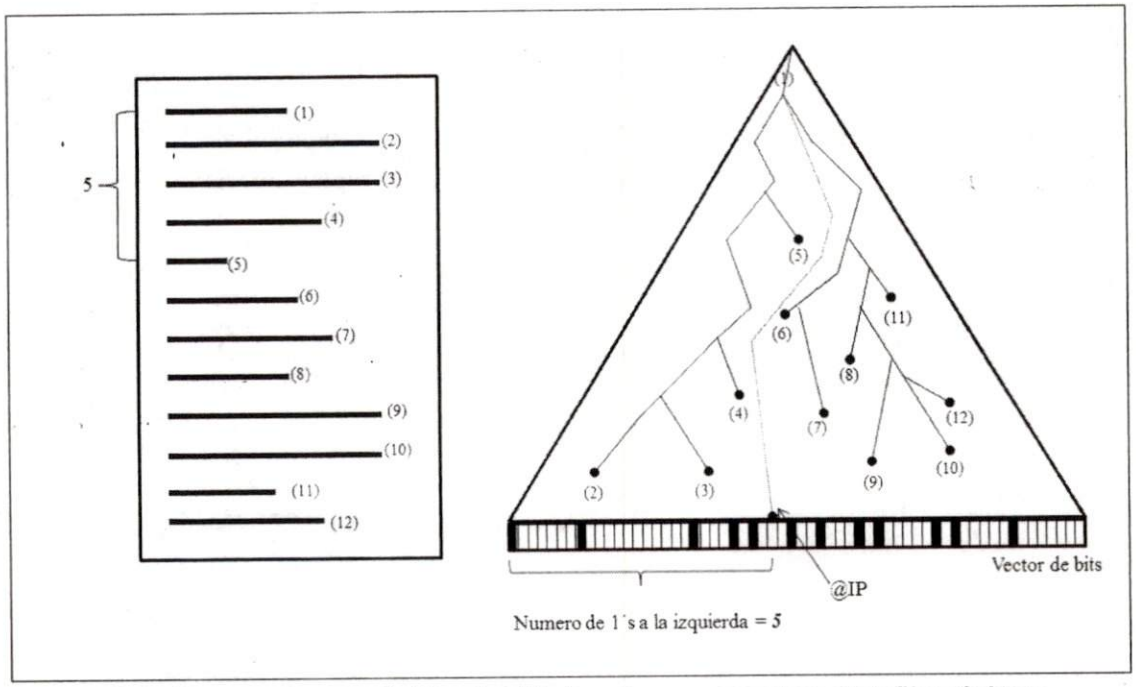


Figura 4-6 Relación entre el vector de bits y la tabla de ruteo para encontrar el prefijo más largo

Entonces, *el número de bits "1" a la izquierda corresponde al índice del prefijo más largo en la tabla de ruteo ordenada.* Una dirección IP de valor numérico 2^{k-1} (111111...111) apunta a la última posición del vector de bits, por lo que para saber cuál es el índice del prefijo más largo se tienen que contar todos los bits *"1"* en el vector completo. Esto es ineficiente y poco equitativo en comparación con direcciones IP de menor valor numérico.

Para evitar esta inequidad y mejorar la velocidad de búsqueda se puede hacer uso de un contador de bits puestos en *"1"* que lleve la cuenta de prefijos (*"1"*s) en cada posición del vector. El contador tiene que ser un arreglo de la misma longitud que la del vector de bits siendo su tamaño 2^k elementos y dado que es un contador de prefijos de longitud máxima k, entonces cada casilla es un contador de k bits; con k bits en cada casilla se puede contar hasta 2^{k-1}. La regla para la construcción del contador es la siguiente:

Desde la primera hasta la última posición del vector de bits y del contador:

- Se cuenta el número de bits puestos en *"1"* a la izquierda en el vector de bits
- En el arreglo contador se coloca el conteo de bits puestos en *"1"* menos 1(la resta se efectúa porque el índice de la tabla de ruteo comienza en cero).

El uso de un arreglo contador de bits puestos en *"1"* en el vector de bits es el primer tipo de codificación que presentamos. En la figura 4-7 se muestra un ejemplo de un vector de bits de 16 elementos y su respectivo contador construido bajo las reglas anteriores.

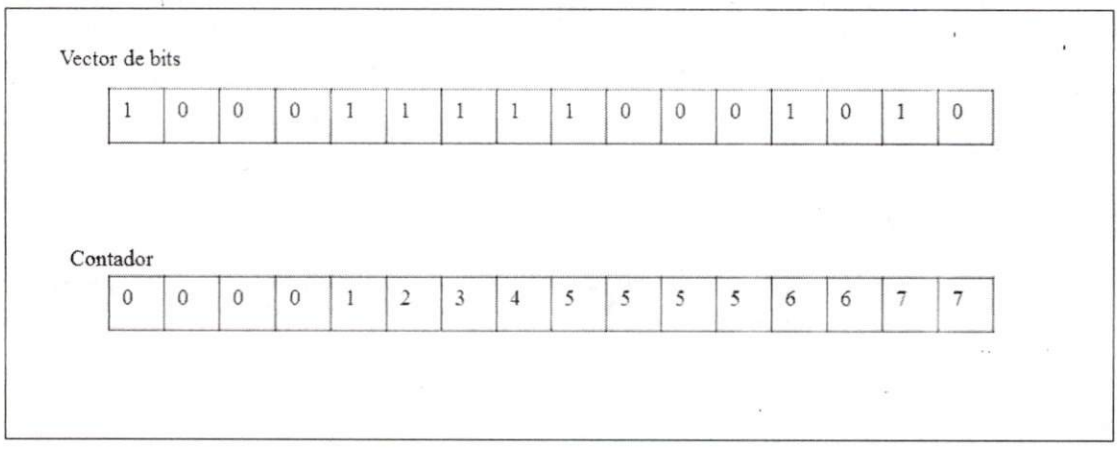


Figura 4-7 Construcción del contador mediante el vector de bits

De esta manera, la búsqueda del prefijo más largo consiste simplemente en tomar a la dirección IP destino como un valor numérico ***IP*** y acceder a esa posición del contador para utilizar el valor que almacena como el índice que da la posición del prefijo buscado dentro de la tabla de ruteo, por lo tanto: ***índice = contador[dirección IP].*** En la figura 4-8 se ilustra la búsqueda mediante este algoritmo.

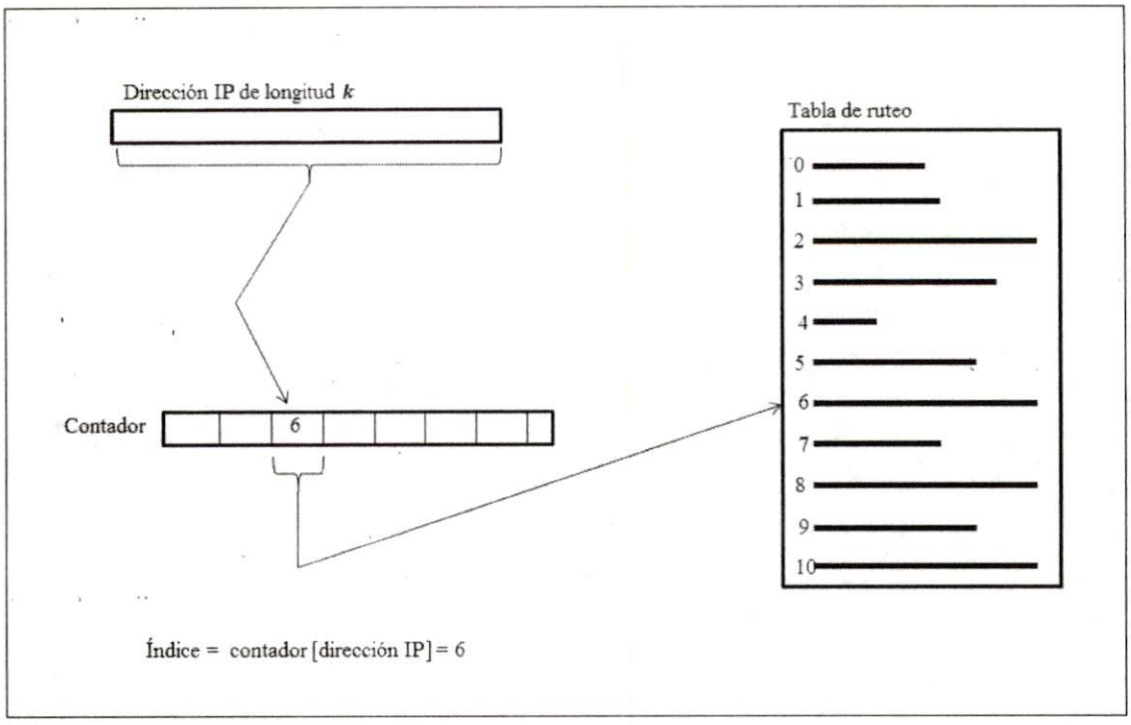


Figura 4-8 Algoritmo de búsqueda representado de forma gráfica

En la figura 4-9 se presenta un ejemplo de un árbol de profundidad k=3, su vector de bits y su arreglo contador junto con la tabla de ruteo correspondiente. Nótese que para cualquier dirección de longitud k=3 sólo se necesita colocarse en la posición del contador dada por la dirección IP e inmediatamente se obtendrá el índice del prefijo más largo. El índice de acceso a la tabla de prefijos comienza con 0 y no con 1, esto es importante pues si no se lleva correctamente el conteo se puede obtener un prefijo equivocado pues ***tabla[i]*** $\neq$ ***tabla[i-1]***.

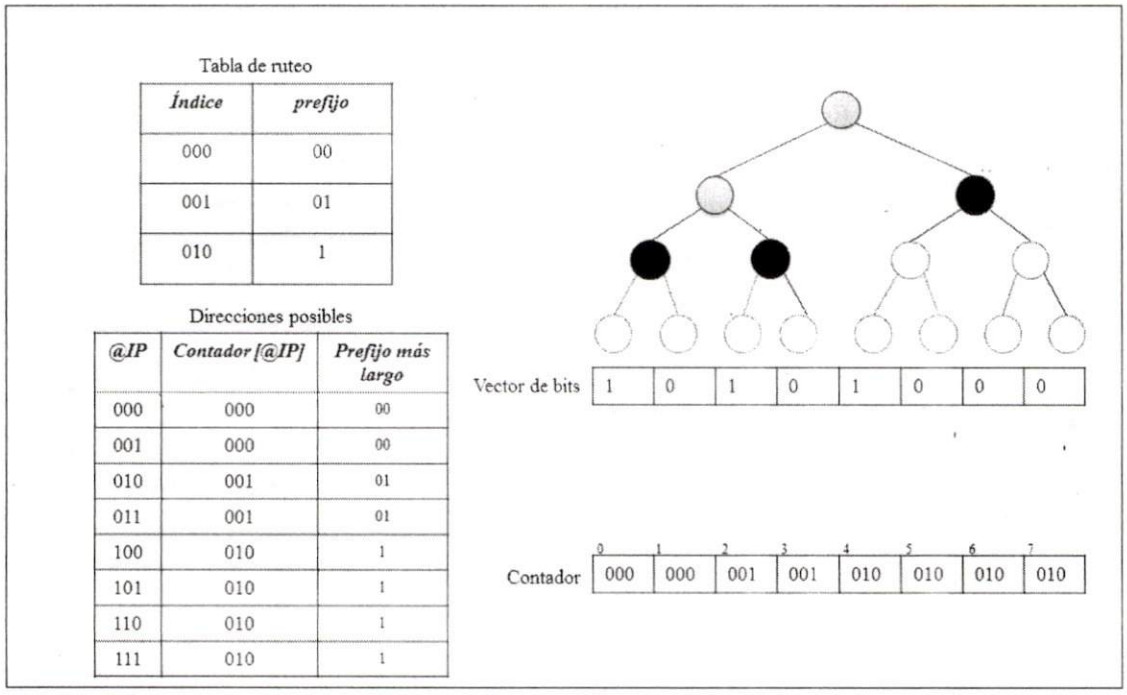


Figura 4-9 Ejemplo de búsqueda en un árbol de profundidad k=3

4.2.3 Función de costo en memoria

El algoritmo consiste en construir el árbol binario, hacerlo completo, construir su tabla equivalente, construir el vector de bits y construir el contador a partir del vector de bits. La búsqueda del prefijo más largo sólo consiste en posicionarse en una casilla del contador por lo que tanto el árbol como el vector de bits son desechables. También quedó definido que se puede hacer una búsqueda de cualquier longitud k construyendo su respectivo árbol de profundidad k. A continuación se analiza el costo del algoritmo en cuanto al tamaño de las estructuras de datos requeridas.

El espacio requerido en memoria dependerá de la profundidad k del árbol binario que se construya pues se requerirá un contador de 2^k elementos de k bits cada uno de ellos. Se define a f_m como la función de costo en bits por realizar búsquedas del prefijo más largo de máxima longitud k enunciada por ecuación.

$$f_m = k\,2^k \quad \text{tal que } k \text{ es la profundidad del árbol.} \tag{4.1}$$

La función f_m está dado por el tamaño del arreglo contador multiplicado por el tamaño de cada entrada del arreglo. Nótese que el valor de k no es una variable, es un parámetro de búsqueda. Por ejemplo sí deseamos encontrar prefijos de longitud máxima 16 fijamos a k=16 y para completar la búsqueda hasta 32 bits (IPv4) se tiene que repetir el algoritmo o bien efectuar otro tipo de búsqueda.

El costo en instrucciones y memoria es independiente de la tabla de ruteo; sin embargo, el costo en memoria tiene un comportamiento exponencial con respecto a k. Para direcciones IPv4 e IPv6, debido al costo, no es factible hacer búsquedas con estructuras de datos contenidas totalmente en memoria principal; aunque se pueden efectuar búsquedas de prefijos de menor longitud que el tamaño de las direcciones y volver a aplicar este algoritmo o cualquier otro tipo de búsqueda para cubrir los prefijos de longitudes mayores cuando sea necesario.

4.2.4Función de costo en instrucciones

El número de instrucciones requeridas para efectuar una búsqueda es igual a una operación de acceso a memoria pues el índice del prefijo más largo es igual a: **índice=contador[dirección IP]** por lo que podemos decir que la búsqueda es optimizada en cuanto al número de instrucciones. Definimos a f_i como la función de costo en instrucciones de acceso a memoria por realizar búsquedas de prefijos más largos de longitud máxima k que está dada por la ecuación (4.2). Nótese que f_i es una constante que no cambia y no es afectada por el valor de k.

$$f_i = 1 \quad \text{operación de acceso a memoria} \tag{4.2}$$

Este tipo de codificación del vector de bits optimiza el número de instrucciones de acceso a memoria pues se pueden realizar búsquedas completas de 32 o 128 bits haciendo una sola extracción de bits, sin embargo se tiene que pagar un costo en memoria dado por la ecuación (4.1).

4.3 Algoritmo de búsqueda mediante conteos parciales en el vector de bits

En la sección anterior se describió la búsqueda directa que consta de un arreglo contador de bits *"1"* en el vector de bits. En este tipo de codificación se tiene una optimización en acceso a memoria pero se presenta un comportamiento exponencial al espacio en memoria requerido de acuerdo a la profundidad k del árbol. Para reducir el costo en memoria se propone asumir que un árbol binario está compuesto de subárboles y que en cada uno de ellos se puede aplicar el algoritmo de búsqueda, aplicando la filosofía de *divide y vencerás*.

4.3.1 Conteos parciales

El vector de bits de tamaño 2^k bits es la representación de la posición e intervalos de direcciones IP que representan todos los prefijos del árbol en la profundidad k. Búsquedas del prefijo más largo consisten en hacer conteos de bits puestos en *"1"* del vector de bits; por ello sólo se cuenta con "un solo" vector de bits que corresponde a la profundidad k del árbol. En esta sección se presentan formas de hacer conteos parciales de los bits puestos en *"1"* del vector de bits para hacer reducir el tamaño de las estructuras que guardan los conteos de bits *"1"*.

Recuérdese que los k bits de una dirección IP son utilizados como un valor binario para acceder a una entrada de un arreglo contador de 2^k casillas determinando así, el número de bits puestos en *"1"* en el vector de bits. Sin embargo, si se toman p bits, $p < k$ se tendrá un apuntador a la tabla al primer prefijo más largo de longitud p; de esta forma se define una búsqueda parcial del prefijo más largo. Visto desde el árbol binario se tiene que en la profundidad p se efectuó un corte del árbol de profundidad k y se tiene un conteo parcial

de las hojas (prefijos) de los 2^p subárboles a la izquierda del subárbol donde apuntan los p bits más significativos de la dirección IP.

El árbol binario se puede partir en cualquier profundidad y se pueden tener tantas particiones o cortes como niveles en el árbol. En cada profundidad donde se realiza un corte se define un conteo parcial de hojas en los subárboles definidos o bien de bits puestos en *"1"* en el vector de bits. Para obtener el prefijo más largo de una dirección, dado que se tienen n conteos parciales, sólo se tienen que sumar los n conteos parciales y el resultado será el índice de acceso a la tabla de ruteo ordenada. En la figura 4-10 se presentan dos cortes en el árbol de búsqueda de profundidad k, el primer corte se realiza en la profundidad p y el segundo está situado en la profundidad q; $p < q < k$. En cada profundidad se determina el número de bits "*1*" según la IP destino.

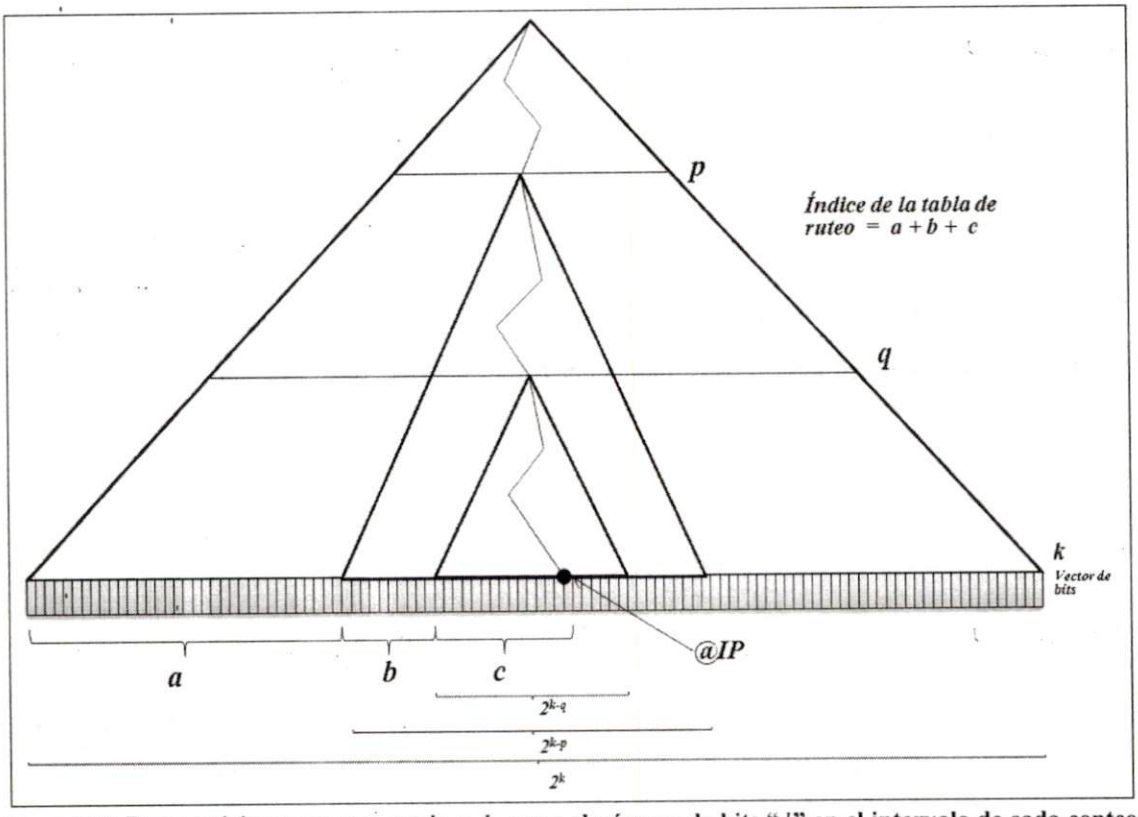


Figura 4-10 Dos particiones en $p < q < k$; a, b, c son el número de bits "1" en el intervalo de cada conteo parcial

Para el ejemplo de búsqueda del prefijo más largo con tres conteos parciales del vector de bits, mostrado en la figura 4-10, se utiliza el árbol binario para realizar de forma gráfica dicha cuenta de bits "*1*". Sin embargo, para efectuar búsquedas sólo se hace uso de arreglos contadores de bits puestos en *"1"* en intervalos definidos por las profundidades de corte del árbol binario. Del árbol se puede ver que para n cortes en el árbol se necesitan $n+1$ contadores de bits "*1*". En el ejemplo de la figura 4-10 se tienen 3 contadores, uno en la profundidad p de tamaño 2^p casillas de k bits cada una de ellas, el segundo contador está situado en q y es de tamaño 2^q casillas de $k-p$ bits cada una de ellas y el tercer contador es de tamaño 2^k elementos de $k-q$ bits cada uno de ellos. Se puede ver que el tamaño de los contadores está dado por la profundidad en que se efectuó el corte (p o q para este ejemplo) y con respecto a k.

Para llenar de valores a los contadores se distinguen dos tipos de búsqueda: búsqueda parcial y la última profundidad de búsqueda, esto porque los n cortes en el árbol representan n búsquedas parciales y se necesita un último conteo para completar la búsqueda en la profundidad k del árbol denotado como última profundidad de búsqueda. Los contadores tienen diferentes reglas para ser llenados; en una búsqueda parcial en la profundidad q se tiene un contador de tamaño 2^q elementos de $(k-p)$ bits cada uno de ellos, tal que p es la profundidad de la búsqueda anterior, $p=\{0,1,2,3.....k-2\}$ y $p < q$. El contador se llena bajo las siguientes reglas.

Para llenar el arreglo contador situado en la profundidad q que tiene 2^q elementos.

1. Se recorre el vector de bits en intervalos de 2^{k-q} bits.
2. Se cuenta el número de bits puestos en *1* en cada bloque de 2^{k-q} bits y se coloca en la casilla del contador a manera de *offset* (el *offset* comienza con un cero y se coloca en la primera casilla del contador pues no hay prefijos a la izquierda del primer nodo del árbol en un recorrido *en-orden*).
3. Cada 2^{k-p} bits se resetea el *offset* a cero (p es la profundidad de la partición anterior a q y puede ser cero cuando no hay más búsquedas en niveles superiores).

4. Se analiza si el primer elemento de los 2^{k-q} bits es igual a *1*, si no es así, el *offset* se detiene (el conteo se detiene) hasta encontrar un bloque de 2^{k-q} bits que contenga un *1* al inicio o bien se inicializa a cero si ya han pasado 2^{k-p} bits. Esto porque la presencia de un *1* al inicio del bloque indica que existe un nodo en q.

En la figura 4-11 se muestra de forma gráfica cómo dar valores al contador. Nótese que el vector de bits es dividido en intervalos dados por las profundidades de corte en el árbol; para el contador de q elementos se analiza el vector de bits cada 2^q bits y se cuenta a manera de *offset* el número de bits *"1"*; cada 2^p bits se inicializa el *offset* y si el primer elemento de los q bloques del vector de bits es cero se detiene la cuenta del *offset* hasta encontrar al principio de otro bloque un *1* (al menos que se tenga que inicializar el *offset*).

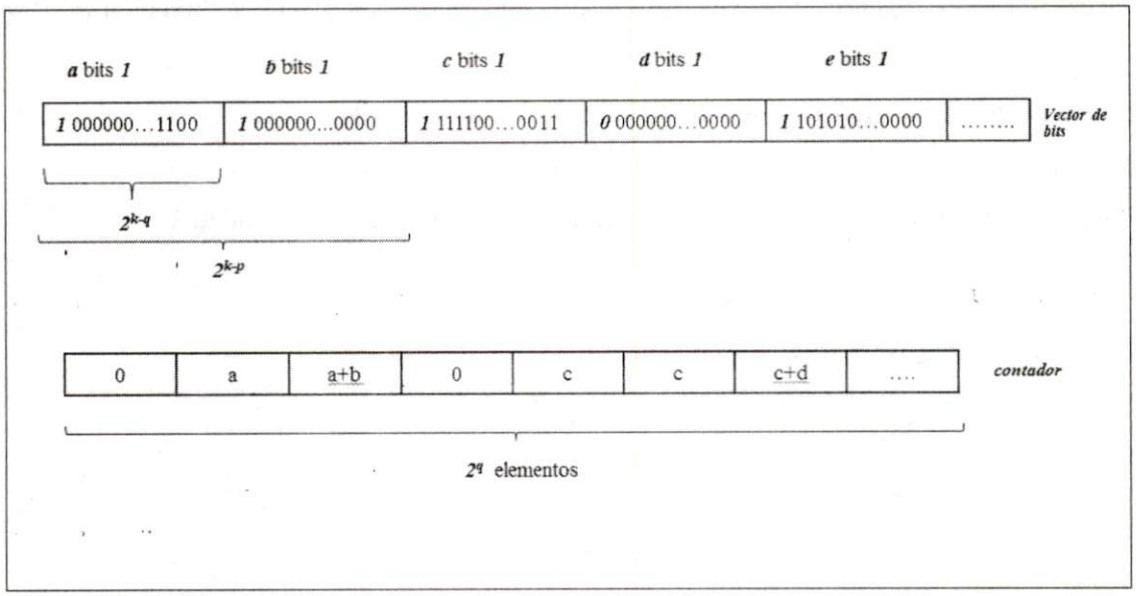


Figura 4-11 Valores de los contadores

Para la última profundidad de búsqueda que también puede ser denotada como búsqueda directa se crea un arreglo contador de tamaño 2^k $(k-r)$ donde r es la última partición en el árbol y sus valores son obtenidos bajo las siguientes reglas:

1. Se recorre el vector de bits en intervalos de 2^{k-r} bits y bit a bit.
2. Para cada elemento del contador se cuenta el número de bits "*1*" en el bloque de 2^{k-r} bits del vector de bits que se encuentran desde el inicio del bloque, se resta en 1 y se coloca el valor obtenido en el contador (para valores iguales a cero no se efectúa la resta).

Este tipo de búsqueda determina otra forma de codificación del vector de bits; dicha codificación dependerá del número de particiones en el árbol y la posición en que se encuentra cada una de ellas.

La búsqueda del prefijo más largo mediante varios contadores se realiza mediante el uso de la dirección IP destino. Por ejemplo para las dos profundidades de búsquedas p y q realizadas en el árbol de profundidad k (ver figura 4-10) se tienen tres contadores y el acceso a cada contador es de la siguiente forma: los primeros p bits de la dirección IP para acceder al primer contador, para el segundo contador se utilizan los primeros q bits de la dirección destino y para el tercer contador se utilizan los k bits que forman a la dirección IP. En la figura 4-12 se muestra el algoritmo de búsqueda de forma gráfica para dos búsquedas parciales internas y una búsqueda final en el árbol de profundidad k.

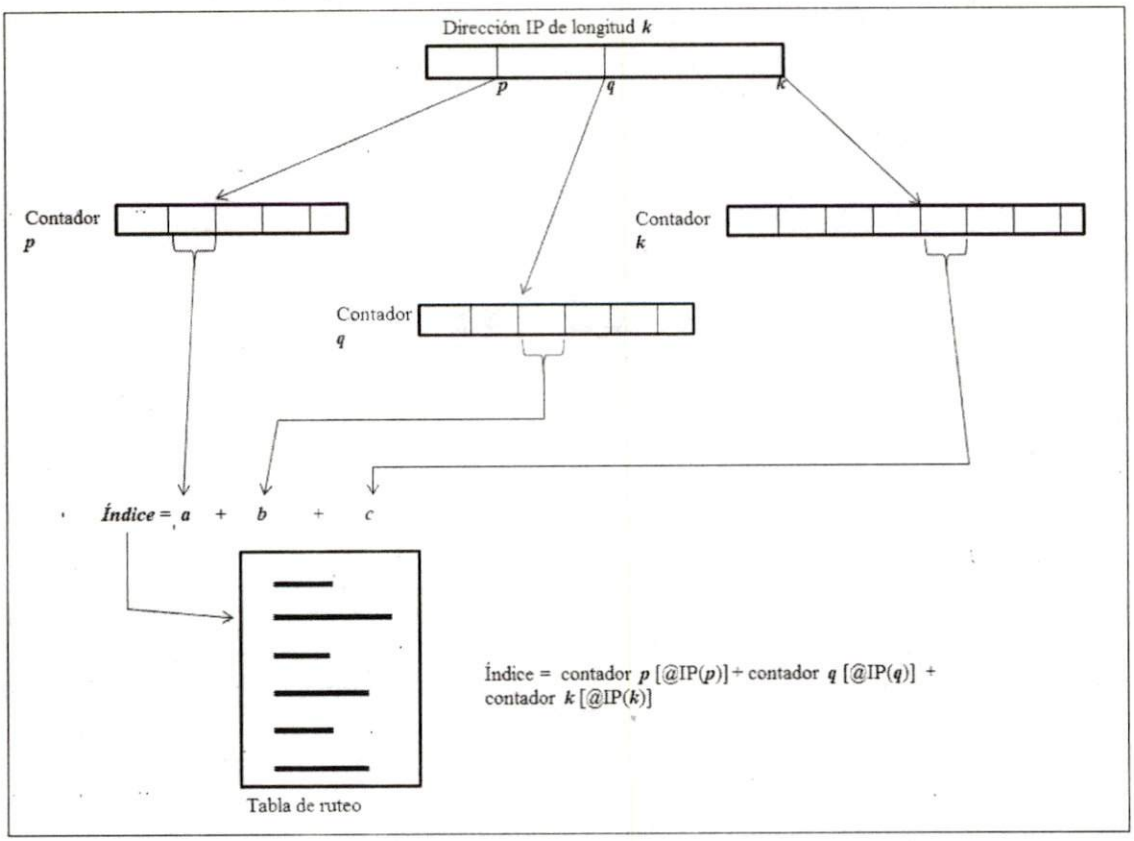
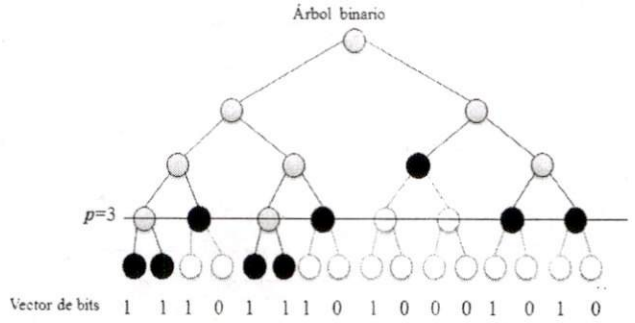


Figura 4-12 Algoritmo de búsqueda representado de forma gráfica para dos búsquedas parciales en p y q en un árbol de profundidad k

En la figura 4-13 se propone un ejemplo con un árbol de profundidad k=4 y sólo una profundidad de búsqueda en p=3. A la izquierda se encuentra la tabla de prefijos numerados desde cero hasta ocho. Se tienen dos contadores el primero de tamaño $(2^3 \times 3) = 24$ y el segundo contador es de $(2^4 \times 1)=16$ y poseen valores según las reglas antes establecidas. Se integran también todas las direcciones IP con el valor de los contadores evaluados, el índice resultante y el prefijo más largo que se puede corroborar con la tabla de ruteo contenida en la figura.

Tabla de ruteo

Índice	*Prefijo*
0	0000
1	0001
2	001
3	0100
4	0101
5	011
6	10
7	110
8	111



	0	1	2	3	4	5	6	7
Contador 1	0	2	3	5	6	6	7	8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contador 2	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Direcciones IP y conteos parciales

@IP	*Contador1 [@IP(3)]*	*Contador2 [@IP(4)]*	*Índice*	*Prefijo (LPM*)*
000,0	0	0	0	0000
000,1	0	1	1	0001
001,0	2	0	2	001
001,1	2	0	2	001
010,0	3	0	3	0100
010,1	3	1	4	0101
011,0	5	0	5	011
011,1	5	0	5	011
100,0	6	0	6	10
100,1	6	0	6	10
101,0	6	0	6	10
101,1	6	0	6	10
110,0	7	0	7	110
110,1	7	0	7	110
111,0	8	0	8	111
111,1	8	0	8	111

* LPM (longest prefix matching) notación de búsquedas del prefijo más largo

Figura 4-13 Ejemplo de una búsqueda partiendo el árbol de k=4 en p=3. El ejemplo contiene la tabla de ruteo, el árbol binario completo, el vector de bits y los contadores construidos a partir del vector de bits; se agregan también todas las direcciones IP destino de longitud 4 para corroborar el algoritmo

4.3.2 Construcción de las reglas para el contenido de arreglos contadores.

El algoritmo de búsqueda del prefijo más largo de una dirección IP hace uso de arreglos que llevan una cuenta parcial de bits puestos en *"1"* del vector de bits. El número de arreglos contadores depende del número de conteos parciales (cortes en el árbol) que se realicen en las búsquedas, por ejemplo, si se tienen dos cortes en el árbol se necesitan tres arreglos contadores de bits puestos en *"1"*. Cada contador tiene valores según las reglas mencionadas en la sección anterior; las reglas para dar valores a un arreglo contador de p elementos están basadas en la lógica de contar a manera de *offset* el número de bits puestos en *"1"* cada 2^p intervalos de direcciones IP; en cada conteo parcial se hace más preciso el conteo hasta completarlo con el último conteo parcial.

Se distinguieron 2 tipos de reglas: unas para lo que se nombró como la última profundidad de búsqueda (es una búsqueda directa) y reglas para hacer conteos parciales. Se definió la regla número 4 para búsquedas parciales como:

> *Se analiza si el primer elemento de los 2^{k-q} bits es igual a 1, si no es así el offset se detiene (el conteo se detiene) hasta encontrar un bloque de 2^{k-q} bits que contenga un 1 al inicio o bien se reinicia si ya han pasado 2^{k-p} bits. Esto porque la presencia de un 1 al inicio del bloque indica que existe un nodo en q.*

Ya que en la regla anterior podría prestarse a confusión, a continuación se explica a más detalle. Pensemos en el siguiente ejemplo (ver figura 4-14): se tienen direcciones IP de 4 bits de longitud y una serie de prefijos que definen un árbol binario completo como lo muestra la figura 4-14.a. El árbol se parte en la profundidad 3 por lo que se tienen dos conteos parciales de los cuales uno representa una búsqueda parcial y el otro una búsqueda directa. A la primera búsqueda parcial se le asigna un contador llamado "*contador 1*" y a la última búsqueda directa se le asigna un contador llamado "*contador 2*".

El *contador 2* del ejemplo corresponde a una búsqueda directa (en la profundidad 4) en los que se cuenta el número de bits puestos en *1* de los ocho subárboles independientes que se forman partiendo el árbol en la profundidad 3. En la figura 4-14.b se muestra dicho

contador, nótese que cada casilla es de 1 bit y se cuenta el número de prefijos a la izquierda de cada subárbol. El conteo siempre comienza con el valor de ***cero*** para hacer uso de todas las combinaciones de los bits utilizados para contar (es por ello que el número de prefijos se resta en 1). Para los subárboles que no existen en la profundidad 3 (subárboles 5 y 6) el valor del contador en esas casillas tienen un valor de ***0***.

Entonces; el *contador 2* de la figura 4-14.b completa el conteo de prefijos a la izquierda en los 8 sub-árboles de la partición en la profundidad 3. Sin embargo, se necesita un primer contador que determina el número ***"total"*** de prefijos a la izquierda considerando todos los subárboles. Como se tienen 8 sub-árboles se tiene un contador de 8 casillas y para contar el número de prefijos a la izquierda se necesitan 4 bits en cada casilla. En la figura 4-14.c se muestra un posible *contador* que cuenta el total de bits puestos en ***"1"*** del vector de bits a la izquierda en intervalos de 2 bits. Nótese que los subárboles 5 y 6 no existen y que el vector de bits para estos nodos tiene el valor "1 0 0 0" lo que significa que el primer ***1*** representa a cuatro direcciones IP; sin embargo, el conteo se lleva a cabo cada dos bits por lo que se tiene un error al contar los prefijos a la izquierda. Por lo tanto cuando no existen subárboles en la profundidad de búsqueda se detiene el conteo hasta el siguiente subárbol que si exista, esto se muestrea en la figura 4-14.c donde se pone el valor correcto de los conteos parciales nombrado como *contador 1*.

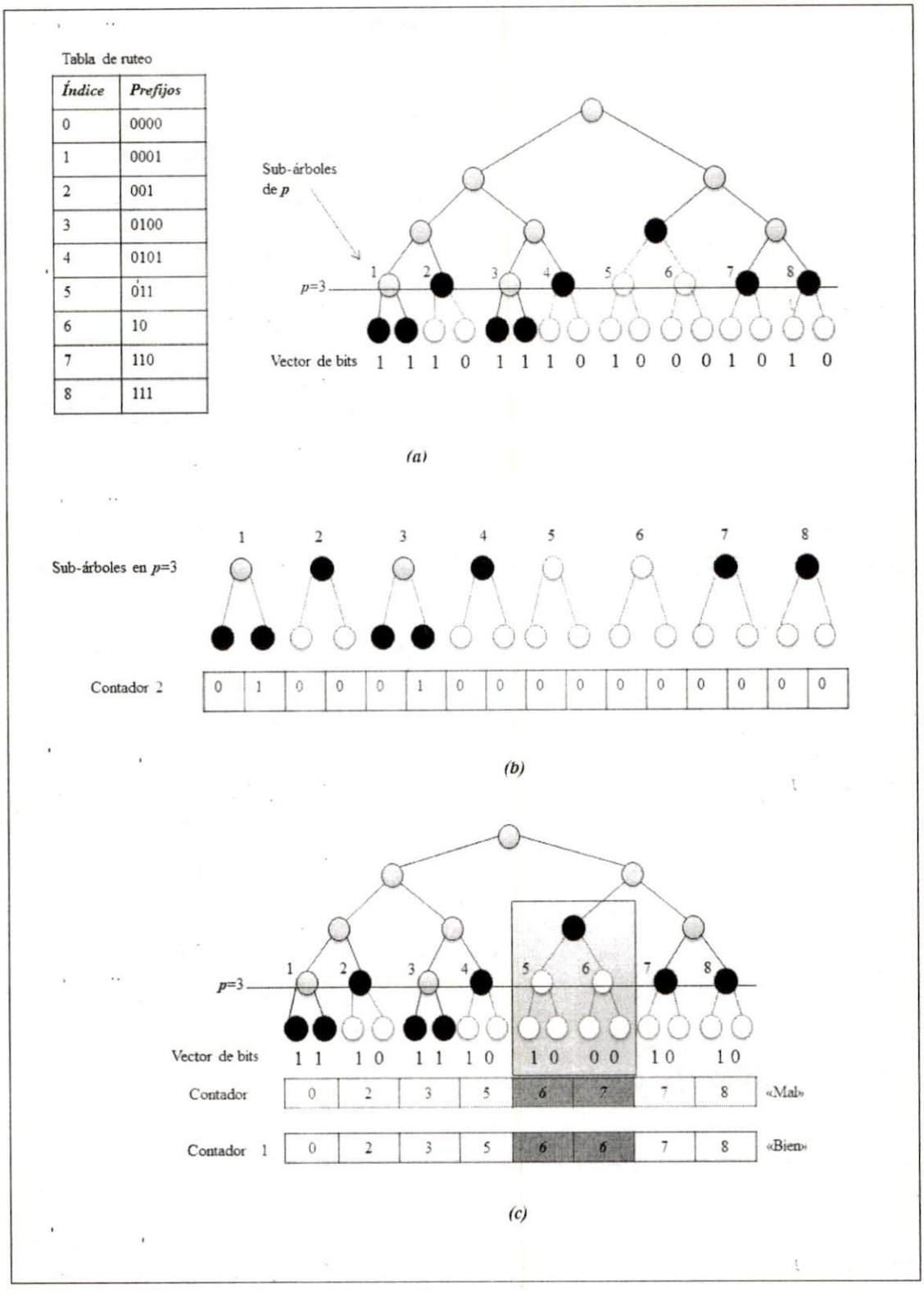


Figura 4-14 Ejemplo de construcción de los arreglos contadores

4.3.3 Función de costo en memoria

El costo en memoria por hacer n cortes en el árbol y hacer conteos parciales está dado por los arreglos contadores ya que tanto el árbol como el vector de bits son desechables una vez construidos los contadores. Se define a f_m como la función de costo en memoria en bits por hacer n cortes dentro de un árbol binario, f_m depende tanto del número n de cortes en el árbol como de las posiciones en que se efectúan dichos cortes. Sea r un conjunto donde $r = \{\alpha_0=0, \alpha_1, \alpha_2, \alpha_3, \dots, \alpha_i, \alpha_{i+1}, \dots, \alpha_{n-1}, \alpha_n, \alpha_{n+1}=k\}$, tal que los α_i son las posiciones de los cortes en el árbol, $\alpha_i \leq \alpha_{i+1}$, $\alpha_i \in \mathbb{N}$, k es la profundidad del árbol y $\alpha_{n+1}=k$; entonces r es un conjunto que incluye la profundidad 0 y la profundidad total del árbol k por lo tanto $n=|r|-2$.

La función de costo en memoria está definida como $f_m = f_m(r,n)$ dados sus grados de libertad y queda definida según la ecuación.

$$f_m(n,r) = \sum_{i=1}^{n+1} 2^{\alpha_i}(k - \alpha_{i-1}) \tag{4.3}$$

La función de costo en memoria es una sumatoria que para n cortes en el árbol se tiene el costo de $n+1$ arreglos contadores; cada término de la suma está compuesto por el número de elementos que debe tener el arreglo contador 2^{α_i} multiplicado por los bits necesarios para poder contar los bits "*1*" en el vector de bits de acuerdo a la profundidad del corte

4.3.4 Función de costo en instrucciones

El costo en instrucciones por efectuar búsquedas del prefijo más largo en n cortes del árbol de acuerdo a un vector r (establecidos en la sección anterior) está dado por dos tipos de instrucciones: de acceso a memoria o extracción de bits y por operaciones aritmético lógicas. En este trabajo son despreciables las instrucciones aritmético-lógicas ya que éstas se realizan regularmente en un ciclo de operación del procesador mientras que las de acceso

a memoria ocupan más ciclos de operación de acuerdo al tipo de memoria (i.e. RAM, *cache*, disco duro). El desarrollo de tecnología de acceso a memoria crece en menor proporción a la tecnología de brinda la frecuencia máxima de operación en un procesador por lo que en un sistema digital el "cuello de botella" no es la frecuencia de operación sino el tiempo que se tarda en acceder a memoria.

Se define a f_i como la función que determina el número de instrucciones realizadas por efectuar búsquedas de acuerdo a las variables n y r dada por la ecuación.

$$f_i(n) = n+1 \tag{4.4}$$

Nótese que f_i es función únicamente de n ya que las profundidades de búsqueda no repercuten en el costo, sin embargo el número de cortes en el árbol determina el $n+1$ de conteos parciales y por lo tanto el número de operaciones que se deben de realizar.

La función $f_i(n) = n+1$ determina la ecuación de una recta con pendiente 1 y ordenada al origen 1; no tiene mínimos pues es una función creciente. Minimizar el costo en instrucciones es posible de acuerdo al valor de n; mientras menos cortes se tengan en el árbol menos instrucciones de búsqueda se realizan. Para $n = 0$ se minimiza el número de instrucciones a $f(0)=1$ y se dice que sólo se efectúa la búsqueda directa definida en secciones anteriores. Sin embargo, al minimizar el número de instrucciones dadas por la ecuación (4.4) crece el espacio requerido en memoria dado por la ecuación (4.3), por lo tanto no es posible minimizar ambos parámetros al mismo tiempo.

4.3.5 Minimización de la función de costo en memoria

El objetivo de hacer conteos parciales en el vector de bits es reducir el tamaño en memoria requerido para una búsqueda directa, en donde el costo en memoria está dado por un solo arreglo contador de tamaño $2^k k$ que crece de forma exponencial de acuerdo a k. La función dada por la ecuación (4.3) muestra que al hacer búsquedas parciales se sigue teniendo un comportamiento exponencial, sin embargo en cada término de la suma es de menor tamaño que $2^k k$. Encontrar los valores de n y r (número de cortes en el árbol y posición de cada una de ellos) que determinen el mínimo de memoria requerida y construir los arreglos contadores es nuestro objetivo, pues aunque se tengan que hacer más operaciones de extracción de bits y operaciones aritméticas (despreciables), el costo puede ser manejable por un enrutador para un valor de k más grande.

Minimizar el costo en memoria se hace de acuerdo a f_m y a sus grados de libertad n y r. Tanto f_m como sus argumentos están definidos en los enteros positivos y el costo mínimo también debe de ser un entero positivo. La profundidad k del árbol no es una variable, es una constante que se encuentra en el intervalo [1,128] ya que en IPv6 se pueden tener prefijos de máxima longitud igual a 128; definiendo así 128 problemas diferentes de los cuales se tiene que encontrar el valor óptimo de n y r para la construcción de los arreglos contadores.

Para encontrar el mínimo costo asumimos que f_m está contenido en los reales para que sea una curva de clase C, continua y diferenciable, la cual es posible derivar y con ello encontrar sus puntos críticos, obteniendo así una vecindad de números enteros que contengan el mínimo para poder así determinar el número entero mínimo de la función (aproximación ya que en realidad $\propto_i$ tiene valores discretos).

Dado que f_m está dado por la ecuación (4.3).

$$f_m(n,r) = \sum_{i=1}^{n+1} 2^{\propto_i}(k - \propto_{i-1})$$

Desarrollando la sumatoria se tiene.

$$f_m(n,r) = 2^{\propto_1}(k-\propto_0) + 2^{\propto_2}(k-\propto_1) + 2^{\propto_3}(k-\propto_2) + \cdots + 2^{\propto_i}(k-\propto_{i-1}) + 2^{\propto_{i+1}}(k-\propto_i) + , \ldots , +2^k(k-\propto_n)$$

La función f_m depende de n variables $\propto_i$. Para determinar el mínimo de la función se hacen las derivadas parciales con respecto a cualquier $\propto_i$:

$$\frac{\partial}{\partial \propto_i}(f_m(n,r)) = \frac{\partial}{\partial \propto_i}\left(\sum_{i=1}^{n+1} 2^{\propto_i}(k-\propto_{i-1})\right)$$

$$= \frac{\partial}{\partial \propto_i}\left(2^{\propto_1}(k-\propto_0) + 2^{\propto_2}(k-\propto_1) + 2^{\propto_3}(k-\propto_2) + \cdots + 2^{\propto_i}(k-\propto_{i-1}) + 2^{\propto_{i+1}}(k-\propto_i) + , \ldots , +2^k(k-\propto_n)\right)$$

$$= \frac{\partial}{\partial \propto_i}\left(2^{\propto_i}(k-\propto_{i-1}) + 2^{\propto_{i+1}}(k-\propto_i)\right)$$

Recordamos que: $\frac{d}{dx}(c^x) = c^x ln(c) \quad \forall\, c > 0$

Entonces,

$$\frac{\partial f_m}{\partial \propto_i} = 2^{\propto_i} ln(2)(k-\propto_{i-1}) - 2^{\propto_{i+1}}$$

Igualando $\frac{\partial f_m}{\partial \propto_i} = 0$ para obtener los puntos críticos

$$2^{\propto_i} ln(2)(k-\propto_{i-1}) - 2^{\propto_{i+1}} = 0$$

Entonces $\propto_i$

$$\propto_i = \log_2 2^{\propto_{i+1}} - \log_2[ln(2)(k-\propto_{i-1})]$$

$$\propto_i = \propto_{i+1} \log_2 2 - \log_2(ln(2)) - \log_2((k-\propto_{i-1}))$$

Pero $\quad -\log_2(ln(2)) = 0.528766373 = y$

Al final del desarrollo se tiene la ecuación con variables continuas

$$\propto_i = \propto_{i+1} - \log_2((k-\propto_{i-1})) + y \qquad (4.5)$$

Lo que significa que la profundidad que minimiza la función sólo depende de las dos profundidades adyacentes a ella. Entonces $\propto_i$ es función de $\propto_{i-1}$ y de $\propto_{i+1}$.

El mínimo depende del número de búsquedas internas que se efectúen.

Si se tiene un corte en el árbol, la búsqueda debe realizarse a la profundidad $\propto_1$ tal que:

$\propto_{i+1} = k$ y $\propto_{i-1} = 0$

$\propto_1 = k - \log_2(k - \propto_0) + y$

Si se tienen dos cortes en el árbol, las profundidades de búsqueda son $\propto_1, \propto_2$ tal que:

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y$

$\propto_2 = k - \log_2(k - \propto_1) + y$

Si se tienen tres cortes en el árbol, las profundidades de búsqueda son $\propto_1, \propto_2, \propto_3$ tal que:

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y$

$\propto_2 = \propto_3 - \log_2(k - \propto_1) + y$

$\propto_3 = k - \log_2(k - \propto_2) + y$

Si se tienen cuatro cortes en el árbol, las profundidades de búsqueda son $\propto_1, \propto_2, \propto_3, \propto_4$ tal que:

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y$

$\propto_2 = \propto_3 - \log_2(k - \propto_1) + y$

$\propto_3 = \propto_4 - \log_2(k - \propto_2) + y$

$\propto_4 = k - \log_2(k - \propto_3) + y$

En general, si se tienen ***n*** cortes en el árbol se pueden tener ***n*** ecuaciones no lineales con ***n*** incógnitas sujetas a: $\propto_{i-i} < \propto_i$, $\propto_i \in \mathbb{N}$. Ya que ***n*** también es una variable se necesita resolver todos los sistemas de ecuaciones para todos los valores de ***n*** posibles; el mínimo o mínimos absolutos de la función es aquel que para un ***n*** y ***r*** la función tenga un punto mínimo.

Por ejemplo si deseamos efectuar búsquedas en un árbol de profundidad ***k*** =16 con dos búsquedas internas necesitamos resolver el siguiente sistema de ecuaciones:

$$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y \quad \ldots\ldots\ldots \quad (1)$$

$$\propto_2 = k - \log_2(k - \propto_1) + y \quad \ldots\ldots\ldots \quad (2)$$

Sustituyendo el valor de k, el valor $\propto_0 = 0$, y el valor de $\propto_2$ en $\propto_1$:

$$\propto_1 = 16 - \log_2(16 - \propto_1) - \log_2(16) + 2y$$

$$\propto_1 + \log_2(16 - \propto_1) = 16 - \log_2(16) + 2y$$

Ya que no se puede despejar $\propto_1$, se necesita dar solución al sistema numéricamente. Se puede ver que el resultado de $\propto_1$ es un valor real, sin embargo tenemos que redondearlo al entero que está por encima o por debajo de él y no tenemos ningún criterio de decisión. La precisión de $\propto_1$ no es tan importante pues sólo se tomará su valor entero redondeado. Para el ejemplo ya resuelto de forma analítica se tiene que $\propto_1 = 10.6$ y $\propto_2 = 14.09$ siendo los valores que minimizan a la función con un k=16 y n=2: $\propto_1 = 10$ y $\propto_2 = 14$ (en este ejemplo se propusieron dos cortes en el árbol y se encontraron los valores analíticos continuaos; después se corroboraron los valores discretos mediante el uso de software).

Otra forma de obtener el mínimo de la función es mediante el uso de un programa que dado un valor de k encuentre tanto el valor óptimo de n como el de r mediante la comprobación de todos los valores posibles de la función en memoria. El algoritmo de búsqueda recibe un valor de k (profundidad del árbol) y de forma iterativa encuentra el valor óptimo de r para una búsqueda parcial, luego se efectúa el mismo procedimiento para dos, tres, cuatro etc. búsquedas parciales obteniendo el costo mínimo entre estas.

Para n cortes en un árbol de profundidad k se tiene un determinado número de posibles valores de las componentes de r; entonces se debe encontrar que valor de r es el que minimiza la función de costo en memoria.

El pseudocódigo del software que calcula los valores de ***n*** y ***r*** es el siguiente:

*Para todos los valores de **k**:*

Comienza

Para 1 búsqueda parcial hasta k-1 búsquedas parciales:

Comienza

Calcula_n_r(k, n ,1 ,1) //1: es el inicio de la recursión y 1: es la profundidad donde //se coloca la primera profundidad de búsqueda

Termina

Termina

El procedimiento recursivo "*Calcula_n_r*" calcula los posibles valores de ***n*** y ***r*** desde 1 hasta ***k***-1 y guarda el costo mínimo en las variables ***n***, ***r***. Se pasan 4 parámetros: la profundidad ***k***, el número de búsquedas a acomodar en ***r***, la búsqueda a acomodar y su posición en el árbol.

Procedimiento Calcula_n_r(entero: profundidad, entero: num_busquedas, entero: búsqueda_actual, entero: posición_actual)

Comienza

Si posición_actual ≤ num_busquedas, entonces: //condición para terminar la recursión

Comienza

Para i=posición_actual, hasta profundidad - num_busquedas + búsqueda_actual

Comienza

Calcula_n_r(profundidad, num_busquedas, búsqueda_actual+1, i+1)

If(búsqueda_actual== num_busquedas)

Comienza

Costo <- función_costo(num_busquedas, r)

Termina

Termina

Termina

Termina

La función *"función_costo"* da como resultado el número de bits necesarios por efectuar búsquedas parciales de acuerdo a n y r y simplemente calcula el costo mediante la ecuación (4.2). El pseudocódigo es el siguiente:

```
Entero: función_costo(entero: n , arreglo entero: r)
Comienza
    Suma=0
    Para i=1 hasta i=n+1
    Comienza
      Suma= Suma + potencia (2, r[i])* (k-r[i-1])
    Termina
Regresa Suma
Termina
```

Utilizando el algoritmo de búsqueda de mínimos de la función de costo en memoria se encontró que para algunas profundidades se tiene más de una solución ya que el método analítico solo encuentra el mínimo que pertenece a los reales, sin embargo no analiza el comportamiento de la función en las vecindades de números enteros más próximos. Por ejemplo, se corrió el algoritmo en el intervalo de k = [1,32] encontrando que para algunos valores de k se tienen varios valores de r que minimizan el costo:

- $k = 4$
  r=[0, 2, 4]
  r=[0, 3, 4]
  r=[0, 1, 3, 4]
  r=[0, 2, 3, 4]

- $k = 8$
  r=[0, 3, 5, 7, 8]
  r=[0, 2, 5, 7, 8]

- $k = 16$
  r=[0, 4, 7, 10, 13, 15, 16]
  r=[0, 3, 7, 10, 13, 15, 16]

4.4 Algoritmo de optimización en el último conteo parcial o de búsqueda directa

Primero se definió una búsqueda directa basada en un arreglo contador, después se partió el árbol en n cortes de búsqueda con el objetivo de reducir el tamaño en memoria necesario para los arreglos contadores. En esta sección se hace una optimización en la última etapa de búsqueda donde se aprovecha que el vector de bits se obtiene a partir de prefijos que forman un árbol binario completo, y como se verá, existen grupos de bits que conforma al vector de bits que son válidos y que no todas las combinaciones de bits en estos grupos son posibles.

La optimización consistirá en reducir el tamaño de los contadores de bits "*1*" dados por la ecuación (4.3) la cual representa el costo de efectuar n cortes en el árbol en sus $n+1$ términos o arreglos contadores. El último término de la ecuación (4.3), es precisamente el que siempre tendrá 2^k elementos y el tamaño en bits de cada uno de ellos está dado por el último corte efectuado. Este término es una cota pues no existe ningún contador de mayor o igual tamaño en las demás profundidades de búsqueda. Si remplazamos este último término por una estructura de datos que conforme otro tipo de codificación del vector de bits y que no utilice demasiadas instrucciones de extracción de bits se tendrá una optimización pues se reducirá el costo en memoria y mantendremos un número reducido de instrucciones para la obtención del índice del prefijo más largo.

4.4.1 Combinaciones válidas en el vector de bits

Al inicio se partió de una tabla de ruteo con la cual se elaboró un árbol binario completo, se construyó una tabla de ruteo equivalente que se ordena de forma ascendente según el valor numérico de sus prefijos. A partir del árbol binario completo de profundidad k se construye el vector de bits de tamaño 2^k bits que es la representación plana de la posición relativa de los prefijos junto con el intervalo de bits que representa cada uno de ellos.

Pensemos que el vector de bits de tamaño 2^k bits está conformado por 2 vectores de bits de dos árboles de profundidad $2^k/2$, estos dos nuevos vectores se vuelven a partir en dos y así sucesivamente hasta tener 2^k vectores de un bit cada uno de ellos, y cada vector de un bit representará a un árbol de profundidad cero. Siguiendo esta lógica se puede deducir que un vector de bits de longitud 2^k es la combinación de vectores de bits de longitudes 2^x tal que $2^x \leq 2^k$ y que x es un entero positivo.

Dado que el vector de bits está conformado por otros vectores se desea conocer si un vector de bits de longitud 2^k puede poseer cualquier valor binario o bien puede poseer un valor específico y determinado por un árbol binario completo. En la figura 4-15.a se puede observar un árbol de profundidad cero el cual consta sólo de la raíz (salida por defecto) y su vector de bits es de tamaño $2^0 = 1$ por consiguiente, el vector de bits únicamente puede tener un solo valor y es *1*. Un árbol completo de profundidad 1 puede tener a sus dos hijos o a ninguno de ellos por lo que sólo se pueden tener dos árboles distintos de profundidad 1 y por lo tanto dos vectores de bits distintos de longitud $2^1=2$ con el valor de ***11*** y ***10*** respectivamente. Las combinaciones ***00*** y ***01*** no pueden existir pues el árbol es completo y por lo menos se tiene a la raíz como ancestro. En la figura 4-15.b se puede observar los dos árboles posibles de profundidad 1. En la figura 4-15.c se muestran los 5 posibles árboles de profundidad k=2 con sus respectivos vectores de bits de longitud $2^2=4$.

Se pueden construir todos los árboles posibles de profundidad k con sus posibles vectores de bits o bien, se puede notar de la figura 4-15 que los vectores de bits de longitud c son las combinaciones posibles de vectores de longitud $c/2$ más la combinación que contempla sólo a la raíz. Comenzando con el árbol de profundidad cero se tiene un vector de bits con el valor *1*, para deducir los posibles valores del vector de bits perteneciente a un árbol de profundidad 1, se combina el valor del vector anterior de longitud 1, como el único valor es *1* entonces se combina *1* y *1* obteniendo el valor ***11*** más la combinación que corresponde sólo a la raíz ***10***. Para vectores de longitud 4 (pertenecientes a un árbol de profundidad 2) se combinan los valores de los vectores de bits de longitud 2 (***11***, ***10***), obteniendo como resultado ***1010***, ***1110***, ***1111***, ***1011*** y la combinación que contempla sólo a la raíz ***1000*** (ver figura 4.15.c para corroborar). Siguiendo esta lógica ahora se pueden obtener los valores

posibles de longitud 8 combinando los valores de longitud 4; de esta forma se pueden tener todos los vectores de bits posibles sin tener que construir todos los árboles binarios.

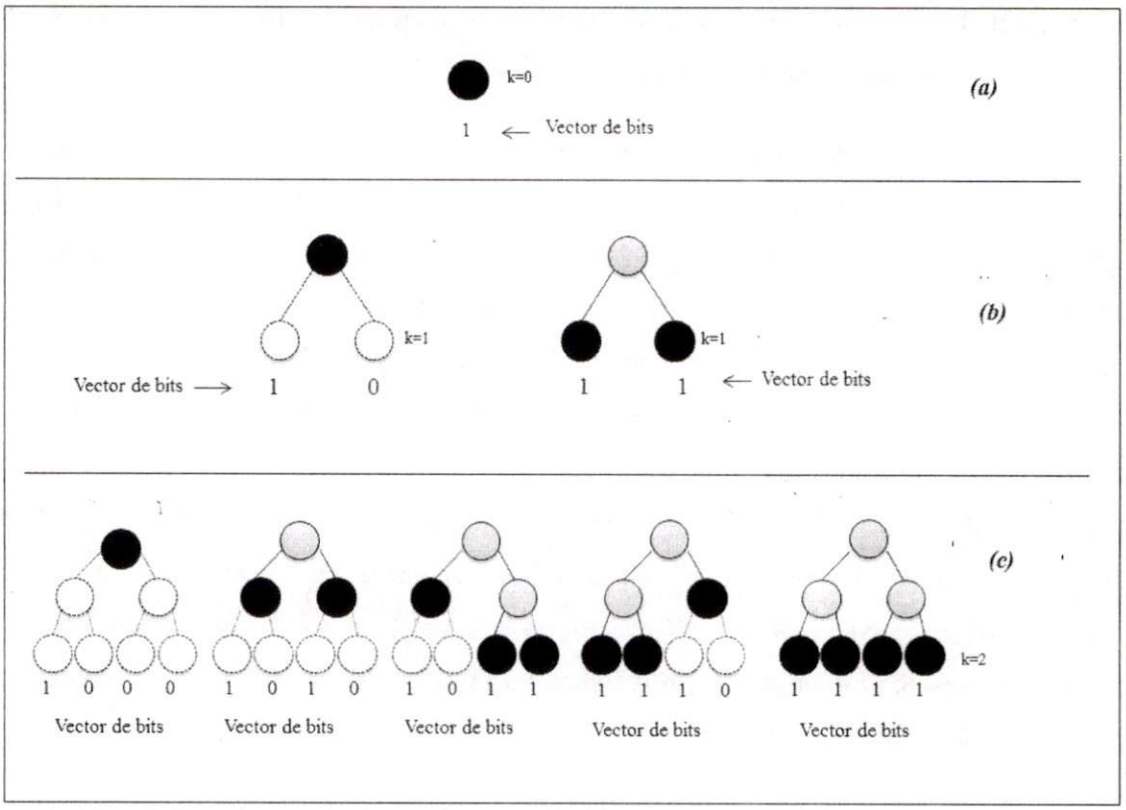


Figura 4-15 Vectores de bits posibles para arboles de profundidad cero, uno y dos

A partir de la deducción de los posibles valores de los vectores de bits para distintas profundidades, es claro comprender que en cualquier profundidad se tienen combinaciones de bits válidas en el vector de bits, por ejemplo para k=16 se tiene un vector de bits de tamaño 2^{16} bits y dentro de él existen combinaciones posibles de bits de vectores de profundidades menores a 16. Si se recorre el vector en intervalos de 2 bits se notará que sólo existen la combinaciones ***10***, ***11***, ***00***; si se recorre el vector en intervalos de 4 bits se tendrán las combinaciones ***1000***, ***1010***,***1011***,***1110***,***1111*** y ***0000***; así sucesivamente se tendrán las combinaciones válidas de longitud 8, 16, 32, 64, etc. Nótese que los grupos de bits válidos dentro de un vector de bits también incluyen la combinación donde todos sus elementos son ***0*** ya que existe una combinación donde se tiene un ***1*** y el resto son ***0***

(10...00000); si se divide este vector entre dos se tendrá una combinación válida donde todos sus elementos son *0*.

Un vector de bits se puede dividir en grupos de bits de $c = 2, 4, 8, 16, 32, 64, 128$....etc. Y para cada c se tiene un determinado número de combinaciones válidas en sus respectivos vectores de bits. Se define la ecuación recursiva $a(x)$ como la función que calcula el número de combinaciones válidas dada una longitud de c bits y $x = log_2 c$. La función recursiva $a(x)$ está dada por la ecuación que se obtuvo por inspección.

$$a(x) = 1 + (a(x-1))^2 \qquad a(0)=1 \tag{4.6}$$

4.4.2 Conteos parciales y apuntadores a combinaciones válidas en el vector de bits

En la sub secçión anterior se demostró que existe un número limitado de combinaciones o grupos válidos de bits de longitud c dentro del vector de bits y que el número de combinaciones válidas está dado por $a(x)$. Ahora bien ¿cómo se puede aprovechar esta cualidad del vector de bits para realizar búsquedas con estructuras de datos de menor tamaño en memoria? Pensemos que se hace un corte a un árbol en la profundidad p con lo cual se tienen que hacer dos conteos parciales (posteriormente se pueden agregar n búsquedas parciales); primero se selecciona la longitud de un grupo de bits que conforman alguna combinación posible en el vector de bits (por ejemplo un valor c), después se encuentra la profundidad $p(c)$ que abarca los c bits en el vector de bits $p(c)= k- log_2 c$ y se toma como una profundidad de búsqueda. Desarrollando la ecuación (4.3) para $n=1$ y $r = \{0, p, k\}$ se tiene $f(1, \{0,p,k\})=2^p(k)+2^k(k-p)$ lo que significa que se tienen dos contadores, uno situado en p y el otro en k (cada uno de ellos construido bajo sus propias reglas). La idea es quitar el término $2^k(k-p)$ (arreglo contador) ya que con él se completa la búsqueda en un intervalo de c bits válidos y sustituirlo por un contador de combinaciones válidas. En la figura 4-16 se muestra un árbol con su vector de bits el cual está seccionado en intervalos

de c bits y la profundidad de búsqueda $p(c)$ que representa a los c bits. Recuerde que c es una potencia de dos (c = 2, 4, 8, 16, 32, 64, 128...etc.) y es una combinación válida del vector de bits.

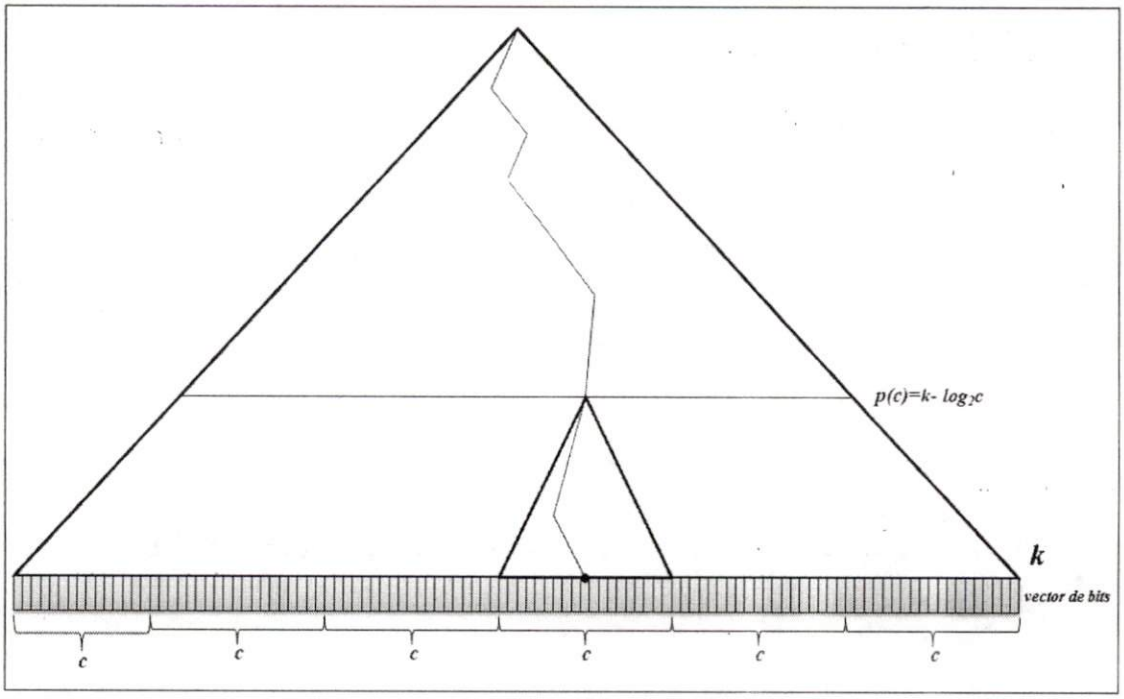


Figura 4-16 Combinaciones válidas en el vector de bits y selección de p de acuerdo a c

El contador de combinaciones válidas que sustituye al arreglo contador de $2^k(k-p)$ depende de la longitud c pues como se expresa en la ecuación (4.6) el número de combinaciones válidas es proporcional a c. Otra característica que debe tener el contador de combinaciones válidas es que su tamaño debe de ser menor que el del arreglo que sustituye de tamaño $2^k(k-p)$ si no, no habría ganancia alguna. La estructura de datos que deseamos obtener es un arreglo bidimensional que tenga tantos renglones como combinaciones válidas de c bits más uno, y c columnas de $log_2 c$ bits cada una de ellas.

Las reglas de construcción del arreglo contador de combinaciones válidas son las siguientes:

1. Cada renglón del arreglo corresponde a cada una de las combinaciones válidas:
2. Para cada combinación válida de longitud c se tienen c columnas de $log_2 c$ bits cada una de ellas que son un conteo de bits puestos en *1* en forma de *offset*.

3. Para cada renglón se recorre bit a bit, del bit más significativo al menos significativo la combinación válida contando el número de bits puestos en ***1*** que existan en la combinación valida y anotando el número de bits "***1***" que se tiene en cada casilla a manera de *offset* restando un 1 (si el número de unos es cero no se efectúa la resta).

Por ejemplo si se elige un valor $c = 4$ bits, se tienen $a(log_2 c)+1 = 5$ combinaciones válidas ***1000***, ***1010***, ***1011***,***1110***, ***1111*** más la combinación ***0000***. Entonces el contador tiene la forma de la tabla 4-1. El acceso a la tabla es mediante un índice y para fines didácticos se coloca la combinación válida en cada renglón. Pensemos en la combinación ***1011***, en el contador se lleva el conteo de bits puestos en ***1*** en la primera posición: 1 bit puesto en ***1***, en la segunda posición: 1 bit puesto en ***1***, en la tercera: 2 bits puestos en ***1*** y en la cuarta posición: 3 bits puestos en (recuerde efectuar la resta en 1).

Este arreglo bidimensional es independiente a cualquier tabla de prefijos y al mismo árbol junto con su vector de bits y su costo es una constante que sólo dependerá de la longitud de *c* bits que seleccionemos.

Tabla 4-1 Contador de combinaciones válidas para $c = 4$

índice	*Combinación*	*c1*	*c2*	*c3*	*c4*
0	0000	00	00	00	00
1	1000	00	00	00	00
2	1010	00	00	01	01
3	1011	00	00	01	10
4	1110	00	01	10	10
5	1111	00	01	10	11

Una vez teniendo el contador de combinaciones válidas se elimina el contador correspondiente al último término de la ecuación (4.3) de tamaño $2^k(k-p)$ y en el penúltimo contador se agrega un apuntador a una combinación válida $2^p(k+|ap|)$ de esta forma el costo total para una sola búsqueda parcial será el costo del primer contador con su respectivo apuntador más el costo del arreglo bidimensional de combinaciones válidas. El costo es: $2^p(k+|ap(c)|)+|matriz(c)|$.

Para hacer un poco más claro este algoritmo se presenta una búsqueda con un corte en el árbol en profundidad p que depende de c; sin embargo, se pueden efectuar tantos conteos parciales como $p-1$ niveles haya en el árbol, y el resultado de las búsquedas será la suma de los conteos parciales independientes a c más el conteo de la última profundidad de búsqueda que depende de c.

El algoritmo de búsqueda para conteos parciales que dependen de c se muestra de forma gráfica en la figura 4-17. La figura representa un sólo corte en el árbol para hacer entendible la optimización pero se pueden tener tantos cortes como niveles en el árbol. Primero se toman los p bits más significativos de la dirección IP de (longitud k) y se usa ese valor numérico para acceder al contador. El contador es de $k+|ap(c)|$ bits, los k bits son un resultado de un primer conteo parcial, $ap(c)$ es un apuntador a una combinación válida; el resto de los $k-p$ bits menos significativos de la dirección IP apuntan a la columna del arreglo contador denotado como $matriz(c)$, el segundo conteo parcial se obtiene entre el cruce del renglón y columna correspondiente al arreglo $matriz(c)$. La suma de los dos resultados de los conteos parciales es el índice del prefijo más largo en la tabla de ruteo.

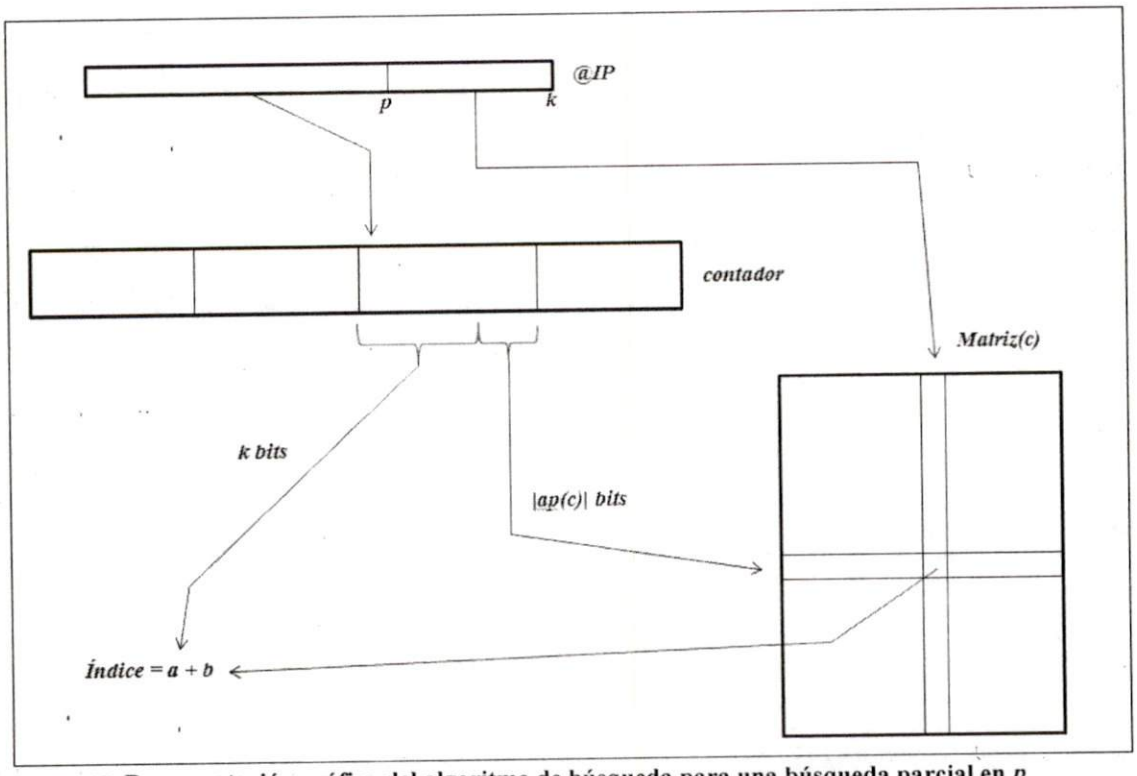


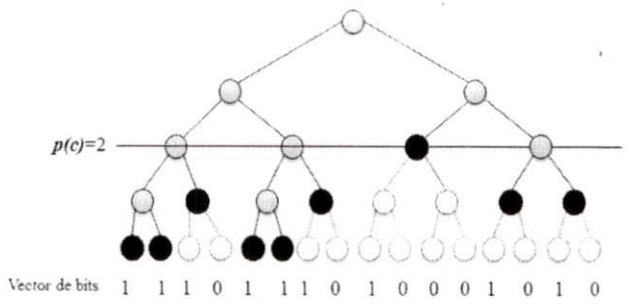
Figura 4-17 Representación gráfica del algoritmo de búsqueda para una búsqueda parcial en *p*

En la figura 4-18 se propone un ejemplo con un árbol de profundidad k=4 y un solo corte en el árbol en $p(c)$=2 por lo tanto se tienen combinaciones válidas de c = 4 bits. A la izquierda se encuentra la tabla de ruteo numerados desde cero hasta ocho. Se tiene el árbol binario con su respectivo vector de bits; se tiene también el arreglo contador que efectúa un conteo parcial y que apunta a una combinación válida; nótese que cada nodo en la profundidad 2 en el árbol cubre a cuatro elementos del vector de bits y el valor de estos elementos es la combinación valida. El contador combinaciones válidas llamado matriz se incluye junto con las combinaciones válidas asociadas a cada renglón.

En el ejemplo se integran todas las direcciones IP con el valor de los contadores evaluados, el índice resultante y el prefijo más largo que se puede corroborar con la tabla de prefijos contenida en la figura.

Tabla de ruteo

Índice	*Prefijo*
0	0000
1	0001
2	001
3	0100
4	0101
5	011
6	10
7	110
8	111



Contador

0		1		2		3	
0000	100	0011	100	0110	001	0111	010

* **

Matriz de combinaciones válidas c=4

	0	1	2	3	
0	00	00	00	00	0000
1	00	00	00	00	1000
2	00	00	01	01	1010
3	00	00	01	10	1011
4	00	01	10	10	1110
5	00	01	10	11	1111

Combinaciones válidas

@IP	*contador[@IP(2)]*	*matriz[contador[@IP(2)]][@IP(4)]*	*Suma*	*Prefijo más largo*
00 00	0	0	0	0000
00 01	0	1	1	0001
00 10	0	2	2	001
00 11	0	2	2	001
01 00	3	0	3	0100
01 01	3	1	4	0101
01 10	3	2	5	011
01 11	3	2	5	011
10 00	6	0	6	10
10 01	6	0	6	10
10 10	6	0	6	10
10 11	6	0	6	10
11 00	7	0	7	110
11 01	7	0	7	110
11 10	7	2	8	111
11 11	7	2	8	111

* Primer conteo parcial de bits puestos en *1* en el vector de bits.

** Apuntador a uno de los renglones de la matriz de combinaciones válidas.

Figura 4-18 Ejemplo de una búsqueda partiendo el árbol de k=4 en p=2 y c=4. El ejemplo contiene la tabla de prefijos, el árbol binario completo, el vector de bits y los contadores construidos a partir del vector de bits; se agregan también todas las direcciones IP destino de longitud 4 para corroborar el algoritmo

4.4.3 Función de costo en memoria

Hemos observado que se puede efectuar un conteo parcial en un árbol binario en una profundidad $p(c)$ que depende de la longitud c de los grupos de bits en que se divide el vector de bits. Sin embargo, esto no impide realizar más de un corte en el árbol con profundidades inferiores a p por lo que se pueden combinar conteos parciales independientes a c más un conteo parcial en $p(c)$ en el que se agregue un apuntador a una combinación válida.

Se define a f_m como la función de costo en bits por realizar conteos parciales más un último conteo basado en combinaciones válidas del vector de bits. La función f_m tiene tres grados de libertad, el primero es el número de conteos parciales en el árbol, el segundo es la posición de cada corte y por último la longitud c de los grupos de bits con combinaciones válidas. Se define a n como el número de cortes en el árbol para efectuar conteos parciales y un conjunto $r = \{\alpha_0=0, \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_{n-1}, \alpha_n, \alpha_{n+1}=k\}$ tal que los α_i son las profundidades de búsqueda en el árbol, $\alpha_i \leq \alpha_{i+1}$, $\alpha_i \in \mathbb{N}$. Entonces r es un vector que indica las profundidades de los cortes en el árbol, la profundidad 0 y la profundidad total del árbol k por lo tanto $n=|r|-2$. Nótese que r incluye todas las profundidades de los cortes independientes o no de c.

La función de costo en memoria estará definida como $f_m=f_m(r,n,c)$ dados sus grados de libertad y queda definida según la ecuación.

$$f_m(n,r,c) = 2^{\propto_n}(k - \propto_{n-1} + |ap(c)|) + |matriz(c)| + \sum_{i=1}^{n-1} 2^{\propto_i}(k - \propto_{i-1}) \tag{4.7}$$

Se puede observar que la ecuación (4.7) está compuesta por $n-1$ términos independientes a c, mismos que pertenecen a los contadores de los conteos parciales en el árbol, se tiene también un último contador que completa los n cortes en el árbol pero que posee en cada elemento un apuntador a una combinación válida. También se incluye el tamaño del arreglo bidimensional de combinaciones válidas llamado matriz y que depende de c.

La norma $|ap(c)|$ del apuntador es el número de bits necesarios para apuntar a una combinación válida y es función de c, está definida en la ecuación.

$$|ap(c)| = \lceil \log_2(1 + a(\log_2 c)) \rceil \tag{4.8}$$

La norma del arreglo contador de combinaciones posibles $|matriz(c)|$ es el número de bits necesarios para definir los $a(\log_2 c)$ renglones (combinaciones válidas) por c columnas de $\log_2 c$ bits cada una de ellas, y se define por la ecuación.

$$|matriz(c)| = (c)(\log_2 c)(1 + a(\log_2 c)) \tag{4.9}$$

4.4.4 Función de costo en instrucciones

El costo en instrucciones por hacer n cortes en el árbol binario de acuerdo a un vector r y un grupo de bits de longitud c que determinan una combinación válida (establecidos en la sección anterior) está dado por dos tipos de instrucciones: de acceso a memoria y por operaciones aritmético lógicas. En este trabajo son despreciables las instrucciones aritmético-lógicas ya que estas se realizan regularmente en un ciclo de operación del procesador mientras que las de acceso a memoria ocupan más ciclos de operación de acuerdo al tipo de memoria de acceso (i.e. RAM, *cache*, disco duro). El desarrollo de tecnología de acceso a memoria crece en menor proporción a la tecnología de brinda la frecuencia máxima de operación en un procesador por lo que en un sistema digital el "cuello de botella" no es la frecuencia de operación sino el tiempo que se tarda en acceder a memoria.

Se define a f_i como la función que determina el número total de instrucciones realizadas por efectuar búsquedas de acuerdo a las variables n, r y c dadas por la ecuación (4.10). Nótese que f_i es función únicamente de n ya que las profundidades de los cortes (dependiente e independiente a c) dadas por r no repercuten en el costo, sin embargo el número de particiones en el árbol determina el número de conteos parciales y por lo tanto el número de operaciones que se deben de realizar. La función $f_i(n)$ está dada por.

$$f_i(n) = n+1 \tag{4.10}$$

La función $f_i(n)$ depende de las por $n+1$ operaciones de acceso a memoria (extracción del contenido de los n contadores más el contenido del contador bidimensional de combinaciones válidas. La función $f_i(n) = n+1$ determina la ecuación de una recta con pendiente 1 y ordenada al origen 1; no tiene mínimos pues es una función monótona creciente. Minimizar el costo en instrucciones es posible de acuerdo al valor de n; mientras menos particiones se tengan en el árbol menos instrucciones de búsqueda se realizan; para $n = 0$ se minimiza el número de instrucciones a $f_i(0)=1$ y se dice que sólo se efectúa la búsqueda directa definida en secciones anteriores. Sin embargo, al minimizar el número de instrucciones dadas por la ecuación (4.10) crece el espacio requerido en memoria dado por la ecuación (4.7). Por lo tanto, no es posible minimizar ambos parámetros al mismo tiempo. Al no contar con la posibilidad de minimizar tanto el costo en memoria como el costo en instrucciones, es responsabilidad del diseñador qué paramentos utilizar para balancear ambos costos de acuerdo a sus necesidades y características.

4.4.5 Minimización de la función de costo en memoria

El objetivo de diseñar un nuevo algoritmo de búsqueda del prefijo más largo al cual le es asociada una nueva función de costo en memoria en bits es reducir el tamaño requerido para los contadores, tanto de conteos parciales como de combinaciones válidas. En un principio se utilizó un contador de tamaño $2^k k$, después se redujo el tamaño realizando conteos parciales, ahora, se reduce aún más ya que se elimina el último término en la sumatoria que es el que tiene mayor número de elementos. Encontrar el mínimo de la función permitirá encontrar los valores adecuados de los parámetros c, n, r para distintas profundidades de búsqueda y así poder ofrecer parámetros que sean viables para tablas de ruteo que correspondan a versiones de direccionamiento IPv4 o IPv6, capacidad de procesamiento y memoria de los enrutadores.

Para minimizar el costo en memoria se debe tomar en cuenta a f_m y a sus grados de libertad n, r y c. Tanto f_m como sus variables están definidas en los enteros positivos y el costo mínimo también debe de ser un entero positivo. La profundidad del árbol k no es una variable, es una constante que se encuentra en el intervalo [1,128]. La cota superior es 128 pues los prefijos en IPv6 tiene como longitud máxima 128 y define 128 problemas diferentes de los cuales se tiene que encontrar el valor mínimo de la función para construir los arreglos contadores.

Para encontrar el mínimo costo asumimos que f_m está contenido en los reales para que sea una curva de clase C, continua y diferenciable, la cual es posible derivar y con ello encontrar sus puntos críticos, obteniendo así una vecindad de números enteros que contengan el mínimo de la función para poder así determinar cuál entero es mínimo de la función que está contenida en los enteros positivos.

Dado que el costo f_m está dado por la ecuación (4.7), el costo del apuntador y matriz están dados por (4.8) y (4.9) respectivamente:

$$f_m(n,r,c) = 2^{\propto_n}(k - \propto_{n-1} + |ap(c)|) + |matriz(c)| + \sum_{i=1}^{n-1} 2^{\propto_i}(k - \propto_{i-1})$$

$$|ap(c)| = \lceil \log_2(1 + a(\log_2 c)) \rceil$$

$$|matriz(c)| = (c)(\log_2 c)(1 + a(\log_2 c))$$

$$a(x) = 1 + a(x-1)^2 \ , \qquad a(0) = 1 \ y \ x \in \mathbb{N}$$

Se puede minimizar a f_m con respecto a c de la siguiente forma: utilizando las ecuaciones (4.8) y (4.9) se determinó el valor de la matriz y del apuntador para distintos valores de c obteniendo los resultados de la tabla 4-2 que a continuación se presenta.

Tabla 4-2 Valores de apuntadores y matrices en bits

valor de c	*|matriz (c)| en bits*	*|ap(c)| en bits*
2	6	2
4	48	3
8	648	5
16	43392	10
32	73332960	19
64	8.06655E+13	38
128	3.95386E+25	76
256	3.98801E+48	151
512	1.74729E+94	301
1024	1.4723E+185	602

Pensando que en una búsqueda directa el costo es de $\boldsymbol{k\,2^k}$ bits y para fines de aplicación $\boldsymbol{k} \leq 128$, si $\boldsymbol{k}$ =128 el costo total de una búsqueda directa es $2^{128}128$= 4.35561E+40 bits. Entonces, $\boldsymbol{c}$ debe ser menor o igual que 128 ya que para valores mayores, por ejemplo 256, el costo de la matriz (3.98E+48 bits) es mayor que el costo de la búsqueda directa (4.3E+40 bits). Por lo tanto, los valores admisibles de $\boldsymbol{c}$ son: $\boldsymbol{c = 2,4,8,16,32,64,128}$. Así, que $\boldsymbol{c}$ ya no es una variable, ahora basta dar solución al problema para los posibles valores de $\boldsymbol{c}$ y tomar el valor mínimo como resultado final.

Para determinar las profundidades de búsqueda $\propto_i$ que minimizan la función se efectúa la derivada parcial de $\boldsymbol{f_m}$ con respecto de $\propto_i$

$$\frac{\partial}{\partial \propto_i} f_m(n,r,c) = \frac{\partial}{\partial \propto_i}\left(\sum_{i=1}^{n-1} 2^{\propto_i}(k-\propto_{i-1}) + 2^{\propto_n}(k-\propto_{n-1}+|ap(c)|) + |matriz(c)|\right)$$

$$= \frac{\partial}{\partial \propto_i}\left(2^{\propto_1}(k-\propto_0) + 2^{\propto_2}(k-\propto_1) + 2^{\propto_3}(k-\propto_2)+,\ldots,+2^{\propto_i}(k-\propto_{i-1})\right.$$
$$\left. + 2^{\propto_{i+1}}(k-\propto_i)+,\ldots,+2^{k-\log_2 c}(k-\propto_{n-1}+|ap(c)|) + |matriz(c)|\right)$$

$$= \frac{\partial}{\partial \propto_i}\left(2^{\propto_i}(k-\propto_{i-1}) + 2^{\propto_{i+1}}(k-\propto_i)\right)$$

Siguiendo un procedimiento similar al de la sección anterior; la derivada se iguala a cero y se obtiene los puntos críticos $\propto_i$.

Como resultado se tiene la ecuación (4.11) que es igual a la (4.5), ambas son independientes de ***c***.

$$\propto_i = \propto_{i+1} - \log_2\big((k - \propto_{i-1})\big) + y \tag{4.11}$$

El mínimo depende del número de cortes en el árbol que se efectúen y depende del valor de ***c***.

Si se tiene un corte en el árbol, la búsqueda debe realizarse a la profundidad:

$\propto_1 = k - \log_2 c$

Si se tienen dos cortes en el árbol, las profundidades de búsqueda son: $\propto_1, \propto_2$

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y$

$\propto_2 = k - \log_2 c$

Si se tienen tres cortes en el árbol, las profundidades de búsqueda son: $\propto_1, \propto_2, \propto_3$

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y$

$\propto_2 = \propto_3 - \log_2(k - \propto_1) + y$

$\propto_3 = k - \log_2 c$

Si se tienen cortes en el árbol, las profundidades de búsqueda son: $\propto_1, \propto_2, \propto_3, \propto_4$

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + y$

$\propto_2 = \propto_3 - \log_2(k - \propto_1) + y$

$\propto_3 = \propto_4 - \log_2(k - \propto_2) + y$

$\propto_4 = k - \log_2 c$

En general, si se tienen ***n*** cortes en el árbol se pueden tener ***n*** ecuaciones no lineales con ***n-1*** incógnitas sujetas a: $\propto_{i-i} < \propto_i$, $\propto_i \in \mathbb{N}$ y al valor de ***c=2, 4, 8, 16, 32, 64, 128***. Los mínimos absolutos están dados por el mínimo obtenido de resolver los sistemas de ecuaciones para todos los valores de ***n*** posibles.

Por ejemplo si deseamos efectuar búsquedas en un árbol de profundidad ***k***=16 con dos cortes, necesitamos resolver el siguiente sistema de ecuaciones:

$\propto_1 = \propto_2 - \log_2(k - \propto_0) + cy$ ..... (1)

$\propto_2 = k - \log_2 c$ .....(2)

Sustituyendo el valor de k, el valor $\propto_0 = 0$, y el valor de $\propto_2$ en $\propto_1$:

$\propto_1 = 16 - \log_2 c - \log_2(16) + y$

El valor de c se limitó al conjunto de valores: $c = \{2, 4, 8, 16, 32, 65, 128\}$. Entonces se resuelve el sistema para $\propto_1$ con los posibles valores de c y mediante la ecuación (4.7) con los valores de n=2 y $r = \{0, \propto_1, \propto_2, 16\}$ eligiendo $\propto_1, \propto_2$ que determinen un menor costo en bits. Para este caso se tiene un valor de $c = 8$, $\propto_1 = 8.943, \propto_2 = 12$; el valor entero óptimo encontrado con ayuda de software para $\propto_1$ es 9.

Otra forma de obtener el mínimo absoluto de la función para cualquier número de cortes en el árbol, es mediante el uso de un programa que compruebe todos los posibles valores de n, r y c y dado un valor de k encuentren los valores que minimicen a n, c y r. El algoritmo de búsqueda recibe un valor de k (profundidad del árbol) y de forma iterativa encuentra el valor óptimo de r para un corte en el árbol, luego se efectúa el mismo procedimiento para dos, tres, cuatro etc. Posteriormente se encuentra el costo mínimo entre los valores obtenidos.

Para n cortes en un árbol de profundidad k se tiene un determinado número de posibles valores de las componentes de r; entonces se debe encontrar que valor de r es el que minimiza la función de costo en memoria.

El pseudocódigo del software que calcula los valores de ***n*** y ***r*** es el siguiente:

Para todos los valores de ***k*** y de ***c*** *:*

Comienza

Para 1 búsqueda parcial hasta k-log$_2$c búsquedas parciales:

Comienza

Calcula_n_r(k, n ,1 ,1) //1: es el inicio de la recursión y 1: es la profundidad donde //se coloca la primera profundidad de búsqueda

Termina

Termina

El procedimiento recursivo "*Calcula_n_r*" calcula los posibles valores de ***n*** y ***r*** desde 1 hasta ***k***- $\log_2$ ***c*** y guarda el costo mínimo en las variables ***n***, ***r***. Se pasan cuatro parámetros: la profundidad ***k***, el número de búsquedas a acomodar en ***r***, la búsqueda a acomodar y su posición en el árbol.

Procedimiento Calcula_n_r(entero: profundidad, entero: num_busquedas, entero: búsqueda_actual, entero: posición_actual)

Comienza

Si posición_actual ≤ num_busquedas, entonces: //condición para terminar la recursión

Comienza

Para i=posición_actual, hasta profundidad - num_busquedas + búsqueda_actual

Comienza

Calcula_n_r(profundidad, num_busquedas, búsqueda_actual+1, i+1)

If(búsqueda_actual== num_busquedas)

Comienza

Costo <- función_costo(num_busquedas, r)

Termina

Termina

Termina

Termina

La función "*función_costo*" da como resultado el número de bits necesarios por efectuar búsquedas parciales de acuerdo a ***n*** y ***r*** y simplemente calcula el costo mediante la ecuación (4.5). El pseudocódigo es el siguiente:

```
Entero: función_costo(entero: n , arreglo entero: r)
Comienza
    Suma=0
    Para i=1 hasta i=n
    Comienza
      Suma= Suma + potencia (2, r[i])* (k-r[i-1])
    Termina
Suma = Suma+ |matriz(c)|
Suma = Suma + potencia (2, r[n+1])* (k-r[n]+ |ap|)
Regresa Suma
Termina
```

Utilizando el algoritmo de búsqueda que minimiza a la función en memoria se encontró que para algunas profundidades se tiene más de una solución. El método analítico solo encuentra el mínimo que pertenece a los reales; sin embargo, no analiza el comportamiento de la función en las vecindades de números enteros más próximos. Por ejemplo, se corrió el algoritmo para todos los valores de k que minimizan el costo en el intervalo de k = [3,32] encontrando que para algunos valores de k se tienen varios valores de r que minimiza el costo:

- k=16

  r=[0, 3, 6, 8, 16]
  r=[0, 4, 7, 10, 13, 16]

- k=8

  r=[0, 3, 6, 8]
  r=[0, 4, 6, 8]
  r=[0, 1, 4, 6, 8]
  r=[0, 2, 4, 6, 8]

- k=4

  r=[0, 2, 3, 4]
  r=[0, 1, 3, 4]

4.5 Análisis de costo en memoria e instrucciones para los distintos algoritmos de búsqueda

El análisis comenzó con un algoritmo de búsqueda del prefijo más largo que optimiza el número de instrucciones; pero que ocupa una cantidad de memoria que crece de forma exponencial de acuerdo a la longitud máxima de los prefijos; después se propuso una algoritmo de búsqueda que atenuara este comportamiento exponencial mediante conteos parciales en el vector de bits; este algoritmo sirvió como intermediario para entender la propuesta que se efectúa tomando en cuenta combinaciones válidas en el vector de bits estableciendo el algoritmo que integra búsquedas parciales con una optimización de la última búsqueda mediante combinaciones permitidas en el vector de bits.

Ahora, en esta sección se presenta un análisis de costo en memoria e instrucciones de todas los algoritmos de búsqueda del prefijo más largo destacando sus cualidades y comparándolas para encontrar las opciones de parámetros adecuados según cada dispositivo de encaminamiento o enrutador como son la capacidad de procesamiento, tamaño de la memoria principal y longitud máxima de los prefijos en la tabla que contiene.

4.5.1 Solución de búsqueda directa

Con este algoritmo el costo en instrucciones cumple nuestro primer objetivo pues sólo ocupa una instrucción de acceso a memoria para cualquier profundidad de búsqueda y para cualquier longitud máxima de prefijos en la tabla de ruteo, esto se demostró con la ecuación (4.2). Sin embargo, el costo en memoria está dado por la función $f_m = k2^k$ donde k es la longitud de la dirección IP. En la tabla 4-3 se muestra el costo en memoria en MB (Megabytes) y el costo en instrucciones por realizar una búsqueda básica en un intervalo de k = [16,32]; nótese que para k=19 el costo en memoria es 1.1875MB que es el tamaño de la memoria *cache* convencional de cualquier computadora comercial de bajo poder de procesamiento. Para k=32, que cubre todos los bits de IPv4, se necesitan 16.33GB (Gigabytes) que por lo regular sólo se puede tener en memoria RAM de computadoras con un alto poder de procesamiento.

Tabla 4-3 Costo en memoria e instrucciones para k en el intervalo [16,32]. El costo en MB está dado por $f_{MB}=(2^k k)$

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

4.5.2Solución mediante conteos parciales

A la solución directa se integraron conteos parciales para reducir el tamaño requerido en memoria del único arreglo contador definido por ella; encontrando que se pueden efectuar ***n*** cortes en el árbol pare realizar conteos parciales donde la profundidad de cada una de ellos está dado por ***n*** ecuaciones no lineales con ***n*** incógnitas. El número de operaciones por efectuar ***n*** cortes en el árbol está dado por la ecuación (4.4). En la tabla 4-4 se muestran la solución que minimiza en memoria el costo de realizar este tipo de búsquedas con los valores de ***n*** y ***r*** para distintos ***k*** en el intervalo [16,32] junto con su respectivo costo en memoria en MB y número de operaciones de acceso a memoria. Las operaciones aritméticas no se incluyen pues como se menciona con anterioridad los accesos a memoria son los cuellos de botella en el procesamiento de la información.

Tabla 4-4 Costo en memoria e instrucciones para k en el intervalo [16,32]. El costo está dado por la ecuación 3.2. bajo los parámetros de n, r, c.

k	*n*	*r*	*Costo MB*	*Accesos a memoria*
16	5	r=[0 3 7 10 13 15 16]	0.026702881	6
17	5	r=[0 4 8 11 14 16 17]	0.053407669	6
18	5	r=[0 5 9 12 15 17 18]	0.106819153	6
19	6	r=[0 2 6 10 13 16 18 19]	0.213639736	7
20	6	r=[0 3 7 11 14 17 19 20]	0.427280426	7
21	6	r=[0 4 8 12 15 18 20 21]	0.854562759	7
22	6	r=[0 5 9 13 16 19 21 22]	1.709129333	7
23	7	r=[0 2 6 10 14 17 20 22 23]	3.418262005	8
24	7	r=[0 3 7 11 15 18 21 23 24]	6.836524963	8
25	7	r=[0 4 8 12 16 19 22 24 25]	13.67305183	8
26	7	r=[0 5 9 13 17 20 23 25 26]	27.34610748	8
27	8	r=[0 2 6 10 14 18 21 24 26 27]	54.69222021	9
28	8	r=[0 3 7 11 15 19 22 25 27 28]	109.3844414	9
29	8	r=[0 4 8 12 16 20 23 26 28 29]	218.7688847	9
30	8	r=[0 5 9 13 17 21 24 27 29 30]	437.5377731	9
31	9	r=[0 2 6 10 14 18 22 25 28 30 31]	875.0755534	10
32	9	r=[0 2 7 11 15 19 23 26 29 31 32]	1750.151108	10

La técnica de efectuar conteos parciales en el vector de bits para reducir el tamaño de los contadores demuestra un grado de codificación más eficiente en memoria que el de una búsqueda directa pues el espacio requerido en bits es menor. Por ejemplo, se estableció un k=19 como la profundidad adecuada para realizar búsquedas dentro de una memoria *cache* de una computadora convencional bajo el esquema de búsqueda directa; en cambio, efectuando búsquedas parciales se puede establecer k=22 (se piensa que una computadora convencional pose entre 1 y 2 MB de memoria *cache*). Para k=32 se requieren estructuras de tamaño 1.7GB que en contraste con la búsqueda directa se necesitan 16.33GB por lo que para k=32 se redujo el tamaño a 1/19 del tamaño original.

Al reducir el costo en memoria se puede observar que existe un aumento tanto en el número de instrucciones aritméticas como de extracción de bits; sin embargo, el número de instrucciones aumenta en forma lineal en contraste con el costo en memoria que aumenta de forma exponencial. Elegir cual es el grado de codificación dependerá para una tabla de ruteo, de las longitudes de los prefijos que la componen y por supuesto de la capacidad física de un enrutador, pues a mayor frecuencia de operación y número de núcleos se pueden efectuar más rápido las operaciones aritméticas y mientras mayor sea la capacidad

de memoria *cache* o RAM se pueden contener estructuras más grandes en memoria principal.

4.5.3 Solución mediante conteos parciales más apuntadores a combinaciones válidas

El costo por realizar conteos parciales en el vector de bits dio como resultado una notable disminución del tamaño en memoria requerido para las estructuras de datos utilizadas, sin embargo el análisis también incluye una optimización en la última profundidad de búsqueda para la que se agregó una nueva variable a la función de costo y se encontró el valor adecuado de desempeño de todos sus parámetros. La función f_m dada por (4.7) puede tener n búsquedas parciales dadas por r de las cuales $n-1$ profundidades de búsqueda son independientes de una combinación de bits válidos en el vector de bits y la última profundidad de búsqueda está dada por el valor óptimo de la longitud de la combinación válida c.

En la tabla 4-5 se muestran la solución para distintas profundidades de búsqueda k en el intervalo [16,32], en las columnas de la tabla se encuentran el número de operaciones n, el vector r de las profundidades de búsqueda y el valor de la longitud de la combinación válida, todos ellos obtenidos según las ecuaciones de minimización. Nótese que el vector r indica las profundidades adecuadas e independientes a la combinación válida c y que la última profundidad de búsqueda es igual a $k-log_2 c$. Se pueden contrastar los costos en memoria y operaciones dados por (4.7) y (4.10) respectivamente con las tablas 4-3 y 4-4 para todos los valores de k. La demanda en memoria requerida sigue siendo exponencial sin embargo para k=32 se tiene una compresión que necesita el uso de una memoria de 466.8MB en comparación de la búsqueda básica que necesita 16.3GB. Las operaciones aritméticas no se incluyen pues como se menciona con anterioridad los accesos a memoria son los cuellos de botella en el procesamiento de la información.

Tabla 4-5 Costo en memoria e instrucciones para k en el intervalo [16,32]. El costo está dado por la ecuación 3.5. bajo los parámetros n, r, c.

k	*n*	*r*	*c*	*Costo MB*	*Acceso a memoria*
16	4	r=[0 3 7 10 13 16]	8	0.01213169	5
17	4	r=[0 3 7 10 13 17]	16	0.02322483	5
18	4	r=[0 4 8 11 14 18]	16	0.04127884	5
19	4	r=[0 5 9 12 15 19]	16	0.07738876	5
20	5	r=[0 2 6 10 13 16 20]	16	0.1496067	6
21	5	r=[0 3 7 11 14 17 21]	16	0.29404163	6
22	5	r=[0 4 8 12 15 18 22]	16	0.58291245	6
23	5	r=[0 5 9 13 16 19 23]	16	1.16065598	6
24	6	r=[0 2 6 10 14 17 20 24]	16	2.31614304	7
25	6	r=[0 3 7 11 15 18 21 25]	16	4.6271143	7
26	6	r=[0 4 8 12 16 19 22 26]	16	9.24905777	7
27	6	r=[0 5 9 13 17 20 23 27]	16	18.4929466	7
28	7	r=[0 2 6 10 14 18 21 24 28]	16	36.9807262	8
29	7	r=[0 2 6 10 14 18 21 24 29]	32	66.0008569	8
30	7	r=[0 3 7 11 15 19 22 25 30]	32	123.259745	8
31	7	r=[0 4 8 12 16 20 23 26 31]	32	237.777521	8
32	7	r=[0 4 9 13 17 21 24 27 32]	32	466.813076	8

4.5.4 Análisis y comparación entre los distintos algoritmos

Recordando algunos de los objetivos que inspiran este trabajo exigen un algoritmo de búsqueda que no requiera un gran número de instrucciones de acceso a memoria y que las estructuras de datos que utilice puedan estar contenidas en la memoria principal, comparamos las distintas formas de búsqueda propuestas para distintas longitudes máximas de prefijos.

Presentar medidas de desempeño de costo en memoria e instrucciones permiten al lector establecer en su caso particular si es factible aplicar alguno de los algoritmos o no, así como determinar qué parámetros de funcionamiento pueden ser utilizados; mismos que dependen de la tabla de ruteo que manejen ya que existirá una estadística de las longitudes máximas de los prefijos que la componen. La elección de los parámetros de operación dependerá de forma importante del hardware mismo, en particular del tamaño de la memoria principal *cache* o RAM ya que en ésta deben almacenarse por completo los

arreglos contadores para evitar las operaciones de intercambio de datos SWAP. Si la estructura de datos es contenida en memoria *cache* se podrá acceder a ella a la frecuencia máxima del procesador. Las operaciones que se realizan después de obtener el contenido de los registros son sumas por lo cual son conmutativas y se puede programar de forma concurrente para aprovechar los núcleos de un procesador si éste los tiene. Las operaciones serán efectuadas de acuerdo a la frecuencia máxima del procesador por lo que la capacidad de respuesta de un enrutador estará dada sólo por la memoria principal, número de núcleos y frecuencia de operación.

Primero se establece una comparación de consumo de memoria entre los distintos algoritmos de búsqueda. En la figura 4-19 se pueden observar los costos en memoria en bits para diferentes profundidades de k en el intervalo [3,32]; en el eje de las abscisas se encuentran las profundidades de búsqueda k y en el eje de las ordenadas se encuentra el costo en bits por realizar el algoritmo con una búsqueda directa, con conteos parciales y con conteos parciales más una optimización de la última profundidad de búsqueda; el costo se encuentra expresado en forma logarítmica con base 10 por lo que se pueden ver los costos de los diferentes algoritmos como trazas lineales. Se puede observar también que el costo para cualquier k se ve disminuido cuando se optimiza la técnica de búsqueda directa.

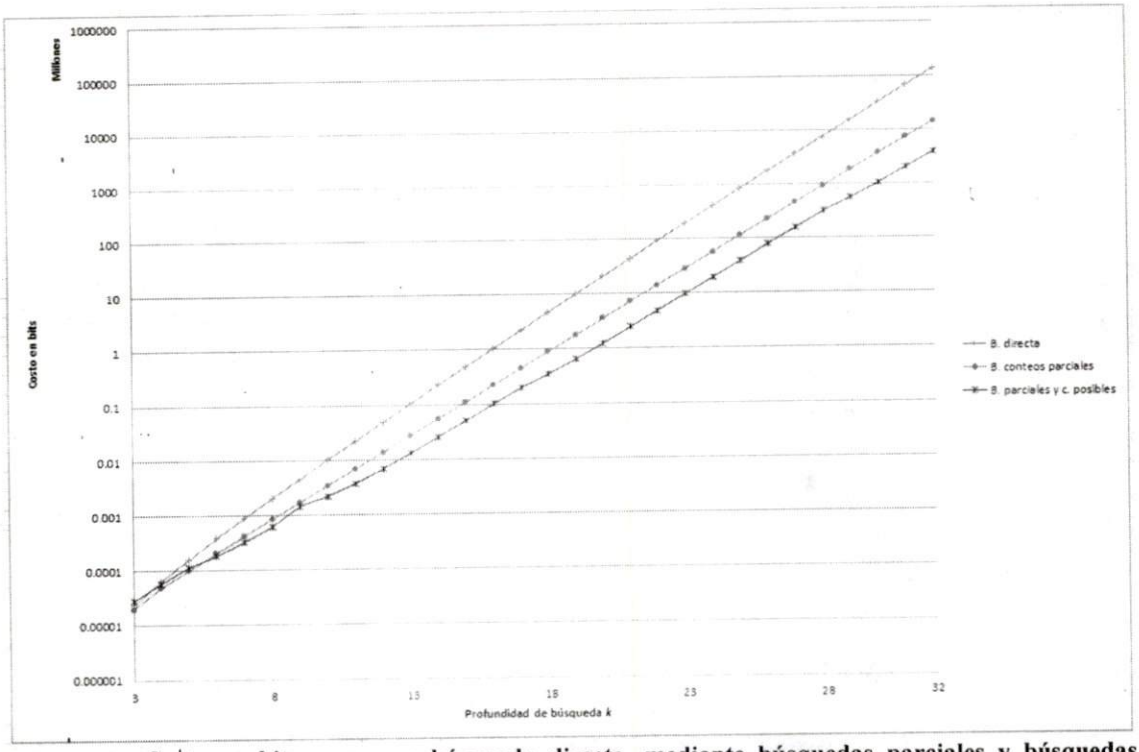


Figura 4-19 Costo en bits para una búsqueda directa, mediante búsquedas parciales y búsquedas parciales más una optimización en la última profundidad de búsqueda; cada valor de k (longitud máxima del prefijo) tiene una correspondencia de costo en cada uno de los algoritmos propuestos

El costo en memoria en bits para los arreglos contadores se puede reducir si utilizamos alguna de las técnicas antes descritas. Sin embargo, uno de los objetivos a cumplir es el número de instrucciones utilizadas para la obtener el prefijo más largo. En la figura 4-20 se muestra una gráfica que al igual que la figura 4-19 posee en el eje de las abscisas las distintas profundidades de búsqueda $\boldsymbol{k}$ en el intervalo [3,32] y en el eje de las ordenadas se encuentra la correspondencia del costo de instrucciones de acceso a memoria por efectuar búsquedas básicas, búsquedas parciales y optimización en la última profundidad de búsqueda. El costo para una búsqueda básica es una constante mientras que los costos por dividir al árbol en profundidades de búsqueda crecen de forma lineal de acuerdo a $\boldsymbol{k}$.

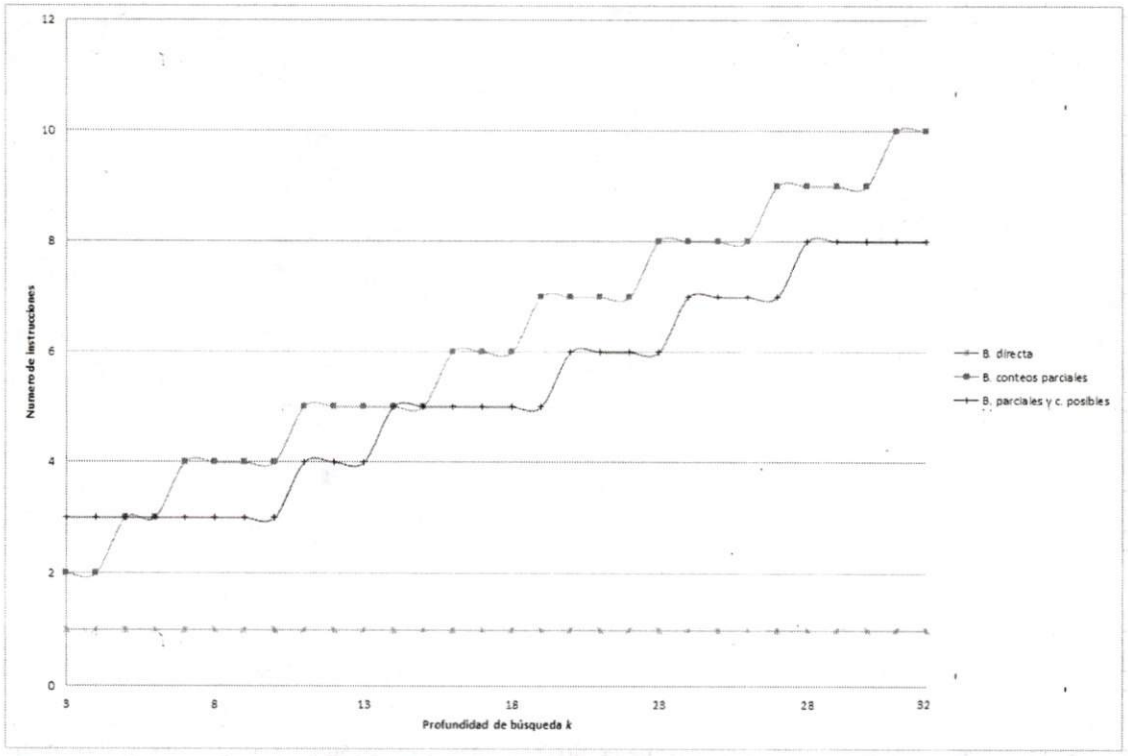


Figura 4-20 Costo en instrucciones para una búsqueda básica, mediante búsquedas parciales y búsquedas parciales más una optimización en la última profundidad de búsqueda; cada valor de k (longitud máxima del prefijo) tiene una correspondencia de costo en instrucciones en cada uno de los algoritmos propuestos

4.6 Simulaciones e implementaciones

Una vez realizado el análisis matemático, la simulación e implementación de la propuesta nos permite determinar qué tan factible es realizar búsquedas utilizando este algoritmo. El tiempo de búsqueda está determinado por el número total de operaciones realizadas como lo son las instrucciones de acceso a memoria (disco duro, de acceso aleatorio y *cache*) y las operaciones aritmético-lógicas. Sin embargo, no consideramos el tiempo que se utiliza para obtener las estructuras que nos ayudarán a realizar la codificación.

Recordemos que las estructuras que conforman la codificación son elaboradas con un vector de bits; el tiempo en construir el o los arreglos contadores dependerá de cuántos elementos haya en el vector de bits ya que éste se recorre bit a bit.

El tiempo que transcurre desde que una tabla de ruteo es cargada a memoria principal hasta que se construye el vector de bits dependerá tanto del número de prefijos en ella así como de la longitud de los mismos. En este trabajo utilizamos una tabla de ruteo real[10] que consta de 148610 prefijos de diferentes longitudes. Utilizamos una computadora mono procesador a 2.0GHz con una memoria *cache* de 1MB y 3GB de memoria RAM; el sistema operativo que utilizamos fue Fedora 15 y el compilador fue *gcc*. En la tabla 4-6 se muestran los tiempos transcurridos para cargar a los prefijos en memoria, construir el árbol binario, hacerlo completo, hacer una tabla de prefijos equivalentes y por último construir el vector de bits. Cabe mencionar que al hacer el árbol binario completo se crearon más prefijos en la tabla, en este caso de 148,610 prefijos se obtuvieron 219,044 prefijos equivalentes. El tiempo en realizar dichas actividades es totalmente manejable por una computadora, además que este proceso sólo se realiza cada vez que se tenga una nueva tabla de ruteo.

Los tiempos fueron medidos mediante el uso de la librería para lenguaje C: **sys/time.h** utilizando la función ***gettimeofday***, obteniendo así un tiempo dado en nanosegundos.

Tabla 4-6 Tiempos de construcción de estructuras de datos que ayudan a conformar la compresión

Tipo de estructura	*Costo nano segundos*	*Costo segundos*
Cargar tabla de ruteo	126090	0.00012609
Construir árbol binario	75737	0.000075737
Construir árbol binario completo	34571	0.000034571
Construir tabla de ruteo equivalente	26836	0.000026836
Construir vector de bits	4169	0.000004169

En este capítulo se presentó la contribución de nuestro trabajo de investigación en donde ofrecemos un algoritmo de búsqueda del prefijo más largo, el cual, es minimizado en instrucciones y memoria para ofrecer al lector parámetros de diseño según sus propias necesidades.

[10] La tabla de ruteo se consiguió de http://bgp.potaroo.net/index-ale.html

Capítulo
5 Trabajos Relacionados

En capítulos anteriores se presentaron distintos algoritmos para obtener el prefijo más largo correspondiente a una dirección IP destino de un paquete que pasa por un enrutador. Los algoritmos de búsqueda del prefijo más largo por lo regular utilizan estructuras de datos auxiliares para efectuar dicha búsqueda, por lo general tablas *hash* y arboles binarios y variante de ellos. En nuestra propuesta se establecieron algoritmos que ocupa un árbol binario completo para realizar diversos tipos de codificaciones mediante arreglos contadores.

El objetivo de este capítulo es proponer un marco de comparación con los algoritmos que proponemos y algoritmos que compartan los mismos objetivos que inspiran este trabajo de tesis y que por supuesto estén basados en algún tipo de compresión o codificación de las estructuras que utilicen. Recordemos que la inspiración inicial para la solución al problema de reexpedición de paquetes en base a los prefijos de red es el algoritmo de la universidad de LULEA. Este algoritmo aunque posee parámetros inamovibles de operación, es un buen modelo para la elaboración de codificaciones; y nuestra propuesta es una generalización de dicho algoritmo.

De la extensa familia de algoritmos de búsqueda del prefijo más largo que existen en la actualidad, el algoritmo de la universidad de LULEA es una propuesta que en esencia comparte nuestros objetivos de diseño pues está basado en la compresión del árbol binario y a diferencia de los otros algoritmos presentados (incluyendo a los que están basados en la compresión [13][14][15]), en una sección del algoritmo de LULEA se tienen valores constantes tanto en instrucciones como en tamaño en memoria requerida; además de presentar equidad ante cualquier tabla de ruteo y tráfico de Internet. Es por esto que lo utilizamos como marco de comparación pues con los demás algoritmos de búsqueda de

prefijos existen diferencias tanto en el método como en los objetivos específicos que no permitirían hacer una comparación que permita mostrar las cualidades de nuestra propuesta.

Se mostrará que se puede considerar el algoritmo de la universidad de LULEA como un caso particular de operación de nuestro algoritmo; se comparará de acuerdo al costo requerido en memoria y el costo total en instrucciones realizadas para determinar cuál minimiza memoria e instrucciones de mejor forma.

Recuérdese que nosotros hacemos un análisis matemático para estructurar nuestras propuestas de algoritmos; las comparaciones que se efectuarán con el algoritmo de LULEA serán basadas en el modelo matemático que proponemos. En la sección de simulaciones también hacemos comparaciones para corroborar experimentalmente lo que matemáticamente hemos demostrado.

5.1 Algoritmo de la universidad de LULEA

Este algoritmo realizado en la universidad de LULEA [12] situada al norte de Suecia establece búsquedas del prefijo más largo mediante el uso de un árbol binario completo y la compresión del mismo. Presentamos su funcionamiento básico para que el lector pueda identificar ciertas similitudes con algún caso en particular uno de los algoritmos que proponemos.

Los autores establecen un esquema de búsqueda completo del prefijo más largo para direcciones IP de 32 bits. Sin embargo, se podrá distinguir que una sección del algoritmo de búsqueda está contenida en uno de los algoritmos propuestos. A continuación se presenta la operación del algoritmo de la universidad de LULEA de acuerdo a sus reglas de construcción de las estructuras de datos que utiliza junto con los nombres que cada una de ellas tiene.

5.1.1Funcionamiento básico

El algoritmo de búsqueda del prefijo más largo que establece LULEA se efectúa mediante tres niveles: nivel 1, nivel 2 y nivel 3. En el primer nivel se pueden encontrar prefijos de longitud máxima 16; si el prefijo excede los 16 bits se efectúa la búsqueda en el segundo nivel, en donde se pueden encontrar prefijos de longitud máxima 24; si el prefijo excede los 24 bits entonces se completa la búsqueda en el nivel tres que cubre los prefijos de hasta 32 bits.

Los prefijos contenidos en la tabla de ruteo son representados por un árbol binario completo de profundidad 32; el árbol es seccionado en la profundidad 16 y en la profundidad 24. De la profundidad cero hasta la 16 se efectúan búsquedas correspondientes al nivel 1, de la profundidad 16 a la profundidad 24 se realizan búsquedas correspondientes al nivel 2 y por último de la profundidad 24 a la 32 se realizan búsquedas correspondientes al nivel 3. En la figura 5-1 se muestran los niveles de búsqueda.

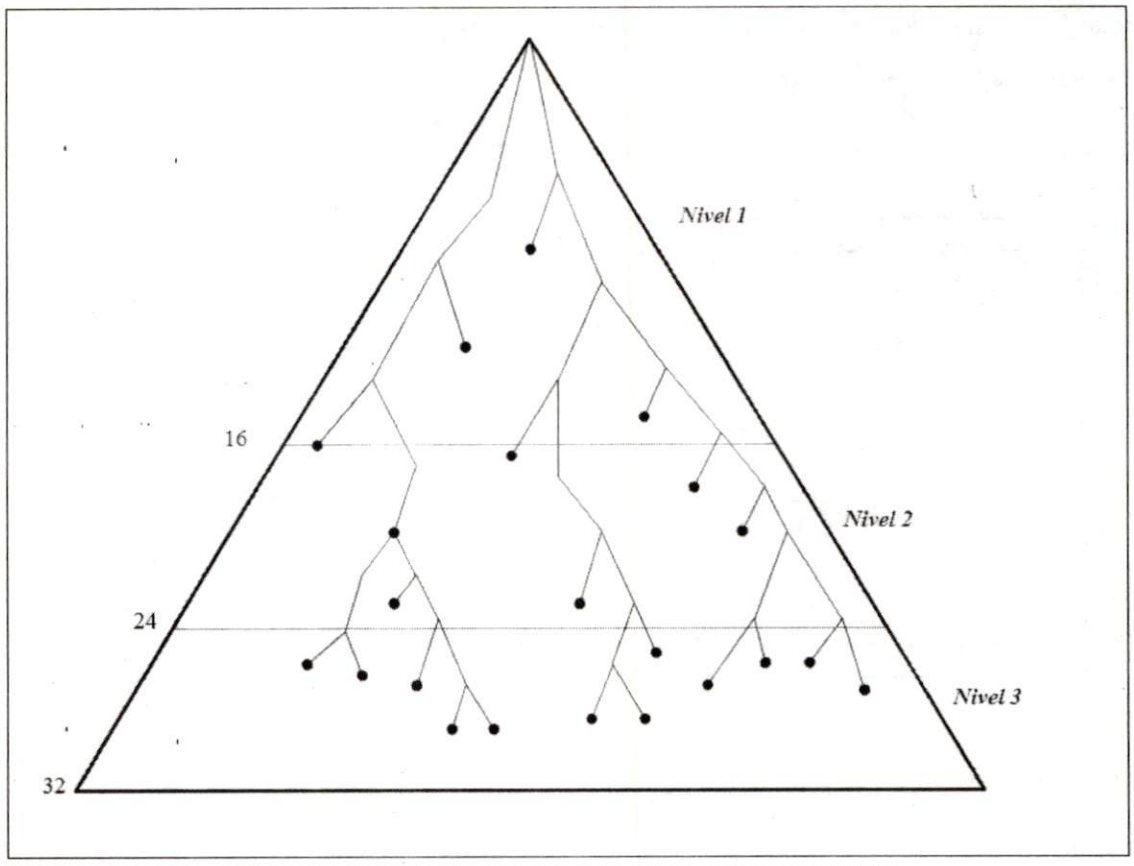


Figura 5-1 Tres niveles de búsqueda del algoritmo de LULEA

En el primer nivel de búsqueda se efectúa una compresión del árbol binario completo mediante tres arreglos de los cuales dos son unidimensionales y uno es bidimensional. Un primer arreglo es llamado ***base index*** y es de tamaño 2^{10} elementos por 16 bits cada uno de ellos; el segundo arreglo es llamado ***code word*** y es de tamaño 2^{12} elementos por 16 bits cada uno de ellos. Estos dos arreglos tienen valores que están determinados por la estructura del árbol binario. Un tercer arreglo llamado ***maptable*** contiene valores constantes que no dependen de la distribución de los prefijos de la tabla de ruteo.

En la búsqueda se distingue bien de qué longitud son los segmentos y dónde comienzan los bits de la dirección IP destino para usarlos como una llave para decodificar. Entonces, según sea la dirección IP destino se tendrá una posición en el base index, code word y maptable; se suma el contenido de cada uno de ellos y el resultado será índice del prefijo más largo de una dirección IP destino en la tabla de ruteo ordenada. En la figura 5-2 se

muestra el algoritmo de búsqueda del prefijo más largo correspondiente al nivel 1 del esquema de LULEA.

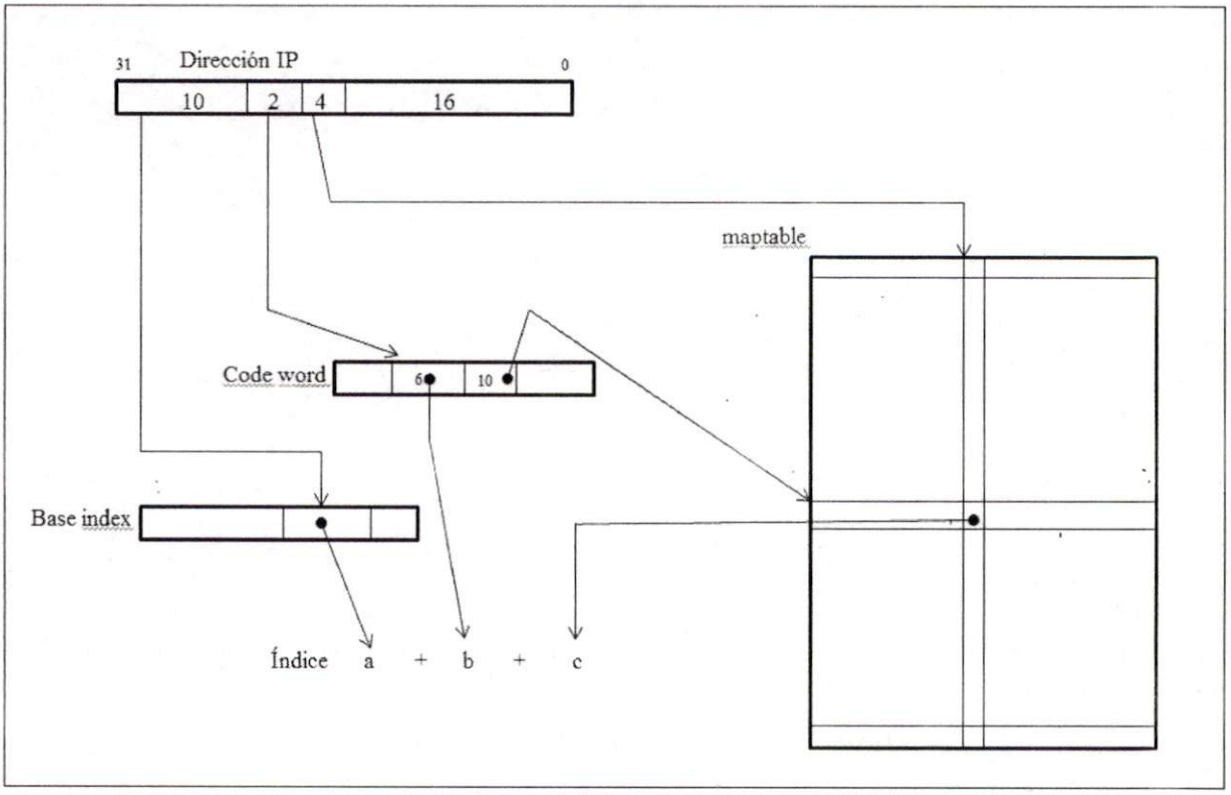


Figura 5-2 Algoritmo de búsqueda gráfico del esquema de la universidad de LULEA

5.2 Parámetros de comparación

La propuesta está inspirada por el planteamiento del problema que consiste en reexpedir el paquete que está dentro del enrutador lo más rápido posible, para ello distinguimos dos objetivos de comparación de nuestra propuesta: reducir el número de instrucciones de búsqueda y reducir el tamaño de las estructuras de datos a utilizar. Los demás objetivos no los ponemos en comparación pues el algoritmo de la universidad de LULEA presenta las mismas características de nuestra propuesta.

Minimizar tanto el número total de operaciones de búsqueda como el tamaño en memoria repercute directamente en la capacidad de respuesta del enrutador. Se distinguen dos tipos de instrucciones: aritmético-lógicas y de acceso a memoria o de extracción de bits; recordemos que las operaciones aritmético-lógicas son hechas a la frecuencia del procesador mientras que el acceso a memoria puede ser altamente costoso si la información no se encuentra en memoria principal. Por ejemplo, en una computadora se encuentran las unidades de almacenamiento disco duro, memoria de acceso aleatorio RAM y memoria *cache*; el acceso a la información en un disco duro es mucho más tardado que un acceso a la memoria *cache*. Por lo tanto, las instrucciones de acceso a memoria en comparación con las instrucciones aritmético-lógicas pueden ser mucho más lentas.

Minimizar el tamaño de las estructuras de datos utilizadas ayuda a evitar que estas se encuentren dispersas en distintas memorias de almacenamiento, manteniéndolas alineadas para su fácil acceso. Si el tamaño de las estructuras es lo suficientemente pequeño como para ser contenida en memoria *cache* de un procesador se efectuarían operaciones de extracción de bits a la frecuencia máxima de operación.

En base a los argumentos antes mencionados, podemos establecer los parámetros de comparación entre nuestro algoritmo y el esquema de LULEA:

1. El número de instrucciones de acceso a memoria utilizada para la búsqueda del prefijo más largo.
2. El tamaño de las estructuras de datos utilizadas que conforman la compresión.

5.3 El esquema de LULEA como caso particular de nuestro algoritmo

Para que se dé la compresión del árbol binario que representa a los prefijos en la tabla de ruteo, el algoritmo de la universidad de LULEA se construye un arreglo lineal que representa la posición de los prefijos en el árbol. Por último, efectúa una codificación de este arreglo para hacer búsquedas de menor espacio y menor número de instrucciones. Esta codificación está dada por los tres arreglos antes mencionados (*base index, code word y maptable*).

Recordemos que en este trabajo hemos presentado un vector de bits que representa la información de ruteo contenida en un árbol binario; sin embargo el algoritmo que utiliza conteos parciales y combinaciones válidas en el vector de bits, es quien tiene una similitud con el primer nivel de búsqueda del algoritmo de la universidad de LULEA pues es un caso específico de él. A continuación describimos el primer nivel de búsqueda de LULEA en función del razonamiento que hicimos para comprender de mejor forma su similitud.

5.3.1 Número de veces que se aplica el algoritmo

Como se mencionó, se tienen direcciones de 32 o 128 bits en consecuencia los prefijos tienen una longitud máxima de 32 o 128 bits, sin embargo se puede hacer una búsqueda de prefijos de longitud máxima $k \leq 128$ para después aplicar el algoritmo una o más veces, o bien efectuar otro tipo de búsqueda.

Nuestra propuesta no tiene restricciones en cuanto al número de veces que se puede aplicar el algoritmo ni en cuanto a la longitud máxima de los prefijos a encontrar, esto porque se da solución a todos los valores de k posibles.

El algoritmo de la universidad de LULEA también representa a la tabla de ruteo mediante un árbol binario completo de profundidad k=32 y aplica el algoritmo de búsqueda en tres niveles diferentes mostrados en la figura 4.1 los cuales corresponden a k_1=16, k_2=24,

k_3=32. Entonces si el prefijo no excede los 16 bits nada más se aplica el algoritmo una única vez, pero si el prefijo es mayor a 16 bits; pero menor o igual a 24 bits, se aplicará el algoritmo dos veces, uno en el primer nivel y otro en algún sub árbol del segundo nivel; por lo tanto si el prefijo excede los 24 bits el algoritmo se aplica tres veces.

5.3.2 Primera profundidad de búsqueda

En nuestra propuesta realizamos búsquedas en árboles de cualquier profundidad ***k*** dejándolo como parámetro abierto mientras que el algoritmo de la universidad de LULEA tiene un primer nivel de búsqueda en ***k*** = 16 (nivel 1) pues en el año de publicación (1997) la mayor parte de los prefijos se encontraban alrededor de 16 bits; hoy en día con la llegada de IPv6 los prefijos tienen una mayor longitud.

Una vez definida la longitud ***k*** de búsqueda, ambas propuestas representan a los prefijos de la tabla de ruteo mediante un árbol binario completo por lo que la tabla puede sufrir algún cambio en su contenido de tal forma que se tenga la misma información de ruteo. El árbol binario completo es la estructura auxiliar de búsqueda y posteriormente es desechada por ambas propuestas.

El árbol binario completo de profundidad ***k*** es una estructura que en cada una de sus ramas está representado un prefijo; es decir, si se efectúa un recorrido desde la raíz hasta un nodo terminal se tendrá un prefijo, además por ser un árbol completo, todas las direcciones IP están cubiertas por una hoja (no hay huecos). Una forma de obtener una representación de los prefijos y de las direcciones IP que cubren es mediante el uso de un arreglo de 2^k elementos de 1 bit cada uno de ellos; a esta estructura se le nombró en el capítulo 3 como **vector de bits** de tal forma que cuando se encuentre un *1* en el vector de bits se sabrá que existe un prefijo que representa un determinado grupo de direcciones IP hasta el siguiente *1*. El algoritmo de la universidad de LULEA también establece la construcción de un arreglo que represente a los prefijos en el árbol, esta estructura es de tamaño 2^k elementos de 1 bit cada uno de ellos y es conocido como **bit-vector**. En la figura 5-3 se puede

observar una fracción del árbol binario y el fragmento del árbol binario correspondiente que tiene los valores según las siguientes tres reglas:

1. Un ***1*** si en la profundidad 16 existe un nodo que no es prefijo (bits 6, 12 y 13 de la figura 5-3).
2. Un ***1*** si se tiene una "cabeza genuina[11]" (bits 0, 4, 7, 8, 14 y 15 de la figura 5-3).
3. Un ***0*** en los demás casos (bits 1, 2, 3, 5, 9, 10 y 11 de la figura 5-3).

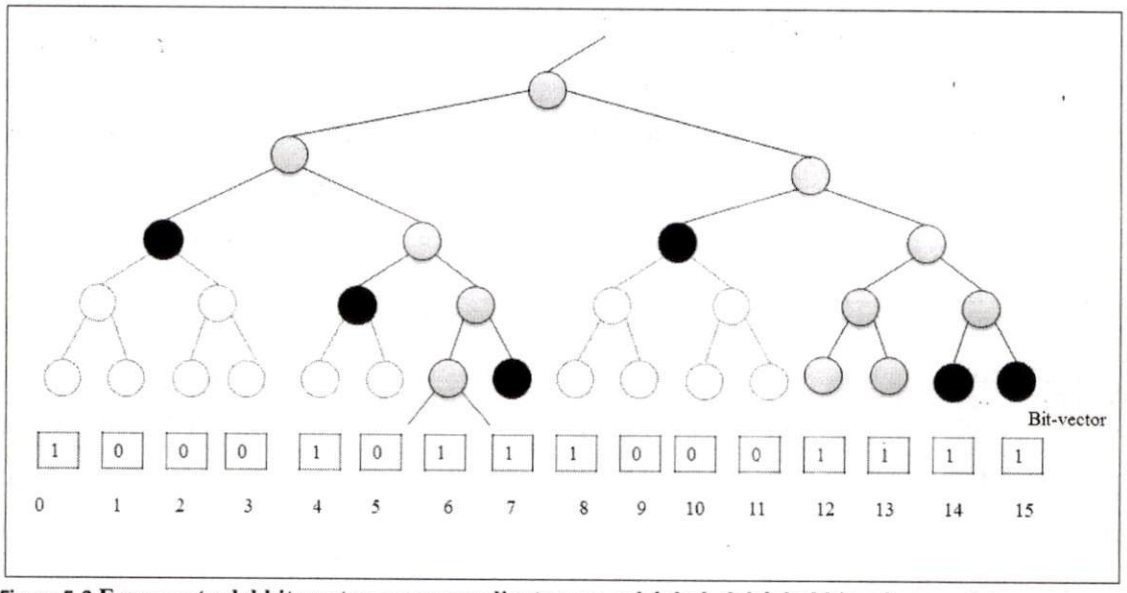


Figura 5-3 Fragmento del bit-vector correspondiente a un subárbol del árbol binario completo.

En esta propuesta las reglas 2 y 3 de construcción son las mismas que nuestra propuesta pues a lo que LULEA llama "cabeza genuina" nosotros la consideramos como un prefijo en cualquier profundidad menor o igual a ***k*** y tiene un valor específico en el vector de bits la cual es marcada con un ***1***.

Con las reglas de construcción de ambas propuestas se obtiene exactamente el mismo resultado por lo que en la construcción de estructuras auxiliares se sigue el mismo método. Una dirección IP destino se puede considerar como un valor numérico que define una posición en alguna entrada tanto en el vector de bits como en el bit-vector. La búsqueda consiste en contar el número bits puestos en ***1*** a la izquierda de donde se sitúa la dirección

[11] En el algoritmo de la universidad de LULEA se denota una "cabeza genuina" como un prefijo en una , profundidad menor a 16.

IP. El resultado será el índice del prefijo más largo en la tabla de ruteo ordenada de forma ascendente que corresponde al árbol de profundidad k.

En nuestra propuesta se contemplan tres formas de contar el número de bits "*1*" en el vector de bits: mediante un arreglo contador de tamaño $2^k k$, mediante una serie de arreglo contadores de tamaño dado por la ecuación (4.3) en el capítulo 4 y mediante una serie de contadores de los cuales el último corresponde a una optimización en memoria del último contador pues se parte de un árbol binario completo; el costo está dado por la ecuación (4.7) en el capítulo 4.

Para nuestra propuesta que consta de n cortes en el árbol dadas por un arreglo r que como se menciona en el capítulo anterior $r = \{\alpha_0=0, \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_{n-1}, \alpha_n, \alpha_k= \alpha_{n+1},=k\}$ tal que los α_i son las profundidades de búsqueda en el árbol, $\alpha_i \leq \alpha_{i+1}$, $\alpha_i \in \mathbb{N}$; entonces r es un vector que posee todas las profundidades de búsqueda incluyendo la profundidad 0 y la profundidad total del árbol k por lo tanto $n=|r|-2$; y que además se cuenta con una optimización en la última profundidad de búsqueda que depende de una combinación de c bits válidos en el vector de bits; se tuvo una definición de la función de costo en bits $f_m=f_m(r,n,c)$ la cual está definida en la ecuación (4.7) del capítulo anterior. La idea de tener una función asociada de costo en memoria en bits a este algoritmo fue reducir el tamaño de las estructuras que el método utiliza. Encontrando que para k=16 se tiene un $r = \{0,3, 7, 10, 13, 16\}$, $n = 4$ y $c = 8$.

El primer nivel de búsqueda del algoritmo de la universidad de LULEA es un caso particular para f_m con un $k=16$. Sin embargo se tienen los valores de los parámetros $n=2$, $r = \{0, 10, 12, 16\}$ y $c = 16$. El valor de los parámetros repercute en el costo en memoria y el número de instrucciones de acceso a memoria por lo que podremos comparar los resultados obtenidos.

Nosotros definimos a $f_m=f_m(r,n,c)$ con tres grados de libertad r, n, c y el valor de k es un parámetro fijo con el cual se resuelven 128 problemas diferentes. El algoritmo de LULEA no define estos parámetros es por ello que es un caso particular de nuestro algoritmo. Para ellos se tiene un k único igual a 16 y dos búsquedas internas con tres arreglos contadores. Dos contadores están situados en las profundidades $\alpha_1=10$ y $\alpha_2=12$ los cuales tienen los

nombres de "**base index**" y "**code word**" respectivamente; un tercer contador es una optimización en la última profundidad de búsqueda y corresponde al arreglo contador de combinaciones válidas denotado como ***matriz(c)*** y al que ellos le nombran **maptable**.

5.3.3 Un error en el algoritmo de LULEA

En el primer nivel de búsqueda se forma un árbol binario completo de profundidad 16 en el cual se hacen tres conteos parciales en las profundidades 10, 12 y 16 del árbol. Los dos primeros contadores ***base index*** y ***code word*** hacen contentos de bits puestos en ***1*** del **bit vector** de los 2^{10} y de los 2^{12} sub-árboles que forman respectivamente. Las reglas de construcción están dadas por [12] y expresan que tanto el *base index* como el *code word* cuentan los bits puestos en ***1*** del vector de bits en intervalos de 64 y 16 bits respectivamente.

Si se diera el caso especial en que en la profundidad 10 (profundidad de corte del *base index*) no existiera uno o más sub-árboles; es decir, que haya prefijo de longitud menor a 10 se presenta la situación expresada en el capítulo 4 sección 4.3.2 con el ejemplo de la figura 3-14 se tendría un grave error pues el conteo de bits puestos en ***1*** no sería correcto. Este error ocurre si en la tabla de ruteo existen prefijos de longitud menor a 10 pues en el árbol binario un prefijo constituye una rama desde la raíz hasta una hoja. De esta forma, para hacer conteos parciales se tiene que tener cuidado de contar el "número de prefijos a la izquierda" pues en los caso en que no existan su-árboles en la profundidad de corte ***p*** se tienen que contemplar el número de direcciones IP del ancestro inmediato que cubra todos los subárboles de ***p***. En la figura 5-4 se muestra el árbol binario de profundidad 16 que determina el primer nivel de búsqueda de LULEA y su primer corte determinado por el base *index*. Se presentan también los intervalos de direcciones que abarcan y el caso particular en el que un prefijo no tenga una longitud de 10 bits. Nótese el valor del bit vector en ese intervalo y cómo influye en el valor del *base index*. También se incluye el valor del *base index* en ese intervalo y el valor que éste debería tener para ser correcto.

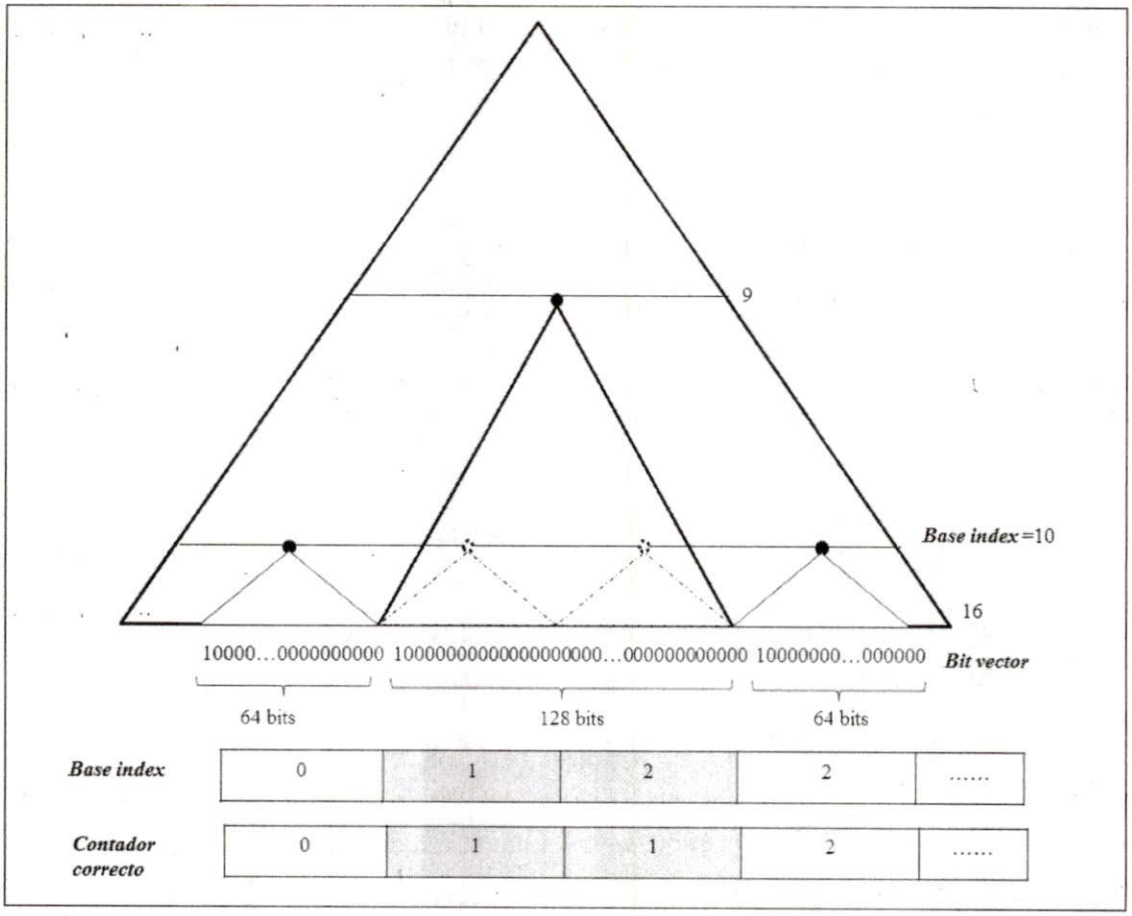


Figura 5-4 Error en el algoritmo de la universidad de LULEA

5.4 Análisis comparativo de costo en memoria e instrucciones

Se presentó el algoritmo de la universidad de LULEA como un caso particular en uno de los algoritmos que proponemos. Sin embargo, no comparten los mismos parámetros de operación (número de cortes en el árbol, posición de los cortes para efectuar conteos parciales y conteos parciales mediante combinaciones válidas en el vector de bits) por lo que se tienen diferentes costos tanto en memoria como en número de instrucciones. El objetivo de esta sección es comparar el primer nivel de búsqueda de LULEA con la propuesta que optimiza el número total de instrucciones y la propuesta que minimiza el

tamaño en memoria de las estructuras utilizadas para tablas de ruteo en la cual sus prefijos tienen una longitud máxima de 16 bits.

5.4.1 Búsqueda mediante conteos parciales y combinaciones válidas *vs.* LULEA

Para nuestra propuesta se tiene el caso particular de k=16 con los siguientes parámetros de operación optimizados en memoria: $r = \{0,3, 7, 10, 13, 16\}$, n = 4 y c=8. Con lo cual se tiene $f_m(r,n,c)$ = 0.01213169MB con cinco operaciones de extracción de bits y cuatro operaciones aritméticas dadas por f_i. Sin embargo, como se puede ver en la tabla 5-1 el algoritmo de la universidad de LULEA tiene un costo en memoria $f_m(r,n,c)$ = 0.01493835MB con $r = \{0,10, 12, 16\}$, n = 2 y c=16 con tres operaciones de acceso a memoria y dos operaciones aritméticas; de esta forma se puede ver que el algoritmo que proponemos optimiza en un mayor grado el costo en memoria sin embargo el algoritmo de la universidad de LULEA es más eficiente en el número de instrucciones totales de acceso a memoria y aritmético-lógicas. Los costos están dados por $f_m(r,n,c)$ y son valores

Tabla 5-1. Comparación en costo de operaciones y memoria entre el algoritmo de la universidad de LULEA y nuestra propuesta optimizada en memoria.

	LULEA	Nuestra Propuesta con c, n, r, optimizados en memoria
n	2	4
r	{0, 10, 12, 16}	{0, 3, 7, 10, 13, 16}
c	16	8
Costo en MB	0.01493835MB	0.01213169MB
Instrucciones de acceso a memoria	3	5
Instrucciones aritméticas	2	4

Como se pudo observar en la tabla 5-1 nuestra propuesta es más eficiente en memoria. Sin embargo, se tienen más instrucciones tanto de accesos a memoria como aritméticas con respecto al de LULEA por lo que en algunas circunstancias podría tener un mejor desempeño el algoritmo de la universidad de LULEA. Ahora bien, si fijamos en nuestro algoritmo con n=2 y encontramos los valores óptimos de r y c para dos búsquedas internas podremos igualar el desempeño en instrucciones tanto de accesos a memoria como aritméticas de la universidad de LULEA y el costo estará dado por la tabla 5-2 así como los valores de c y de r .

Tabla 5-2 Comparación en costo de operaciones y memoria entre el algoritmo de la universidad de LULEA y nuestra propuesta optimizada en memoria y en instrucciones.

	LULEA	Nuestra Propuesta con n=2, c, r, óptimos en memoria
n	2	2
r	{0, 10, 12, 16}	{0, 9, 13, 16}
c	16	8
Costo en MB	0.01493835MB	0.012768984MB
Instrucciones de acceso a memoria	3	3
Instrucciones aritméticas	2	2

Demostramos matemáticamente con la funciones f_m y f_i que el algoritmo de la universidad de LULEA es un caso particular del algoritmo de conteos parciales y combinaciones válidas en el vector de bits con el valor de k=16. Nuestro algoritmo ofrece parámetros que minimizan el costo en memoria y el algoritmo de LULEA intenta minimizar el número de instrucciones. Sin embargo, la flexibilidad de nuestra propuesta de codificación permite igualar el número de instrucciones y mejorar el tamaño de memoria requerida para las estructuras de datos.

5.4.2 Búsqueda directa *vs.* LULEA

El algoritmo de búsqueda directa optimiza el número de instrucciones de acceso a memoria pues se efectúa exactamente una operación de extracción de bits y cero operaciones aritméticas; el costo en memoria está dado por el único contador de bits puestos en *1* del vector de bits y es de tamaño $k2^k$ tal que k es la profundidad de árbol binario y la longitud máxima de los prefijos a encontrar.

Podemos comparar el algoritmo de búsqueda directa y el primer nivel de búsqueda de LULEA en base a su desempeño para búsquedas de prefijos en enrutadores actuales. La tabla 5-3 muestra el costo en memoria e instrucciones de acceso a memoria y aritméticas del esquema de LULEA para el primer nivel de búsqueda (prefijos de máxima longitud 16) y el costo para algunas profundidades de búsqueda. Nótese que el costo en memoria es mayor en la búsqueda directa, sin embargo las computadoras básicas actuales por lo menos tienen una memoria *cache* entre 1 y 2 MB (megabytes); con lo que se pueden efectuar perfectamente búsquedas del prefijo de máxima longitud 19 bits con estructuras de datos contenidas en su totalidad en esta memoria. El desempeño es claramente mejor en la búsqueda directa pues el esquema de LULEA efectúa más instrucciones y obtiene un prefijo de menor longitud (16) mientras que en la búsqueda directa se obtiene un prefijo más largo por ciclo de reloj y de mayor longitud.

Tabla 5-3 **Comparación en costo de operaciones y memoria entre el algoritmo de la universidad de LULEA y nuestra propuesta optimizada en instrucciones.**

	Longitud máxima del prefijo	Costo MB	Accesos a memoria	Operaciones aritméticas
LULEA	*Nivel 1*=16	0.01493835	3	2
Algoritmo de búsqueda directa	*k*=16	0.125	1	0
	k=17	0.265625	1	0
	k=18	0.5625	1	0
	k=19	1.1875	1	0
	k=20	2.5	1	0

5.4.3 Simulaciones

Para corroborar los resultados del modelo matemático que proponemos, implementamos algunos casos particulares de operación de las variantes de nuestro algoritmo. En capítulos anteriores utilizamos como ejemplo computadoras que tenían una memoria *cache* de 1MB; expresamos que en una búsqueda directa utilizábamos 1.1MB para encontrar prefijo de máxima longitud 19 bits. También dijimos que utilizando nuestro algoritmo de conteos parciales más combinaciones válidas en el vector de bits podíamos hacer búsquedas de hasta 23 bits sin exceder un tamaño de memoria requerida de 1MB. Por ello, implementamos la búsqueda directa con k=16 y k=19 (longitud máxima de prefijos); también implementamos la búsqueda mediante conteos parciales con combinaciones válidas para k=16 y k=23 con n, r, c optimizados en memoria (particiones en el árbol, profundidades de las particiones y tamaño de la combinación válida). Para comparar el esquema de LULEA hicimos la implementación del mismo y una implementación de búsqueda mediante conteos parciales con combinaciones válidas para un k=16 y k=23 con n=2 y r, c optimizados en memoria para igualar el número de instrucciones. La arquitectura usada fue una computadora con sistema operativo Fedora 15 y compilador gcc; el hardware utilizado fue un monoprocesador a 2GHz y memoria *cache* de 1MB con 3GB de memoria RAM.

En la tabla 5-4 se muestra el tiempo que cada implementación se tardó en encontrar 1 millón de prefijos en nanosegundos y segundos de una tabla de ruteo real. El tiempo es un promedio ya que se realizó el experimento mil veces para tener un promedio más acertado. Nótese que la búsqueda directa ocupa más tiempo que el algoritmo de la universidad de LULEA. Sin embargo, es de esperarse pues en una computadora existen muchos procesos que se encuentran en espera de obtener los recursos del sistema y éstos requieren estar también en memoria principal por lo que el tamaño en memoria es un factor muy importante. Nótese también que cuando fijamos a n=2 obtenemos un tiempo 20% menor de búsqueda con respecto a LULEA y si utilizamos los parámetros de n, r, c óptimos para k=16 el desempeño es casi idéntico con respecto al de LULEA. Para k=23 se necesitan 0.000101206 segundos que es un tiempo mucho más grande que todos los algoritmos. Sin

embargo, en esta búsqueda en una sola iteración se encuentran hasta 23 bits del prefijo más largo.

Tabla 5-4 **Tiempo de ejecución para 1 millón de prefijos en varios algoritmos**

Tipo de búsqueda	*k*	*n*	*r*	*c*	*Costo en nano segundos*	*Costo en segundos*
Directa	16	0	-----	--	53253	0.000053253
Directa	19	0	-----	--	55538	0.000055538
LULEA (nivel 1)	16	2	[0,10,12,16]	16	52032	0.000052032
Conteos parciales y combinaciones válidas n=2 r= óptimo	16	2	[0, 9,13,16]	8	39662	0.000039662
Conteos parciales y combinaciones válidas n=óptimo , r= óptimo	16	4	[0,3,7,10,13]	8	51630	0.000051630
Conteos parciales y combinaciones válidas n=óptimo , r= óptimo	23	5	[0, 5,9,13,16,19,23]	16	101206	0.000101206

En este capítulo se presentó el algoritmo de la universidad de LULEA como un trabajo relacionado con nuestra propuesta; se realizaron también algunas simulaciones que nos permitieron comprobar experimentalmente las cualidades principales de nuestro algoritmo. Terminamos este capítulo reconociendo el alto desempeño del algoritmo de la universidad de LULEA. Sin embargo, en nuestra propuesta tenemos un mejor rendimiento pues se minimizan de forma más adecuada los parámetros de operación, además de que el algoritmo propuesto es escalable a IPv6.

Capítulo
6 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. Observamos el proceso de encaminamiento de la información, distinguiendo que en cada dispositivo de encaminamiento o enrutador se tiene que determinar qué enlace de salida es el más apropiado para reexpedir el paquete en base a una dirección destino.

Abordamos el problema de toma de decisión que tiene un enrutador para reexpedir un paquete por uno de los enlaces de salida en base a una dirección destino en base a los prefijos de red. Existen muchas formas de encontrar el prefijo más largo en una tabla de ruteo. Sin embargo, en este trabajo enfocamos nuestra atención en los algoritmos de búsqueda basados en la compresión y codificación. Hicimos un análisis matemático con el cual proponemos algoritmos de búsqueda del prefijo más largo que conforman propuestas que optimizan el número de instrucciones y el espacio en memoria requerido para las estructuras de datos utilizadas.

Inspirados en el algoritmo de la universidad de LULEA. En este trabajo de tesis, logramos construir algoritmos para la búsqueda del prefijo más largo que dependiendo del grado de codificación pueda ajustar su complejidad en memoria o bien en instrucciones.

Se proponen tres formas de codificar el vector de bits. El algoritmo de búsqueda directa esta intrínsecamente minimizado en instrucciones de acceso a memoria (solo un acceso) pero la cantidad de memoria que utiliza no es fácilmente manejable por una computadora. Definimos la segunda forma de codificar el vector de bits (en base a conteos parciales dados por cortes en el árbol). Esta forma de codificar el vector de bits define el segundo algoritmo que funciona de acuerdo a las variables n y r, sin embargo se utilizó para llegar al

último algoritmo el cual ofrece una mayor optimización en memoria y que no afecta de forma significativa a las operaciones de acceso a memoria.

En resumen se plantearon tres algoritmos que dependen de sus parámetros de operación; la principal contribución de este trabajo de tesis fue el análisis de las formas de codificación del vector de bits para la estructuración de un algoritmo que pueda ser minimizado en memoria o bien en instrucciones de acceso a memoria. El análisis ofrece la posibilidad de construir un algoritmo que determine el prefijo más largo de máxima longitud k y que a conveniencia del lector decida el grado de codificación del vector de bits de acuerdo a las distintas formas presentadas. Se encontró de forma matemática la minimización en memoria y en instrucciones de acceso a memoria de los algoritmos propuestos para que el lector tenga un marco de referencia para utilizar uno de ellos junto con sus parámetros adecuados.

Los objetivos plantados al inicio de este trabajo se cumplieron en su totalidad pues proponemos algoritmos de búsqueda del prefijo más largo que demuestran la hipótesis de que existen mecanismos de codificación y compresión del vector de bits (estructura de datos utilizada) que minimizan la cantidad de memoria requerida o bien el número de instrucciones de acceso a memoria para la obtención del prefijo más largo. Además de que nuestra propuesta ofrece las siguientes ventajas:

- Estructuras de datos de tamaño independiente al número y longitud de prefijo en la tabla de ruteo.
- Equidad de acceso a las estructuras de datos dada cualquier dirección IP destino.
- Numero constante de instrucciones para la obtención del prefijo más largo constantes a cualquier dirección IP de longitud k.
- Parámetros abiertos como la longitud máxima de prefijos a encontrar (k), cantidad de costes en el árbol (n) poción de los cortes en el árbol (r) etc.

Las desventajas de nuestra propuesta son las siguientes:

- Aunque los algoritmos propuestos tiene como valor abierto la longitud máxima de los prefijos a encontrar (k); su escalabilidad a IPv6 puede ser cuestionada ya que el consumo de memoria crece de forma exponencial de acuerdo a k.

- Para elaborar la codificación del vector de bits se necesita el uso del árbol binario completo lo que significa que se necesita tiempo y recurso de procesamiento en una estructura que será desechada al final de la codificación.
- No se puede minimizar su complejidad en memoria e instrucciones al mismo tiempo; sin embargo, al minimizar en memoria el crecimiento de instrucciones de acceso a memoria es mínimo.

Finalmente encontramos que de forma matemática y en base a simulaciones nuestra propuesta es más eficiente que el algoritmo de comparación (LULEA) pues es un caso particular para uno de nuestros algoritmos; sin embargo, nuestra aportación es un análisis profundo de codificación de vector de bits que determina el costo tanto en instrucciones como en memoria utilizada para búsquedas del prefijo más largo. Proponemos métricas que se pueden ajustar a la arquitectura de un enrutador y a la tabla de ruteo contenida en él. Tanto la búsqueda directa como búsquedas con conteos parciales basados en combinaciones posibles del vector de bits pueden ser utilizadas.

Capítulo
7 Trabajo futuro

A continuación se presentan las oportunidades de investigación para continuar este trabajo:

Las diferentes formas de codificación de bits consideran la búsqueda de prefijos de longitud máxima k tal que $1 \leq k \leq 128$ por lo que si un prefijo en la tabla excede k se tendrá que hacer una nueva búsqueda para completar el prefijo más largo. Nosotros no proponemos algún criterio de decisión para volver a aplicar el algoritmo o bien aplicar otro tipo de búsqueda por lo que no tenemos un argumento para aplicar más de más vez el algoritmo.

El algoritmo de búsqueda: "conteos parciales más apuntadores a combinaciones válidas" tiene un arreglo contador de combinaciones posibles llamado "***matriz(c)***" y es función de una combinación posible de c bits en el vector de bits. La ***matriz(c)*** es una constante dada por la ecuación 4.6 y contempla todas las combinaciones validas en el vector de bits, sin embargo, puede darse la situación en que se tenga un árbol binario que no contemple a todas la combinaciones válidas y por consecuencia jamás sean utilizadas en el algoritmo de búsqueda por lo que se puede reducir aún más el tamaño de la ***matriz(c)***. En este trabajo no se contempla un análisis de los valores de c que existen en las tablas de ruteo reales y actúales.

Para cualquier forma de codificación de vector de bits que proponemos se utilizan uno o más arreglos contadores que llevan un conteo acumulativo del número de bits puestos en ***"1"*** de una sección del vector de bits a manera de *offset*. Por lo tanto, 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 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 que no se necesitan para contar. En la figura 7-1 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 tiempo de construcción del árbol tiene como cota el tiempo en que se construye árbol de profundidad 32 o 128 (IPv4 o IPv6) es mencionado pues las tablas de ruteo cambian conforme pasa el tiempo porque la red no es estática y cambia de topología constantemente y cada vez que esto suceda se tendrán que construir los arreglos contadores con ayuda de un árbol binario. Establecer la compresión de árbol (vector de bits) y la codificación del vector de bits sin tener que reconstruir el árbol o en su defecto utilizarlo mejora el desempeño para una red dinámica.

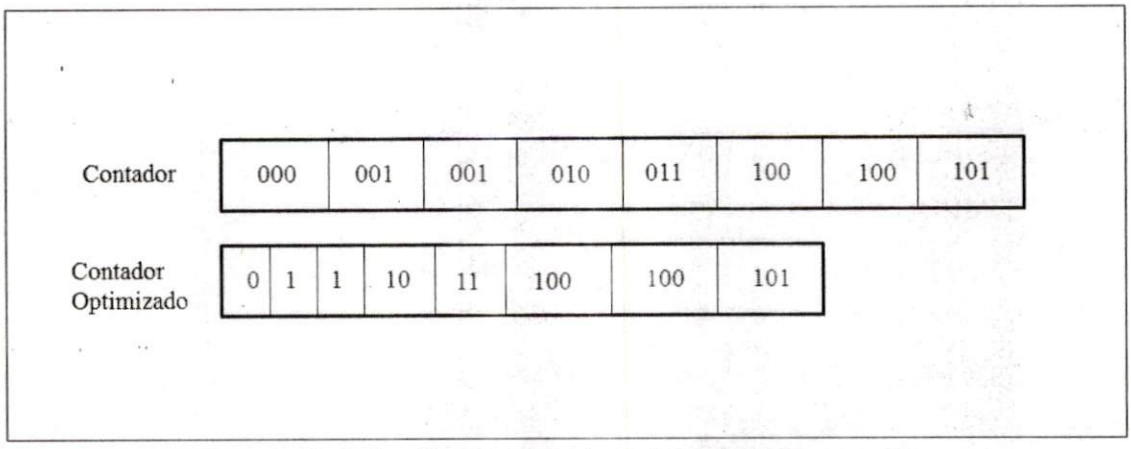


Figura 7-1 Optimización de un arreglo contador de 3 bits por casilla.

En el capítulo de trabajos relacionados se presenta un análisis de desempeño entre nuestros algoritmos t el algoritmo de la universidad de LULEA. Sin embargo hace falta hacer elaborar más copara raciones con otras propuestas para resaltar las cualidades de nuestro algoritmo.

Referencias

[1] DARPA INTERNET PROGRAM, RFC971, Internet Protocol, Defense Advanced Research Projects Agency, Septiembre 1981.

[2] Miguel Á. Ruiz-Sánchez, Ernst W. Biersack and Walid Dabbous , "Survey and Taxonomy of IP Address Lookup Algorithms", IEEE Network, Vol. 15 No. 2, pp. 8-23, Marzo/abril 2001.

[3] V. Srinivasan, George Varghese, Faster IP Lookups using Controlled Prefix Expansion, SIGMETRICS 1998 Madison USA, ACM.

[4] Butler Lampson, Venkatachary Srinivasan and George Varghese, IP Loookups Using Multiway and Multicolumn Search, IEE / ACM Transactions on Networking, Vol. 7 No3, junio 1999.

[5] Marcel Waldvogel, George Varghese, Jon Turner and Bernhard Plattner, Sca- lable High Speed IP Routing Lookups, SIGCOMM 1997 Cannes Francia.

[6] Ju Hyoung Mun, *Student Member, IEEE,* Hyesook Lim, *Member, IEEE,* and Changhoon Yim, *Member, IEEE* "Binary Search on Prefix Lengths for IP Address Lookup", IEEE communications letters, vol. 10, no. 6, junio 2006

[7] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookup," in *Proc. ACM SIGCOMM*, pp. 25-35.

[8] V. Srinuvasan and G. Varghese, "Fast address lookups using controller prefix expansion," *ACM Trans. Computer Systems*, vol. 17, pp. 1-40, Feb. 1999.

[9] Yeim-Kuan Chang and Yung-Chieh "A Fast and Memory Efficient Dynamic IP Lookup Algorithm Based on B-Tree" Lin Department of Computer Science and Information Engineering National Cheng Kung University, 2009 International Conference on Advanced Information Networking and Applications.

[10] Zhuo Huang, David Lin, Jih-Kwon Peir, Shigang Chen, S. M. Iftekharul Alam, Fast Routing Table Lookup Based on Deterministic Multi-hashing, Department of Computer &

Information Science & Engineering, University of Florida Gainesville, FL, 32611, USA, 2010

[11] Weidong Wu Packet, Forwarding Technologies, Auerbach Publications, Boca Raton FL, 2008.

[12] 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.

[13] W. Eatherton, "Full Tree Bit Map," Tesis de Maestría, Washington University, 1999.

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

[15] P. Crescenzi, L. Dardini, R. Grossi, "IP Address Lookup Made Fast and Simple," 7th Annual European Symposium on Algorithms ESA'99, also as Technical Report TR-99-01 Università di Pisa.