Genetic Algorithm Based Prefix Partitioning for Memory-Efficient IPv4/6 Lookup

F. Sánchez, M. Ruiz, and C. Jalpa

***Abstract*—IP address lookup speed in core routers has a significant impact on service quality in communications in the Internet. The rapid growth of traffic on the Internet and the development of communication links working at speeds of several gigabits per second make IP lookup a challenge for IP routers. There are several IP lookup schemes based on counts of set bits of a bit-vector that encode the routing information in structures of a constant size that guarantee fast IP lookups. However, these schemes have exponential memory complexity and then prefix partitioning schemes are used to save memory consumption, performing the IP lookup in stages. In this work, we propose a prefix-partitioning scheme that optimizes the available memory usage in IP lookups in accordance with the prefix distribution in forwarding tables and an IP lookup scheme based in a precalculated count of set bits of a bit-vector. A complete IPv4/6 lookup has an instruction cost of two memory accesses in the best case and twice the number of stages in the worst case. The proposed prefix-partitioning scheme is based on heuristic search using genetic algorithms to find an optimal partitioning. Our proposal can be implemented in general purpose hardware for IPv4/6 lookups.**

***Index Terms*—Prefix Partitioning, Genetic Algorithms, Heuristic Search, IP lookup, Longest Prefix Matching, Bit-vector encoding.**

I. Introducción

En Internet los dispositivos de encaminamiento o "routers" tienen como finalidad reexpedir los paquetes de información que llegan a ellos para acercarlos a su destino. Estos dispositivos contienen una base de datos a la que se le denomina tabla de encaminamiento. Dicha tabla incluye información acerca de las redes de computadoras que el dispositivo puede alcanzar en forma de prefijos de red. Un prefijo es una cadena de bits que representa al intervalo de direcciones IP cuyos bits más significativos coinciden con el prefijo. En la tabla también se almacenan los datos acerca del siguiente dispositivo (*Next Hop Information*, NHI) que se encuentra en la ruta correspondiente. Para realizar el proceso de encaminamiento, el dispositivo busca en su tabla el prefijo que tenga el mayor número de bits coincidentes con la dirección IP de destino del paquete que está reenviando.

Fidel U. Sánchez, Posgrado en Ciencias y Tecnologías de la Información, Universidad Autónoma Metropolitana, CDMX, fusa@xanum.uam.mx.

Miguel. A. Ruiz, Departamento de Ingeniería Eléctrica, Universidad Autónoma Metropolitana, CDMX, mars@xanum.uam.mx.

César Jalpa V, Departamento de Ingeniería Eléctrica, Universidad Autónoma Metropolitana, CDMX, cjv@xanum.uam.mx.

A este proceso se le conoce como búsqueda del prefijo con mayor coincidencia (*Longest Prefix Matching*, LPM) o simplemente búsqueda IP [1].

El crecimiento de Internet representa un gran desafío para los algoritmos de búsqueda LPM, ya que no solamente hay un aumento constante del tamaño de las tablas de encaminamiento, sino que ahora se cuenta con enlaces de comunicación de hasta 100 Gbps [2]. Además, la aparición del protocolo IPv6 [3], cuya adopción se está dando rápidamente, ha extendido el tamaño de las direcciones de 32 bits, utilizado por IPv4, a 128 bits. Los dispositivos de encaminamiento deben realizar varias decenas de millones de búsquedas IP por cada segundo (*Million Lookups Per Second*, MLPS), para así evitar convertirse en cuellos de botella.

La exigencia de velocidad requiere que los algoritmos de búsqueda LPM realicen un uso eficiente de la memoria. Por un lado, está el hecho de que al aumentar el tamaño de las direcciones y el número prefijos aumentará también el tamaño de las tablas de encaminamiento. Por otra parte, se tiene que, en los equipos modernos, la latencia de las operaciones aritmético-lógicas es despreciable en comparación con la latencia de los accesos a memoria [4]. Así que el factor determinante en el desempeño de un algoritmo de búsqueda LPM es la cantidad de accesos a memoria más que las operaciones aritmético-lógicas. Las tecnologías de las memorias son más costosas mientras menores son las latencias, por lo cual, la cantidad de memoria de alta velocidad disponible para los algoritmos es bastante limitada.

Una estructura de datos utilizada para representar y realizar la búsqueda de los prefijos de una tabla de encaminamiento es el *trie* binario [5]. Para la representación, los nodos se asocian con los prefijos que llevan hasta ellos recorriendo el *trie* desde la raíz considerando, uno tras otro, el valor de los bits que forman al prefijo: si el bit es un cero se desciende al hijo de la izquierda y si es un uno hacia el de la derecha. La búsqueda del prefijo de mayor coincidencia se realiza básicamente de la misma manera, pero de acuerdo ahora al valor de los bits de la dirección IP de destino del paquete. Aunque este esquema de representación y búsqueda es bastante sencillo tiene como principal desventaja que el tiempo de búsqueda es proporcional a la profundidad del *trie*, es decir, a la longitud de los prefijos. Además de que, sus requerimientos de memoria son proporcionales al número de prefijos en la tabla de encaminamiento. A pesar de estas desventajas, diversos esquemas de búsqueda utilizan esta estructura, al menos para explicar las ideas en las que se fundamentan o con variaciones que permiten reducir la memoria necesaria y la profundidad del

trie. Una de las variaciones que es más recurrente se puede visualizar como el equivalente de partir el *trie* en varios niveles y realizar la búsqueda por etapas en cada uno de esos niveles. La partición del *trie* posibilita la reducción de la memoria necesaria y permite por consiguiente mejorar el desempeño de las búsquedas.

En este contexto, la principal aportación de este trabajo consiste en un esquema que determina mediante búsqueda heurística (algoritmos genéticos) los niveles de partición en el *trie* binario que optimizan la cantidad de memoria requerida para la estructura de datos que representa a los prefijos. Aunado a esto proponemos un esquema de búsqueda LPM basado en conteos precalculados de valores 1 en un vector de bits. El tamaño de dicho vector es independiente a la distribución de prefijos y proporcional a la altura de los *subtries* en cada nivel del *trie* segmentado. Así la búsqueda LPM puede reducirse incluso en un solo par de accesos a memoria para determinar el NHI y por tanto la ruta correspondiente del paquete.

II. Trabajo Relacionado

Existen diferentes esquemas de búsqueda LPM como los descritos en [6], [7], [8], [9], [10], [11] y [12] que utilizan un *trie* binario como estructura principal de su algoritmo. Otras propuestas utilizan al *trie* binario como un auxiliar, únicamente para expresar la idea principal de su algoritmo como el esquema de búsqueda LPM para IPv4 que se propone en [13]. Este esquema, llamado *Lulea*, tiene una latencia de búsqueda muy baja, pocos requerimientos de memoria y tres etapas de búsqueda en los niveles 16, 24 y 32 del *trie* binario. En la primera etapa se codifica un vector de bits en tres contadores de tamaño independiente a la distribución de prefijos. Otros esquemas de búsqueda LPM para IPv4 consideran *tries* de diferentes profundidades como *treebitmap* [14] donde los *tries* de cada etapa tienen una profundidad de 3, esto para reducir el tamaño de la memoria requerida.

Algunos ejemplos de búsqueda LPM que utilizan un *trie* binario en IPv6 es *flashtrie* [15] o símiles como [16] y [17] que realizan búsquedas por etapas. Sin embargo, entre más etapas, mayor es la cantidad de accesos a memoria en cada búsqueda. En los esquemas de búsqueda LPM para IPv6 es más evidente la necesidad de optimizar el número de etapas, la elección de niveles y donde serán colocadas las etapas de cada búsqueda puesto que existe una mayor necesidad de minimizar la memoria requerida que en esquemas de búsqueda en IPv4.

La propuesta hecha por [18] ofrece una optimización en memoria por medio de particionamiento del *trie* binario en un número fijo de etapas y manteniendo constante la complejidad en instrucciones de la búsqueda LPM. La optimización se logra al comprobar el costo de cada una de las posibles combinaciones en las que se puede partir un *trie* binario para un número fijo de etapas. El esquema utiliza programación dinámica ofreciendo una optimización del tiempo de ejecución para determinar la mejor partición del *trie*, sin embargo, ante la ocurrencia de una nueva actualización en la información de encaminamiento, la distribución de prefijos cambia en el *trie* y por tanto el algoritmo tiene que empezar desde el principio una vez más.

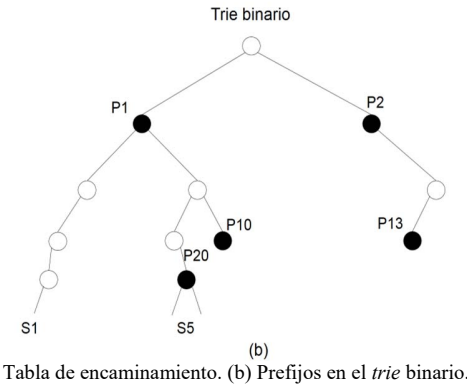
III. Búsqueda con vectores de bits

La Fig. 1(a) ilustra, como ejemplo, un fragmento de una tabla de encaminamiento con prefijos cuya máxima longitud es de 4 bits. Los prefijos de mayor longitud no se muestran. En la Fig. 1(b) se puede observar el *trie* binario correspondiente a dicho fragmento, S1 y S5 son los *subtries* donde se encuentran los prefijos de longitud mayor a 4 bits. En general los prefijos de red no son disjuntos, es decir, un prefijo puede ser prefijo de otro prefijo, tal como se muestra en la Fig. 1, donde el prefijo P2 es prefijo del P13. Si todos los prefijos fueran disjuntos, todos estarían en las hojas del *trie* y el resultado de una búsqueda LPM se tendría siempre al final del recorrido.

Tabla de encaminamiento

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

(a)



(b)

Fig. 1. (a) Tabla de encaminamiento. (b) Prefijos en el *trie* binario.

Una manera de obtener prefijos disjuntos es utilizando la técnica conocida como *Leaf-Pushing* [19] que básicamente consiste en eliminar los prefijos de los nodos internos del *trie*, remplazándolos por nuevos prefijos equivalentes ubicados en las hojas. Cuando los prefijos de longitud menor de 4 bits, ilustrados en la Fig. 1(a), se trasladan a las hojas, utilizando *leaf-pushing*, el fragmento de tabla resultante es el que se muestra en la fig. 2(a). En adelante nos referiremos a esta tabla como tabla extendida de prefijos disjuntos.

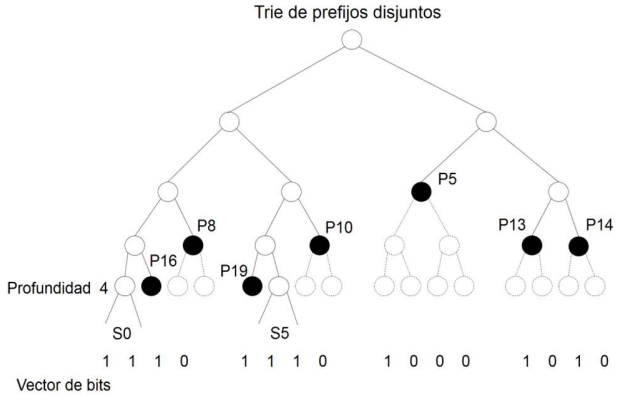
Una búsqueda LPM en una tabla de prefijos disjuntos se reduce entonces a determinar a qué intervalo de los cubiertos por cada prefijo corresponde la dirección IP. Encontrar el intervalo correspondiente a una dirección IP se puede lograr con la ayuda de una estructura de datos llamada *vector de bits*. Un vector de bits es un arreglo cuyas entradas son bits y hay una entrada por cada dirección IP posible. Por ejemplo, para IPv4 se requeriría un vector de 2^{32} entradas y un prefijo de l bits de longitud cubre un intervalo de 2^{32-l} direcciones. Para

identificar los intervalos disjuntos en el vector de bits, se coloca un valor de 1 en la primera entrada del intervalo de direcciones que abarca un prefijo, en las entradas restantes se colocan valores de 0. La búsqueda LPM se realiza con el vector de bits y la tabla extendida de prefijos disjuntos en la que los prefijos siguen el mismo orden que los intervalos correspondientes en el vector de bits de izquierda a derecha. La Fig. 2(b) muestra el vector de bits correspondiente con la tabla de prefijos disjuntos de la Fig. 2(a) si las direcciones fueran de tan solo 4 bits (tratando a S0 y S5 como prefijos de red).

Tabla de apuntadores/NHI

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

(a)



(b)

Fig. 2. (a) Tabla extendida de prefijos disjuntos. (b) *Trie* binario de prefijos disjuntos y vector de bits correspondiente.

Para la búsqueda LPM primero se utiliza la dirección IP como índice en el vector de bits, luego se cuenta el número de bits puestos en 1 desde el extremo izquierdo del vector hasta la posición indexada por la dirección y para finalizar, el resultado de la cuenta de 1*s* se toma como índice en la tabla extendida para obtener la NHI que corresponde a la dirección (ver Fig. 3).

Este esquema es muy simple pero el tiempo necesario para realizar el conteo, depende de la latencia de acceso a memoria y del valor numérico de la dirección IP. En el peor de los casos serían necesarios 2^{32} accesos a memoria para el valor numérico más grande (en el caso de IPv4). Para evitar tener que recontar los bits del vector puestos en 1 en cada búsqueda, en vez del vector de bits, se puede utilizar un arreglo con las cuentas de 1 precomputadas y el valor numérico de la dirección ahora indexa este arreglo en vez de indexar el vector de bits (Fig. 4). Así con solamente 2 accesos a memoria, uno para el arreglo contador y otro para la tabla extendida se completa una búsqueda LPM. Con este esquema, para un *trie* de profundidad K, se requieren $2^K \times K$ bits de memoria además de la necesaria para almacenar la tabla extendida. Entonces para direcciones de 32 bits como en IPv4, $K = 32$, para el arreglo contador se necesitan 16 GBytes y para IPv6, $K = 128$, esta cantidad crece de forma explosiva.

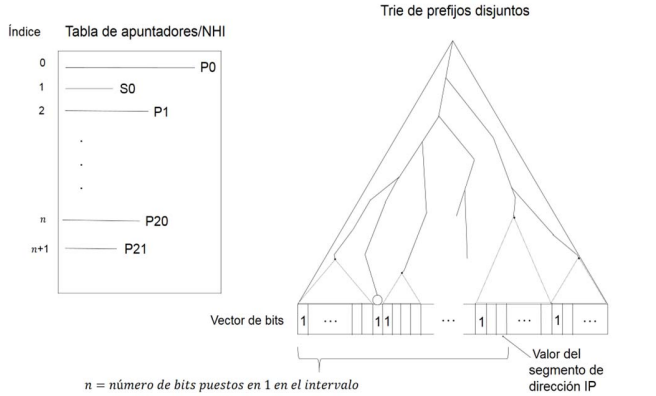


Fig. 3. Relación entre vector de bits y tabla extendida de prefijos disjuntos para realizar una búsqueda LPM.

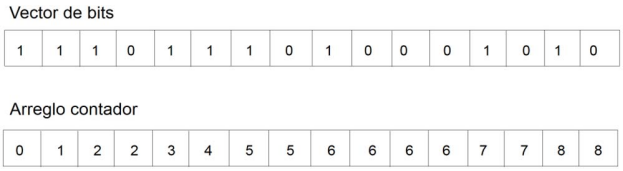


Fig. 4. Vector de bits con su correspondiente arreglo contador.

IV. Particionamiento Eficiente del *Trie* Binario

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

La Fig. 5 ilustra las estructuras de datos del esquema completo de búsqueda LPM en IPv4 o IPv6 dividido en L etapas. Esencialmente una búsqueda LPM consiste en dividir a la dirección IP destino del paquete en L segmentos de longitud acorde al número y tamaño de cada etapa definida en el esquema. Cada segmento de dirección indexa a su respectivo arreglo contador de bits y este a su vez indexa a la tabla extendida de prefijos disjuntos y por ende al NHI correspondiente.

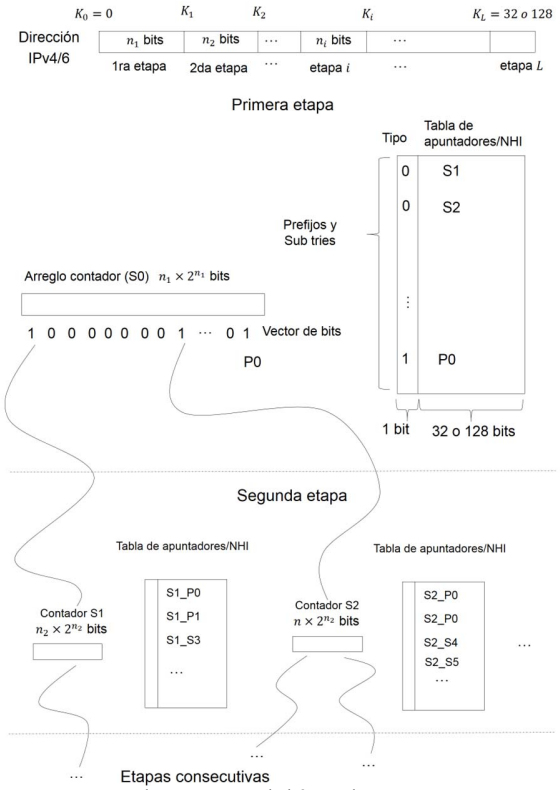


Fig. 5. Esquema completo en etapas de búsqueda IP.

En la Fig. 5 se puede apreciar con detalle la forma y tamaño de los arreglos contadores y las tablas extendidas que se utilizarían en una búsqueda. Un arreglo contador en la etapa i tiene un costo en bits igual a $2^{n_i} \times n_i$ donde n_i es la longitud del $i-ésimo$ segmento de dirección IP ($n_i = K_i - K_{i-1}$). Una tabla extendida en la etapa i contiene prefijos disjuntos que tienen una longitud de K_{i-1} y hasta de K_i bits. Cada entrada de la tabla contiene un prefijo y un campo de 32 o 128 (IPv4 o bien IPv6) bits que identifican al NHI o bien a un apuntador a la siguiente etapa de la búsqueda distinguidas por una marca de un bit. La cantidad total de memoria requerida para codificar de esta manera la información de encaminamiento es la suma de la memoria necesaria para cada uno de los arreglos contadores junto con sus respectivas tablas extendidas. El costo en memoria depende entonces del número de etapas y de los niveles en el *trie* donde se ubica cada etapa. Determinar el número y ubicación de las etapas para minimizar el costo en memoria es un problema combinatorio. Si un *trie* de profundidad K se divide en L etapas de búsqueda $\{K_0 = 0, K_1, K_2, \dots, K_L = K\}$, el número total de combinaciones sin repetición es $\frac{K!}{(L-1)!\times(K-L-1)!}$ donde, L puede tomar cualquier valor entre 0 y K. En el caso de IPv4, K tiene un valor de 32 mientras que en IPv6 K tiene un valor de 128. Entonces, por ejemplo, si dividimos la búsqueda LPM en 10 etapas con $K = 128$ (IPv6), el número de combinaciones es de alrededor de 19 billones. Para algunas de estas combinaciones la cantidad de memoria necesaria para las tablas extendidas y los arreglos contadores será la mínima necesaria. Probar todas estas combinaciones con un algoritmo determinístico para encontrar la óptima, podría consumir mucho tiempo. En este trabajo proponemos un proceso de exploración heurística mediante algoritmos genéticos para determinar la ubicación de las etapas (los niveles) en el *trie* que optimicen la cantidad de memoria requerida en un tiempo mucho menor que un algoritmo determinístico y que aventaje a [18] en el caso de actualización como se explica en seguida.

A. Algoritmos Genéticos

Los procesos evolutivos ocurren cuando hay una población de individuos que son capaces de reproducirse y existe alguna diferencia entre ellos [20]. El ciclo de la evolución comienza cuando los individuos de una población con mejores capacidades de adaptación sobreviven al proceso de selección natural, mientras que los individuos más débiles desaparecen. El ciclo continua cuando parejas de individuos se reproducen heredando sus aptitudes a sus descendientes pero el mecanismo de mutación hace posible la existencia de descendientes con aptitudes diferentes a las de sus ancestros. Los descendientes, resultado de un ciclo de la evolución remplazan a sus ascendentes para volver a comenzar con un nuevo ciclo. Después de haberse repetido muchas veces el ciclo de la evolución, la población resultante tiene mejores cualidades de adaptación que sus ancestros originales.

Los algoritmos genéticos modelan a los procesos evolutivos y están basados en poblaciones cuyos elementos representan soluciones a distintos problemas de optimización que suelen ser difíciles de tratar [21]. Un ejemplo de su utilidad en redes de comunicaciones esta descrita en [22]. El enfoque de algoritmos genéticos que se propone en este trabajo es adecuado por lo siguiente. Las tablas de encaminamiento IPv4/6 son actualizadas constantemente en intervalos pequeños de tiempo ya sea por cambios de NHI, por inserción-eliminación de prefijos de red e inclusive por disponibilidad de subredes. En una actualización, la distribución de prefijos en el *trie* cambia, pero no de forma radical puesto que la topología de Internet no cambia bruscamente de un momento a otro; esto significa que una solución que optimicé la memoria requerida para la búsqueda LPM en un momento dado no será extremadamente diferente de la solución óptima después de la actualización. Los algoritmos genéticos permiten hacer uso de una serie de soluciones que en su momento fueron óptimas para usarlas como punto de partida después de que la tabla de encaminamiento haya sido actualizada. Por otro lado, como se verá en seguida, el mapeo entre los niveles del *trie* y una posible solución es directo lo que implica una mayor eficiencia en instrucciones del algoritmo genético. Puesto que el costo en memoria por codificar un *subtrie* en un arreglo contador es constante, la función de aptitud del genético se reduce a una simple formula.

Desde el punto de vista biológico, un cromosoma es una estructura que contiene información genética de un individuo. La evolución es un proceso que opera sobre los cromosomas y no sobre los individuos que ellos codifican. Nosotros representamos a un cromosoma como una secuencia de bits que contiene una posible solución al problema de particionamiento. La Fig. 6(a) ilustra como un individuo o cromosoma consiste en un arreglo V_c de $K+1$ bits de los cuales existen L bits con un valor de 1 que representan los niveles que forman las particiones del *trie* de profundidad K. Dicho de otra manera, un cromosoma es una posible solución válida (válida más no necesariamente óptima) que se representa con un arreglo de

bits, donde un bit puesto en uno en su entrada q representa una etapa de búsqueda en el nivel q del *trie* binario. El mapeo entre un cromosoma y una partición de niveles en el *trie* es directo: los bits del cromosoma puestos en uno representan las etapas de búsqueda.

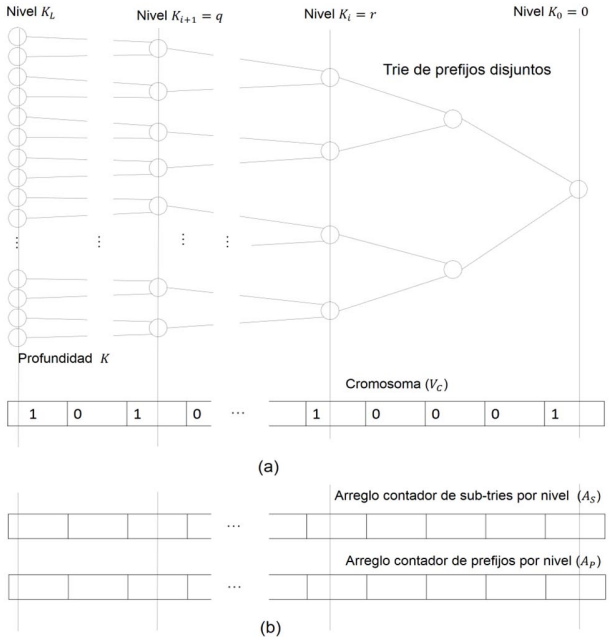


Fig. 6. (a) Niveles en el *trie* codificados en un cromosoma. (b) Arreglos que guardan la distribución de prefijos y *subtries* por nivel.

Una etapa está delimitada con un 1 en el nivel K_i y el nivel K_{i-1} del *trie* y se le asigna un costo en memoria que es igual a la cantidad de memoria necesaria para todos los arreglos contadores correspondientes a los *subtries* generados en la profundidad K_{i-1} junto con sus tablas extendidas. Para el cálculo del costo total en memoria de un cromosoma hacemos el uso de un arreglo A_p que contiene en la posición j la cuenta de los prefijos que existen en el nivel j y un arreglo A_s que tiene la cuenta del número de *subtries* que existen en cada nivel j (Ver la Fig. 6(b)). Los arreglos A_p y A_s tienen tantas entradas como niveles tenga el *trie*. La ecuación (1) expresa el costo por partir el *trie* binario desde el nivel $K_i = r$ hasta el nivel $K_{i+1} = q$ utilizando los arreglos A_s y A_p. La ecuación consiste en el número de *subtries* generados en el nivel r ($\mathrm{A_s}[r]$) y de cada uno de sus arreglos contadores de costo $(q-r)\times 2^{q-p}$. También se considera el costo de las tablas extendidas que consiste en la sumatoria de los prefijos entre los niveles r y q más el número de *subtries* que se generan en la siguiente etapa ($\mathrm{A_s}[q]$) todo esto multiplicado por el tamaño del NHI o el apuntador ($|\mathrm{IP}|$). Finalmente, el costo de un cromosoma está dado por los $L+1$ niveles en el *trie* que forman a las particiones y sus costos $C_{r,q}$ correspondientes.

$$C_{r,q} = (A_s[r]\times(q-r)\times 2^{q-p}) + \left(\left(A_s[q] + \sum_{i=p}^{q} A_p[i]\right)\times |IP|\right) \quad (1)$$

La Fig. 7 muestra un ejemplo de uno de los ciclos del proceso evolutivo. Suponemos una búsqueda de cinco etapas. La primera etapa comienza siempre en el nivel cero y la última termina siempre en el nivel K así que en cada cromosoma V_i deben existir cuatro entradas (además de la primera y la última) con valor 1. La Fig. 7(a) muestra la población aleatoria o bien previamente optimizada. La Fig. 7(b) ilustra el proceso de selección que se lleva a cabo mediante torneos. En este ejemplo se enfrentan el cromosoma V_1 contra el cromosoma V_2. En este caso el costo de V_1 es menor que el costo de V_2 por lo que el vencedor es V_1 y por lo tanto el contenido de V_1 se copia en V_2 obteniendo V'_2. La Fig. 7(c) presenta la simulación de la reproducción sexual entre V_1 y V_3. Se elige una posición de cruce aleatoria y se intercambian los valores a la izquierda de dicha posición de ambos cromosomas. Dado que cada cromosoma solo puede tener 4 bits con valor de 1, si al reproducir dos individuos uno adquiere más niveles y otro pierde niveles entonces se equilibran los valores de forma aleatoria. Finalmente, la Fig. 7(d) muestra la simulación de la mutación que permite que individuos en una generación posean características únicas no heredas de sus ancestros. En este ejemplo, del cromosoma V_i se elige un bit puesto en 1 y un bit puesto en 0 de forma aleatoria y se intercambian.

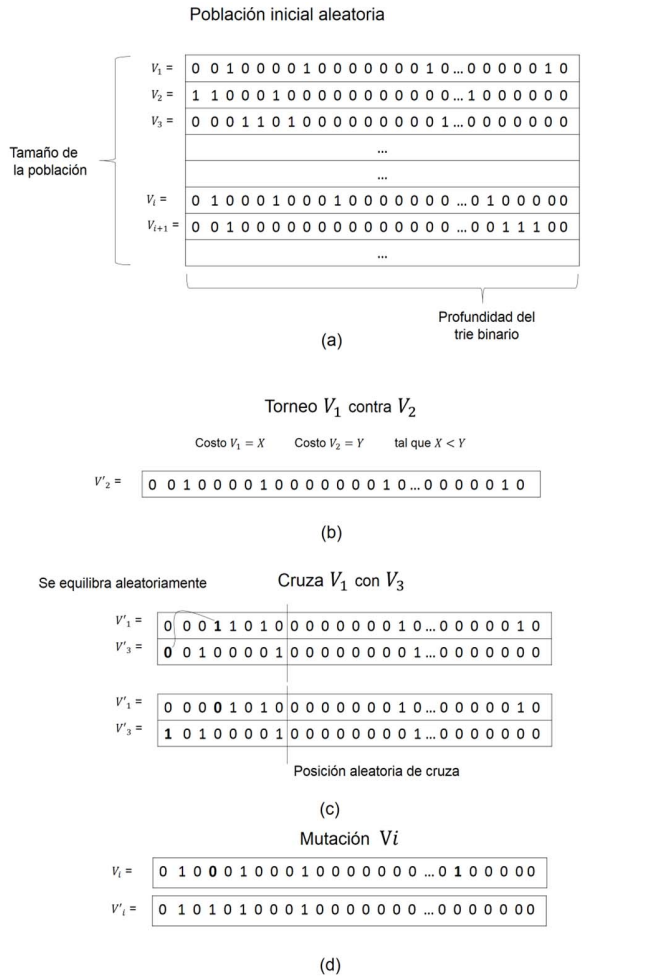


Fig. 7. (a) Población. (b) Supervivencia del más fuerte mediante torneo. (c) Reproducción sexual. (d) Mutación.

V. Implementación y Resultados

A. Implementación

El algoritmo genético fue implementado en un paradigma de programación secuencial mono proceso. Utilizamos una computadora de propósito general, con el fin de demostrar que, con pocos recursos computacionales se puede obtener una solución de la partición del *trie* binario rápidamente. La programación del algoritmo genético se realizó de acuerdo con el ciclo evolutivo de una población inicial aleatoria distribuida uniformemente. Experimentalmente determinamos que con los siguientes parámetros de operación el algoritmo genético funciona correctamente. El tamaño de la población puede ser tan pequeña como 100 individuos. En cada ciclo simulamos la supervivencia del más apto, para ello realizamos la función torneo enfrentando la población en parejas diferentes descartando a los más débiles. Después de este proceso, seleccionamos solamente al 30% de la población de forma aleatoria para reproducirse. Al azar se forman parejas diferentes y se realiza la función de cruce. La mutación de individuos en una nueva generación se realizó con una probabilidad de 0.001%. Cada vez que se completa un ciclo de la evolución se evalúa la aptitud de la población, esto significa que calculamos el promedio del costo de los individuos que conforma a la población. El proceso antes descrito se repite hasta que la aptitud de la población entre una y otra generación tenga una diferencia menor de 0.001%. Una vez terminado el proceso, los niveles del *trie* binario, codificados en el cromosoma de menor costo de la población optimizada son la solución al problema de particionamiento en etapas de la búsqueda LPM.

Cuando la tabla de encaminamiento es actualizada, el proceso del algoritmo genético se repite, pero en lugar de utilizar una población inicial aleatoria, se utiliza la población anterior. Esto permite obtener que al algoritmo converja a la nueva solución más rápidamente que con una población inicial aleatoria. Esta es una ventaja sobre la exploración determinista y por ende en la programación dinámica propuesta en [18].

B. Análisis de Resultados

Para validar el funcionamiento del algoritmo genético utilizamos las tablas de encaminamiento IPv4/6 del enrutador AS6447 [17] correspondiente al 20 de marzo de 2019 y las actualizaciones correspondientes a ese día. Las actualizaciones de las tablas de encaminamiento están contenidas en una base de datos que contiene principalmente a los prefijos de red que serán insertados o eliminados aunados a la fecha y hora en que se produce dicha actualización. La fecha y hora de una actualización en la base de datos utilizada tiene precisión de un segundo. En la tabla de encaminamiento IPv4 encontramos 742,313 prefijos. Después de aplicar *leaf-pushing* obtuvimos 1,082,558 prefijos disjuntos mientras que en la tabla de encaminamiento IPv6 encontramos 69,096 prefijos, después de aplicar *leaf-pushing* obtuvimos 342,912 prefijos disjuntos.

Experimentalmente determinamos que, con más de 5 etapas de búsqueda, en la tabla de encaminamiento IPv4, no se tiene una ganancia significativa en ahorro de memoria. Esto porque mientras haya más niveles en el *trie*, existe una mayor cantidad de apuntadores en las tablas extendidas. Similarmente en IPv6, con más de 18 etapas de búsqueda no encontramos una ganancia significativa en ahorro de memoria requerida.

Para mostrar la importancia de seleccionar los niveles adecuados en el *trie* binario para la tabla IPv4 proponemos un ejemplo con tres etapas de búsqueda LPM y contrastamos los costos en memoria utilizando nuestra propuesta en el primer ciclo de la evolución y después de haber convergido en una solución optimizada. Elegimos al cromosoma {3,10,32} con el costo de 9.1GB de la primera iteración y por el otro lado lo contrastamos con el cromosoma de mayor aptitud después de la evolución que es {19,24,32} y tiene un costo total de 9.5MB. De igual forma, para demostrar la importancia de la posición de los niveles en el *trie* binario en la tabla IPv6 elegimos 18 niveles de búsqueda LPM; en este caso, obtuvimos que el costo en memoria es de 2.2TB para un cromosoma aleatorio en la primera iteración mientras que, el cromosoma de la solución optimizada tiene un costo de 5.1MB.

En nuestra implementación, definimos a un periodo de actualización como el tiempo que transcurre para que la tabla de encaminamiento sea actualizada. En la base de datos utilizada el periodo de actualizaciones es de un segundo; en este caso encontramos desde 0 hasta 3 millares de inserciones-eliminaciones de prefijos de red en un periodo de actualización. La Fig. 8(a) muestra un gráfico correspondiente al tiempo de convergencia del algoritmo genético en 500 periodos de actualización de la tabla de encaminamiento IPv4 con 5 etapas de búsqueda. La Fig. 8(b) muestra los tiempos de convergencia del algoritmo genético en 500 periodos de actualización de la tabla IPv6 cuando la búsqueda LPM es dividida en 18 etapas. Como puede verse en las gráficas, el tiempo en que converge el algoritmo genético en cada periodo de tiempo no es constante debido a que por ejemplo puede haber un número diferente de inserciones-eliminaciones de prefijos en cada periodo de actualización. Nótese también que después de la primera vez que se obtiene una optimización en memoria requerida (el primer valor en tiempo de las gráficas) el algoritmo genético tiene una convergencia más rápida debido a que se parte de soluciones previamente optimizadas.

Actualmente se está migrando al protocolo IPv6 y se espera que el tamaño de las tablas para este protocolo tenga un crecimiento y tamaño parecido a las tablas de encaminamiento para IPv4. Para comprobar la eficacia de nuestra propuesta en situaciones futuras utilizamos *V6Gene* [24] para crear una tabla IPv6 sintética con un mayor número de prefijos utilizando como semilla la tabla IPv6 real utilizada anteriormente; además de considerar los aspectos de implementación en IPv6 como los descritos en [25] y [26]. Inicialmente la tabla sintética tiene 472,331 prefijos, después de aplicar *leaf-pushing* obtuvimos 2,421,136 prefijos disjuntos. Al igual que la tabla IPv6 real determinamos que con más de 18 etapas de búsqueda LPM ya no existe una mayor ganancia en memoria requerida. La Fig. 8(c) muestra los tiempos de convergencia del algoritmo genético en 500 periodos de actualización de la tabla IPv6 sintética; note que dichos tiempos de convergencia tienen un comportamiento similar al de las tablas antes utilizadas.

Para determinar con certeza la calidad de las soluciones del algoritmo genético se programó un algoritmo de fuerza bruta que genera todas las posibles soluciones de partir el *trie* binario en L niveles, (combinaciones si repetición, sección IV) calculando en cada una de ellas su costo dado por la Ecuación

1. Nuestro algoritmo genético obtuvo la solución óptima en todas las simulaciones realizadas.

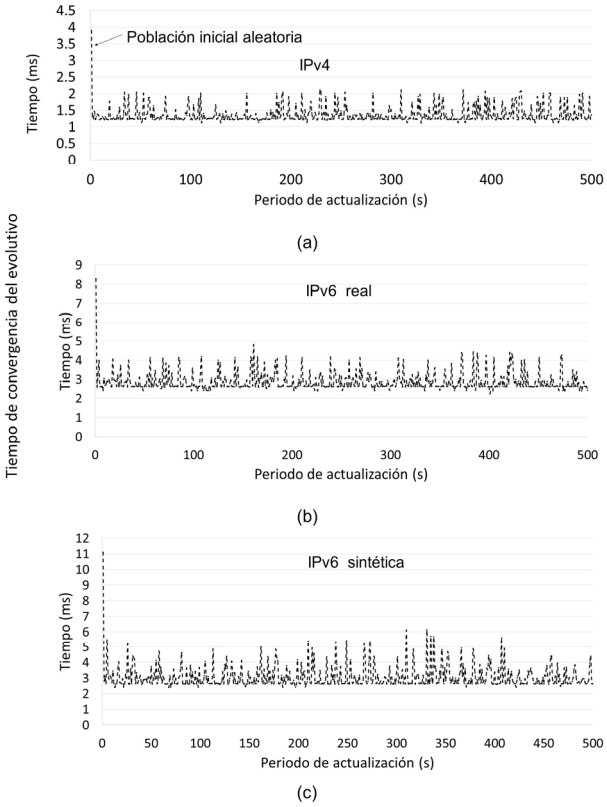


Fig. 8. Tiempo que tarda en converger el algoritmo genético en cada actualización. (a) Tabla IPv4 con 5 etapas de búsqueda LPM. (b) Tabla IPv6 real con 18 etapas de búsqueda LPM (c) Tabla IPv6 sintética con 18 etapas de búsqueda LPM.

Realizamos los experimentos anteriores utilizando tablas IPv4 real, IPv6 real e IPv6 sintética de distintos años, días y horas obteniendo resultados similares. La Fig. 8 muestra que la latencia del algoritmo genético es muy pequeña comparada con la precisión de los periodos de actualización de las tablas utilizadas. Suponiendo casos donde existieran periodos de actualización con una mayor precisión, la capacidad de respuesta del algoritmo genético permite que la precisión de dichos periodos sea inclusive de decenas de milisegundos, precisando que, los algoritmos genéticos pueden ser optimizados en tiempo de ejecución si son programados en paradigmas de programación paralelos. En contraste, la propuesta de particionamiento de búsqueda exhaustiva de niveles óptimos hecha en [18] ofrece una gran optimización en tiempo de ejecución al precomputar de forma recursiva varias posibles soluciones en etapas, sin embargo se tiene que considerar por lo menos una instrucción atómica para calcular el costo de cada combinación de niveles lo que significa que el tiempo de ejecución es directamente proporcional al número total de posibles combinaciones. En el caso de IPv6 ($K = 128$) dividida en $L = 18$ etapas de búsqueda se necesitarían varios billones de instrucciones para determinar la partición óptima cada vez que la tabla de encaminamiento sea actualizada. Lo cual tomaría un tiempo de ejecución del orden de segundos, lo que contrasta fuertemente con nuestra propuesta

La Tabla I muestra los rangos de costo en memoria requerida correspondiente a los periodos de actualización de la Fig. 8. Los resultados presentados son los costos totales en memoria por contener tablas extendidas de prefijos y arreglos contadores dados por la Ecuación 1 y las tablas de encaminamiento. Las memorias actuales de rápido acceso superan los 50MB lo que implica que tanto los contadores como las tablas extendidas en IPv4/6 reales e IPv6 sintética pueden ser contenidos en ellas. La rapidez de una búsqueda LPM dependerá de la latencia de la memoria que contenga a los contadores y tablas extendidas. En el mejor de los casos el costo en instrucciones del esquema es dos accesos a memoria y en el peor de los casos es dos veces la cantidad de etapas en que se divida la búsqueda LPM.

TABLA I
RANGOS DE COSTOS TOTALES DE MEMORIA REQUERIDA PARA LAS TABLAS DE ENCAMINAMIENTO UTILIZADAS DURANTE 500 SEGUNDOS DE ACTUALIZACIÓN

Tabla	Costo mínimo (MB)	Costo Máximo (MB)
IPv4 real	5.5	5.7
IPv6 real	4.57	4.63
IPv6 sintética	52.1	52.5

VI. CONCLUSIONES

Nuestra propuesta consiste en la optimización de la cantidad de memoria utilizada en un esquema de búsqueda LPM basada en exploración heurística mientras que otras soluciones utilizan métodos exhaustivos como la programación dinámica. La principal ventaja que encontramos en nuestra propuesta es la baja latencia y la alta calidad de soluciones de la exploración heurística, además de un mapeo directo entre estructuras de datos del esquema de búsqueda LPM y el algoritmo genético. Mientras que la desventaja evidente de la exploración heurística es no poder ofrecer una solución que garantice ser óptima. Sin embargo, nuestra propuesta demuestra ser eficiente en la optimización de memoria requerida en distintos modelos de direccionamiento IPv4 e IPv6, así como modelos sintéticos IPv6 con mayores requerimientos. La latencia del algoritmo genético es muy pequeña en comparación con la precisión de los periodos de actualización de la base de datos utilizada; además como trabajo futuro, se podría mejorar su desempeño de ejecución si su implementación se basara en programación paralela. Determinamos que cuando se utiliza una partición optimizada las estructuras de datos utilizadas pueden ser contenidas en memorias de rápido acceso tanto para tablas IPv4 reales así como para tablas IPv6 reales y sintéticas.

El escenario propuesto para validar nuestra propuesta fueron tablas de encaminamiento IPv6 e IPv4 actuales de un router público global y en el caso de IPv6 creamos una tabla sintética para simular el futuro de IPv6. El esquema de búsqueda LPM propuesto garantiza un costo en instrucciones de dos accesos a memoria en el mejor de los casos mientras que en el peor de los casos el costo es de dos veces la cantidad de etapas en que se divida la búsqueda.

AGRADECIMIENTOS

Agradecemos a la Universidad Autónoma Metropolitana unidad Iztapalapa y al Consejo Nacional de Ciencia y

Tecnología (CONACyT) por el apoyo recibido. También agradecemos al Dr. Pedro Lara Velázquez y la Dra. Martha Montes de Oca por sus valiosos comentarios para mejorar este artículo.

REFERENCIAS

[1] 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, March/April 2001.

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

[3] Available: https://tools.ietf.org/html/rfc2460

[4] John Hennessy and David Patterson, In Praise of Computer Architecture: A Quantitative Approach. Fourth Edition. Ch. 8.

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

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

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

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

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

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

[11] Tong Yang , Gaogang Xie , Alex X. Liu , Qiaobin Fu, Yanbiao Li, Xiaoming Li, and Laurent Mathy, "Constant IP Lookup With FIB Explosion" IEEE/ACM Transactions on Networking, Vol. 26, No. 2, August 2018.

[12] Anat Bremler-Barr, DavidHay, and Yaron Koral, "CompactDFA: Scalable Pattern Matching Using Longest Prefix Match Solutions" IEEE/ACM Transactions on Networking, Vol. 22, No. 2, April 2014.

[13] 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.

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

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

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

[17] Chinam Kim and Hyukjun Lee, "A High-Bandwidth PCM-Based Memory System for Highly Available IP Routing Table Lookup" IEEE Computer architecture Letters, Vol. 17, No. 2, December 2018.

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

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

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

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

[22] M. T. A. P. Wickramarathna; Nalin Wickramaarachchi, "Transmission Network Planning Using Genetic Algorithm", IEEE/PES Transmission & Distribution Conference and Exposition: Latin America, August 2006.

[23] http://bgp.potaroo.net/index-bgp.html

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

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

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

Fidel Ulises Sánchez Jiménez recibió el título de Ingeniero en electrónica de la Universidad Autónoma Metropolitana campus Iztapalapa en la ciudad México en el año de 2010 y el título de Maestro en Ciencias por la misma universidad en el año 2012. Actualmente es estudiante de doctorado en el área de redes de computadoras en la Universidad Autónoma Metropolitana.

Miguel Ángel Ruiz Sánchez recibió el título de Doctor en Informática de la Université de Nice-Sophia Antipolis en Francia. Miguel Ángel Ruiz Sánchez es actualmente profesor en el Departamento de Ingeniería Eléctrica de la UAM-Iztapalapa, al que se integró en 1995. Él ha sido Coordinador de la Licenciatura en Ingeniería Electrónica de la División de CBI de la UAM-I. Sus temas de interés incluyen: Mecanismos eficientes de reexpedición de paquetes en ruteadores IP, Redes de sensores y sistemas basados en microcontroladores. Entre los cursos que ha impartido se encuentran: Redes de computadoras, Telefonía, Sistemas de comunicación, Diseño de sistemas con microcontroladores, Diseño lógico y Circuitos de conmutación.

César Jalpa Villanueva obtuvo el grado de Doctor en Informática de la Universidad de Niza-Sophia Antipolis, Francia, en el año 2000. En el año 1991, el grado de Maestría en Ingeniería Eléctrica con especialidad en Comunicaciones por el Centro de Investigación y Estudios Avanzados del I.P.N. y en el año de 1988, el título de Ingeniero en Electrónica de la Universidad Autónoma Metropolitana Campus Iztapalapa. Es profesor del Área de Redes y Telecomunicaciones del Departamento de Ingeniería Eléctrica en la Universidad Autónoma Metropolitana Campus Iztapalapa.