PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

OPTIMIZACIÓN DE ENRUTAMIENTO EN REDES IP USANDO ALGORITMO ANTNET CALIBRADO POR UN ALGORITMO GENÉTICO

LUIS ARMANDO GONZÁLEZ COS

TESIS DE GRADO
MAGÍSTER EN INGENIERÍA INFORMÁTICA

(Mayo, 2012)

Pontificia Universidad Católica de Valparaíso
Facultad de Ingeniería
Escuela de Ingeniería Informática

OPTIMIZACIÓN DE ENRUTAMIENTO EN REDES IP USANDO ALGORITMO ANTNET CALIBRADO POR UN ALGORITMO GENÉTICO

LUIS ARMANDO GONZÁLEZ COS

Profesor Guía: **BRODERICK CRAWFORD LABRÍN**

Programa: **Magíster en Ingeniería Informática**

(Mayo, 2012)

Abstract

In this work, we design and implement a TCP/IP network routing algorithm based on AntNet algorithm which is calibrated, in its parameters, using an genetic algorithm, we describe the algorithms used: Antnet and Genetic Algorithms and describe the network routing problematic and the AntNet solution for optimize the traffic in a network, maximizing the throughput. The parameters required for the AntNet algorithm are generated running a Genetic algorithm specially designed in C language, with the appropriate chromosome design. The solution was tested using the classics NSFNET and NTTNET networks in order to measure the results obtained in terms of optimize the throughput across the network. The results obtained show that the implementation of an Antnet calibrated for a genetic algorithm was successful because the value of the metrics generated (throughput and delay) was better than the values of classic algorithms used to test network routing algorithms.

Keywords: Network Routing, AntNet Algorithm, Mobile Agents, Genetic Algorithm

Resumen

En este trabajo se diseña e implementa un algoritmo de ruteo en redes TCP/IP, basado en el algoritmo AntNet, el cual es calibrado, en sus parámetros, utilizando un algoritmo genético. Se realiza una descripción de los algoritmos utilizados, tanto de Antnet, como de Algoritmos Genéticos. Se describe la problemática del ruteo en redes y la solución de AntNet para optimizar el tráfico en una red, maximizando el throughput. Los parámetros requeridos por AntNet son generados mediante un algoritmo genético, implementando en lenguaje C, especialmente diseñado con un esquema de cromosomas que permiten variar diversos parámetros en rangos de valores mínimos y máximos. El algoritmo se probó exitosamente en una red Experimental pequeña y se aplicó a las redes clásicas de benchmarking de algoritmos de enrutamiento, como son NSFNET y NTTNET. Los resultados obtenidos fueron comparados con los valores de la aplicación de otros algoritmos tradicionales a las redes de benchmarking y en los casos estudiados se obtuvieron mejoras en las métricas de comparación (throughput y demora promedio de los paquetes) respecto de los algoritmos clásicos de enrutamiento e incluso sobre la versión original de los autores de Antnet, lo cual indica que la calibración multinivel de Antnet por medio de un algoritmo genético fue exitosa.

Keywords: Network Routing, AntNet Algorithm, Mobile Agents, Genetic Algorithm

INDICE

1. Introducción

Actualmente las redes IP han tenido un crecimiento exponencial en la cantidad de nodos y tráfico entre ellos, sean estos computadores o hosts y dispositivos de enrutamiento, el protocolo TCP/IP es el protocolo estándar utilizado por todos los mensajes transmitidos a través de la red. Junto al incremento en el tamaño de la red, han surgido las denominadas Redes Sociales que han incrementado la demanda por ancho de banda y confiabilidad de las redes, ya que la transmisión de datos y *streaming* de audio y vídeo requiere de una infraestructura y confiabilidad que permitan al usuario final contar con una adecuada calidad del servicio.

Al fluir los mensajes por la red, por la naturaleza propia del protocolo TCP/IP, es probable que se produzcan retardos, pérdidas de paquetes, o degradación de la tasa de rendimiento de la red, esto se debe a que el protocolo no dispone de mecanismos de control o gestión para controlar esos eventos ya que TCP/IP no fue diseñado para ofrecer calidad del servicio, es por ello que se hace necesario aplicar técnicas de optimización al tráfico de paquetes en redes dinámicas para reducir la congestión aumentando la eficiencia en el rendimiento de la red utilizando de mejor forma la capacidad de los enlaces que la forman.

Las redes IP son dinámicas con frecuentes actualizaciones de la topología de la red, es decir se caracterizan porque los nodos componentes de la misma pueden incorporarse de manera aleatoria, de modo que los restantes nodos están dinámicamente analizando las rutas por donde despachar los mensajes que fluyen en la red.

La figura 1.1 representa una red IP típica con sus elementos componentes: Los nodos (routers, hosts) y los enlaces o líneas físicas de transmisión de paquetes.

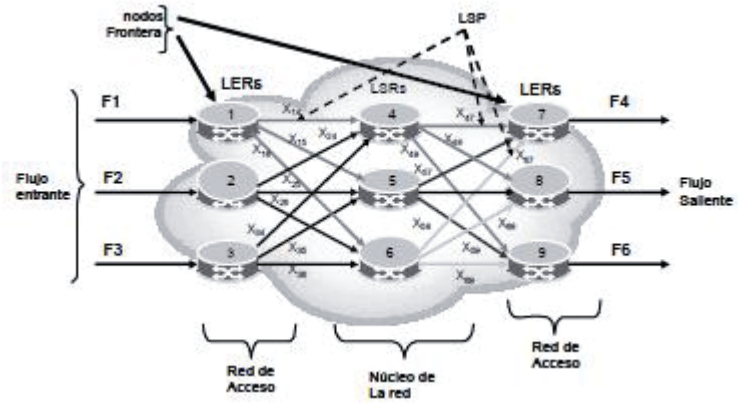

Figura 1-1: Gráfico de una Red IP

Los routers son dispositivos que conectan los diferentes segmentos que conforman una red IP, seleccionando un camino para que los paquetes lleguen a su dirección de destino. Los routers actúan en la capa de red del modelo OSI por lo cual tienen acceso a la información de direccionamiento de los nodos, esta información es utilizada para determinar el camino óptimo entre el punto de origen y el punto de destino de un mensaje. Además, los routers guardan y dirigen los paquetes a los segmentos interconectados de la red sin preocuparse por su topología. Los routers utilizan un algoritmo de enrutamiento para decidir el camino de los datos transmitidos por la red, basándose en la información contenida en una tabla de rutas y en el encabezado del paquete donde se ubica la dirección fuente y la de destino. La tabla de rutas lista las direcciones la red y los caminos entre los nodos. El algoritmo de enrutamiento, entonces, compara la información del encabezado y las direcciones de red en la tabla de rutas para determinar el mejor camino hacia el cual enviar el paquete de información.

De acuerdo a [1], una red de telecomunicaciones, como lo es una red IP, puede ser modelada por un conjunto de nodos N, un conjunto de arcos o enlaces entre los nodos L y los costos o distancias entre los arcos $d_{i,j}$. El problema de enrutamiento consiste en encontrar los caminos de costo mínimo entre todos los pares de nodos de la red. Si los costos $d_{i,j}$ fuesen fijos, entonces el problema se reduce a un problema de camino mínimo entre nodos, el cual puede resolverse por métodos tradicionales basados en el algoritmo de Dijkstra [2] u otros de similar complejidad polinomial. Sin embargo en una red IP los costos no son fijos, ya que las aplicaciones varían sus requerimientos de ancho de banda produciendo variaciones en los costos de los arcos entre dos nodos, o bien cambios dinámicos en la topología de la red provocan modificaciones en el número de ellos, con los consecuentes cambios en el número de arcos y sus costos asociados. Por la naturaleza aleatoria de los cambios en una red de telecomunicaciones, el problema de enrutamiento se transforma en un problema de complejidad exponencial.

Si el enrutamiento entre nodos fuera un problema determinístico, es decir un tipo de problemas donde las mismas entradas, generan siempre las mismas salidas, podría ser resuelto mediante procedimientos analíticos conocidos, pero dado el comportamiento dinámico de las redes IP, se requiere de técnicas heurísticas las cuales buscan soluciones en la región de soluciones factibles, mediante estrategias que supuestamente conducen al óptimo deseado, pero sin garantía de lograrlo, por el cual se obtienen soluciones aproximadas o cercanas al óptimo.

De acuerdo a [3] en una red IP la estrategia global de enrutamiento se genera como el resultado colectivo de decisiones descentralizadas, realizadas por los nodos individuales (routers) basadas en observaciones locales de los individuos, cuya estructura típica de funcionamiento se puede modelar, apropiadamente, bajo el paradigma de inteligencia de enjambre (Swarm Intelligence), sobre el cual se construye el algoritmo de optimización por Colonia de Hormigas (ACO: Ant Colony Optimization). ACO se ha utilizado para la resolución de distintos problemas de optimización, como el *TSP* o Problema del Vendedor Viajero, *Set Covering Problem*, etc. La aplicación de ACO

para resolver el problema de enrutamiento en redes se denomina AntNet, presentado por [4]. AntNet cuenta con parámetros que regulan su procesamiento, estos son el tamaño del paquete en la red IP, tiempo de simulación, Importancia relativa de la heurística en la feromona y el tiempo de proceso del paquete en cada nodo.

Para implementar una solución al problema de enrutamiento en redes, en este proyecto, se implementa una metaheurística basada en el Algoritmo ACO propuesto por [5], cuyos parámetros son regulados por un algoritmo genético. Según [6] las metaheurísticas son una clase de métodos aproximados diseñados para resolver problemas complejos de optimización, donde las heurísticas clásicas y métodos de optimización fallan para obtener soluciones efectivas y eficientes. Una metaheurística se define formalmente como un proceso iterativo de generación, el cual guía una heurística subordinada combinando inteligentemente diferentes conceptos para explorar y explotar el espacio de búsqueda, las estrategias aprendidas se utilizan para estructurar la información, de modo de encontrar eficientemente soluciones cercanas al óptimo.

En ACO, las hormigas presentan una convivencia social, que permite facilitar la realización de tareas con un objetivo común mediante el trabajo coordinado, que se obtiene al seguir los rastros de feromonas, búsqueda de caminos en forma aleatoria, o reconociendo imágenes que se encuentran en su memoria de corto plazo. Las hormigas encuentran el camino más corto entre el hormiguero y las fuentes de comida, explotando el rastro de feromona existente en el camino, dejado por las hormigas que ya han pasado por ese punto, y usando cierto conocimiento previo almacenado en su memoria, es decir las hormigas dejan un rastro de feromonas que puede ser detectado por el resto de la colonia, mediante una comunicación indirecta conocida como estigmergía (stigmergy).

De acuerdo a Grassé [7], quien introdujo el término estigmergía como un concepto para describir la comunicación indirecta, mediante modificaciones en el medioambiente, que dejaba la casta trabajadora de una especie de termitas, su definición original fue “Estimulación de trabajadores de acuerdo al rendimiento que han logrado”. Aun cuando Grassé realizó sus estudios sobre termitas, el mismo término estigmergía fue posteriormente utilizado para indicar la comunicación indirecta mediante modificaciones en el medio ambiente realizadas por otros insectos, por ejemplo por hormigas, que se estimulan unas a otras modificando el medio ambiente con rastros de feromonas. La estigmergía es un mecanismo espontáneo, indirecto de coordinación entre los agentes, estimula el funcionamiento de una acción subsecuente, produce estructuras complejas, aparentemente inteligentes, sin la necesidad de planificación, control, o comunicación entre los agentes, genera la colaboración eficiente entre agentes extremadamente simples.

Para ir de un punto a otro, las hormigas llegan a bifurcaciones donde deben escoger entre dos o más rutas alternativas para llegar a su destino, deben decidir por una de las posibles rutas alternativas a seguir. La hormiga

realiza una elección de la ruta, ya sea, en forma aleatoria al haber bajos niveles de feromonas, o bien selecciona la ruta que presenta una diferencia notable en la cantidad de feromonas depositada en cada camino.

Las hormigas se desplazan a una velocidad aproximadamente constante, por lo cual, las que eligen el camino más corto se demoran menos que las otras en recorrer su trayecto desde al origen al destino, esto genera una mayor acumulación de feromonas en el camino más utilizado, es decir las feromonas se acumulan en mayor cantidad en el camino más corto y más utilizado, esto guía a las hormigas a su destino en la forma más rápida. Basada en el rastro de feromonas, las hormigas siguen la ruta cuya probabilidad de selección es de mayor a menor. Un aspecto importante en el comportamiento de las hormigas lo constituye su capacidad limitada para elegir la ruta a seguir, ya que básicamente se reduce al rastreo de las feromonas, por otra parte las feromonas tienen una propiedad de evaporación constante, de manera que al pasar el tiempo el rastro de feromona desaparece.

Para aplicar la metaheurística propuesta, se utilizará un algoritmo genético, cuya forma usual fue descrita por [8], basado en los trabajos preliminares de [9]. Los algoritmos genéticos son procesos estocásticos de búsqueda de soluciones a problemas de optimización, basados en el proceso de selección natural y sobrevivencia de los individuos más aptos de una especie, identificado por Darwin en su clásica teoría del origen de las especies [10].

De acuerdo a [11] los algoritmos genéticos comienzan su proceso de búsqueda con un conjunto inicial de soluciones factibles, generadas aleatoriamente, llamado población inicial. A cada individuo de la población se le denomina cromosoma y representa una solución al problema que se desea optimizar. Cada cromosoma es un *string* de caracteres, generalmente se utilizan strings binarios, los cuales adecuadamente codificados mapean una o mas variable de decisión. Las cromosomas evolucionan a través de sucesivas iteraciones, denominadas generaciones. Durante cada generación las cromosomas son evaluadas usando una función objetivo, denominada *fitness*. Para crear la próxima generación se crean nuevas cromosomas a partir de dos operadores genéticos: i) mezcla de dos cromosomas (padres) de la generación actual usando un operador de cruzamiento para generar cromosomas hijos ii) modificando una cromosoma mediante un operador de mutación. Estos operadores genéticos imitan los procesos de herencia de características de una generación a otra y la selección natural, descrita por Darwin, donde los individuos menos aptos (menor valor de *fitness*) tienen menores valores de ser seleccionados para la siguiente generación. La nueva generación se forma seleccionado aleatoriamente algunos padres e hijos acorde a su valor de *fitness* y rechazando otros de modo de que el tamaño de la población permanezca constante. Las cromosomas que tengan mejores valores de *fitness* tienen una mayor probabilidad de ser elegidas para formar parte de la siguiente generación.

Después de varias generaciones el algoritmo converge a la mejor cromosoma, la cual se espera que represente el óptimo global, o al menos un óptimo local de la solución al problema. Un algoritmo genético tiene parámetros tales como, tamaño de la población, numero de iteraciones u otro criterio de término, probabilidad de selección una cromosoma para aplicar el operador de cruzamiento y la probabilidad de mutación de una cromosoma.

Los algoritmos genéticos difieren de los métodos tradicionales de optimización en los siguientes aspectos [8]:

a) Trabajan con una codificación del conjunto solución, no con las soluciones en sí.
b) El proceso de búsqueda se realiza en una población de soluciones, no sobre soluciones individuales
c) Utiliza la función de *fitness*, como guía en la búsqueda de soluciones, no requiere de conocimiento adicional.
d) Utilizan operadores genéticos, los cuales son reglas de transición probabilísticas, no determinísticas.

En este trabajo, el algoritmo genético será utilizado como una metaheurística cuyo string de cromosomas codificará los valores de los parámetros de AntNet, algoritmo que será ejecutado y retornará el valor del *throughput* obtenido con esos valores en sus parámetros. Por cada llamada a AntNet se creará una instancia de cromosoma donde se almacenará la codificación de los parámetros y el *throughput* obtenido se almacenará como su valor de *fitness*, de modo que cada cromosoma generada para cada población del algoritmo genético involucra una llamada a AntNet para determinar el valor de la función *fitness*, de esta manera al ejecutarse el algoritmo genético por una determinada cantidad de generaciones, se espera que este converja hacia valores de *throughput* que converjan hacia un valor optimo, obteniendo en última instancia los valores de los parámetros de AntNet que generaron ese valor óptimo.

Los resultados que se obtengan con la aplicación de la metaheurística propuesta serán contrastados, con los modelos clásicos de benchmarking para redes IP, basados en los algoritmos tradicionalmente utilizados para el enrutamiento en redes: OSPF, SPF, BF, Q-E, PQ-R y el algoritmo original AntNet, aplicados a las redes NSFNET y NTTNET utilizadas para comparación de algoritmos de enrutamiento según lo presentado por [12].

Para el desarrollo de este trabajo se han definido los siguientes objetivos:

1.1. Objetivo general

Optimizar el enrutamiento en una red IP maximizando su throughput, mediante el desarrollo e implementación de un algoritmo AntNet con parámetros calibrados por un algoritmo genético.

1.2. Objetivos específicos

- Diseñar e implementar un algoritmo AntNet que permita optimizar el enrutamiento en una red IP maximizando su throughput.
- Diseñar e implementar un algoritmo genético, que aplique una metaheurística de multinivel que calibre un algoritmo AntNet.
- Realizar pruebas experimentales de simulación que permitan calibrar el modelo.
- Determinar la calidad de las soluciones encontradas en relación a los modelos clásicos de benchmarking para redes IP.

1.3. Alcances y Límites

Este trabajo se enmarca en el contexto académico del Programa de Magister en Ingeniería Informática, realizando un aporte teórico en el ámbito de las metaheurísticas, aplicada al caso del algoritmo AntNet, con sus parámetros y operadores generales, calibrados por un algoritmo genético con sus propios parámetros y operadores, basados en valores recomendados por estudios previos. Se excluye de esta investigación la utilización de operadores que difieran de los tradicionalmente utilizados en los algoritmos mencionados. También se excluye la calibración de los parámetros propios del algoritmo genético.

La aplicación del modelo se realizará en una red básica experimental y en las redes NSFNET y NTTNET, contrastándose los valores obtenidos de throughput y demora promedio, con los presentados por los algoritmos tradicionales de enrutamiento: OSPF, SPF, BF, Q-R, PQ-R y el algoritmo original AntNet.

2. El Problema de Enrutamiento

De acuerdo a [13] el enrutamiento es un mecanismo que permite que los paquetes de información transmitida en una red IP, sean enviados desde un nodo de origen a un nodo de destino a través de una secuencia de nodos intermedios de distribución. El enrutamiento es necesario porque en las redes no todos los nodos están directamente conectados, debido a que el costo de conectar todos ellos entre sí, se vuelve prohibitivo. El enrutamiento selecciona caminos que cumplan los objetivos y restricciones del tráfico, determinando cuales nodos serán atravesados por los paquetes de información.

El enrutamiento es una actividad distribuida de construcción y uso de tablas de rutas en cada nodo, las cuales guardan información que es utilizada por el algoritmo de enrutamiento, para tomar las decisiones de envío de paquetes a su destino, es decir indica cual es el nodo que debe seleccionarse para que cada paquete entrante continúe su viaje hacia su nodo de destino. Para cada nodo su tabla de rutas consiste en un conjunto de 3-tuplas de la forma (destino, distancia estimada, próximo nodo) definida para todos los destinos de la red.

Para lograr la confiabilidad de una la red, el algoritmo de enrutamiento utilizado por un router debe resolver el Problema de Enrutamiento *(NRP: Network Routing Problem)*, de modo de encontrar los caminos por los cuales el tráfico de paquetes fluya en la red, entre los nodos de origen y su nodo de destino, maximizando el rendimiento de la red y minimizando los costos.

El enrutamiento determina el nivel de servicio ofrecido. Los routers, actualizan sus tablas de rutas mediante un algoritmo especialmente creado para este propósito, direccionando los datos a través de la red de modo de minimizar una función de costo, típicamente puede ser la distancia física, la demora entre nodos, etc. En una red solo algunos nodos se conectan entre sí, ya que sería impráctico conectar todos los nodos directamente. Los algoritmos de enrutamiento enfrentan desafíos importantes debido a la complejidad de la topología de las redes modernas. Existen algoritmos centralizados con problemas de escalabilidad, algoritmos estáticos que no pueden adaptarse al ritmo de los cambios constantes de las redes, y además existen algoritmos distribuidos y dinámicos que tienen problemas de estabilidad y oscilación [14].

El enrutamiento constituye el núcleo principal de un sistema de control de tráfico de una red, en conjunto con el control de flujo y control de congestión, determina el rendimiento general de la red en términos de calidad y cantidad del servicio entregado [12].

Tradicionalmente los algoritmos de enrutamiento se han basado en minimizar su función objetivo, sin considerar el tráfico en sí, es decir sin optimizar el *throughput* de la red, esto no era de mayor importancia en los inicios de Internet, es por ello que tradicionalmente se utilizan los algoritmos *RIP (Routing Información Protocol)*, basado en el método denominado vector de distancias y el *OSPF (Open Shortest Path First)*, que es el de mayor utilización basado en el método conocido como *link-state method.* Ambos métodos escogen el camino con el menor costo entre pares de nodos, lo cual puede llevar a "cuellos de botella" o congestión, ya que siempre se escoge el mejor camino para enviar todo el tráfico mientras otros caminos permanecen sin uso.

Los algoritmos tradicionales de enrutamiento no tienen la suficiente flexibilidad para satisfacer las demandas crecientes de tráfico en las redes modernas, los requerimientos actuales de tráfico y calidad de servicio en las redes actuales, requieren optimizar el *throughput* de modo de satisfacer adecuadamente la demanda de los servicios en la red.

En redes dinámicas, el flujo de paquetes no es estático y sigue un comportamiento estocástico, lo cual lo hace muy difícil de modelar [12], el algoritmo de enrutamiento debe manejar un conjunto de funcionalidades básicas para manejar la congestión, control, encolamiento y tráfico generado por los usuarios.

Existen muchos posibles problemas de ruteo, de acuerdo a las características de la red y del tráfico. Las condiciones del tráfico cambian constantemente y la estructura de la red también puede cambiar debido a fallas en routers o enlaces o nuevos nodos que se agregan. La tarea principal del algoritmo de enrutamiento es direccionar los paquetes de datos desde su nodo de origen a su nodo de destino maximizando el rendimiento de la red y minimizando los costos (demora de los paquetes).

2.1. Modelo de Redes TCP/IP

De acuerdo a [15] las redes TCP/IP son redes de datos de topología irregular de conmutación de paquetes con una capa de red tipo IP, que sigue al modelo ISO de OSI con una capa de transporte muy simple. Este modelo ha permitido la consolidación de Internet, la cual es una red de área ancha (*WAN*), donde se adoptan esquemas de organización jerárquica de redes y subredes.

En términos generales, las subredes pueden verse como nodos simples conectados a dispositivos puente entre redes (*gateway*). Los *gateway* realizan las tareas de la capa de red como el enrutamiento. Al agrupar diversos *gateway* se definen áreas lógicas dentro de las cuales todos los *gateway* tienen el mismo nivel jerárquico, realizándose un enrutamiento directo entre ellos. La comunicación entre áreas solo se produce por un *gateway* de

frontera o puerta de enlace, de modo que la complejidad del diseño y gestión de los protocolos de enrutamiento se reduce. El router es el dispositivo físico que realiza las funciones de *gateway*.

De acuerdo a [1] para el modelamiento de redes de comunicación, se utilizan grafos dirigidos ponderados en nodos. Los enlaces entre pares de nodos son vistos como tuberías de bits caracterizados por el ancho de banda [bits/s] y la demora de transmisión [seg]. Cada nodo es de tipo almacenamiento y retransmisión, cuenta, además, con un buffer donde almacena los paquetes que llegan y salen del nodo, representando las colas de llegada y salida del nodo. Los paquetes se clasifican en dos clases: datos y ruteo, usualmente los paquetes de datos son servidos en colas con baja prioridad, mientras que los de ruteo son servidos por colas de alta prioridad. Para cada paquete de entrada, el componente de ruteo del nodo usa la información almacenada en su tabla local de rutas, para escoger el siguiente nodo al cual direccionar el paquete para seguir su camino a su destino. El tiempo que demora mover un paquete desde un nodo a su vecino depende del tamaño del paquete y de las características de transmisión del enlace.

2.2. Seudocódigo de Algoritmo de Enrutamiento

En general los algoritmos de enrutamiento deben cumplir con un conjunto de funcionalidades básicas, que se ilustra en la Figura 2-1, la cual representa un algoritmo de enrutamiento de alto nivel:

En cada nodo de la red

1. Adquisición y organización de información actualizada respecto del estado local, identificando el flujo de tráfico local y el estado de los recursos disponibles localmente
2. Construir una visión global del estado de la red, posiblemente intercambiado la información de estado local
3. Usar la visión global para fijar valores de la tabla de rutas local y definir la política de ruteo local que optimice alguna medida de rendimiento de la Red
4. Enviar hacia adelante el tráfico generado por el usuario, de acuerdo a la política de ruteo definida
5. Asincrónica y concurrentemente con los otros nodos, repetir las actividades anteriores sobre el tiempo

Figura 2-1: Algoritmo de enrutamiento de alto nivel

2.3. Formulación del Problema de Enrutamiento

Los modelos basados en la teoría de grafos pueden aplicarse a optimización de tráfico en redes y análisis de redes de comunicaciones, para ello se asigna un conjunto de entidades como nodos y otro conjunto como los enlaces. Los routers y computadores son considerados, teóricamente, como nodos, y las líneas de transmisión, son los enlaces. De modo que una red puede considerarse matemáticamente como un conjunto de nodos y enlaces de un grafo G = (N, L), donde N es el conjunto de nodos y L es el conjunto de los enlaces.

La instancia de una red de comunicaciones se mapea directamente en el grafo con N nodos y los enlaces son vistos como cañerías con un ancho de banda [bit/seg] y una demora de transmisión [seg] y son utilizados siguiendo un esquema de multiplexación estadística. Los paquetes se subdividen en paquetes de datos y paquetes de ruteo, todos en su tipo tienen la misma prioridad, de modo que son despachados en una forma FIFO, pero los paquetes de ruteo tienen prioridad sobre los paquetes de datos.

Dado un grafo G = (N, L) que representa una red de comunicaciones, el problema a resolver es encontrar el camino de costo mínimo entre cada par de vértices del grafo. Esto difiere del problema clásico de camino mínimo en grafos que tienen una solución determinista o bien pueden ser resueltos por algoritmos polinomiales, ya que se vuelve extremadamente difícil cuando los costos en los arcos varían en el tiempo en forma estocástica, como es el caso de las redes de comunicaciones IP [16].

Una de las características distintivas del problema de ruteo en redes de comunicación es su estado no estacionario, ya que una de las características principales del flujo de tráfico es que éste cambia en el tiempo, en algunos casos en forma imposible de predecir, por otra parte nuevos nodos y enlaces se agregan aleatoriamente a la red o bien nodos o enlaces pueden ser eliminados de la red. Este es un problema importante y difícil, ya que tiene una fuerte influencia en el comportamiento total de la red y porque las características de la red, tales como carga de tráfico y su topología pueden variar aleatoriamente.

El problema de ruteo en redes de comunicación es muy diferente de los clásicos problemas NP-duros, ya que en casos simplificados es posible reducir un problema a una forma estándar de un problema de optimización combinatorial. En casos reales la dinámica de la topología y del tráfico y por ende de los costos asociados a los enlaces hace imposible realizar una definición formal del optimo a encontrar [1].

2.4. Métricas de Rendimiento

El rendimiento de una red se mide, dependiendo del tipo de servicio esperado, sobre un determinado intervalo de tiempo y puede expresarse en valores instantáneos, acumulados o promedios. Las métricas estándares utilizadas son [12]:

a) Throughput: El número de paquetes de datos correctamente enviados [bits/s], es un índice global de rendimiento que se asocia a la cantidad de tráfico servido. Se expresa como la suma de bits o paquetes de datos correctamente enviados en un intervalo de tiempo.
b) Demora punto a punto: El tiempo necesario para que un paquete alcance su nodo de destino. El rango de demora de los paquetes varía desde valores muy bajos para pares de nodos cercanos o conectados por enlaces de alta velocidad hasta valores muy altos, en caso de nodos muy lejanos y alcanzables solo a través de enlaces de bajo ancho de banda.

Los resultados obtenidos con este proyecto serán comparados con el estado del arte de los algoritmos de enrutamiento, disponibles en la literatura de telecomunicaciones y aprendizaje de máquina, los cuales son utilizados como caso típico de benchmarking en esta área, aplicados a las redes NSFNET y NTTNET. Según lo presentado por [12] estos algoritmos son:

- **OSPF:** Es una implementación del algoritmo estático de camino mínimo, donde el camino tomado por un determinado paquete se determina solo considerando el mínimo costo entre su origen y destino. Implementa el protocolo IGP (*Interior Gateway Protocol*) de Internet, los costos de cada enlace son asignados estáticamente en base a sus características físicas, las tablas de rutas se asignan directamente con el valor del camino mínimo (menor tiempo) para paquetes de datos de 512 bytes. Usualmente el camino mínimo se determina en base al tiempo que demoran los paquetes en recorrer la red.

- **SPF:** Es un algoritmo adaptativo, con métricas dinámicas para la evaluación de los costos de cada enlace, los cuales son calculados como los pesos promedio entre los valores máximos y mínimos de la demora en la transmisión de los paquetes a través de la red. Una adaptación de este algoritmo fue implementada en la segunda versión de Arpanet, y sus sucesivas revisiones. Este puede variar las políticas enrutamiento de acuerdo a las condiciones del tráfico. Los costos de los enlaces son calculados como el promedio ponderado entre los menores y mayores valores reales que reflejen la demora en intervalos fijos, esto indica la utilización del nodo, tamaño de colas, demoras en la transmisión, etc.

- **BF:** Implementa el algoritmo de Bellman-Ford, el cual es asíncrono distribuido con métricas dinámicas, con cálculo de costos de enlace en forma similar a SPF. Se utiliza principalmente para ruteo en redes locales debido a que viene incluido en el protocolo de enrutamiento de la versión *BSD* de *Unix*.

- **Q-R:** Es una versión online asíncrona del algoritmo de Bellman-Ford, Q-R. Su característica principal es que aprende en línea los valores de costo de los enlaces $Q_k(d,n)$, los cuales se estiman como el tiempo en que se alcanza el nodo d, desde el nodo k, vía el nodo vecino n, enviando un paquete P desde k a n con destino d, inmediatamente se genera un paquete de retorno desde n a k, de modo que al finalizar el viaje de los paquetes determina el valor aprendido para $Q_k(d,n)$, en base a las diferencias entre los tiempos de llegada de ambos paquetes. En Q-R, el mejor enlace es aquel con el menor valor de $Q_k(d,n)$, el cual es determinísticamente escogido por los paquetes.

- **PQ-R:** Es una versión predictiva de Q-R, en Q-R los paquetes siempre escogerán el mejor enlace $Q_k(d,n)$, esto implica que enlaces con una alta carga temporal, nunca serán seleccionados, a menos que todos los otros enlaces salientes del nodo tengan un valor mayor de $Q_k(d,n)$, PQ-R implementa un modelo de tasa de variación de enlaces, llamado tasa de recuperación, y lo utiliza para probar nodos que aunque no tengan el mejor valor de $Q_k(d,n)$, tengan una alta tasa de recuperación.

Estos algoritmos serán utilizados como benchmarking con distintos patrones de flujo de tráfico de datos. Con el fin de homologar los resultados obtenidos en este proyecto con los benchmarking, se utilizarán como métricas de evaluación el throughput en [bits/s] y el tiempo de demora para los paquetes de datos en [seg].

3. El Algoritmo AntNet

La congestión del tráfico, es uno de los principales problemas que enfrenta una red, trae como consecuencia el retardo y la pérdida de paquetes. El control de la congestión en redes es un problema complejo el cual ha sido resuelto aplicando técnicas de camino mínimo empleando modelos lineales, basado en una función objetivo que cumpla con restricciones de conservación del flujo, sin embargo los modelos lineales no son adecuados para representar las características de tráfico de una red dinámica, es por ello que han surgido nuevos algoritmos basados en agentes móviles, que evalúan el estado de la red en forma dinámica con el objeto de optimizar permanentemente el tráfico de ella, este es el caso del algoritmo AntNet creado por Dorigo y Caro [12], basado en sistemas de colonias de hormigas.

3.1. Algoritmo de Colonia de Hormigas (ACO)

El algoritmo de optimización por Colonia de Hormigas (ACO: Ant Colony Optimization) es una metaheurística, propuesta por [5], para solucionar problemas de optimización combinatoria, inspirada en el comportamiento de las colonias de hormigas en su proceso de búsqueda y transporte de comida entre una fuente de alimentación y su nido utilizando mecanismos de comunicación indirecta mediante rastros de feromonas dejadas en el camino. ACO se basa en una colonia de hormigas artificiales, implementada como agentes computacionales simples que trabajan de manera cooperativa y se comunican indirectamente mediante rastros de feromona artificiales, son algoritmos constructivos, es decir en cada iteración, cada hormiga construye una solución al problema recorriendo un grafo que representa las instancias del problema, cuyas aristas representan los posibles pasos que una hormiga puede dar para moverse de un nodo a otro, las aristas almacenan dos tipos de información para guiar el movimiento de cada hormiga:

a) La Visibilidad: Mide la preferencia heurística de moverse desde un nodo a otro nodo teniendo en cuenta valores de costos.
b) El rastro de feromona artificial: Mide la "deseabilidad aprendida" del movimiento de un nodo a otro, imitando de esta forma, a la feromona real que depositan las hormigas en su camino. Esta información es actualizada dinámicamente durante la ejecución del algoritmo, con las soluciones que encuentra cada hormiga en su camino.

ACO ha sido aplicado para encontrar soluciones aproximadas a diversos problemas de optimización, simulándolo mediante software. Mediante el movimiento de las hormigas en el grafo se construyen soluciones o caminos en el grafo y modifican el problema retroalimentando con la información obtenida hasta que encuentran una

solución óptima al problema. ACO ha sido aplicado a diversos problemas de optimización combinatoria, como el Problema de enrutamiento en redes, Problema del vendedor viajero, Problema de corte de piezas, etc.

Basado en la propuesta original de [4], la Figura 3-1 representa el pseudo código de alto nivel del algoritmo ACO básico:

```
1.  procedure ACO metaheuristic()
2.    Initialization();
3.    while (¬ stopping criterion)
4.      schedule_activities
5.        ants_construct_solutions using pheromone();
6.        pheromone_updating();
7.        daemon_actions(); /* OPTIONAL */
8.      end schedule_activities
9.    end while
10. return best_solution_generated;
```

Figura 3-1: Algoritmo ACO de alto nivel

En un principio, el procedimiento *Initialization()* considera parámetros tales como: el rastro inicial de feromona, el número de hormigas en la colonia, los pesos que definen la proporción de la información heurística en la regla de transición probabilística.

Posteriormente el algoritmo cuenta con tres bloques principales, en el primero *ants_construct_solutions using pheromone()* agentes (hormigas) construyen soluciones incrementales en un proceso gobernado por decisiones estocásticas que dependen de las valores de las variables que representan las feromonas. En el segundo bloque *pheromone_updating()*, las soluciones generadas se utilizan para actualizar las variables de feromona, intentando que el proceso completo de generación abarque la región del espacio de soluciones donde se espera que se encuentren las mejores soluciones. Las variables de feromona, regulan procesos tales como: autorizar / denegar a las hormigas para actualizar las feromonas, disminuir niveles de feromonas imitando los procesos naturales de evaporación de modo de favorecer la exploración, de acuerdo a comunicaciones desde el procedimiento *daemon_actions()*. Este último procedimiento implementa actividades desde un punto de vista global, las cuales no pueden ser realizadas por las hormigas, como por ejemplo determinar la calidad de las soluciones generadas, depositar nuevas cantidades de feromona en los caminos de alguna solución que se desee potenciar.

De acuerdo a [4], la implementación de ACO, se basa en una representación de una red mediante un grafo $G = (N, L)$, donde N es el conjunto de nodos y L es el conjunto de los enlaces. En este grafo el comportamiento de las hormigas tiene dos modos de trabajo: hacia adelante (*forward*) y hacia atrás *(backward)*. Las hormigas están en el modo *forward* cuando se mueven desde su nido a la fuente de comida, y están en modo *backward* cuando lo hacen en sentido contrario. Las hormigas en modo *forward* construyen una solución de encaminamiento, escogiendo probabilísticamente el próximo nodo a moverse, entre aquellos que forman su vecindario, es decir los nodos con los cuales el nodo actual tiene un enlace $(en\ G = (N, L)\ dos\ nodos\ i, j \in N\ son\ vecinos\ si\ existe\ el\ arco(i, j) \in L)$. Para realizar esta elección probabilística, la hormiga se basa en el rastro de feromona depositado en el grafo previamente por otras hormigas *backward*. Las hormigas *forward* no depositan feromonas cuando se mueven. Cuando una hormiga en modo *forward* alcanza su destino, ésta cambia a modo *backward* y comienza su viaje hacia atrás hasta su punto de origen. Las hormigas *backward* utilizan una memoria explicita acerca del camino recorrido, de modo que la elección de su camino hacia atrás es determinista. En su viaje de retorno, las hormigas *backward* depositan feromonas en los arcos que atraviesan.

Las hormigas memorizan los nodos que visitan en el modo *forward*, además del costo de los arcos atravesados, de modo que evalúan el costo de la solución generada y usan esta información para regular el monto de feromona que depositaran en su modo *backward.*

En las colonias de hormigas reales, la intensidad de la feromona disminuye en el tiempo, debido a la evaporación, en ACO esto se modela aplicando reglas de evaporación de feromona que reducen la influencia de las feromonas depositadas por las hormigas iniciales, ya que las primeras soluciones encontradas son, probablemente, de menor calidad que las que se generan en el proceso iterativo.

La actualización de feromona es una función de la solución generada que permite dirigir a las hormigas por el mejor camino, permitiendo que dejen mayores monto de feromona en los caminos mas cortos, de modo que en la elección probabilística de las próximas hormigas habrá una mayor convergencia hacia los caminos mas cortos.

En el grafo $G = (N, L)$, la variable τ_{ij} representa el rastro de feromona depositado por las hormigas en el arco (i, j), este rastro es escrito y leído por las hormigas. La intensidad del rastro de feromona es proporcional a la utilidad, que las hormigas estimen, que éste arco presta para construir nuevas soluciones. En cada nodo se almacena información local acerca del nodo o en sus arcos salientes, esta información es leída por las hormigas y utilizada en forma aleatoria para seleccionar el siguiente nodo hacia el cual se moverán.

La probabilidad de que la hormiga k, que se encuentra en el nodo i escoga ir al nodo j está dada por la ecuación (3.1):

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}} \quad Si\ j \in N_i^k, 0\ en\ otro\ caso \tag{3.1}$$

Donde p_{ij}^{k} Es la probabilidad de que la hormiga k vaya del nodo i al nodo j

η_{ij} Es la información heurística, el conocimiento previo sobre la ruta, corresponde al inverso de la distancia entre nodos (i, j), es decir $\eta_{ij} = 1/d_{i,j}$

N_i^k Es el posible vecindario de la hormiga k cuando se encuentra en el nodo i.

τ_{ij} Es el rastro de feromona depositado por las hormigas en el arco (i, j)

α, β Son parámetros que determinan la influencia de la feromona y el conocimiento previo, respectivamente.

La feromona se evapora localmente y globalmente. La evaporación local de feromona se modela de acuerdo a la ecuación (3.2):

$$\tau_{ij} = (1 - \varepsilon)\tau_{ij} - \varepsilon\tau_0 \tag{3.2}$$

Donde τ_0 es el valor inicial del rastro de feromona global y ε $(0 < \varepsilon < 1)$ es la tasa de evaporación local de feromona, mientras que la evaporación global se modela mediante la ecuación (3.3):

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho\, \Delta\tau_{ij}^{bs} \tag{3.3}$$

Dónde $\Delta\tau_{ij}^{bs} = \frac{1}{C^{bs}}$ es la cantidad de feromona depositada por la mejor hormiga y ρ es la tasa de evaporación global.

3.2. Descripción de AntNet

AntNet es la aplicación de ACO al problema de enrutamiento en redes, mediante el cual se envían agentes inteligentes (hormigas) sobre la red, quienes se comunican indirectamente mediante la información que ellas dejan tras de sí en los routers de su camino, en forma análoga a las hormigas en la naturaleza que dejan un rastro de

feromonas en su camino, luego de sucesivas iteraciones, se espera que esta información conduzca a caminos óptimos de enrutamiento entre los Routers de la red.

AntNet se basa en el comportamiento de una colonia de hormigas en su actividad de búsqueda de alimentos. El algoritmo explora una red de datos con el propósito de construir tablas de enrutamiento, y mantenerlas adaptadas a las condiciones de tráfico en la red. Comienza con la creación de hormigas artificiales en cada nodo de la red, representada por un grafo con nodos y enlaces bidireccionales, a intervalos de tiempo regulares. Cada nodo se caracteriza por el número de vecinos que tiene y por su tabla de enrutamiento que contiene las probabilidades de elegir el vecino *n* en el siguiente salto, siendo *d* el nodo destino. Las hormigas eligen aleatoriamente un nodo destino y seleccionan el siguiente salto entre los nodos vecinos no visitados previamente. La probabilidad de seleccionar un nodo particular, es proporcional al valor que tiene la tabla de enrutamiento para este nodo, así como una cola local, la cual es creada en cada nodo por el tráfico local. El identificador de cada nodo visitado se pone en una pila que forma parte de la estructura de datos de la hormiga. También se almacena el tiempo que le toma a cada hormiga para llegar al nodo destino.

Cuando una hormiga alcanza el nodo destino, ésta genera una *hormiga de regreso* y le transfiere el contenido de la pila a la hormiga nueva, y entonces ésta muere. La hormiga de regreso toma la misma ruta que su antecesora, pero en dirección contraria. Conforme la hormiga avanza hacia el nodo origen, actualiza las tablas de enrutamiento de acuerdo a lo siguiente:

- Cuando el tiempo utilizado por la hormiga para llegar al nodo destino fue menor que el tiempo memorizado por el modelo, entonces la hormiga actualiza la tabla de enrutamiento, en otras palabras, deposita mayor cantidad de feromona.
- Cuando el tiempo utilizado por la hormiga para llegar al nodo destino fue mayor que el tiempo memorizado por el modelo, entonces la hormiga no efectúa cambio alguno en la tabla de enrutamiento.

3.3. Procedimiento de AntNet

Para tener una primera aproximación al funcionamiento del algoritmo AntNet, éste puede describirse mediante el siguiente procedimiento, el cual posteriormente, en el subcapítulo 3.4 será convertido a un algoritmo en pseudocódigo:

En cada nodo de la red

1. En intervalos regulares y concurrentemente con el flujo de tráfico de cada nodo, se lanza una hormiga artificial hacia un nodo de destino seleccionado aleatoriamente.

2. Las hormigas artificiales actúan concurrente e independientemente, comunicándose entre ellas en una vía indirecta a través de las feromonas que ellas leen y escriben localmente en cada nodo.

3. Cada hormiga artificial avanza hacia su nodo de destino. En cada nodo intermedio aplica una política estocástica Greedy para seleccionar el próximo nodo a moverse, haciendo uso de i) La información local de las feromonas en el nodo ii) Información heurística local del nodo dependiente del problema y iii) La memoria de la hormiga.

4. Mientras se mueve la hormiga colecciona información acerca del intervalo de tiempo, el estado de congestión y los identificadores de los nodos recorridos.

5. Cuando cada hormiga llega a su nodo de destino, se elimina y se genera una nueva hormiga de retorno que hereda sus estructuras de datos y vuelve hacia el nodo de origen siguiendo exactamente el mismo camino recorrido pero en sentido inverso.

6. Durante su viaje hacia atrás la hormiga actualiza los estados locales de la red y las feromonas de cada nodo visitado en función del camino recorrido y el tiempo empleado.

7. Cuando la hormiga artificial retorna a su nodo origen, es eliminada.

3.4. Descripción del algoritmo AntNet

El algoritmo AntNet se describe en términos de dos conjuntos homogéneos de agentes móviles (hormigas artificiales), llamadas *Fordward Ants y Backward Ants*:

- *Forward Ants* quienes recogen información acerca del estado de la red
- *Backward Ants* quienes usan la información recolectada para adaptar las tablas de ruteo de los Routers en su camino.

En AntNet cada router contiene una tabla de rutas, donde cada destino está asociado a sus interfaces y cada interface tiene asociada una probabilidad que indica la importancia de seguir esa interface en las condiciones actuales. El router contiene un modelo estadístico para almacenar la media y la varianza de los tiempos de viaje hacia todos los destinos de su tabla de rutas. Los agentes móviles se comunican en una forma indirecta por el paradigma de la estigmergía a través de la información que concurrentemente leen y actualizan en dos estructuras de datos en cada nodo que forma parte de su camino, lo que se ilustra en la Figura 3-2, propuesta por [12], la cual muestra las estructuras en cada nodo, utilizadas por los agentes móviles, en este caso, el nodo k tiene L vecinos en una red con N nodos. La tabla de rutas es un vector de distancias o tiempos para alcanzar cada nodo, pero sus entradas son valores probabilísticos que representan la probabilidad de elegir cada ese nodo para llegar al nodo destino, además otra estructura almacena estadísticas acerca del tráfico local y desempeña un rol de modelo adaptativo local para el tráfico que sale a cada posible destino.

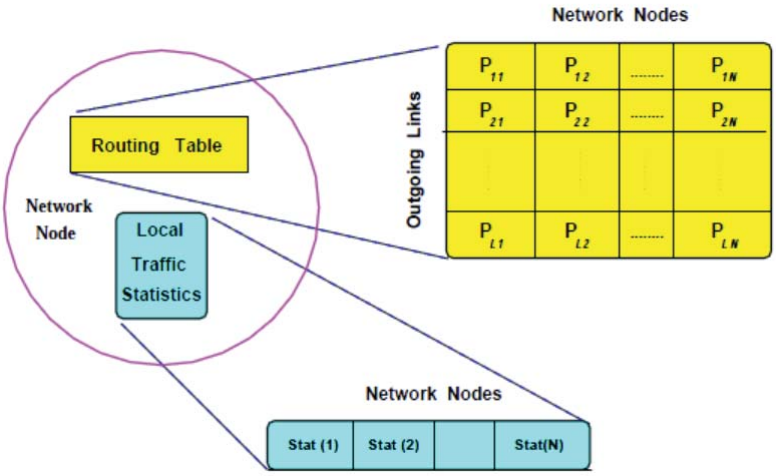

Figura 3-2: Estructura de tabla de rutas y estadísticas en un nodo

Una tabla de rutas T_k está organizada como un vector distancia pero con entradas probabilísticas. T_k define la política actual de ruteo adoptada por el nodo k, esto es para cada posible nodo de destino d y por cada nodo vecino n, T_k almacena la probabilidad P_{nd} que representa la bondad o deseabilidad, bajo la política de ruteo actual, de escoger el nodo n como próximo nodo cuando el nodo de destino es d. Por lo tanto cada nodo k, su tabla de rutas tendrá N destinos posibles, indicado en la ecuación (3.4), con la siguiente relación de probabilidad de alcanzar cada destino:

$$\sum_{i \in N_k} P_{nd} = 1 \; j = 1, \dots N \tag{3.4}$$

donde:

d representa cada posible destino del router.
N_k Representa el conjunto de nodos vecinos del nodo *k*.
P_{nd} Es la probabilidad de que la hormiga en el nodo *k* salte al nodo *i* al cuando el nodo de destino es *j*, ($j \neq k$).

El arreglo $M_k(\mu_d, \sigma_d^2, w_d)$ define un modelo paramétrico estadístico simple para la distribución del trafico sobre la red, tal como este es visto por el nodo k. Este modelo es adaptativo y descrito por muestras de la media y varianzas calculadas sobre el tiempo de viaje experimentado por los agentes móviles sobre una ventana móvil de observación, w_d es usada para almacenar el mejor valor de los tiempos de viaje de los agentes móviles.

Para cada nodo de destino d en la red, una estimación de la media y la varianza μ_d, σ_d^2 representa una aproximación al tiempo estimado para llegar a d y la estabilidad de la ruta. Los autores del algoritmo [12] recomiendan utilizar estrategias aritmética exponencial y de ventanas, de modo que las expresiones recomendadas para su cálculo están dadas por las ecuaciones (3.5) y (3.6)

$$\mu_d = \mu_d + \eta(o_{kd} - \mu_d) \tag{3.5}$$

$$\sigma_d^2 = \sigma_d^2 + \eta((o_{kd} - \mu_d)^2 - \sigma_d^2) \tag{3.6}$$

donde:

o_{kd} es el nuevo tiempo de viaje observado entre los nodos k y d

En cada nodo k, cada agente móvil, en su viaje a su nodo de destino d, selecciona el nodo n a moverse entre los vecinos que aún no ha visitado o sobre todos los vecinos, en caso de haberlos visitado a todos previamente. El

vecino n es seleccionado con una probabilidad (deseabilidad) P'_{nd} calculada como la suma normalizada de la entrada probabilística P_{nd} en tabla de rutas con un factor de corrección heurística ln que toma en cuenta el largo de la cola en el enlace n del nodo k. Esto queda indicado por las ecuaciones (3.7) y (3.8).

$$P'_{nd} = \frac{P_{nd}+\alpha ln}{1+\alpha(|N_k|-1)} \tag{3.7}$$

$$ln = 1 - \frac{q_n}{\sum_{n'-1}^{|N_k|} q_{n'}} \tag{3.8}$$

El valor de α mide la importancia de la correccion heurística.

3.5. La Hormiga Artificial

El algoritmo AntNet utiliza agentes computacionales simples, en este caso hormigas artificiales, que trabajan de manera cooperativa y se comunican mediante rastros de feromona artificial; simulando el comportamiento de una colonia de hormigas naturales.

La hormiga artificial, es el agente básico del algoritmo AntNet, y de acuerdo a [1], presenta las siguientes características:

- Deposita cierta cantidad de feromona en cada iteración del algoritmo que se aplique.
- Busca soluciones válidas de costo mínimo para el problema a solucionar.
- Tiene una memoria L que almacena información sobre el camino seguido hasta el momento, esto es, L almacena la secuencia generada. Esta memoria puede usarse para (i) construir soluciones válidas, (ii) evaluar la solución generada, y (iii) reconstruir el camino que ha seguido la hormiga.
- Tiene un estado inicial, que normalmente corresponde a una secuencia única, y una o más condiciones asociadas de parada.
- Comienza en el estado inicial, y se mueve siguiendo estados válidos, construyendo la solución asociada incrementalmente.
- El movimiento se lleva a cabo aplicando una regla de transición, en función de tres parámetros: los rastros de feromona que están disponibles localmente, los valores heurísticos de la memoria privada de la hormiga, y las restricciones del problema.

- Si durante el procedimiento de construcción una hormiga se mueve desde el nodo r hasta el s, esta puede actualizar el rastro de feromona τ_{rs}asociado al arco a_{rs}. A este proceso se le denomina actualización en línea de los rastros de feromona paso a paso.
- El procedimiento de construcción acaba cuando se satisface alguna condición de parada, normalmente cuando se alcanza un estado objetivo.
- Una vez que la hormiga ha construido la solución, puede reconstruir el camino recorrido, y actualizar los rastros de feromona de los arcos y/o componentes visitados, utilizando un proceso llamado actualización en línea a posteriori; este es el único mecanismo de comunicación entre las hormigas, utilizando la estructura de datos que almacenan los niveles de feromona de cada arco o componente (memoria compartida).

3.6. Algoritmo AntNet

De acuerdo a los autores del algoritmo AntNet [12], dada una red de datos con **N** nodos, *s* un nodo de origen y *d* un nodo de destino, se define la siguiente notación:

- $F_{s\to d}$: *Forward Ant*, la cual va del nodo origen *s* a un nodo destino *d.*
- $B_{d\to s}$ *: Backward Ant,* se genera cuando una *Forward Ant* $F_{s\to d}$ alcanza su nodo destino *d.*
- $V^{i}_{s\to d}=\{s, v_1, v_2, v_3, \dots d\}$ Camino recorrido por $F_{s\to d,}$ incluye los nodos de origen s, d y los nodos intermedios v_k.
- M^{v_k}: Es un modelo estadístico de la demora esperada en el nodo v_k, el cual consiste en un vector de N-1 estructuras de datos formadas por (μ_d, σ_d^2, W_d), donde μ_d, σ_d^2 representan la media muestral y la varianza respectivamente, del tiempo de viaje, para alcanzar el nodo de destino d desde el nodo actual v_k, W_d es el mejor tiempo de viaje observado. Las estadísticas se basan en las demoras experimentadas por las hormigas en sus viajes del nodo de origen al destino y vuelta atrás. M^{v_k} representan un punto de vista local del tráfico global de la red.
- T^{v_k}: Es la tabla de feromonas en el nodo v_k.
- R^{v_k}: Es la tabla de rutas de datos en el nodo v_k.
- $T'_{s\to d}$: Almacena los valores de tiempo de viaje experimentados en cada movimiento de un nodo a otro.

AntNet se basa en la operación de dos tipos de hormigas o agentes:

a) *Forward Ant*, denotada como $F_{s\to d}$, significa que las hormigas son lanzadas en forma aleatoria, en intervalos regulares y concurrentemente con el tráfico de datos, este proceso de generación de agentes ocurre sin ninguna sincronización de los nodos. Estos agentes simulan paquetes de datos que se mueven buscando su destino, hacen

uso indirecto de la estigmergía a través de la información que depositan en los nodos. La tarea específica de cada *Forward Ant* es buscar un camino de demora mínima entre sus nodos de origen y destino, pasando desde cada nodo a otro adyacente.

En cada nodo intermedio, aplica decisiones estocásticas para seleccionar el próximo nodo a moverse, basado en i) las variables de feromona locales en el nodo, ii) el estado de las colas de los enlaces, iii) la información en la memoria de la hormiga (para evitar ciclos infinitos). En su viaje al destino, la *Forward Ant* colecciona información acerca del tiempo de viaje e identificación de los nodos recorridos.

b) *Backward Ant*, denotada como $B_{d\to s}$, ésta es generada cuando una *Forward Ant* $F_{s\to d}$ alcanza su nodo destino *d.* La *Forward Ant* $F_{s\to d}$ se convierte en $B_{d\to s}$, lo que significa que retorna a *s* a través del mismo camino recorrido por $F_{s\to d}$, $V^{i}_{s\to d}$, pero en sentido contrario. En su viaje de retorno, $B_{d\to s}$ utiliza las colas de mayor prioridad con el fin de recorrer rápidamente el camino de retorno, En cada nodo visitado $v_k \in V^{i}_{s\to d}$, la *Backward Ant* actualiza la tabla de rutas con la información recolectada por $F_{s\to d}$., actualizando la información de las rutas locales de cada nodo en el camino. En particular $B_{d\to s}$, actualiza las siguientes estructuras de datos: $M^{v_k}, T^{v_k}, R^{v_k}$. La feromona y la tabla de rutas se actualizan evaluando.los valores almacenados por $F_{s\to d}$.

Tanto la feromona como las tablas de rutas se actualizan en base a la evaluación de la ruta de acceso que fue seguido por la hormiga *Forward Ant* $F_{s\to d}$ desde ese nodo hacia el destino. La evaluación se realiza comparando el tiempo de viaje experimentado con el tiempo previsto de viaje estimado de acuerdo con el modelo de demora local. Cuando la hormiga *Backward Ant*, ha alcanzado el nodo de origen s, ésta es eliminada de la red.

Los paquetes de datos son ruteados de acuerdo a las políticas de decisión estocásticas basadas en la información en las tablas de rutas de cada nodo, las cuales se derivan de las tablas de feromonas usadas para rutear a las hormigas. Sólo los mejores enlaces a saltos siguientes, son almacenados en las tablas de encaminamiento. De esta forma el tráfico de los paquetes de datos es esparcido sobre los mejores caminos disponibles, con la esperanza de optimizar el tráfico en la red, la utilización de sus recursos y el balance de la carga.

Cada hormiga lleva consigo un stack $S_{s\to d}(k)$, k representa el k-esimo router visitado, con $S_{s\to d}(0)$ =s y $S_{s\to d}(m)$ =d donde m es el número de saltos de $F_{s\to d}$ para llegar desde el nodo s al d.

Como ejemplo del comportamiento de las hormigas Forward y Backward, considérese la **Figura 3-3**, donde la *Forward Ant*, $F_{1\to 5}$, se mueve a lo largo del camino formado por los nodos 1→2→3→4→5, al llegar al nodo 5,

ésta se transforma en la *Backward Ant* $B_{5\to1}$ que realizará el viaje en sentido contrario. Al recorrer los nodos en su camino 4→3→2→1, la *Backward Ant* utiliza el contenido de la lista $V_{1\to5}(k)$ y $T'_{1\to5}(k)$ para actualizar los valores de $M^{v_k}(\mu_d, \sigma_d^2, W_d)$. En caso de encontrar buenos sub-caminos, se actualizan también los valores de $M^{v_k}(\mu_i, \sigma_i^2, W_i)$, $i = k+1 \ldots d-1$, además de T^{v_k}, incrementando la feromona en el nodo previo k+1 y disminuyendo la probabilidad de elección de los restantes nodos vecinos.

En este ejemplo, $P^{i}_{1\to5}$={1, 2, 3, 4,5}, con $S_{1\to5}(0)$ =1, $S_{1\to5}(1)$ =2, $S_{1\to5}(2)$ =3, $S_{1\to5}(3)$ =4 y $S_{1\to5}(4)$ =5

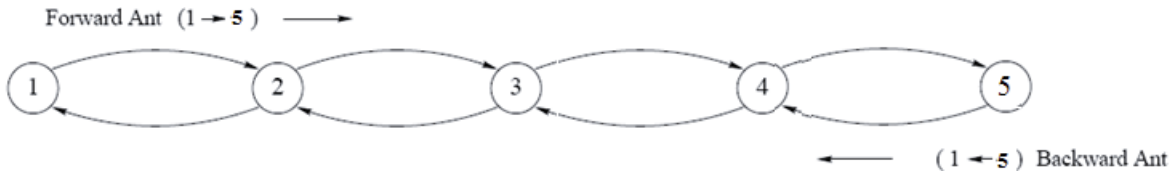

Figura 3-3: Ejemplo hormigas Forward y Backward

A continuación la Figura 3-4 presenta el pseudo código original del algoritmo AntNet, de acuerdo a lo planteado por sus autores [12]:

```
procedure AntNet ForwardAnt(source node, destination node)
    k ← source node;
    hops fw ← 0;
    V [hops_fw] ← k; /* V = LIST OF VISITED NODES */
    T′ [hops_fw] ← 0; /* T′ = LIST OF NODE-TO-NODE TRAVELING TIMES */
    while (k ≠ destination node)
            tarrival ← get_current_time();
            n ← select_next_hope_node(V, destination_node, T k, Lk); /* Lk = LINK QUEUES */
            wait_on_data_link queue(k, n);
            cross_the_link(k, n);
            Tk→n ←get_current_time() − tarrival;
            k ← n;
            if (k ∈ V ) /* CHECK IF THE ANT IS IN A LOOP AND REMOVE IT */
                    hops_cycle ←get_cycle_length(k, V );
                    hops_fw ← hops_fw – hops_cycle;
            else
                    hops_fw ← hops_fw + 1;
                    V [hops_fw] ← k;
                    T′ [hops_fw] ← Tk→n;
            end if
    end while
    become a backward ant(V, T′ );
end procedure
```

```
procedure AntNet BackwardAnt(V, T′ ) /* INHERITS THE MEMORY FROM THE FORWARD ANT */
    k ← destination_node;
    hops_bw ← hops_fw;
    T ← 0;
    while (k ≠ source_node)
            hops_bw ← hops_bw − 1;
            n ← V [hops bw];
            wait_on_high_priority_link_queue(k, n);
            cross_the_link(k, n);
            k ← n;
            for (i ← hops_bw + 1; i ≤ hops_fw; i ← i + 1) /* UPDATES FOR ALL SUB-PATHS */
                    δ ← V [i];
                    Tk→ δ ← T + T′ [i]; /* INCREMENTAL SUM OF THE TRAVELING TIMES
            EXPERIENCED BY FwAnts→d */
                    T ← Tk→ δ ;
                    if (Tk→ δ ≤ Isup( μ δ , σ δ ) ∨ δ ≡ d) /* Tk→ δ IS A GOOD TIME, OR IS THE
            DESTINATION NODE */
                            M_δ^k ← update_traffic_model (k, δ, T_{k→δ,} M_δ^k)
                            r ← get_reinforcement (k, δ, T_{k→δ,} M_δ^k)
                            T_δ^k ← update_pheromone_table (T_δ^k, r)
                            R_δ^k ← update_data_routing_table (R_δ^k, T_δ^k)
                    end if
            end for
    end while
end procedure
```

Figura 3-4 Seudo-código original de algoritmo AntNet

AntNet opera de la siguiente forma:

1. A intervalos regulares Δt, desde cada nodo s de la red se envía proactivamente hacia el nodo d. una *Forward Ant* $F_{s\to d}$, con el objetivo descubrir caminos factibles de bajo costo en s y d, además de determinar el estado de carga de tráfico en el camino recorrido. Las hormigas *Forward* representan una simulación del tráfico de paquetes de datos, cuyos destinos son seleccionados aleatoriamente según la ecuación 3.9, para calcular P_d como la probabilidad de seleccionar el destino d para para una hormiga:

$$P_d = \frac{f_{sd}}{\sum_{d'=1}^{N} f_{sd'}} \tag{3.9}$$

Donde f_{sd} es el número de bits destinados a d que han pasado por s.

2. Almacenamiento de Información:

Durante la fase de ida, las hormigas *Forward* almacenan información de su camino y de las condiciones de tráfico encontradas en cada nodo k y almacenados en estructuras de memoria de la hormiga, la lista

$V_{v_0 v \to v_m} = [v_0, v_1, \ldots, v_m]$ Almacena el orden del conjunto de nodos visitados, donde v_i es el identificador del nodo visitado en el i-esimo paso. $T'_{v_0 v \to v_m} = [T_{v0 \to v1}, v_1, T_{v1 \to v2}, \ldots, T_{vm-1 \to vm}]$ almacena los valores del tiempo de viaje experimentado al moverse de un nodo a otro.

3. Políticas de ruteo adoptadas por hormigas *Forward*

En cada nodo k visitado, las hormigas *Forward* deben escoger el próximo nodo a moverse en su camino a su destino d. Si todos los nodos vecinos de k ya han sido visitados, entonces el próximo nodo se escoge en forma random sin ninguna preferencia entre los vecinos, de acuerdo a la ecuación 3.10:

$$P_{nd} = \begin{cases} \frac{1}{|N_k|-1} & n \in N_k \\ 0 & e.o.c. \end{cases} \quad (3.10)$$

Por otra parte si no todos los nodos vecinos han sido visitados, se aplica una decisión estocástica calculada como la suma normalizada de la entrada probabilística P_{nd} en tabla de rutas, con un factor de corrección heurística ln basada en las variables de feromona que toma en cuenta el largo de la cola en el enlace n del nodo k, de acuerdo a las ecuaciones (3.7) y (3.8).

4. Anulación de loops.

Cuando AntNet detecta que se produce un loop infinito para una hormiga, en el caso que el tiempo de viaje entre s y d, es mayor a la mitad del tiempo de vida de la hormiga, esta es eliminada de la red.

5. Cambio de hormigas *Forward* a *Backward*

Cuando la *Forward Ant* $F_{s \to d}$ alcanza su nodo de destino d, se transforma virtualmente en una *Backward Ant* $B_{d \to s}$, heredando todas sus estructuras de memoria.

6. Actualización de Tablas de rutas y modelos estadísticos de tráfico

Cuando $B_{d \to s}$, arriba al nodo k, ejecuta la siguiente secuencia de actualización: i) actualiza el modelo de tráfico local M^{v_k} con los valores correspondientes al tiempo de viaje entre k y d, ii) evaluación del camino k→d, en términos del tiempo real y el tiempo esperado por el modelo asignando al camino un puntaje directamente proporcional al tiempo de viaje, iii) utiliza el puntaje para reforzar los mejores caminos.

4. Algoritmos Genéticos

Los Algoritmos Genéticos (AG) son métodos matemáticos de optimización planteados por Holland [9] para describir algunos de los procesos de evolución y selección natural. En la naturaleza cada especie, necesita adaptarse a los cambios en su medioambiente, con el objeto de maximizar la esperanza de vida de sus descendientes. El conocimiento que cada especie gana, se almacena en sus cromosomas. Con el paso del tiempo esos cambios en las cromosomas hacen que la especie sea más robusta y más adecuada para sobrevivir y con ello tiene mayor probabilidad de traspasar sus genes a las generaciones futuras. Los AG intentan simular el comportamiento de la naturaleza, para formar un método de optimización que se aplica a problemas reales de búsqueda, se basan en la idea darwiniana de la selección del más apto [10] combinada con otros operadores genéticos.

La idea básica de los AG es simular las vías en que la naturaleza usa la evolución. Los AG plantean la solución a un problema de optimización como un proceso iterativo de búsqueda. Se comienza con un conjunto aleatorio de soluciones tentativas, que constituyen la población inicial de individuos, sobre la cual se aplican operadores de selección, para generar una nueva población a partir de combinaciones entre los individuos, donde los mejores tienen mayores posibilidades de reproducirse, las soluciones inadecuadas tienden a desaparecer, con cada nueva iteración la población se va refinando más, consiguiéndose, de esta forma, mejores soluciones al problema, convergiendo finalmente con una población final de individuos en torno a la solución óptima.

Estas técnicas se basan en un modelo de proceso natural de evolución y se utilizan en sistemas de aprendizaje automático, para resolver problemas de optimización en una emulación de la supervivencia de los individuos más aptos. Los algoritmos genéticos aplican la idea de la selección de acuerdo a la aptitud y la combinan con otros operadores genéticos para producir métodos de gran robustez y aplicabilidad, mantienen una población que se renueva constantemente. La nueva generación se construye probabilísticamente, a partir de los individuos más aptos de la generación anterior, combinando su "Código genético".

En los Algoritmos Genéticos las variables de diseño son codificadas en cadenas de longitud finita, las cuales son análogas a los cromosomas en un sistema biológico. John Holland desarrolló los primeros estudios, sus objetivos han sido abstraer y explicar los procesos de adaptación en sistemas naturales, diseñando un sistema artificial que, mediante programación, emula los mecanismos de los sistemas naturales. Holland demostró que, bajo un control adecuado y con ciertas transformaciones, se mejora la aptitud promedio de la población. Actualmente el desarrollo de la ciencia y la tecnología no solo busca una solución, sino la mejor solución dado un problema definido, es decir consiste en un problema de optimización.

El éxito de los algoritmos genéticos, no solo se debe a su sencillez y poderío, sino también al hecho de que estos requieren muy poca información sobre el espacio de búsqueda, pues trabajan sobre un conjunto de soluciones o parámetros codificados-cromosomas o individuos (en su forma más simple como cadenas binarias). Inicialmente esta población de cromosomas puede ser generada aleatoriamente; el algoritmo busca la solución, mediante la aproximación de la población, en lugar de una aproximación de "punto-por-punto" como en otros métodos. El algoritmo evalúa la función objetivo, pero no requiere de información adicional.

Los Algoritmos Genéticos son procedimientos iterativos que mantienen una población de posible diseños, o diseños candidatos, de un tamaño fijo. Cada paso iterativo es llamado una generación. El conjunto de diseño inicial, generado al azar, es denominado población inicial.

Sin embargo en un proceso regido por la selección natural, los individuos de la población compiten unos con otros para convertirse en progenitores, en función de un valor llamado ajuste (fitness). Después que han sido seleccionados, se les aplica una serie de funciones denominadas operadores genéticos: mutación, inversión y cruzamiento, para crear nuevos diseños, los cuales normalmente tienen valores de ajuste más elevados, lo que significa que se ha obtenido un diseño mejor. Después, estas nuevas cadenas remplazan a los miembros de las antiguas poblaciones.

Las técnicas de optimización no se evalúan únicamente por su capacidad para localizar el óptimo (eficacia), sino también, deben ser evaluadas por su eficiencia (costo): En este contexto, los Algoritmos genéticos son competitivos en términos de la calidad de la solución que ofrecen, con respecto a su costo.

4.1 Componentes de un Algoritmo Genético

Al implementar un AG, debe considerarse que estos presentan los siguientes componentes:

a) **Arquitectura Genética:** Existe una relación entre una cadena de genes y una solución del problema. Debe utilizarse una codificación que represente la cadena de genes que tenga una solución asociada. En la implementación de problemas de Transporte se utiliza una cadena de números que representen cada una de las ciudades. Cada individuo es una cromosoma que representa una solución al problema y se compone de un string de genes.

b) **Individuos**: Cada solución parcial del sistema a optimizar esta codificada en forma de cadena o strings. A cada uno de estos string se les denomina individuo. Un ejemplo de individuo lo constituye el string: 11010110.

c) **Población**: A un conjunto de individuos se le denomina población. El Método de Algoritmos Genéticos consiste en ir obteniendo en forma sucesiva distintas poblaciones de forma que en cada nueva población los individuos sean mejores soluciones (en promedio) que los individuos de las poblaciones que les preceden. La población inicial que sirve como punto de partida del algoritmo, puede crearse aleatoriamente o bien usando información propia del problema a resolver.

d) **Operadores Genéticos**: Son las distintas funciones que se aplican a las poblaciones y que permiten obtener poblaciones nuevas. Se Destacan los operadores de Selección, Mutación y Cruzamiento:

i) **Operador de Selección:** Emula el principio de selección natural que indica que los mejores individuos de cada generación son los que se reproducen para la siguiente generación. Para su implementación computacional se usa normalmente el método de la ruleta propuesto por Goldberg en [8], el cual consiste en asignarle una fracción de importancia a cada individuo, dependiendo de su valor de la función objetivo, lo que permite establecer la siguiente probabilidad de selección de un individuo:

$$Ps(x_i) = \frac{f(x_i)}{\sum_{i=1}^{n} f(x_i)} \quad (4.1)$$

Donde Ps = Probabilidad de selección (Fracción de Importancia)

Por ejemplo considérese la siguiente tabla:

N° Individual	**String**	**Función Objetivo**	*Fracción Importancia*	**Fracción Acumulada**
X_1	01110110	130	16.4	16.4
X_2	00111001	42	5.3	21.7
X_3	11101010	320	40.3	62.0
X_4	01010001	227	28.6	90.6
X_5	11101010	75	9.4	100.0
Total		794	100.0	

Figura 4-1: Ejemplo Tabla de valores AG

Con los valores anteriores, se simula la operación de una ruleta, la cual con cada giro permite seleccionar un individuo para reproducirlo a la nueva población. Debido a que los individuos más aptos (con mayor

valor de la función objetivo) se les asignan una mayor área, se espera que estos sean seleccionados con mayor frecuencia.

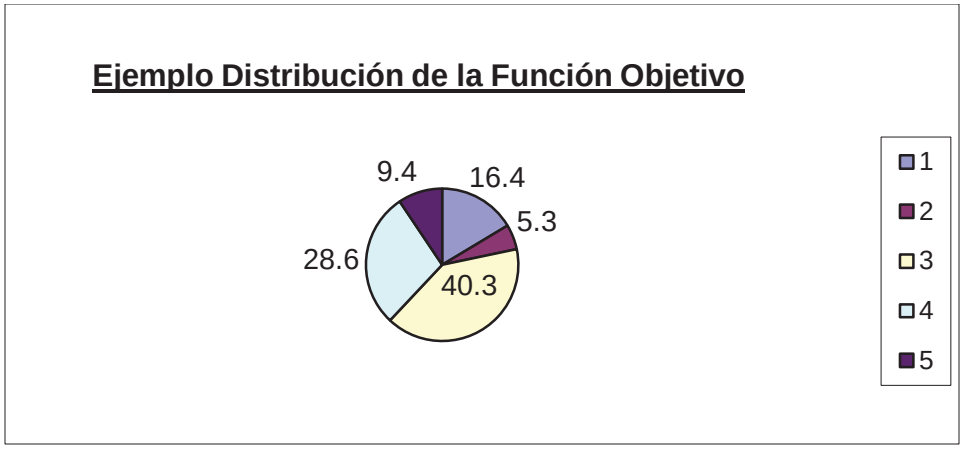

Figura 4-2: Ejemplo Distribución Función Objetivo

El funcionamiento de la ruleta se simula generando números aleatorios uniformes. En el ejemplo anterior, si se generan los números 0.348, 0.821 y 0.156, estos dan origen a los individuos X_3, X_4 y X_1, respectivamente como miembros de la nueva población a generar.

ii) **Cruzamiento *(Crossover):*** Transfiere las características deseables de las soluciones encontradas en una generación a la siguiente. Estos operadores producen individuos viables, a partir de padres distintos viables, cada hijo debe mantener alguna de las características deseables de los padres. Existen diversas técnicas para implementar este operador: Crossover, OX, PMX, siendo la más utilizada el Crossover, donde el cruzamiento se realiza en dos etapas. En la primera se determinan parejas de individuos a cruzar y en la segunda se determina un punto para realizar el intercambio entre los individuos de cada pareja, a partir de ese punto se mezclan los strings con el objeto de generar dos nuevos individuos:

Individuos	*Cruzamiento*	Descendientes
1011|0100	➔	10111110
1100|1110		11000100

Punto de Intercambio generado aleatoriamente

Figura 4-3: Ejemplo Cruzamiento

iii) **Mutación:** Transfiere las características de diversidad a la población. Estos operadores permiten alterar aleatoriamente los códigos genéticos de los individuos generados en una nueva población, lo que normalmente elimina buenas soluciones. Sin embargo el proceso de mutación es beneficioso por que puede producir individuos mejorados cuya probabilidad de supervivencia aumenta y por otra parte evita que el algoritmo genere siempre la misma solución. Goldberg recomienda utilizar una probabilidad de 0,8 % para ese operador [8].

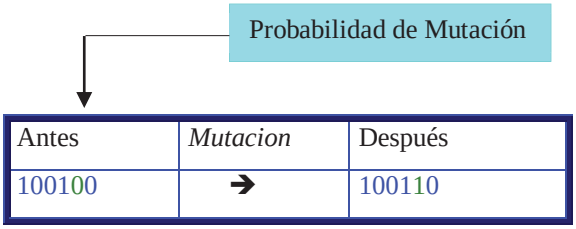

Figura 4-4: Ejemplo Mutación

e) **Función de Evaluación**: Es una medida de la bondad de los individuos en cuanto a la solución que codifican. Esta medida se utiliza como parámetro de los operadores y guía la obtención de nuevas poblaciones. Ésta corresponde a la función objetivo que se desea optimizar.

4.2 Forma general de un Algoritmo Genético:

La forma general de un AG, se grafica mediante el siguiente diagrama:

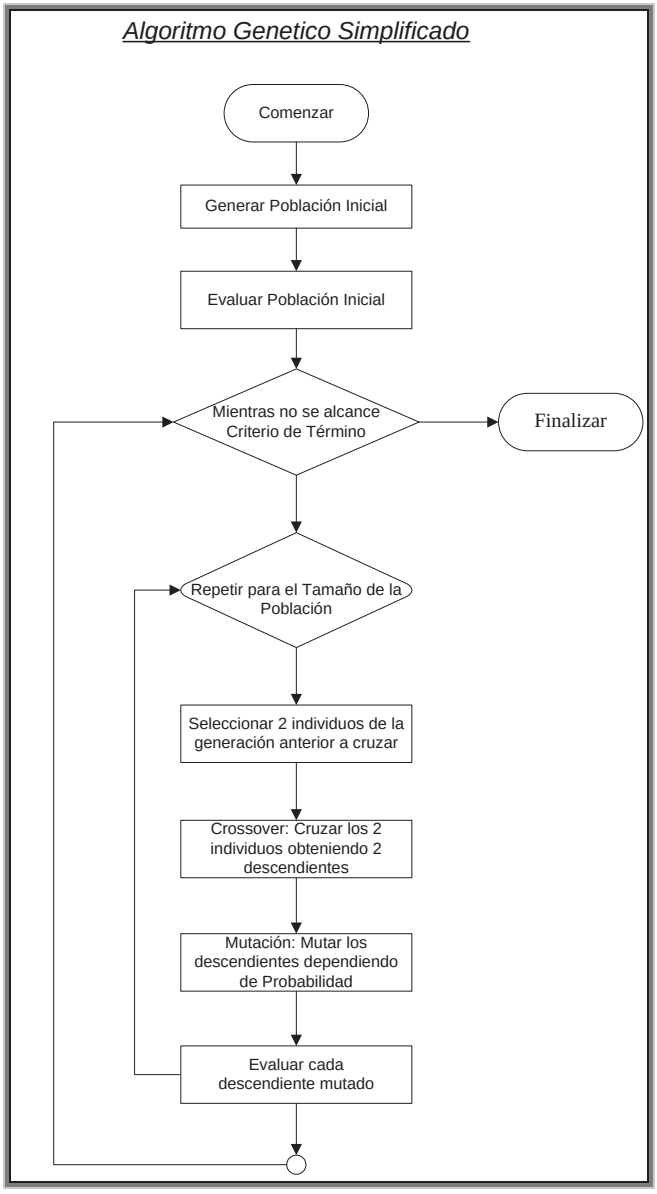

Figura 4-5: Diagrama Algoritmo Genético

A continuación se describen cada uno de los pasos que conforman el AG y los métodos empleados para su implementación:

1. **Codificación:** Determinar la estructura de la cromosoma que codificará las variables de decisión en su dominio y representarlas por un string de caracteres, que permita su utilización en el AG, de modo de asignarle su correspondiente valor de la función de *fitness.*

2. **Generar población inicial**: Consiste en obtener una población de individuos inicial en forma aleatoria, para lo cual se generan cadenas de combinaciones arbitrarias, que representen posibles soluciones del problema, utilizando el operador de selección.

3. **Evaluar Población Inicial**: Cada individuo de la población inicial es evaluado con el objeto determinar el valor de la función objetivo.

4. **Generar Nueva Población:** Se Aplican a la población actual los operadores genéticos (Selección, Cruzamiento, Mutación) para generar una nueva población, tratando de que se reproduzcan aquellos individuos que presentan mejores valores de la función objetivo.

5. **Evaluar Criterio:** Repetir desde el paso 3, hasta que se haya cumplido cierto criterio de convergencia (por ejemplo que la varianza de los individuos este por debajo de cierto valor de umbral). O bien una cantidad predeterminada de iteraciones.

5. Descripción de la Implementación

Para resolver el problema de optimización de enrutamiento se implementó un programa en lenguaje C, tomando las recomendaciones de [17], generándose un programa C que implementa un algoritmo genético, sencillo y eficiente con implementación de mutación gaussiana. El programa se denomina Agantnet.cpp, el cual a su vez llama a un programa Java que implementa el algoritmo AntNet, ambos se describen a continuación:

5.1. Diseño de Estructuras

El algoritmo genético implementa genes que representan los individuos de la población, en este caso cada individuo contiene los valores de un conjunto seleccionado de parámetros del algoritmo AntNet y la función objetivo a maximizar es el throughput de la red, es decir la cantidad de paquetes correctamente direccionados en un periodo de tiempo. Los parámetros del algoritmo AntNet y sus límites de variación considerados se ilustran en la tabla 5.1:

Parámetro	**Descripción**	**Límite inferior**	**Límite superior**
packetSize	Tamaño de paquetes en la red [bits]	9.000	12.000
simTime	Tiempo de la simulación de la red [seg]	800	1.100
Alpha	Importancia Relativa de la heurística en la feromona [%]	0.3	0.7
processTime	Tiempo de proceso del paquete en cada nodo [ms]	0.1	0.4

Figura 5-1: Parámetros de AntNet considerados

Para el problema de calibrar un algoritmo AntNet mediante un algoritmo genético la representación de parámetros se ilustra en la figura 5.2:

packetSize		simTime		Alpha		processTime		fitness	rfitness	cfitness
Límite Inferior [bits]	Límite Superior [bits]	Límite Inferior [seg]	Límite Superior [seg]	Límite Inferior [%]	Límite Superior [%]	Límite Inferior [ms]	Límite Superior [ms]	[bits/s]	[bits/s]	[bits/s]

Figura 5-2: Representación de Parámetros de AntNet

Con los parámetros anteriores se implementó una estructura de genotipo en C que permite codificar un número variable de parámetros, en este caso 4 pero se permiten más. La estructura contempla 3 vectores donde para cada variable se almacena su valor y sus límites inferior y superior de manera de generar valores solo en los rangos permitidos, además contempla el valor del *fitness*, el *fitness* acumulado y el valor relativo del *fitness* respecto del total. Por último la población se representa como un arreglo de esta estructura, donde el último elemento almacena el mejor de cada población, esto se ilustra en la siguiente figura 5.3:

```
struct genotype /* genotipo para dada individuo de la poblacion *{
double gene[NVARS];       /* Vector de variables */
  double fitness;         /* fitness */
  double delay;           /* Demora del paquete */
  double upper[NVARS];  /* limite superior de la variable */
  double lower[NVARS];  /* limite inferior de la variable*/
  double rfitness;        /* fitness relativo*/
  double cfitness;        /* fitness acumulado*/
};
struct genotype population[POPSIZE+1];    /* population */
```

Figura 5.3: Estructura de genotipo

5.2. Parámetros Algoritmo Genético

Para realizar las pruebas de implementación, basado en las recomendaciones de [17], se utilizaron los siguientes valores de sus parámetros:

- Tamaño Población: 50
- Número máximo de generaciones: 1.000
- Probabilidad de Cruzamiento: 0.8
- Probabilidad de Mutación: 0.15

5.3. Programa Agantnet.cpp

El código fuente de este programa se encuentra en Anexo 1, su función principal implementa los pasos propios de un algoritmo genético, tal como se indica en la siguiente figura:

Programa Agantnet

```
int main(void) {
      generation = 0;
      initialize();
      evaluate();
      keep_the_best();
      while(generation<MAXGENS){
              generation++;
              select();
              crossover();
              mutate();
              report();
              evaluate();
              elitist();
      }

   }
```

Las principales funciones de este programa son:

- ***Initialize():*** Lee valores permitidos de variables y genera la población inicial en forma aleatoria en los rangos permitidos. Inicializa los valores de los genes en su dominio e inicializa en cero el valor del *fitness*. Lee los rangos de valores de los parámetros desde un archivo de texto 'Agantnetdata.txt', de acuerdo al formato ilustrado en anexo 3. Genera aleatoriamente los valores entre sus límites, para cada gen de cada individuo en la población.

- ***evaluate():*** Determina el valor del fitness de cada individuo ejecutando el algoritmo Antnet con los valores de los parámetros generados para el individuo. Para cada individuo de la población genera un string de llamada al programa java de Antnet, con una instrucción del tipo:
  system("java -jar C:\Programs\AntNetSimulation\dist\AntNetSimulation.jar 9123.00 964.10 0.6332 0.2266")
  El programa java al iniciarse toma los valores de los parámetros, se ejecuta y retorna con una instrucción del tipo *System.exit((int) total).* La variable total almacena, ambos, el valor del throughput y el valor de la demora promedio obtenidos de la ejecución. Al retornar el control a la función *evaluate(),* esta interpreta los valores recibidos y los almacena en la estructura de genotipo utilizada como *fitness y delay*.

- ***keep_the_best():*** Almacena el mejor individuo de cada población. El algoritmo está diseñado para maximizar, es decir almacena el individuo con mayor *fitness*, bajo un modelo elitista, de modo de asegurar que el mejor miembro sobrevive en cada población.

- ***select():*** Aplica el operador de selección de individuos para generar una nueva población, aplicando el método de selección por ruleta, es decir eligiendo en forma proporcional al individuo con mejor *fitness* y aplicando la función de crossover.

- ***crossover():*** Aplica el operador de cruzamiento, seleccionado dos padres y generando dos nuevos hijos.

- ***mutate():*** Aplica el operador de mutación para generar perturbaciones aleatorias en las cromosomas de los individuos.

- ***elitist():*** Selecciona el mejor miembro de la población anterior, en caso que la generación actual no tenga un miembro mejor al de la generación anterior.

.

5.4. Clase AntNet

Para implementar el algoritmo AntNet, se creó la clase Antnet en Java, que implementa un grafo para representar una red IP y además la clase implementa los procesos de agentes artificiales (hormigas) que cruzan la red.

Los parámetros que utiliza esta clase son:

- numberOfVertices: Número de nodos de la red en el gráfico.
- packetSize: Tamaño del paquete para todos los paquetes utilizados en la simulación.
- simTime: Tiempo de corrida de la simulación.
- Alfa: valor que pesa la importancia relativa de la información heurística en relación a los valores de la feromona.
- processTime Tiempo de proceso en cada nodo al tratamiento de los paquetes.

El código fuente Java de esta clase se encuentra en Anexo 2.

6. Pruebas de Aplicación

6.1. Red Experimental

Para iniciar los experimentos de laboratorio y con el objeto de probar el funcionamiento de los programas en su totalidad, se realizaron pruebas con la red de la figura 6.1, denominada Red Experimental, basado en lo recomendado por [18]:

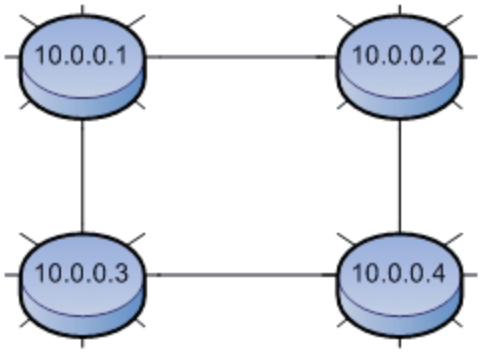

Figura 6.6-1: Red Experimental

En la red experimental se observa que hay 4 nodos, con las conexiones ilustradas, esta se implementa en la Clase Java AntNet, de acuerdo a lo ilustrado en la figura 6.2:

```
AntNet antnet = new AntNet(numberOfNodes, packetSize, simTime, alpha, processTime);
NetworkGraph graph = antnet.getNetworkGraph();
graph.makeEdge(0, 1, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(0, 2, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(1, 3, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 3, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.initializeGraph();
antnet.createDataPacketGenerator(deltaPackets, maxHops);
antnet.createAgentGenerator(graph.getNetworkVertices().get(0), deltaAgents, ttl);
antnet.createAgentGenerator(graph.getNetworkVertices().get(1), deltaAgents, ttl);
antnet.createAgentGenerator(graph.getNetworkVertices().get(2), deltaAgents, ttl);
antnet.createAgentGenerator(graph.getNetworkVertices().get(3), deltaAgents, ttl);
activate(antnet);
```

Figura 6.2: Implementación java Red Experimental

6.1.1. Resultados Red Experimental

La aplicación de los programas que forman la metaheurística propuesta, a la red experimental arrojó los siguientes resultados:

Número	Mejor	Demora	Fitness	Desviación	Parámetros Mejor Individuo			
Generación	Fitness [bits/s]	[seg]	Promedio [bits/s]	Estándar [bits/s]	packetSize [bits]	simTime [seg]	Alpha [%]	processTime [ms]
1	8934.402	0.542	4448.407	2030.962	11826	1082.6	0.5292	0.1123
2	9031.447	0.501	5216.711	2054.682	11706	1070	0.5544	0.106
3	9067.278	0.411	4708.764	2195.762	10500	1027.1	0.5632	0.109
4	9610.063	0.535	5414.768	2301.431	10500	1027.1	0.5632	0.106
5	9610.063	0.538	6403.988	2257.433	10500	1027.1	0.5632	0.106
6	10440.252	0.502	5961.618	2564.73	10434	1016.9	0.6908	0.106
7	10440.252	0.362	7196.374	1925.199	10434	1016.9	0.6908	0.106
8	10440.252	0.441	7467.722	1664.172	10434	1016.9	0.6908	0.106
9	11125.786	0.519	7420.66	2065.222	11187	1098.5	0.5352	0.106
10	11926.14	0.365	8432.644	1870.003	10293	1067	0.3684	0.1381
11	11926.14	0.365	8417.595	1457.447	10293	1067	0.3684	0.1381
12	11926.14	0.365	8238.641	2175.13	10293	1067	0.3684	0.1381
13	11926.14	0.365	8132.428	1060.08	10293	1067	0.3684	0.1381
14	11926.14	0.365	8250.367	1209.563	10293	1067	0.3684	0.1381
15	11926.14	0.365	8178.273	1504.711	10293	1067	0.3684	0.1381
16	11926.14	0.365	8191.443	1633.257	10293	1067	0.3684	0.1381
17	11926.14	0.365	7545.329	2003.726	10293	1067	0.3684	0.1381
18	11926.14	0.365	7451.185	2317.093	10293	1067	0.3684	0.1381
19	11926.14	0.365	7493.762	1776.189	10293	1067	0.3684	0.1381
20	11926.14	0.365	7784.068	1756.489	10293	1067	0.3684	0.1381
21	11926.14	0.365	8126.809	1692.064	10293	1067	0.3684	0.1381
22	11926.14	0.365	8563.868	1729.603	10293	1067	0.3684	0.1381
23	11926.14	0.365	7434.892	1846.495	10293	1067	0.3684	0.1381
24	11926.14	0.365	8452.393	1435.821	10293	1067	0.3684	0.1381
25	11926.14	0.365	8170.103	2007.279	10293	1067	0.3684	0.1381
26	11926.14	0.365	8393.122	1721.695	10293	1067	0.3684	0.1381
27	11926.14	0.365	7873.319	1912.489	10293	1067	0.3684	0.1381
28	11926.14	0.365	8344.059	2039.911	10293	1067	0.3684	0.1381
29	11926.14	0.365	8226.166	1735.64	10293	1067	0.3684	0.1381
30	11926.14	0.365	8268.215	2028.356	10293	1067	0.3684	0.1381
31	11926.14	0.365	7413.089	2171.819	10293	1067	0.3684	0.1381
32	11926.14	0.365	8238.641	2175.13	10293	1067	0.3684	0.1381
33	11926.14	0.365	8132.428	1060.08	10293	1067	0.3684	0.1381
34	11926.14	0.365	8250.367	1209.563	10293	1067	0.3684	0.1381
35	11926.14	0.365	8178.273	1504.711	10293	1067	0.3684	0.1381
36	11926.14	0.365	8178.273	1504.711	10293	1067	0.3684	0.1381
37	11926.14	0.365	8178.273	1504.711	10293	1067	0.3684	0.1381

Best fitness = 12819.066

Tabla 6-1: Resultados de Aplicación en Red experimental

En esta prueba se observa que el algoritmo converge en la décima generación obteniendo un fitness total de 11.926,140 [b/s], Demora promedio de paquetes de 0.365 [s], con valores para los parámetros packetSize, simTime, alpha y processTime de 10293, 1067, 0.3684 y 0.1381 respectivamente.

6.2. Red NSFNET

La red Nsfnet (National Science Foundation Network, USA) representa la red de la Fundación Nacional de ciencias de Estados Unidos y es comúnmente utilizada, como una red típica para probar el desempeño de nuevos algoritmos de ruteo, consta de 14 nodos y 21 links, representada en la figura 6.3, y Tabla 6-2 Representación Red NSFNET, donde se ilustran los nodos y la demora entre ellos en [ms]:

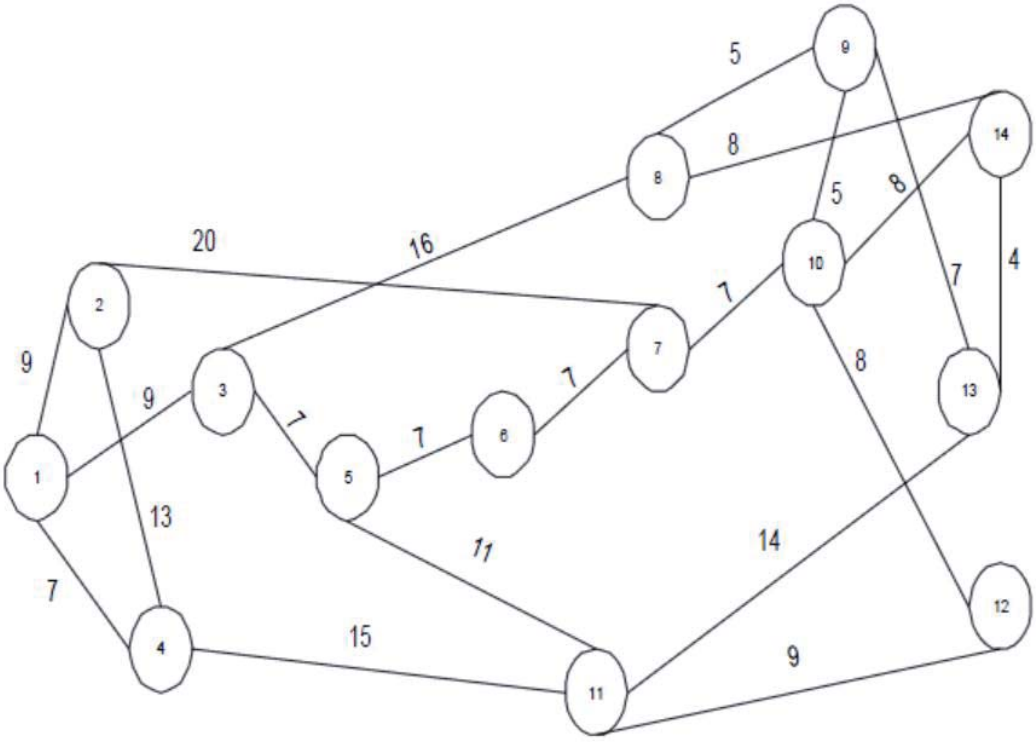

Figura 6-3: Red Nsfnet

Nodo Origen	Nodo Destino	Demora [ms]
1	2	9
1	3	9
1	4	7
2	4	13
2	7	20
3	5	7
3	8	16
4	11	15
5	6	7
5	11	11
6	7	7
7	10	7
8	9	5
8	14	8
9	10	5
9	13	7
10	12	8
10	14	8
11	12	9
11	13	14
13	14	4

Tabla 6-2 Representación Red NSFNET

En la red nsfnet se observa que hay 14 nodos, con las conexiones ilustradas en la Tabla 6-2 Representación Red NSFNET, esta red se implementa en la Clase Java AntNet, de acuerdo a lo ilustrado en la figura 6.4:

```
int numberOfNodes = 14;
AntNet antnet = new AntNet(numberOfNodes, packetSize, simTime, alpha, processTime);
NetworkGraph graph = antnet.getNetworkGraph();
graph.makeEdge(0, 1, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(0, 2, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(0, 3, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(1, 3, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(1, 6, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 4, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(3, 10, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(4, 5, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
```

```
graph.makeEdge(4, 10, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(5, 6, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(6, 9, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(7, 8, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(7,13, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(8, 9, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(8, 12, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(9, 11, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(9, 13, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(10, 11, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(10, 12, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(12, 13, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.initializeGraph();
antnet.createDataPacketGenerator(deltaPackets, maxHops);
for (int i=0; i<numberOfNodes;i++) {
  antnet.createAgentGenerator(graph.getNetworkVertices().get(i), deltaAgents, ttl);
}
activate(antnet);
```

Figura 6.4: Implementación java Red NSFNET

6.2.1. Resultados Red NSFNET

La aplicación de los programas de la metaheurística propuesta, a la red NSFNET arrojó los siguientes resultados:

Numero Generación	Mejor Fitness [bits/s]	Demora [seg]	Fitness Promedio [bits/s]	Desviación Estándar [bits/s]	Parámetros Mejor Individuo			
					packetSize [bits]	simTime [seg]	Alpha [%]	processTime [ms]
1	9023.449	0.631	4820.024	2032.873	11826	1082.6	0.5292	0.1123
2	9666.667	0.644	5197.351	2085.891	9645	1018.7	0.5544	0.109
3	9666.667	0.76	5774.767	2022.883	9645	1018.7	0.5544	0.109
4	10264.151	0.663	5984.878	2243.289	10392	1082.6	0.5292	0.106
5	10264.151	0.73	6280.18	2048.543	10392	1082.6	0.5292	0.106
6	10264.151	0.641	6681.974	1935.896	10392	1082.6	0.5292	0.106
7	10264.151	0.479	6944.593	2033.987	10392	1082.6	0.5292	0.106
8	10531.447	0.772	7189.701	1849.68	11454	1082.6	0.4428	0.106
9	11154.088	0.871	7039.772	2082.398	10506	1082.9	0.5572	0.106
10	11154.088	0.838	7280.05	1638.416	10506	1082.9	0.5572	0.106
11	11154.088	0.802	7183.525	1972.853	10506	1082.9	0.5572	0.106
12	11855.469	0.675	7215.838	2245.096	10506	1082.9	0.688	0.1024
13	11855.469	0.732	6972.485	2588.001	10506	1082.9	0.688	0.1024
14	11855.469	0.689	7800.719	2214.196	10506	1082.9	0.688	0.1024
15	11855.469	0.689	7711.481	1985.423	10506	1082.9	0.688	0.1024
16	11855.469	0.689	7951.613	1811.31	10506	1082.9	0.688	0.1024

Numero Generación	Mejor Fitness [bits/s]	Demora [seg]	Fitness Promedio [bits/s]	Desviación Estándar [bits/s]	Parámetros Mejor Individuo			
					packetSize [bits]	simTime [seg]	Alpha [%]	processTime [ms]
17	11855.469	0.689	7483.64	2030.731	10506	1082.9	0.688	0.1024
18	11855.469	0.689	7446.146	2132.757	10506	1082.9	0.688	0.1024
19	11855.469	0.689	7765.052	1840.18	10506	1082.9	0.688	0.1024
20	11855.469	0.689	7484.392	2052.83	10506	1082.9	0.688	0.1024
21	11855.469	0.689	8214.971	1577.377	10506	1082.9	0.688	0.1024
22	11855.469	0.689	8176.637	2231.752	10506	1082.9	0.688	0.1024
23	11855.469	0.689	8244.591	2074.245	10506	1082.9	0.688	0.1024
24	11855.469	0.689	8177.233	2184.814	10506	1082.9	0.688	0.1024
25	11855.469	0.689	7613.669	2478.124	10506	1082.9	0.688	0.1024
26	11855.469	0.689	7759.198	2213.744	10506	1082.9	0.688	0.1024
27	11855.469	0.689	7456.246	2132.302	10506	1082.9	0.688	0.1024
28	11855.469	0.689	7005.842	2397.861	10506	1082.9	0.688	0.1024
29	16222.249	0.603	7162.209	2729.142	10506	1082.9	0.688	0.1024
30	16222.249	0.603	7162.209	2682.674	10506	1082.9	0.688	0.1024
31	16222.249	0.603	7125.922	2645.382	10776	1096.1	0.5908	0.1021
32	16222.249	0.603	7162.209	2729.142	10776	1096.1	0.5908	0.1021
33	16222.249	0.603	7640.377	2341.122	10776	1096.1	0.5908	0.1021
34	16222.249	0.603	7716.578	2089.278	10776	1096.1	0.5908	0.1021
35	16222.249	0.603	7450.278	2517.317	10776	1096.1	0.5908	0.1021
36	16222.249	0.603	7196.966	2673.84	10776	1096.1	0.5908	0.1021
37	16222.249	0.603	7363.473	2322.179	10776	1096.1	0.5908	0.1021
38	16222.249	0.603	8128.762	2283.15	10776	1096.1	0.5908	0.1021
39	16222.249	0.603	8427.046	2026.733	10776	1096.1	0.5908	0.1021
40	16222.249	0.603	8087.415	2431.997	10776	1096.1	0.5908	0.1021

Best fitness = 16222.249

Tabla 6-3: Resultados Red NSFNET

La Tabla 6-3: Resultados Red NSFNET, muestra que el algoritmo converge en la 29ª generación obteniendo un fitness total de 16222.249 [b/s], Demora promedio de los paquetes de 0.603 [s], con valores para los parámetros packetSize, simTime, alpha y processTime de 10706, 1096,1, 0,5908 y 0.1021 respectivamente.

6.3. Red NTTNET

La red NTTNET (Nippon Thelephone Telegraph of Japan) representa la red nacional de telefonía y telégrafos de Japón, y al igual que el caso de NFSNET es comúnmente utilizada, como una red típica para probar el desempeño de nuevos algoritmos de ruteo, consta de 55 nodos links, representada en la figura 6.5, donde se ilustran los nodos y la demora entre ellos en [ms]:

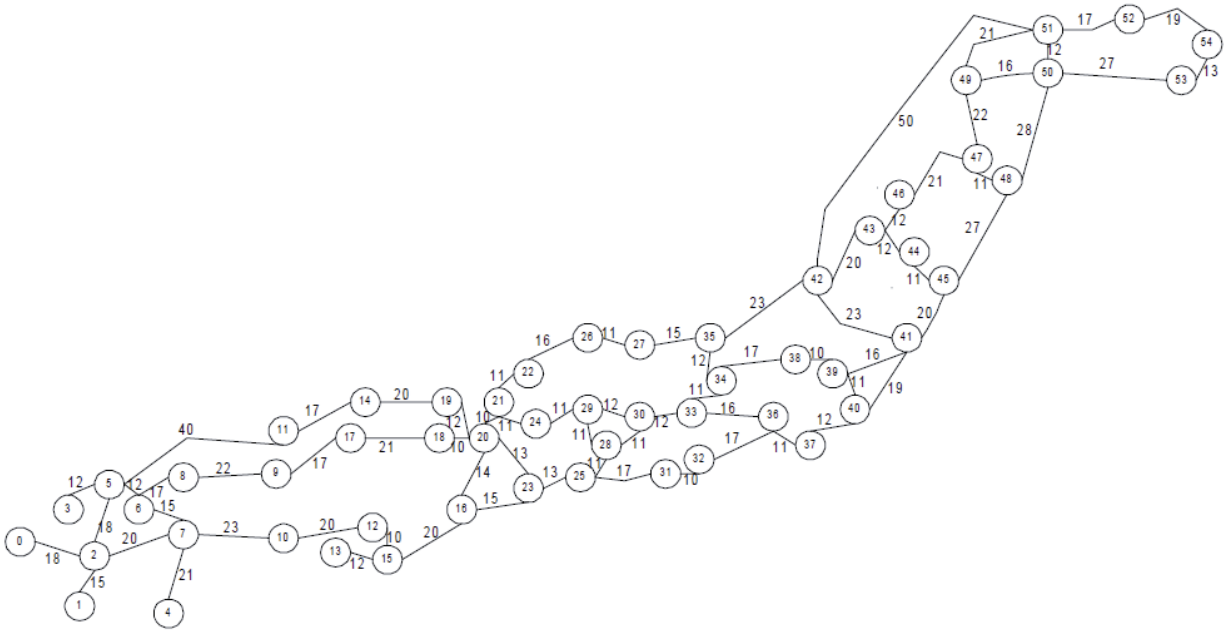

Figura 6-5: Red NTTNET

Nodo Origen	Nodo Destino	Demora [ms]
0	2	18
1	2	15
2	5	18
2	7	20
3	5	12
4	7	21
5	6	12
5	11	40
6	7	15
6	8	17
7	10	23
8	9	22
9	17	17

Nodo Origen	Nodo Destino	Demora [ms]
10	20	12
11	14	17
12	15	10
13	15	12
14	19	20
15	16	20
16	20	14
16	23	15
17	18	21
18	20	10
19	20	12
20	21	10
20	23	13

Nodo Origen	Nodo Destino	Demora [ms]
21	22	11
21	24	11
22	26	16
23	25	13
24	29	11
25	28	11
25	31	10
26	27	11
27	35	15
28	30	11
29	30	12
30	33	12
31	32	10
32	36	17
33	34	11
33	36	16
34	35	12
34	38	17
35	42	23
36	37	11
37	40	12
38	39	10
39	40	11

Nodo Origen	Nodo Destino	Demora [ms]
39	41	16
40	41	19
41	42	23
41	45	20
42	51	50
42	43	20
43	44	12
43	46	12
44	45	11
45	48	27
46	47	21
47	48	11
47	49	22
48	50	28
49	50	16
49	51	21
50	51	12
50	53	27
51	52	17
52	54	19
53	54	13

En la red NTTNET se observa que hay 55 nodos, con las conexiones ilustradas, esta se implementa en la Clase Java AntNet, de acuerdo a lo ilustrado en la figura 6.6:

```
 int numberOfNodes = 55;
AntNet antnet = new AntNet(numberOfNodes, packetSize, simTime, alpha, processTime);
NetworkGraph graph = antnet.getNetworkGraph();
graph.makeEdge(0, 2, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(1, 2, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 5, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(3, 5, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(4, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(5, 6, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(5,11, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(6, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(6, 8, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
```

```
graph.makeEdge(7,10, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(8, 9, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(9,17, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(10,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(11,14, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(12,15, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(13,15, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(14,19, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(15,16, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(16,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(16,23, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(17,18, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(18,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(19,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(20,21, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(20,23, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(21,22, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(21,24, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(22,26, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(23,25, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(24,29, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(25,28, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(25,31, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(26,27, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(27,35, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(28,30, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(29,30, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(30,33, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(31,32, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(32,36, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(33,34, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(33,36, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(34,35, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(34,38, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(35,42, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(36,37, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(37,40, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(38,39, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(39,40, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(39,41, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(40,41, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(41,42, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(41,45, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(42,51, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(42,43, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(43,44, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(43,46, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(44,45, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
```

```
graph.makeEdge(45,48, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(46,47, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(47,48, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(47,49, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(48,50, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(49,50, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(49,51, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(50,51, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(51,52, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(52,54, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(53,54, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.initializeGraph();
antnet.createDataPacketGenerator(deltaPackets, maxHops);
for (int i=0; i<numberOfNodes;i++) {
  antnet.createAgentGenerator(graph.getNetworkVertices().get(i), deltaAgents, ttl);
}
activate(antnet);
```

Figura 6.6: Implementación java Red NTTNET

6.3.1. Resultados Red NNTNET

La aplicación de los programas de la metaheurística propuesta a la red NTTNET arrojó los siguientes resultados:

Numero Generación	Mejor Fitness [bits/s]	Demora [seg]	Fitness Promedio [bits/s]	Desviación Estándar [bits/s]	Parámetros Mejor Individuo			
					packetSize [bits]	Generación	Fitness [bits/s]	[seg]
1	24482.8	0.676	14547	3921.77	9110	1946	0.554	0.3293
2	24470.5	0.432	10616	3473.54	9110	1905	0.607	0.3293
3	24470.5	0.287	11280	3254.66	9560	1905	0.607	0.3058
4	25670.4	0.233	12313	5236.15	9560	1905	0.607	0.2992
5	26455.8	0.645	15681	4022.56	9800	1905	0.607	0.2293
6	26455.8	0.391	15001	4285.36	9800	1905	0.607	0.2293
7	26455.8	0.664	15477	3973.56	9800	1905	0.599	0.2293
8	26455.8	0.233	11769	3006.58	9876	2320	0.599	0.2197
9	27232.9	0.366	11424	3348.6	9876	2320	0.599	0.2197
10	27232.9	0.756	12729	5190.25	9876	2320	0.599	0.2197
11	27232.9	0.543	12444	5015.06	9876	2320	0.599	0.2158
12	27232.9	0.762	14727	3071.76	9876	2320	0.599	0.2158
13	27232.9	0.387	12189	5745.41	1079	2320	0.596	0.2158
14	27498.7	0.607	14774	3743.64	10146	2320	0.596	0.2158
15	27498.7	0.534	14557	4272.09	10146	2187	0.589	0.2053
16	27498.7	0.707	14296	5588.34	10146	2187	0.589	0.2053
17	27498.7	0.439	14915	3482.55	10146	2187	0.589	0.2053
18	27498.7	0.711	14702	3057.74	10146	2187	0.589	0.2053
19	27997.5	0.798	12698	4467.31	10146	2187	0.589	0.2053

Numero Generación	Mejor Fitness [bits/s]	Demora [seg]	Fitness Promedio [bits/s]	Desviación Estándar [bits/s]	Parámetros Mejor Individuo			
					packetSize [bits]	Generación	Fitness [bits/s]	[seg]
20	27997.5	0.397	11844	4077.61	10146	2187	0.589	0.2053
21	27997.5	0.779	12374	5367.91	10146	2187	0.589	0.2053
22	28124.5	0.641	14539	4649.55	10146	2187	0.589	0.2052
23	28124.5	0.613	10195	4384.42	10146	2124	0.607	0.2052
24	28124.5	0.711	11045	3975.99	10225	2124	0.607	0.2011
25	28124.5	0.588	14960	4177.34	10225	2124	0.607	0.2011
26	31129.2	0.169	13151	3623.68	10225	2124	0.607	0.2011
27	31129.2	0.169	14353	5270.42	10225	2124	0.607	0.2011
28	31129.2	0.169	12230	5791.72	10225	2124	0.607	0.1944
29	31129.2	0.169	13771	5398.58	10225	2056	0.602	0.1944
30	32330.1	0.148	14037	5485.8	10225	2056	0.602	0.1944
31	32330.1	0.148	10392	3285.97	10225	2056	0.602	0.1944
32	32330.1	0.148	11954	4228.26	10225	2056	0.602	0.1944
33	38250.2	0.122	13951	5656.73	10442	2056	0.602	0.1935
34	38250.2	0.122	14124	3772.94	10442	2056	0.602	0.1935
35	38250.2	0.122	15341	3262.16	10442	2056	0.602	0.1935
36	38250.2	0.122	11993	4316.06	10442	2056	0.602	0.1921
37	38250.2	0.122	12053	4886.5	10442	2056	0.602	0.1921
38	38250.2	0.122	15775	4509.86	10442	2056	0.602	0.1921
39	38250.2	0.122	14032	4950.43	10442	2056	0.602	0.1921
40	38250.2	0.122	10596	3586.81	10442	2056	0.602	0.1921

Best fitness = 38250.228

Tabla 6-4: Resultados Red NTTNET

Los resultados obtenidos para la red NTTNET muestran que el algoritmo converge en la 33ª generación obteniendo un fitness total de 38250.228 [b/s], Demora promedio de los paquetes de 0.122 [s], con valores para los parámetros packetSize, simTime, alpha y processTime de 107442, 2056, 0,6022 y 0.1934 respectivamente.

7. Análisis de Resultados

Para evaluar la performance del algoritmo construido, los resultados se comparan con los generados por los algoritmos considerados el estado del arte en ruteo para telecomunicaciones aplicados a las redes NSFNET y NTTNET, según lo presentado por [12] estos son **AntNet Original, OSPF, SPF, BF, Q-R y PQ-R.** Para ello se graficará los valores de los resultados de los estudios previos, con los valores obtenidos por la metaheurística propuesta, denominada AgaAntNet.

7.1. Comparación Resultados Red NSFNET

7.1.1. Throughput Red NSFNET

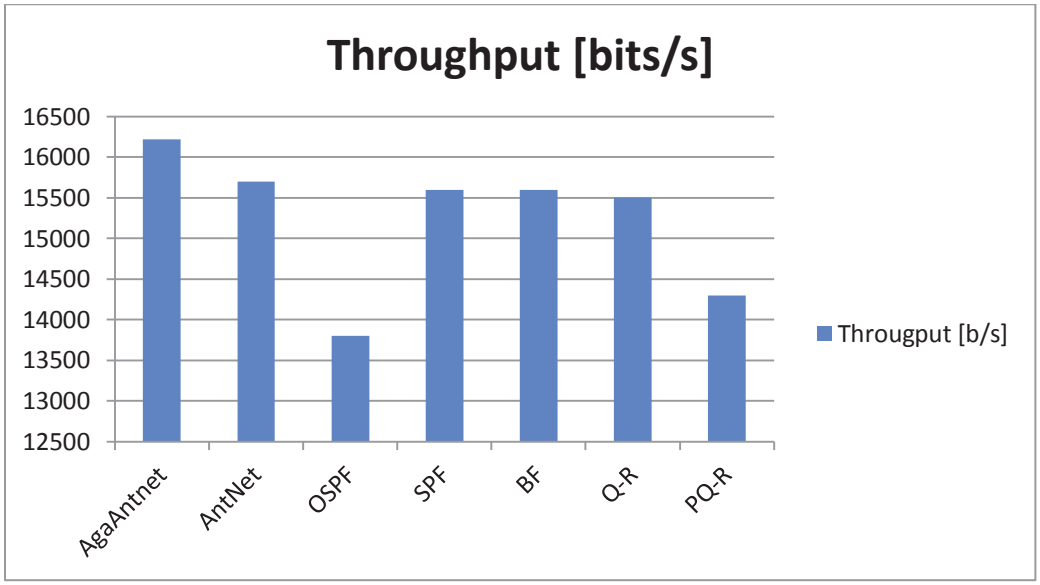

Figura 7.1: Resultados de Throughput para NSFNET

El valor óptimo del throughput para NFSNET, se obtiene después de 30 iteraciones de la metaheurística, cada una de ellas con una población de 100 individuos converge a un valor de 16.222,249 [bits/s], el cual es 3.32% superior al valor del algoritmo Antnet original y además es superior a los algoritmos tradicionales, con un 17.55% superior respecto de OSPF. Se observa además que los valores generados por el algoritmo AntNet original, es similar a los obtenidos con los algoritmos SPF, BF y Q-R y claramente OSPF y PQ-R obtienen los peores resultados.

7.1.2. Demora Red NSFNET

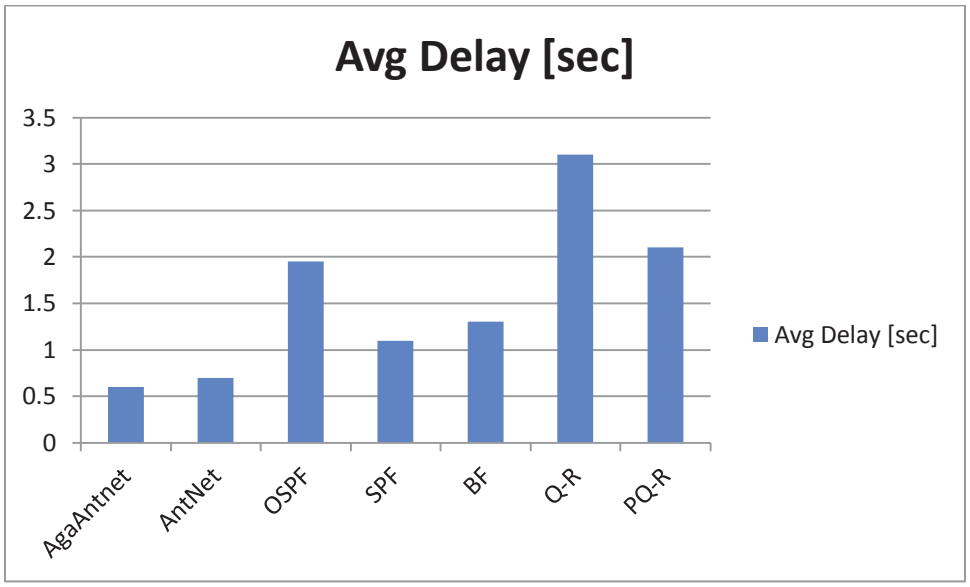

Figura 7.2: Resultados de Demora para NSFNET

El valor óptimo de la demora para NFSNET, obtenido después de 30 iteraciones del algoritmo genético, cada una de ellas con una población de 100 individuos fue de 0.603 [seg], el cual es 13.9% inferior al valor del algoritmo Antnet original y además es inferior a los algoritmos tradicionales, con un 80.55% superior respecto de Q-R. Q-R presenta el peor comportamiento con una demora promedio de 3.1 [seg], le siguen en orden decreciente PQ-R y OSPF con 2,1 [seg] y 1.95 [seg] respectivamente.

7.2. Comparación Resultados Red NTTNET

7.2.1. Throughput Red NTTNET

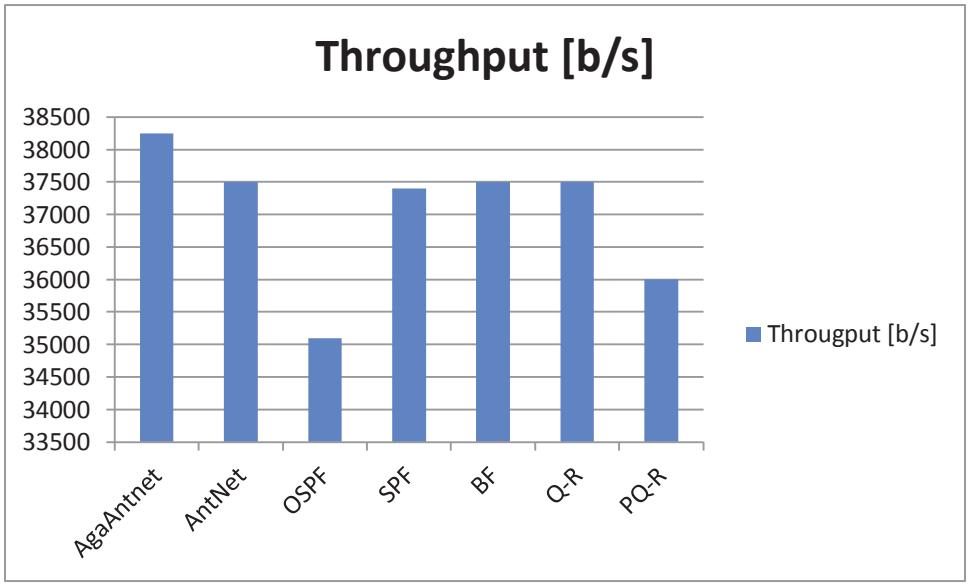

Figura 7.3: Resultados de Throughput para NTTNET

El valor óptimo del throughput obtenido para NTTNET, después de 30 iteraciones del algoritmo genético, cada una de ellas con una población de 100 individuos fue de 38.250.228 [b/s], el cual es 2% superior al valor del algoritmo AntNet original y además es superior a los algoritmos tradicionales, con un 8.97% superior respecto de OSPF, que presenta el peor rendimiento en este caso, seguido por PQ-R. Los algoritmos AntNet original, SPF, BF y Q-R presentan rendimientos similares, cuyas diferencias no son significativas.

7.2.2. Demora Red NTTNET

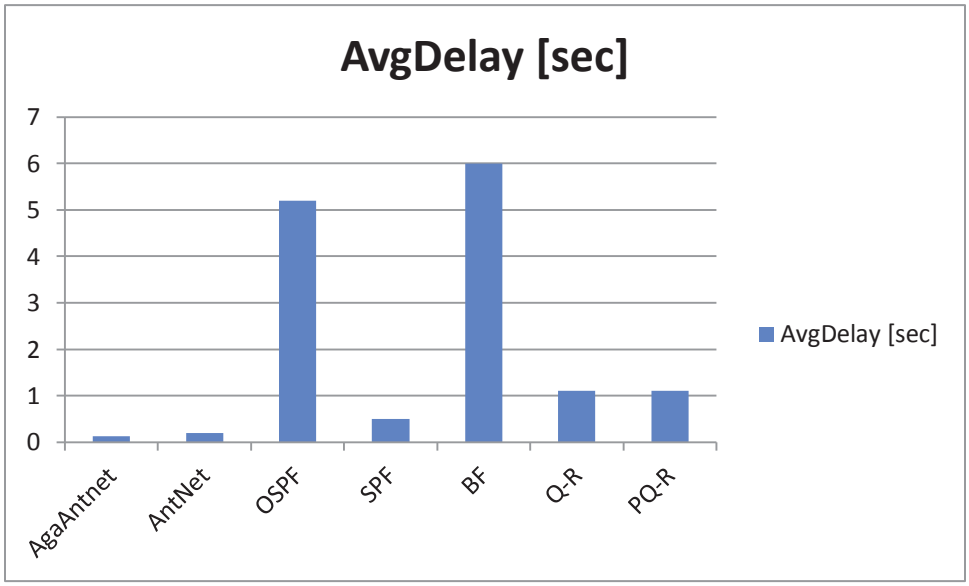

Figura 7.4: Resultados de Demora para NTTNET

El valor óptimo de la demora para NTTNET, obtenido después de 30 iteraciones del algoritmo genético AgaAntNet, cada una de ellas con una población de 100 individuos fue de 0.12 [s], el cual es 40% inferior al valor del algoritmo Antnet original y además es inferior a los algoritmos tradicionales, con un 98% superior respecto de B-F. El algoritmo B-F presenta la menor demora promedio con 6 [seg], seguido OSPF con 5.2 [seg]. Los algoritmos Q-R y PQ-R, presentan demoras similares en torno a 1.1 [seg] seguidos por SPF con 0.4 [seg].

8. Conclusiones

Durante el desarrollo de este trabajo se han alcanzado las siguientes conclusiones:

Cada día las redes de telecomunicaciones basadas en TCP/P enfrentan mayores exigencias de rendimiento, que les permitan satisfacer adecuadamente los mayores niveles de tráfico que requieren los usuarios, quienes crecientemente demandan mayores niveles de servicio para datos, streamings de videos y audio en tiempo real. Los routers deben contar con algoritmos de enrutamiento que les permitan adaptarse a las necesidades dinámicas de las redes IP. Es por ello que AntNet surgió como un algoritmo que ha demostrado ser mejor que los algoritmos de enrutamiento tradicionales.

El algoritmo AntNet se caracteriza por ser un proceso continuo de construcción de tablas de rutas, como resultado del proceso colectivo de aprendizaje, utilizando hormigas *Forward y Backward* denominadas agentes artificiales. Cada par de agentes *Forward-Backward* cuenta con la capacidad de hallar una buena ruta, en una forma estocástica y actualizar las tablas de ruteo para un camino dado. Para realizar esta elección probabilística, la hormiga *Backward* se basa en el rastro de feromona depositado en el grafo previamente por otras hormigas *Backward*, las hormigas *Forward* no depositan feromonas cuando se mueven. Aun cuando el par no puede resolver el problema global de optimización de la red por si solas, la interacción entre todos los agentes generados determina que finalmente se obtenga el óptimo del comportamiento de la red.

El componente básico del algoritmo AntNet es la hormiga artificial, que realiza múltiples labores en su recorrido a través de los nodos: busca soluciones válidas de costo mínimo, deposita feromona, tiene una memoria que almacena información sobre el camino seguido, la cual utiliza para i) construir soluciones válidas, ii) evaluar la solución generada, y iii) reconstruir el camino que ha seguido. La hormiga se mueve a través de la red aplicando una regla de transición, en función de tres parámetros: los rastros de feromona que están disponibles localmente, los valores heurísticos de la memoria privada de la hormiga, y las restricciones del problema.

Para diseñar e implementar un algoritmo de enrutamiento basado en AntNet que permita optimizar el enrutamiento en una red IP maximizando su throughput, se construyó un programa Java, que al ser invocado retiradamente, con sus parámetros adecuados (tamaño de paquetes en la red, tiempo de la simulación de la red, importancia relativa de la heurística en la feromona y tiempo de proceso del paquete en cada nodo) permite generar los parámetros considerados representativos del funcionamiento de una red: El throughput en [bits/s] y la demora promedio en [seg] de los paquetes en alcanzar un nodo de destino desde un nodo de origen,

Para la utilización del programa anterior, se diseñó e implementó una metaheurística de multinivel, en la cual un algoritmo genético calibra los parámetros del algoritmo AntNet. Los parámetros requeridos por AntNet son generados mediante el algoritmo genético, implementando en lenguaje C, especialmente diseñado con un esquema de cromosomas que codifican los parámetros de AntNet en rangos de valores mínimos y máximos de modo que el algoritmo genético, genera en forma aleatoria, diversos valores de los parámetros, invoca la ejecución de AntNet y almacena los resultados obtenidos para el throughput y la demora promedio.

El algoritmo desarrollado es general y versátil, ya que presenta la posibilidad de aplicarse a diferentes redes de comunicaciones con topología definida mediante una representación del grafo correspondiente al problema considerado. La metaheurística propuesta se probó exitosamente en una red Experimental pequeña, donde se observó que el algoritmo convergió rápidamente, después de 10 iteraciones, determinando los valores óptimos de los parámetros que optimizan el throughput de la red.

La metaheurística propuesta se aplicó a las redes clásicas de benchmarking de algoritmos de enrutamiento, como son NSFNET y NTTNET. Los resultados obtenidos fueron comparados con los valores de la aplicación de otros algoritmos tradicionales a las redes de benchmarking.

Los valores obtenidos por la metaheurística propuesta para las métricas de evaluación (throughput y demora promedio de paquetes), son por lo menos tan buenas como las de algoritmo AntNet original y son mejores que los valores de los algoritmos clásicos.

En los casos estudiados se obtuvieron mejoras en las métricas de comparación (throughput y demora promedio de los paquetes) respecto de los algoritmos clásicos de enrutamiento e incluso sobre la versión original de los autores de AntNet, lo cual indica que la calibración multinivel de AntNet por medio de un algoritmo genético fue exitosa

Al principio de cada simulación el algoritmo no presenta las mejores métricas, pero después de alrededor de 30 iteraciones, con una población de 100 individuos, se encuentra el rendimiento óptimo para cada red.

El valor óptimo del throughput para NFSNET, converge a un valor de 16.222,249 [bits/s], el cual es 3.32% superior al valor del algoritmo Antnet original y además es superior a los algoritmos tradicionales, con un 17.55% superior respecto de OSPF. Se observa además que los valores generados por el algoritmo AntNet original, es similar a los obtenidos con los algoritmos SPF, BF y Q-R y claramente OSPF y PQ-R obtienen los peores resultados

El valor óptimo de la demora para NFSNET, converge a 0.603 [seg], el cual es 13.9% inferior al valor del algoritmo Antnet original y además es inferior a los algoritmos tradicionales, con un 80.55% superior respecto de Q-R. Q-R presenta el peor comportamiento con una demora promedio de 3.1 [seg], le siguen en orden decreciente PQ-R y OSPF con 2,1 [seg] y 1.95 [seg] respectivamente.

El valor óptimo del throughput obtenido para NTTNET, converge a 38.250.228 [b/s], el cual es 2% superior al valor del algoritmo AntNet original y además es superior a los algoritmos tradicionales, con un 8.97% superior respecto de OSPF, que presenta el peor rendimiento en este caso, seguido por PQ-R. Los algoritmos AntNet original, SPF, BF y Q-R presentan rendimientos similares, cuyas diferencias no son significativas.

El valor óptimo de la demora para NTTNET, converge a 0.12 [s], el cual es 40% inferior al valor del algoritmo Antnet original y además es inferior a los algoritmos tradicionales, con un 98% superior respecto de B-F. El algoritmo B-F presenta la menor demora promedio con 6 [seg], seguido OSPF con 5.2 [seg]. Los algoritmos Q-R y PQ-R, presentan demoras similares en torno a 1.1 [seg] seguidos por SPF con 0.4 [seg].

Como conclusión general se puede indicar que las metaheurísticas constituyen una poderosa herramienta que permite, explorar y explotar algoritmos heurísticos utilizados para la resolución de problemas complejos sin soluciones analíticas conocidas. Mediante la implementación de una heurística en forma general, como es caso del algoritmo genético implementado en este proyecto, y la adecuada codificación de sus parámetros identificando sus valores máximos y mínimos, es posible generalizar la implementación de la metaheurística basada en un algoritmo genético para utilizarlo como guía en la ejecución de otros algoritmos heurísticos en otros problemas distintos al de enrutamiento en redes estudiado en este proyecto, para ello bastará con adaptar cada heurística a utilizar para que al finalizar cada ejecución retorne el valor obtenido para su función objetivo, el cual será utilizado como *fitness* en el algoritmo genético.

9. Bibliografía

[1] M. Dorigo y T. Stutzle, Ant Colony Optimization, Boston: The MIT Press, 2004.

[2] E. W. Dijkstra, «A note on two problems in connection with graphs,» *Numerische Mathematik,* nº 1, pp. 267-271, 1959.

[3] F. Ducatelle, G. Di Caro y L. Gambardella, «Principles and applications of swarm intelligence for adaptive routing in telecommunications networks,» nº 4, 2010.

[4] G. Di Caro, Ant Colony Optimization and its Application to Adaptive Routing in Telecommunication Networks, 2004: Universite Libre de Bruxelles.

[5] M. Dorigo y G. Di Caro., «The Ant Colony Optimization meta-heuristic,» *New Ideas in Optimization,* 1999.

[6] G. L. Ibrahim H. Osman, «Metaheuristics: A bibliography,» *Annals of Operational Research,* vol. 63, pp. 513-628, 1996.

[7] P.-P. Grassé, «La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. La theorie de la stigmergie ssai dinterpretation du comportement des termites constructeurs.,» *Insectes Sociaux,* vol. 6, nº 1, pp. 41-80, 10 Marzo 1959.

[8] D. Golberg, Genetic Algorithms in Search Optimization and Machine Learning, Boston: Addison-Wesley Longman Publishing Co., Inc., 1989.

[9] J. H. Holland, Adaptation in Natural and Artificial Systems, Ann Arbor: University of Michigan Press, 1975.

[10] C. Darwin, El origen de las especies, Madrid: Espasa, 2008.

[11] C. R. Gen Mitsuo, Genetic Algorithms & Engineering Design, New York: John Wiley & Sons, Inc., 1997.

[12] G. Di Caro y M. Dorigo, «AntNet: Distributed Stigmergetic Control for Communications Networks,» *Research, Journal of Artificial Intelligence,* pp. 317-365, 1998.

[13] M. Dorigo, E. Bonabeau y G. Theraulaz, Swarm Intelligence From Natural to Artificial System, New York: Oxford University Press, 1999.

[14] J. Labrador y M. Aguilar, «Un algoritmo de enrutamiento distribuido para redes de comunicación basado en sistemas de hormigas,» *IEEE LATIN AMERICA TRANSACTIONS,* pp. 616-625, 2007.

[15] A. Tanenbaum, Computer Networks, Englewood Cliffs: Prentice Hall, 1996.

[16] M. Dorigo y T. Stutzle, «The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and

Advances,» de *Handbook of Metaheuristics*, Kluwer Academic Publisher, 2002.

[17] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Berlin: Springer, 1999.

[18] S. Berj, Simulering of Antnet en routingsalgoritme baseret på ACO metaheuristikken, Roskilde: Roskilde Universitetscenter, 2007.

[19] B. Benjamín y R. Sosa, «A New approach for AntNet routing,» National University of Asuncion, Paraguay.

Anexo 1: Listado programa Agantnet

```
#include "stdafx.h"

/*************************************************************************/
/* Implementacion de un algoritmo genetico para parametrizar un algoritmo */
/* Antnet para optmizar el rendimiento de un red IP                        */
/* El fitness utilizado es el troughput de la red                          */
/*************************************************************************/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Parametros generales del algoritmo genetico */

#define POPSIZE 50              /* Tamaño de la poblacion */
#define MAXGENS 30              /* Max. numero de generaciones */
#define NVARS 4                 /* No. de variables reales*/
#define PXOVER 0.8              /* Probabilidad de crossover */
#define PMUTATION 0.15          /* Probabilidad de mutation */
#define TRUE 1
#define FALSE
int generation;                 /* numero generacion */
int cur_best;                   /* mejor individuo */
FILE *galog;                    /* archivo de salida */
FILE *logpob;                   /* archivo de salida detalle de poblacion*/
struct genotype /* genotypo (GT), cada individuo de la poblacion */
{
  double gene[NVARS];       /* String de variables */
  double fitness;           /* Fitness (throughput) */
  double delay;             /* Demora promedio de los paquetes  */
  double upper[NVARS];      /* Limite superior de cada variable */
  double lower[NVARS];      /* Limite inferior de cada variable */
  double rfitness;          /* fitness relativo*/
  double cfitness;          /* fitness acumulado */
};
struct genotype population[POPSIZE+1];      /* poblacion */
struct genotype newpopulation[POPSIZE+1]; /* nueva poblacion para reemplazar la generacion anterior */
/* Declaration de procedimientos */
void initialize(void);
double randval(double, double);
void evaluate(void);
void keep_the_best(void);
void elitist(void);
void select(void);
void crossover(void);
void Xover(int,int);
void swap(double *, double *);
```

```
void mutate(void);
void report(void);
void printpopulation(void);
double gaussian(void);
/*************************************************************************/
/* Función de inicializacion, inicializa los valores de genes, dentro del limite */
/* definido para cada variable, también inicializa a cero todos los valores de */
/* fitness.                                                              */
/* Lee los limites de las variables desde el archive Agantnetdata.txt    */
/* cuyo formato es                                                       */
/* var1_limite_inferior var1_limite_superior                             */
/* var2_limite_inferior var2_limite_superior                             */
/* varn….                                                .               */
/*************************************************************************/
void initialize(void)
{
        FILE *infile;
        int i, j;
        double lbound, ubound;
        if ((infile = fopen("Agantnetdata.txt","r"))==NULL)
    {
    fprintf(galog,"\nCannot open input file!\n");
    exit(1);
    }
/* inicializa variables dentro de sus limites */
        for (i = 0; i < NVARS; i++)
   {
    fscanf(infile, "%lf",&lbound);
    fscanf(infile, "%lf",&ubound);

    for (j = 0; j < POPSIZE; j++)
        {
        population[j].fitness = 0;
                population[j].delay = 0;
        population[j].rfitness = 0;
        population[j].cfitness = 0;
        population[j].lower[i] = lbound;
        population[j].upper[i]= ubound;
        population[j].gene[i] = randval(population[j].lower[i], population[j].upper[i]);
        }
   }
        fclose(infile);
}

/*************************************************************/
/* Generador de valores aleatorios entre dos limites         */
/*************************************************************/
double randval(double low, double high)
{
```

```
        double val;
        val = ((double)(rand()%1000)/1000.0)*(high - low) + low;
        return(val);
}

/*************************************************************/
/* Funcion de evaluacion de fitness  */
/*************************************************************/
void evaluate(void){
  int mem;
  int i;
  int iretval;
  double retval;
  double x[NVARS+1];
  char buffer [250], formatbuffer[100], valuebuffer[100];
  int k;
  formatbuffer[0] = '\0';
  valuebuffer[0] = '\0';
  for (mem = 0; mem < POPSIZE; mem++) {
          k=0;
    for (i = 0; i < NVARS; i++){
        x[i+1] = population[mem].gene[i];
//                      printf("  %d     %d     %12.4f   %12.4f   %15.4f   \n   ",mem,i,population[mem].lower[i],
population[mem].upper[i],x[i+1]);
                    k += sprintf(valuebuffer  +k,"%12.4f ",population[mem].gene[i]);
          }

    sprintf (buffer, "java -jar C:\\Programs\\AntNetSimulation\\dist\\AntNetSimulation.jar %s",valuebuffer);
          printf ("[%s] ",buffer);

          iretval=system(buffer);
          retval= iretval / 10000.0;
          population[mem].fitness = retval;
          population[mem].delay=iretval-retval*10000;
          printf("[Fitness retornado: %12.4f]\n",population[mem].fitness);
  }
}
/*************************************************************/
/* Keep_the_best: Almacena el mejr miembro de la poblacion   */
/* en la ultima posicion del arreglo que representa la poblacion  */
/*************************************************************/

void keep_the_best()
{
int mem;
int i;
cur_best = 0; /* indice del mejor individuo */

for (mem = 0; mem < POPSIZE; mem++)
```

```
    {
    if (population[mem].fitness > population[POPSIZE].fitness)
        {
        cur_best = mem;
        population[POPSIZE].fitness = population[mem].fitness;
        }
    }
/* copia los genes del mejor */
for (i = 0; i < NVARS; i++)
    population[POPSIZE].gene[i] = population[cur_best].gene[i];
}
/*************************************************************/
/* elitist: Almacena el major miembro entre la poblacion actual   */
/* y la anterior                                                  */
/*************************************************************/
void elitist()
{
int i;
double best, worst;          /*  major y peor valor de fitness */
int best_mem, worst_mem; /* indices de ellos */
best = population[0].fitness;
worst = population[0].fitness;
for (i = 0; i < POPSIZE - 1; ++i)
    {
    if(population[i].fitness > population[i+1].fitness)
        {
        if (population[i].fitness >= best)
            {
            best = population[i].fitness;
            best_mem = i;
            }
        if (population[i+1].fitness <= worst)
            {
            worst = population[i+1].fitness;
            worst_mem = i + 1;
            }
        }
    else
        {
        if (population[i].fitness <= worst)
            {
            worst = population[i].fitness;
            worst_mem = i;
            }
        if (population[i+1].fitness >= best)
            {
            best = population[i+1].fitness;
            best_mem = i + 1;
            }
```

```
        }
    }

if (best >= population[POPSIZE].fitness)
   {
   for (i = 0; i < NVARS; i++)
     population[POPSIZE].gene[i] = population[best_mem].gene[i];
   population[POPSIZE].fitness = population[best_mem].fitness;
   }
else
   {
   for (i = 0; i < NVARS; i++)
     population[worst_mem].gene[i] = population[POPSIZE].gene[i];
   population[worst_mem].fitness = population[POPSIZE].fitness;
   }
}
/*************************************************************/
/* funcion de seleccion standard aplica el metodo de la ruleta */
/*************************************************************/
void select(void)
{
int mem, i, j;
double sum = 0;
double p;
/* fitness stotal de la population */
for (mem = 0; mem < POPSIZE; mem++)
    {
    sum += population[mem].fitness;
    }
/* fitness relativo*/
for (mem = 0; mem < POPSIZE; mem++)
    {
    population[mem].rfitness =  population[mem].fitness/sum;
    }
population[0].cfitness = population[0].rfitness;
/* fitness acumulado */
for (mem = 1; mem < POPSIZE; mem++)
    {
    population[mem].cfitness =  population[mem-1].cfitness +
                    population[mem].rfitness;
    }
for (i = 0; i < POPSIZE; i++)
    {
    p = rand()%1000/1000.0;
    if (p < population[0].cfitness)
        newpopulation[i] = population[0];
    else
        {
        for (j = 0; j < POPSIZE;j++)
```

```
            if (p >= population[j].cfitness &&
                    p<population[j+1].cfitness)
                newpopulation[i] = population[j+1];
        }
    }
for (i = 0; i < POPSIZE; i++)
    population[i] = newpopulation[i];
}
/*************************************************************/
/* crossover selection: selecciona dos padres y los cruza aplicando    */
/* un punto simple de cruzamiento                                       */
/*************************************************************/
void crossover(void)
{
int mem, one;
int first  =  0;
double x;
for (mem = 0; mem < POPSIZE; ++mem)
    {
    x = rand()%1000/1000.0;
    if (x < PXOVER)
        {
        ++first;
        if (first % 2 == 0)
            Xover(one, mem);
        else
            one = mem;
        }
    }
}
/*************************************************************/
/* Xover: realiza el cruzamiento de dos padres */
/*************************************************************/
void Xover(int one, int two)
{
int i;
int point; /*punto de cruzamiento */
if(NVARS > 1)
  {
  if(NVARS == 2)
      point = 1;
  else
      point = (rand() % (NVARS - 1)) + 1;

  for (i = 0; i < point; i++)
      swap(&population[one].gene[i], &population[two].gene[i]);
  }
}
/*************************************************************/
```

```
/* swap: cambia dos variables
/*****************************************************************/
void swap(double *x, double *y)
{
double temp;

temp = *x;
*x = *y;
*y = temp;

}
/*****************************************************************/
/* mutate aplica operador de mutacion                            */
/*****************************************************************/
void mutate(void)
{
int i, j;
double lbound, hbound;
double x;

for (i = 0; i < POPSIZE; i++)
     for (j = 0; j < NVARS; j++)
          {
//          x = rand()%1000/1000.0; //Random uniforme mutacion.
                                    x = gaussian();//mutacion  Gaussiaana
          if (x < PMUTATION)
               {
               lbound = population[i].lower[j];
               hbound = population[i].upper[j];
               population[i].gene[j] = randval(lbound, hbound);
               }
          }
}
double gaussian(void) {
int i;
double r[12];
double q,m,sd,s;
s=0.0;
for (i = 0; i < 12; i++) {
            r[i]=rand()%1000/1000.0;
            s += r[i];
}
m=s/12;
s=0.0;
for (i = 0; i < 12; i++) {
            s += (r[i]-m)*(r[i]-m);
}
sd=sqrt(s/11.0);
s=0.0;
```

```
for (i = 0; i < 12; i++) {
            s += (r[i]-s);
}
q=m+sd*s;
return q;
}
void report(void)
{
  int i;
  double best_val;
  double avg;
  double stddev;
  double sum_square;
  double square_sum;
  double sum;
  double bdelay;
  sum = 0.0;
  sum_square = 0.0;
  for (i = 0; i < POPSIZE; i++)
            {
            sum += population[i].fitness;
            sum_square += population[i].fitness * population[i].fitness;
            }
  avg = sum/(double)POPSIZE;
  square_sum = avg * avg * POPSIZE;
  stddev = sqrt((sum_square - square_sum)/(POPSIZE - 1));
  best_val = population[POPSIZE].fitness;
  bdelay = population[POPSIZE].delay;
  fprintf(galog, "\n%5d, %10.3f, %10.3f,%10.3f, %10.3f, ", generation,best_val, bdelay, avg, stddev);
  for (i = 0; i < NVARS; i++){
  fprintf (galog,"%12.4f, ",population[POPSIZE].gene[i]);
  }
}
void printpopulation(void){
  int i,j;
  fprintf(logpob, "\nGeneración      Pob  Fitness   packetSize    simTime       alpha        processTime");
  for (i = 0; i < POPSIZE; i++) {
            fprintf(logpob,"\n%d, %d %10.4f ", generation,i, population[i].fitness);
            for (j = 0; j < NVARS; j++){
                        fprintf (logpob,"%10.4f ",population[i].gene[j]);
            }
  }
  fprintf(logpob,"\n");
}
int main(void) {
            int i;
            time_t t;
    struct tm tstruct;
```

```
        if ((galog = fopen("Agantnet.log","w"))==NULL){
     exit(1);
    }
        if ((logpob = fopen("logpob.log","w"))==NULL){
     exit(1);
    }
        t = time(0);
   tstruct = *localtime(&t);
   printf("The time: %s", ctime(&t));
        fprintf(galog,"\nThe time: %s", ctime(&t));
        generation = 0;
        fprintf(galog, "\nNumero\tMejor\Fitness\tPacket\tDesviación\tParametros Mejor Individuo    \n");
        fprintf(galog,
"Generación\tFitness\Delay\tPromedio\tEstandard\tpacketSize\tsimTime\talpha\tprocessTime\n");
        initialize();
        evaluate();
        keep_the_best();
        while(generation<MAXGENS){
     generation++;
     select();
     crossover();
     mutate();
     evaluate();
     elitist();
          report();
          printf("Generacion %d   Fitness: %6.3f\n",generation,population[POPSIZE].fitness);
          printpopulation();
        }
        fprintf(galog,"\n\n Simulation completed\n");
        fprintf(galog,"\n Best member: \n");
        for (i = 0; i < NVARS; i++){
        fprintf (galog,"\n var(%d) = %3.3f",i,population[POPSIZE].gene[i]);
  }
        fprintf(galog,"\n\n Best fitness = %3.3f",population[POPSIZE].fitness);
        t = time(0);
   tstruct = *localtime(&t);
   fprintf(galog,"\nThe time: %s", ctime(&t));
        printf("\nThe time: %s", ctime(&t));
        fclose(galog);
        fclose(logpob);
        printf("Success\n");
}
```

Anexo 2: Listado programa AntNnet

```
/**
 * Esta clase recoge gráfico de la red y los procesos (agentes artificiales y los paquetes de datos)
 * implementacion del algoritmo de AntNet.
 */
public class AntNet extends Process{

    /** Red Grafica. */
    private NetworkGraph graph;
    private List<AgentGenerator> agentGenerators;
    private DataPacketGenerator packetGenerator;
    private double simTime;
    private int packetSize;
    private double alpha;

    /**
     * Este constructor crea un objeto AntNet.
     * @ Param numberOfVertices número de nodos de red en el gráfico.
     * @ Param packetSize tamaño del paquete de todos los paquetes en la simulación.
     * @ Param simTime tiempo de las corridas de simulación
     * @ Param alfa valor que pesa la importancia relativa de la información heurística en relación a los valores de la feromona.
     * @ Param processTime Tiempo de proceso en nodo al tratamiento de los paquetes.
     */
    public AntNet(int numberOfVertices, int packetSize, double simTime, double alpha, double processTime){
        graph = new NetworkGraph(numberOfVertices, processTime);
        this.simTime = simTime;
        this.alpha = alpha;
        this.packetSize = packetSize;
        agentGenerators = new ArrayList<AgentGenerator>();
    public void createAgentGenerator(NetworkVertex node, double deltaT, double timeToLive){
        AgentGenerator agentGenerator = new AgentGenerator(node, packetSize, simTime, deltaT, timeToLive, alpha);
        agentGenerators.add(agentGenerator);
    }
    public void createDataPacketGenerator(double deltaPackets, int maxHops){
        packetGenerator = new DataPacketGenerator(graph.getNetworkVertices(), packetSize, simTime, deltaPackets, maxHops);
    }
    public void actions(){
        activate(packetGenerator);
        for(AgentGenerator generator : agentGenerators){
            activate(generator);
        }
        hold(simTime + 10000);
        report();
    }

    /**
     */
    public NetworkGraph getNetworkGraph(){
        return graph;
    }
    public static void main(String[] args){
    try {
        int maxQueueLenght = 1000;
```

```
int edgeLenght = 20;
double transmissionSpeed = 0.3;
int propagationSpeed = 200000000;
double deltaPackets = 1;
int maxHops = 15;
double deltaAgents = 3;
double ttl = 15;
int packetSize = (int)  Double.valueOf(args[0]).doubleValue();
double simTime =        Double.valueOf(args[1]).doubleValue();
double alpha =          Double.valueOf(args[2]).doubleValue();
double processTime =    Double.valueOf(args[3]).doubleValue();
FileWriter  fw   = new FileWriter("c:\\tmp\\ant.txt");
PrintWriter log  = new PrintWriter(fw);
int numberOfNodes = 55;
log.println("--> Simulando NTTNET con alpha: "+alpha);
AntNet antnet = new AntNet(numberOfNodes, packetSize, simTime, alpha, processTime);
NetworkGraph graph = antnet.getNetworkGraph();
graph.makeEdge(0, 2, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(1, 2, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 5, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(2, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(3, 5, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(4, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(5, 6, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(5,11, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(6, 7, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(6, 8, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(7,10, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(8, 9, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(9,17, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(10,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(11,14, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(12,15, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(13,15, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(14,19, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(15,16, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(16,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(16,23, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(17,18, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(18,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(19,20, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(20,21, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(20,23, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(21,22, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(21,24, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(22,26, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(23,25, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(24,29, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(25,28, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(25,31, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(26,27, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(27,35, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(28,30, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(29,30, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
graph.makeEdge(30,33, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
```

```
        graph.makeEdge(31,32, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(32,36, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(33,34, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(33,36, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(34,35, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(34,38, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(35,42, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(36,37, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(37,40, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(38,39, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(39,40, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(39,41, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(40,41, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(41,42, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(41,45, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(42,51, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(42,43, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(43,44, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(43,46, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(44,45, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(45,48, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(46,47, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(47,48, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(47,49, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(48,50, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(49,50, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(49,51, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(50,51, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(51,52, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(52,54, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.makeEdge(53,54, maxQueueLenght, maxQueueLenght, edgeLenght, transmissionSpeed, propagationSpeed);
        graph.initializeGraph();
        antnet.createDataPacketGenerator(deltaPackets, maxHops);
        for (int i=0; i<numberOfNodes;i++) {
          antnet.createAgentGenerator(graph.getNetworkVertices().get(i), deltaAgents, ttl);
        }
        activate(antnet);
        double ttotal=0, totald=0;
        for(NetworkVertex node : graph.getNetworkVertices()){
           ttotal += node.getTroughput();
           totald += node.getProcessTime();
        }
        ttotal *= 100000+totald;
        System.exit( (int) ttotal);

        log.close();
        fw.close();
       } catch (IOException e){
           e.printStackTrace();
        }
}
}
```

Anexo 3: Ejemplo Archivo Agantnetdata.txt

Este archivo almacena los límites de cada parámetro de acuerdo a lo estipulado en la Tabla 4.0-1: Parámetros de AntNet considerados, es decir

Parámetro
Linea 1: packetSize
Linea 2: simTime
Linea 3: Alpha
Linea 4: processTime

Contenido del archivo Agantnetdata.txt:

```
9000.0 12000.0
800.0 1100.0
0.3 0.7
0.1 0.4;
```