VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"BÚSQUEDA Y TRANSMISIÓN GLOBAL EN SISTEMAS DE REDES ENTRE IGUALES E INTERNET EMPLEANDO ÁRBOLES LOCALES DE EXPANSIÓN MÍNIMA"

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRA EN INGENIERÍA
(COMPUTACIÓN)

P R E S E N T A:

TANIA PÉREZ MARTÍNEZ

DIRECTOR DE TESIS: DR. JULIO SOLANO GONZÁLEZ
CODIRECTOR: DR. IVAN STOJMENOVIC

México, D.F. 2005.

Búsqueda y transmisión global en sistemas de redes entre iguales e Internet empleando Árboles Locales de Expansión Mínima

Este trabajo fue desarrollado bajo el financiamiento de CONACyT No. 37017-A, dentro del proyecto de investigación: *Scalable wireless ad-hoc, sensor and local area networks organization and data communication*, y ha sido base del artículo:

O. Escalante, T. Pérez, J. Solano and I. Stojmenovic, ***RNG-based searching and broadcasting algorithms over Internet graphs and peer-to-peer computing systems***, 3rd ACS/IEEE Int. Conf. On Computer Systems and Applications, Cairo, Egypt, Jan. 3-6, 2005

Tania Pérez Martínez

Director: Dr. Julio Solano González

Codirector: Dr. Ivan Stojmenovic

México D.F.
Mayo 2005

Resumen

En el proceso de transmisión global (broadcasting), un mensaje es enviado desde un nodo-fuente hacia todos los nodos en una red. El mecanismo clásico de broadcasting es llamado blind-flooding, en este, cada nodo retransmite el mensaje recibido a todos sus vecinos sin importar si alguno de ellos lo ha recibido previamente de otra fuente. A pesar de las ventajas intrínsecas que representa este método, un incremento ya sea en el número de peticiones o en el tamaño del área de ruteo en una red, produce un overhead de comunicación que limita su escalabilidad, especialmente en redes con topologías dinámicas.

En este trabajo se propone un nuevo esquema de broadcasting y búsqueda en Internet basado en un algoritmo que construye un Árbol Local de Expansión Mínima (ALEM) con la finalidad de aproximarse, con un conocimiento local, a la ruta óptima de broadcast generada por el Árbol de Expansión Mínima (AEM). Los mensajes son enviados solamente a los enlaces que pertenecen al ALEM, un enlace está incluido en él si y solo si ambos nodos terminales lo contienen en sus respectivos árboles locales. Se hará una extensión de la definición existente del ALEM, usando la distancia geográfica, para poder aplicar métricas más significantes en Internet como lo es el retraso en la transmisión. El ALEM es una red sobrepuesta conectada que se define en base a la información local de cada nodo, un registro que incluye la lista de sus nodos-vecinos y el peso de sus respectivas conexiones.

Para evaluar su desempeño, este nuevo esquema será comparado con los métodos existentes de Flooding, Rumor Mongering (o Gossip) y otro basado en Grafos de Vecindad Relativa (GVR) propuesto anteriormente por nuestro grupo de investigación y una extensión de éste nombrado: Grafos de Vecindad Relativa con remoción Cuadrilateral (GVRC). Los cuatro planteados y aplicados a redes entre iguales (peer-to-peer) descentralizadas y no estructuradas como lo es Gnutella. El esquema propuesto, garantiza la entrega a cada nodo con una reducción considerable en el número de mensajes con respecto a Flooding,GVR,GVRC y Rumor Mongering, este último además de no garantizar la transmisión a cada nodo, emplea parámetros cuyos valores óptimos dependen de la densidad de la red subyacente.

Contenido

Capítulo 1

Introducción

El objetivo es reducir el número de transmisiones redundantes construyendo una red eficiente a partir de una red subyacente (Internet), sobre la cual se enviarán los mensajes de broadcast.

Para que una red dinámica descentralizada se adecue a los cambios generados por la interconexión de dispositivos espontáneos de comunicación, cualquier protocolo que funcione dentro de este sistema, necesitará conocer las modificaciones causadas en su topología, ya sea del conjunto de vecinos de cada nodo, de sus posiciones o del status de los enlaces y nodos. Algunos ejemplos que emplean un esquema de este tipo puede observarse en los protocolos de estado de enlace: OSPF y MOSPF y en los protocolos de *multicast* DVMRP, QoSMIC y YAM que propagan las actualizaciones de la topología en Internet. Esta tarea se realiza generalmente por medio de alguna técnica de transmisión global (*broadcasting*), transmitiendo un mensaje a todos los nodos. El *broadcasting* también tiene otras aplicaciones, por ejemplo, en la búsqueda de alguna información en particular entre todos los nodos de un sistema de redes entre iguales (*peer-to-peer* o P2P).

Blind-Flooding es el método clásico de propagación de mensajes de *broadcast* a través de una red y tiene su origen con un nodo que disemina un mensaje entre sus vecinos. Cuando un nodo recibe el mensaje por primera vez, envía una copia a todos sus vecinos, excepto al nodo del cual lo recibió, sin importar si alguno de ellos le ha llegado previamente. El proceso termina después de que cada nodo ha recibido al menos una copia del mensaje. A pesar de su amplio uso, *Blind-Flooding* tiene desventajas importantes, lo que hace inapropiado su uso en muchas redes dinámicas, entre otras razones, por la cantidad de mensajes redundantes que se generan para poder cubrir completamente una red.

Las alternativas propuestas para mitigar los problemas presentados por el método de *blind-flooding* incluyen diferentes tipos de algoritmos, entre ellos los que seleccionan a sus nodos de manera probabilística como el protocolo Rumor Mongering o Gossip [PS] o aquellos novedosos, que generan un grafo de comunicación óptimo para propagar los mensajes de *broadcast*, como en [EPSS], donde para tal efecto se aplica el algoritmo

Grafos de Vecindad Relativa (GVR) o en [LXLNZ] al generar un Grafo de Vecindad Relativa con remoción Cuadrilateral (GVRC).

En este trabajo se propone un esquema basado en el algoritmo que construye un Árbol Local de Expansión Mínima (ALEM), una estructura planteada para aproximarse a la ruta óptima de *broadcast* generada por el Árbol de Expansión Mínima (AEM) y se muestra como el efectuar un *broadcast* sobre el ALEM usando alguna métrica apropiada de distancia, puede mejorar el desempeño de los cuatro esquemas: *blind flooding*, Rumor Mongering, GVR y GVRC. Los ejemplos de las medidas de distancia que pueden ser aplicadas incluyen el retraso, la congestión, los números aleatorios o la clásica distancia geográfica.

La idea de construir un ALEM, es la de evitar congestionar la red completamente con los mensajes de *broadcast* al no emplear todas las rutas disponibles en ella sino al seleccionar un subconjunto específico de la topología original para transmitirlos. La infraestructura de comunicación generada tiene algunas propiedades importantes:

- Control en la topología. El sub-grafo que se genera tiene un reducido número de enlaces promedio por nodo al eliminar los enlaces de mayor peso, mantiene la conectividad y es bidireccional. No emplea algún parámetro en su funcionamiento.

- Ruteo. Garantiza la entrega de mensajes al 100% de nodos usando las propiedades geométricas del grafo de comunicación.

El trabajo principal se ha hecho sobre grafos de Internet generados sintéticamente que la modelan de acuerdo a los esquemas que han reportado los estudios actuales sobre su topología y se emplean los sistemas *peer-to-peer* para modelar una aplicación. Las redes *peer-to-peer* se desarrollan de manera *ad-hoc*[1] y los nuevos nodos se pueden conectar a cualquier nodo que deseen dentro de la red. Por tanto, la topología global exacta es desconocida y los nodos individuales tienen solamente una vista muy local de la red. La principal ventaja del ALEM es que genera una red sobrepuesta conectada con base únicamente en la información local disponible de cada nodo. Las redes sobrepuestas usan solamente los enlaces existentes de la red subyacente, lo que evita la generación de tráfico para establecer rutas de enlaces virtuales en, por ejemplo, los esquemas de Tablas Hash Distribuidas ([BKKMS]) para realizar búsquedas de información en sistemas *peer-to-peer* estructurados.

[1] Una red *ad-hoc* es un conjunto de nodos formando una red temporal que carece de una administración centralizada. Los nodos actúan como ruteadores, retransmitiendo los paquetes del protocolo de comunicación. Estas redes están sometidas a cambios frecuentes en su topología, los nodos nuevos pueden unirse inesperadamente a la red o los nodos existentes pueden salirse de pronto. Lo más importante es que toda la comunicación se lleva a cabo vía *broadcast*.

Objetivos del trabajo de tesis

Diseñar una red optima de comunicación que conecte todos los nodos y satisfaga sus requerimientos de *broadcast* con un costo total mínimo al aproximarse a la ruta de *broadcast* óptima generada por el Árbol de Expansión Mínima (AEM)

Evaluar el desempeño del algoritmo propuesto y comparar los resultados con la solución planteada anteriormente (GVR) por nuestro grupo de investigación.

Verificar el costo de *broadcast* sobre la estructura ALEM aplicándolo a los sistemas *peer-to-peer* y valorarlo con otras propuestas conocidas.

Contribuciones de la investigación

Este trabajo de investigación fue desarrollado para proponer un esquema de *broadcast* en Internet aplicando el algoritmo ALEM, con la idea de optimizar la red de comunicación sobre la que se propagarían los mensajes de *broadcast*. Este método puede ser particularmente útil en redes sobrepuestas por la capa de aplicación, donde la construcción de una red de este tipo toma ventaja de la infraestructura actual de Internet y mejora justamente a nivel aplicación algunas limitantes referentes a las políticas de ruteo del protocolo de Internet (IP) para las cuales éste no ha sido diseñado: el *broadcast*. Una tarea que pareciera debería ser implementada por los dispositivos de red (ruteadores y switches) se convierte en una funcionalidad implementada en cada *host*.

Dentro de este creciente tipo de redes, se encuentran las redes *peer-to-peer*, en donde cada *peer* mantiene un número de conexiones TCP punto-a-punto permanentemente abiertas sobre la cual son enrutados los mensajes del protocolo. En particular, el diseño de Gnutella comprende un protocolo a nivel aplicación que realiza el *broadcast* (*flooding*) en toda su red de dos mensajes básicos: *Ping*, que es esencialmente la petición a un *peer* para anunciarse a si mismo y *Query*, un mensaje que sirve para hacer peticiones de archivos. La topología de esta red sobrepuesta y los mecanismos de ruteo empleados, tienen una influencia primordial sobre las propiedades: confiabilidad[2], escalabilidad[3] y desempeño, de la aplicación. El éxito de la revolución de los sistemas P2P, dependerá en gran medida de la habilidad de sus aplicaciones para ofrecer una comunicación eficiente[4] entre un enorme y creciente número de *peers* autónomos y dispersos en toda Internet.

[2] Donde la confiabilidad es la probabilidad que todos los nodos entregen y reciban el mensaje de broadcast.

[3] El valor que representa una red para un usuario individual escalará con el número total de participantes. Idealmente, cuando se incrementa el número de nodos, el espacio de almacenamiento agregado y la disponibilidad de los archivos debería crecer linealmente, el tiempo de respuesta debería permanecer constante, mientras el tráfico debido a las búsquedas debe permanecer alto o aumentar

[4] El término eficiente deberá ser interpretado en términos de alcance y redundancia de un mensaje, latencia de una petición y tamaño de la tabla de ruteo de cada nodo.

La relevancia de este trabajo no se restringe únicamente al área de las aplicaciones *peer-to-peer* que comparten archivos. El concepto de redes sobrepuestas por la capa de aplicación está siendo propuesto cada vez más por diversas aplicaciones (ver por ejemplo [ABKM] y [CRZ]), así que el plantear, para estos sistemas descentralizados y no estructurados, un método eficiente de *broadcast* puede ser un importante tema de desarrollo, especialmente para el tipo de aplicaciones distribuidas en Internet.

Estructura de la tesis

El trabajo de tesis esta organizado de la siguiente manera:

Capítulo 2. Explica como se aplica el concepto de ALEM para realizar la diseminación de mensajes de *broadcast* en topologías Internet.

Capítulo 3. Expone los esquemas que modelan la topología de Internet de acuerdo a las propiedades que han reportado los estudios actuales sobre su estructura. Se presentan los generadores de topologías que se usaran para el proceso de simulación de *broadcasting*.

Capítulo 4. Describe de manera general los sistemas *peer-to-peer*. Enfocándose en Gnutella, una aplicación que comparte archivos de manera distribuida en Internet y el protocolo que emplea para realizar la búsqueda de archivos en su red.

Capítulo 5. Describe el simulador de *broadcasting*, las clases implementadas y los algoritmos.

Capítulo 6. Presenta los resultados del proceso de simulación de *broadcast* sobre topologías Internet y sistemas *peer-to-peer* y su análisis.

Capítulo 7. Presenta las conclusiones generales de la tesis y propone el trabajo posterior que queda por realizar.

Apéndice A. Contiene la descripción de las clases empleadas en la simulación.

Apéndice B. Contiene el código fuente completo que implementa las clases en lenguaje de programación Java.

Capítulo 2

Árbol de expansión mínima (AEM)

Sin una vista aérea,¿cómo encontrar la mejor ruta a un destino desconocido?

En esta sección se propone el uso del algoritmo de Prim [P], que genera un árbol de expansión mínima (AEM), como base de un algoritmo de *broadcasting* distribuido, llamado Árbol Local de Expansión Mínima (ALEM).

Un AEM, es un grafo conectado que emplea la longitud total mínima de los enlaces y da como resultado un grafo con un enlace menos al número de vértices. El AEM se emplea tradicionalmente en redes para determinar árboles de *broadcast* usando la información global de su topología [LBCJ], aunque si bien, el AEM genera una ruta de *broadcast* óptima, al cambiar una red con forma de malla en una de árbol en la que no se puede producir ningun circuito cerrado, tiene un par de inconvenientes importantes: primero, no es un grafo tolerante a fallas, en el sentido de que el número alternativo de rutas para que un mensaje se transmita a través de cada nodo es nulo, es decir, el grafo no permanece conectado cuando algún nodo falla y esto limita la probabilidad de su entrega [JO] y segundo, tampoco puede ser construido de manera local, ya que cada nodo es incapaz de determinar que enlace pertenece a esta estructura usando únicamente la información de los nodos restringida por alguna vecindad constante de *hops* y en todo caso, es más costoso construir un AEM de una manera distribuida por la cantidad de mensajes que se necesitan intercambiar para generarla [LWS]. Así que se han propuesto algunas estructuras (ALEM y GVR entre otras) para aproximarse a él.

El algoritmo de Prim es un árbol de costo mínimo, donde el costo de un enlace es la distancia euclidiana entre sus nodos y se desarrolla según el siguiente proceso:

Asumir que G=(N, E) es una gráfica conectada con pesos en cada una de sus aristas y que V es un subconjunto de N. Si entre todas las aristas que unen V y N-V, la arista (u,v) tiene el menor peso, entonces hay un árbol de expansión mínima de G que incluye a la arista (u, v).

El algoritmo de Prim comienza con un árbol trivial que consiste solamente en el vértice inicial *u*. Esto divide a N en dos conjuntos separados, $V=\{u\}$ y N-V. El algoritmo añade iterativamente la arista de menor peso que va de V a N-V. Cada una de estas aristas contribuirá a definir un nuevo vértice que será agregado a V hasta que todos los vértices pertenezcan a el.

Las siguientes líneas resumen, en pseudocódigo, el algoritmo de Prim:

asignar el vértice u a V
    mientras V tenga menos de n vértices
        encontrar la arista de menor peso que conecte V con N-V
        añadir esta a V

Por ejemplo, si se construye el AEM del grafo P (figura 2.1) empezando por el vértice 0, entonces se obtiene la secuencia de los árboles parciales de expansión mínima mostrados en la figura 2.3. En este ejemplo, el costo del AEM del grafo P (figura 2.2) es 12.

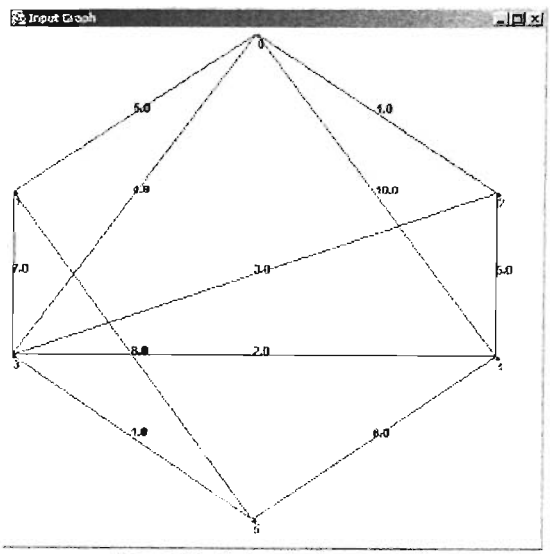


Figura 2.1. El ejemplo de un grafo P con el peso de sus aristas.

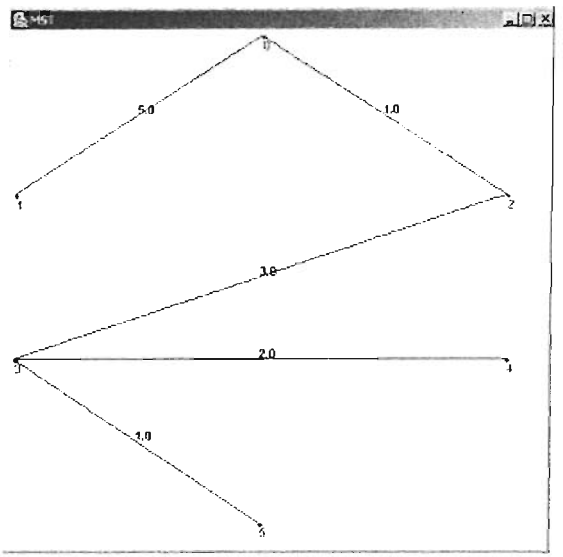
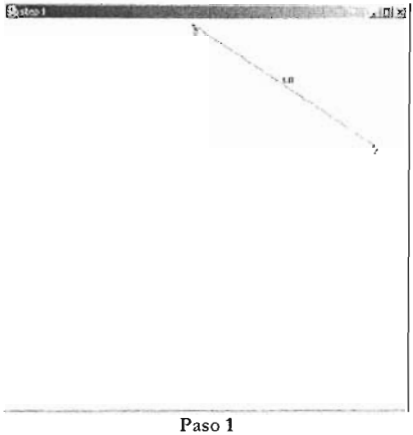


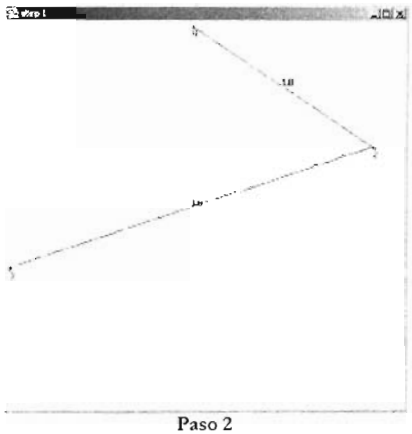
Figura 2.2. AEM del grafo P



Paso 1

V={0} N-V={1,2,3,4,5}

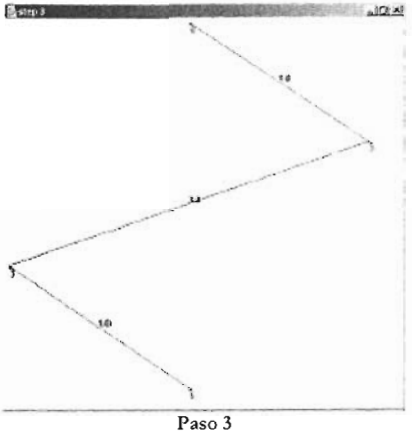
N-V	V más cercano	costo menor
1	0	5
2	0	1
3	0	4
4	0	10
5	0	∞



Paso 2

V={0,2} N-V={1,3,4,5}

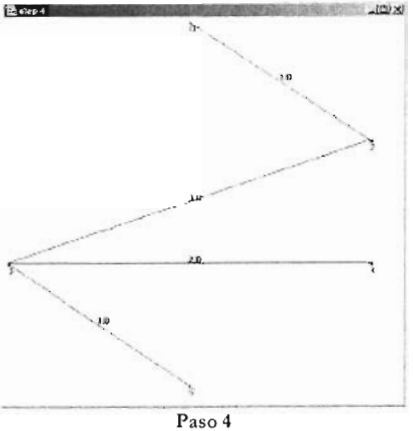
N-V	V más cercano	costo menor
1	0	5
3	2	3
4	2	5
5	0	∞



Paso 3

V={0,2,3} N-V={1,4,5}

N-V	V más cercano	costo menor
1	0	5
4	3	2
5	3	1



Paso 4

V={0,2,3,5} N-V={1,4}

N-V	V más cercano	costo menor
1	0	5
4	3	2

Figura 2.3. Cuatro muestras del algoritmo de Prim del grafo P.

Árbol local de expansión mínima (ALEM)

En este algoritmo, cada nodo construye independientemente su AEM de manera local del subgráfo delimitado por sus vecinos, en donde los únicos enlaces que se mantendrán serán aquellos que pertenezcan al ALEM de los respectivos nodos terminales.

Este algoritmo fue planteado para redes inalámbricas por Li *et al* [LHS] en 2003, como una propuesta para definir topologías en donde cada nodo consuma la mínima potencia de transmisión posible preservando la conectividad de la red. Está diseñado para redes que carecen de una autoridad central (redes *peer-to-peer*), por lo tanto, cada nodo esta obligado a tomar decisiones con base en la información que haya recopilado de la red. Incluso para ser menos susceptible al impacto de la movilidad, cada nodo depende solamente de la información recolectada localmente a través de sus nodos vecinos, lo que genera también un menor número de mensajes en la red durante este proceso. El concepto de algoritmos locales fue propuesto en [GHK] como una estructura distribuida donde el comportamiento local de cada nodo logra un objetivo global deseado.

Li *et al.* [LHS] demostraron que la topología de red derivada del algoritmo ALEM tiene las siguientes propiedades:

1. La topología resultante preserva la conectividad. Para el propósito de diseminación de información (*broadcast*), se debe garantizar una topología de comunicación conectada. La trasformación ALEM preserva la conectividad si el AEM derivado es único –podría no serlo si existieran múltiples enlaces con la misma longitud. Para asegurar la singularidad del AEM, Li *et al.* [LHS] propusieron usar como clave primaria de comparación, la longitud del enlace y como secundaria y terciaria (si hiciera falta), los respectivos identificadores (ordenados de alguna manera) de los nodos terminales del enlace.

2. Si se emplea la distancia euclidiana como métrica de peso, el grado de cualquier nodo en la topología resultante esta limitado a 6, es decir, se preserva el grado máximo de cualquier AEM de un conjunto finito de puntos en el plano [MS]. El grado se define como el número de enlaces que inciden en el nodo.

3. La topología puede ser transformada en una con enlaces bidireccionales después de remover todos los enlaces unidireccionales sin que se perjudique la conectividad de la red.

Con el fin de corroborar el algoritmo y evaluar sus resultados, se implementó y simuló para redes inalámbricas. En esta sección se muestran los resultados. Posteriormente se hace una extensión de su definición para aplicarlo a topologías Internet, empleando para ésta otras métricas más convenientes, como son el retraso en la transmisión, el ancho de banda de cada enlace o la designación de un número aleatorio bajo alguna distribución. Se pudo constatar que el ALEM es conectado para cualquier métrica de distancia, lo cual es demostrado en [EPSS].

Algo que vale la pena resaltar es que un ALEM definen un grafo plano en una red inalámbrica, siempre y cuando se use como peso de cada enlace la distancia Euclidiana entre los nodos; al aplicar este algoritmo en Internet no, ya que el patrón de conexión se establece según el principio de conectividad preferente y no bajo el esquema de cercanía (distancia Euclidiana) entre nodos.

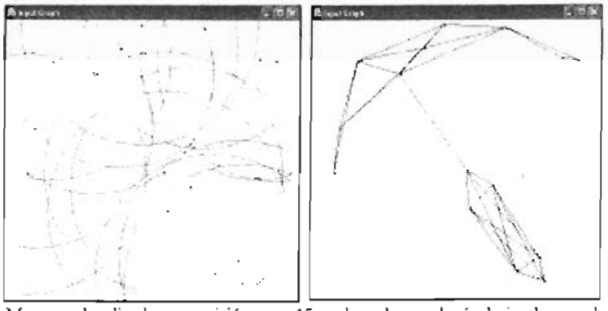
Construcción del ALEM en redes inalámbricas

Las redes inalámbricas se modelan usando Grafos Unitarios Aleatorios (figura 2.4), los cuales son construidos de la siguiente manera: un nodo A tiene un rango de transmisión $t(A)$. Dos nodos A y B en la red son vecinos (por tanto unidos por un enlace), si la distancia Euclidiana entre ellos es menor que el mínimo rango de transmisión entre ellos, $d(A,B) < min\ \{t(A), t(B)\}$. Si todos los radios de transmisión R son iguales, entonces, el grafo correspondiente se conoce como Grafo Unitario [SSS]. El rango de transmisión critico que preserva la conectividad de la red esta definido por la longitud del enlace más largo del AEM usando la distancia Euclidiana como peso de los enlaces[SB]. El valor de R es crucial, ya que un valor mayor que el necesario causa interferencia en la comunicación e incremento en el consumo de energía, mientras que un valor menor, impide el proceso de comunicación al no garantizar la conectividad de todos los nodos.

El término aleatorio en los grafos unitarios se deriva de la distribución aleatoria que se asigna a los nodos sobre un área de 1 x 1 unidades. En este nivel de abstracción, un nodo en la red esta representado por un vértice en el grafo, mientras que un enlace está representado por una arista. Sin perdida de formalidad, los términos nodo y vértice se usan indistintamente para referirse a la misma entidad, haciendo la misma consideración para enlace y arista.

La topología de red construida bajo un rango de transmisión común R se designará como un grafo no dirigido $G = (N, E)$ en el plano, donde N es el conjunto de nodos en la red y $E = \{(u, v): d(u, v) = R, u, v ?\ N\}$ es el conjunto de enlaces de G. En el proceso de construcción de la topología, cada nodo u debe establecer su vecindad

visible $VV_u(G)$, que es un subgrafo $G_u = (N_u, E_u)$ de G, el cual se define de acuerdo al conjunto de nodos que quedan dentro del radio de transmisión del nodo u.



Muestra el radio de transmisión para 15 nodos y la topología derivada usando la máxima potencia de transmisión.

Figura 2.4. Modelo gráfico de un GUA.

Cada nodo u debe aplicar el algoritmo árbol de expansión mínima (AEM) de Prim a su vecindad visible $VV_u(G)$, para obtener su árbol local de expansión mínima $A_u = (N(A_u), E(A_u))$. Para generar una estructura final que consista solamente en enlaces bidireccionales, hay que eliminar todos los enlaces unidireccionales. Para eso, el nodo u mantendrá el enlace (u, v) ? $E(A_u)$ en la topología final de red si y solo si el enlace (v, u) existe también en $E(A_v)$. Donde el nodo v pertenece al conjunto de vecinos de u, $V(u) = \{v ? N_u : u ? v\}$. Hay que notar que la topología derivada de los ALEM no es una simple superposición de todos los AEM locales. Para construir un AEM eficiente en función de la potencia de transmisión, es suficiente con que el peso de cada enlace este representado por la distancia Euclidiana, ya que el consumo de energía incrementa estrictamente en función de la distancia entre nodos [LHS].

Las siguientes líneas resumen, en pseudocódigo, el algoritmo ALEM:

para cada nodo u en la topología de red
definir su vecindad visible $VV_u(G)$
aplicar el algoritmo de Prim a $VV_u(G)$
para cada nodo u en la topología de red
para cada nodo v vecino del nodo u en A_u
si u es un vecino de v en A_v
el enlace E_{uv} *(u, v) pertenece al ALEM*
de otra manera
el enlace E_{uv} *(u, v) no pertenece al ALEM*

Los resultados son obtenidos por simulación sobre Grafos Unitarios Aleatorios para diversos números de nodos n: 20, 50, 100, 200, 500 y 1000. Se evalúan ambos algoritmos ALEM y AEM, y se comparan las siguientes características:

- Grado promedio (número promedio de vecinos por nodo)
- Promedio del número máximo de vecinos.
- Porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5
- Grado máximo encontrado dentro de cada prueba

Para generar y comparar los árboles, se usaron como función de peso de los enlaces tanto el retraso en la transmisión como la distancia Euclidiana entre nodos. El valor del retraso es generado de manera aleatoria de una distribución uniforme. Para cada prueba se corrieron 50 experimentos y los resultados obtenidos se muestran de la tabla 2.1 hasta la 2.8.

Lo que se puede observar de las tablas 2.1 hasta la 2.6, es que el máximo grado obtenido, tomando la distancia entre nodos como el peso de los enlaces, es de 5 para ambos algoritmos. El grado promedio de los nodos bajo el ALEM es muy cercano al AEM, el cual se caracteriza por tener el menor grado promedio ($2 - \frac{2}{n} \to 2, \text{si } n \to \infty$) de todos los subgrafos expandidos [LHS].

n = 20	AEM		ALEM	
	distancia	retraso	distancia	retraso
Grado Promedio	1.9	1.9	1.945	2.03
Promedio del Núm. Máx. de Vecinos	3.15	4.25	3.2	4.3
Grado Máximo	4	5	4	5

Tabla 2.1. AEM *vs* ALEM para 20 nodos usando distancia y retraso como peso de los enlaces.

n = 50	AEM		ALEM	
	distancia	retraso	distancia	retraso
Grado Promedio	1.96	1.96	2.04	2.176
Promedio del Núm. Máx. de Vecinos	3.35	4.45	3.35	4.55
Grado Máximo	4	6	4	6

Tabla 2.2. AEM *vs*. ALEM para 50 nodos usando distancia y retraso como peso de los enlaces.

n = 100	AEM		ALEM	
	distancia	retraso	distancia	retraso
Grado Promedio	1.98	1.98	2.039	2.237
Promedio del Núm. Máx. de Vecinos	3.4	5.35	3.4	5.7
Grado Máximo	4	6	4	7

Tabla 2.3. AEM vs. ALEM para 100 nodos usando distancia y retraso como peso de los enlaces.

n = 200	AEM		ALEM	
	distancia	retraso	distancia	retraso
Grado Promedio	1.99	1.99	2.099	2.306
Promedio del Núm. Máx. de Vecinos	3.9	5.7	4	6.1
Grado Máximo	5	8	5	8

Tabla 2.4. AEM vs. ALEM para 200 nodos usando distancia y retraso como peso de los enlaces.

n = 500	AEM		ALEM	
	distancia	retraso	distancia	retraso
Grado Promedio	1.996	1.996	2.123	2.394
Promedio del Núm. Máx. de Vecinos	4	6.55	4	6.95
Grado Máximo	4	8	4	8

Tabla 2.5. AEM vs. ALEM para 500 nodos usando distancia y retraso como peso de los enlaces.

n = 1000	AEM		ALEM	
	distancia	retraso	distancia	retraso
Grado Promedio	1.998	1.998	2.124	2.416
Promedio del Núm. Máx. de Vecinos	4	6.6	4	7.25
Grado Máximo	4	9	4	9

Tabla 2.6. AEM vs. ALEM para 1000 nodos usando distancia y retraso como peso de los enlaces.

Si el número de nodos incrementa, la conectividad de la red también y a pesar de esto, el algoritmo ALEM mantiene relativamente un grado bajo a pesar de que la densidad de la red aumenta. Si se usa el retraso en la transmisión como métrica de peso, pareciera no haber limite en el crecimiento progresivo del grado promedio de los nodos cuando el tamaño de la red crece. También se puede apreciar que al ser más grande la red, obviamente el radio máximo entre nodos se vuelve más pequeño ya que el área de distribución de los nodos no varía.

AEM	grado	1	2	3	4	5	>5
n = 20	distancia	28.75%	53.25%	17.25%	0.75%	0%	0%
	retraso	42.25%	34.75%	15.75%	5.25%	2%	0%
n = 50	distancia	24.40%	56.10%	18.60%	0.90%	0%	0%
	retraso	36.60%	39.50%	16.50%	6.20%	1.10%	0.10%
n = 100	distancia	23.75%	55.00%	20.75%	0.50%	0%	0%
	retraso	40.20%	34.35%	16.00%	6.60%	2.4%	0.45%
n = 200	distancia	23.25%	55.40%	20.475%	0.85%	0.025%	0%
	retraso	39.00%	34.25%	18.15%	6.40%	1.825%	0.375%
n = 500	distancia	23.04%	55.27%	20.74%	0.95%	0%	0%
	retraso	40.03%	33.27%	17.02%	7.17%	1.92%	0.59%
n = 1,000	distancia	23.16%	54.94%	20.82%	1.08%	0%	0%
	retraso	39.00%	33.97%	18.03%	6.72%	1.89%	0.39%

Tabla 2.7. Para el algoritmo AEM, porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5.

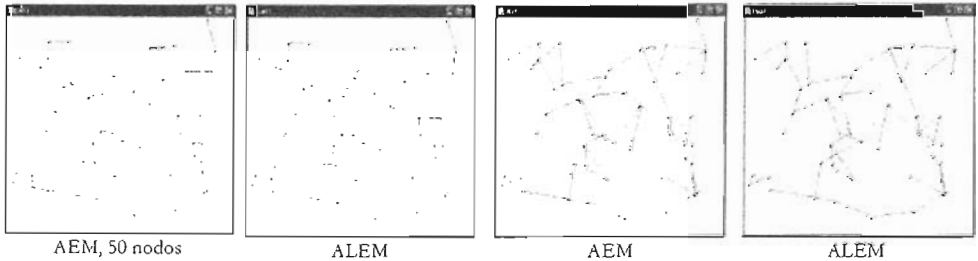
ALEM	grado	1	2	3	4	5	>5
n = 20	distancia	25.50%	55.50%	18.00%	1.00%	0%	0%
	retraso	34.25%	39.00%	18.50%	6.00%	2.25%	0%
n = 50	distancia	19.80%	57.00%	22.20%	1.00%	0%	0%
	retraso	27.30%	39.40%	23.50%	8.10%	1.6%	0.1%
n = 100	distancia	20.60%	55.45%	23.40%	0.55%	0%	0%
	retraso	29.30%	36.50%	21.00%	8.65%	3.55%	1%
n = 200	distancia	16.375%	58.45%	24.15%	1.00%	0.025%	0%
	retraso	26.575%	35.35%	23.925%	10.10%	3.325%	0.725%
n = 500	distancia	14.59%	59.53%	24.83%	1.05%	0%	0%
	retraso	24.05%	35.36%	23.98%	11.81%	3.67%	1.13%
n = 1,000	distancia	15.06%	58.65%	25.10%	1.19%	0%	0%
	retraso	23.14%	34.93%	25.37%	11.69%	4%	1%

Tabla 2.8. Para el algoritmo ALEM, porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5.

Las tablas 2.7 y 2.8 muestran el porcentaje de nodos que tienen desde un grado 1 hasta un grado >5. Para estas tablas se puede concluir que para ambos protocolos AEM y ALEM y usando la distancia como peso de los enlaces, más de la mitad de los nodos tienen grado 2 y ningún nodo tiene un grado mayor a 5. En redes inalámbricas, este dato es importante ya que un grado bajo implica menor interferencia con los vecinos.

La figura 2.5 presenta algunos grafos de ambos algoritmos, se mostrara que el AEM es un subconjunto del ALEM y que al usar el retraso como función de peso, no se obtiene grafos planos. Podemos concluir que el algoritmo ALEM construye una topología de red comparable a la que genera el algoritmo AEM sin la necesidad de una entidad central para obtenerla, construyendo una topología mínima que se aproxima a una estructura de árbol.

Finalmente podemos constatar que los resultados obtenidos por Li *et al.* son verificables y comparables plenamente a través de nuestro simulador.



AEM, 50 nodos ALEM AEM ALEM

Usando la distancia como peso Usando el retraso como peso

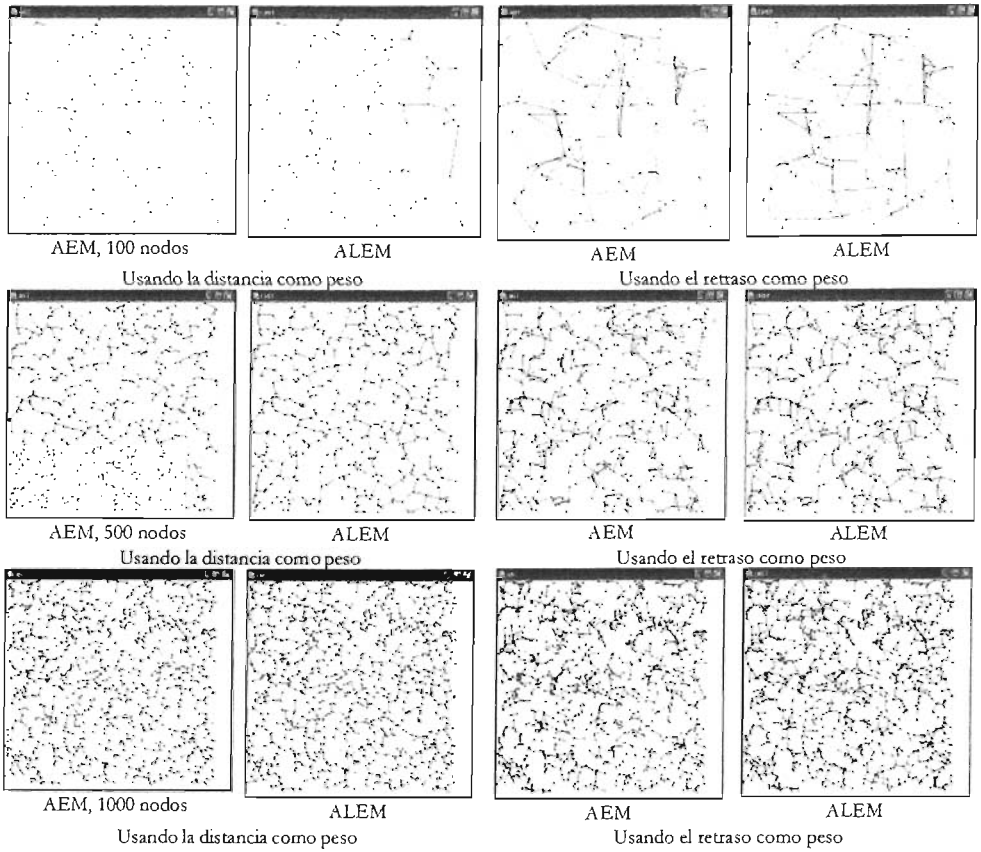


Figura 2.5. Comparación gráfica entre los algoritmos AEM y ALEM tomando como función de peso el retraso y la distancia. Cada renglón representa la misma distribución de los nodos.

Construcción del ALEM en Internet

El ALEM es aplicado en una topología Internet para definir un grafo de comunicación sobrepuesto de acuerdo a los conceptos descritos por Li *et al.* [LHS]. Aquí es importante resaltar algunos aspectos respecto a la aplicación de este algoritmo. La noción de vecinos descrita en [LHS] se refiere a la proximidad física entre nodos, de hecho, el ALEM es construido sobre un conjunto de nodos inicialmente desconectados cuya interconexión esta definida en términos de su cercanía vista como distancia Euclidiana. En el caso de Internet, la propuesta abordada es ligeramente distinta, porque el grafo inicial esta de antemano conectado con una distribución de

enlaces que obedece un patrón particular (leyes de potencia). Aquí el concepto de vecinos se refiere a la proximidad debida a la interconexión y no a la proximidad física, en este sentido, se dice que dos nodos son vecinos si existe una conexión directa entre sí, sin importar su ubicación en el espacio. A pesar de que en general, las redes sobrepuestas pueden usar enlaces virtuales entre dos nodos que no están directamente conectados (pero existe una ruta entre ellos), en este trabajo se investigarán solamente redes sobrepuestas que son subgrafos de la red Internet subyacente. Por ejemplo, las figuras 2.6.a y 2.6.b muestran dos topologías sobrepuestas de la topología física subyacente mostrada en la figura 2.6.c. Suponiendo que C_1 y C_3 pertenecen al mismo sistema autónomo mientras que C_2 y C_4 están en otro y que el enlace físico entre C_1 y C_4 tiene un retraso mucho mayor que los otros enlaces de la figura 2.6.c. Entonces en un escenario de desacoplo entre las topologías subyacente y sobrepuesta, donde se generan enlaces virtuales, el *broadcasting* hacia todos los nodos en la red de comunicación 2.6.a atravesará el enlace C_1C_4 cuatro veces. Si se pudiera construir una red sobrepuesta eficiente, el mensaje solo necesitaría viajar una vez en todos los enlaces físicos, figura 2.6.b.

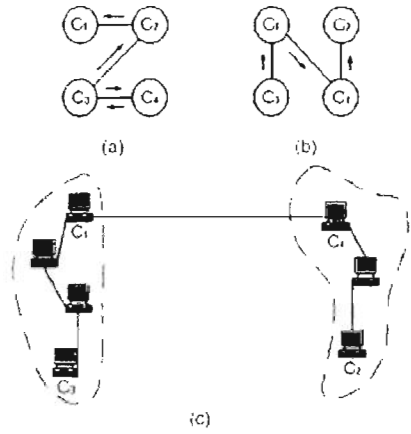


Figura 2.6. Topologías sobrepuestas (a) y (b), y subyacente (c).

No es difícil observar, que el algoritmo de *broadcasting* se desempeña mejor en una red de este último tipo, de hecho, el costo mínimo teórico para realizar el proceso de *broadcasting* en toda la red subyacente será la transmisión de un mensaje por nodo alcanzado, así que en una red con N nodos resultarán $N-1$ mensajes. El protocolo de *broadcasting* basado en el algoritmo ALEM es una derivación de este, comienza con los enlaces del grafo original y va eliminando algunos enlaces en el proceso.

En la aplicación descrita en esta tesis, la topología de red inicial $G = (N, E)$ define un grafo tipo Internet, en donde cada uno de los nodos, del conjunto de nodos N, tiene una localización estática y en donde los enlaces, del conjunto de enlaces $E = \{(u, v) : u, v \in N\}$, están establecidos de acuerdo al cumplimiento de las leyes de potencia [FFF] que rigen a las topologías Internet. Cada nodo además, tendrá asignado a él un identificador único, como puede ser su dirección MAC o IP.

En el proceso de construcción de la topología, cada nodo u debe establecer su vecindad visible $VV_u(G)$, la cual se definirá haciendo uso de la información relevante que le envíen todos sus vecinos inmediatos: el peso de cada uno de sus enlaces e identificador del nodo al cual se conecta. El nodo u solamente puede reportar su

propio registro de datos a todos sus vecinos de 1-*hop*, por ejemplo de los nodos v_1 hasta v_n con el peso a_n de su conexión, siendo n el número de vecinos:

ID	hacia-ID	peso
u	v_1	a_1
...		
u	v_n	a_n

Como se puede advertir, la información total recabada por cada nodo es local y puede ir definiendo una familia de estructuras, nombradas k-esimo árbol local de expansión mínima ($ALEM_k$). En el presente trabajo se ha considerado $0<k<=2$ y se deja para una investigación futura ampliar el rango de valores para k.

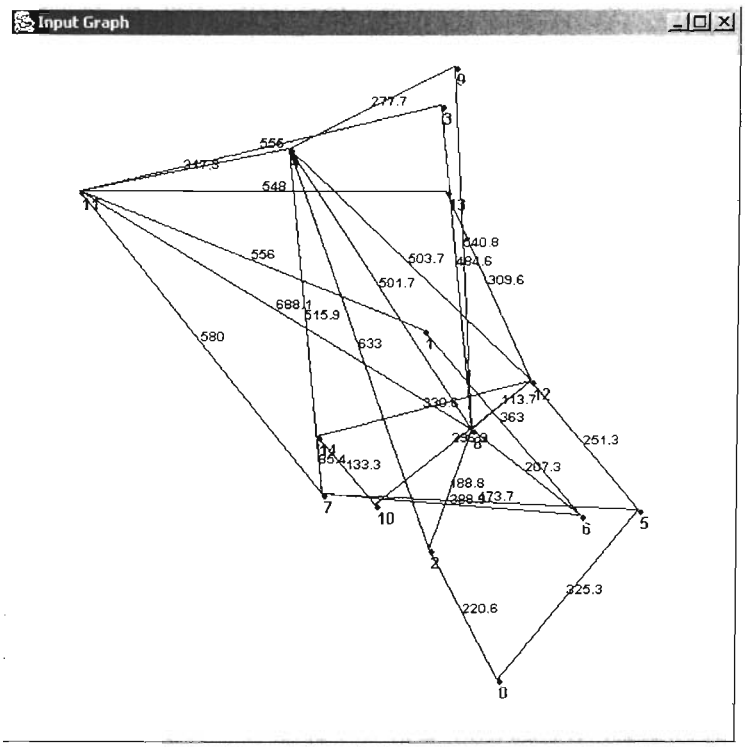


Figura 2.7. Ejemplo del grafo P con la distancia como el peso de sus aristas. Esta conformado por 15 vértices y 27 aristas, lo que significa un grado promedio igual a 3.6.

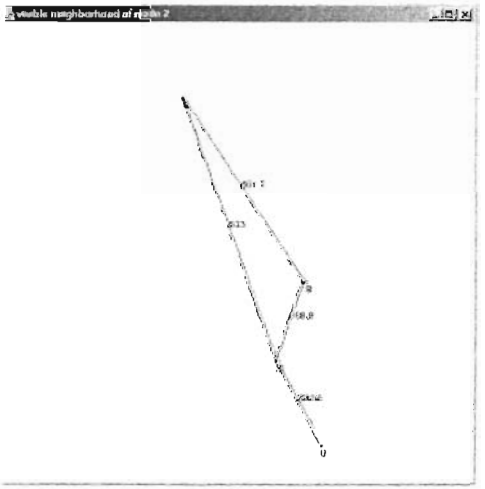
Del concepto generalizado que emplea [LHS] para construir el ALEM y que se refiere a la distancia geométrica entre nodos para ir formando la estructura, hemos implementado para Internet: $ALEM_1$ (k=1), donde como en [LHS], la proximidad

física se establece por los enlaces a los vecinos próximos y los enlaces que existen entre ellos, y $ALEM_2$ (k=2), hasta los vecinos de los vecinos, lo que resulta si se usa completamente la información recabada. Aunque para k=1 se tenga un cúmulo mayor de información sobre la vecindad que la necesaria, solo se pone atención en la información imprescindible y la restante no es tomada en cuenta.

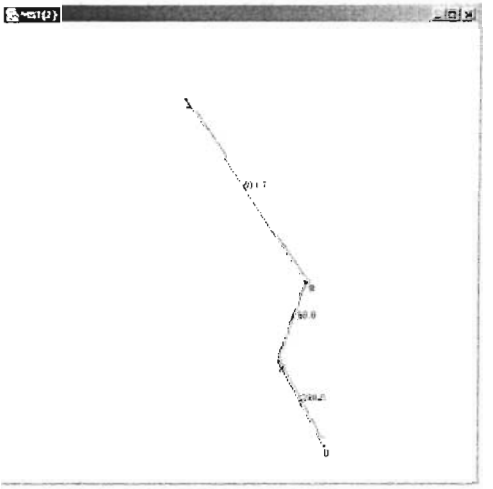
Consecuentemente se pueden definir para cada nodo dos vecindades visibles, una apoyándose en un conocimiento local debido a los vecinos de 1-*hop* y los enlaces entre ellos y la otra conformada hasta los vecinos de 2-*hops*. Una vez establecida la vecindad visible $VV_u(G)$, lo que sigue en el proceso del algoritmo en la conformación del árbol, corresponde con los mismos pasos que se efectuaron para una red inalámbrica.

$ALEM_1$. Si asumimos que el grafo P de la figura 2.7 corresponde a una topología Internet, entonces para el nodo 2, la Vecindad Visible $VV_2(P)$ y el Árbol de Expansión Mínima de 2, $A_2 = (N(A_2), E(A_2))$ serán los mostrados gráficamente en la figura 2.8.

De A_2 y obteniendo A_0 y A_8, se puede apreciar que el nodo 2 tiene enlaces bidireccionales con sus vecinos 0 y 8, así que los enlaces (2,0) y (2,8) pertenecerán a la estructura $ALEM_1$ final. Haciendo el mismo procedimiento para los nodos restantes es fácil obtener la topología $ALEM_1$ del grafo P (figura 2.9). Para este ejemplo sencillo de pocos nodos, la estructura $ALEM_1$ redujo de 27 aristas del grafo P original a 20.



Vecindad Visible del nodo 2, $VV_2(P)$.



AEM local del nodo 2, A_2.
Conjunto de vecinos del nodo 2, $V(2) = \{0, 8\}$.

Figura 2.8. Un ejemplo de la Vecindad Visible de un-*hop* y AEM local del grafo P.

Quizá sea necesario advertir que si cada nodo tomara únicamente como vecindad a los nodos de 1-*hop* de proximidad (sin los enlaces entre ellos), se obtendría una topología

tipo estrella cuyo AEM resulta en esta misma y no tiene sentido construir algún ALEM que dará como resultado la misma estructura inicial.

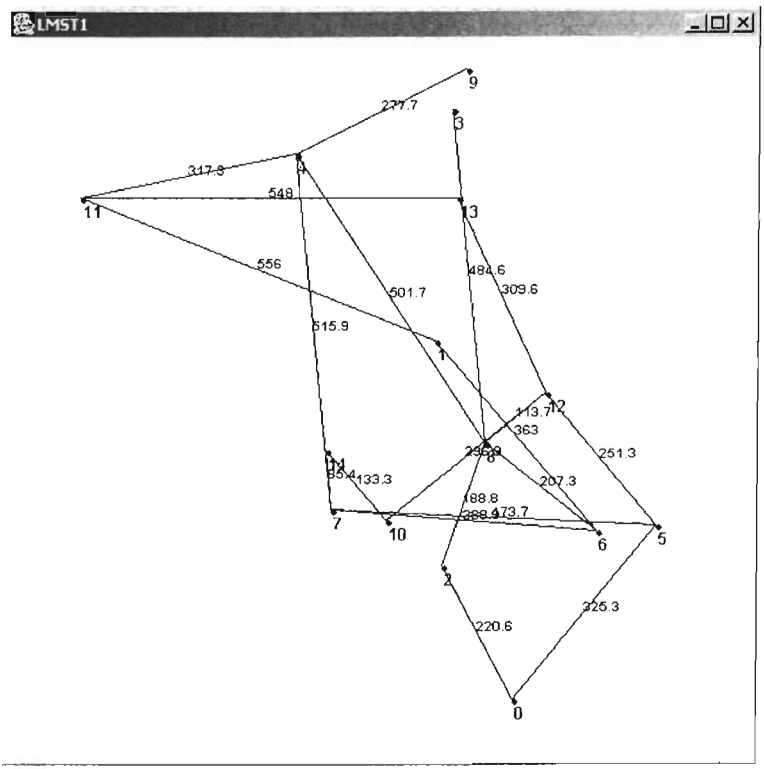


Figura 2.9. ALEM$_1$ del grafo P, con 20 aristas y un grado promedio igual a 2.667.

ALEM$_2$. Si el nodo u decidiera emplear toda la información que ha recabado por medio de sus vecinos inmediatos (el peso de sus enlaces y el identificador del nodo al cual se conecta), entonces su Vecindad Visible $VV_u(P)$ se extiende hasta los vecinos de dos-*hops*. Usando como ejemplo el mismo grafo P (figura 2.7) que se empleó para construir la estructura de árbol previa, se muestra gráficamente en la figura 2.11, la Vecindad Visible $VV_u(P)$ y el Árbol de Expansión Mínima $A_u = (N(A_u), E(A_u))$ del nodo u igual a 2. De A_2 y obteniendo A_0 y A_8, se puede apreciar que el nodo 2 tiene enlaces bidireccionales con sus vecinos 0 y 8, así que los enlaces (2,0) y (2,8) pertenecerán a la estructura final ALEM$_2$ (figura 2.10). En este caso simple que contiene pocos nodos, el Árbol Local de Expansión Mínima ALEM$_2$ redujo de 27 a 16 el número de aristas.

Hay que recordar que el k-esimo árbol local de expansión mínima ($ALEM_k$) se ha propuesto para aproximarse al árbol de expansión mínima AEM. En [LWS] se demuestra que el AEM es un subgrafo de $ALEM_k$ para cualquier k, lo que implica que todos los k-esimos árboles locales de expansión mínima son conectados cuando el grafo original también lo es.

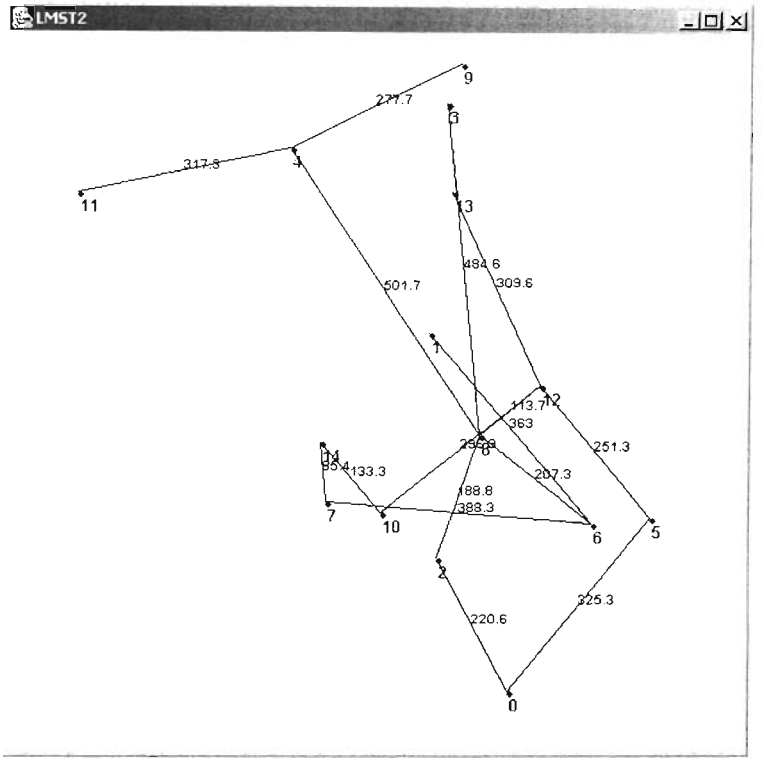


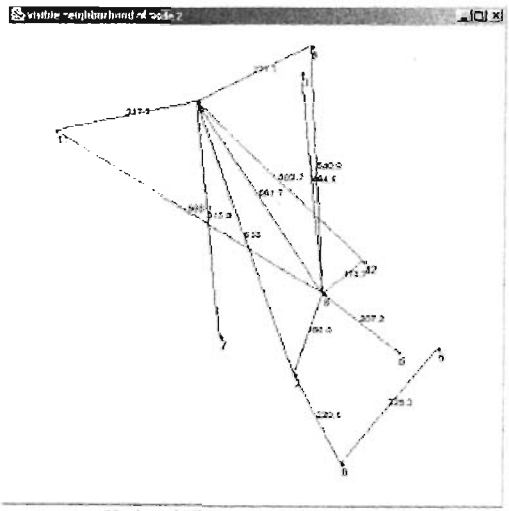
Figura 2.10. $ALEM_2$ del grafo P, con 16 aristas y un grado promedio igual a 2.133.

Lema. El árbol de expansión mínima AEM Euclidiano es un subgrafo del $ALEM_k$ para cualquier k.

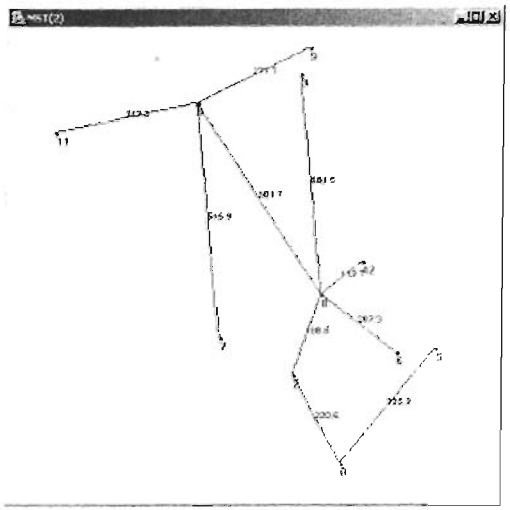
Prueba. Considerar cualquier enlace uv del AEM. Asumir que al AEM se añaden los enlaces de acuerdo a su longitud en orden ascendente. Al decidir agregar un enlace uv, no existía alguna ruta que conectase u y v usando los enlaces previamente añadidos. Esta propiedad se mantiene cuando el nodo u decide añadir el enlace uv al árbol de expansión mínima $AEM(N_k(u))$ de sus k-esimos vecinos $N_k(u)$. Esto implica que el

enlace w, pertenece al $AEM(N_k(u))$ y $AEM(N_k(v))$. Consecuentemente el AEM es un subconjunto de todas las estructuras $ALEM_k$ para cualquier k.

Esta familia de estructuras definen un árbol a partir de una topología Internet inicial, bajo la idea *a priori* de que esta nueva topología minimizará el costo en la transmisión de mensajes de *broadcast* al eliminar los enlaces redundantes de mayor peso y disminuyendo considerablemente el número de enlaces promedio por nodo preservando la conectividad de la red.



Vecindad Visible del nodo 2, $VV_2(P)$.

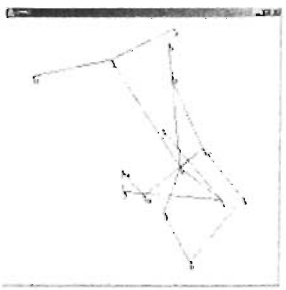


AEM local del nodo 2, A_2.
Conjunto de vecinos del nodo 2, $V(2) = \{0, 8\}$.

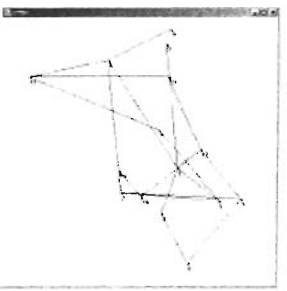
Figura 2.11. Figura 2.8. Un ejemplo de la Vecindad Visible de dos-*hops* y AEM local del grafo P.

La secuencia de grafos en la figura 2.8 muestra algunos ejemplos gráficos de las estructuras $ALEM_1$ y $ALEM_2$.

a) Topología Internet: nodos = 15, Grado Promedio = 3.6



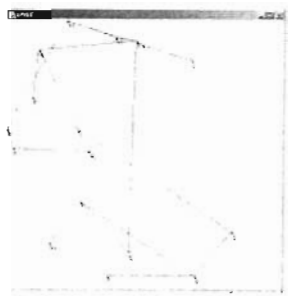
b) $ALEM_2$: Grado Promedio=2.1



c) $ALEM_1$: Grado Promedio=2.7

a) Topología Internet: nodos = 15, Grado Promedio = 5.2

b) $ALEM_2$: Grado Promedio=2



c) $ALEM_1$: Grado Promedio=2.4

a) Topología Internet: nodos = 15, Grado Promedio = 6.7

b) $ALEM_2$: Grado Promedio=2

c) $ALEM_1$: Grado Promedio = 2.3

Figura 2.8. Diferencias gráficas entre a) la topología de entrada con diferente grado promedio, b) $ALEM_2$ y c) $ALEM_1$. Usando la distancia entre nodos como función de peso.

Capítulo 3

Topología de Internet

A pesar de observar en Internet una estructura aparentemente azarosa, se ha descubierto que su topología se rige por algunas leyes-de-potencia.

Como el principal objetivo de esta tesis es el proponer una alternativa al *flooding* en Internet, nos enfrentamos ante la necesidad de tener un modelo que simule su topología. Este capítulo hace una revisión de la literatura sobre este tema, presenta los modelos que se usarán en las simulaciones y los mecanismos empleados para generarlas.

El estudio y las simulaciones hechas sobre Internet asumen ciertas propiedades topológicas (diámetro, distribución de la conexión entre nodos, etc.) que se han aplicado para generarla sintéticamente. Se han propuesto cuatro leyes de potencia y sugerido usarlas para medir el realismo de los modelos que la generan y se ha demostrado por ejemplo, que los grafos generados aleatoriamente, que tradicionalmente describían topologías complejas, no reflejan correctamente las propiedades de su topología.

En general, la estructura topológica de una red es típicamente modelada usando un grafo. El término grafo se usará, a lo largo de este trabajo, para designar un conjunto de puntos (vértices o nodos) que modelan a los dispositivos de red, conectados por líneas bidireccionales (aristas o enlaces) que representan conexiones de comunicación; donde no se permiten ni enlaces múltiples ni *loops* que conecten a un nodo consigo mismo.

Aunque la teoría de grafos tiene una historia de poco más de dos siglos, tan solo en años recientes ésta ha sido aplicada rutinariamente a estructuras inmensas como Internet, estructuras con millones de vértices y enlaces. Al mirar el dibujo de un grafo, es difícil no enfocarse en el arreglo de los puntos-vértices y las líneas-enlaces, pero en la teoría de grafos todo lo que importa es el patrón de conexión: la topología, no la geometría [H]. Pero ¿por qué es tan importante caracterizar la topología de una red? Porque la estructura influye siempre en el funcionamiento. Por ejemplo, la topología de las redes sociales afecta el esparcimiento de enfermedades y de información, y la topología de una red eléctrica afecta la robustez y estabilidad de la transmisión de

energía 5]. Los grafos de la figura 2.1 muestran que el confiar únicamente en su representación visual o en su estructura geométrica puede ser un terrible error, por ejemplo las dos topologías en la figura 2.1 tienen cada una 100 vértices y aproximadamente el mismo número de enlaces (230 versus 231).

Medir las propiedades de un grafo, como serían su diámetro o la distribución del grado de los vértices o mejor aun, si se pudiera catalogar con pleno detalle los vértices y los enlaces de una grafo tan grande, constituirían apenas ambos, un primer paso para entender su estructura; el siguiente paso sería desarrollar un modelo matemático de ésta, el cual típicamente toma la forma de un algoritmo que generara grafos con las mismas propiedades estadísticas [H].

En términos estrictos, la topología de Internet se refiere a la estructura física en un momento determinado y el grafo de Internet se refiere a la abstracción matemática que representa dicha estructura, sin embargo, sin perdida de formalidad, usaremos ambos términos indistintamente.

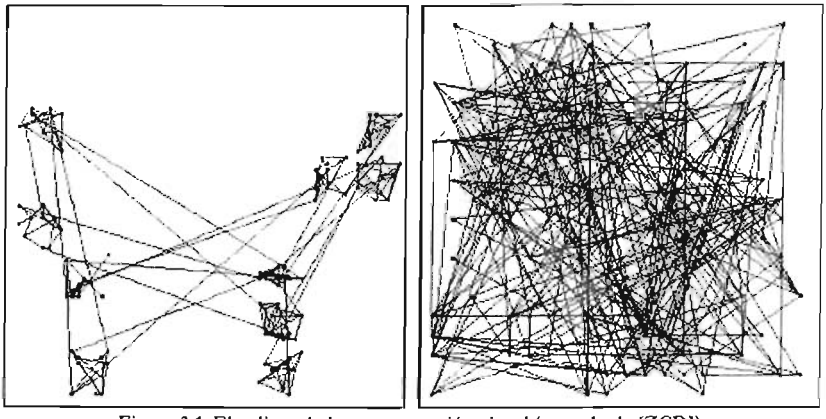


Figura 3.1. El peligro de la representación visual (tomado de [ZCD]).

Internet no es un sistema que haya sido diseñado deliberadamente, se asemeja más a un mecanismo social que ha crecido a través de algún proceso evolutivo determinado por patrones de densidad de población alrededor del mundo[YJB].

Algunas razones que pudieran explicar el acrecentado interés en Internet consisten, primero, en el incremento dramático durante los últimos años, de la cantidad de datos topológicos disponibles de su estructura. Segundo, el crecimiento de la red hasta un límite nunca antes visto y su propagación, han logrado influenciar muchos ámbitos de nuestras vidas, lo que ha creado la necesidad de entender la topología, el origen y la evolución de su estructura. Finalmente, el enorme poder de cómputo disponible en casi cualquier PC, ha permitido estudiar este sistema a un detalle sin precedente [FDBV]. Los beneficios de entender la topología de Internet [SSK], estriban en:

1. Poder diseñar protocolos más eficientes que aprovechen estas propiedades topológicas.

   Los protocolos de comunicación y seguridad tienen un desempeño pobre en topologías ofrecidas por generadores distintos para los cuales estos han sido optimizados y son también frecuentemente ineficientes cuando se liberan para su uso en Internet [YJB]. En particular, los protocolos diseñados para Internet deberían ser probados en topologías generadas tipo Internet [MaP].

2. Poder generar modelos artificiales más precisos para propósitos de simulación.

   Es un hecho que el modelo de una topología tiene un efecto fundamental en los resultados de simulación de protocolos, un ejemplo de la consecuencia de las decisiones de topología en la investigación de redes, fue el trabajo de Albert et al. [AJB] el cual mostró que las topologías que cumplen con las leyes de potencia son resistentes a fallas aleatorias en sus nodos (porque hay muy pocos nodos con muchos enlaces, así que estos no dominan la topología) pero por lo mismo, vulnerables a ataques premeditados (virus), mientras que las redes conectadas de manera aleatoria son sensibles tanto a los ataques como a las fallas.

3. Obtener estimaciones de parámetros topológicos, que pueden ser útiles para el análisis de protocolos y para especular sobre la topología de Internet a futuro, ya que existen múltiples razones –sociológicas, matemáticas y comerciales- para estudiar la evolución de esta red [KKRRT].

   Aunque si bien, predecir la evolución de un sistema dinámico como lo es Internet no es trivial, ya que existen múltiples factores (sociales, económicos y tecnológicos) que pueden alterar significativamente su topología [FFF].

Niveles en la topología

Internet es una red de conexiones físicas entre computadoras, ruteadores, puentes y muchos más dispositivos de telecomunicaciones. La topología de Internet ha sido estudiada desde enfoques y aproximaciones distintas de acuerdo a su nivel de interconexión, ver la figura 3.2. A nivel de ruteador, los nodos son los ruteadores y los enlaces son las conexiones físicas entre ellos separados, a nivel IP, por un *hop* [TGJSW]. A nivel de interdominio (o Sistema Autónomo) cada dominio, compuesto de centenares de ruteadores y computadoras coordinados por un dominio administrativo que comparte información y políticas de ruteo (y que posee uno o varios sistemas autónomos [BT]), está representado por un simple nodo y un enlace se presenta entre dos dominios si hay por lo menos una ruta que los conecta [AB,CDZ]. Una característica fundamental de estos dominios es su ruteo en localidad: la trayectoria entre dos nodos cualquiera en un dominio permanece enteramente dentro de él.

Debe mencionarse que el modelado de la topología de Internet se basa en la información actualmente disponible, la cual no es del todo correcta debido principalmente a que el obtener un mapa de la distribución de nodos en Internet es una labor extremadamente compleja [GT, TR]. A nivel de sistemas autónomos, la información disponible es relativamente extensa por que ésta puede ser obtenida de las tablas empleadas por el BGP (Border Gateway Protocol) [NLANR, G] y aun así, no hay ninguna razón a priori para pensar que las rutas seguidas por el protocolo BGP en los sistemas autónomos capturan completamente la topología a este nivel [CCGJSW]. En particular porque en la manera en que el ruteo BGP funciona, una muestra instantánea de una tabla de ruteo puede ser que solo revele los enlaces que pertenezcan a las rutas de mayor preferencia más no a todas las que existen almacenadas en el ruteador, lo que hace que solo se conozca parcialmente la conectividad de los sistemas autónomos en Internet [BC]. Una topología de Internet real a nivel de sistemas autónomos es aún desconocida aunque si bien puede ser inferida de las tablas de ruteo del BGP [BT], sería imprudente deducir las propiedades de Internet a través únicamente, de estos datos [BC].

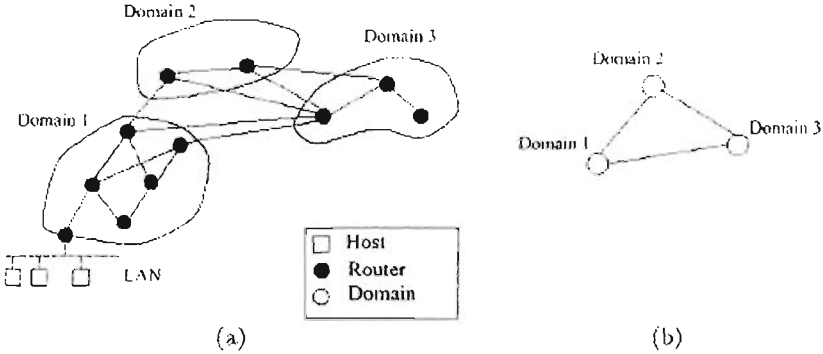


Figura 3.2. Estructura de Internet a (a) nivel de ruteadores y (b) nivel inter-dominio (tomada de [FFF]).

Pero es mucho más difícil aún obtener información completa de la topología a nivel de ruteadores, la cual se consigue por medio de mecanismos de trazado de ruta y robots de rastreo [GT, CM]. Estas gráficas representan a Internet en un nivel mucho más fino de granularidad y contiene aproximadamente 17 veces más nodos y enlaces que el grafo a nivel de sistemas autónomos, aunque si bien ambas son representaciones de Internet, no esta claro, cómo es que ambas topologías de la red (vistas como un grafo), tengan tanto en común [TGJSW]. Por lo expuesto en [GT, CM, MMB] se sabe que es a nivel de ruteadores donde los trabajos de investigación aun no pueden considerarse como concluyentes, pues el campo es dinámico, e Internet demasiado extenso para ser cubierto por un solo grupo de investigación [CM, NLANR], sin embargo, en la medida en que puedan conocerse las propiedades a este nivel, podrán desarrollarse protocolos enfocados en la capa IP que optimicen los recursos disponibles, porque son éstas

topologías, las que brindan mayor precisión para la simulación de algoritmos en la capa de red [MaPa].

A pesar de todo, los datos que se tienen de las topologías, reflejan lo mejor que se ha podido obtener, es claro que para ambas estructuras –sistemas autónomos y ruteadores- se está lejos de tener representaciones perfectas de Internet. No solo porque están sujetas a errores y omisiones, sino porque reflejan únicamente su topología y no contienen ninguna información acerca de las características de los enlaces [TGJSW]. Aunque si bien, algunas técnicas para estimar las capacidades de los enlaces a lo largo de una ruta son conocidas [D, LB], también se ha reportado sobre el alto consumo de tiempo para lograrlo y hasta donde se sabe, nadie a intentado obtener una topología de la totalidad de Internet (a nivel-ruteadores) con la información sobre las características de sus enlaces [TGJSW].

Si bien el estudio de la topología a ambos niveles es igual de importante, en este trabajo nos enfocamos en la aplicación que se le pudiera dar al algoritmo de *broadcasting* a nivel de ruteadores, pues es en este nivel en el que se concentran las aplicaciones de usuario que analizaremos posteriormente.

Leyes de potencia

Las leyes de potencia que caracterizan a Internet son de la forma $y = x^{\alpha}$ [FFF] y son expresiones que permiten representaciones compactas de las topologías por medio de sus exponentes. Estas relaciones (x, y), trazadas en una escala log-log definen la pendiente α de la línea resultante, el exponente de la ley de potencia [MMB]. Se pueden usar estas pendientes para evaluar si dos topologías tienen propiedades similares. Así por ejemplo, Faloutsos et al. [FFF] mostraron que en diferentes topologías de Internet a nivel interdominio, todas ellas de distintos tamaños y obtenidas en diferentes periodos de tiempo, se tenían pendientes casi iguales. Sus observaciones también indicaron que la topología a nivel ruteador, tenía pendientes significativamente diferentes a aquellas topologías a nivel interdominio.

Faloutsos et al. [FFF] descubrieron cuatro leyes de potencia en tres muestras de topologías a nivel interdominio tomadas entre noviembre de 1997 y diciembre de 1998, con 3015, 3530 y 4389 nodos y 5156, 6432 y 8256 enlaces respectivamente y en una a nivel ruteador tomada en 1995 con 3888 nodos y 5012 enlaces. En [GT], Govindan *et al.* reportaron que estas mismas leyes de potencia permanecían siendo válidas en una muestra de 1999 a nivel ruteador. Recientemente, Magoni et al [MagP], encontraron en topologías a nivel interdominio otras leyes de potencia adicionales, comprobando que éstas también se mantienen a nivel ruteadores.

La validación de estas leyes de potencia esta determinada por el coeficiente de correlación lineal (CCL) de Pearson; este valor varia de –1 a 1, pero normalmente se toma el valor absoluto. Se ha considerado que para un coeficiente mayor a 0.98 se tiene

una relación lineal válida entre las dos variables, lo que es equivalente a medir un buen ajuste de bs datos originales con respecto a la aproximación lineal por medio de mínimos cuadrados [MagP, PaS]. El método de mínimos cuadrados asume que la curva que mejor se ajustará a los datos originales, será aquella que minimice la suma total del error al cuadrado, siendo el error, la diferencia entre los datos originales y los valores de la función.

Se ha sugerido usar las siguientes 4 leyes para medir el realismo de las gráficas generadas sintéticamente [FFF]:

Ley de potencia 1 (exponente de orden R). El número de conexiones de salida, d_v, de un nodo v, es proporcional al orden del nodo, r_v, elevado a una constante R: $d_v \propto r_v^R$. El orden es la posición del nodo en una tabla clasificada (numéricamente decreciente) de acuerdo a su número de conexiones de salida.

Esta ley de potencia se evalúa calculando el número de conexiones de salida para cada nodo, ordenando en forma descendente los valores calculados, asignando un índice a cada uno de estos valores, mismo que corresponde al orden r_v. El exponente de orden R se define por la pendiente que resulta de graficar d_v versus r_v en una escala log-log. La información recolectada para topologías de interdominio y para la de ruteadores exhibe una distribución cuasi lineal, con una ligera desviación en ambos extremos [PaS], ver la figura 3.3.

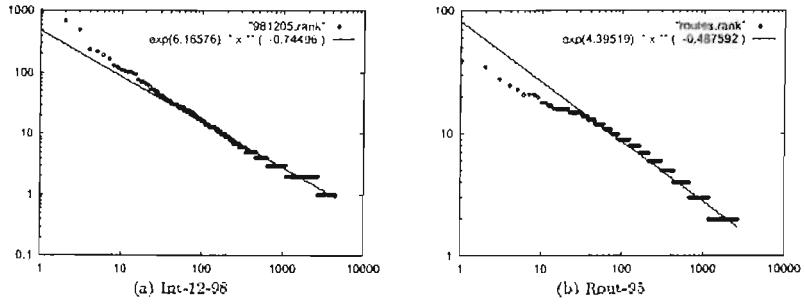


Figura 3.3. Trazo de la Ley de Potencia 1 (d_v versus r_v), de una gráfica (a) a nivel inter-dominio con R = -0.74 y (b) a nivel ruteadores con R = -0.49. Tomadas de [FFF]

De los datos recabados por [FFF] para topologías de interdominio, el exponente R resultó ser: -0.81, -0.82 y -0.74, y para la de ruteadores -0.48. Esta diferencia en los valores, hizo sugerir que el exponente de grado puede distinguir gráficas de distinta naturaleza y usarlo como una métrica para caracterizar familias de gráficas. Y más aún, este parámetro no depende de el tamaño de la topología, lo que augura ser un buen indicador [MP].

La constante de proporcionalidad de la ley-de-potencia 1 que estimó [FFF], requiere que el número mínimo de conexiones de salida de la topología sea 1 Así, la ley de potencia se puede redefinir como: $d_v = \frac{1}{N^R} r_v^R$ y el número de enlaces, E, de una gráfica como: $E = \frac{1}{2(R+1)}\left(1 - \frac{1}{N^{R+1}}\right)N$, siendo N el número de nodos de la gráfica.

Ley de potencia 2 (exponente de grado-enlaces de salida[5] ?). La frecuencia, f_d, de un número de enlaces de salida, d, es proporcional al número de enlaces de salida elevado a una constante ? : $f_d \propto d^O$.

En [FFF], el valor del exponente ? es prácticamente constante en las gráficas de las topologías de interdominio: -2.15, -2.16 y –2.2. Mientras para la topología a nivel ruteador, es –2.48, ver la figura 3.4. La intuición detrás de esta ley de potencia sugiere que la distribución de las conexiones de salida de los nodos de Internet no es arbitraria. La observación cualitativa indica que el número de conexiones de salida más pequeños son los más frecuentes [FFF].

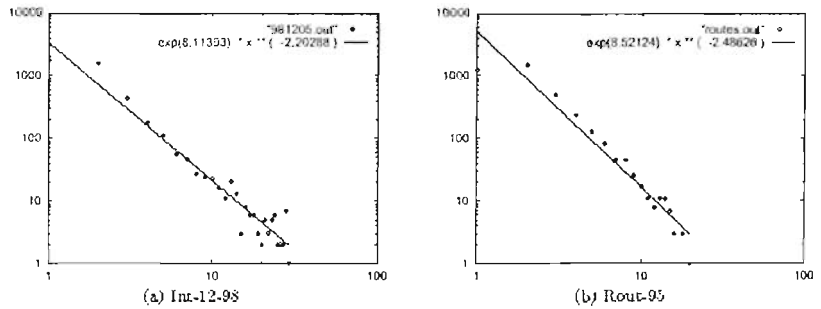


Figura 3.4. Trazo de la Ley de Potencia 2 (f_d versus d), de una gráfica (a) a nivel inter-dominio con ? = -2.2 y (b) a nivel ruteadores con ? = -2.48. Tomadas de [FFF]

Muchos investigadores han corroborado, que la conectividad o la distribución del grado de los nodos, en la WWW y en redes sociales –en general- obedece también a una ley de potencia: la probabilidad de que al escoger aleatoriamente un nodo tenga un grado i, es proporcional a $1/i^x$, para algún exponente $x > 1$ constante [CJPP]. Para sistemas reales, x se encuentra dispersa entre un rango de 2.1 y 3, lo que suscita una

[5] De enlaces-de-salida o simplemente de enlaces, porque se están considerando gráficas no direccionadas [MaPa].

importante cuestión acerca de la universalidad en la conformación de las redes (la probabilidad de encontrar un vértice con i enlaces es proporcional a i^{-3}) [AB].

Ley de potencia 3 (exponente de saltos-*hops* H). El número total de pares de nodos, $P(h)$, dentro de h saltos, es proporcional al número de saltos elevado a una constante H : $P(h) \propto h^H$, $h \ll \delta$ siendo δ el diámetro de la topología.

Esta ley cuantifica la conectividad y las distancias (por saltos) entre los nodos de Internet, usando el número total de pares de nodos $P(h)$ dentro de una vecindad de h saltos; tomando en cuenta a los pares que se generan consigo mismo y dos veces a los otros pares. Es decir, para $h = 0$, solamente se tienen los pares consigo mismo: $P(0) = N$. Para el diámetro, δ, de la gráfica, $h = \delta$, se tienen los pares consigo mismo más todos los otros posibles pares: $P(\delta) = N^2$, lo cual representa al número máximo posible de pares de nodos [FFF].

En [FFF], para los tres conjuntos de datos a nivel interdominio se tienen prácticamente valores iguales para el exponente de saltos: 4.6, 4.7 y 4.86. Lo que muestra que este exponente describe un aspecto de la conectividad de la gráfica en un solo número. Para los datos a nivel ruteador, el exponente de saltos es 2.8. Ver la figura 3.5.

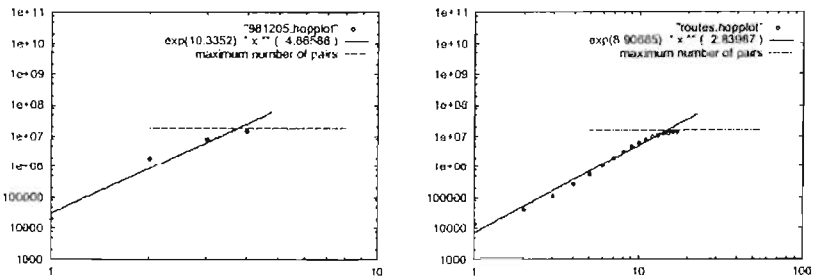


Figura 3.5. Trazo de la Ley de Potencia 3 ($P(h)$ dentro de h saltos versus h), de una gráfica (a) a nivel inter-dominio con H = 4.86 y (b) a nivel ruteadores con H = 2.8. (Tomadas de [FFF])

Una aplicación para redes derivada de esta ley, esta planteada en [FFF], y se refiere a cuando se tiene la necesidad de llegar a un nodo destino sin conocer su posición exacta, en cuyo caso se necesita seleccionar un diámetro efectivo para idealmente saber cuantos saltos se requieren para llegar a una parte suficientemente extensa de la red a fin de restringir el *broadcast* o la búsqueda de ese nodo a esa área. Es decir, dos nodos cualesquiera se encuentran dentro del δ_{ef} de saltos con una probabilidad bastante alta. El diámetro efectivo, δ_{ef}, se define como: $\delta_{ef} = \left(\frac{N^2}{N+2E} \right)^{1/H}$. Por ejemplo, el diámetro efectivo para una topología a nivel inter-dominio esta un poquito arriba de 4.

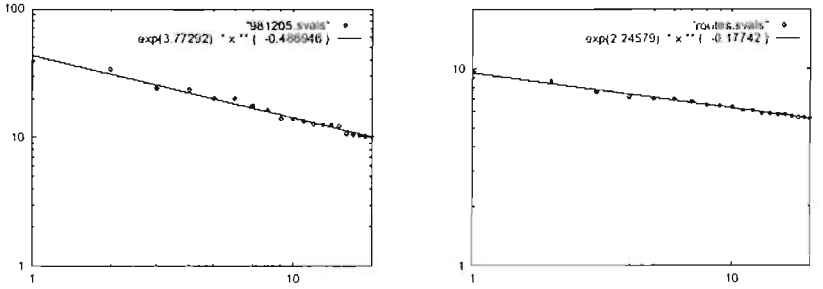
Redondeando el diámetro efectivo a 4, aproximadamente el 80% del par de nodos se encuentra dentro de esta distancia. El techo del diámetro efectivo es 5, lo cual cubre más del 95% del par de nodos. De cualquier manera, esto no es más que una aproximación porque los cálculos están hechos para valores de h que son mucho menores que el diámetro de la red [SS].

Por ejemplo, el diámetro de la WWW es 19, lo que significa que si se seleccionan dos páginas al azar, entre un total de 800 millones que existen en la Web, el número de enlaces en promedio que las separarán será sorprendentemente 19, a pesar de tener la opción de vagar sobre rutas muy largas [H].

Ley de potencia 4 (exponente de eigen-valores E). Los eigen-valores, λ_i, de la matriz de adyacencia de una gráfica, son proporcionales al orden, i, elevado a una constante, E: $\lambda_i \propto i^E$.

Esta ley de potencia se muestra en [FFF] solamente para los primeros 20 eigen-valores. Los exponentes de eigen-valores para las tres gráficas de inter-dominio son prácticamente iguales: -0.47, -0.50 y -0.48, mientras que para la gráfica de ruteadores es significativamente diferente a las anteriores, -0.177. Ver la figura 3.6.

Los eigen-valores de una gráfica están estrechamente relacionados a muchas propiedades topológicas básicas, como son el diámetro, el numero de enlaces, el número de rutas de cierta longitud entre vértices, como se puede ver en[FDBV].



(a) Int-12-98 (b) Rout-95

Figura 3.6. Trazo de la Ley de Potencia 4 (λ_i en orden decreciente), de una gráfica (a) a nivel inter-dominio con E = -0.48 y (b) a nivel ruteadores con E = -0.177. (Tomadas de [FFF])

Medina *et al.* [MMB] encontraron que:

- Las leyes de potencia 1 y 2 son las más adecuadas para distinguir los diferentes tipos de topologías. Siendo así que los exponentes de orden R y de enlaces de

salida ? proporcionan un medio poderosos para diagnosticar la apariencia de una topología con relación a una tipo Internet.

- Las leyes de potencia 3 y 4 se observan en casi todas las topologías, no importa que estas sean de diferentes clases (aleatorias, jerárquicas o regulares). Sin embargo las distintas topologías difieren en los valores de sus exponentes H y E.

Modelos de Internet

El crecimiento explosivo de Internet ha venido acompañado de un amplio rango de problemas en torno a su funcionamiento, relacionados al ruteo, a la reservación de recursos y a su administración. El estudio de los algoritmos y sus políticas, orientados a estos problemas a veces implica simulaciones o análisis empleando una abstracción o un modelo de la estructura actual de la red; la razón es clara, las redes que son suficientemente grandes para ser interesantes de analizar son también caras y difíciles de controlar, por lo mismo raramente están disponibles para propósitos experimentales, además de que resulta generalmente más eficiente evaluar soluciones empleando una simulación. Así que los modelos de red son muy importantes, ya que las conclusiones obtenidas acerca del desempeño y conveniencia de un algoritmo variarán dependiendo de los métodos usados para modelarla [ZCD].

El estado de arte en el modelado de redes incluye tres clases:

1. **Modelos aleatorios**. Representado por el modelo de Waxman [W], que se deriva y extiende del modelo clásico de Erdos-Renyi [B] de generar graficas aleatorias (asignar una probabilidad uniforme para crear un enlace entre un par de nodos) por asignar de manera aleatoria la localización de los nodos en un plano y crear los enlaces con una probabilidad que es función de la distancia euclidiana entre nodos.

   Con lo que respecta a la generación de graficas realistas, Waxman introdujo lo que parece ser uno de los modelos de redes más populares. Este modelo fue exitoso al representar las primeras redes pequeñas como lo fue ARPANET. A medida que el tamaño y la complejidad de la red incrementó, fueron necesarios modelos más detallados [FFF].

2. **Modelos estructurales**. Forman grafos que reflejan la jerarquía de las estructuras de los dominios que están presentes en Internet (*host*-ruteadores-sistemas autónomos). Contiene los modelos Transit-Stub[6] [CDZ] y Tiers[7]

[6] El código fuente de la implementación de Transit-Stub se encuentra en: http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html

[7] El código fuente de Tiers esta disponible en: ftp://ftp.nexen.com/pub/papers

[Do], ambos tuvieron originalmente la intención de modelar la topología de Internet a nivel de ruteadores [TGJSW].

En Transit-Stub, el proceso de generación empieza en el nivel de jerarquía superior (WAN/dominios de transito) y continúa hacia abajo hasta el nivel inferior (LAN). Dentro de los dominios de transito los nodos son conectados aleatoriamente. Ligados a cada dominio de transito están algunos dominios anexos generados de manera similar. Unidos a cada nodo anexo se encuentran las LANs, que son modeladas con una topología de estrella con un nodo de ruteo en el centro. Se añaden aleatoriamente, con base a un parámetro especifico, otros enlaces de dominios de transito-a-anexos y de anexos-a-anexos para crear redundancia, ver la figura 3.7.

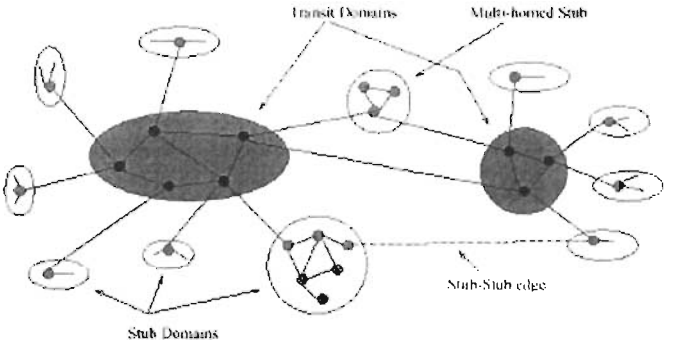


Figura 3.7. Ejemplo de la estructura de dominios de Internet (tomada de [ZCD]).

Tiers también usa tres niveles de jerarquía, referidos como WAN, MAN y LAN. Este modelo produce sub-graficas conectadas uniendo todos los nodos en un solo dominio usando un árbol de expansión mínima. Los enlaces redundantes se añaden con el fin de incrementar la distancia euclidiana dentro de una red del mismo nivel o entre redes de distinto nivel, pero siempre a los nodos más cercanos, lo que asegura cierto grado de conectividad local dentro de un límite geográfico restringido. Los nodos LAN son conectados usando una topología estrella. Ver la figura 3.8.

Un nodo *gateway* es aquel que conecta dos tipos de redes y es representado topológicamente como dos nodos interconectados, por ejemplo, un nodo LAN y un nodo MAN. La métrica del enlace entre estos dos tipos de nodos reflejará el funcionamiento interno del *host*, por ejemplo, el retardo en el procesamiento de la transferencia de datos de una red a otra. Esta

aproximación permite estimar mejor el desempeño de la red en lugar de usar un modelado más estricto en donde todos los enlaces representen una conexión física en la red [CDZ].

Este tipo de modelos fueron ampliamente usados por la comunidad de investigación en Internet, en el entendido de que Internet refleja una estructura deliberadamente jerárquica. Hasta que en 1999 el artículo de Faloutsos *et al.* revelara que la distribución del grado promedio de los nodos es una ley de potencia [TGJSW]. Después de todo, las redes reales, no son del todo regulares ni del todo aleatorias [H].

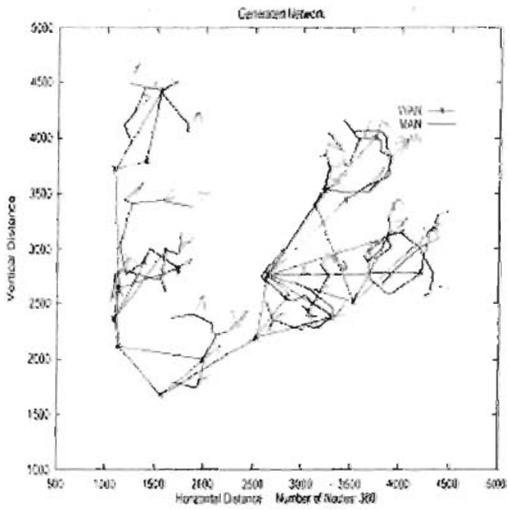


Figura 3.8. Una típica red Tiers, tomada de [CDZ].

3. **Modelos de las leyes de potencia (ley de potencia de grado).** Con el fin de generar topologías apegadas a los resultados empíricos existentes, surgen estos modelos βA, PaS, CZF, MP, JCJ], que producen gráficas con una distribución de ley de potencia del grado promedio de los nodos pero difieren en la manera en que los nodos se conectan entre si [TGJSW].

Pero aún con todas estas propuestas, la pregunta principal permanece abierta: ¿cómo validar un modelo de Internet? En este trabajo, no se usará algún modelo estructural (Transit-Stub o Tiers), ya que Tangmunarunkit *et al.* [TGJSW], en un trabajo enfocado en determinar que modelo de red representa mejor la estructura jerárquica de Internet a gran escala, encontraron que los generadores de topologías que cumplen con la ley de potencia de grado, producen una forma de jerarquía (a pesar de no incluir esta

propiedad explícitamente en su metodología de diseño) que se asemeja más a la libre e imprecisa naturaleza jerárquica de Internet más de lo que logran los modelos estructurales en sus topologías de riguroso modelado jerárquico. Además, con el fin de comparar que tipo de modelos reflejan las propiedades puramente locales de una gráfica, como son las Leyes de Potencia, Medina et al [MMB] concluyeron, que los modelos que cumplen únicamente con la ley de potencia de grado modelan mejor Internet que un modelo estructural (Transit-Stub) u otro de generación de grafos aleatorios (Waxman) que además [SS] esta lejos de modelar una topología de red real. Lo que hace a los modelos estructurales obsoletos [MP].

Para el modelado de la topología de Internet, se emplearon cuatro modelos distintos, uno de los cuales se considera obsoleto actualmente, pero esta incluido como referencia y punto de comparación. Los otros tres cumplen con la distribución de ley de potencia del grado promedio de los nodos, la cual se considera como el principal y el más importante parámetro en el modelado de la topología de Internet [BT].

Waxman (1988) [W]. Este modelo distribuye geográficamente los nodos de manera aleatoria en un espacio cartesiano de dos dimensiones y usa la distancia euclidiana entre ellos como el factor más importante en la topología para regir su conectividad: $P(u,v) = \beta e^{-d(u,v)/L\alpha}$. Donde la probabilidad $P(u,v)$ de conectar a los nodos u y v es inversamente proporcional a la distancia $d(u,v)$ entre ellos, L es el diámetro euclidiano de la red y α y β son parámetros.

Este modelo nunca intentó el modelar específicamente la topología de Internet [MP], así que es natural que con base a las leyes de potencia, la topología de Waxman no sea representativa de una topología Internet, con lo que respecta al exponente de orden R y al de enlaces de salida? ; con lo que respecta al exponente de saltos H , la conclusión puede ser engañosa debido el escaso número de datos con el que se calculó ya que estos indican que la topología de Waxman genera la relación de ley de potencia, al igual que para el exponente de eigen-valores E [MMB, PaS].

Barabasi-Albert (1999) [BA, AB]. Con el fin de producir gráficas que cumplan con la ley de potencia de los enlaces de un nodo, este modelo toma en cuenta, como posibles causas, tres mecanismos generales que guían la evolución de la estructura de una gráfica a lo largo del tiempo:

1. **Crecimiento progresivo**. Resulta de observar el desarrollo que la mayoría de las redes siguen a través del tiempo, al añadir nuevos nodos y nuevos enlaces a la estructura de red existente, lo que logra formar redes que sufren un incremento gradual en su tamaño.

2. **Conectividad preferente**. Expresa un fenómeno que se encuentra frecuentemente en una red –social- y es que existe una mayor probabilidad de que un nodo nuevo o existente se conecte o reconecte a un nodo que cuenta ya con un gran número de vecinos que (re)conectarse a un nodo con baja interconexión. Siguiendo un modelo de conectividad preferente lineal, donde $\Pi(d_i)$ denota la probabilidad de que el nuevo nodo v escoja al nodo i, $\Pi(d_i)=\frac{d_i}{\sum_{j\in C} d_j}$ donde d_i es el número de conexiones del nodo destino y C, el conjunto de nodos vecinos candidatos.

3. **Reconexión**. Permite cierta flexibilidad en la formación de redes, seleccionando uniformemente m nodos, removiendo uno de sus enlaces y conectando el nodo a otro nodo existente de acuerdo al modelo de conectividad preferencial lineal, donde con cierta probabilidad un enlace será desconectado de un nodo de bajo grado y reconectado a otro de alto grado. Según [CCGJSW], este evento puede ser visto en Internet como un usuario que cambia a otro proveedor de acceso a la red.

   Cabe notar que la operación de reconexión pudiera desconectar la gráfica y la gráfica generada podría contener *loops* al nodo mismo y enlaces duplicados [BT].

El proceso comienza con alguna estructura de gráfica inicial, digamos m_0 nodos aislados, y a medida que el modelo va evolucionando, alguno de los siguientes eventos locales tiene alguna probabilidad de ocurrir. El primer evento consiste en añadir un nuevo vértice, junto con m ($m = m_0$) nuevos enlaces que conectarán al nodo con la gráfica existente asumiendo el compromiso de realizar-conseguir-obtener una conectividad preferente. El segundo evento consiste en añadir m nuevos enlaces, independientes al nuevo nodo añadido en el primer evento, seleccionando aleatoriamente m vértices preexistentes con una probabilidad uniforme como nodos-origen y conectándolos a m preexistentes nodos-destino siguiendo la regla de conectividad preferente. El tercer evento consiste en reconectar m enlaces existentes seleccionando aleatoriamente m vértices y removiendo un enlace determinado para cada uno de ellos, reconectándolo a un nodo diferente siguiendo la propiedad de conectividad preferente [CCGJSW].

Las gráficas resultantes, desarrolladas según este algoritmo, logran un estado estable, donde, por ejemplo la distribución del grado de los nodos permanece inalterable a lo largo del tiempo y sigue una ley de potencia con un exponente que es función de los parámetros iniciales [CCGJSW].

Palmer-Steffan (2000) [PaS]. Con el fin de generar topologías que cumplan con las leyes de potencia y definan valores reales para el peso de los enlaces, Palmer y

Steffan propusieron un algoritmo que opera de manera recursiva sobre la matriz de adyacencia.

La matriz de adyacencia A de una gráfica de dos dimensiones y de N nodos, es una matriz N x N, cuyo valor $a(i, j) = 1$ si existe un enlace(i, j) ó 0 en cualquier otro caso. Este generador produce una matriz simétrica con diagonal cero, con el fin de producir enlaces bi-direccionales y evitar enlaces de un nodo consigo mismo. Esto significa tener que trabajar únicamente con el triángulo superior de la matriz y copiar el resto de las celdas.

La idea básica es subdividir recursivamente esta matriz en cuatro cuadrantes del mismo tamaño (ver figura 3.9) y distribuir los enlaces dentro de las particiones con una función de probabilidad que es una generalización de una distribución 80-20. Una distribución 80-20 divide el espacio en dos mitades y coloca el 80% de los valores en la mitad izquierda y el 20% en la mitad derecha.

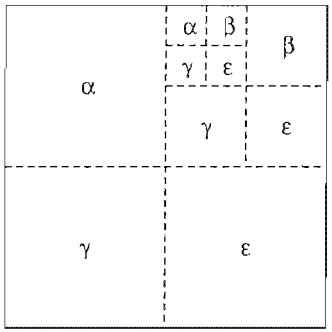


Figura 3.9. Modelo de la matriz recursiva.

Empieza con una matriz de adyacencia vacía y se va colocando uno a uno cada enlace en la matriz hasta llegar a M (en la tabla 3.1, se encuentran definidos los parámetros de este algoritmo). Cada enlace escoge una de las cuatro particiones con probabilidades P_α, $P_{\beta,\gamma}$ y P_ε respectivamente, donde $\alpha + \beta + \gamma + \varepsilon = 1$. La partición escogida es de nuevo subdividida en cuatro particiones más pequeñas y el procedimiento se repite hasta alcanzar una celda simple (=partición de 1x1). Esta será la celda de la matriz de adyacencia ocupada por un enlace.

$$P_\alpha \propto \frac{\alpha}{N_1} \qquad P_{\beta,\gamma} \propto \frac{\beta+\gamma}{N_2} \qquad P_\varepsilon \propto \frac{\varepsilon}{N_1}$$

Para cumplir con la distribución 80-20, $\alpha = \varepsilon = 0.4$, $\beta = \gamma = 0.2$ y donde:

$$N_1 = \left(\frac{N}{2}\right)^2 - \frac{N}{2} \quad , \quad N_2 = \left(\frac{N}{2}\right)^2$$

Por ejemplo, para una matriz simétrica de 4x4, existen 12 valores potenciales no-cero, pero solamente 2 de ellos, están en el cuadrante superior-izquierdo ($N_1 = 2$ y $N_2 = 4$).

Aquí aparece un punto sutil, se pueden tener enlaces duplicados (los enlaces que caen en la misma celda de la matriz de adyacencia) pero solo se puede mantener uno de ellos, así que esta selección múltiple incrementará el peso de este enlace. Aquí el usuario decidirá como relacionar el peso de los enlaces con las cualidades de la red; por ejemplo, se puede asumir que el peso corresponde a alguna medida de el tráfico que pasa a través de un enlace y definir los enlaces de más peso como los de mayor latencia.

Símbolo	Significado
N	Número de nodos en la gráfica
M	Número de enlaces en la gráfica
$\alpha, \beta, \gamma, \varepsilon$	Porcentaje de enlaces que se quieren colocar en cada sub-arreglo recursivo $\alpha+\beta+\gamma+\varepsilon=1$
N_1	Número de valores $d(i,j)$ potencialmente no-cero en el cuadrante superior-izq/inferior-der de la matriz A.
N_2	Número de valores $d(i,j)$ potencialmente no-cero en el cuadrante superior-der/inferior-izq.

Tabla 3.1. Parámetros del generador recursivo.

Intuitivamente, esta técnica esta generando "comunidades" en la gráfica[CZF]:

- Las particiones α y ε representan grupos separados de nodos que corresponden a comunidades (por ejemplo, personas entusiastas por el *soccer* y automóviles).
- Las particiones β y γ son los enlaces que relacionan estos dos grupos (enlaces que pudieran denotar a amigos con intereses separados).
- La naturaleza recursiva de las particiones significa que se tendrán automáticamente sub-comunidades dentro de las comunidades existentes (por ejemplo, personas entusiastas por autos y motocicletas dentro del grupo de automóviles).

La evaluación de este algoritmo de acuerdo a las cuatro leyes de potencia de Barabasi, muestran que el trazo que exhibe la primera ley (grado *vs.* orden) se asemeja a una curva 80-20 típica, para las otras 3 métricas, exhibe una excelente relación de ley de potencia, con un coeficiente de correlación lineal = 0.92 [PaS].

Magoni-Pansiot (2002) [MP]. Es un algoritmo que extrae una sub-gráfica de un mapa real de Internet. Se hace un muestreo aleatorio de los nodos con el fin de producir un árbol y asegurar la conectividad de la futura gráfica, se le añaden enlaces redundantes para generar una gráfica que tenga las propiedades topológicas apropiadas; como son las leyes de potencia 1 y 2, y otras propiedades comunes que se encontraron en mapas reales de Internet a nivel ruteadores y que se definirán más adelante. Con lo que respecta a las otras dos leyes de potencia encontradas por Faloutsos et al., la ley de potencia 4 (exponente de eigen-valores) no se tiene ningún interés en cumplirla dado que no se considera un indicador primordial, ya que los modelos Waxman y Transit-Stub cumplen con ella y ninguna de las dos se considera como un modelo que genere de manera precisa topologías Internet. La ley de potencia 3 no ha demostrado ser una ley ya que ésta se ha derivado de una aproximación [MP].

De las definiciones presentadas por Magoni et al [MagP] (ver la figura 3.10), un nodo que pertenezca a un ciclo o que se encuentre en la ruta que conecta a dos ciclos es llamado un nodo de malla. La malla de una gráfica es el conjunto de todos los nodos en ella. Lo interesante es examinar el tamaño de una malla y obtener resultados acerca de la conectividad de ésta, como son el número de puntos de corte (por ejemplo, puentes) y el tamaño del componente bi-conectado más grande. El estudio de la malla brinda información acerca del grado de confiabilidad-seguridad *vs.* fallas de conexión y la posibilidad de balancear cargas usando rutas alternas.

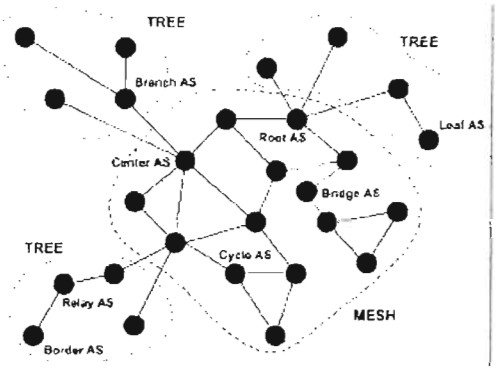


Figura 3.10. Diferentes tipos de Sistemas-Autónomos (tomada de [MagP]

La floresta de una topología, es simplemente el conjunto de nodos que no pertenecen a la malla y estos nodos están localizados en árboles. Los árboles están conectados a la malla por nodos especiales llamados raíces y se considera que estos pertenecen a la malla. Las propiedades concernientes a los árboles son interesantes para estudiar la conectividad y confiabilidad de una red, porque cada nodo que pertenezca a un árbol (excepto los nodos hoja) es un punto de corte y si éste falla puede provocar que la grafica se desconecte.

En una topología Internet a nivel ruteador se tiene una malla cuyo tamaño representa el 33% de el tamaño total de Internet, esto significa que un nodo de cada tres pertenece a la malla. Con respecto a las propiedades de conectividad de la malla de Internet, se observó que, el número de puntos de corte que existen en la malla es igual al 3.7% de el número total de los nodos dentro de ella. A pesar de que estos nodos son raros, es interesante ver como dividen la malla. Para examinar este punto, se calcula el tamaño de los componentes bi-conectados más grandes dentro de ella. En Internet, el componente más grande contiene el 87% de los nodos de la malla, esto significa que a pesar de que la malla de Internet contiene 3.7% de puntos de corte, estos nodos solo fraccionan a una pequeña parte de la malla (13%), ya que la mayor parte se encuentra bi-conectada. La proporción de árboles vs. el tamaño de Internet es de 11.7% y la mayoría de ellos tienen una profundidad de uno (son nodos que están directamente conectados a las raíces), lo que significa que más de un nodo dentro de la malla de cada tres es un nodo raíz. Para un mapa de referencia de Internet a nivel ruteadores, se ha encontrado que el grado promedio de los nodos es igual a 3.15 [MaPa].

Magoni et al. [MagP] definieron, de la misma manera que hicieron Faloutsos et al., otras dos leyes de potencia (referentes a los árboles) en mapas reales de Internet a nivel interdominio, sin embargo, también han corroborado la presencia de las mismas en mapas reales a nivel ruteador [MaP], de tal forma que probar la presencia de ambas en las gráficas extraídas, es significativo para asegurar la similitud de éstas con Internet.

Ley de potencia 5 (exponente de orden del árbol T). El tamaño s_t de un árbol t es proporcional al rango r_t del árbol elevado a una constante T: $s_t \propto r_t^{T}$.

Esta ley representa la relación que establece un árbol y su tamaño. El tamaño de un árbol se define por la suma de los vértices que le componen, incluyendo su raíz. Los árboles se ordenan en orden decreciente de acuerdo al tamaño de árbol, s_t, y el índice del árbol en esta secuencia se define como el rango del árbol, r_t.

Ley de potencia 6 (exponente de tamaño de árbol S). La frecuencia f_s de un tamaño de árbol, s, es proporcional al tamaño del árbol elevado a una constante S: $f_s \propto s^{S}$.

Esta ley estudia la distribución de el tamaño de los árboles. La frecuencia de un tamaño de árbol, f_s es el número de árboles que tienen un tamaño s. Las figuras 3.11 y 3.12 muestran dos distribuciones concernientes a los árboles. La primera es la frecuencia de los tamaños de árbol, solamente se muestra el inicio de esta distribución. La segunda es la frecuencia de las profundidades de árbol. Ambos muestran resultados a nivel interdominio. Lo que se puede ver dada la distribución de la figura 3.12, es que ningún árbol tiene realmente arborescencia. El 90% de los árboles están compuestos simplemente de hojas conectadas directamente a su raíz.

Las gráficas extraídas por este método cumplen con los valores de las propiedades topológicas (para árboles y mallas) y las leyes de potencia 1, 2 y 5 para todos los tamaños de topologías generadas, la ley de potencia 6 solo se cumple para tamaños por encima de los 1000 (2000) nodos. Las gráficas extraídas son dos ordenes de magnitud más pequeñas que sus mapas originales.

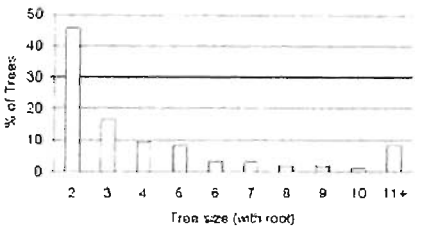


Figura 3.11. Distribución por tamaño de árbol. Tomada de [MagP]

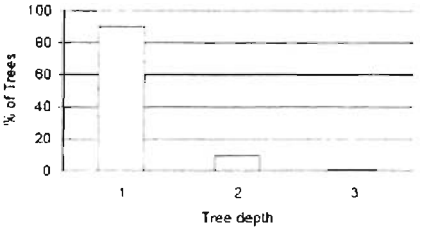


Figura 3.12. Distribución por profundidad de árbol. Tomada de [MagP]

Generadores de Topología

El objetivo de los generadores de topología, no es el producir replicas exactas de Internet, pero sí el de elaborar gráficas cuyas propiedades sean similares a las gráficas de Internet [TGJSW]. En este trabajo se usarán tres generadores de topologías de red, que producen topologías de acuerdo a los modelos de red presentados anteriormente.

BRITE[8]. Medina et al. [MMB] desarrollaron un generador de topologías, llamado BRITE, con el cual es posible experimentar y probar con diferentes combinaciones las posibles causas que generan una topología Internet que cumpla con las leyes de potencia.

BRITE toma algunas técnicas de modelado para investigar las topologías de red. Es así que puede generar topologías Waxman y también usar el modelo de Barabasi-Albert para construir redes que cumplan con las leyes de potencia.

Como posibles causas generadoras de una topología Internet se toman en cuenta los fenómenos de:

[8] Las siglas/el acrónimo provienen de: Boston university Representative Internet Topology gEnerator. Este generador de topologías se encuentra disponible en http://www.cs.bu.edu/fac/matta/software.html

1. **Crecimiento progresivo** [BA]. Se refiere a las redes "abiertas" que se forman por la continua añadidura de nuevos nodos y por consiguiente, el incremento gradual en el tamaño de la red.

2. **Conectividad preferente** [BA]. Expresa la tendencia que muestra un nodo nuevo al integrarse a una red y conectarse a los nodos más populares o a los densamente conectados.

3. **Distribución de los nodos** [MMB]. Indica la manera en la cual los nodos de una red son colocados en el espacio. A diferencia de los modelos aleatorios, [MMB] supone que la topología de Internet muestra un alto grado de agrupamiento de sus nodos. Donde aquellas áreas fuertemente pobladas tienen nodos densamente conectados, mientras que el resto de los nodos están escasamente enlazados.

4. **Localidad de los enlaces** [MMB]. Observa la tendencia que presenta un nuevo nodo de conectarse a nodos existentes que estén más cercanos a él en distancia.

Observando la relación que guardan las leyes de potencia con sus posibles causas, [MMB] encontró que: con respecto al exponente de orden R, la conectividad preferente parece ser una condición necesaria para obtener la ley de potencia 1, mientras que el crecimiento progresivo juega un papel mucho menos importante en su generación. Para el exponente de enlaces de salida de un nodo, ?, la conectividad preferente y el crecimiento progresivo son dos fenómenos necesarios para que exista la ley de potencia 2. La distribución de los nodos (ver la figura 3.13) no juega un papel preponderante en esta ley, ya que los resultados usando la colocación de los nodos según la distribución de Pareto es consistente con los resultados de una distribución aleatoria de los nodos. Para obtener un exponente de saltos H que parezca al de una topología Internet basta con generar una con la característica de crecimiento progresivo. Las topologías de BRITE con conectividad preferente cumplen con la generación de topologías Internet con el exponente de eigen-valores E.

Además de las leyes de potencia, se observó que las topologías generadas con las características de conectividad preferente y crecimiento progresivo, tienen una longitud de ruta promedio que se aproxima a los 12 *hops* y un diámetro de 30 *hops* a medida que el tamaño de red incrementa. Tal como Paxson [Pa] constató.

Como conclusión general, los fenómenos de conectividad preferente y crecimiento progresivo son los contribuyentes clave para obtener una topología tipo Internet. (se encontraron como principales causas de todas las leyes de potencia).

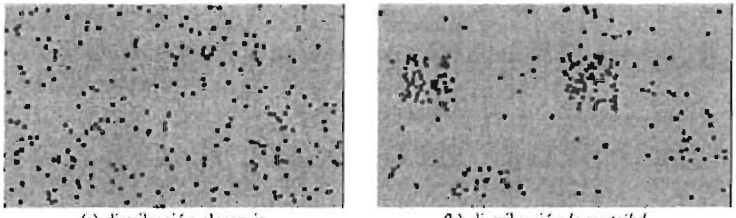
La tabla 3.2 enlista los parámetros de BRITE y a continuación se describirá cada uno de ellos.

El plano Los nodos de la topología generada son distribuidos en un plano dividido en una cuadrícula de *HS* x *HS*. Cada uno de estos cuadrados de dimensión mayor es a su vez subdivididos en cuadrados más pequeños de *LS* x *LS* Cada uno de estos cuadrados de dimensión-menor podrá tener asignado a lo más un nodo.

Parámetro	Significado	Valores
HS	Dimensión de la cuadrícula	entero > 1
LS	Tamaño del lado de los cuadrados más pequeños	entero = 1
NP	Distribución de los nodos en el plano	0:aleatorio, 1:distribución *heavy-tailed*
M	Número de enlaces agregados por nodo nuevo	entero = 1
PC	Conectividad preferente	0:ninguna, 1:única, 2:ambos
IG	Crecimiento progresivo	0:inactivo, 1:activo

Tabla 3.2. Parámetros de BRITE

Distribución de los nodos. Una distribución *aleatoria* de los nodos en el plano se logra seleccionando aleatoriamente cuadrados de dimensión-menor y colocando ahí, evitando colisiones, un nodo. Para lograr una distribución *heavy-tailed* de los nodos, a cada uno de los cuadrados de dimensión-mayor, el generador toma n número de nodos para asignarlos a ese cuadrado de acuerdo a la distribución de Pareto: $f(n) = \frac{ak^a n^{-a-1}}{1-(k/P)^a}$ y un nodo es colocado aleatoriamente en uno de los cuadrados de dimensión-menor, evitando colisiones. Esta distribución consigue el agrupamiento de nodos (ver la figura 3.13).



(a) distribución aleatoria (b) distribución *heavy-tailed*

Figura 3.13. Distribución de los nodos en el plano (tomado de [MMB].

Número de enlaces por cada nuevo nodo El parámetro m controla el número de nodos vecinos a los cuales un nuevo nodo se conecta cuando se une a la red, en otras palabras, el número de nuevos enlaces que serán añadidos a la topología. Mayor que sea el valor de m más densa será la topología generada. El conjunto de nodos por los

cuales un vecino es seleccionado por un nodo nuevo se referirá como el *conjunto de vecinos candidatos*.

Crecimiento progresivo. Este parámetro puede tomar uno de los dos valores:

- *inactivo* coloca todos los nodos de una vez en el plano antes de añadir cualquier enlace. En cada paso, un nodo es seleccionado aleatoriamente y m enlaces son usados para conectarlo con m vecinos candidatos de *todos* los nodos restantes.

- *activo* coloca los nodos en el plano gradualmente a medida que cada uno se vaya uniendo a la red. En este caso, un nuevo nodo considera como vecinos candidatos solamente a aquellos nodos que ya estén unidos previamente a la red.

Inicialmente, antes de operar en modo *inactivo* o *activo*, el generador produce un pequeño *backbone* de m_0 nodos conectados aleatoriamente. Los nodos restantes son conectados después.

Conectividad preferente. Este parámetro controla la activación o desactivación de dos fenómenos: la conectividad preferente y la localidad de las conexiones. Hay tres posibles valores para este parámetro:

- *ninguna* indica que la conectividad preferente esta desactivada. En este caso un nodo nuevo se conectará a un nodo vecino candidato usando la función de probabilidad de Waxman [W]. Este proceso se repite hasta conectar el nuevo nodo a m nodos.

- *única* significa que la conectividad preferente esta activada. En este caso, un nodo nuevo v se conectará a un nodo vecino candidato i con la siguiente probabilidad: $\frac{d_i}{\sum_{j \in C} d_j}$, donde d_i es el número presente de enlaces incidentes del nodo i y C es el conjunto de nodos vecinos candidatos. Este proceso implica que un nodo nuevo que se una a la red seleccionará con una alta probabilidad aquellos nodos densamente conectados. Esto se repite hasta conectar v a m nodos.

- *ambos* combina la preferencia y la colocación de los enlaces. En este caso, para un nodo nuevo v, se calculará para cada nodo vecino candidato i una probabilidad de Waxman w_i (lo que da preferencia por los nodos más cercanos). La probabilidad final de conectarse a un nodo i se calcula como: $\frac{w_i d_i}{\sum_{j \in C} w_i d_j}$. Este proceso es repetido hasta conectar v a m nodos.

REC[9]. Este generador fue desarrollado por Palmer et al [PaS] para obtener las topologías que generan su algoritmo (matriz recursiva).

Una vez compilado el archivo gen.c, el programa corre con los siguientes parámetros:

./gen [-pgm fname | -pgm2 fname] logn m alpha beta gamma epsilon

con lo cual se generará aleatoreamente una gráfica con 2^{logn} nodos con m enlaces distintos y cada uno con su respectivo peso. Los parámetros de la probabilidad recursiva son:

$alpha + beta + gamma + epsilon = 1$

Los dos argumentos opcionales:

-pgm fname
-pgm2 fname

generaran el archivo nombrado en un formato pgm. Utilizando la primera opción, se visualizaran las probabilidades de la matriz de adyacencia (entre más oscuro, menos probable) y la segunda forma visualizará la actual matriz de adyacencia creada (entre más oscuro menos enlaces).

El formato de salida es una representación ASCII de la gráfica. La primera línea contiene únicamente un entero, el número de nodos en la gráfica. Cada una de las líneas subsecuentes contiene tres enteros: u v w, los cuales representan un enlace del nodo u al nodo v con peso w.

NEM[10]. Es un software desarrollado por Magoni [M], que puede realizar tres procesos básicos relacionados al análisis y modelado de redes.

1. **Convertir archivos** de redes a otros formatos (operador *Cn*)
2. **Analizar la topología** de redes (operador *A*)
3. **Generar redes** (archivo de entrada **.specif*)

NEM no es un software interactivo, solo trabaja con archivos y debe ser llamado desde un archivo *batch* que tiene como extensión **.process*. Cada línea en el archivo de proceso le da una orden a NEM, cada línea en blanco es ignorada y cualquier línea que empiece

[9] El código fuente de este generador puede encontrarse en http://www-2.cs.cmu.edu/~crpalmer/pubs.html/

[10] Las siglas/el acrónimo provienen de: Network Manipulator. Este generador de topologías se encuentra disponible en http://www-r2.u-strasbg.fr/nem/

con // es un comentario y también será ignorado. Cualquier otra línea, deberá tener el siguiente formato:

<nombre_de_archivo_de_red> | <nombre_de_archivo_specif> [operador]...

donde <> significa un parámetro obligatorio, [] significa un parámetro opcional y | significa XOR. El extracto de un archivo **.process* típico es:

// comentarios
tsnet.alt C1
misparámetros.specif C5
mired.nm A

El primer argumento debe contener un nombre de archivo, con el cual NEM procederá de acuerdo a su extensión. Si el archivo corresponde al nombre de una red con alguna de las extensiones de la tabla 3.3, el archivo se cargará en el manipulador, ver la figura 3.14. Si el archivo corresponde a uno de especificación (**.specif*), NEM generará una nueva red utilizando los parámetros definidos en el.

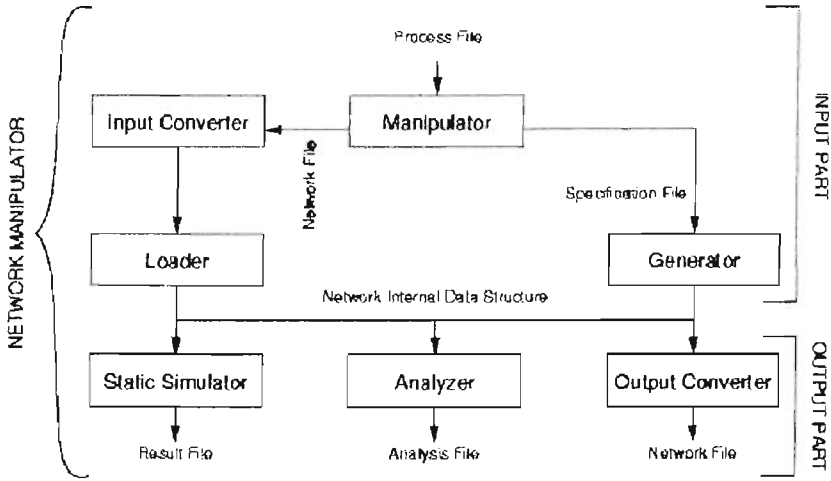


Figura 3.14. Arquitectura de NEM, tomada de [M]

Al nombre de archivo le siguen una serie de operadores, cada uno separado por un espacio en blanco, los cuales determinarán lo que deberá de hacerse con los datos que han sido generados o cargados en el manipulador. Un operador esta representado por una letra mayúscula *A*, *B*, *C*, O, *P* o *Z*.

Convertir archivos de redes a otros formatos (operador *Cn*). En el archivo que define al proceso (**.process*), poner simplemente una línea con el nombre del archivo de red a convertir seguido del operador *Cn*, donde *n* es un entero que indica el formato de salida. La tabla 3.3 muestra la relación entre el número y el formato, aquí se especificarán los códigos:

// lsiit_nm_basic = 1
// lsiit_nm_regular = 2
// lsiit_nm_advanced = 3
// lbnl_ns_basic = 4
// opnet_modeler = 5
// gt_itm_alternate = 6
// sgi_h3 = 40

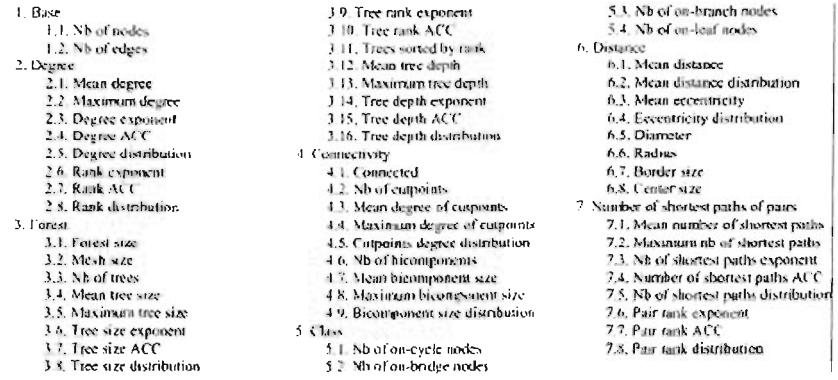
Afiliación	Formato del archivo	Extensión	Entrada	Salida	Número
LSIIT	nem	*.nm	si	si	1, 2, 3
LBNL	ns-2	*.tcl	no	si	4
OPNET Inc.	OPNET Modeler	*.xml	no	si	5
GT	ITM (transit-stub)	*.alt	si	si	6
SGI	H3Viewer	*.h3	no	si	40
ASCOM NEXION	Tiers	*.tiers	si	no	
BU	BRITE	*.brite	si	no	
UM	Inet2.x	*.inet	si	no	
ISI	Mercator	*.mercator	si	no	
NLAR	ASmap	*.as_map	si	no	

Tabla 3.3. Formatos de los archivos de red

El código que produce los formatos de red de salida están localizados en *network.cpp* en la función: *void network::convert()*. Se pueden incluso agregar otros formatos.

Analizar la topología de una red (operador *A*). En el archivo que define al proceso (**.process*) poner una línea con el nombre del archivo de red a analizar seguido del operador *A*. Ver en la tabla 3.3 los formatos de entrada permitidos.

El análisis produce un archivo que tiene el mismo prefijo que el nombre del archivo de red y tiene la extensión **.analysis*. Al análisis le concierne solamente la topología de la red (vista como una gráfica) y los valores entregados son:



El contenido del archivo de análisis puede ser modificado, comentando algunas partes de el código localizado en *graph_theory.cpp* en la función: *void graph::output(std::ofstream &)*.

Generar redes (archivo de entrada **.specif*). Primero se debe crear un archivo de especificación, el cual es un archivo ASCII que contiene la definición de los parámetros necesarios para crear una red. Una línea en este archivo debe tener el siguiente formato:

<clave_del_parámetro> <valor_del_parámetro>

Cada parámetro está definido por una cadena de caracteres ASCII (la clave), que de ser cambiada no será reconocida por NEM, seguido de un valor que puede ser un entero, un flotante o una cadena. El valor por omisión, para cada parámetro, esta definido en el código y será usado si el parámetro no se encuentra definido-determinado en el archivo de especificación. El orden en el cual los parámetros estén definidos no importa. Un extracto de este archivo es:

// número de nodos en la gráfica generada (por omisión: 1000)
nodes_nb 2000
// nivel de topología de la gráfica generada (por omisión: router_level)
network_level router_level

El usuario puede escoger entre los métodos de generación dados en la tabla 3.4. Algunos parámetros como por ejemplo: *nodes_nb*, son comunes para todos los métodos, mientras que otros solamente tienen sentido y funcionan para un modelo específico. Para aprender a usar estos parámetros de manera correcta, favor de referirse a sus respectivos artículos.

En el archivo que define al proceso (**.process*) poner una línea con el nombre del archivo de especificación (**.specif*) seguido del operador *Cn*, donde *n* es un entero que indica el formato de salida. La tabla 3.3 muestra la relación entre el número y el formato. Si no se usara el operador de conversión, el archivo de red sería generado

internamente en el manipulador pero no se produciría ninguna salida, por tanto, la red generada se perdería.

Nombre del modelo	Autor(es)	Valor del parámetro	Referencia
Map sampling	Magoni, Pansiot	*magoni_pansiot_sampling*	[MP]
Waxman modificado	Waxman, Magoni	*waxman_modified*	[W]
Scale-free extendido	Albert, Barabàsi	*albert_barabasi_ext*	[AB]
PLOD	Palmer, Steffan	*palmer_steffan_plod*	[PaS]
Modelo A	Aiello, Chung, Lu	*aiello_chung_lu_model_a*	[ACL]

Tabla 3.4. Modelos de generación de topologías

Capítulo 4

Sistemas *peer-to-peer*, un caso de estudio: Gnutella

Esta red emplea la infraestructura que ofrece Internet y resuelve sus servicios (búsqueda de archivos y localización de peers) empleando broadcasting en la capa de aplicación.

La importancia y el impacto de reducir el costo en la transmisión de mensajes de *broadcast* en Internet puede ser evaluado de mejor manera en el contexto de una aplicación. En las siguientes secciones de este capítulo se hará una revisión de los protocolos de comunicación que han sido implementados en la red Gnutella y de algunas alternativas, mostrando que la solución actual limita su escalabilidad[11] y costo. Aquí se propondrá una alternativa de *broadcasting* a la sugerida en [PS] y en el capítulo 6 se hará una comparación de los resultados obtenidos cuando se usa el algoritmo ALEM propuesto en esta tesis.

Los sistemas *peer-to-peer* (P2P) empezaron a ser ampliamente conocidos bajo el consorcio Napster; dentro de ésta aplicación, el concepto de redes P2P se empleó para compartir archivos de música (MP3). Aunque el concepto no era nada nuevo, la popularidad que desencadenó este tipo de aplicación creó una fuerte actividad de investigación en torno a su arquitectura. Estos sistemas dependen típicamente de la participación voluntaria de los *peers* para contribuir con sus recursos en la conformación de la red de comunicación. La incorporación de cualquier usuario en un sistema P2P es de manera *ad-hoc* y dinámica, así que el carácter de permanencia temporal y carente de una administración central plantea el reto de conformar un mecanismo y una arquitectura que organice los *peers* de tal manera que puedan cooperar para brindar un servicio útil a la comunidad de usuarios. Por ejemplo, en una aplicación que comparte archivos, el reto es lograr organizar los *peers* para generar un índice global distribuido de manera que cada *peer* pueda rápida y eficientemente localizar cualquier contenido dentro del sistema. Adicionalmente, el sistema deberá tomar en cuenta la

[11] el término escalabilidad se refiere al incremento lineal de los recursos de la red (espacio de almacenamiento y disponibilidad de archivos) al adherir nuevos nodos, sin alterar su funcionamiento (tiempo de respuesta) debido al impacto del crecimiento y al inminente aumento en el tráfico por búsquedas.

conveniencia de delegarle a un *peer* la responsabilidad para realizar alguna tarea específica, ya que existe una significante cantidad de heterogeneidad en los recursos de los participantes en estos sistemas.

En este trabajo, nos interesamos en los sistemas *peer-to-peer* (P2P) que no tienen un control centralizado o una organización jerárquica, donde el software de aplicación que se ejecuta en cada nodo es equivalente en funcionalidad y donde la topología de la red no se genera en relación a la localización de los archivos dentro del sistema. Son el tipo de redes, completamente descentralizadas y no estructuradas, que según [BKKMS] tienen el potencial de cambiar significativamente los sistemas distribuidos de gran escala que serán construidos en el futuro. El reto principal en estos sistemas es el diseñar e implementar un sistema distribuido robusto compuesto de computadoras caseras en dominios administrativos que no tengan relación entre si.

A pesar del auge de estas redes, el problema concerniente a los términos relacionados a estos sistemas es confuso y multifacético, Schollmeier [Sch] ha tratado de concluir el tema definiendo:

> *Un sistema de red distribuido podría llamarse P2P, si los participantes comparten una parte de sus propios recursos de hardware (capacidad de almacenamiento, de procesamiento, de acceso a la red, a impresoras, etc.), para poder brindar el Servicio y los contenidos ofrecidos por el sistema (por ejemplo, compartir archivos) Estos son accesibles directamente por otros peers sin la necesidad de pasar por entidades intermedias. Los participantes de esta red serán tanto proveedores como demandantes de los recursos, trátese de Servicio o contenido.*

En este sentido se llegó a observar, que los contenidos publicados a través de Napster, excedían los 7 TB de almacenamiento en un solo día sin requerir de una planificación centralizada o una gran inversión en hardware, ancho de banda o espacio de almacenamiento. El diseño de los sistemas P2P tenia que incluir el control de una enorme cantidad de recursos [RW].

La descentralización de estos sistemas es particularmente interesante, ya sea para evitar puntos de falla o congestionamiento. Cuando un sistema P2P llega a ser completamente descentralizado, no existe un nodo que coordine centralmente las actividades o una base de datos que guarde centralmente información global acerca del sistema, por lo tanto, los nodos tienen que organizarse entre sí, con base en cualquier información local disponible e interactuando con los nodos vecinos [AH]. Las barreras para empezar y hacer crecer un sistema de este tipo son escasas, ya que usualmente no requieren ningún acuerdo especial, financiero o administrativo [BKKMS].

Aunado a los principios que identifican a los sistemas *peer-to-peer*, es una práctica común el dividir estas redes en dos sub-definiciones que distingan a los sistemas *peer-to-peer* que carecen de alguna entidad central de aquellos que la contienen.

Un sistema de red distribuido es clasificado como un sistema peer-to-peer "híbrido", si es en primera instancia un sistema peer-to-peer y en segunda, si una entidad central es necesaria para brindar parte de los servicios de red ofrecidos por el sistema.

Una diferencia significativa de los sistemas *peer-to-peer* "híbridos" con respecto a los sistemas Cliente/Servidor es que los clientes no comparten ninguno de sus recursos y la entidad central brinda todos los servicios y contenidos que se ofrecen en estos sistemas [Sch].

Un sistema de red distribuido es considerado como un sistema peer-to-peer "puro" si es en primera instancia un sistema peer-to-peer y en segunda si cualquier Entidad Terminal escogida arbitrariamente puede ser removida de la red sin que el sistema sufra de alguna perdida del servicio.

Gnutella, como un sistema de este tipo, un sistema P2P "puro", implementa sus servicios de búsqueda de archivos y localización de *peers* a nivel aplicación vía *broadcasting*. Para lograr esto, los mensajes son enrutados a través de la red Gnutella usando *flooding*. Cada nodo actúa como un ruteador y transmisor de los mensajes enviados por los otros nodos, lo que inmediatamente hace surgir la duda en cuanto a costo y escalabilidad; algunos nodos pueden ser inundados con mensajes para retransmitir y podrían no tener suficientes recursos requeridos para llevar a cabo esta tarea (memoria, ciclos de CPU o ancho de banda). Este problema se ha observado ya en el sistema real de Gnutella, donde el hecho de que los nodos no puedan manejar la proporción de mensajes a retransmitir ha propiciado la fragmentación de la red [PS]. En este sentido, los sistemas *peer-to-peer* presentan un problema fundamental porque la proporción de comunicación para efectuar peticiones de archivos entre los nodos es muy alta. Para obtener un desarrollo exitoso de las redes P2P descentralizadas, es necesario que se planteen nuevos métodos para realizar *broadcasting* a nivel aplicación, que se enfoquen en resolver los problemas de costo y escalabilidad.

Gnutella

Gnutella es un sistema descentralizado que comparte archivos, cuyos participantes forman una red virtual que se comunican en un estilo *peer-to-peer* vía el protocolo Gnutella [GPS], el cual es un protocolo simple para búsqueda de archivos distribuidos. Gnutella fue desarrollado a principios del 2000 por Justin Frankel de la compañía Nullsoft (una subsidiaria de AOL y creadora del reproductor de MP3 WinAMP). El desarrollo de Gnutella fue interrumpido poco después de que sus resultados se hicieran públicos y el protocolo actual fue interpretado según ingeniería inversa usando el código obtenido en el sitio web de Nullsoft justo antes de que se clausurara. Hoy existen numerosas aplicaciones (referidas como clientes Gnutella) que emplean el protocolo Gnutella con sus propias especificaciones a fin de permitir a sus usuarios acceder a esta red [I].

Como Gnutella es un sistema no estructurado y los archivos (o los apuntadores a ellos) no están colocados en sitios determinados de manera premeditada, entonces, para compartirlos, un usuario (digamos el nodo A) deberá iniciar con una computadora en red que corra alguno de los clientes[12] Gnutella. Este nodo actuará como servidor y cliente, por tanto es generalmente nombrado "servent" (de SERVidor y cliENTe). El nodo A entonces, se conecta a otra computadora en red (nodo B) que corre bajo el protocolo Gnutella para anunciar su existencia. Queda afuera de las especificaciones del protocolo, como es que se obtiene la dirección del nodo B. En su turno, el nodo B le avisará a todos sus nodos vecinos (nodos C, D y F) que A existe. Este patrón continuará recursivamente con cada nuevo nivel de nodos, anunciando a los vecinos que el nodo A esta participando. Una vez que el nodo A ha anunciado su existencia al resto de la red, el usuario de este nodo podrá preguntar por el contenido de los datos que se comparten a través de la red [I].

El *broadcasting* que se lleva a cabo durante la divulgación de la existencia de algún nodo, esta basado en un *flooding* modificado y termina cuando el TTL (*time-to-live*) del paquete de información expire, lo que significa que en cada nivel, el contador del TTL disminuirá en uno desde algún valor inicial hasta que llegue a cero, en este punto, el *broadcasting* se suspenderá. Para prevenir que los usuarios ajusten el valor TTL inicial muy alto, la mayoría de los servents Gnutella rechazarán los paquetes con un valor excesivamente elevado. Desde la perspectiva de los usuarios, maximizar la posibilidad de encontrar el archivo solicitado significará usar un valor TTL tan alto como sea posible, por eso se establece un punto de acuerdo para esta red. Un valor TTL bajo, minimizará el uso de los recursos de la red, un valor alto, maximizará la calidad de servicio que la red brinda a los usuarios. Un valor optimo dependerá (entre otros factores) de la topología de la red, de las características de tráfico de un sitio en particular y de la hora del día en que se haya efectuado la petición[I].

Toda la comunicación Gnutella ocurre encima del protocolo TCP/IP. Una vez que la conexión TCP/IP entre dos servents se establece, el mensaje de conexión Gnutella "GNUTELLA CONNECT/<versión_del_protocolo>" puede ser enviado por uno de los clientes. El servent responderá a esta conexión con el mensaje "GNUTELLA OK", estableciendo con esto una conexión Gnutella válida entre estos dos servents. Cualquier otra respuesta al mensaje original enviado por el servent iniciador, puede ser tomada como el rechazo a establecer una comunicación. Después de que la conexión se establece, dos servents pueden comunicarse entre si intercambiando descriptores propios del protocolo Gnutella (ver la tabla 4.1). Este protocolo define las reglas por las cuales estos descriptores se deben intercambiar.

Dado que el protocolo Gnutella esta posicionado encima del *stack* de TCP/IP, las direcciones IP son usadas como parte de los descriptores para identificar a los nodos. Cada servent guardará un registro de los identificadores de los descriptores que ha

[12] A pesar de la gran variedad de estos clientes, entre los más populares se encuentran BearShare (http://www.bearshare.com) y ToadNode, disponibles únicamente para Windows y LimeWire (http://www.limewire.com), disponible para Windows, UNIX/Linux y Macintosh [I].

recibido recientemente[13], de tal manera que si un servent recibe el mismo tipo de descriptor con el mismo ID, éste es ignorado y no reenviado más [PS].

Descriptor	Descripción
Ping (broadcast)	Empleado para **descubrir activamente los *peers*** y para probar la red. Un servent que recibe este descriptor se espera que responda con uno o más descriptores *Pong*.
Pong	La respuesta a un *Ping*. Incluye la dirección de un servent conectado e información referente a los archivos que éste hace disponibles a la red. Con esta información un *peer* puede decidir conectarse a más servents de la misma manera que lo hizo con el primero.
Query (broadcast)	El mecanismo principal empleado para hacer **búsquedas en una red distribuida**. Un servent que recibe este descriptor responderá con un *QuerryHit* si en su conjunto de datos locales se encuentra el archivo solicitado.
Query Hit	La respuesta a un *Query*. Este descriptor brinda la suficiente información (dirección IP, número de puerto, tamaño del archivo y velocidad de su red) al destinatario como para poder adquirir los datos que solicitó el *Query* correspondiente.
Push	Permite hacer contribuciones de archivos a servents que se encuentran detrás de *firewalls*.

Tabla 4.1. Descriptores Gnutella.

La única información que un nodo Gnutella necesita almacenar, es la dirección de sus vecinos inmediatos, esto permite a Gnutella tratar con la naturaleza dinámica de una red P2P típica, donde los nodos frecuentemente se unen y dejan la red. De esta manera se evita el problema de la convergencia en la información de ruteo debida a la lentitud en la propagación [PS]. En Gnutella, las conexiones se establecen de manera *ad-hoc* (no hay restricción en el número de *peers* en la red de comunicación) y no estructurada, así que una fracción sustancial de ellas sufren de latencia-alta; el 20% de los *peers* tiene latencia[14] de al menos 280ms mientras que otro 20% de 70ms a lo más, el resto se encuentra más cercano al primer grupo [SGG].

Como los usuarios entran y salen del sistema, el protocolo Gnutella mantiene su red usando los descriptores *Ping* y *Pong*. Ocasionalmente los *peers* establecen nuevas conexiones con otros *peers* descubiertos a través de este par de descriptores. Esto hace que comúnmente se tengan diferentes redes Gnutella disjuntas coexistiendo simultáneamente en Internet [SGG].

Las reglas del protocolo Gnutella especifican también, que el mensaje en respuesta a un descriptor en particular debe ser enviado por la misma ruta que éste tomó, así, el descriptor *Pong* debe enviarse exclusivamente por la ruta por la cual llegó el descriptor *Ping*, lo mismo para *QueryHit* y *Query* y para los descriptores *Push* y *Query*

[13] Aunque las especificaciones del protocolo Gnutella no indican por cuanto tiempo el ID de los descriptores necesita ser conservado, se asume que este debe ser al menos un múltiplo de el tiempo de vida típico de un mensaje [PS].

[14] Latencia ess el tiempo que le toma a un mensaje recorrer una conexión en la red.

respectivamente. Los descriptores *Ping* y *Query* son los únicos mensajes enviados vía *broadcast* a todos los vecinos y el servent que se reconoce como el servent objetivo de algún descriptor en particular no lo reenviará más a la red. El proceso de *download*, es llevado a cabo usando el protocolo HTTP haciendo uso propiamente del *stack* TCP/IP y obviando la red Gnutella, con base en la información extraída del descriptor *QueryHit* [I]. Esto requiere que todos los servents Gnutella corran un pequeño servidor HTTP para responder a estas peticiones.

De modo tal que el tráfico de un sistema P2P puede ser clasificado de manera amplia, en dos categorías: señalización y transferencia de datos. El tráfico por señalización incluye entre otros, los mensajes que establecen la conexión TCP, las peticiones de búsqueda y sus respuestas. En comparación con la trasferencia de datos, los mensajes de petición *Query* y sus respuestas *QueryHit*, son mucho más pequeños en tamaño, del orden de varios cientos de bytes. Los contenidos compartidos en estos sistemas, como son los archivos de audio, video y software, tienden a ser mucho más grandes, por ejemplo, un archivo MP3 de audio de 4.8 MB tomará 5 minutos en una conexión de 128 Kbps, o un video MPEG-4 codificado a 500 Kbps, tardará 2 horas en bajarse. Por consiguiente, la transferencia actual de datos es el componente dominante del tráfico total en estos sistemas y tiene un impacto significativo en Internet. El termino "red P2P" es empleado típicamente para referirse a las conexiones *peer-to-peer* a nivel aplicación vía conexiones TCP punto-a-punto, que permanecen abiertas con el conjunto de vecinos, usada para la señalización entre *peers*, sin considerar la ruta empleada para bajar los datos [SW].

Ripeanu *et al.* [RFI] observaron que el 55% de el tráfico por señalización en la red Gnutella corresponde a los mensajes *Ping* y *Pong*, descriptores que sirven únicamente para mantener la conectividad de la red, mientras que el 36% es utilizado "realmente por el usuario", generando los mensajes *Query*, lo que se estima que genera aproximadamente 330 TB por mes. Este volumen de tráfico representa una fracción significante del tráfico total de Internet, lo que hace particularmente dependiente de el uso eficiente de los recursos el futuro crecimiento de la red Gnutella.

Problemas de Gnutella

Estos problemas tienen que ver más con el hecho según [KGZ], que Gnutella y el tipo de aplicaciones P2P que comparten recursos de bajo nivel, como música y video, requieren proveer únicamente mecanismos rudimentarios de búsqueda, basados más en fuerza bruta que en algoritmos sofisticados, porque no existe una convención para nombrar los archivos y por tanto muchas veces, el mismo contenido es guardado por diferentes nodos bajo varios nombres completamente distintos De cualquier manera, el hecho de que Gnutella haya sido liberado al público sin ser probado y evaluado apropiadamente hace que muchos de los conflictos significantes hagan parte integral del diseño mismo del protocolo [I]. En este trabajo nos enfocaremos en abordar dos de los más importantes:

- **Costo**. Reducir el tráfico innecesario en la red.

  Desde la perspectiva del usuario, Gnutella es un protocolo simple y hasta ahora eficaz: el porcentaje de éxitos en una búsqueda es razonablemente alto, es tolerante a fallas con respecto a las fallas de los servents y acepta bastante bien los cambios dinámicos de la "población de los *peers*". Sin embargo desde una perspectiva de red, esto se consigue a un costo muy alto en el consumo de ancho de banda: las peticiones de búsqueda de archivos (descriptores *Query*) son enviadas a toda la red usando *broadcast*, así que cada nodo recibirá la petición (sin importar si esta previamente ya le ha llegado vía otro nodo) y buscará en su base de datos local por posibles aciertos.

  Por ejemplo, asumiendo un TTL típico de 7 y un promedio de 4 conexiones C por *peer* (cada *peer* enviará el mensaje a otros 3), el número total de mensajes originados por un mensaje Gnutella (incluyendo las respuestas) puede ser calculado como [AH]:

$$2\sum_{i=0}^{TTL} C(C-1)^i = 26240$$

  En un escenario real, algunos experimentos recientes han mostrado que el tráfico de la red Gnutella acumula hasta 3.5Mbps (o 353,396 peticiones en 2.5 horas) [AH].

- **Escalabilidad**. El problema de la escalabilidad esta ligado con el costo del método empleado para efectuar el rastreo de la información disponible en un sistema distribuido descentralizado. Las preguntas que necesitamos responder son: ¿cómo encontrar algún dato, digamos un archivo, que ha sido agregado en un sistema P2P extenso, de manera escalable, sin valerse de servidores centrales o de alguna estructura jerárquica? ¿Habrá alguna manera de conocer directamente en cuales nodos está localizado algún contenido especifico, o se necesitará buscar "aleatoriamente" en la red entera para encontrarlo?

  Se ha abordado este problema de muchas maneras, una de ellas es manteniendo una base de datos central que mapee el nombre de un archivo con la localización del nodo que lo guarda. Napster (http://www.napster.com/) por ejemplo, adoptó esta modalidad para los títulos de las canciones, pero ésta tiene problemas inherentes de confiabilidad que lo hace vulnerable a los ataques a su base de datos, y también de escalabilidad, ya que están acotados a la limitante de almacenamiento de la base de datos en el servidor y a su capacidad de respuesta a las peticiones.

  Del otro lado del espectro, se encuentra Gnutella (http://gnutella.wego.com/) y su mecanismo de *broadcast*, un método que tampoco es escalable debido al ancho de banda y a los ciclos de cálculo consumidos por el excesivo número

de nodos que tienen que manipular estos mensajes. Como dato curioso, se sabe que, el día siguiente de que Napster fue clausurado, los reportes indicaron que la red Gnutella se colapsó sometida a su propia carga, creada cuando un gran número de usuarios migró a ésta para poder seguir compartiendo música.

Para reducir el costo de *broadcast*, los nodos pueden ser organizados en la red de acuerdo a una estructura jerárquica, como los sistemas de nombre de dominio (DNS) en Internet. Los buscadores empiezan en la jerarquía más alta (Super Nodos, asignados dinámicamente si tienen suficiente ancho de banda y capacidad de procesamiento, para cubrir una pequeña parte de la red) y siguiendo las referencias transmitidas de un nodo a otro recorren una ruta única descendente hacia el nodo que contiene la información deseada. El recorrido directo por una ruta única consume menos recursos que un *broadcast*. La mayoría de los sistemas actuales más populares, como KaZaA, Grokster y MusicCity Morpheus, están basados en la plataforma P2P FastTrack (http://www.fasttrack.nu/) que adopta este método. La desventaja de este procedimiento, es que los nodos de mayor jerarquía en el árbol toman una fracción mayor de la carga que los nodos hoja y por tanto requieren un hardware más caro y una administración más cuidadosa. Una falla o la remoción de la raíz del árbol o de algún nodo lo suficientemente alto en la jerarquía puede ser catastrófico [BKKMS].

Los algoritmos de búsqueda distribuida simétrica previenen los inconvenientes de los métodos anteriores. Las búsquedas se llevan a cabo siguiendo referencias de nodo a nodo hasta que el nodo apropiado que contiene los datos, es encontrado. Esto es posible debido a que son sistemas estructurados, en los cuales la topología de la red sobrepuesta está fuertemente controlada y los archivos (o los apuntadores a ellos) están colocados en sitios especificados de manera precisa [LRS]. A diferencia de las estructuras jerárquicas, en este, ningún nodo juega algún rol especial, una búsqueda puede empezar en cualquier nodo y cada nodo se involucrará en una pequeña fracción de la búsqueda de la ruta en el sistema. Como resultado, ningún nodo consume una cantidad excesiva de recursos mientras ayuda en el proceso de búsqueda. Estos nuevos algoritmos (CAN [RFHK], Chord [SMKKB], Pastry y Tapestry [ZKJ]) presentan una interfase general y simple, una tabla hash distribuida (THD). Cada archivo agregado en el sistema, es insertado en una THD y es registrado especificándole una clave única. Para implementar una THD, el algoritmo debe ser capaz de determinar que nodo es responsable de guardar el archivo asociado con alguna clave dada para que las peticiones de búsqueda puedan ser enrutadas eficientemente hacia el nodo que contiene el archivo deseado [LRS]. Para resolver este problema, cada nodo mantiene la información (dirección IP) de un pequeño número de nodos vecinos en el sistema, formando una red sobrepuesta a fin de enrutar mensajes para guardar y obtener claves. Las aplicaciones distribuidas que hacen uso de dicha infraestructura heredan sus propiedades: robustez, escalabilidad, facilidad en el mantenimiento de las tablas

de ruteo por nodo y balanceo en la distribución de las claves entre los nodos participantes [BKKMS].

FreeNet [CSW] (http://freenet.org/) fue uno de los primeros sistemas que empleó algoritmos de este tipo. Pero el objetivo principal de FreeNet era garantizar el anonimato de sus participantes, lo que generó algunos retos en el diseño del sistema, desarrollando un servicio de almacenamiento de archivos, donde estos eran puestos en otros nodos para guardarse, replicarse y persistir, en lugar de un servicio que compartiera archivos, donde estos solamente fueran copiados a otros nodos si se hiciera la petición directa. Para proporcionar el anonimato, FreeNet evita asociar un documento con cualquier servidor predecible, entre otras medidas, asignando al contador TTL un valor aleatorio a fin de ocultar las distancias (así que no todas las búsquedas son siempre exitosas) o conformar una topología previsible entre los servidores. Como consecuencia de esto, los documentos impopulares podrían simplemente desaparecer del sistema porque ningún servidor tiene la responsabilidad de mantener replicas y una búsqueda podría necesitar visitar a veces una gran parte de la red FreeNet [BKKMS].

Dada la diferencia en el nivel de desempeño producido por las THD, es natural preguntarse porque proponer un algoritmo nuevo para la búsqueda de archivos en un sistema P2P si se cuenta con este método; ya que para localizar un archivo específico, la operación *lookup()* que se proporciona en las THD requiere únicamente de $O(\log n)$ pasos, en comparación, Gnutella requiere $O(n)$ [CRBLS]. Aquí las razones:

1. Los clientes P2P son sumamente transitorios. Es difícil mantener la estructura requerida para enrutar las peticiones de búsqueda en una población de nodos altamente dinámica como lo son los actuales sistemas P2P [LRS]. Después de cada falla, la mayoría de las THD requieren de $O(\log n)$ operaciones de reparación para preservar la eficacia y la exactitud del ruteo. Las fallas en donde los nodos no hayan informado de antemano a sus vecinos y no hayan transferido la información relevante requieren más tiempo y esfuerzo en el sistema para (a) descubrir la falla y (b) replicar los datos perdidos o los apuntadores. Si el sistema es altamente dinámico, el trafico causado por la operación de reparación puede llegar a ser sustancial

2. Las búsquedas por palabras-clave son más comunes e importantes que las peticiones estrictamente iguales. Las THD tienen la ventaja de llevar a cabo búsquedas de igualdad rigurosa: dado el nombre exacto de un archivo, este se traduce a una clave y se ejecuta la operación *lookup(clave)* correspondiente, y en todo caso, al desempeñar búsquedas por palabra-clave, las THD son menos eficientes: dada una secuencia de palabras-clave, encontrar el archivo que les iguale. El uso común de

los sistemas P2P actuales, que comparten archivos de música y video, requieren justamente el empatar y comparar palabras-clave. Por ejemplo, para encontrar la canción "Ray of Light" de Madonna, un usuario hará típicamente una búsqueda de la forma "madonna ray of light" y contará con que el sistema localice el archivo que iguale todas las palabras-clave de la petición. Esto es especialmente importante porque no existe una convención para nombrar los archivos en los sistemas P2P y por tanto muchas veces, el mismo contenido es guardado por diferentes nodos bajo varios nombres completamente distintos. Lo que Gnutella resuelve llevando a cabo búsquedas locales en cada nodo [CRBLS].

3. La mayoría de las peticiones son para buscar archivos populares. Las THD manejan recordatorios exactos, sabiendo el nombre de un archivo es posible encontrarlo a pesar de que exista una sola copia en el sistema. En comparación, Gnutella no puede encontrar con seguridad copias únicas de estos archivos llamados impopulares, a menos que se haga una petición que llegue a todos los nodos (*broadcast*). Pero, según [CRBLS], la mayoría de las peticiones en las aplicaciones P2P que comparten archivos son para buscar archivos relativamente bien-replicados, llamados archivos populares. Dado que la esencia de los sistemas P2P que comparten archivos ocasiona que haya más copias en el sistema de un archivo que es solicitado frecuentemente, Gnutella puede lidiar con ellos y encontrarlos fácilmente.

El éxito de la revolución de los sistemas P2P, dependerá en gran medida de la habilidad de sus aplicaciones en ofrecer una comunicación eficiente entre un enorme y creciente número de *peers* autónomos y dispersos en toda Internet [J].

Técnicas propuestas en la búsqueda de archivos en Gnutella

En esta sección se describirá brevemente algunas técnicas propuestas para realizar la búsqueda eficientemente de archivos en redes P2P no estructuras, como lo es Gnutella.

Captura de los mensajes *Query* [Ma].

La técnica de controlar los resultados de los mensajes *Query* ha sido considerada para ayudar a resolver los problemas de escalamiento en el método de búsqueda de archivos en Gnutella. La idea es que un servent también capture los mensajes *QueryHit* que el observe, así que al recibir un *Query* busque en su registro el resultado de esta petición, si la tiene, entonces contesta con el *QueryHit* registrado y ya no reenvía el mensaje *Query* a

sus vecinos. Este método funciona bien dado que la gente tiende a buscar archivos populares.

Ajustar el alcance [LCCLS].

Una solución sencilla para reducir el tráfico de *broadcast* generado por los mensajes *Query*, se logra fijando un valor bajo de TTL en los mensajes *Query* iniciales. Si después de un periodo de tiempo, no se ha recibido alguna replica, el mensaje *Query* es reenviado con un ligero incremento en el valor del TTL. El *re-broadcasting* de un *Query* continúa hasta que se obtiene una respuesta o hasta que el valor del TTL ha alcanzado un límite máximo predefinido. Este método reduce el número de *broadcasts* necesarios para obtener los datos si estos se encuentran cerca (en *hops*) del nodo que esta efectuando la petición, pero incrementa el número total de mensajes enviados por petición cuando los datos están lejos o son inalcanzables. De cualquier manera, el ir ajustando el valor del TTL y expandir poco a poco el alcance del *Query* reduce el número de mensajes en comparación al empleado por Gnutella en donde el valor del TTL permanece constante. Un problema inherente es que no evita la duplicidad de los mensajes de petición enviados.

Caminantes con recorridos aleatorios [ALPH, LCCLS].

En lugar de que un nodo haga el *broadcast* de un mensaje *Query* a todos sus vecinos, éste escogerá al azar un *peer* y reenviará el mensaje únicamente a ese, lo que elimina inmediatamente el crecimiento exponencial en el tráfico de mensajes. Se puede obtener una mejoría en la eficiencia de la búsqueda, si en la selección del *peer* se elige a aquel con un alto número de vecinos.

Aunque con este método, el usuario percibe un retraso en la respuesta a la petición, se ha mostrado que enviar solamente un mensaje *Query* ("caminante") por petición en lugar de enviar múltiples caminantes paralelos, reduce considerablemente el costo de una búsqueda. Aun en los experimentos con métodos que eliminan a los caminantes considerando el TTL y usando entre 16 y 64 caminantes paralelos por petición, y otros donde el caminante se comunica periódicamente con el nodo que lo originó preguntando si debe de continuar su búsqueda por mejores resultados o finalizar.

Enrutar las peticiones [R].

El dirigir los mensajes *Query* puede llevarse a cabo únicamente por *peers* que comparten entre si la información de sus recursos, lo que permite enviar los mensajes solamente a aquellos que probablemente contengan el archivo especificado. Una de las técnicas que se ha propuesto es el tener *peers* que usen una función hash que mapee todas las palabras-clave que representen sus archivos compartidos en una tabla hash. Periódicamente cada *peer* actualizará su tabla hash de palabras-clave con aquellas de sus *peers* que le permiten direccionar las peticiones para ciertas palabras a lo largo de las rutas más probables.

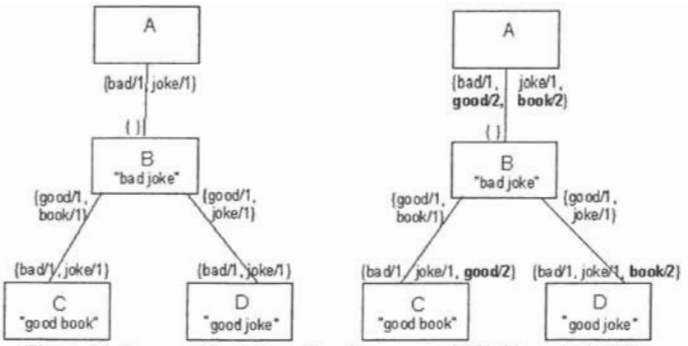


Figura 4.1. Propagación de las tablas de ruteo por TTL. Tomada de [R]
Por simplicidad, las palabras-clave no están en valores de la función *hash*.

Como ejemplo, se puede considerar la red mostrada en la figura 4.1, el hecho de que esta red represente un árbol es una coincidencia, no es necesario tener una raíz y pueden existir circuitos en ella. Después de un primer paso, los *peers* han intercambiado las tablas de ruteo para los archivos que están a un-*hop* de distancia. Esto se puede ver del lado izquierdo de la figura, por simplicidad son mostrados el conjunto de palabras en lugar de los arreglos con los valores de la función hash. Por ejemplo, la tabla {bad/1, joke/2} debe ser representada realmente como el arreglo [8, 2, 8, 1, 8, ...] si la función hash de "bad" es 3 y la de "joke" es 1. Después de un segundo paso, los *peers* han intercambiado las tablas de ruteo de los archivos que están a dos-*hops* de distancia. Esto se puede ver del lado derecho de la figura y se observa que *A* tiene ahora los valores de ruteo para todos los archivos y *B* ahora sabe que no puede conseguir ningún archivo a través de *A*.

A fin de transmitir eficientemente las tablas hash, se usa alguna técnica de compresión y se propone tener actualizaciones de las tablas por incrementos o por parches en lugar de enviar repetidamente todas las tablas entre los *peers*.

Control de flujo y heterogeneidad [BM].

El protocolo Gnutella considera a todos los *peers* iguales, aunque de hecho, algunos de ellos tienen mayores capacidades (poder de calculo, ancho de banda disponible, etc.) que otros, lo que estas diferencias pueden ocasionar sobre algunos *peers* es sobrecargarlos; para evitar esto, se puede usar un protocolo de control de flujo. Cada *peer* debe de verificar periódicamente el estatus de sus mensajes de entrada para asegurarse de no estar siendo saturado, si esto sucediera, debido a un *peer*-transmisor de gran capacidad, éste buscará entre sus vecinos a uno de capacidad apropiada para convertirse en el nuevo receptor y modificar la ruta de conexión. Si no se encontrara un *peer* adecuado, se le pediría al emisor bajar su tasa de transmisión. Este proceso ocasiona que la red se auto regule, de tal manera que los *peers* de capacidades similares se conectarían entre si y esto protegería la saturación de otros.

Optimización de la red sobrepuesta de comunicación (GVR y GVRC) [EPSS, LXLNZ].

Para propagar los mensajes de *broadcast* se construye un grafo de comunicación aplicando el algoritmo de Grafos de Vecindad Relativa (GVR) empleando el modelo uno-a-uno. Cuando un nodo *A* desea difundir un mensaje, primero identifica cuáles nodos pertenecen al GVR y entonces envía el mensaje únicamente a esos nodos, y así sucesivamente, con lo que la información se propaga a toda la red sin las desventajas del *flooding*. La principal ventaja del GVR es que genera una red conectada poco densa (la proporción entre el grado promedio y el número de nodos es pequeña) con base solamente en la información local recopilada por cada nodo.

El GVR es un concepto teórico geométrico y gráfico propuesto por Toussaint [T], en el cual se establece lo siguiente:

Un enlace (u,v) existe, si la distancia entre los vértices u y v, d(u,v), es menor o igual a la distancia existente a cualquier otro vértice w, donde w es vecino tanto de u como de v. En otras palabras, un enlace (u,v) existe si $\forall w \neq u,v : d(u,v) \leq max\,[d(u,w), d(v,w)]$

Para que un enlace(*u,v*) este incluido, la intersección de dos círculos centrados en *u* y *v* y con diámetro *uv* (en la figura 4.2 es el área sombreada) no deberá contener algún vértice *w* del conjunto de vértices.

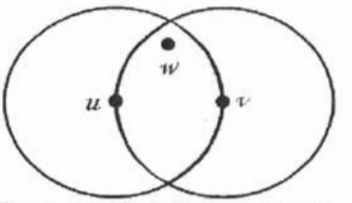


Figura 4.2. El enlace(u,v) no esta incluido en el GVR debido a la presencia del nodo w. Tomado de [SSS].

Toussaint demostró varias propiedades importantes de los grafos de vecindad relativa, las cuales son necesarias para su utilización en la tarea de *broadcasting*. Un GVR es un grafo conectado y plano. Lo cual resulta de suma utilidad, pues un grafo plano con *n* vértices puede tener a lo más *3n-6* aristas [SSS].

La extensión propuesta por [LXLNZ] incluye sobre el Grafo de Vecindad Relativa la remoción de algunos otros enlaces extra que poseen conexiones lentas. La generación de la red de comunicación Grafo de Vecindad Relativa con remoción Cuadrilateral (GVRC) esta conformada esencialmente de tres operaciones. La primera incluye la recopilación de la información local de cada nodo sobre una vecindad de 2-*hops*.

En un ambiente de sincronización total y en base a la estampa de tiempo del nodo fuente *S* distante 2-*hops* y del retraso en la recepción del mensaje del nodo vecino de 1-*hop B*, el nodo *A* calcula el peso del enlace *SB* entre sus vecinos de uno y dos *hops*. La segunda operación elimina las conexiones lentas, lo que significa que el nodo *A* excluirá el enlace *AB* si éste tiene el retraso más alto de entre las

conexiones AB, SA y BS. El nodo A también eliminará el enlace AS si tiene el retraso mayor de entre SB o BA (hay que notar que el peso de las conexiones se asume de manera simétrica). Si SB posee el peso mayor de entre los tres enlaces, A hace nada. Esta operación es la misma que se ha aplicado de manera independiente en [EPSS] y referida como RNG.

La tercera operación se aplica en los cuadrángulos formados cuando un vecino distante 2-*hops* tiene dos o más rutas hacia un nodo dado, lo que significa que el enlace con el peso más alto en cualquier cuadrilátero es removido. Para programar esto de manera eficiente, se han considerado todos los pares que forman los vecinos distantes por 2-*hops* S y A, y encontrado todos los vecinos comunes B_1, B_2, ..., B_k. Digamos que d_i es el peso más alto entre SB_i y B_iA. Esto es, $d_i=max(peso(SB_i), peso(B_iA))$, $1\leq i\leq k$. y $D=min\ d_i$ esto es, el mínimo entre ellos. Los enlaces SB_i y B_iA, ambos se preservan, pero el enlace con el peso más alto dentro de cualquier par SB_i, B_jA, para $j\neq i$, $1\leq j\leq k$ es removido. GVRC será el grafo que se obtiene al aplicar ambas remociones, una debida al GVR y la otra a esta eliminación cuadrilateral. Se ha propuesto emplear una triada de valores para asegurar la singularidad de los grafos GVR y GVRC, la clave primaria de comparación será el peso del enlace, la secundaria y terciaria, los respectivos identificadores (ordenados de manera ascendente) de los nodos terminales del enlace. La efectividad de este algoritmo es demostrada en [LXLNZ] mediante simulación, en este trabajo de investigación se hará de la misma manera.

Las siguientes líneas resumen, en pseudocódigo, el algoritmo GVRC:

obtener el GVR de la topología de red inicial
para cada nodo A en GVR
definir sus vértices adyacentes B
para cada vértice B_k en B
obtener sus vértices adyacentes S
para cada vértice S_k en S
obtener peso$A\ B_k$
obtener peso$B_k\ S_k$
para cada vértice B_j adyacente a A
si B_j es adyacente a S_k se ha encontrado otra ruta
obtener pesoAB_j
obtener pesoB_jS_k
si se ha encontrado otra ruta entre vecinos, eliminar enlaces

Selección aleatoria de vecinos (Gossip) [PS]

Portmann y Seneviratne [PS] presentaron una alternativa para superar las limitaciones del *flooding* en términos de costo y escalabilidad para redes P2P descentralizadas y no estructuradas, como lo es Gnutella. La propuesta consiste en el empleo de un protocolo de enrutamiento llamado propagación de rumores (PR) (también conocido

como *Gossip* o "chismorreo") empleando como marco referencial el protocolo Gnutella [GPS]. Este es un método escalable ya que cada nodo retransmite un número fijo de mensajes independientemente del número de nodos en la red. Aunado a que ningún nodo debe de esperar una respuesta de confirmación en la recepción del mensaje [LMM].

Los protocolos de propagación de rumores pertenecen a la clase de protocolos probabilísticos empleados para el ruteo de mensajes en donde los nodos-vecinos a los cuales se envía un mensaje son escogidos al azar. [PS] evaluaron un tipo específico de algoritmo llamado **propagación de rumores con contador invidente**. La estructura principal de este algoritmo es la siguiente:

Un nodo A inicia un broadcast al enviar un mensaje m a B de sus vecinos elegidos de forma aleatoria

cuando un nodo p recibe un mensaje m de un nodo q
si p ha recibido m no más de F veces
p envía m a B vecinos elegidos de forma aleatoria solo si p sabe que aun no han recibido m

El nodo *p* sabe si su vecino *q* ha recibido ya el mensaje *m* solo si este se lo ha enviado previamente o si lo ha recibido de él.

En el caso que el número de vecinos válidos de un nodo sea menor que *B*, el mensaje será enviado a ese reducido número de vecinos.

El parámetro *B* especifica el número máximo de vecinos a los cuales un mensaje *m* será reenviado. El parámetro *F* determina el número de veces que un nodo reenvía el mismo mensaje *m* a *B* de sus vecinos. Los parámetros *F* y *B* pueden ser empleados para controlar las propiedades del algoritmo.

Debido a su naturaleza, un protocolo Gossip no puede garantizar que todos los nodos sean alcanzados durante el *broadcast* de un mensaje. Lo cual puede ser un factor decisivo, si tomamos en cuenta que el alcance o el número de nodos que reciben un mensaje en particular, es una métrica de desempeño importante para muchas aplicaciones P2P, particularmente en aquellas empleadas para compartir archivos [J].

El protocolo Gossip puede tener algunas variantes dependiendo de algunas consideraciones, como podrían ser: las capacidades de memorización y el uso simultaneo en ambas direcciones de un mismo enlace. Se ha definido **Gossip-Simple** como el proceso de gossip basado en *flooding* sencillo en el cual un enlace puede ser usado únicamente en una sola dirección para transmitir el mismo mensaje. Por lo tanto, en esta variante no es posible considerar las transmisiones simultaneas en ambos sentidos. Bajo el entendido de que esta versión ha sido considerada en [LMM] y también en nuestros experimentos. **Gossip-Doble** es definido como el proceso en el cual un enlace puede ser usado en ambas direcciones para enviar el mismo mensaje, cada nodo memoriza el mensaje recibido pero no a los emisores previos excepto al

ultimo. Los experimentos con esta versión de Gossip, producen valores que coinciden con los reportados en [PS], así que sospechamos que no fue implementada en los nodos, una lista con los emisores previos. Existen aun otras variantes en el protocolo Gossip, en donde por ejemplo, se mantiene una lista con los emisores previos, pero es permitido el uso simultáneo de un enlace en ambas direcciones; dando como resultado que dos nodos puedan transmitir el mismo mensaje entre si simultáneamente antes de incluirse uno a otro en sus respectivas listas de emisores. Ninguna variante dentro del esquema de Gossip Simple o Doble fue implementada dentro de este trabajo y consecuentemente no aparece reportada en los datos experimentales.

Los resultados de la tabla 4.2 se obtuvieron usando topologías generadas usando el modelo de Barabasi-Albert con n nodos y un grado de nodo promedio d, con valores n=1000 y d=4, 6 y 8 respectivamente. Debido a la naturaleza paramétrica de el protocolo Gossip, el *porcentaje de nodos alcanzados* (el porcentaje de nodos que han recibido una copia del mensaje enviado) varía de acuerdo a los valores de B y F. El número de mensajes promedio en el protocolo Gossip-Doble es comparado con el costo de *flooding* de Gnutella (representando el 100%) para obtener el *porcentaje de costo* de Gossip. En el capítulo 6 se presenta un análisis más profundo del costo de *broadcasting* en el protocolo Gossip.

		d=4		d=6		d=8	
B	F	Costo	Alcance	Costo	Alcance	Costo	Alcance
2	1	30%	55%	27%	68%	19%	67%
2	2	53%	82%	51%	92%	42%	93%
3	1	49%	76%	46%	87%	38%	88%
3	2	69%	93%	67%	98%	61%	99%

Tabla 4.2. Protocolo Gossip Doble.

Sea cual fuere la solución propuesta para lidiar con la escalabilidad de Gnutella, la tendencia de los desarrolladores se enfoca más en tratar de mantener la compatibilidad con los clientes viejos, intentando hacer pequeñas modificaciones sin perder en el proceso a muchos usuarios. De los métodos de optimización propuestos para realizar búsquedas de archivos, queda claro que una red P2P no estructurada, como lo es Gnutella, puede hacerse que escale de manera más eficiente, lo que significaría realizar grandes modificaciones que no son posibles de implementar mientras se quiera mantener la compatibilidad con la versión anterior del protocolo Gnutella [BM].

Topología de la red Gnutella

Dada la gran diversidad de redes que exhiben la estructura de ley-de-potencia y sus propiedades, nos interesa saber si Gnutella también cae en esta categoría. En [J] se ha reportado, según datos experimentales tomados en diciembre del 2000, que la red Gnutella obedece a las cuatro leyes de potencia descritas por Faloutsos *et al* [FFF]. El trazo

logarítmico de los resultados obtenidos para estas leyes se muestra en las figuras 4.1, 4.2, 4.3 y 4.4 respectivamente.

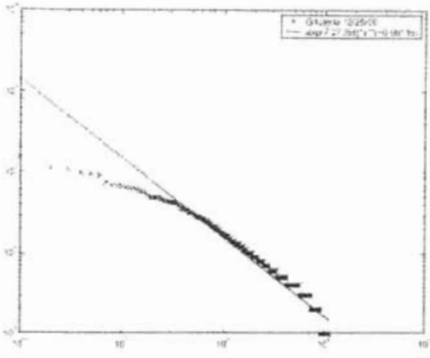


Figura 4.1. Ley de potencia 1.
Exponente de orden R = -0.98 y coeficiente de correlación = 0.94. Tomada de [J]

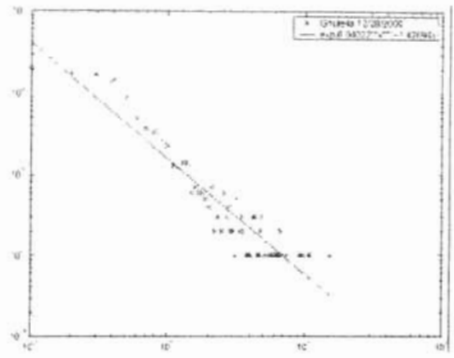


Figura 4.2. Ley de potencia 2.
Exponente de grado ? = -1.4 y coeficiente de correlación = 0.96

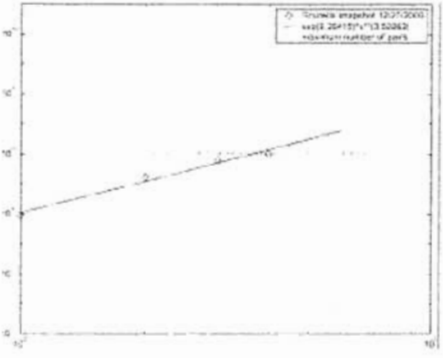


Figura 4.3. Ley de potencia 3.
Exponente de saltos H = 3.52 y coeficiente de correlación = 0.99

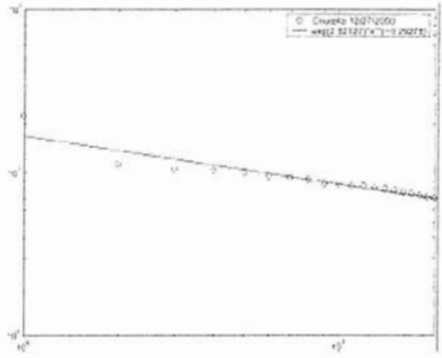


Figura 4.4. Ley de potencia 4.
Exponente de eigen-valores E = -0.29 para los primero 20 eigenvalores y coeficiente de correlación = 0.94

Para Gnutella, el exponente de orden R de la ley de potencia 1, es igual a –0.98. La ley de potencia 2 tiene una importancia particular, porque es una de las más frecuentemente citadas en los estudios sobre topologías de red, incluso se ha visto que Freenet, otra aplicación P2P que comparte archivos, también la presenta [J]. Esto significa que la mayoría de los servents tienen un grado pequeño y solo algunos cuantos tienen un grado muy alto [RFI, JAB]. Gnutella tiene de manera inherente un gran porcentaje de clientes que confía en un pequeño porcentaje de servidores. El 25% de los usuarios Gnutella no comparten ningún archivo, además, aproximadamente el 75% de los usuarios

comparten menos de 100 archivos y solamente el 7% comparten más de 1000. Un cálculo sencillo revela que este 7% de usuarios en conjunto, ofrecen más archivos que todos los demás[SGG]. Este hecho ilustra que a pesar de pretender que "cada *peer* sea servidor y cliente", en Gnutella esto parece no ser lema.

El hecho de que los *peers* se conecten y desconecten de la red Gnutella tiene implicaciones en la naturaleza de su topología. En la práctica, los *peers* tienden a encontrar a los nodos que son altamente disponibles y que tienen un alto grado de conectividad, esto establece una conectividad preferente [SGG]. Como Barabasi y Albert [BA] mostraron, la conectividad de los nodos en una red que continuamente se expande agregando nuevos nodos y en la cual los nodos expresan una conectividad preferente, tienen un patrón de organización que obedece a una distribución de ley de potencia. Más aún, dada la naturaleza de la red Gnutella en donde participan usuarios con las mejores conexiones a Internet (DSL y más), se tendrá una estructura que exhibe por si misma un patrón de conectividad de ley-de-potencia [RFI].

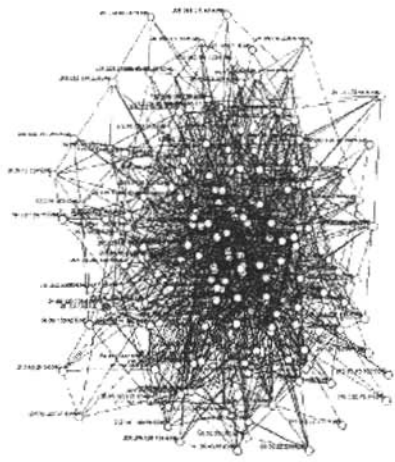


Figura 4.5. Red Gnutella mostrando los nodos con grado>10. Tomado de [J].

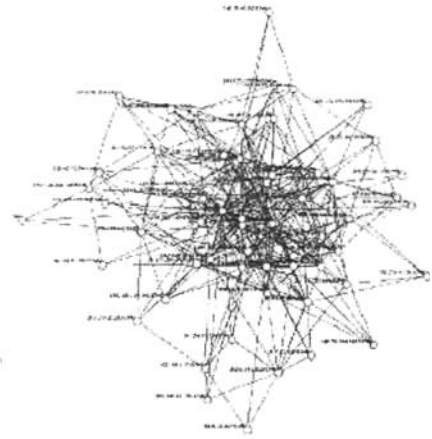


Figura 4.6. Red Gnutella mostrando los nodos con grado >20. Tomado de [J].

Las leyes de potencia 3 y 4 aplican, como ya se ha mencionado, para casi todos los tipos de topologías, incluyendo aleatorias, regulares y jerárquicas, por tanto no es de sorprenderse que en la topología de la red Gnutella también se observen. La presencia de estas dos leyes de potencia no representan alguna propiedad capaz de diferenciar una gráfica, pero el valor de sus exponentes sí lo son. La figura 4.3 está aproximada para los primeros cuatro *hops*, y por datos experimentales se ha observado, que el exponente de saltos H es consistente para al menos 4 muestras de la red tomadas en diferentes momentos y se acerca al valor de 3.5. Este valor se encuentra entre los valores de los exponentes reportados por Faloutsos [FFF] para las topologías a nivel ruteadores (H =2.8) e inter-dominios (H =4.86) de Internet. Al

igual que en [FFF], es posible que los resultados para esta ley de potencia en particular sean engañosos dado el pequeño número de datos a trazar. Esta limitante es impuesta por el hecho de que estas gráficas tienen un diámetro pequeño [J].

Una aplicación de la ley de potencia 3 que parece ser particularmente aplicable a Gnutella es el concepto del diámetro efectivo δ_{ef}, que esencialmente se refiere al número de *hops* requeridos para alcanzar una porción de red "lo suficientemente grande". Sustituyendo los valores para una muestra de diciembre del 2000 de la topología Gnutella en la ecuación definida en [FFF] para el diámetro efectivo, [JoAB] reportó un valor más apropiado para el TTL máximo, el cual debe de ser 4 en lugar de 7 (el valor por omisión especificado para el protocolo Gnutella).

Los resultados empíricos obtenidos en [J] muestran una fuerte relación de la red Gnutella con las leyes de potencia. Por lo mismo se usarán los modelos que generan topologías con una distribución de ley de potencia del grado promedio de los nodos propuestos por Barabasi-Albert, Palmer-Steffan y Waxman. Los mismos que empleamos para generar topologías Internet.

Capítulo

Proceso de simulación

La simulación de broadcasting sobre un grafo de Internet se desempeña según dos procesos: primero se genera una topología con un número de nodos y enlaces definidos especificando un grado promedio de los nodos. Segundo, el concepto ALEM es aplicado para reducir el número de enlaces redundantes en el grafo.

En este capítulo convergen las ideas y conceptos plasmados en los capítulos previos. Se presenta una descripción del proceso de simulación de los algoritmos de *broadcast* sobre topologías Internet y redes *peer-to-peer*. Se describen las herramientas implementadas, empleadas para aplicar los conceptos de los Grafos Unitarios Aleatorios a las topologías Internet. Se toman en cuenta la descripción de las métricas de costo que se emplearon para evaluar el desempeño de los algoritmos. Y por último se describen las clases usadas en el proceso de simulación.

En el proceso de *broadcasting* un nodo inicial disemina un mensaje a lo largo de toda la red iniciando por enviar una copia de este mensaje a todos sus vecinos. La cantidad de tiempo que le toma a un nodo el recibir un mensaje y reenviarlo a sus vecinos selectos es llamado *ronda*. Algunos nodos tienen rondas corriendo en paralelo. El algoritmo termina, o *converge*, cuando todos los nodos en la red han recibido al menos una copia del mensaje. El proceso de *broadcasting* converge en $O(diam)$ rondas (donde *diam* es el diámetro de la red), ya que le toma al menos *diam* rondas para que un dato viaje de un extremo a otro de la red.

En el esquema de *Blind-Flooding*, siempre que un nodo recibe el mensaje por primera vez, lo reenvía a sus vecinos (sin importar si lo han recibido previamente de algún otro nodo vecino), excepto al nodo del cual lo acaba de recibir. Este es el método clásico empleado para diseminar información de manera rápida en una red con suficiente ancho de banda y sin propensión a perder la conexión de los enlaces [HKB]. Sin embargo, este método tiene desventajas, prohibiendo su uso en redes con ambientes dinámicos o cuando las peticiones de información son frecuentes y deben de ser comunicadas a todos los participantes. La principal desventaja de *Blind-Flooding* es el uso excesivo de ancho de banda si se emplea una red subyacente muy densa. Está claro

que este sistema no es escalable para aplicaciones en Internet ya que genera un gran número de mensajes redundantes y usa todas las rutas existentes a través de la red. Por esta razón, en la práctica, el mecanismo de *flooding* es típicamente implementado en combinación con uno o más de los siguientes mecanismos diseñados para restringir su alcance y limitar los mensajes redundantes:

Mecanismo 1. Limitado por el Tiempo-de-vida (TTL) de un mensaje. El TTL es un mecanismo regulador que previene que los mensajes vayan más allá de un número de *hops* específico, definido por un valor TTL inicial. Es implementado incluyendo en el encabezado de cada mensaje un campo con el valor TTL, cuando un nodo recibe un mensaje, primero verifica que el valor del TTL sea mayor que cero, si es así, el mensaje continúa expandiéndose con un valor TTL disminuido; de otra manera, el mensaje es desechado.

Mecanismo 2. Identificación única del mensaje (UID). Este mecanismo previene que el mismo mensaje sea transmitido más de una vez por cada nodo. Es implementado incluyendo en el encabezado de cada mensaje un UID (una etiqueta ID única o un número secuencial único). Cuando un nodo recibe un mensaje, verifica si lo ha recibido previamente, si es así, el mensaje no se reenvía y es desechado; de otra manera, el nodo guarda el nuevo UID en una tabla local y continúa su expansión.

Mecanismo 3. Identificación del recorrido. Este mecanismo evita que las trayectorias de los mensajes recorran circuitos cerrados. Se implementa incluyendo en cada mensaje un encabezado que registra los nodos de la red que ya han sido visitados. Antes de reenviar los mensajes, cada nodo simplemente verifica en el encabezado si esta dentro de la lista, si es así, el mensaje no se reenvía y es desechado; de otra manera, el nodo añade su nombre al encabezado y continúa su expansión.

Se considerará en este trabajo el esquema de *Blind-flooding* como el proceso de *broadcast* que opera bajo la combinación de los mecanismos 1 y 2.

A pesar de la definición estricta de las reglas de *Blind-Flooding*, en donde los mensajes son retransmitidos a todos los vecinos excepto a aquel del cual previamente se recibió, consideramos que esta descripción queda un poco incompleta, con dos variantes que pueden ser usadas para completarla.

El tipo de *Blind-Flooding* definido en el sistema Gnutella, es un *broadcast* que sigue un modelo donde los enlaces pueden ser usados para reenviar el mismo mensaje en ambas direcciones (por ejemplo, del nodo A al nodo B y viceversa). Nos referiremos a él como el modelo ***Flooding*-Doble**. El número total de mensajes en el modelo *flooding*-doble del protocolo *blind-flooding* que necesitan ser transmitidos en la red para lograr un sólo *broadcast* es, de manera general:

$$c = \sum_{i=1}^{N} m_i \qquad (5.1)$$

m_i: número de mensajes enviados por el nodo i
N: numero de nodos en la red

Asumiendo que el valor TTL inicial para el *broadcast* es lo suficientemente alto como para cubrir completamente la red, el costo puede ser calculado relativamente fácil. Durante un *broadcast*, cada nodo recibe el mensaje al menos una vez. La primera vez, el mensaje es recibido, el nodo receptor lo reenvía a todos sus vecinos, excepto a aquel del cual lo recibió. Las llegadas subsecuentes del mensaje son ignoradas. Por tanto, el número de mensajes que cada nodo reenvía por *broadcast* está limitado por el número de sus vecinos menos uno. Solamente el nodo que inicia el *broadcast* envía el mensaje a todos sus vecinos, lo cual resulta en un incremento de uno en el costo total. Lo que significa que cada nodo n retransmite a d-1 vecinos en promedio, más un mensaje de más del nodo fuente debido a que no tiene predecesor en el proceso. Por tanto, el costo total c de un *broadcast* puede ser calculado simplemente como una función del número de nodos N y el número promedio de vecinos d. Es interesante notar, que el costo total crece linealmente con el grado de nodos promedio y con el tamaño de la red.

$$c = 1 + \sum_{i=1}^{N} (d_i - 1) = 1 + N(d - 1) \qquad (5.2)$$

c : costo en número de mensajes transmitidos
d_i : grado del servent i
$d = 2L/N$: grado de servents promedio
N : número de servents
L : número de enlaces

En la implementación, cada nodo recibe algunas copias del mismo mensaje antes de decidir en su retransmisión. Cuando se toma la decisión, el nodo elimina solamente el último vecino del cual fue recibido el mismo mensaje, y entonces transmite simultáneamente a todos los otros (incluyendo a aquellos de los cuales fue recibido el mensaje previamente). Por eso es posible que el mismo mensaje viaje simultáneamente en el mismo enlace en ambas direcciones.

Para tener una idea acerca de la escalabilidad en los servicios de Gnutella basados en *broadcasting*, es necesario conocer la carga en términos de los recursos empleados por servent en lugar de la carga total en toda la red. Tratando de efectuar una aproximación del ancho de banda promedio que es consumido por cada servent en un escenario *peer-to-peer* típico, se asumirá de manera conservadora que: los N servents de la red inician un *broadcast* con la misma velocidad constante r =1 bit/min; lo cual resulta en una velocidad total de $N*r$ para toda la red. También se asumirá un tamaño de mensaje de 22 bytes, el cual representa el tamaño mínimo de un mensaje Gnutella.

Con base en la fórmula 5.2, se puede calcular el número total de mensajes generados en la red por cada *broadcast* ejecutado. Este valor será multiplicado por el tamaño de

mensaje y por la velocidad de transmisión total. Este valor se multiplicará por 2, ya que cada mensaje transmitido consume tanto ancho de banda en la dirección de entrada como en la salida en cada nodo. El número resultante representa la cantidad total de ancho de banda consumido en la red. Finalmente, para obtener un promedio del ancho de banda usado por cada nodo, se tendrá que dividir por N. La tabla 5.1 muestra los resultados en Kbits/s para diferentes tamaños de red *N* y grado de nodos promedio *d*. [PS]

	d = 3	d = 4	d = 5	d = 6
N = 100	1.18	1.76	2.35	2.94
N = 1,000	11.74	17.61	23.47	29.34
N = 10,000	117.34	176.01	234.67	293.34
N = 100,000	1173.34	1760.00	2346.67	2933.34

Tabla 5.1. Consumo promedio de ancho de banda (en kBit/s) por nodo para un escenario P2P.

La tabla 5.1 muestra que para redes grandes, el ofrecer servicios usando *broadcast* requiere un ancho de banda promedio considerable. Por ejemplo, en una red de 10,000 servents y un grado promedio de 4, el ancho de banda promedio por servent es de 176 kBit/s. Este valor va más allá de los recursos de un usuario típico conectado vía modem. Sin embargo, este dato pareciera no tener una repercusión importante en Gnutella ya que el porcentaje de usuarios conectados con modem (de 64Kbps o menos) es aproximadamente el 8%, mientras que el 60% usa conexiones de banda ancha (Cable, DSL, T1 o T3) y el 30% conexiones con ancho de banda muy altos (al menos de 3Mbps). Sin embargo, en una red Gnutella real, se ha observado que el alto costo que requiere un *broadcast* y la falta de recursos de un gran número de participantes a ocasionado la fragmentación de la red en pequeñas subredes [PS]. Así que si el desarrollo de una red P2P se interesa en captar la atención de una mayor cantidad de usuarios, de los cuales la mayoría cuenta con conexiones lentas, sería prioritario cambiar al menos, la técnica de *broadcast* empleada.

En este trabajo también se define y se considera el modelo ***Flooding*-Simple**, como el proceso en el cual cada enlace en la red puede ser usado solamente una vez para enviar el mismo mensaje y consecuentemente únicamente en una dirección. Esto es, mientras el nodo *A* envía al nodo *B*, *B* no puede enviar simultáneamente a *A*. El nodo *B* procesa cada mensaje recibido de *A* y si encuentra el mismo mensaje en la cola para transmitirlo al nodo emisor *A*, éste es eliminado de la cola y nunca transmitido. En este modelo, *B* puede recibir el mismo mensaje de varios vecinos antes de poder retransmitirlo a los vecinos restantes. El número total de mensajes en el modelo Flooding-Simple es igual al número total de enlaces en la red usados para *broadcasting*, ya que cada enlace es utilizado exactamente una vez durante todo el proceso. El *flooding*-simple se justifica debido a que cada nodo en el *flooding*-doble necesita memorizar cada

mensaje recibido, para evitar reenviarlo múltiples veces a todos sus vecinos (excepto al emisor). Así que se propone simplemente extender la memorización para incluir a todos los emisores del mismo mensaje y guardarse tanto tiempo como el mensaje esté esperando en la cola para transmitirse a cualquiera de los vecinos. De hecho esta memorización puede restringirse a cada enlace independientemente, de tal manera que si el mismo mensaje llega de un extremo, se elimine de la cola del otro. Obviamente este modelo reduce el número total de mensajes. La comparación detallada del número de mensajes que comprende cada variante se hace en el capítulo 6.

El *broadcasting* basado en ALEM, GVR y GVRC puede ser usado con ambos modelos, *flooding*-simple y *flooding*-doble. En el caso de *flooding*-simple, el número de mensajes enviado es igual al numero de enlaces en el grafo. En el caso de *flooding*-doble, el número de mensajes enviado es $1+(d-1)n$, donde n es el número de nodos en la red y d es la densidad (número de vecinos) promedio en la estructura ALEM (o GVR y GVRC según sea el caso), aplicando alguna métrica. Estas medidas son usadas en la definición de la métrica de costo.

Métricas de desempeño

El desempeño de los algoritmos se evaluó usando básicamente dos métricas: *alcance* y *costo*. El alcance se define como el porcentaje de nodos abarcados por un protocolo de *broadcast* en particular. *Blind-flooding*, GVR, GVRC y el protocolo basado en ALEM tienen un alcance del 100% mientras que el protocolo Gossip no garantiza la entrega del mensaje a todos los nodos y por tanto, los valores de nodos alcanzados son incluidos en las tablas. Hay que tomar en cuenta, que el alcance, o el número total de nodos que reciben un mensaje en particular, es una métrica de desempeño importante para muchas aplicaciones P2P, particularmente en aquellas utilizadas para compartir archivos, porque de esta capacidad dependerá el éxito en la búsqueda de información.

Como métrica de costo se empleará el numero de mensajes generados y reenviados como resultado de un *broadcast*, ya que este número afecta directamente a los recursos disponibles en la red, tanto en los nodos como en los enlaces. Como no es sencillo interpretar una medida absoluta, emplearemos el costo relativo con respecto a un esquema de *broadcasting* estándar. Este esquema será *blind-flooding*, el costo (relativo), es calculado como la razón del número de mensajes total de un protocolo dado y el número de mensajes en *flooding* simple o doble basados en el esquema *blind-flooding*.

El desempeño del *flooding* simple o doble, cuando todos los enlaces de la red son usados, puede ser comparado teóricamente de la siguiente manera. En el caso del ***flooding*-simple**, el número de mensajes enviados es igual a $nd/2$ (lo que representa en realidad el número total de enlaces de una red), donde n es el número de nodos en la red y d es su densidad promedio. Si se usa el esquema de ***flooding*-doble**, el número de mensajes se incrementa hasta $1+(d-1)n$. Por lo tanto, el costo relativo del flooding-simple sobre flooding-doble es $nd/(2+2n(d-1))=1/(2/(nd)+2(1-1/d))$. Cuando n es un

número grande, esto se aproxima a $d/(2d-2)=0.5/(1-1/d)$. Para $d=4$ el costo relativo es 2/3, para $d=6$ esto es 3/5, para $d=8$ es 4/7, para $d=10$ es 5/9 y para un valor d alto, el costo relativo se aproxima al 50%, tal y como se puede esperar, ya que se emplea una sola dirección de cada enlace en lugar de ambas.

Los costos para *blind-flooding*, GVR, GVRC y el método basado en ALEM, ya han sido comentados a lo largo de este trabajo. El costo de *broadcasting* por el protocolo Gossip, es muy difícil de obtener por medio de alguna expresión analítica, como se establece en [PS] y [LMM], aún para topologías relativamente sencillas, así que como ellos, también recurrimos a la simulación del protocolo para comparar el desempeño de Gossip.

El simulador

Para analizar el proceso de *broadcasting*, se desarrollaron una serie de clases y métodos programados en Java haciendo uso de la biblioteca JDSL (Java Data Structures Library) [JDSL] para implementar algunas de las rutinas relacionadas a la teoría de grafos. Se usó Java como lenguaje de programación porque tiene algunas ventajas importantes:

- Su alta portabilidad a múltiples plataformas
- La disponibilidad de muchas estructuras de datos ya implementadas en sus bibliotecas estándar.
- Una construcción simple para la visualización y una interfaz gráfica de usuario amistosa.
- Gracias a su organización completamente modular por clases es posible estructurar código sencillo y legible.
- Alta flexibilidad en su código para re-usarse en diferentes situaciones.

El único inconveniente es que el procesar una gran cantidad de datos requiere una atención particular en el diseño de las clases. La tabla 5.2 contiene el nombre de las clases y una breve descripción de su función. Para obtener mayor información acerca de los métodos y las variables hay que referirse al apéndice A. Se incluye adicionalmente el código fuente del simulador en el apéndice B.

En términos generales, los pasos para obtener el $ALEM_1$, $ALEM_2$, AEM, GVR o GVRC de una topología Internet son los siguientes:

- La topología Internet es generada según los modelos Barabasi-Albert, Waxman o Palmer-Steffan, empleando el generador de topologías BRITE para los dos primeros y REC para el tercero. El grafo resultante es almacenado en un archivo.

- El archivo es un parámetro de entrada de la clase *InternetMap*.

- *InternetMap* lee el archivo y mapea sus datos en una estructura de datos conveniente para su manipulación posterior que almacenará en memoria.

- El grafo almacenado es empleado ya sea en la clase *LMST*, *MST*, *GVR*, *GVRC* o *RumorMongering* para obtener como salida los valores referentes al análisis del comportamiento de cada algoritmo (ALEM$_1$, ALEM$_2$, GVR, GVRC y Gossip).

Este proceso es repetido 50 veces para cada renglón de las tablas presentadas como resultados del siguiente capítulo.

Nombre de la clase	Función
LMST	Implementa los algoritmos de *broadcasting* ALEM$_1$ y ALEM$_2$.
MST	Implementa el algoritmo AEM de Prim usando ya sea la distancia euclidiana entre los nodos terminales de un enlace o el retraso en la transmisión como función de peso.
Prim	Ejecuta el algoritmo de Prim de un grafo usando el patrón definido en JDSL.
RNG	Implementa el algoritmo de *broadcasting* basado en los Grafos de Vecindad Relativa.
RNGQ	Implementa el algoritmo de *broadcasting* sobre Grafos de Vecindad Relativa con remoción Cuadrilateral.
RumorMongering	Implementa el algoritmo Gossip sobre algún grafo de entrada.
EdgeInfo	Manipula la información relevante de los enlaces, como su longitud y nombre.
VertexInfo	Manipula las propiedades de los vértices, como su nombre y ubicación.
InternetMap	Mapea los datos del archivo de salida de los simuladores BRITE o Recursivo a una estructura de datos conveniente.
GraphTools	Implementa diversas funciones útiles para trabajar con los grafos.
ConnectivityTester	Verifica la conectividad de cualquier grafo.

Tabla 5.2. Clases que contiene el simulador

Capítulo

Resultados de la simulación y su análisis

Con la topología formada por el algoritmo ALEM es posible obtener un ruteo global (broadcast) usando información local y llevando a cabo decisiones de transmisión locales.

Se llevó a cabo una serie de simulaciones en varias topologías de red estáticas para probar empíricamente los efectos del algoritmo propuesto sobre Gnutella e Internet. Por simplicidad se asumirá que los grafos de las topologías no cambiaran durante el proceso de simulación de nuestro algoritmo, de cualquier manera, si se asume que el tiempo para construir el ALEM y completar un *broadcast* es corto comparado con el tiempo que tomaría cambiar la topología de la red, los resultados obtenidos con esta condición son indicativos del desempeño en el sistema real.

Desempeño de $ALEM_k$ sobre topologías Internet

Los grafos de Internet empleados en este trabajo fueron generados usando los modelos: Barabasi-Albert [BA, AB], Palmer-Steffan [PaS] y Waxman [W]. El generador de topologías usado para generar los modelos Barabasi-Albert y Waxman fue BRITE y para el modelo de Palmer se uso el generador Recursivo. La simulación de *broadcasting* sobre un Grafo de Internet se desempeña según dos procesos: primero se genera una topología con un número de nodos y enlaces definidos especificando un grado promedio de los nodos. Segundo, el concepto ALEM es aplicado para reducir el número de enlaces redundantes en el grafo. Para generar topologías tipo Internet, se usaron tres modelos descritos en el capítulo 2. El número de nodos n varia desde 50 hasta 1000 y el número de enlaces es modificado cambiando el grado promedio de nodos d de el grafo Internet de entrada.

Es interesante saber si todas las propiedades del algoritmo ALEM propuesto por [LHS] se preservan cuando se aplican sobre la topología de Internet. Para evaluar su

comportamiento, se ha simulado el algoritmo sobre el modelo de Barabasi-Albert [BA] para diversos números de nodos n (50, 100, 500 y 1000) y valores del grado promedio d del grafo de entrada (4, 6, 8, 10 y 12). Se comparan las siguientes características:

- Grado promedio (número promedio de vecinos por nodo)
- Promedio del número máximo de vecinos.
- Grado máximo encontrado dentro de cada prueba
- Porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5

Para generar y comparar los árboles, se usaron como función de peso de los enlaces tanto el retraso en la transmisión como la distancia Euclidiana entre nodos. Las estadísticas muestran [RW], que los picos de mayor latencia en los enlaces de Internet punto-a-punto están entre 50 y 150ms. En los experimentos se asignó la métrica de retraso en la comunicación entre pares de nodos de manera aleatoria, usando una distribución de Gauss centrada alrededor de 100ms. Para cada prueba se corrieron 50 experimentos y los resultados obtenidos se muestran de la tabla 6.1 hasta la 6.8. El trazo de estos datos se muestra desde la figura 6.1 hasta la 6.5.

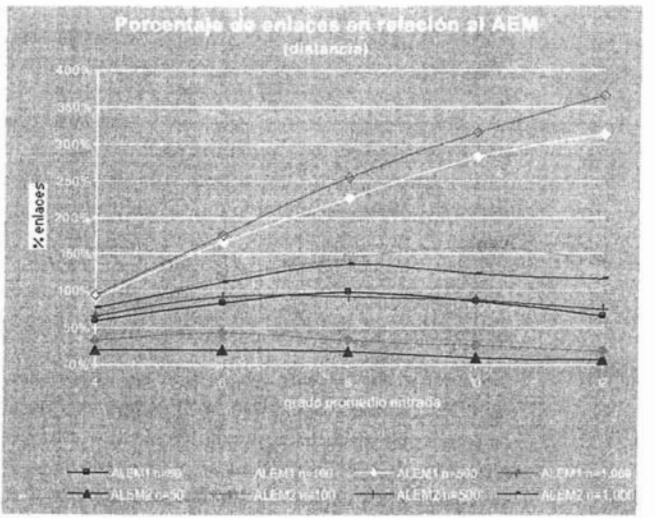


Figura 6.1. Porcentaje de incremento de los enlaces $ALEM_1$ y $ALEM_2$ con respecto al AEM usando la distancia Euclidiana como función de peso de los enlaces.

No hay que olvidar, que la familia de estructuras comprendida por $ALEM_1$ y $ALEM_2$ se plantearon con el afán de aproximarse, con un conocimiento local, a la ruta óptima de *broadcast* generada por el AEM. El incremento en el número de enlaces con respecto al AEM en ambas estructuras, se muestra en la fgura 6.1. Se puede advertir que conforme la cantidad de nodos en la red aumenta, el incremento de enlaces en las estructuras $ALEM_1$ y $ALEM_2$ también aumenta, pero puede haber entre ambos

algoritmos una diferencia porcentual mínima de 17% (n =1000 y d =4) y una máxima de 250% (con n =1000 y d =12). El desempeño del algoritmo $ALEM_2$ es preferible, porque: 1. trata de disminuir la diferencia de enlaces mientras el grado promedio de entrada d aumenta y 2. porque la diferencia en el incremento del número de enlaces con respecto al AEM es menor que su contraparte $ALEM_1$. El comportamiento de ambos algoritmos es similar cuando se usa el retraso como peso de los enlaces.

n = 50	entrada	AEM		$ALEM_1$		$ALEM_2$	
		distancia	retraso	distancia	retraso	distancia	retraso
Grado Promedio	4	1.96	1.96	3.1	3.18	2.34	2.5
Promedio del Núm. Máx. de Vecinos	18	8.2	9.8	13	13.8	8.8	10.4
Grado Máximo	20	9	13	17	17	10	14
Grado Promedio	6	1.96	1.96	3.57	3.86	2.34	2.66
Promedio del Núm. Máx. de Vecinos	21.2	7.4	9.6	10.6	12	8	10
Grado Máximo	24	9	12	12	14	9	12
Grado Promedio	8	1.96	1.96	3.9	4.5	2.3	2.79
Promedio del Núm. Máx. de Vecinos	21.8	5.8	7.6	8.6	11.6	6.2	9.2
Grado Máximo	25	6	11	9	14	7	13
Grado Promedio	10	1.96	1.96	3.62	4.58	2.15	3.02
Promedio del Núm. Máx. de Vecinos	25.4	6.8	8.4	8.8	12.2	7.2	9.6
Grado Máximo	29	8	10	10	13	8	11
Grado Promedio	12	1.96	1.96	3.26	4.14	2.104	2.71
Promedio del Núm. Máx. de Vecinos	29.6	7.4	8	9	11.4	7.4	9.2
Grado Máximo	34	10	10	11	14	10	11

Tabla 6.1. Comparación entre AEM, $ALEM_1$ y $ALEM_2$ para 50 nodos.

n = 100	entrada	AEM		$ALEM_1$		$ALEM_2$	
		distancia	retraso	distancia	retraso	distancia	retraso
Grado Promedio	4	1.98	1.98	3.33	3.41	2.65	2.78
Promedio del Núm. Máx. de Vecinos	27.2	13.2	14.8	19	20.4	14.4	15.8
Grado Máximo	32	15	17	21	22	15	19
Grado Promedio	6	1.98	1.98	4.32	4.6	2.83	3.15
Promedio del Núm. Máx. de Vecinos	30.4	9.4	11.2	15.6	18.2	11.4	12.2
Grado Máximo	35	11	13	17	21	13	14
Grado Promedio	8	1.98	1.98	4.85	5.25	2.63	3.08
Promedio del Núm. Máx. de Vecinos	32.8	7.6	10.2	14	17	8.8	10.8
Grado Máximo	39	9	12	15	19	10	14
Grado Promedio	10	1.98	1.98	5.16	5.89	2.48	3.18
Promedio del Núm. Máx. de Vecinos	37.2	8.8	8.8	13.4	16	9	11.8
Grado Máximo	41	10	10	15	17	10	15
Grado Promedio	12	1.98	1.98	4.8	5.79	2.34	3.24
Promedio del Núm. Máx. de Vecinos	40.2	7.6	10.6	12	16	8.2	13.2
Grado Máximo	43	10	15	14	18	10	16

Tabla 6.2. Comparación entre AEM, $ALEM_1$ y $ALEM_2$ para 100 nodos.

n = 500	entrada	AEM		ALEM 1		ALEM 2	
		distancia	retraso	distancia	retraso	distancia	retraso
Grado Promedio	4	1.996	1.996	3.79	3.81	3.3	3.4
Promedio del Núm. Máx. de Vecinos	65.2	28	31.2	51	52.6	36	39
Grado Máximo	85	35	41	60	62	42	45
Grado Promedio	6	1.996	1.996	5.29	5.4	3.81	4.106
Promedio del Núm. Máx. de Vecinos	67.2	21.2	26.8	45.2	48.8	28.2	32
Grado Máximo	76	30	36	56	59	38	41
Grado Promedio	8	1.996	1.996	6.48	6.71	3.82	4.32
Promedio del Núm. Máx. de Vecinos	83.6	20	24.2	49.6	53.6	26.8	32.2
Grado Máximo	103	34	35	71	71	41	45
Grado Promedio	10	1.996	1.996	7.63	8	3.73	4.38
Promedio del Núm. Máx. de Vecinos	90.4	14.4	18.6	36.8	44	18.4	26.2
Grado Máximo	101	20	21	43	49	21	31
Grado Promedio	12	1.996	1.996	8.23	8.82	3.5	4.36
Promedio del Núm. Máx. de Vecinos	95.6	14.6	21	36.4	42.6	19.4	26.8
Grado Máximo	110	18	24	41	49	22	30

Tabla 6.3. Comparación entre AEM, $ALEM_1$ y $ALEM_2$ para 500 nodos.

n = 1000	entrada	AEM		ALEM 1		ALEM 2	
		distancia	retraso	distancia	retraso	distancia	retraso
Grado Promedio	4	1.998	1.998	3.88	3.89	3.54	3.58
Promedio del Núm. Máx. de Vecinos	75.4	37.6	40	66	66.2	52.4	52.6
Grado Máximo	81	45	46	71	73	62	59
Grado Promedio	6	1.998	1.998	5.49	5.57	4.21	4.42
Promedio del Núm. Máx. de Vecinos	95.8	27.6	37.8	67.4	74.8	40.8	45.8
Grado Máximo	123	30	47	87	96	46	53
Grado Promedio	8	1.998	1.998	7.08	7.21	4.69	5.04
Promedio del Núm. Máx. de Vecinos	98.6	24.4	32	63.2	71	36	42.6
Grado Máximo	123	31	32	85	99	51	60
Grado Promedio	10	1.998	1.998	8.28	8.54	4.44	5.04
Promedio del Núm. Máx. de Vecinos	126.8	23.8	35.8	61.2	71.4	29.4	40.8
Grado Máximo	146	32	46	72	80	38	49
Grado Promedio	12	1.998	1.998	9.3	9.71	4.31	5.04
Promedio del Núm. Máx. de Vecinos	139.4	18.2	29.4	57.8	67.2	26	37.6
Grado Máximo	158	23	46	69	83	32	48

Tabla 6.4. Comparación entre AEM, $ALEM_1$ y $ALEM_2$ para 1,000 nodos.

Intentando averiguar el comportamiento del algoritmo ALEM propuesto por [LHS] sobre la topología de Internet, se trazo la figura 6.2. Observamos que una de las propiedades más importantes del algoritmo, como es el acotar el número máximo de vecinos a 6, no se preserva y tampoco lo logra el AEM, este valor depende más del grado máximo de entrada y del número de nodos del grafo original.. Pero es importante ver que entre más alto sea el grado promedio (y por tanto el promedio en el

número máximo de vecinos) del grafo de entrada, los tres algoritmos reducen más el número de enlaces, como se muestra en la figura 6.3. De cualquier manera, si la estructura óptima a la cual nos queremos acercar corresponde al AEM, entonces la estructura de árbol que más se le acerca es $ALEM_2$. Otra propiedad que se puede observar, es la dferencia de resultados cuando se emplea la distancia euclidiana o el retraso en la transmisión como parámetros de la función de peso de los enlaces. Siempre se obtuvieron valores menores para la distancia que para el retraso.

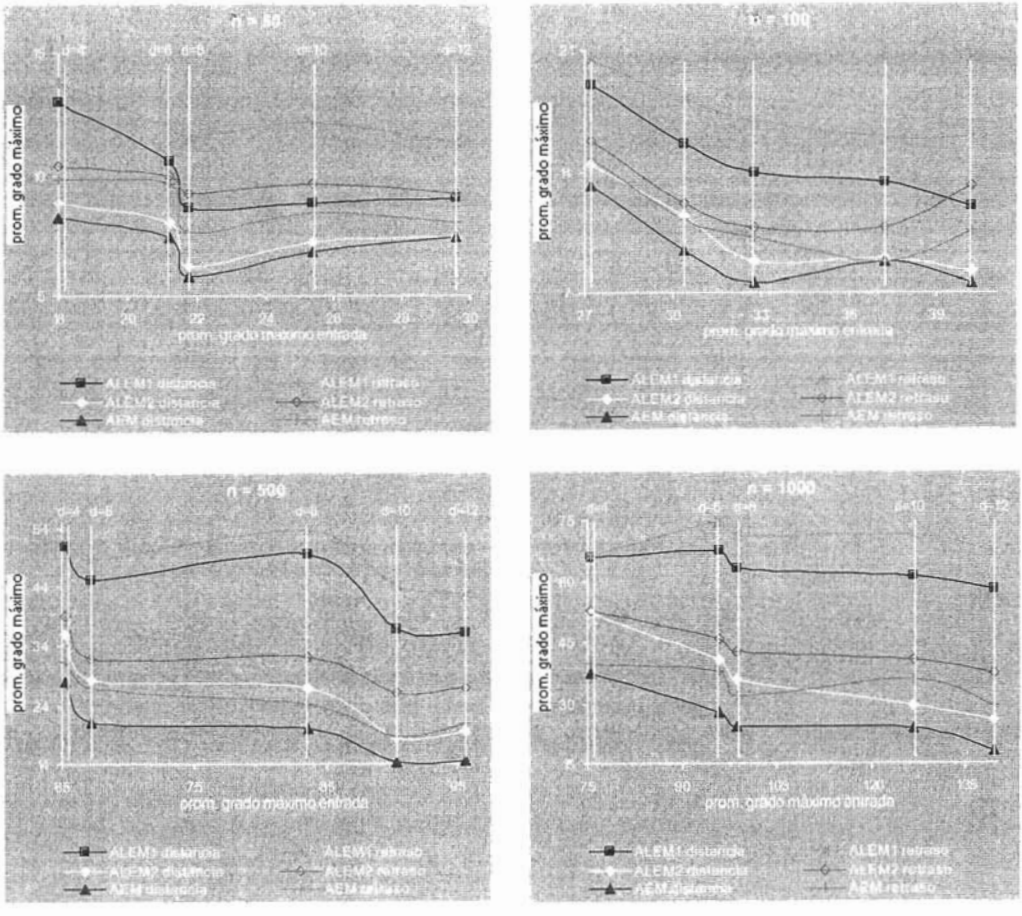


Figura 6.2. Traza el promedio de grado máximo de cada estructura, con respecto al número de nodos y el promedio del grado máximo del grafo original, para n=50, 100, 500 y 1000.

En la figura 6.3 se aprecia que la reducción en el número máximo de vecinos del grafo de entrada va incrementando mientras el grado promedio original también aumenta. Vemos, que sin importar mucho el número de nodos del grafo original, el AEM reduce desde un promedio de 53% de enlaces del grafo de entrada para d =4 hasta un 82% para d =12, para $ALEM_2$ ocurre desde un 43% hasta un 80% respectivamente y para $ALEM_1$ desde un promedio de 23% hasta un 65%; aunque si bien, el $ALEM_1$ con n=1000 obtiene una reducción ínfima de tan solo 12% para d=4 y de 58% para d=12. De nuevo observamos que el comportamiento del $ALEM_2$ es el que más se acerca al del AEM.

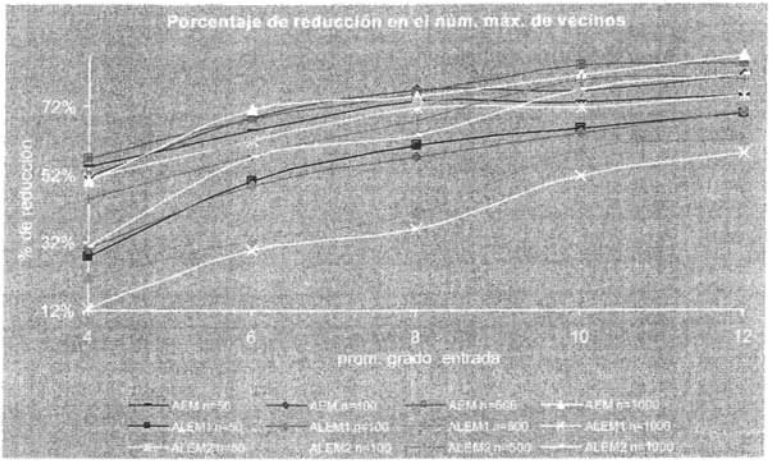


Figura 6.3. Reducción en el número máximo de vecinos del grafo de entrada usando la distancia como función de peso de los enlaces.

La figura 6.4 muestra una propiedad interesante; como se había visto anteriormente, el grado promedio de los nodos bajo el algoritmo AEM se caracteriza por tener el menor grado promedio de todos los subgrafos expandidos ($2 - \frac{2}{n} \to 2, \text{si } n \to \infty$) y vemos que el $ALEM_2$-distancia tiende a este valor mientras más aumenta la densidad de la red original y más rápido aun entre más pequeño sea el número de nodos.

La figura 6.5 es el trazo comparativo de los datos de las tablas 6.5 hasta la 6.8, usando la distancia Euclidiana como peso de los enlaces. Muestra el porcentaje promedio de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada d, y número de nodos n. En esta secuencia de trazos, es interesante ver el proceso de aproximación del $ALEM_2$ hacia el resultado del AEM. Tratando de resumir el desarrollo de esta

transformación, diremos que mientras más bajo sea el número de nodos n del grafo original y mientras más alto sea el grado promedio de entrada d, más fácil es el acoplo del $ALEM_2$ al AEM. Pudiera ser que el $ALEM_1$ quisiera seguir este mismo comportamiento, pero le tarda muchísimo más alcanzar el resultado porque vemos que la mayoría de las muestras se aproximan más al grafo original. También resalta que el algoritmo AEM siempre tiene el mismo trazo, manteniendo un grado bastante bajo (en promedio, el 80% de nodos tienen menos de 3 vecinos) independientemente del tamaño de la red o de su densidad.

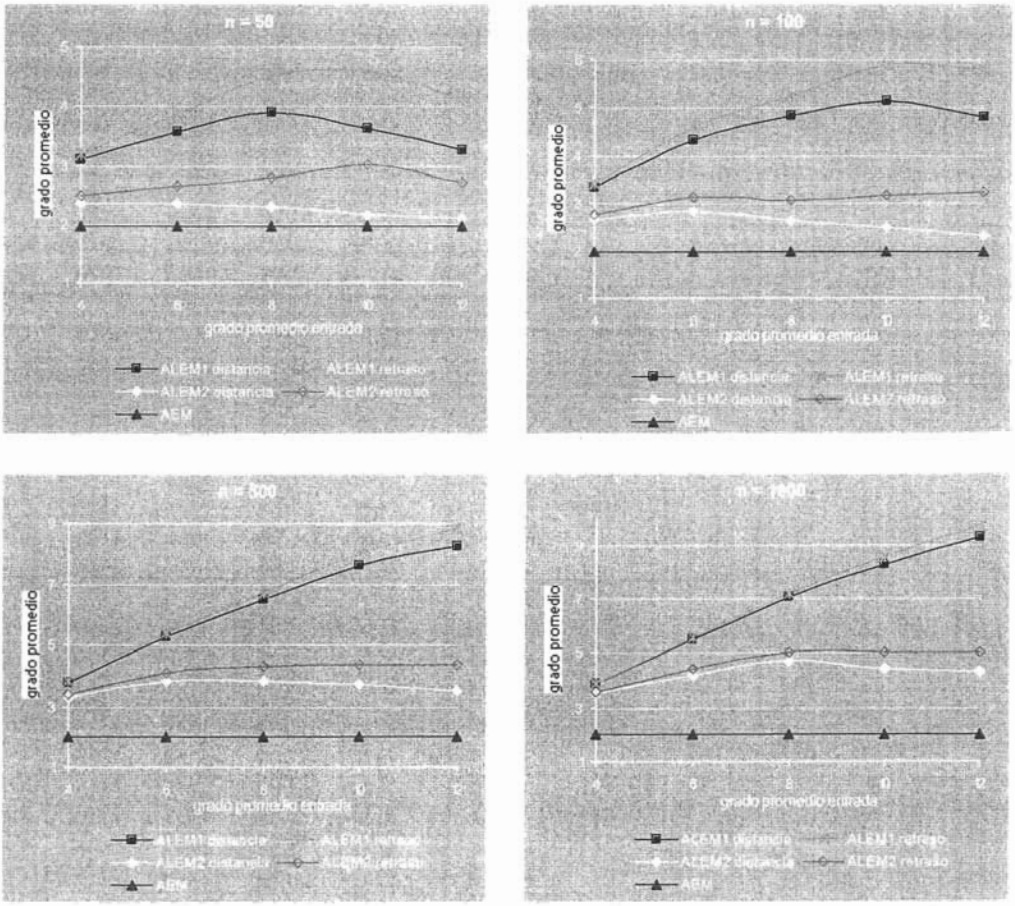


Figura 6.4. Traza el grado promedio de AEM, ALEM1 y ALEM2 en función del grado promedio del grafo de entrada, para n=50, 100, 500 y 1000.

AEM		d	1	2	3	4	5	>5
n = 50	distancia	4	55.20%	24.00%	10.00%	3.60%	2.00%	5.20%
		6	50.00%	28.80%	11.60%	1.60%	4.40%	3.60%
		8	48.80%	28.00%	11.20%	4.80%	4.80%	2.40%
		10	47.20%	29.60%	12.40%	5.60%	3.20%	2.00%
		12	48.80%	27.60%	12.80%	6.40%	1.60%	2.80%
	retraso	4	58.40%	21.20%	10.00%	4.40%	0.80%	5.20%
		6	55.20%	26.40%	8.00%	3.20%	2.80%	4.40%
		8	56.40%	20.00%	11.20%	4.00%	4.40%	4.00%
		10	53.60%	25.20%	11.20%	4.00%	1.20%	4.80%
		12	53.20%	24.40%	11.20%	4.40%	2.40%	4.40%
n = 100	distancia	4	55.40%	25.80%	8.20%	4.20%	1.40%	5.00%
		6	51.80%	27.60%	9.00%	4.60%	2.80%	4.20%
		8	48.80%	29.40%	9.60%	6.20%	2.00%	4.00%
		10	53.00%	21.60%	14.00%	5.40%	2.60%	3.40%
		12	47.60%	27.40%	14.40%	5.60%	2.40%	2.60%
	retraso	4	58.80%	24.00%	7.00%	2.60%	2.40%	5.20%
		6	55.80%	23.20%	9.60%	4.00%	2.80%	4.60%
		8	53.60%	24.80%	10.00%	3.80%	3.40%	4.40%
		10	53.40%	24.20%	12.00%	4.20%	1.20%	5.00%
		12	55.60%	22.60%	9.40%	5.80%	2.00%	4.60%
n = 500	distancia	4	56.88%	24.76%	8.40%	3.84%	1.76%	4.36%
		6	54.96%	25.20%	9.64%	3.72%	2.52%	3.96%
		8	53.84%	24.96%	10.40%	4.28%	1.80%	4.72%
		10	52.16%	26.52%	10.24%	4.44%	2.64%	4.00%
		12	51.28%	27.00%	10.56%	4.92%	2.48%	3.76%
	retraso	4	58.40%	24.08%	8.04%	2.96%	2.16%	4.36%
		6	58.20%	23.60%	8.44%	3.36%	1.84%	4.56%
		8	57.72%	23.88%	7.20%	4.32%	1.88%	5.00%
		10	55.80%	23.56%	10.32%	3.80%	2.40%	4.12%
		12	54.80%	24.20%	10.36%	4.84%	2.28%	3.52%
n = 1,000	distancia	4	55.38%	26.16%	8.66%	3.76%	1.96%	4.08%
		6	53.90%	26.70%	9.68%	3.86%	2.06%	3.80%
		8	53.68%	25.06%	10.50%	4.86%	2.12%	3.78%
		10	53.06%	26.12%	10.22%	4.78%	2.10%	3.72%
		12	52.02%	26.64%	11.26%	4.10%	2.06%	3.92%
	retraso	4	57.46%	24.82%	7.74%	4.18%	1.82%	3.98%
		6	56.88%	25.28%	8.18%	3.84%	2.04%	3.78%
		8	56.38%	23.16%	9.88%	4.08%	2.22%	4.28%
		10	56.08%	23.52%	10.10%	4.46%	2.04%	3.80%
		12	55.32%	23.74%	10.34%	4.00%	2.50%	4.10%

Tabla 6.5. Porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada *d*, usando el algoritmo AEM.

ALEM 1		d	1	2	3	4	5	>5
n = 50	distancia	4	12.80%	44.00%	18.80%	10.00%	2.40%	12.00%
		6	8.80%	22.80%	28.00%	16.40%	10.80%	13.20%
		8	6.00%	14.80%	24.00%	27.60%	11.20%	16.40%
		10	4.40%	24.00%	24.80%	20.80%	14.80%	11.20%
		12	10.00%	26.40%	24.00%	22.80%	9.20%	7.60%
	retraso	4	11.20%	45.20%	17.60%	10.00%	3.60%	12.40%
		6	6.40%	19.20%	29.20%	18.00%	11.60%	15.60%
		8	3.20%	12.00%	18.40%	27.60%	16.00%	22.80%
		10	3.20%	8.00%	23.60%	24.00%	17.20%	24.00%
		12	4.80%	14.80%	25.60%	19.20%	18.40%	17.20%
n = 100	distancia	4	10.00%	47.80%	14.80%	9.60%	5.60%	12.20%
		6	3.20%	15.00%	33.40%	18.00%	8.20%	22.20%
		8	2.00%	6.80%	20.60%	28.60%	13.20%	28.80%
		10	2.00%	7.40%	15.40%	20.60%	21.40%	33.20%
		12	2.20%	10.60%	16.80%	18.00%	20.20%	32.20%
	retraso	4	8.80%	48.20%	15.80%	8.20%	6.80%	12.20%
		6	2.40%	12.80%	33.80%	18.60%	8.20%	24.20%
		8	0.80%	4.80%	19.00%	29.40%	14.20%	31.80%
		10	0.60%	4.80%	11.60%	19.20%	21.40%	42.40%
		12	0.40%	4.20%	15.60%	16.20%	17.80%	45.80%
n = 500	distancia	4	2.84%	49.52%	19.04%	8.92%	6.28%	13.40%
		6	0.44%	6.44%	37.76%	19.28%	10.32%	25.76%
		8	0.36%	1.76%	10.28%	31.36%	15.76%	40.48%
		10	0.16%	0.80%	3.96%	12.08%	25.72%	57.28%
		12	0.00%	0.68%	2.76%	7.52%	13.80%	75.24%
	retraso	4	2.08%	50.24%	19.12%	8.84%	6.16%	13.56%
		6	0.12%	5.36%	38.40%	19.40%	10.48%	26.24%
		8	0.12%	1.24%	8.28%	32.08%	16.76%	41.52%
		10	0.04%	0.40%	3.12%	10.88%	25.36%	60.20%
		12	0.00%	0.28%	1.48%	5.60%	13.40%	79.24%
n = 1,000	distancia	4	1.34%	49.06%	19.74%	9.68%	5.78%	14.40%
		6	0.24%	4.12%	38.98%	19.02%	10.40%	27.24%
		8	0.12%	0.72%	5.62%	31.88%	18.08%	43.58%
		10	0.08%	0.26%	2.22%	9.20%	25.56%	62.68%
		12	0.02%	0.30%	1.20%	4.04%	11.10%	83.34%
	retraso	4	1.28%	49.06%	19.78%	9.58%	5.86%	14.44%
		6	0.10%	3.44%	39.00%	19.58%	10.48%	27.40%
		8	0.06%	0.56%	4.96%	31.80%	18.68%	43.94%
		10	0.06%	0.28%	1.42%	7.94%	26.28%	64.02%
		12	0.04%	0.10%	0.78%	3.18%	9.94%	85.96%

Tabla 6.6. Porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada *d*, usando el algoritmo $ALEM_1$.

ALEM 2		d	1	2	3	4	5	>5
n = 50	distancia	4	36.00%	36.40%	11.20%	7.20%	2.80%	6.40%
		6	32.40%	34.40%	19.60%	4.80%	2.80%	6.00%
		8	34.00%	32.40%	17.20%	7.60%	4.40%	4.40%
		10	39.20%	31.20%	16.00%	7.60%	3.20%	2.80%
		12	40.00%	32.40%	15.60%	7.20%	2.00%	2.80%
	retraso	4	32.80%	38.80%	10.40%	6.00%	3.60%	8.40%
		6	24.40%	36.40%	19.20%	8.00%	4.40%	7.60%
		8	23.20%	30.80%	24.40%	8.00%	5.60%	8.00%
		10	16.00%	32.00%	26.80%	10.40%	4.00%	10.80%
		12	28.40%	30.40%	17.20%	10.40%	4.80%	8.80%
n = 100	distancia	4	28.40%	38.80%	12.80%	7.60%	4.20%	8.20%
		6	22.80%	30.80%	22.60%	8.80%	6.80%	8.20%
		8	25.00%	32.80%	20.60%	10.40%	4.60%	6.60%
		10	32.80%	27.80%	20.40%	8.20%	5.60%	5.20%
		12	34.20%	30.00%	19.80%	7.60%	4.00%	4.40%
	retraso	4	26.20%	39.80%	13.20%	8.60%	3.60%	8.60%
		6	16.40%	29.00%	27.40%	11.00%	5.20%	11.00%
		8	16.00%	30.80%	23.20%	13.80%	6.40%	9.80%
		10	20.00%	26.20%	21.60%	12.40%	8.60%	11.20%
		12	16.60%	30.20%	20.20%	13.00%	8.80%	11.20%
n = 500	distancia	4	10.92%	47.00%	17.24%	8.20%	5.36%	11.28%
		6	9.04%	23.76%	30.72%	14.04%	7.24%	15.20%
		8	10.24%	22.12%	23.40%	19.12%	8.88%	16.24%
		10	12.04%	21.36%	21.76%	19.44%	10.28%	15.12%
		12	13.60%	24.36%	23.68%	14.96%	10.04%	13.36%
	retraso	4	9.44%	47.24%	17.40%	8.52%	5.40%	12.00%
		6	6.28%	21.80%	31.56%	15.56%	7.76%	17.04%
		8	5.32%	18.96%	24.32%	20.76%	10.64%	20.00%
		10	6.16%	17.24%	22.44%	20.44%	13.04%	20.68%
		12	5.52%	19.64%	21.96%	18.80%	13.20%	20.88%
n = 1,000	distancia	4	6.30%	47.68%	18.72%	9.26%	5.40%	12.64%
		6	5.26%	18.08%	33.88%	15.48%	8.86%	18.44%
		8	4.10%	13.62%	21.62%	24.68%	11.90%	24.08%
		10	6.32%	15.42%	21.08%	20.32%	14.32%	22.54%
		12	7.68%	16.64%	20.80%	18.60%	14.22%	22.06%
	retraso	4	5.80%	47.50%	19.08%	8.98%	5.80%	12.84%
		6	3.78%	16.86%	34.26%	16.22%	9.24%	19.64%
		8	2.76%	10.72%	21.80%	24.90%	13.16%	26.66%
		10	3.54%	11.60%	19.44%	20.90%	16.60%	27.92%
		12	3.62%	12.88%	18.24%	19.96%	16.10%	29.20%

Tabla 6.7. Porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada *d*, usando el algoritmo $ALEM_2$.

Entrada	d	1	2	3	4	5	>5
n = 50	4	0.00%	49.20%	18.40%	10.40%	4.40%	17.60%
	6	0.00%	0.00%	32.40%	24.00%	12.40%	31.20%
	8	0.00%	0.00%	0.00%	26.80%	21.20%	52.00%
	10	0.00%	0.00%	0.00%	0.00%	28.80%	71.20%
	12	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
n = 100	4	0.00%	51.60%	17.60%	8.60%	6.60%	15.60%
	6	0.00%	0.00%	38.20%	19.80%	10.60%	31.40%
	8	0.00%	0.00%	0.00%	31.00%	20.00%	49.00%
	10	0.00%	0.00%	0.00%	0.00%	28.80%	71.20%
	12	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
n = 500	4	0.00%	51.20%	19.28%	9.24%	6.24%	14.04%
	6	0.00%	0.00%	39.24%	20.24%	11.52%	29.00%
	8	0.00%	0.00%	0.00%	33.48%	18.72%	47.80%
	10	0.00%	0.00%	0.00%	0.00%	29.16%	70.84%
	12	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
n = 1,000	4	0.00%	49.64%	19.82%	9.84%	5.90%	14.80%
	6	0.00%	0.00%	39.88%	20.02%	10.88%	29.22%
	8	0.00%	0.00%	0.00%	33.02%	19.28%	47.70%
	10	0.00%	0.00%	0.00%	0.00%	27.64%	72.36%
	12	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%

Tabla 6.8. Porcentaje de nodos que tienen grado 1, 2, 3, 4, 5 y >5 en la topología de entrada.

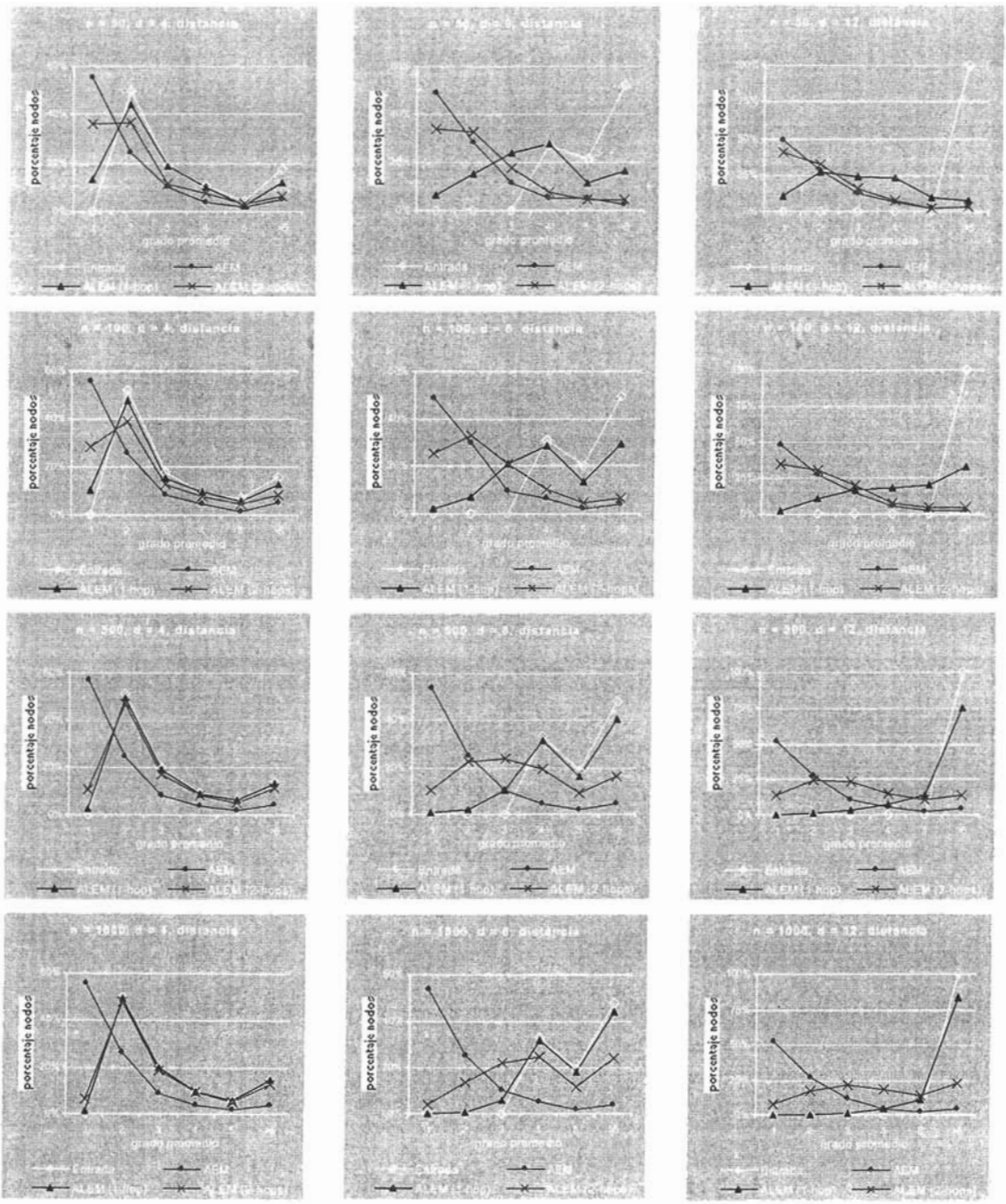


Figura 6.5. Trazos comparativos del porcentaje promedio de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada *d* y número de nodos *n*, usando la distancia como peso.

Los siguientes resultados muestran los datos de aplicar los algoritmos: $ALEM_1$, $ALEM_2$, AEM, GVR y GVRC sobre las topologías Internet generadas sintéticamente. Las características que nos interesan obtener son las siguientes:

- Grado Promedio, es el promedio de la densidad de la red en el grafo original.
- Grado promedio de $ALEM_1$, $ALEM_2$, AEM, GVR o GVRC, es la densidad promedio de la red de comunicación sobrepuesta después de aplicar el algoritmo $ALEM_1$, $ALEM_2$, AEM, GVR o GVRC sobre el grafo original.
- %Costo, es el porcentaje de la proporción entre el número de mensajes transmitidos en el proceso de *broadcasting* basado en $ALEM_1$, $ALEM_2$, AEM, GVR o GVRC y el *flooding*-doble de Gnutella. Lo que significa, el porcentaje de enlaces del grafo original que permanecen en el $ALEM_1$, $ALEM_2$, AEM, GVR o GVRC respectivamente.

Para construir las estructuras de árbol se usó la distancia Euclidiana como función de peso de los enlaces porque como mostramos anteriormente, se obtienen importantes mejorías en los resultados en comparación con el retraso en la transmisión. El valor de cualquier propiedad para un modelo y un tamaño de topología dado, será el promedio de los valores medidos para cada uno de los 20 grafos generados.

Modelo Barabasi-Albert

Para el análisis de la aplicación, los parámetros del modelo se establecieron según los siguientes valores:

$n = 50, 100, 500$ y 1000
$m = 2, 3, 4, 5$ y 6

donde n representa el número de nodos y m permite controlar el grado promedio del grafo de entrada. Los resultados del proceso de simulación son dados en las tablas 6.9 a la 6.12.

Barabasi-Albert (*flooding*-doble) n = 50										
grado	AEM		$ALEM_1$		$ALEM_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	1.96	50	3.104	79.6	2.344	59.6	3.176	81.4	2.672	68.6
6	1.96	34	3.568	61.4	2.344	40.2	3.896	67	3.104	53.4
8	1.96	25	3.896	50.6	2.296	29.8	4.472	58.4	3.424	44.6
10	1.96	20	3.624	38.2	2.152	22	4.536	47.6	3.472	36.4
12	1.96	17	3.256	28.6	2.104	18.4	4.512	39.8	3.432	30.4

Tabla 6.9. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 50 nodos.

Barabasi-Albert (*flooding*-doble) n = 100										
grado	AEM		ALEM$_1$		ALEM$_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	1.98	50	3.328	84	2.652	66.8	3.416	86	2.964	75
6	1.98	33	4.324	73	2.828	47.4	4.604	77.6	3.684	62.2
8	1.98	25	4.852	61.6	2.628	33.2	5.472	69.8	4.068	51.8
10	1.98	20	5.156	52.8	2.48	25.2	5.892	60.4	4.152	42.2
12	1.98	17	4.8	40.8	2.336	19.6	6.028	51.4	4.288	36.8

Tabla 6.10. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, ALEM$_1$, ALEM$_2$, GVR y GVRC para 100 nodos.

Barabasi-Albert (*flooding*-doble) n = 500										
grado	AEM		ALEM$_1$		ALEM$_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	2	50	3.786	94.4	3.304	82.4	3.814	95.2	3.51	87.4
6	2	33	5.29	88	3.812	63.2	5.402	90	4.536	75.2
8	2	25	6.484	81	3.822	47.4	6.762	84.4	5.324	66.4
10	2	20	7.626	76.4	3.73	37.2	8.066	80.6	5.93	59.4
12	2	16	8.23	68.6	3.494	29	8.956	74.6	6.336	52.6

Tabla 6.11. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, ALEM$_1$, y ALEM$_2$, GVR y GVRC para 500 nodos.

Barabasi-Albert (*flooding*-doble) n = 1000										
grado	AEM		ALEM$_1$		ALEM$_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	2	50	3.874	96.6	3.538	88.2	3.888	96.8	3.67	91.2
6	2	33	5.488	91	4.216	69.8	5.562	92.4	4.872	81
8	2	25	7.084	88.2	4.69	58.4	7.242	90.2	5.9	73.6
10	2	20	8.278	82.6	4.44	44	8.6	85.8	6.564	65.2
12	2	16	9.34	78	4.31	35.4	9.85	82	7.142	59

Tabla 6.12. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, ALEM$_1$, ALEM$_2$, GVR y GVRC para 1000 nodos.

Los resultados referentes al grado promedio del algoritmo ALEM$_2$ son muy interesantes. Partiendo de un grado promedio bajo del grafo de entrada (d=4) se obtiene un comportamiento esperado: a medida que d aumenta, el grado promedio del ALEM$_2$ incrementa paulatinamente también; después puede ser desconcertante observar un valor de d en donde se observa un punto de inflexión.

Lo interesante es, que a partir de este punto de inflexión, mientras d aumenta, el grado promedio del ALEM$_2$ tiende a 2, lo que corresponde al grado promedio del AEM (el valor óptimo). Lo cual significa, como se muestra en la figura 6.6, que el acoplamiento de la distribución de grado promedio del ALEM$_2$ al AEM, se establece después de este punto de inflexión y de manera paulatina conforme el grado promedio de la topología Internet d aumenta. También se observa, que mientras más nodos n tenga la topología Internet inicial, el acoplamiento es más lento conforme d aumenta. Por ejemplo, para n=500, se podría decir que el punto de inflexión se observa en d=8, a partir de ese valor de d en adelante, el ALEM$_2$ se va aproximando al AEM. En cambio, para n=50 el punto de inflexión se tiene en d=6 y a partir de ese punto el ALEM$_2$ se aproxima al

AEM hasta que en d=12 se tiene casi el mismo comportamiento. Esto es un resultado muy interesante, porque el grado promedio de Internet es bastante alto, así que este algoritmo puede tener un buen desempeño en el *broadcasting* de Internet.

Por otra parte, el costo en la transmisión de mensajes es significativamente menor en $ALEM_2$ que en su contraparte $ALEM_1$ y los algoritmos GVR y GVRC. En [CISS] se demuestra que ALEM es un subconjunto de GVR, lo cual explica porque siempre el $ALEM_1$ y $ALEM_2$ muestran un mejor desempeño. Todo lo descrito anteriormente se observa en los otros dos modelos de topologías: Palmer-Steffan y Waxman.

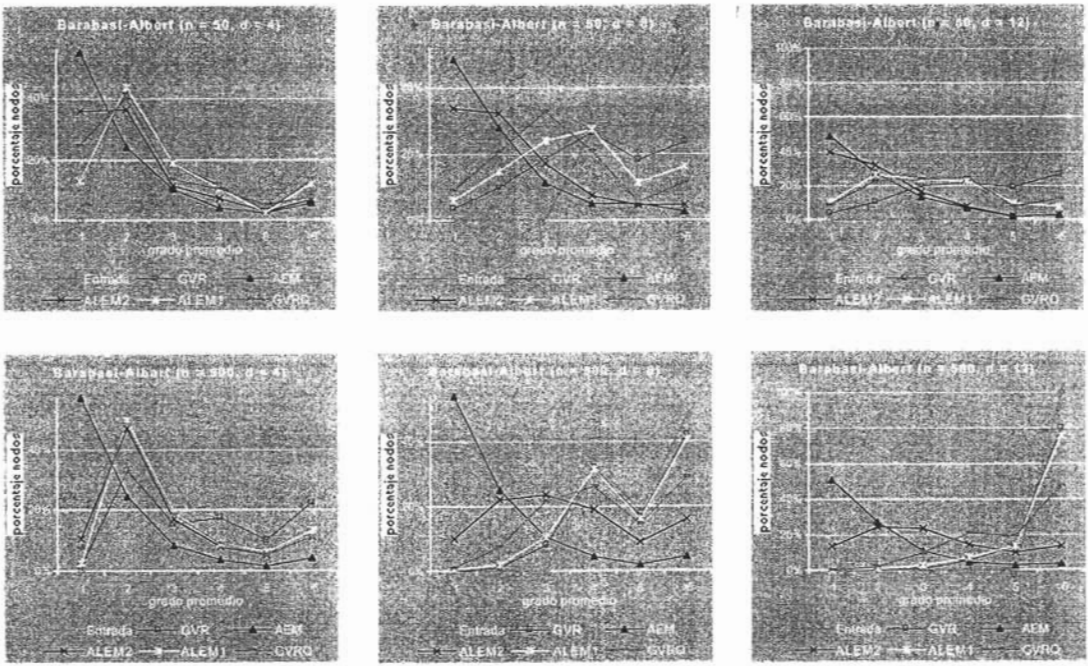


Figura 6.6. Trazos comparativos del porcentaje promedio de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada d (4, 8 y 12) y número de nodos n (50 y 500).

Modelo Palmer-Steffan

Para el análisis de la aplicación, los parámetros del modelo se establecieron según los siguientes valores:

$\alpha = \varepsilon = 0.4$,

$\beta = \gamma = 0.1$

n = 64, 128, 256 y 1024

Este algoritmo no proporciona la posición de los nodos, se ha asignado a cada nodo un conjunto de coordenadas (x, y), escogidas aleatoriamente.

Palmer-Steffan (*flooding*-doble) n = 64										
grado	AEM		ALEM 1		ALEM 2		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	1.97	49	3.671	91.4	3.08	76.8	3.693	91.7	3.236	80.8
6	1.97	32	4.672	77.3	2.974	49.2	4.833	80	3.712	61.2
8	1.97	24	4.92	61	2.73	33.6	5.408	67.1	3.964	49
10	1.97	19	4.707	46.7	2.618	25.8	5.611	55.6	3.97	39.4
20	1.97	9	3.118	15.1	2.244	10.8	4.829	23.6	3.656	17.8
30	1.97	6	2.565	8	2.092	6.2	3.869	12.3	3.132	9.8

Tabla 6.13. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 64 nodos.

Palmer-Steffan (*flooding*-doble) n = 128										
grado	AEM		ALEM 1		ALEM 2		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	1.98	49	3.62	90	3.19	79	3.64	91	3.38	84
6	1.98	33	5.02	83.22	3.63	60	5.119	84.89	4.198	69.6
8	1.98	24	5.964	74.2	3.516	43.4	6.242	77.4	4.648	57.6
10	1.98	19	6.202	61.6	3.182	31.4	6.788	67.4	4.864	48
20	1.98	9	4.652	22.8	2.57	12.6	6.91	34	4.746	23.4
30	1.98	6	3.79	12	2.406	7.4	6.21	20.4	4.4	14

Tabla 6.14. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 128 nodos.

Palmer-Steffan (*flooding*-doble) n = 512										
grado	AEM		ALEM 1		ALEM 2		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	el grafo de entrada esta desconectado									
6	2	33	5.565	92.25	4.845	80.5	5.578	92.5	5.032	83.2
8	2	24	7.106	88.4	5.236	65	7.174	89	5.92	73.6
10	2	19	8.268	82.2	5.062	50	8.45	84.2	6.46	64
20	2	9	9.724	48.2	3.852	18.8	11.75	58.2	7.376	36.4
30	2	6	8.178	26.8	3.504	11	12.072	39.8	7.298	24

Tabla 6.15. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 512 nodos.

Palmer-Steffan (*flooding*-doble) n = 1024										
grado	AEM		ALEM 1		ALEM 2		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	el grafo de entrada esta desconectado									
6	2	33	5.74	95	5.215	86.5	5.75	95	5.35	88.67
8	2	24	7.425	92.25	5.988	74.2	7.458	92.75	6.443	80
10	2	19	8.86	88	6.175	61.5	8.95	89	7.226	71.8
20	2	9	12.505	62	4.9	24	13.853	68.8	8.771	43.6
30	2	6	12.01	39.75	4.295	14	15.59	51.25	9.025	29.5

Tabla 6.16. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 1024 nodos.

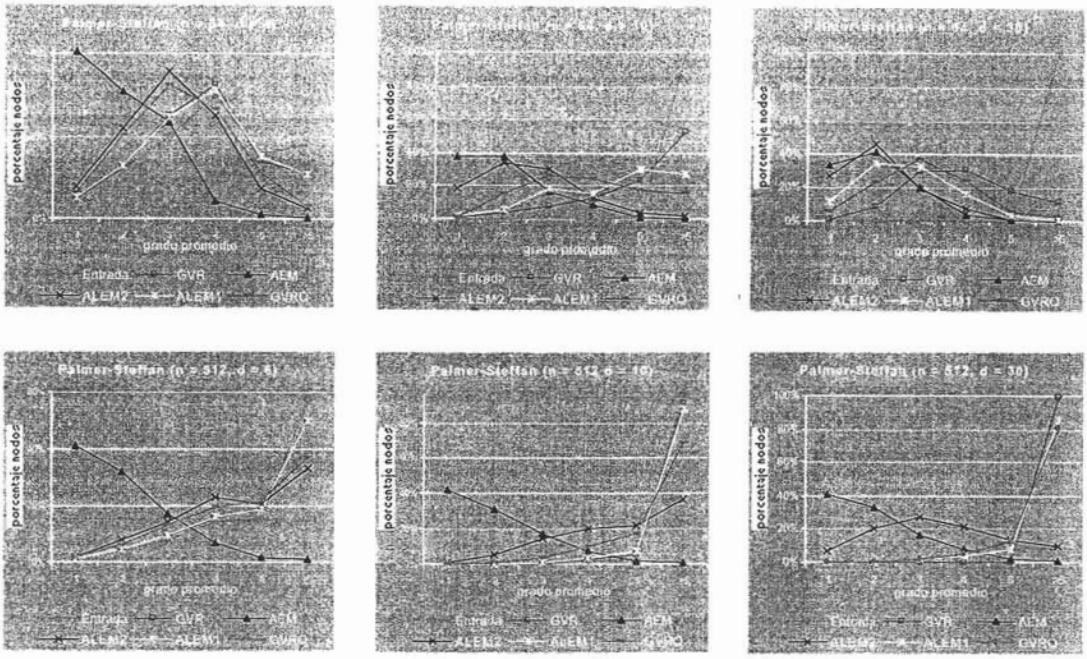


Figura 6.7. Trazos comparativos del porcentaje promedio de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada *d* (4 ó 6, 10 y 30) y número de nodos *n* (64 y 512).

Modelo Waxman

El modelo de Waxman fue uno de los primeros modelos empleados para la generación de topologías de red. Su innovación consistió en proponer que la conexión entre los nodos no fuera aleatoria, definiendo una función de probabilidad que toma en cuenta la distancia entre el par de nodos. Si bien no es posible modelar topologías Internet ya que no puede reproducir las propiedades de las leyes de potencia, ofrece una buena referencia cuando se evalúan otros modelos.

El valor de los parámetros para generar los grafos en nuestros experimentos fueron:

$\alpha = 0.2$
$\beta = 0.2$
$n = 50, 100, 500$ y 1000
$m = 2, 3, 4, 5, 10$ y 15

De acuerdo con [MP] y [ZCD], una combinación de $\alpha = 0.2$ y β en el rango [0.15, 0.2] da como resultado una topología con un adecuado número de enlaces y de diámetro para ser comparada con otros modelos.

Waxman (*flooding*-doble) n = 50										
grado	AEM		$ALEM_1$		$ALEM_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	1.96	49	3.516	87.9	2.768	69.2	3.532	88.3	3.15	75
6	1.96	32	4.148	68.9	2.488	41.2	4.384	72.8	3.376	50
8	1.96	24	4.268	53.1	2.352	29	4.744	59.1	3.424	42.4
10	1.96	19	3.72	36.9	2.104	20.6	4.496	44.7	3.432	34.2
20	1.96	9	2.2	10.5	1.976	9.4	3.008	14.6	2.56	12.6
30	1.96	6	2.108	6.8	1.968	6	2.872	9.2	2.345	7.4

Tabla 6.17. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 50 nodos.

Waxman (*flooding*-doble) n = 100										
grado	AEM		$ALEM_1$		$ALEM_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	1.98	49	3.666	91.3	3.18	79.4	3.678	91.6	3.356	83.6
6	1.98	33	4.818	79.9	3.26	54.2	4.922	81.9	3.976	66
8	1.98	24	5.324	66.2	2.872	35.8	5.682	70.6	4.22	52.6
10	1.98	19	5.504	54.6	2.584	25.4	6.06	60.2	4.344	43
20	1.98	9	3.414	16.5	2.104	10	5.116	25.2	3.72	18.2
30	1.98	6	2.476	7.7	2.016	6	3.788	12.2	3.032	9.8

Tabla 6.18. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 100 nodos.

Waxman (*flooding*-doble) n = 500										
grado	AEM		$ALEM_1$		$ALEM_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	2	49	3.897	97	3.668	91.4	3.902	97.1	3.72	92.6
6	2	33	5.644	93.6	4.72	78.4	5.664	94	5.05	83.6
8	2	24	7.116	88.4	4.996	62	7.212	89.8	5.896	73.2
10	2	19	8.32	83	4.95	49	8.506	84.6	6.582	65.4
20	2	9	11.02	54.6	3.21	15.6	12.384	61.6	7.376	36.4
30	2	6	9.675	32	2.658	8.2	12.53	41	7.22	23.8

Tabla 6.16. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 500 nodos.

Waxman (*flooding*-doble) n = 1000										
grado	AEM		$ALEM_1$		$ALEM_2$		GVR		GVRC	
promedio	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo	grado prom.	%costo
4	2	49	3.944	98	3.808	94.8	3.942	98	3.838	95.6
6	2	33	5.789	96	5.158	85.4	5.8	96	5.356	89.2
8	2	24	7.472	93	5.938	73.6	7.528	93.6	6.53	81
10	2	19	9.032	90	6.182	61.2	9.096	90.4	7.223	71.8
20	2	9	14.06	70	4.395	21.5	14.905	74	9.087	45
30	2	6	15.155	50	3.35	10.5	17.245	57	9.205	30

Tabla 6.16. Costo relativo de *flooding*-doble con el *broadcasting* basado en AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC para 1000 nodos.

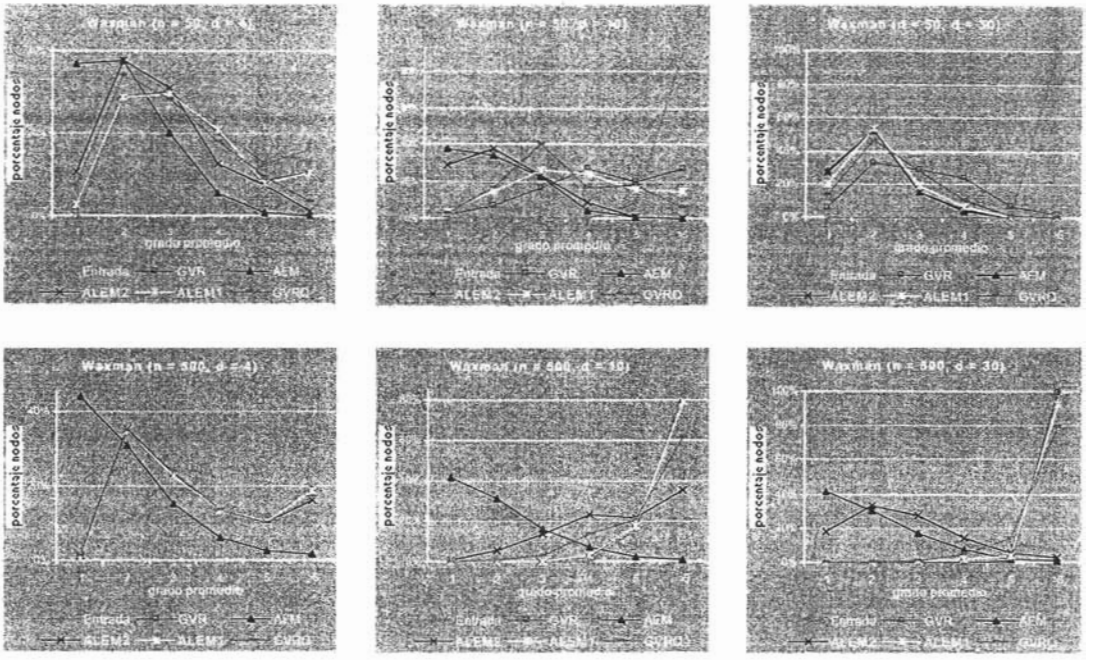


Figura 6.8. Trazos comparativos del porcentaje promedio de nodos que tienen grado 1, 2, 3, 4, 5 y >5 para diversos grados de entrada *d* (4, 10 y 30) y número de nodos *n* (50 y 500).

No es de extrañarse que el costo del algoritmo GVRC es menor que el correspondiente al $ALEM_1$ ya que para el primero la información local incluye hasta los vecinos de 2-*hops* y para el segundo únicamente hasta 1-*hop*. Y como en la construcción de su estructura esta incluido el algoritmo GVR, es normal obtener un costo menor también que este.

Los resultados mostrados para los 3 modelos de generación de topologías Internet tienen diferencias importantes, lo que podría mostrar que los algoritmos $ALEM_1$ y $ALEM_2$ son dependientes del tipo de topología, mostrando un mejor desempeño de costo para el modelo Barabasi-Albert. De cualquier manera, se puede observar en los tres modelos un dato consistente, a medida que la densidad de la red aumenta el costo de los algoritmos disminuye.

Desempeño de los protocolos $ALEM_k$ y Gossip en redes *peer-to-peer*.

En [PS], el protocolo Gossip es presentado como una alternativa al método *blind-flooding*. En esta sección mostramos una comparación entre AEM, $ALEM_1$, $ALEM_2$, GVR, GVRC y Gossip midiendo las métricas de desempeño: alcance y costo, que se han definido previamente en el capitulo 5. La tabla 6.17 muestra la comparación de los costos con respecto al *flooding* (doble) de Gnutella representando el 100%. Dado que los métodos AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC garantizan el alcance de todos los nodos de la red pero Gossip no, entonces para hacer una comparación más apropiada, se muestran algunas combinaciones de los parámetros B y F que emplea Gossip. Los datos de esta tabla son el resultado de los diferentes modelos sobre topologías del modelo Barabasi-Albert con n nodos, donde n=1000 y grado promedio d, donde $d = 4$, 6, 8, 10, 16 y 20. Los valores referentes al número de nodos n y grado promedio del grafo de entrada d, fueron seleccionados para cumplir con la información recolectada con 5 diferentes muestras de la topología Gnutella en [J].

Se puede observar que el método $ALEM_1$ y GVR tienen un comportamiento muy parecido entre si y conforme la densidad de la red aumenta presentan un peor desempeño respecto a Gossip. Como es de esperarse, AEM presenta en todo momento una ruta de *broadcasting* superior a Gossip. El método $ALEM_2$ muestra un ahorro mayor cuando Gossip abarca, durante su *broadcasting*, a más nodos de la red.

Barabasi-Albert (*flooding*-doble) n = 1000													
grado		Gossip		AEM	$ALEM_1$	$ALEM_2$	GVR	GVRC	AEM	$ALEM_1$	$ALEM_2$	GVR	GVRC
promedio	B F	%alcance	%costo	%costo	%costo	%costo	%costo	%costo	Gossip	Gossip	Gossip	Gossip	Gossip
4	2 2	82.3	53.4	50	96.6	88.2	96.8	91.2	0.94	1.81	1.65	1.81	1.71
4	3 1	76.3	48.7	50	96.6	88.2	96.8	91.2	1.03	1.98	1.81	1.99	1.87
4	3 2	93.4	69.2	50	96.6	88.2	96.8	91.2	0.72	1.40	1.27	1.40	1.32
6	2 2	91.8	51.2	33	91.0	69.8	92.4	81.0	0.64	1.78	1.36	1.80	1.58
6	3 1	87.1	46.3	33	91.0	69.8	92.4	81.0	0.71	1.97	1.51	2.00	1.75
6	3 2	97.7	66.9	33	91.0	69.8	92.4	81.0	0.49	1.36	1.04	1.38	1.21
8	2 2	92.8	41.6	25	88.2	58.4	90.2	73.6	0.60	2.12	1.40	2.17	1.77
8	3 1	88.3	38.0	25	88.2	58.4	90.2	73.6	0.66	2.32	1.54	2.37	1.94
8	3 2	98.7	61.1	25	88.2	58.4	90.2	73.6	0.41	1.44	0.96	1.48	1.20
10	2 2	92.3	36.4	20	82.6	44.0	85.8	65.2	0.55	2.27	1.21	2.36	1.79
10	3 1	88.0	29.5	20	82.6	44.0	85.8	65.2	0.68	2.80	1.49	2.91	2.21
10	3 2	98.9	56.9	20	82.6	44.0	85.8	65.2	0.35	1.45	0.77	1.51	1.15
16	2 2	91.3	21.7	12	67.3	17.4	74.0	41.4	0.55	3.10	0.80	3.41	1.91
16	3 1	86.4	17.4	12	67.3	17.4	74.0	41.4	0.69	3.87	1.00	4.25	2.38
16	3 2	98.8	38.4	12	67.3	17.4	74.0	41.4	0.31	1.75	0.45	1.93	1.08
20	2 2	91.0	17.1	10	58.0	14.2	67.0	38.3	0.58	3.39	0.83	3.92	2.24
20	3 1	86.8	13.8	10	58.0	14.2	67.0	38.3	0.72	4.20	1.03	4.86	2.78
20	3 2	98.5	30.3	10	58.0	14.2	67.0	38.3	0.33	1.91	0.47	2.21	1.26

Tabla 6.17. Comparación de costos y alcance usando la distancia como función de peso de los enlaces. Todos los métodos son comparados usando el *flooding*-doble de Gnutella

En la tabla 6.18 se presenta una comparación entre Gossip-simple y los métodos de *broadcasting* AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC con respecto al *flooding*-simple. El costo de los protocolos Gossip-simple y Gossip-doble fueron muy similares porque su desempeño lo dictamina mayormente los parámetros *B* y *F*. Esta implementación de Gossip (Gossip-simple) es la que creemos es sugerida en [PS], además es la que toma en cuenta todos los conceptos presentados en [LMM], permitiendo a cada nodo mantener una lista de vecinos de los cuales haya recibido algún mensaje, porque en este caso estos nodos no necesitan recibir una nueva copia del mensaje *m*. Los datos del protocolo Gossip-simple que se muestran en la tabla 6.18 fueron obtenidos por nuestro simulador.

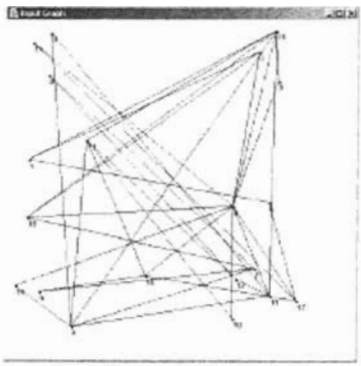
De los datos de la tabla 6.18 podemos ver que de manera general, que para cualquier valor de *d*, los esquemas $ALEM_1$, GVR y GVRC brindan un costo mayor que el protocolo Gossip, mientras que sucede lo contrario para $ALEM_2$. Esto es muy importante, porque la topología de la red Gnutella es arbitraria y con varios grados de conectividad entre los *peers* [KGZ], aunque el grado máximo de un nodo que se presenta en la red, según [SGG] es de 20 conexiones.

El algoritmo $ALEM_2$ es altamente competitivo y se visualiza como un método interesante de *broadcasting* para los sistemas *peer-to-peer*, con la característica extra de abarcar el 100% de los nodos de la red.

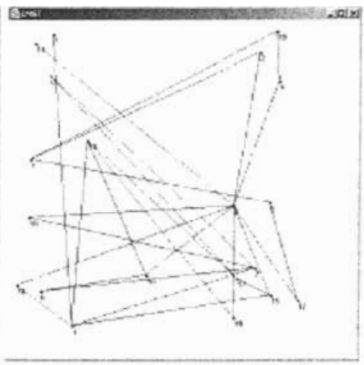
Barabasi-Albert (*flooding*-simple) n = 1000													
grado	Gossip			AEM	$ALEM_1$	$ALEM_2$	GVR	GVRC	AEM	$ALEM_1$	$ALEM_2$	GVR	GVRC
promedio	B F	%alcance	%costo	%costo	%costo	%costo	%costo	%costo	Gossip	Gossip	Gossip	Gossip	Gossip
4	2 2	84.4	51.7	33.3	64.4	58.8	64.5	60.8	0.64	1.25	1.14	1.25	1.18
4	3 1	81.3	49.4	33.3	64.4	58.8	64.5	60.8	0.67	1.30	1.19	1.31	1.23
4	3 2	95.5	62.0	33.3	64.4	58.8	64.5	60.8	0.54	1.04	0.95	1.04	0.98
6	2 2	93.4	48.3	19.8	54.6	41.9	55.4	48.6	0.41	1.13	0.87	1.15	1.01
6	3 1	90.1	45.6	19.8	54.6	41.9	55.4	48.6	0.43	1.20	0.92	1.22	1.07
6	3 2	98.8	56.5	19.8	54.6	41.9	55.4	48.6	0.35	0.97	0.74	0.98	0.86
8	2 2	93.2	39.4	14.3	50.4	33.4	51.5	42.1	0.36	1.28	0.85	1.31	1.07
8	3 1	90.3	37.8	14.3	50.4	33.4	51.5	42.1	0.38	1.33	0.88	1.36	1.11
8	3 2	99.1	52.1	14.3	50.4	33.4	51.5	42.1	0.27	0.97	0.64	0.99	0.81
10	2 2	94.5	35.1	11.1	45.9	24.4	47.7	36.2	0.32	1.31	0.70	1.36	1.03
10	3 1	89.6	30.0	11.1	45.9	24.4	47.7	36.2	0.37	1.53	0.81	1.59	1.21
10	3 2	99.4	48.1	11.1	45.9	24.4	47.7	36.2	0.23	0.95	0.51	0.99	0.75
16	2 2	92.8	17.4	6.4	35.9	9.3	39.5	22.1	0.37	2.06	0.53	2.27	1.27
16	3 1	87.4	13.8	6.4	35.9	9.3	39.5	22.1	0.46	2.60	0.67	2.86	1.60
16	3 2	99.2	30.6	6.4	35.9	9.3	39.5	22.1	0.21	1.17	0.30	1.29	0.72
20	2 2	92.2	14.3	5.3	30.5	7.5	35.3	20.2	0.37	2.13	0.52	2.47	1.41
20	3 1	86.8	11.4	5.3	30.5	7.5	35.3	20.2	0.46	2.68	0.66	3.09	1.77
20	3 2	98.9	25.1	5.3	30.5	7.5	35.3	20.2	0.21	1.22	0.30	1.40	0.80

Tabla 6.17. Comparación de costos y alcance usando la distancia como función de peso de los enlaces. Todos los métodos son comparados usando el *flooding*-simple de Gnutella

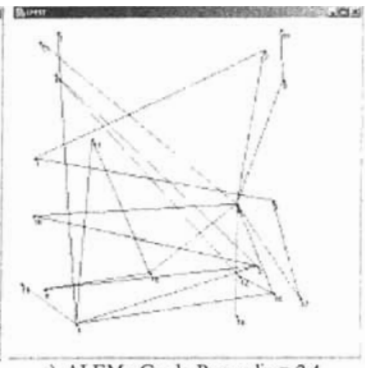
La secuencia de grafos en las figuras 6.18 y 6.19 muestran ejemplos gráficos de las estructuras AEM, $ALEM_1$, $ALEM_2$, GVR y GVRC con diferente grado promedio de entrada e igual número de nodos.



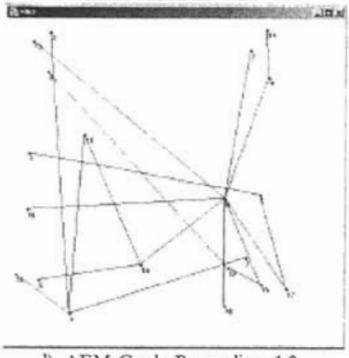
a) Topología Internet: nodos = 20, Grado Promedio = 3.7



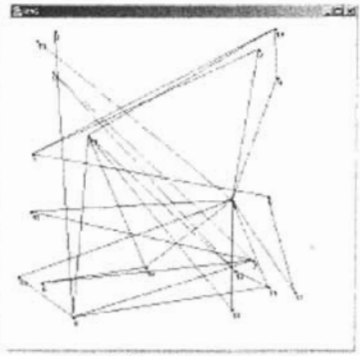
b) ALEM$_1$: Grado Promedio = 2.8



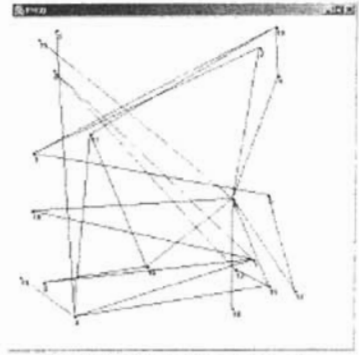
c) ALEM$_2$: Grado Promedio = 2.4



d) AEM: Grado Promedio = 1.9

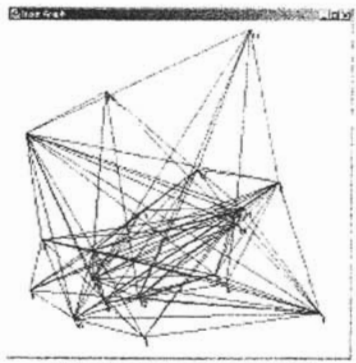


e) GVR: Grado Promedio = 2.9

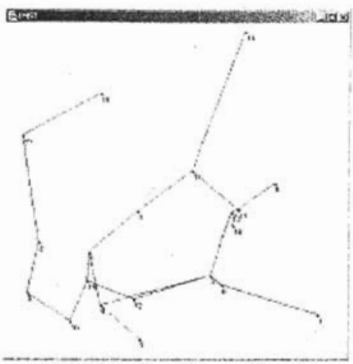


f) GVRC: Grado Promedio = 2.6

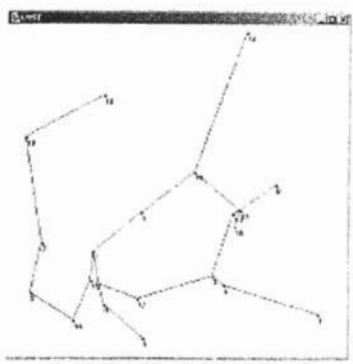
Figura 6.18. Diferencias gráficas entre a) la topología inicial con 37 enlaces y 20 nodos, b) ALEM$_1$ con 9 enlaces menos que la entrada, c) ALEM$_2$ con 13 enlaces menos, d) AEM con 18 enlaces menos, e) GVR con 8 enlaces menos y f) GVRC con 11 enlaces menos. Usando la distancia geográfica entre nodos como función de peso.



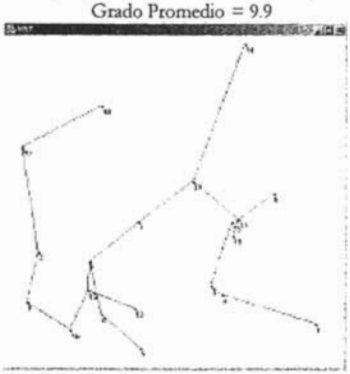
a) Topología Internet: nodos = 20, Grado Promedio = 9.9



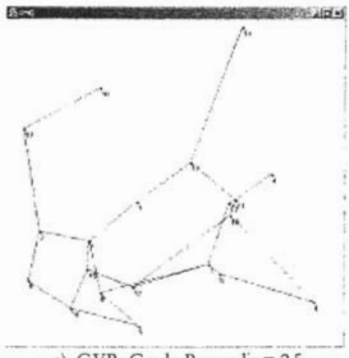
b) $ALEM_1$: Grado Promedio = 2.1



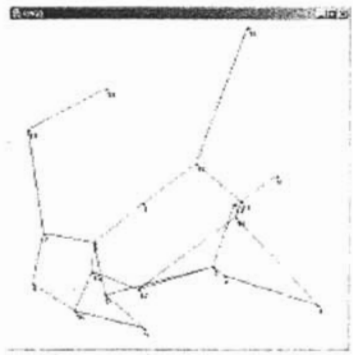
c) $ALEM_2$: Grado Promedio = 2



d) AEM: Grado Promedio = 1.9



e) GVR: Grado Promedio = 2.5



f) GVRC: Grado Promedio = 2.5

Figura 6.19. Diferencias gráficas entre a) la topología inicial con 99 enlaces y 20 nodos, b) $ALEM_1$ con 78 enlaces menos que la entrada, c) $ALEM_2$ con 79 enlaces menos, d) AEM con 80 enlaces menos, e) GVR con 74 enlaces menos y f) GVRC con 74 enlaces menos. Usando la distancia geográfica entre nodos como función de peso.

Capítulo 7

Conclusiones

Los datos experimentales muestran, que los algoritmos locales pueden ser altamente competitivos con aquellos globales.

Hemos introducido un esquema de *broadcasting* adoptando el concepto de algoritmos locales para aproximarnos a la ruta óptima de *broadcast* generada por un Árbol de Expansión Mínima. En una red donde la naturaleza de la topología es dinámica (*peers* que se unen y dejan la red inesperadamente) y carente de una autoridad central, es deseable, para ser menos susceptible al impacto de la movilidad, que cada nodo tome decisiones con base en la información que recolecte localmente a través de sus nodos vecinos.

La secuencia de estructuras $ALEM_k$ propuestas en este trabajo y principalmente $ALEM_2$, tienen el merito de acercarse al número mínimo de enlaces posible que une al conjunto de nodos de una red tomando en cuenta las características (número de nodos, grado promedio, etc) que establecen las aplicaciones de los sistemas *peer-to-peer* e Internet.

Los resultados referentes al algoritmo $ALEM_2$ muestran un comportamiento para tomar en cuenta dentro de la familia de estructuras $ALEM_k$. Existe un valor del grado promedio de entrada d, en donde se observa un punto de inflexión en el grado promedio del $ALEM_2$. Lo que notamos es que antes de este punto, $ALEM_2$ tiene una distribución de grado parecida más al grafo de entrada que a AEM -pero aún bajo esta circunstancia, el costo de transmitir mensajes de *broadcasting* usando $ALEM_2$ es más bajo que para GVR, GVRC y Gossip.

Lo interesante es que a partir de este punto de inflexión, el grado promedio del $ALEM_2$ tiende a 2 y su distribución de grado corresponde más a AEM. Esto es un resultado muy interesante, porque deja ver que este algoritmo se desempeña cuasi-AEM en redes con una densidad alta como lo es Internet y las redes *peer-to-peer*.

Por otra parte, el costo en la transmisión de mensajes es significativamente menor en $ALEM_2$ que en su contraparte $ALEM_1$, GVR, GVRC y Gossip. En esto hay un dato interesante porque los algoritmos $ALEM_2$ y GVRC al tratar de construir un grafo de

comunicación óptimo usan ambos la información local restringida por una vecindad de 2-*hops*.

Los datos experimentales han mostrado que los algoritmos construidos con un conocimiento local del entorno pueden ser competitivos con aquellos globales (AEM), aunque no es de sorprenderse el ver que los mejores algoritmos son globalizados ya que se puede efectuar una mejor decisión con un conocimiento pleno de la red que con una vista local.

Trabajo futuro

Continuar la investigación para la familia de estructuras $ALEM_k$ y verificar para $k>2$ si es necesario seguir expandiendo el entorno que conforma el conocimiento local de cada nodo a fin de valorar su aproximación al AEM y evaluar el costo que implicaría la recopilación de la información.

Simular la conformación dinámica de un sistema *peer-to-peer* y diseñar, si fuera necesario, la variante, dentro de la familia de estructuras $ALEM_k$, necesaria para competir con esquemas probados en ambientes *ad-hoc*.

Diseñar el protocolo que habilite llevar a la práctica el algoritmo $ALEM_2$ sobre un cliente *peer-to-peer* dentro de la red Gnutella y verificar la aplicación efectuando búsquedas de contenidos dentro del sistema.

El contenido de esta investigación necesitará darse a conocer mediante la redacción de un artículo que someta a juicio, el desempeño de la familia de estructuras $ALEM_k$ propuesta en este trabajo de investigación versus la generación del grafo GVRC que ha sido planteado y editado recientemente.

Referencias

[ABKM] D. Andersen, H. Balakrishnan, M. F. Kaashoek, R. Morris, *The case for resilient overlay networks*, Proc. 18th ACM SOSP, Banff, Canada, October 2001.

[CRZ] Y. Chu, S. Rao, H. Zhang, *A case for end-system multicast*, Proceedings of ACM Sigmetrics, Santa Clara, CA, June 2000.

[P] R. Prim, *Shortest connection networks and some generalizations*, The Bell System Technical Journal, vol. 36, pp. 1389-1401, 1957.

[LBCJ] J. Lipman, P. Boustead, J. Chicharo, J. Judge, *Optimized flooding algorithms for ad hoc networks, DSPCS'03&WTTSP'03*, University of Wollongong, Wollongong, pp.330-336.

[JO] E. H. Jennings, C. M. Okino, *Topology control for efficient information dissemination in ad-hoc networks*, 2002 Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2002), San Diego, CA, July 14-18, 2002.

[LWS] X. Li, Y. Wang, W. Song, *Applications of k-local MST for topology control and broadcasting in Wireless ad hoc networks*, IEEE Transaction on Parallel and Distributed Systems (IEEE TPDS), vol. 15, no.12, Dec. 2004.

[LHS] N. Li, J.C. Hou y L. Sha, *Design and analysis of an MST-based topology control algorithm*, Proc. IEEE INFOCOM, 2003.

[EGHK] D. Estrin, R. Govindan, J. Heidemann, S. Kumar; *Next century challenges: Scalable coordination in sensor networks*, Proc. MOBICOM, 1999, Seattle, pp. 263-270.

[MS] C. Monma, S. Suri, *Transitions in geometric minimum spanning trees*, Proc. ACM Symposium on Computational Geometric, North Conway, New Hampshire, United States, 1991, pp. 239-249.

[EPSS] O. Escalante, T. Pérez, J. Solano, I. Stojmenovic, *RNG-based searching and broadcasting algorithms over Internet graphs and peer-to-peer computing systems*, 3rd ACS/IEEE Int. Conf. On Computer Systems and Applications, Cairo, Egypt, Jan. 3-6, 2005.

[SB] P. Santi, D. Blough, *The critical transmitting range for connectivity in sparse wireless ad hoc networks*, IEEE Transactions on Mobile Computing, 2, 1, 1-15, 2003.

[FFF] M. Faloutsos, P. Faloutsos y C: Faloutsos, *On power-law relationships of the internet topology*, SIGCOMM, 1999, PP. 251-262.

[H] B. Hayes. *Graph Theory in Practice: Part I*, American Scientist magazine, Volume 88, Number 1, January-February 2000, pp. 9-13.

[S] S. H. Strogatz, *Exploring complex networks*, Nature, vol.410, March 2001.

[YJB] S. Yook, H. Jeong, A. Barabàsi, *Modeling the Internet's Large-Scale Topology*, Proceedings of the National Academy of Sciences 99, 13382-13386 (2002).

[FDBV] I. J. Farkas, I. Derényi, A. Barabási, T. Vicsek, *Spectra of "real-world" graphs: Beyond the semicircle law*, Phys. Rev. E 64, 026704

[SSK] R. Siamwalla, R. Sharma, S. Keshav, *Discovering Internet Topology*, Technical report, Cornell University Computer Science Department, July 1998.

[MaP] D. Magoni and J. J. Pansiot, *Evaluation of Internet topology generators by power law and distance indicators*, 10th IEEE International Conference On Networks, Singapour, pp. 401-406

[AJB] R. Albert, H. Jeong, A. Barabasi, *Error and attack tolerance of complex networks*, Nature vol. 406 num. 6794, pp. 378-382, 2000.

[KKRRT] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, *The Web as a graph: measurements, models, and methods*, International Conference on Combinatorics and Computing, 1999.

[TGJSW] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Schenker, W. Willinger, *Network Topology Generators: Degree-Based vs Structural*, in Proc. of ACM SIGCOMM, 2002.

[BT] T. Bu, D. Towsley, *On Distinguishing between Internet Power Law Topology Generators*, Proc. of IEEE/INFOCOM'02, Dec. 2002.

[CDZ] K. L. Calvet, M. B. Doar y E. W. Zegura, *Modeling Internet Topology*, IEEE Transactions on Communications, pp. 160-163, December 1997.

[GT] R. Govindan, H. Tangmunarunkit, *Heuristics for Internet Map Discovery*, Proceedings of IEEE INFOCOM'00, Tel Aviv, Israel, March 2000.

[TR] W. Theilmann, K. Rothermel, *Dynamic distance maps of the Internet*, Proceedings of IEEE INFOCOM'00, March 2000.

[NLANR] National Laboratory for Applied Network Research, *http://moat.nlanr.net/rawdata/*

[G] L. Gao, *On Inferring Autonomous System Relationships in the Internet*, IEEE Global Internet, IEEE/ACM Transactions on Networking, v.9 n.6, p.733-745, December 2001.

[CCGJSW] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. J. Shenker, W. Willinger, *The origin of Power-Laws in Internet Topologies Revisited*. Proc. IEEE INFOCOM, June 2002.

[BC] A. Broido, K. Claffy, *Internet topology: connectivity of IP graphs*, Proceeding of SPIE ITCom WWW Conf. (2001).

[CM] K. C. Claffy, D. McRobb, *Measurement and Visualization of Internet Connectivity and Performance*, Workshop on Passive and Active Measurements. Amsterdam, The Netherlands. 2001.

[MMB] A. Medina, I. Matta, J. Byers, *On the Origin of Power-laws in Internet Topologies*. ACM Computer Communication Review, pages 160-163, April 2000.

[MaPa] D. Magoni y J. J. Pansiot, *Analysis and Comparison of Internet Topology Generators*, *NETWORKING'02 - 2nd IFIP International Networking Conference*, Lecture Notes in Computer Science vol. 2345, pp. 364-375, May 19-24, 2002, Pisa, Italy.

[D] A. B. Downey. *Using pathchar to Estimate Link Characteristics*. Proceedings of the ACM SIGCOMM, 1999.

[LB] K.Lai, M. G. Baker, Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. Proceedings of the ACM SIGCOMM, 2000.

[MagP] D. Magoni, J. J. Pansiot, *Analysis of the Autonomous System Network Topology*, Computer Communication Review, 31(3):26-27, July 2001.

[PaS] C. Palmer, J. Steffan, *Generating Network Topologies that Obey Power Laws*, IEEE Globecom 2000, San Francisco, CA, November 2000.

[MP] D. Magoni and J. J. Pansiot, *Internet topology modeler based on map sampling*. ISCC 2002. IEEE Symposium on Computers and Communications, pp. 1021-1027, July 1 -4, 2002, Taormina, Italy.

[CJPP] S. Chakrabarti, M. M. Joshi, K. Punera, D. M. Pennock, *The structure of Broad Topics on the Web*, Proc. 11th International World Wide Web Conference, pages 251--262, New York, NY, 2002. ACM Press.

[AB] R. Albert, A. Barabàsi, *Topology of Evolving Networks: Local Events and Universality*, Physical Review Letters, vol. 85, pp. 5234-5237, 2000.

[SS] J. Spencer, L. Sacks, *Modeling IP Network Topologies by Emulating Network Development Processes*, IEEE International Conference on Software, Telecommunications and Computer Networks (2002).

[ZCD] E. W. Zegura, K. L. Calvert, M. J. Donahoo, *A Quantitative Comparison of Graph-based Models for Internet Topology*. IEEE ACM Transactions on Networking, 5(6): 770-783, December 1997.

[W] B. Waxman, *Routing of Multipoint Connections*. IEEE Journal of Selected Areas in Communications, 6(9): 1617-1622, December 1988.

[B] B. Bollobás, *Random Graphs*. Academic Press, Inc.,Orlando, Florida 1985.

[Do] M. Doar, *A Better Model for Generating Test Networks*, Proceeding of IEEE Global Telecommunications Conference (GLOBECOM), November 1996.

[BA] A. Barabàsi, R. Albert, *Emergence of Scaling in Random Networks*, Science 286, pp. 509-512, Oct. 1999.

[CZF] D. Chakrabarti, Y. Zhan, C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, SIAM Int. Conf. on Data Mining, April 2004.

[JCJ] C. Jin, Q. Chen, S. Jamin, *Inet: Internet Topology Generator*, Technical Report CSE-TR-433-00, EECS Department, University of Michigan, 2000.

[Pa] V. Paxson. *End-to-End Routing Behavior in the Internet*. IEEE/ACM Transactions on Networking, pp. 601-615, December 1998.

[M] D. Magoni, *network manipulator manual*, version 0.9.6, 2002.

[ACL] W. Aiello, F. Chung, L. Lu, *A random graph model for massive graphs*, Proceeding ACM STOC'00, pp. 171-180, 2000.

[PS] M. Portmann, A. Seneviratne, The cost of application-level broadcast in a fully decentralized peer-to-peer network. ISCC 2002. Italy, July 2002.

[BKKMS] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, *Looking up data in P2P systems*, Communications of the ACM, February 2003/Vol. 46, No. 2, pp 43-48.

[Sch] R. Schollmeier, *A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications*, Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01), IEEE.

[RW] R.Rinaldi, M. Waldvogel, *Routing and data location in overlay peer-to-peer networks*, IBM Research Report, 2002.

[AH] K. Aberer, M. Hauswirth, *An Overview on Peer-to-Peer Information Systems*, Workshop on Distributed Data and Structures (WDAS-2002), Paris, France, 2002.

[GPS] Clip2, *The Gnutella Protocol Specification v0.4*, http://www9.limewire.com/developer/**gnutella protocol** 0.4.pdf

[I] I. Ivkovic, *Improving Gnutella Protocol: Protocol Analysis and Research Proposals*, Computer Networking Group Seminar Series, University of Waterloo, November 2001.

[SGG] S. Saroiu, P. K. Gummadi, S. D. Gribble, *A measurement study of peer-to-peer file sharing systems*, University of Washington Technical Report UW-CSE-01-06-02, July 2001.

[SW] S. Sen, J. Wang, *Analyzing peer-to-peer traffic across large networks*, Proceedings of ACM SIGCOMM Internet Measurement Workshop, Marseille, France, November 2002.

[RFI] M. Ripeanu, I. Foster, A. Iamnitchi, *Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for systems design*, IEEE Internet Computing Journal special issue on peer-to-peer networking, vol. 6(1) 2002.

[KGZ] V. Kalogeraki, D. Gunopulos, D. Zeinalipour-Yazti, *A local search mechanism for peer-to-peer networks*, In Proc. of the 11th Int. Conf. on Information and Knowledge.

[LRS] Q. Lv, S. Ratnasamy, S. Shenker. *Can heterogeneity make gnutella scalable?* In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), MIT Faculty Club, Cambridge, MA, USA, March 2002.

[RFHK] S. Ratnasamy, P. Francis, M. Handley, R. Karp, *A scalable content-addressable network.* In Proceedings of SIGCOMM 2001, August 2001.

[SMKKB] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrisnan, *Chord: a scalable peer-to-peer lookup service for internet applications.* In Proceedings of SIGCOMM 2001, August 2001.

[ZKJ] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing.* Technical report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, 94720, April 2001.

[CSW] I. Clarke, O. Sandberg, B. Wiley, *Freenet: A distributed anonymous information storage and retrieval system.* In Proceedings of the Workshop on Design Issues in Anonymity and Unobservability, Berkeley, California, June 2000.

[CRBLS] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S. Shenker, *Making Gnutella-like P2P systems scalable*, In Proceedings of ACM SIGCOMM 2003.

[J] M. A. Jovanovic; *Modeling Large-scale Peer-to-Peer Networks and a Case Study of Gnutella*, Master of Science Thesis. DECE, University of Cincinnati. June 2000.

[Ma] E. P. Markatos, *Tracing a large-scale peer to peer system: an hour in the life of Gnutella*, 2nd IEEE International Symposium on Cluster Computing and the Grid, 2002.

[LCCLS] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker, *Search and replication in unstructured peer-to-peer networks*, Proceedings of 16th ACM International Conference on Supercomputing (ICS'02), New York, USA, June 2002.

[ALPH] L. A. Adamic, R. M. Lukose, A. R. Puniyani, B. A. Huberman, *Search in power-law networks*, Physical Review E, Volume 64, 046135 (2001).

[R] C. Rohrs, *Query routing for the Gnutella network*, Lime Wire LLC, 2002. http://www.limewire.com/developer/query_routing/keyword%20routing.htm

[BM] N. Blundell, L. Mathy, *An overview of gnutella optimization techniques*, Lancaster University, 2002.

[LXLNZ] Y. Liu, L. Xiao, X. Liu, L. M. Ni, and X. Zhang, *Location Awareness in Unstructured Peer-to-Peer Systems*, IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 2, February 2005, 163-174.

[T] G. Toussaint, *The relative neighborhood graph of a finite planar set*, Pattern Recognition, 12, 4, 1980, 261-268.

[SSS] Seddigh, J. Solano, I. Stojmenovic; *RNG and internal node based broadcasting algorithms in wireless one-to-one networks*, ACM Mobile Computing and Communications Review, Vol. 5, No. 2, April 2001 , 37-44.

[LMM] M. Lin, K. Marzullo, S. Masini, *Gossip versus deterministic flooding: low message overhead and high reliability for broadcasting on small networks*, Technical Report CS1999-0637, Department of Computer Science and Engineering, University of California, San Diego, 1999.

[JAB] M. Jovanovic, F. S. Annexstein, K. A. Berman, *Scalability Issues in Large Peer-to-Peer Networks: A Case Study of Gnutella*, Technical Report, University of Cincinnati, 2001.

[JoAB] M. A. Jovanovic, F. S. Annexstein, K. A. Berman, *Modeling peer-to-peer network topologies through "small-world" models and power laws*, IX Telecommunications Forum TELFOR 2001, Belgrade.

[HKB] W. R. Heinzelman, J. Kulik, H. Balakrishnan, *Adaptive protocols for information dissemination in wireless sensor networks*, Proc. MOBICOM, Seattle, 1999, 174-185.

[JDSL] The Java Data Structures Library, Department of Computer Science, Brown University, http://www.jdsl.org/, 2002.

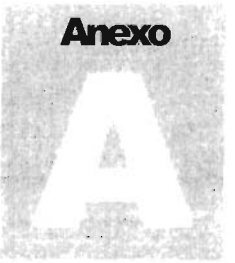
[CISS] J. Cartigny, F. Ingelrest, D. Simplot-Ryl, I. Stojmenovic, *Localized LMST and RNG based minimum-energy broadcast protocols in ad hoc networks*, Ad hoc Networks, Volume 3, Issue 1, January 2005 pp. 1-16.

[JoAB] M. A. Jovanovic, F. S. Annexstein, K. A. Berman, *Modeling peer-to-peer network topologies through "small-world" models and power laws*, IX Telecommunications Forum TELFOR 2001, Belgrade.

[HKB] W. R. Heinzelman, J. Kulik, H. Balakrishnan, *Adaptive protocols for information dissemination in wireless sensor networks*, Proc. MOBICOM, Seattle, 1999, 174-185.

[JDSL] The Java Data Structures Library, Department of Computer Science, Brown University, http://www.jdsl.org/, 2002.

[CISS] J. Cartigny, F. Ingelrest, D. Simplot-Ryl, I. Stojmenovic, *Localized LMST and RNG based minimum-energy broadcast protocols in ad hoc networks*, Ad hoc Networks, Volume 3, Issue 1, January 2005 pp. 1-16.



El simulador de *broadcasting*

Clase	Función
LMST	Implementa los algoritmos de *broadcasting* $ALEM_1$ y $ALEM_2$.
MST	Implementa el algoritmo AEM de Prim usando ya sea la distancia euclidiana entre los nodos terminales de un enlace o el retraso en la transmisión como función de peso.
Prim	Ejecuta el algoritmo de Prim de un grafo usando el patrón definido en JDSL.
RNG	Implementa el algoritmo de *broadcasting* basado en los Grafos de Vecinda d Relativa.
RNGQ	Implementa el algoritmo de *broadcasting* sobre Grafos de Vecindad Relativa con remoción Cuadrilateral.
RumorMongering	Implementa el algoritmo Gossip sobre algún grafo de entrada.
EdgeInfo	Manipula la información relevante de los enlaces, como su longitud y nombre.
VertexInfo	Manipula las propiedades de los vértices, como su nombre y ubicación.
InternetMap	Mapea los datos del archivo de salida de los simuladores BRITE o Recursivo a una estructura de datos conveniente.
GraphTools	Implementa diversas funciones útiles para trabajar con los grafos.
ConnectivityTester	Verifica la conectividad de cualquier grafo.

Class LMST

java.lang.Object
|
+-LMST

public class **LMST**
extends java.lang.Object

Contains methods to obtain a Localized Minimum Spanning Tree (LMST) from a given graph.
Actually, two algorithms were implemented, although final result is the same, difference arises in the time of computing the process and the size of the graph that each one can handle.
The methods findLMST2() and findLMSTwithPrim2() can manipulate bigger graphs.
The methods findLMST() and findLMSTwithPrim() are faster.
Methods with Prim-name use an extension of the IntegerPrimTemplate defined in JDSL 2.
Algorithm for LMST is based work presented by Li *et al.* in the article: *Design and analysis of an MST based topology control algorithm*, Proc. IEEE INFOCOM, 2003.

Constructor Summary

LMST(jdsl.graph.api.Graph input)
Creates a new LMST class with a given input graph

Method Summary

jdsl.graph.api.Graph	findLMST() Obtains the LMST topology of the input graph given in the constructor method.
jdsl.graph.api.Graph	findLMST2() Obtains the LMST topology of the input graph given in the constructor method.
jdsl.graph.api.Graph	findLMSTwithPrim() Obtains the LMST topology of the input graph given in the constructor method.
jdsl.graph.api.Graph	findLMSTwithPrim2() Obtains the LMST topology of the input graph given in the constructor method.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

LMST

public LMST(jdsl.graph.api.Graph input)
Creates a new LMST class with a given input graph

Method Detail

findLMST

public jdsl.graph.api.Graph findLMST()

Obtains the LMST topology of the input graph given in the constructor method.
First all nodes define his Visible Neighborhod and apply the Prim algorithm over it, then each node check if in the MST of its neighbors coincide this node as an neighbor. If it is true, the edge is preserved if not is deleted for the final tree topology.
Returns:
The Localized Minimum Spanning Tree topology

---

findLMST2

public jdsl.graph.api.Graph **findLMST2**()

Obtains the LMST topology of the input graph given in the constructor method.
Each node define his Visible Neighborhod, apply the Prim algorithm over it, gets its neighbor nodes, and for each neighbor define the same, his Visible Neighborhod and apply the Prim algorithm over it. Then check if it coincide as an endpoint. If it is true, the edge is preserved if not is deleted for the final tree topology.
Returns:
The Localized Minimum Spanning Tree topology

---

findLMSTwithPrim

public jdsl.graph.api.Graph **findLMSTwithPrim**()

Obtains the LMST topology of the input graph given in the constructor method. Is the same structure as findLMST() method but the Prim algorithm is implemented as extension of IntegerPrimTemplate defined in JDSL 2.
Returns:
The Localized Minimum Spanning Tree topology

---

findLMSTwithPrim2

public jdsl.graph.api.Graph **findLMSTwithPrim2**()

Obtains the LMST topology of the input graph given in the constructor method. Is the same structure as findLMST2() method but the Prim algorithm is implemented as extension of IntegerPrimTemplate defined in JDSL 2.
Returns:
The Localized Minimum Spanning Tree topology

Class MST

java.lang.Object
|
+-MST

public class MST
extends java.lang.Object

Contains methods to obtain a Minimum Spanning Tree (MST) from a given graph using the Prim algorithm. Actually, two methods were implemented, one using the Euclidean distance between nodes as edges-weight function and other using the delay.

Constructor Summary

MST()

Method Summary

void	findMST(jdsl.graph.api.Graph outGraph, jdsl.graph.api.Graph input, jdsl.graph.api.Vertex node) Obtaing the MST for a given input graph, using the Euclidean distance as weight function.
void	findMSTwithDelay(jdsl.graph.api.Graph outGraph, jdsl.graph.api.Graph input, jdsl.graph.api.Vertex node) Obtaing the MST for a given input graph, using the edges' delay as weight function.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MST

public MST()

Method Detail

findMST

public void findMST(jdsl.graph.api.Graph outGraph,
jdsl.graph.api.Graph input,
jdsl.graph.api.Vertex node)

Obtaing the MST for a given input graph, using the Euclidean distance as weight function.

Parameters:

outGraph - the resulted MST

input - the input graph used to compute the MST

node - the initial node in order to start expanding the tree

findMSTwithDelay

public void findMSTwithDelay(jdsl.graph.api.Graph outGraph,
jdsl.graph.api.Graph input,
jdsl.graph.api.Vertex node)

Obtaing the MST for a given input graph, using the edges' delay as weight function.

Parameters:

outGraph - the resulted MST

input - the input graph used to compute the MST

node - the initial node in order to start expanding the tree

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Prim

java.lang.Object
 |
 +--jdsl.graph.algo.IntegerPrimTemplate
     |
     +--**Prim**

public class Prim
extends jdsl.graph.algo.IntegerPrimTemplate

Implementation of Prim's algorithm using the template-method pattern: IntegerPrimTemplate defined in JDSL 2. Actually execute() implements Prim's algorithm.

Field Summary

Fields inherited from class jdsl.graph.algo.IntegerPrimTemplate

G, INFINITY, locators, Q, source, treeWeight, ZERO

Constructor Summary

`Prim()`

Method Summary

jdsl.graph.api.Graph	`execute`(jdsl.graph.api.InspectableGraph graph, jdsl.graph.api.Vertex vertex) Runs the minimum spanning tree algorithm on a graph started in vertex.
protected void	`treeEdgeFound`(jdsl.graph.api.Vertex v, jdsl.graph.api.Edge vparent, int treeWeight) It is executed when a vertex is added to the minimum spanning tree.
protected void	`vertexNotReachable`(jdsl.graph.api.Vertex v) Called every time a vertex with distance INFINITY comes off the priority queue.
protected int	`weight`(jdsl.graph.api.Edge e) It gets a positive weight for every edge in the graph.

Methods inherited from class jdsl.graph.algo.IntegerPrimTemplate

allVertices, badWeight, destination, doOneIteration, executeAll, executeAll, getLocator, incidentEdges, init, initMap, newPQ, relaxingEdge, setLocator, shouldContinue

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Prim

public Prim()

Method Detail

weight

protected int **weight**(jdsl.graph.api.Edge e)

It gets a positive weight for every edge in the graph. This method gets called by the algorithm when the algorithm needs to know the weight of an edge. Prim's algorithm cannot handle negative weights. In our case, the weight is the Euclidean distance between nodes or the delay in the connection.

Specified by:

weight in class jdsl.graph.algo.IntegerPrimTemplate

Parameters:

e - edge for which the algorithm needs to know a weight

Returns:

the weight value for e

---

vertexNotReachable

protected void **vertexNotReachable**(jdsl.graph.api.Vertex v)

Called every time a vertex with distance INFINITY comes off the priority queue. When it has been called once, it should subsequently be called for all remaining vertices, until the priority queue is empty.

Overrides:

vertexNotReachable in class jdsl.graph.algo.IntegerPrimTemplate

Parameters:

v - vertex which the algorithm just found to be unreachable from the source

---

treeEdgeFound

protected void **treeEdgeFound**(jdsl.graph.api.Vertex v,
jdsl.graph.api.Edge vparent,
int treeWeight)

It is executed when a vertex is added to the minimum spanning tree. The algorithm calls this method at most once per vertex, after the vertex has been "finished" (i.e., when the path from s to the vertex is known). The vertex will never again be touched or considered by the algorithm.

Overrides:

treeEdgeFound in class jdsl.graph.algo.IntegerPrimTemplate

Parameters:

v - vertex that the algorithm just finished

vparent - edge leading into v in the minimum spanning tree

treeWeight - the total weight of all edges known to be in the tree at this point in the execution of the algorithm, including vparent

---

execute

public jdsl.graph.api.Graph **execute**(jdsl.graph.api.InspectableGraph graph,
jdsl.graph.api.Vertex vertex)

Runs the minimum spanning tree algorithm on a graph started in vertex.

Parameters:

graph - the input graph for finding the minimum spanning tree

vertex - the Vertex at which to start the algorithm

Class RNG

java.lang.Object

|

+-RNG

public class RNG
extends java.lang.Object

Contains methods for obtaining a Relative Neighborhood Graph from a given graph. Actually, two algorithms were implemented for RUG, although final result is the same, difference arises when managing edges to be deleted.
applyRNG does not delete edges until whole process is finished. applyRNG2 starts with a Graph only containing nodes and for which edges are attached as soon as they are labeled as connectors of two neighbor nodes
Algorithm for RNG is based work presented by G. Toussaint, 1980.

Constructor Summary
RNG()

Method Summary	
jdsl.graph.api.Graph	applyRNG(jdsl.graph.api.Graph inputGraph) Applies RNG algorithm to a given graph, can use any kind of connected graph as input.
jdsl.graph.api.Graph	applyRNG2(jdsl.graph.api.Graph inputGraph) Applies RNG algorithm to a given graph, can use any kind of connected graph as input.
jdsl.graph.api.Graph	getRNG(jdsl.graph.api.Graph inputGraph) Obtains RNG from a given graph it works only when input graph is a RUG due to intrinsec properties of RUG

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RNG

public RNG()

Method Detail

getRNG

public jdsl.graph.api.Graph getRNG(jdsl.graph.api.Graph inputGraph)

Obtains RNG from a given graph it works only when input graph is a RUG due to intrinsic properties of RUG

Parameters:

inputGraph - the graph containing the set of nodes and edges to generate RNG

Returns:

The Relative Neighborhood Graph of the inputGraph

applyRNG

public jdsl.graph.api.Graph applyRNG(jdsl.graph.api.Graph inputGraph)

Applies RNG algorithm to a given graph, can use any kind of connected graph as input. On this implementation, just one Graph is used, RNG algorithm is applied over entire graph, edges to be deleted are marked and deleted just once the algorithm has covered all edges, this same graph is returned as output.

Parameters:

inputGraph - the graph containing over which apply RNG algorithm

Returns:

The Relative Neighborhood Graph of the inputGraph

---

applyRNG2

public jdsl.graph.api.Graph **applyRNG2**(jdsl.graph.api.Graph inputGraph)

Applies RNG algorithm to a given graph, can use any kind of connected graph as input. On this implementation, outputGraph is generated on fly. RNG algorithm is applied over each pair of nodes, and just relative neighbors are passed to outputGraph

Parameters:

inputGraph - the graph containing over which apply RNG algorithm

Returns:

The Relative Neighborhood Graph of the inputGraph

Package

Class RNGQ

java.lang.Object
|
+-RNGQ

public class RNGQ
extends java.lang.Object
Contains methods for obtaining a Relative Neighborhood Graph with Quadrilateral removals from a given graph. The quadrilateral removals operation is applied on quadrangles formed when a two hop neighbor has two or more paths to given node. The edge with largest cost in any quadrilateral is removed. Algorithm for RNGQ is based work presented by Liu et al. in IEEE Transactions on parallel and distributed systems, 2005.

Constructor Summary

RNGQ()

Method Summary

jdsl.graph.api.Graph	getRNGQ(jdsl.graph.api.Graph inputGraph) Applies RNGQ algorithm to a given graph, can use any kind of connected graph as input.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RNGQ

public RNGQ()

Method Detail

getRNGQ

public jdsl.graph.api.Graph getRNGQ(jdsl.graph.api.Graph inputGraph)
Applies RNGQ algorithm to a given graph, can use any kind of connected graph as input. On this implementation, just one Graph is used, RNGQ algorithm is applied over entire graph, edges to be deleted are marked and deleted just once the algorithm has covered all edges, this same graph is returned as output.
Parameters:
inputGraph - the graph containing over which apply RNGQ algorithm
Returns:
The Relative Neighborhood Graph with Quadrilateral removals from a given input graph

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class RumorMongering

java.lang.Object
|
+-RumorMongering

public class RumorMongering
extends java.lang.Object
Contains methods to implement the Rumor Mongering protocol, according to Portmann and Seneviratne article: The cost of application level broadcast in a fully decentralized peer-to-peer network.

Constructor Summary

RumorMongering()

Method Summary

void	BlindCounter(jdsl.graph.api.Graph input, int B, int F) Given the input graph and the parameters, this class obtain the cost (numMessages/costFlooding) and the percentage of reached nodes due to the Gossip protocol.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RumorMongering

public RumorMongering()

Method Detail

BlindCounter

public void BlindCounter(jdsl.graph.api.Graph input,
int B,
int F)

Given the input graph and the parameters, this class obtain the cost (numMessages/costFlooding) and the percentage of reached nodes due to the Gossip protocol.

Parameters:

input - the graph containing the set of nodes and edges to apply the Gossip protocol

B - number of neighbors, chosen at random, that each node knows have not yet seen the message

F - number of times that each node could receive the message in order to retransmit it

Returns:

null

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class EdgeInfo

java.lang.Object
|
+--EdgeInfo

public class **EdgeInfo**
extends java.lang.Object

Implements methods to store and retrieve information from an Edge in a Graph. Information stored in an Edge is its name, lenght and delay.

Field Summary	
protected double	delay
protected double	lenght
protected boolean	markForDeleting
protected java.lang.String	name
protected boolean	xyEdge

Constructor Summary
EdgeInfo() Standard constructor, used when name, lenght and delay have not been calculated
EdgeInfo(java.lang.String nameEdg, double lenghtEdg) Stores name and lenght edge

Method Summary	
void	delay(double delayEdge) Attaches delay of current edge
double	getDelay() Returns edge's delay
double	getLenght() Returns edge lenght
java.lang.String	getName() Returns edge name
boolean	isMarked() Checks if edge is marked for deletion Returns true if edge needs to be deleted
void	lenght(double lenghtEdge)

	Attaches lenght of current edge
void	markDeleted(boolean mark) Marks edge to be deleted
void	name(java.lang.String nameEdge) Attaches edge name
void	setXYEdge() Specifys that lenght information corresponds euclidean distance between nodes

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

name

protected java.lang.String name

lenght

protected double lenght

xyEdge

protected boolean xyEdge

markForDeleting

protected boolean markForDeleting

delay

protected double delay

Constructor Detail

EdgeInfo

public EdgeInfo(java.lang.String nameEdg,
double lenghtEdg)
Stores name and lenght edge
Parameters:
nameEdg - string containing edge name
lenghtEdg - edge lenght in double format

EdgeInfo

public EdgeInfo()
Standard constructor, used when name, lenght and delay have not been calculated

Method Detail

name

public void name(java.lang.String nameEdge)
Attaches edge name
Parameters:
nameEdge - name of current edge

lenght

public void lenght(double lenghtEdge)
Attaches lenght of current edge
Parameters:
lenghtEdge - lenght of edge in double format

markDeleted

public void markDeleted(boolean mark)
Marks edge to be deleted

Parameters:
mark - contains true if edge must be deleted otherwise contains false

getName

public java.lang.String **getName**()
Returns edge name

getLenght

public double **getLenght**()
Returns edge lenght

setXYEdge

public void **setXYEdge**()
Specifys that lenght information corresponds euclidean distance between nodes

isMarked

public boolean **isMarked**()
Checks if edge is marked for deletion Returns true if edge needs to be deleted

delay

public void **delay**(double delayEdge)
Attaches delay of current edge
Parameters:
delayEdge - delay of edge in double format

getDelay

public double **getDelay**()
Returns edge's delay

Package

Class VertexInfo

java.lang.Object

|

+--VertexInfo

public class **VertexInfo**
extends java.lang.Object

Implements methods needed for storing and retrieving information from nodes in a given Graph. On each vertex it is possible to attach:

- name
- x-y coordiantes
- number of forwarded messages
- last message forwarded

among other particular properties that depends of each algorithm

Constructor Summary
`VertexInfo()`

Method Summary	
void	`added`(boolean add) Used in MST class to define that the current node was already added in the tree
void	`addFwdVertex`(jdsl.graph.api.Vertex vtx) Used in RumorMongering class.
void	`addValidVertex`(jdsl.graph.api.Vertex vtx) Used in RumorMongering class.
void	`delay`(double value) Used in MST class to assign the low cost (minimum delay) the node can be reached in the current tree.
void	`distance`(double d) Used in MST class to assign the low cost (minimum distance) the node can be reached in the current tree.
boolean	`existFwdVertex`(jdsl.graph.api.Vertex vtx) Used in RumorMongering class in order to check if a neighbor is already added in the ForwardList
void	`from`(jdsl.graph.api.Vertex closest) Used in MST class to assign the closest node this vertex can be reached
void	`FwdList`(java.util.ArrayList FwdList) Used in RumorMongering class.
double	`getDelay`() Used in MST class to get the low cost (minimum delay) assigned to node
double	`getDistance`() Used in MST class to get the low cost (minimum distance) assigned to node
jdsl.graph.api.Vertex	`getFrom`() Used in MST class to get the closest node

int	**getFrwdMessages**() Gets number of messages forwarded
boolean	**getHasRcvMessage**() Used in RumorMongering class to inform if the current node has received the broadcast message
boolean	**getKnown**() Used in MST class to know if the current node has been visited
java.lang.String	**getName**() Gets name of node
int	**getRcvMessages**() Used in RumorMongering class to get the times the current node has received the broadcast message
jdsl.graph.api.Vertex	**getSourceNode**() Allows knowing source node
jdsl.graph.api.Vertex	**getValidVertex**(int i) Used in RumorMongering class.
java.util.List	**getValidVertexList**() Used in RumorMongering class.
int	**getX**() Gets x-coordinate of node
int	**getY**() Gets y-coordinate of node
void	**hasRcvMessage**(boolean message) Used in RumorMongering class to notify if the current node has already received the broadcast message
void	**incFwdMessages**(jdsl.graph.api.Vertex vtx) Increments counter every time a node forwards a message
void	**incRcvMessages**() Used in RumorMongering class to count the times the current node has received the broadcast message in order to compare this value with the parameter F
void	**known**(boolean visited) Used in MST class to mark the current node as visited
void	**name**(java.lang.String nameVer) Attaches node name
void	**sendMessage**(int mess) Saves a copy of current message being forwarded
void	**source**(jdsl.graph.api.Vertex vtx) Allows establishing source node in forwarding process
void	**validVertexList**(java.util.ArrayList vvList) Used in RumorMongering class.
boolean	**wasAdded**() Used in MST class to inform if the current node was added in the tree
void	**xCoor**(int xpos) Attaches x coordinate of node
void	**yCoor**(int ypos) Attaches y coordinate of node

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

VertexInfo

public VertexInfo()

Method Detail

added

public void added(boolean add)

Used in MST class to define that the current node was already added in the tree

Parameters:

add - true if it belongs to the MST

---

wasAdded

public boolean wasAdded()

Used in MST class to inform if the current node was added in the tree

Returns:

true if it already belongs to the MST, false otherwise

---

distance

public void distance(double d)

Used in MST class to assign the low cost (minimum distance) the node can be reached in the current tree.

Parameters:

d - minimum distance

---

getDistance

public double getDistance()

Used in MST class to get the low cost (minimum distance) assigned to node

Returns:

minimum distance

---

delay

public void delay(double value)

Used in MST class to assign the low cost (minimum delay) the node can be reached in the current tree.

---

getDelay

public double getDelay()

Used in MST class to get the low cost (minimum delay) assigned to node

Returns:

minimum delay

---

from

public void from(jdsl.graph.api.Vertex closest)

Used in MST class to assign the closest node this vertex can be reached

Parameters:

closest - the closest node

---

getFrom

public jdsl.graph.api.Vertex getFrom()

Used in MST class to get the closest node

Returns:

the closest node

---

known

public void known(boolean visited)

Used in MST class to mark the current node as visited

Parameters:

visited - if this node has been seen

---

getKnown

public boolean getKnown()

Used in MST class to know if the current node has been visited
Returns:
true if the node has been seen, false otherwise

---

hasRcvMessage

public void **hasRcvMessage**(boolean message)
Used in RumorMongering class to notify if the current node has already received the broadcast message
Parameters:
message - true to mark the message has been received, false otherwise

---

getHasRcvMessage

public boolean **getHasRcvMessage**()
Used in RumorMongering class to inform if the current node has received the broadcast message
Returns:
true if the message has been received, false otherwise

---

incRcvMessages

public void **incRcvMessages**()
Used in RumorMongering class to count the times the current node has received the broadcast message in order to compare this value with the parameter F

---

getRcvMessages

public int **getRcvMessages**()
Used in RumorMongering class to get the times the current node has received the broadcast message
Returns:
times the message has been received

FwdList

public void **FwdList**(java.util.ArrayList FwdList)
Used in RumorMongering class. Attaches ForwardList = forward + source nodes
Parameters:
FwdList - list of the forwarders

---

addFwdVertex

public void **addFwdVertex**(jdsl.graph.api.Vertex vtx)
Used in RumorMongering class. Adds in the ForwardList the neighbor that is waiting to be sended the broadcast message (forwarding-nodes)
Parameters:
vtx - the neighbor to be added

---

existFwdVertex

public boolean **existFwdVertex**(jdsl.graph.api.Vertex vtx)
Used in RumorMongering class in order to check if a neighbor is already added in the ForwardList
Parameters:
vtx - the neighbor to check * @return true if the neighbor has already received the broadcast message, false otherwise

---

validVertexList

public void **validVertexList**(java.util.ArrayList vvList)
Used in RumorMongering class. Attaches the list of all possible neighbors that the current node could send the broadcast message
Parameters:
vvList - the list that will contain the possible neighbors

---

addValidVertex

public void **addValidVertex**(jdsl.graph.api.Vertex vtx)
Used in RumorMongering class. Adds a neighbor in the ValidVertexList
Parameters:
vtx - the node to add

---

getValidVertex

public jdsl.graph.api.Vertex **getValidVertex**(int i)

Used in RumorMongering class. Gets a neighbor from the ValidVertexList

Parameters:

i - the position of the node in the list

Returns:

the node

---

getValidVertexList

public java.util.List **getValidVertexList**()

Used in RumorMongering class. Returns the ValidVertexList

---

name

public void **name**(java.lang.String nameVer)

Attaches node name

Parameters:

nameVer - name of the node

---

xCoor

public void **xCoor**(int xpos)

Attaches x coordinate of node

Parameters:

xpos - x-coordiante

---

yCoor

public void **yCoor**(int ypos)

Attaches y coordinate of node

Parameters:

ypos - y-coordiante

---

source

public void **source**(jdsl.graph.api.Vertex vtx)

Allows establishing source node in forwarding process

Parameters:

vtx - node acting as source node in current broadcasting process

---

incFwdMessages

public void **incFwdMessages**(jdsl.graph.api.Vertex vtx)

Increments counter every time a node forwards a message

Parameters:

vtx - node for which counters must be incremented

---

sendMessage

public void **sendMessage**(int mess)

Saves a copy of current message being forwarded

Parameters:

mess - message to be broadcasted

---

getName

public java.lang.String **getName**()

Gets name of node

Returns:

String containing node name

---

getFrwdMessages

public int **getFrwdMessages**()

Gets number of messages forwarded

Returns:

number of messages forwarded

---

getSourceNode

public jdsl.graph.api.Vertex **getSourceNode**()

Allows knowing source node

Returns:

pointer to source node in broadcasting task

---

getX

public int **getX**()

Gets x-coordinate of node

Returns:

x-coordinate of node

---

getY

public int **getY**()

Gets y-coordinate of node

Returns:

y-coordinate of node

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class InternetMap

java.lang.Object
|
+-InternetMap

public class InternetMap
extends java.lang.Object

InternetMap class contains all methods used to load into memory Graphs generated by topology generators such as BRITE and Recursive. In all cases, passed parameters need to establish source file from which to load data and format of input data.
Once data has been loaded into memory, Graph is stored on an instance variable of class InternetMap. InternetMap class also contains methods to pass loaded Graph to other methods in different classes.
Although BRITE provides x-y coordiantes for each node, edges leght and time delay for each connection, Recursive does not, so InternetMap includes some methods to randomly generate x-y coordianates for each node, delay between two nodes is assigned randomly in the range [0, 1] to the edge joining them.

Constructor Summary
InternetMap(java.lang.String file) Prepares input to be readed by further methods

Method Summary	
void	briteXY() Reads file generated by BRITE
void	delay() Reads file generated by Recursive assigning a random delay in range [0, 1] to each edge joining a pair of nodes.
void	distance() Reads file generated by Recursive assigning a random x-y coordinate to each node present on file.
jdsl.graph.api.Graph	getGraph() Allows having access to Graph stored on instance variable of InternetMap
void	getNumberNodes() Allows knowing number of nodes present in input file, value is stored on internal instance variable
void	special() Reads file generated by BRITE but without edges info in order to use it for simulate a Wireless adhoc network

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

InternetMap

public **InternetMap**(java.lang.String file)
throws java.io.IOException
Prepares input to be readed by further methods

Parameters:
file - name of file containing data to be loaded

Throws:
java.io.IOException - an exception if given file does not exists

Method Detail

getGraph

public jdsl.graph.api.Graph **getGraph**()
Allows having access to Graph stored on instance variable of InternetMap
Returns:
Graph containing Internet Map loaded by means of InternetMap class

---

distance

public void **distance**()
throws java.io.IOException
Reads file generated by Recursive assigning a random x-y coordinate to each node present on file.
java.io.IOException

---

delay

public void **delay**()
throws java.io.IOException
Reads file generated by Recursive assigning a random delay in range [0, 1] to each edge joining a pair of nodes.
java.io.IOException

---

briteXY

public void **briteXY**()
throws java.io.IOException
Reads file generated by BRITE
java.io.IOException

---

special

public void **special**()
throws java.io.IOException
Reads file generated by BRITE but without edges info in order to use it for simulate a Wireless adhoc network
java.io.IOException

---

getNumberNodes

public void **getNumberNodes**()
throws java.io.IOException
Allows knowing number of nodes present in input file, value is stored on internal instance variable
java.io.IOException

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

Class graphTools

java.lang.Object
|
+-graphTools

public class **graphTools**
extends java.lang.Object

Contains useful methods for working with graphs.

Constructor Summary
graphTools()

Method Summary	
static jdsl.graph.api.Vertex	cloneNode(jdsl.graph.api.Vertex paternNode, jdsl.graph.api.Graph daughterGraph) Makes a copy of a node from Graph if a pattern node is found
static void	compare(jdsl.graph.api.Graph gra1, jdsl.graph.api.Graph gra2, java.io.FileWriter output) Compares number of edges and degree of two different graphs
static jdsl.graph.api.Graph	copyGraph(jdsl.graph.api.Graph inputGraph) Copies a whole Graph
static jdsl.graph.api.Graph	copyNodesFrom(jdsl.graph.api.Graph inGraph) Makes copy of nodes in a Graph and puts them in a new Graph
static int	countVertices(jdsl.graph.api.VertexIterator vtxIter) Counts number of nodes in a VertexIterator
static jdsl.graph.api.Vertex	getNodeWhichName(java.lang.String name, jdsl.graph.api.Graph graph) Looks for a specific node inside a Graph using name of node as key
static jdsl.graph.api.Vertex	getNodeWhichName(java.lang.String name, jdsl.graph.api.VertexIterator vtxIter) Deprecated. *Use instead public static Vertex getNodeWhichName(String name, Graph graph)*
static int[]	getVertexXY(jdsl.graph.api.Vertex vContent) Gets xy coordiantes from node name
static jdsl.graph.api.VertexIterator	goToVertex(jdsl.graph.api.VertexIterator verIt, int vertex) Allows moving back or forward through a vertexIterator, thus going to a specific node
static double	lenghtEdge(jdsl.graph.api.Vertex a, jdsl.graph.api.Vertex b) gets distance between two nodes
static void	showFw(jdsl.graph.api.Graph graph) Shows on screen messages forwarded by every node in a Graph
static void	showSource(jdsl.graph.api.Graph graph) Shows on screen source node of every node in the Graph

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

graphTools

public graphTools()

Method Detail

lenghtEdge

public static double **lenghtEdge**(jdsl.graph.api.Vertex a,
jdsl.graph.api.Vertex b)

gets distance between two nodes

Parameters:

a - node a

b - node b

Returns:

distance between a and b

getVertexXY

public static int[] **getVertexXY**(jdsl.graph.api.Vertex vContent)

Gets xy coordinates from node name

Parameters:

vContent - node from which to obtain coordinates

Returns:

array containing [x,y] coordiantes

goToVertex

public static jdsl.graph.api.VertexIterator **goToVertex**(jdsl.graph.api.VertexIterator verIt,
int vertex)

Allows moving back or forward through a vertexIterator, thus going to a specific node

Parameters:

verIt - VertexIterator containing all vertices in the Graph

vertex - number ID of node to move to

Returns:

a VertexIterator forwarded to desired node

copyNodesFrom

public static jdsl.graph.api.Graph **copyNodesFrom**(jdsl.graph.api.Graph inGraph)

Makes copy of nodes in a Graph and puts them in a new Graph

Parameters:

inGraph - source node

Returns:

a graph containing only the nodes of input Graph

copyGraph

public static jdsl.graph.api.Graph **copyGraph**(jdsl.graph.api.Graph inputGraph)

Copies a whole Graph

Parameters:

inputGraph - graph to be copied

Returns:

a copy of inputGraph

getNodeWhichName

public static jdsl.graph.api.Vertex **getNodeWhichName**(java.lang.String name,
jdsl.graph.api.VertexIterator vtxIter)

Deprecated. *Use instead public static Vertex getNodeWhichName(String name, Graph graph)*

getNodeWhichName

public static jdsl.graph.api.Vertex **getNodeWhichName**(java.lang.String name,
jdsl.graph.api.Graph graph)

Looks for a specific node inside a Graph using name of node as key.

Parameters:

name - name of the node to be searched

graph – the graph into which it looks for the node

Returns:

pointer to node found

compare

public static void **compare**(jdsl.graph.api.Graph gra1,
jdsl.graph.api.Graph gra2,
java.io.FileWriter output)
throws java.io.IOException

Compares number of edges and degree of two different graphs

Parameters:

gra1 - graph1 to be compared

gra2 - graph2 to be compared

output - name of file to write out comparison information

countVertices

public static int **countVertices**(jdsl.graph.api.VertexIterator vtxIter)

Counts number of nodes in a VertexIterator

Parameters:

vtxIter - contains nodes to be counted

Returns:

number of nodes

showFw

public static void **showFw**(jdsl.graph.api.Graph graph)

Shows on screen messages forwarded by every node in a Graph

Parameters:

graph - graph to be analyzed

showSource

public static void **showSource**(jdsl.graph.api.Graph graph)

Shows on screen source node of every node in the Graph

Parameters:

graph - graph to be analyzed

cloneNode

public static jdsl.graph.api.Vertex **cloneNode**(jdsl.graph.api.Vertex paternNode,
jdsl.graph.api.Graph daughterGraph)

Makes a copy of a node from Graph if a pattern node is found

Parameters:

paternNode - node to be compared to nodes in daughterGraph

Returns:

codeNode is a copy of node contained in daughterGraphs which fits with name in patternNode

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class ConnectivityTester

java.lang.Object

```
|
+-jdsl.graph.algo.DFS
    |
    +-ConnectivityTester
```

public class **ConnectivityTester**
extends jdsl.graph.algo.DFS

ConnectivityTester allows checking if a given graph is completely connected. This class extends Depth-First Search algorithm implemented on JDSL. The method isConnected() runs DFS algorithm starting at a random node, afterwards, entire Graph is traversed to see if it exists parent nodes besides start node, if such, returns false indicating that Graph is not completely connected.

Field Summary

Fields inherited from class jdsl.graph.algo.DFS

BACK_EDGE, CROSS_EDGE, EDGE_TYPE, FINISH_TIME, FORWARD_EDGE, graph_, PARENT, START_TIME, TREE_EDGE, TREE_NUMBER, treeNum_, UNSEEN, UNVISITED, VERTEX_STATUS, VISITED, VISITING, visitResult_

Constructor Summary

ConnectivityTester()

Method Summary

boolean	isConnected(jdsl.graph.api.InspectableGraph g) Checks for connectivity of a Graph

Methods inherited from class jdsl.graph.algo.DFS

cleanup, dfsVisit, execute, execute, finishTime, finishVisit, interestingIncidentEdges, isBackEdge, isCrossEdge, isDone, isForwardEdge, isTreeEdge, isUnseen, isUnvisited, isVisited, isVisiting, parent, startTime, startVisit, status, traverseBackEdge, traverseCrossEdge, traverseForwardEdge, traverseTreeEdge, treeNumber, type

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ConnectivityTester

public ConnectivityTester()

Method Detail

isConnected

public boolean **isConnected**(jdsl.graph.api.InspectableGraph g)

Checks for connectivity of a Graph

Parameters:

g - graph to check connectivity

Returns:

true if graph is completely connected false any other case.

Package

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

Código fuente

```
/*
 * LMST.java
 */

import jdsl.core.api.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import jdsl.core.ref.*;
import java.util.*;
import java.text.*;
import java.io.*;
/**
 * Contains methods to obtain a Localized Minimum Spanning Tree (LMST) from a given graph.
 * Actually, two algorithms were implemented, although final result is the same, difference arises in the
 * time of computing the process and the size of the graph that each one can handle.
 * <p>The methods findLMST2() and findLMSTwithPrim2() can manipulate bigger graphs.
 * <p>The methods findLMST() and findLMSTwithPrim() are faster.
 * <p>Methods with Prim-name use an extension of the IntegerPrimTemplate defined in JDSL 2.
 * <p>Algorithm for LMST is based work presented by Li <i>et al.</i> in the article: <i>Design and
 * analysis of an MST based topology control algorithm</i>, Proc. IEEE INFOCOM, 2003.
 *
 * @author  Tania Pérez
 */

public class LMST {
  private Graph inputGraph;
  private int numVertices;
  private Graph [] graph;
  private Graph[] MSTgraph;
  private MST  mst;
  private Prim prim;

  /** Creates a new LMST class with a given input graph
  */
  public LMST (Graph input) {
     inputGraph = graphTools.copyGraph(input);
     numVertices = inputGraph.numVertices();
     mst = new MST();
     prim = new Prim();
  }
```

```
/**Obtains the LMST topology of the input graph given in the constructor method. <p>
 * First all nodes define his Visible Neigborhod and apply the Prim algorithm over it, then
 * each node check if in the MST of its neighbors coincide this node as an neighbor. If it
 * is true, the edge is preserved if not is deleted for the final tree topology.
 * @return The Localized Minimum Spanning Tree topology
 */
public Graph findLMST(){
    Graph outputGraph = new IncidenceListGraph();
    outputGraph = graphTools.copyNodesFrom(inputGraph);
  graph = new IncidenceListGraph [numVertices];
  MSTgraph = new IncidenceListGraph [numVertices];
  for (int i = 0; i < numVertices; i++) {
      graph[i] = graphTools.copyGraph(inputGraph);
      MSTgraph[i] = new IncidenceListGraph();
  }
  for(int i = 0; i < numVertices; i++){
      graph[i] = buildAdjacentGraph(inputGraph,i);
    VertexIterator newVtxIter = graph[i].vertices();
    Vertex ver1=newVtxIter.nextVertex();
    mst.findMST(MSTgraph[i],graph[i],ver1);
  }
   for(int i = 0; i < numVertices; i++){
      Vertex node = graphTools.getNodeWhichName(""+i,MSTgraph[i]);
      for (VertexIterator vtxIter = MSTgraph[i].adjacentVertices(node); vtxIter.hasNext();){
          Vertex endpoint = vtxIter.nextVertex();
          int j = (Integer.valueOf(((VertexInfo)endpoint.element()).getName())).intValue();
          Vertex test = graphTools.getNodeWhichName(""+i,MSTgraph[j]);
          Vertex source = graphTools.getNodeWhichName(""+j,MSTgraph[j]);
          boolean found = coincideEndpoint(MSTgraph[j],source,test);
          if (found){
              Vertex start = graphTools.getNodeWhichName(""+i,outputGraph);
              Vertex end = graphTools.getNodeWhichName(""+j,outputGraph);
              EdgeInfo edgeInfo = new EdgeInfo();
              Edge edge = MSTgraph[j].aConnectingEdge(test, source);
              edgeInfo = (EdgeInfo)edge.element();
              if(!outputGraph.areAdjacent(start,end)){
                  outputGraph.insertEdge(start,end,edgeInfo);
              }
          }
      }
   }
   return outputGraph;
}

/**Obtains the LMST topology of the input graph given in the constructor method. <p>
* Each node define his Visible Neigborhod, apply the Prim algorithm over it, gets its
* neighbor nodes, and for each neighbor define the same, his Visible Neigborhod and apply the Prim
*  algorithm over it. Then check if it coincide as an endpoint. If it
* is true, the edge is preserved if not is deleted for the final tree topology.
* @return The Localized Minimum Spanning Tree topology
*/
public Graph findLMST2(){
    Graph outputGraph = new IncidenceListGraph();
    outputGraph = graphTools.copyNodesFrom(inputGraph);
    Graph graph = new IncidenceListGraph();
```

```
        Graph graphTest = new IncidenceListGraph();
        Graph MSTgraph = new IncidenceListGraph();
        Graph MSTgraphTest = new IncidenceListGraph();
        for(int i = 0; i < numVertices; i++) {
            graph = buildAdjacentGraph(inputGraph,i);
            VertexIterator newVtxIter = graph.vertices();
            Vertex ver1=newVtxIter.nextVertex();
            mst.findMST(MSTgraph,graph,ver1);
            Vertex node = graphTools.getNodeWhichName(""+i,MSTgraph);
            for (VertexIterator vtxIter = MSTgraph.adjacentVertices(node); vtxIter.hasNext();) {
                Vertex endpoint = vtxIter.nextVertex();
                int j = (Integer.valueOf(((VertexInfo)endpoint.element()).getName())).intValue();
                if(i>j) {
                    graphTest = buildAdjacentGraph(inputGraph,j);
                    Vertex vtxTest = graphTools.getNodeWhichName(""+j,graphTest);
                    mst.findMST(MSTgraphTest,graphTest,vtxTest);
                    Vertex test = graphTools.getNodeWhichName(""+i,MSTgraphTest);
                    Vertex source = graphTools.getNodeWhichName(""+j,MSTgraphTest);
                    boolean found = coincideEndpoint(MSTgraphTest,source,test);
                    if (found) {
                        Vertex start = graphTools.getNodeWhichName(""+i,outputGraph);
                        Vertex end = graphTools.getNodeWhichName(""+j,outputGraph);
                        EdgeInfo edgeInfo = new EdgeInfo();
                        Edge edge = MSTgraphTest.aConnectingEdge(test, source);
                        edgeInfo = (EdgeInfo)edge.element();
                        if(!outputGraph.areAdjacent(start,end)) {
                            outputGraph.insertEdge(start,end,edgeInfo);
                        }
                    }
                }
            }
        }
        return outputGraph;
    }

    /**Obtains the LMST topology of the input graph given in the constructor method.
    * Is the same structure as findLMST() method but the Prim algorithm is implemented
    * as extension of IntegerPrimTemplate defined in JDSL 2.
    * @return The Localized Minimum Spanning Tree topology
    */
    public Graph /*void*/ findLMSTwithPrim() {
        Graph outputGraph = new IncidenceListGraph();
        outputGraph = graphTools.copyNodesFrom(inputGraph);
        graph = new IncidenceListGraph [numVertices];
        MSTgraph = new IncidenceListGraph [numVertices];
        for (int i = 0; i < numVertices; i++) {
            graph[i] = graphTools.copyGraph(inputGraph);
            MSTgraph[i] = new IncidenceListGraph();
        }
        for(int i = 0; i < numVertices; i++) {
            graph[i] = buildAdjacentGraph(inputGraph,i);
            VertexIterator newVtxIter = graph[i].vertices();
            Vertex ver1=newVtxIter.nextVertex();
            MSTgraph[i] = prim.execute(graph[i],ver1);
        }
        for(int i = 0; i < numVertices; i++) {
            Vertex node = graphTools.getNodeWhichName(""+i,MSTgraph[i]);
```

```
        for (VertexIterator vtxIter = MSTgraph[i].adjacentVertices(node); vtxIter.hasNext();){
            Vertex endpoint = vtxIter.nextVertex();
            int j = (Integer.valueOf(((VertexInfo)endpoint.element()).getName())).intValue();
            Vertex test = graphTools.getNodeWhichName(""+i,MSTgraph[j]);
            Vertex source = graphTools.getNodeWhichName(""+j,MSTgraph[j]);
            boolean found = coincideEndpoint(MSTgraph[j],source,test);
            if (found){
                Vertex start = graphTools.getNodeWhichName(""+i,outputGraph);
                Vertex end = graphTools.getNodeWhichName(""+j,outputGraph);
                EdgeInfo edgeInfo = new EdgeInfo();
                Edge edge = MSTgraph[j].aConnectingEdge(test, source);
                edgeInfo = (EdgeInfo)edge.element();
                if(!outputGraph.areAdjacent(start,end)){
                    outputGraph.insertEdge(start,end,edgeInfo);
                }
            }
        }
    }
    return outputGraph;
}

/**Obtains the LMST topology of the input graph given in the constructor method.
 * Is the same structure as findLMST2() method but the Prim algorithm is implemented
 * as extension of IntegerPrimTemplate defined in JDSL 2.
 * @return The Localized Minimum Spanning Tree topology
 */
public Graph findLMSTwithPrim2(){
    Graph outputGraph = new IncidenceListGraph();
    outputGraph = graphTools.copyNodesFrom(inputGraph);
    Graph graph = new IncidenceListGraph();
    Graph graphTest = new IncidenceListGraph();
    Graph MSTgraph = new IncidenceListGraph();
    Graph MSTgraphTest = new IncidenceListGraph();
    for(int i = 0; i < numVertices; i++){
        graph = buildAdjacentGraph(inputGraph,i);
        VertexIterator newVtxIter = graph.vertices();
        Vertex ver1=newVtxIter.nextVertex();
        MSTgraph = prim.execute(graph,ver1);
        Vertex node = graphTools.getNodeWhichName(""+ i,MSTgraph);
        for (VertexIterator vtxIter = MSTgraph.adjacentVertices(node); vtxIter.hasNext();){
            Vertex endpoint = vtxIter.nextVertex();
            int j = (Integer.valueOf(((VertexInfo)endpoint.element()).getName())).intValue();
            if(i>j){
                graphTest = buildAdjacentGraph(inputGraph,j);
                Vertex vtxTest = graphTools.getNodeWhichName(""+j,graphTest);
                MSTgraphTest = prim.execute(graphTest,vtxTest);
                Vertex test = graphTools.getNodeWhichName(""+i,MSTgraphTest);
                Vertex source = graphTools.getNodeWhichName(""+j,MSTgraphTest);
                boolean found = coincideEndpoint(MSTgraphTest,source,test);
                if (found){
                    Vertex start = graphTools.getNodeWhichName(""+i,outputGraph);
                    Vertex end = graphTools.getNodeWhichName(""+j,outputGraph);
                    EdgeInfo edgeInfo = new EdgeInfo();
                    Edge edge = MSTgraphTest.aConnectingEdge(test, source);
                    edgeInfo = (EdgeInfo)edge.element();
                    if(!outputGraph.areAdjacent(start,end)){
                        outputGraph.insertEdge(start,end,edgeInfo);
```

```
                    }
                }
            }
        }
    }
    return outputGraph;
}


private static boolean coincideEndpoint(Graph inGraph, Vertex source, Vertex test){
    for (VertexIterator vtxIter = inGraph.adjacentVertices(source); vtxIter.hasNext();){
        Vertex endpoint = vtxIter.nextVertex();
        if (endpoint == test) return true;
    }
    return false;
}

private static Graph buildAdjacentGraph(Graph inGraph, int i){
    Vertex inVertex;
    Graph outGraph = new IncidenceListGraph();
    NodeSequence queue = new NodeSequence();
    NodeSequence inputQueue = new NodeSequence();
    NodeSequence outputQueue = new NodeSequence();
    VertexIterator vtxIter,vtxIterExt,vtxIterInt;
    vtxIter = inGraph.vertices();
    vtxIter = graphTools.goToVertex(vtxIter,i);
    inVertex = vtxIter.nextVertex();
    VertexInfo vtxInInfo = new VertexInfo();
    vtxInInfo=(VertexInfo)inVertex.element();
    Vertex firstNode = outGraph.insertVertex(vtxInInfo);
    for (vtxIterExt=inGraph.adjacentVertices(inVertex); vtxIterExt.hasNext();){
        VertexInfo vtxInfo = new VertexInfo();
        Vertex vertexExt = vtxIterExt.nextVertex();
        inputQueue.insertLast(vertexExt);
        vtxInfo=(VertexInfo)vertexExt.element();
        Vertex addedNode = outGraph.insertVertex(vtxInfo);
        outputQueue.insertLast(addedNode);
        queue.insertLast(addedNode);
        EdgeInfo edgeInfo = new EdgeInfo();
        Edge edge = inGraph.aConnectingEdge(inVertex,vertexExt);
     edgeInfo=(EdgeInfo)edge.element();
        outGraph.insertEdge(firstNode,addedNode,edgeInfo);
    }
    // 2-hop neighbors
    while (inputQueue.size()>0){
        //takes the element formerly stored at the first position
        Vertex vtxInput = (Vertex)inputQueue.removeFirst();
        Vertex vtxOutput = (Vertex)outputQueue.removeFirst();
        for (VertexIterator iter=inGraph.adjacentVertices(vtxInput); iter.hasNext();){
            Vertex vtx = iter.nextVertex();
            if(vtx!=inVertex){
                Vertex nextVtx = graphTools.getNodeWhichName(
              ((VertexInfo)vtx.element()).getName(),outGraph);
                if (nextVtx == null){
                    nextVtx = outGraph.insertVertex((VertexInfo)vtx.element());
                }
                if (!outGraph.areAdjacent(vtxOutput,nextVtx)){
```

```
                    Edge nextEdge = inGraph.aConnectingEdge(vtxInput,vtx);
                    outGraph.insertEdge(vtxOutput,nextVtx,(EdgeInfo)nextEdge.element());
                }
            }
        }
    }
    // 1-hop edges and edges between those neighbors
    while (queue.size()>0){
        //takes the element formerly stored at the first position
        Vertex principalNode = (Vertex)queue.removeFirst();
        ObjectIterator queueElements = queue.elements();
        while (queueElements.hasNext()){
            Vertex nextNode = (Vertex)queueElements.nextObject();
            if(inGraph.areAdjacent(graphTools.getNodeWhichName(
            ((VertexInfo)principalNode.element()).getName(),inGraph),
            graphTools.getNodeWhichName(
            ((VertexInfo)nextNode.element()).getName(),inGraph))){
                Edge edgeFounded = inGraph.aConnectingEdge(
                graphTools.getNodeWhichName(
                ((VertexInfo)principalNode.element()).getName(),inGraph),
                graphTools.getNodeWhichName(
                ((VertexInfo)nextNode.element()).getName(),inGraph));
                outGraph.insertEdge(principalNode,nextNode,(EdgeInfo)edgeFounded.element());
            }
        }
    }
    return outGraph;
  }
}
```

```
/*
 * MST.java
 */

import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import jdsl.core.ref.*;
import java.util.*;
import java.text.*;
import java.io.*;
/**
 * Contains methods to obtain a Minimum Spanning Tree (MST) from a given graph using the Prim
 * algorithm. Actually, two methods were implemented, one using the Euclidean distance between
 * nodes as edges-weight function and other using the delay.
 * @author  Tania Pérez
 */

public class MST {
    /**Obtaing the MST for a given input graph, using the Euclidean distance as weight function.
     * @param outGraph  the resulted MST
     * @param input     the input graph used to compute the MST
     * @param node      the initial node in order to start expanding the tree
     */
    public void findMST(Graph outGraph, Graph input, Vertex node) {
        expand(input,node);
        while (!allKnown(input)) {
            Vertex next = getMinDistance(input);
            if (next == null) {
                System.err.println("Graph is not connected.  Please add some edges!");
                return;
            }
            expand(input,next);
        }
        VertexIterator vtxIterIn = input.vertices();
        while (vtxIterIn.hasNext()) {
            Vertex initialNode = null, finalNode = null;
            Vertex vertex=vtxIterIn.nextVertex();
            if ((((VertexInfo)vertex.element()).getFrom()) != null) {
                if (!(((VertexInfo)vertex.element()).wasAdded())) {
                    ((VertexInfo)vertex.element()).added(true);
                    initialNode = outGraph.insertVertex((VertexInfo)vertex.element());
                    ((VertexInfo)initialNode.element()).name(((VertexInfo)vertex.element()).getName());
                }else{
                    initialNode = graphTools.getNodeWhichName
                                  (((VertexInfo)vertex.element()).getName(),outGraph);
                }
                Vertex vertex2 = ((VertexInfo)vertex.element()).getFrom();
                if (!(((VertexInfo)vertex2.element()).wasAdded())) {
                    ((VertexInfo)vertex2.element()).added(true);
                    finalNode = outGraph.insertVertex((VertexInfo)vertex2.element());
                }else{
                    finalNode = graphTools.getNodeWhichName
                                (((VertexInfo)vertex2.element()).getName(),outGraph);
                }
                EdgeInfo edgeInfo = new EdgeInfo();
                Edge edge = input.aConnectingEdge(vertex, vertex2);
                edgeInfo = (EdgeInfo)edge.element();
```

```
                outGraph.insertEdge(initialNode,finalNode,edgeInfo);
            }
        }
        VertexIterator vtxIterReset = input.vertices();
        while (vtxIterReset.hasNext()) {
            Vertex vtxReset = vtxIterReset.nextVertex();
            ((VertexInfo)vtxReset.element()).added(false);
            ((VertexInfo)vtxReset.element()).distance(Double.MAX_VALUE);
            ((VertexInfo)vtxReset.element()).from(null);
            ((VertexInfo)vtxReset.element()).known(false);
        }
    }

    // returns true if all of the nodes have been expanded
    private boolean allKnown(Graph inputGraph) {
        VertexIterator vtxIterIn;
        vtxIterIn = inputGraph.vertices();
        while (vtxIterIn.hasNext()) {
            Vertex vertex=vtxIterIn.nextVertex();
            if (!(((VertexInfo)vertex.element()).getKnown()))
                return false;
        }
        return true;
    }

    // returns the closest adjacent node to the current tree
    private Vertex  getMinDistance(Graph inputGraph) {
        Vertex minNode = null;
        double minCost = Double.MAX_VALUE;
        VertexIterator vtxIterIn;
        vtxIterIn = inputGraph.vertices();
        while (vtxIterIn.hasNext()) {
            Vertex vertex=vtxIterIn.nextVertex();
            if (!(((VertexInfo)vertex.element()).getKnown())) {
                // if the edge does not exist then the distance will
                // be Integer.MAX_VALUE, so there is no need to explicitly
                // test for adjacent edges
                if ((((VertexInfo)vertex.element()).getDistance() < minCost) {
                    minCost = ((VertexInfo)vertex.element()).getDistance();
                    minNode = vertex;
                }
            }
        }
        return minNode;
    }

    // expands the given node, putting all of its neighbors into the tables
    // and marks the current node as seen
    private void expand(Graph inputGraph, Vertex /*int*/ node) {
        double d;
        VertexIterator vtxIterIn;
        vtxIterIn = inputGraph.vertices();
        ((VertexInfo)node.element()).known(true);
        while (vtxIterIn.hasNext()) {
            Vertex vertex=vtxIterIn.nextVertex();
            if (!(((VertexInfo)vertex.element()).getKnown())) {
                d = getEdge(inputGraph, node, vertex);
```

```
                if (d < ((VertexInfo)vertex.element()).getDistance()) {
                    ((VertexInfo)vertex.element()).distance(d);
                    ((VertexInfo)vertex.element()).from(node);
                }
            }
        }
    }
// uses the from array to determine which edges are in the MST
 // uses the distance array to calculate the weights
 private double getEdge(Graph inputGraph, Vertex vtxIni , Vertex vtxFin ){
     Edge edge;
     double distance;
     if(inputGraph.areAdjacent(vtxIni, vtxFin)){
         edge = inputGraph.aConnectingEdge(vtxIni, vtxFin);
         distance=((EdgeInfo)edge.element()).getLenght();
     }
     else
         distance = Double.MAX_VALUE;
     return distance;
 }


 /**Obtaing the MST for a given input graph, using the edges' delay as weight function.
  * @param outGraph  the resulted MST
  * @param input     the input graph used to compute the MST
  * @param node      the initial node in order to start expanding the tree
  */
 public void  findMSTwithDelay(Graph outGraph, Graph input, Vertex node){
     expandWithDelay(input,node);
     while (!allKnown(input)) {
         Vertex next = getMinDelay(input);
         if (next == null) {
             System.err.println("Graph is not connected.  Please add some edges!");
             return;
         }
         expandWithDelay(input,next);
     }
     VertexIterator vtxIterIn = input.vertices();
     while (vtxIterIn.hasNext()){
         Vertex initialNode = null, finalNode = null;
         Vertex vertex=vtxIterIn.nextVertex();
         if ((((VertexInfo)vertex.element()).getFrom() != null){
             if (!(((VertexInfo)vertex.element()).wasAdded())){
                 ((VertexInfo)vertex.element()).added(true);
                 initialNode = outGraph.insertVertex((VertexInfo)vertex.element());
             }else{
                 initialNode = graphTools.getNodeWhichName
                               (((VertexInfo)vertex.element()).getName(),outGraph);
             }
             Vertex vertex2 = ((VertexInfo)vertex.element()).getFrom();
             if (!(((VertexInfo)vertex2.element()).wasAdded())){
                 ((VertexInfo)vertex2.element()).added(true);
                 finalNode = outGraph.insertVertex((VertexInfo)vertex2.element());
             }else{
                 finalNode = graphTools.getNodeWhichName
                             (((VertexInfo)vertex2.element()).getName(),outGraph);
             }
```

```
                EdgeInfo edgeInfo = new EdgeInfo();
                Edge edge = input.aConnectingEdge(vertex2, vertex);
                edgeInfo = (EdgeInfo)edge.element();
                outGraph.insertEdge(initialNode,finalNode,edgeInfo);
            }
        }
        VertexIterator vtxIterReset = input.vertices();
        while (vtxIterReset.hasNext()){
            Vertex vtxReset = vtxIterReset.nextVertex();
            ((VertexInfo)vtxReset.element()).added(false);
            ((VertexInfo)vtxReset.element()).delay(Double.MAX_VALUE);
            ((VertexInfo)vtxReset.element()).from(null);
            ((VertexInfo)vtxReset.element()).known(false);
        }
    }

    // returns the closest adjacent node to the current tree
    private Vertex getMinDelay(Graph inputGraph) {
        Vertex minNode = null;
        double minCost = Double.MAX_VALUE;
        VertexIterator vtxIterIn;
        vtxIterIn = inputGraph.vertices();
        while (vtxIterIn.hasNext()) {
            Vertex vertex=vtxIterIn.nextVertex();
            if (!(((VertexInfo)vertex.element()).getKnown())) {
                // if the edge does not exist then the delay will
                // be Integer.MAX_VALUE, so there is no need to explicitly
                // test for adjacent edges
                if (((VertexInfo)vertex.element()).getDelay() < minCost) {
                    minCost = ((VertexInfo)vertex.element()).getDelay();
                    minNode = vertex;
                }
            }
        }
        return minNode;
    }

// expands the given node, putting all of its neighbors into the tables
// and marks the current node as seen
    private void expandWithDelay(Graph inputGraph, Vertex node) {
        double d;
        VertexIterator vtxIterIn;
        vtxIterIn = inputGraph.vertices();
        ((VertexInfo)node.element()).known(true);
        while (vtxIterIn.hasNext()) {
            Vertex vertex=vtxIterIn.nextVertex();
            if (!(((VertexInfo)vertex.element()).getKnown())) {
                d = getDelayOfEdge(inputGraph, node, vertex);
                if (d < ((VertexInfo)vertex.element()).getDelay()) {
                    ((VertexInfo)vertex.element()).delay(d);
                    ((VertexInfo)vertex.element()).from(node);
                }
            }
        }
    }
```

```
    private double getDelayOfEdge(Graph inputGraph, Vertex vtxIni , Vertex vtxFin ) {
            Edge edge;
            double delay ;
            if(inputGraph.areAdjacent(vtxIni, vtxFin)) {
                edge = inputGraph.aConnectingEdge(vtxIni, vtxFin);
                delay=((EdgeInfo)edge.element()).getDelay();
            }
            else
            delay = Double.MAX_VALUE;
        return delay ;
        }
}
```

```
/*
 * Prim.java
 */

import jdsl.graph.api.*;
import jdsl.graph.algo.*;
import java.io.*;
import java.util.*;
import jdsl.graph.ref.*;
import jdsl.core.api.*;
import jdsl.core.ref.*;
/**Implementation of Prim's algorithm using the template-method pattern: IntegerPrimTemplate
defined in JDSL 2.
 * Actually <code>execute()</code> implements Prim's algorithm.
 *
 * @version JDSL 2
 */

public class Prim extends IntegerPrimTemplate {
    private Graph output=new IncidenceListGraph();

    /**It gets a positive weight for every edge in the graph.
     * This method gets called by the algorithm when the algorithm needs to
     * know the weight of an edge. Prim's algorithm cannot handle negative weights.
     * In our case, the weight is the Euclidean distance between nodes or
     * the delay in the connection.
     * @param e edge for which the algorithm needs to know a weight
     * @return the weight value for e
     */
    protected int weight (Edge e) {
        int distance = (int)((EdgeInfo)e.element()).getLenght();
        return distance;
    }

    /**Called every time a vertex with distance INFINITY comes off the priority queue.
     * When it has been called once, it should subsequently be called for all remaining vertices,
     * until the priority queue is empty.
     * @param v vertex which the algorithm just found to be unreachable from the source
     */
    protected void vertexNotReachable(Vertex v){
        System.out.println();
        System.out.println("ERROR: Graphic disconnected");
        System.exit(0);
    }

    /**It is executed when a vertex is added to the minimum spanning tree.
     * The algorithm calls this method at most once per vertex, after the vertex
     * has been "finished" (i.e., when the path from s to the vertex is known).
     * The vertex will never again be touched or considered by the algorithm.
     * @param v vertex that the algorithm just finished
     * @param vparent edge leading into v in the minimum spanning tree
   * @param treeWeight the total weight of all edges known to be in the tree at this point
   * in the execution of the algorithm, including vparent
   */
    protected void treeEdgeFound(Vertex v, Edge vparent, int treeWeight){
        int start,end;
        Integer x = new Integer(0);
```

```
        StringTokenizer st;
        if (vparent!= null){
            st = new StringTokenizer(((EdgeInfo)vparent.element()).getName(),"-");
            start = x.parseInt(st.nextToken());
            end = x.parseInt(st.nextToken());
            output.insertEdge(graphTools.getNodeWhichName(""+start,output),
                              graphTools.getNodeWhichName(""+end,output),
                              (EdgeInfo)vparent.element());
        }
    }

    /**Runs the minimum spanning tree algorithm on a graph started in vertex.
    * @param graph    the input graph for finding the minimum spanning tree
    * @param vertex   the Vertex at which to start the algorithm
    */
    public Graph execute(InspectableGraph graph, Vertex vertex){
        output = graphTools.copyNodesFrom((Graph)graph);
        super.executeAll(graph,vertex);
        return (Graph)output;
    }
}
```

```
/*
 * RNG.java
 */

import jdsl.core.ref.*;
import jdsl.core.api.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import java.io.*;

/**
 * Contains methods for obtaining a Relative Neighborhood Graph from a
 * given graph. Actually, two algorithms were implemented for RUG, although
 * final result is the same, difference arises when managing edges to be
 * deleted.
 * <p>
 * <code>applyRNG</code> does not delete edges until whole process is
 * finished.
 * <code>applyRNG2</code> starts with a Graph only containing nodes and for
 * which edges are attached as soon as they are labeled as connectors of
 * two neighbor nodes
 * <p>
 * Algorithm for RNG is based work presented by G. Toussaint;
 * The RNG of a finite planar set. 1980
 *
 */
public class RNG {

    /**
     * Obtains RNG from a given graph it works only when input graph is a
     * RUG due to intrinsec properties of RUG
     * @param inputGraph      the graph containing the set of nodes and
     *                        edges to generate RNG
     * @return                The Relative Neighborhood Graph of the inputGraph
     */
    public Graph getRNG(Graph inputGraph){
        Vertex vIni;
        Vertex vFin;
        VertexIterator vtxIterIJ, vtxIterK;
        EdgeInfo edgeInfo;
        Graph RNG;
        double dij, dkmax, dk1, dk2;
        EdgeIterator iter1, iter2;
        boolean exit;
        int count, i;
        RNG = new IncidenceListGraph();
        //copy nodes from inputGraph
        RNG=graphTools.copyNodesFrom(inputGraph);
        vtxIterIJ = RNG.vertices();
        //covers all potential edges
        for(i=0; i<RNG.numVertices()-1; i++){
            //forwards to desired node i
            vtxIterIJ=graphTools.goToVertex(vtxIterIJ,i);
            vIni=vtxIterIJ.nextVertex();
            while (vtxIterIJ.hasNext()){
                vFin=vtxIterIJ.nextVertex();
                //vtxIterK is the one for dk1 and dk2
```

```
            vtxIterK = RNG.vertices();
            vtxIterK.reset();
            dij=graphTools.lenghtEdge(vIni, vFin);
            count = 0;
            //exit becomes true if exists a closer node k for making
            //conection between node i and j
            exit = false;
            //cover all nodes but not i neither j
            while ((vtxIterK.hasNext()) && (exit != true)){
                vtxIterK.nextVertex();
                if ((vtxIterK.vertex()!= vIni) && (vtxIterK.vertex()!=vFin)){
                    dk1=graphTools.lenghtEdge(vtxIterK.vertex(), vIni);
                    dk2=graphTools.lenghtEdge(vtxIterK.vertex(), vFin);
                    dkmax=java.lang.StrictMath.max(dk1, dk2);
                    if (dij>dkmax) exit = true;
                    //if dij is longer than dkmax node k is a relative
                    //neighbor edge IJ must not be created
                }
            }
            if (exit == false){
                //any lenght shorter than dij was found
                //an edge from node i to node j can be created
                //edge name format is:
                //node_name_star-node_name_end
                edgeInfo=new EdgeInfo();
                edgeInfo.lenght(graphTools.lenghtEdge(vIni, vFin));
                edgeInfo.name(((VertexInfo)vIni.element()).getName()+
                        "-"+((VertexInfo)vFin.element()).getName());
                RNG.insertEdge(vIni, vFin, edgeInfo);
            }
        }
    }
    return RNG;
}

/**
 * Applies RNG algorithm to a given graph,
 * can use any kind of connected graph as input.
 * On this implementation, just one Graph is used,
 * RNG algorithm is applied over entire graph,
 * edges to be deleted are marked and deleted
 * just once the algorithm has covered all edges,
 * this same graph is returned as output.
 * @param inputGraph      the graph containing over which apply RNG algorithm
 * @return                The Relative Neighborhood Graph of the inputGraph
 */
public Graph applyRNG(Graph inputGraph){
    Vertex vIni;
    Vertex vFin;
    VertexIterator vtxIterIJ, vtxIterK;
    EdgeInfo edgeInfo, edgeInfoKI, edgeInfoKJ;
    double dij, dkmax, dk1, dk2;
    EdgeIterator edgeIter;
    Edge edge, edgeIJ, edgeKI, edgeKJ;
    boolean exit;
    int count, i, currNode=1;
    System.out.println("\nAplying RNG algorithm...");
```

```
vtxIterIJ = inputGraph.vertices();
//vtxIterK is the one for dk1 and dk2
vtxIterK = inputGraph.vertices();
//covers all potential edges
for(i=0; i<inputGraph.numVertices()-1; i++){
    //forwards to desired index i
    vtxIterIJ=graphTools.goToVertex(vtxIterIJ,i);
    vIni=vtxIterIJ.nextVertex();
    currNode++;
    //cover all remain nodes starting at i
    //i stays constant, j steps one every iteration
    //ij represents a potencial edge
    while (vtxIterIJ.hasNext()){
        vFin=vtxIterIJ.nextVertex();
        //checks if connection exists between nodes i and j
        if(inputGraph.areAdjacent(vIni, vFin)){
            dij=((EdgeInfo)inputGraph.aConnectingEdge(vIni,
                                                    vFin).element()).getLenght();
            vtxIterK.reset();
            exit = false;
            //exit becomes true if exists a closer node k for making
            //conection between node i and j
            //cover all vertex but not i neither j
            while ((vtxIterK.hasNext()) && (exit != true)){
                vtxIterK.nextVertex();
                if ((vtxIterK.vertex()!= vIni) &&
                    (vtxIterK.vertex()!=vFin)){
                    //checks if connection between nodes (i k) and (j k) exists
                    if((inputGraph.areAdjacent(vtxIterK.vertex(),vIni)) &&
                      (inputGraph.areAdjacent(vtxIterK.vertex(),vFin))){
                       //allows knowing lenght of edge ki
                       edgeKI=inputGraph.aConnectingEdge(vtxIterK.vertex(),vIni);
                       edgeInfoKI = (EdgeInfo)edgeKI.element();
                       dk1=edgeInfoKI.getLenght();
                       //allows knowing lenght of edge kj
                       edgeKJ=inputGraph.aConnectingEdge(vtxIterK.vertex(),vFin);
                       edgeInfoKJ = (EdgeInfo)edgeKJ.element();
                       dk2=edgeInfoKJ.getLenght();
                       dkmax=java.lang.StrictMath.max(dk1, dk2);
                       if (dij>dkmax){
                                exit = true;
                                //if dij is longer than dkmax node k is a
                                //relative neighbor edge IJ must be deleted
                                edgeIJ = inputGraph.aConnectingEdge(vIni, vFin);
                                ((EdgeInfo)edgeIJ.element()).markDeleted(true);
                                //marks for deleting
                       }
                    }
                }
            }
        }
    }
}
edgeIter = inputGraph.edges();
//removes all edges marked
while(edgeIter.hasNext()){
    edge=edgeIter.nextEdge();
```

```
            if (((EdgeInfo)edge.element()).isMarked()) {
                inputGraph.removeEdge(edge);
            }
        }
        return inputGraph;
    }

    /**
     * Applies RNG algorithm to a given graph, can use any kind of connected graph as input.
     * On this implementation, outputGraph is generated on fly. RNG algorithm is applied over
     * each pair of nodes, and just relative neighbors are passed to outputGraph.
     * @param inputGraph      the graph containing over which apply RNG algorithm
     * @return                The Relative Neighborhood Graph of the inputGraph
    */
    public Graph applyRNG2(Graph inputGraph){
        Vertex vIni;
        Vertex vFin;
        VertexIterator vtxIterIJ, vtxIterK;
        EdgeInfo edgeInfo, edgeInfoKI, edgeInfoKJ;
        double dij, dkmax, dk1, dk2;
        EdgeIterator edgeIter;
        Edge edge, edgeIJ, edgeKI, edgeKJ;
        Graph outputGraph;
        boolean found;
        int count, i, currNode=1;
        System.out.println("\nAplying RNG algorithm...");
        vtxIterIJ = inputGraph.vertices();
        //vtxIterK is the one for dk1 and dk2
        vtxIterK = inputGraph.vertices();
        outputGraph = graphTools.copyNodesFrom(inputGraph);
        //covers all potential edges
        for(i=0; i<inputGraph.numVertices()-1; i++){
            //forwards to desired index i
            vtxIterIJ=graphTools.goToVertex(vtxIterIJ,i);
            vIni=vtxIterIJ.nextVertex();
            currNode++;
            //cover all remain nodes starting at i
            //i stays constant, j steps one every iteration
            //ij represents a potencial edge
            while (vtxIterIJ.hasNext()){
                vFin=vtxIterIJ.nextVertex();
                //checks if connection exists between nodes i and j
                if(inputGraph.areAdjacent(vIni, vFin)){
                    dij=((EdgeInfo)inputGraph.aConnectingEdge(vIni,  vFin).element()).getLenght();
                    vtxIterK.reset();
                    found = false;
                    //exit becomes true if exists a closer node k for making
                    //conection between node i and j
                    //cover all vertex bur not i neither j
                    while ((vtxIterK.hasNext()) && (found != true)){
                        vtxIterK.nextVertex();
                        if ((vtxIterK.vertex()!= vIni) && (vtxIterK.vertex()!=vFin)){
                            //checks if connection between nodes (i k) and (j k) exists
                            if((inputGraph.areAdjacent(vtxIterK.vertex(),vIni)) &&
                              (inputGraph.areAdjacent(vtxIterK.vertex(),vFin))){
                               //allows knowing lenght of edge ki
                               edgeKI=inputGraph.aConnectingEdge(vtxIterK.vertex(),vIni);
```

```
                        edgeInfoKI = (EdgeInfo)edgeKI.element();
                        dk1=edgeInfoKI.getLenght();
                        //allows knowing lenght of edge kj
                        edgeKJ=inputGraph.aConnectingEdge(vtxIterK.vertex(),vFin);
                        edgeInfoKJ = (EdgeInfo)edgeKJ.element();
                        dk2=edgeInfoKJ.getLenght();
                        dkmax=java.lang.StrictMath.max(dk1, dk2);
                        if (dkmax<dij){
                                found = true;
                                //if some value dkmax is shorter than dij
                                //k is a relative node from i found
                                edgeIJ = inputGraph.aConnectingEdge(vIni, vFin);
                                ((EdgeInfo)edgeIJ.element()).markDeleted(true);
                                //marks for deleting
                        }
                    }
                }
            }
            if (!found){
                //any value dkmax was shorter than dij
                //edge ij can be created
                Vertex nodeI, nodeJ;
                EdgeInfo edInf;
                nodeI = graphTools.cloneNode(vIni, outputGraph);
                nodeJ = graphTools.cloneNode(vFin, outputGraph);
                edgeIJ = inputGraph.aConnectingEdge(vIni, vFin);
                edInf = (EdgeInfo)edgeIJ.element();
                outputGraph.insertEdge(nodeI, nodeJ, edInf);
            }
        }
    }
    return outputGraph;
  }
}
```

```
/*
 * RNGQ.java
 */

import jdsl.core.ref.*;
import jdsl.core.api.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import java.io.*;

/**
 * Contains methods for obtaining a Relative Neighborhood Graph with Quadrilateral removals
 * from a given graph.
 * <p>
 * The quadrilateral removals operation is applied on quadrangles formed when a two hop
 * neighbor has two or more paths to given node. The edge with largest cost in any
 * quadrilateral is removed.
 * <p>
 * Algorithm for RNGQ is based work presented by Liu et al. in
 * IEEE Transactions on parallel and distributed systems, 2005.
 *
 */

public class RNGQ {
  Graph gOpt=new IncidenceListGraph();
  double [] weight;
  String [] neighbor;
   Vertex vtxA, vtxB, vtxS, vtxTestS, vtxTestBk, vtxBkMin;
   /**
   * Applies RNGQ algorithm to a given graph,
   * can use any kind of connected graph as input.
   * On this implementation, just one Graph is used,
   * RNGQ algorithm is applied over entire graph,
   * edges to be deleted are marked and deleted
   * just once the algorithm has covered all edges,
   * this same graph is returned as output.
   *
   * @param inputGraph      the graph containing over which apply RNGQ
   *                        algorithm
   * @return                The Relative Neighborhood Graph with Quadrilateral
   *                        removals from a given input graph
   *
   */
   public Graph getRNGQ(Graph inputGraph){
     Graph gNor=new IncidenceListGraph();
     Graph gRNGQ=new IncidenceListGraph();
     RNG gRNG=new RNG();
     VertexIterator vtxIterBk, vtxIterBkBack, vtxIterSk, vtxIterS;
     boolean found;
     int w;
     gNor=graphTools.copyGraph(inputGraph);
     gOpt=gRNG.applyRNG2(gNor);
     for(int i = 0; i < gOpt.numVertices(); i++){
         vtxA = graphTools.getNodeWhichName(""+i,gOpt);
         weight = new double[gOpt.degree(vtxA)];
         neighbor = new String[gOpt.degree(vtxA)];
         vtxIterBk=gOpt.adjacentVertices(vtxA);
```

```
        while (vtxIterBk.hasNext()){
            w=0;
            found = false;
            vtxB=vtxIterBk.nextVertex();
            for (int j=0;j<(gOpt.degree(vtxA));j++){
                weight[j]=Double.MAX_VALUE;
                neighbor[j]=null;
            }
            if (gOpt.areAdjacent(vtxA, vtxB)){
                double weightAB = ((EdgeInfo)gOpt.aConnectingEdge(vtxA, vtxB).element()).
                                  getLenght();
                vtxIterSk=gOpt.adjacentVertices(vtxB);
                while (vtxIterSk.hasNext()){
                    vtxS=vtxIterSk.nextVertex();
                    if (vtxS!=vtxA && gOpt.areAdjacent(vtxA,vtxB)){
                            double weightBS= ((EdgeInfo)gOpt.aConnectingEdge(vtxB,vtxS).element()).
                                             getLenght();
                            weight[w]=java.lang.StrictMath.max(weightAB, weightBS);
                            neighbor[w]=((VertexInfo)vtxB.element()).getName();
                            vtxIterBkBack=gOpt.adjacentVertices(vtxA);
                            while(vtxIterBkBack.hasNext()){
                              vtxTestBk=vtxIterBkBack.nextVertex();
                              if ((vtxB!=vtxTestBk)&&
                                (gOpt.areAdjacent(vtxTestBk,vtxS))&&
                                (gOpt.areAdjacent(vtxA,vtxTestBk))){
                                found = true;
                                w++;
                                double weightABk={(EdgeInfo)gOpt.aConnectingEdge
                                                  (vtxA, vtxTestBk).element()).getLenght();
                                double weightBkS={(EdgeInfo)gOpt.aConnectingEdge
                                                  (vtxTestBk, vtxS).element()).getLenght();
                                weight[w]=java.lang.StrictMath.max(weightABk, weightBkS);
                                neighbor[w]=((VertexInfo)vtxTestBk.element()).getName();
                              }
                            }
                            if (found){
                              int degreeA=gOpt.degree(vtxA);
                              deleteEdges(degreeA);
                              found=false;
                              w=0;
                              for (int j=0;j<degreeA;j++){
                                weight[j]=Double.MAX_VALUE;
                                neighbor[j]=null;
                              }
                            }
                        }
                    }
                }
            }
        }
    }
    return gOpt;
}
```

```
    private void deleteEdges(int neighbors){
        double minCost = Double.MAX_VALUE;
        double weightABk=Double.MIN_VALUE,weightBkS=Double.MIN_VALUE;
        Edge edgeABk=null,edgeBkS=null;
        Vertex vtxMinCost=null, vtxBk;
          for (int i = 0; i < neighbors; i++) {
              if (weight[i] < minCost) {
                  minCost = weight[i];
                  vtxMinCost = graphTools.getNodeWhichName(neighbor[i],gOpt);
              }
          }
          for (int i = 0; i < neighbors; i++) {
              if (weight[i]!=Double.MAX_VALUE){
                  vtxBk=graphTools.getNodeWhichName(neighbor[i],gOpt),
                  if(vtxBk!=vtxMinCost){
                          edgeABk = gOpt.aConnectingEdge(vtxA,vtxBk);
                          weightABk=((EdgeInfo)edgeABk.element()).getLenght();
                          edgeBkS = gOpt.aConnectingEdge(vtxBk,vtxS);
                          weightBkS=((EdgeInfo)edgeBkS.element()).getLenght();
                      if (weightABk>=weightBkS)
                          gOpt.removeEdge(edgeABk);
                      else
                          gOpt.removeEdge(edgeBkS);

                  }
              }
          }
      }
  }
```

```
/*
 * Rumor Mongering.java
 */

import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import jdsl.core.ref.*;
import java.util.*;
import java.util.Random;
import java.text.*;
import java.io.*;

/**
 * Contains methods to implement the Rumor Mongering protocol, according to Portmann
 * and Seneviratne article: The cost of application level broadcast in a fully
 * decentralized peer-to-peer network.
 */
public class RumorMongering {
    Graph inputGraph;
    //A Sequence based on a doubly-linked-list implementation.
    NodeSequence queue = new NodeSequence();
    int B_neighbors;
    int F_times;
    int numMessages = 0;
    int reach = 0;
    /**
     * Given the input graph and the parameters, this class obtain the cost
     * (numMessages/costFlooding) and the percentage of reached nodes due to
     * the Gossip protocol.
     * @param input    the graph containing the set of nodes and edges to apply
     *                 the Gossip protocol
     * @param B        number of neighbors, chosen at random, that each node knows
     *                 have not yet seen the message
     * @param F        number of times that each node could receive the message
     *                 in order to retransmit it
     * @return  null
     */
    public void BlindCounter(Graph input, int B, int F){
        Vertex vtxIni;  //Empty, typing interface for vertices.
        VertexIterator vtxIter;     //Iterator over a set of vertices.
        int numVertices, numEdges, startNode;
        Random rnd = new Random();
        inputGraph = input;
        B_neighbors = B;
        F_times = F;
        numVertices = inputGraph.numVertices();
        numEdges = inputGraph.numEdges();
        //return an iterator over all vertices in the graph
        vtxIter = inputGraph.vertices();
        //randomly generates a number node for starting flooding
        startNode = rnd.nextInt(numVertices);
        //forwards vertex iterator to starting node
        vtxIter = graphTools.goToVertex(vtxIter, startNode);
        //gets node to start flooding
        vtxIni = vtxIter.vertex();
        ((VertexInfo)vtxIni.element()).hasRcvMessage(true);
        ((VertexInfo)vtxIni.element()).incRcvMessages();
```

```
        ((VertexInfo)vtxIni.element()).incRcvMessages();
        //Queue keeps track of nodes sequence to forward messages
        queue.insertLast(vtxIni);
        System.out.println("");
        System.out.println("Vertex Initial "+ ((VertexInfo)vtxIni.element()).getName());
        //BlindCounter process begins
        BlindCounter();
        System.out.println("");
        double avdeg = (double)2*numEdges/numVertices;
        double cflooding = 1+(double)numVertices*(avdeg-1);
        System.out.println("total of messages forwarded: "+ numMessages);
        System.out.println("COST OF DOUBLE RUMOR MONGERING: "+
                           (double)numMessages*100/cflooding + " %");
        System.out.println("FLOODING MESSAGES: "+ (double)cflooding);
        System.out.println("total edges: "+ numEdges);
        vtxIter.reset();
        while(vtxIter.hasNext()){
            vtxIter.nextVertex();
            vtxIni = vtxIter.vertex();
            if (((VertexInfo)vtxIni.element()).getHasRcvMessage())
                reach++;
        }
        System.out.println("");
        System.out.println("total of nodes reached: "+ reach);
        System.out.println("REACH: "+ (double)reach*100/numVertices + " %");
        System.out.println("total nodes: "+ numVertices);
        System.out.println("B: "+ B_neighbors);
        System.out.println("F: "+ F_times);
        System.out.println("average degree: "+ avdeg);
    }
    //BlindCounter is a recursive function
    //will be executed while queue is not empty
    private void BlindCounter(){
        boolean makeAll;
        int selectedNode;
        int valid;
        ArrayList VertexList;
        Vertex node;
        Random rnd = new Random();
        //takes the element formerly stored at the first position
        Vertex senderNode = (Vertex)queue.removeFirst();
        //gets neighbors list for sender node
        VertexIterator neighborsList = inputGraph.adjacentVertices(senderNode);
        //gets the number of validneighbors for the senderNode
        VertexList = new ArrayList();
        ((VertexInfo)senderNode.element()).validVertexList(VertexList);
        int validNeighbors = 0;
        while (neighborsList.hasNext()){
            neighborsList.nextVertex();
            //takes one neighbor from neighbors list
            Vertex neighbor = neighborsList.vertex();
            //checks if this neighbor is valid for sending message
            if (!((VertexInfo)senderNode.element()).existFwdVertex(neighbor)){
                ((VertexInfo)senderNode.element()).addValidVertex(neighbor);
                validNeighbors++;
            }
        }
```

```
        int numNeighbors=inputGraph.degree(senderNode/*,EdgeDirection.OUT*/);
        if (validNeighbors <= B_neighbors){
            makeAll = true;
            valid = validNeighbors;
        }else{
            makeAll = false;
            valid = B_neighbors;
        }
        int i=0;
        while (i < valid){
            if (makeAll){
                node = ((VertexInfo)senderNode.element()).getValidVertex(i);
            }else{
                //randomly generates a number node for flooding
                selectedNode = rnd.nextInt(validNeighbors);
                node = ((VertexInfo)senderNode.element()).getValidVertex(selectedNode);
            }
            if(!((VertexInfo)senderNode.element()).existFwdVertex(node)){
                ((VertexInfo)senderNode.element()).addFwdVertex(node);
                ((VertexInfo)node.element()).addFwdVertex(senderNode);
                ((VertexInfo)node.element()).incRcvMessages();
                //this node has received the message
                ((VertexInfo)node.element()).hasRcvMessage(true);
                if (((VertexInfo)node.element()).getRcvMessages() <= F_times){
                    //insert neighbor in queue for future flooding
                    queue.insertLast(node);
                }
                i++;
                numMessages++;
            }
        }
        //if queue is not empty flooding process continues
        if (queue.size()>0)
            BlindCounter();
    }
}
```

```
/*
 * EdgeInfo.java
*/

import java.io.*;
import java.lang.*;

/**
 * Implements methods to store and retrieve information from an
 * <code>Edge</code> in a <code>Graph</code>.
 * Information stored in an <code>Edge</code> is its name, lenght and delay
 *
 */
public class EdgeInfo {
    protected String name;
    protected double lenght;
    protected boolean xyEdge=false;
    protected boolean markForDeleting;
    protected double delay = Double.MAX_VALUE;

    /**
    * Stores name and lenght edge
    * @param nameEdg    string containing edge name
    * @param lenghtEdg  edge lenght in double format
    *
    */
    public EdgeInfo(String nameEdg, double lenghtEdg){
      name=nameEdg;
      lenght=lenghtEdg;
    }

    /**
     * Standard constructor, used when name, lenght and delay have not been
     * calculated
     */
    public EdgeInfo(){
        lenght=0;
        delay = Double.MAX_VALUE;
        markForDeleting = false;
    }

    /**
     * Attaches edge name
     * @param nameEdge name of current edge
     */
    public void name(String nameEdge){
        name=nameEdge;
    }

    /**
   * Attaches lenght o f current edge
   * @param lenghtEdge  lenght of edge in double format
   */
    public void lenght(double lenghtEdge){
        lenght=lenghtEdge;
    }
```

```
    /**
     * Marks edge to be deleted
     * @param mark       contains <code>true</code> if edge must be
     *                   deleted otherwise contains <code>false</code>
     */
      public void markDeleted(boolean mark){
          markForDeleting = mark;
      }

     /**
     * Returns edge name
     */
      public String getName(){
          return name;
      }

      /**
       * Returns edge lenght
       */
      public double getLenght(){
          return lenght;
      }

      /**
     * Specifys that lenght information corresponds euclidean distance
     * between nodes
     */
      public void setXYEdge(){
          xyEdge=true;
      }

      /**
       * Checks if edge is marked for deletion
     * Returns <code>true</code> if edge needs to be deleted
     */
      public boolean isMarked(){
          return markForDeleting;
      }

      /**
       * Attaches delay of current edge
       * @param delayEdge    delay of edge in double format
       */
      public void delay(double delayEdge){
          delay = delayEdge;
      }

      /**
       * Returns edge's delay
     *
     */
      public double getDelay(){
          return delay;
      }
}
```

```
/*
 * VertexInfo.java
 */

import java.io.*;
import java.lang.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import java.util.*;

/**
 * Implements methods needed for storing and retrieving information
 * from nodes in a given Graph.
 * <p>
 * On each vertex it is possible to attach:
 * <li>name
 * <li>x-y coordiantes
 * <li>number of forwarded messages
 * <li>last message forwarded
 * among other particular properties that depends of each algorithm
 * <p>
 *
 */
public class VertexInfo {
    private String name=null;
    private int x=-1;
    private int y=-1;
    private Vertex sourceVertex=null;
    private int FwdMessages = 0;
    private int message;
    private List list = new ArrayList();
    private List validList = new ArrayList();
    private int RcvMessage = 0;
    private Vertex FwdVertex=null;
    private Vertex validVertex = null;
    private boolean receive = false;
    private double distance = Double.MAX_VALUE;
    private double delay = Double.MAX_VALUE;
    private Vertex from=null;
    private boolean known = false;
    private boolean delete = true;
    private boolean added = false;

    /**Used in MST class to define that the current node was already added in the tree
     * @param add true if it belongs to the MST
     */
    public void added(boolean add){
        added=add;
    }

    /**Used in MST class to inform if the current node was added in the tree
     * @return true if it already belongs to the MST, false otherwise
     */
    public boolean wasAdded(){
        return added;
    }
```

```
    /**Used in MST class to assign the low cost (minimum distance) the node can be reached in the
current tree.
     * @param d minimum distance
     */
    public void distance (double d){
        distance=d;
    }

    /**Used in MST class to get the low cost (minimum distance) assigned to node
     * @return      minimum distance
     */
    public double getDistance (){
        return distance;
    }

    /**Used in MST class to assign the low cost (minimum delay) the node can be reached in the current
tree.
     * @param d minimum delay
     */
    public void delay (double value){
       delay=value;
    }

    /**Used in MST class to get the low cost (minimum delay) assigned to node
     * @return      minimum delay
     */
    public double getDelay (){
        return delay;
    }

    /**Used in MST class to assign the closest node this vertex can be reached
     * @param closest the closest node
     */
    public void from (Vertex closest){
        from=closest;
    }

    /**Used in MST class to get the closest node
     * @return  the closest node
     */
    public Vertex getFrom (){
        return from;
    }

    /**Used in MST class to mark the current node as visited
     * @param visited if this node has been seen
     */
    public void known(boolean visited){
        known = visited;
    }

    /**Used in MST class to know if the current node has been visited
     * @return true if the node has been seen, false otherwise
     */
    public boolean getKnown(){
        return known;
    }
```

```
/**Used in RumorMongering class to notify if the current node has already received the broadcast
 *  message
 * @param message true to mark the message has been received, false otherwise
 */
public void hasRcvMessage(boolean message){
    receive = message;
}

/**Used in RumorMongering class to inform if the current node has received the broadcast message
 * @return true if the message has been received, false otherwise
 */
public boolean getHasRcvMessage(){
    return receive;
}

/**Used in RumorMongering class to count the times the current node has received the broadcast
 *  message
 * in order to compare this value with the parameter F
 */
public void incRcvMessages(){
    RcvMessage++;
}

/**Used in RumorMongering class to get the times the current node has received the broadcast
 *  message
 * @return times the message has been received
 */
public int getRcvMessages(){
    return RcvMessage;
}

/**
 * Used in RumorMongering class. Attaches ForwardList = forward + source nodes
 * @param FwdList      list of the forwarders
 *
 */
public void FwdList(ArrayList FwdList){
    list = FwdList;
}

/**Used in RumorMongering class. Adds in the ForwardList the neighbor that is
 * waiting to be sended the broadcast message (forwarding-nodes)
 * @param vtx the neighbor to be added
 */
public void addFwdVertex(Vertex vtx){
    FwdVertex=vtx;
    if(!list.contains(FwdVertex))
        list.add(FwdVertex);
}

/**Used in RumorMongering class in order to check if a neighbor is already added
 * in the ForwardList
 * @param vtx the neighbor to check
* @return true if the neighbor has already received the broadcast message, false otherwise
*/
public boolean existFwdVertex(Vertex vtx){
```

```
        FwdVertex=vtx;
        if(list.contains(FwdVertex))
            return true;
        else
            return false;
    }

    /**Used in RumorMongering class. Attaches the list of all possible neighbors
     * that the current node could send the broadcast message
     * @param vvList the list that will contain the possible neighbors
     */
    public void validVertexList(ArrayList vvList){
        validList = vvList;
    }

    /**Used in RumorMongering class. Adds a neighbor in the ValidVertexList
    * @param vtx the node to add
    */
    public void addValidVertex(Vertex vtx){
        validVertex = vtx;
        validList.add(validVertex);
    }

    /**Used in RumorMongering class. Gets a neighbor from the ValidVertexList
     * @param i the position of the node in the list
     * @return the node
     */
    public Vertex getValidVertex(int i){
        int j = i;
        validVertex = (Vertex)validList.get(j);
        return validVertex;
    }

    /**Used in RumorMongering class. Returns the ValidVertexList
    */
    public List getValidVertexList(){
        return validList;
    }

    /**
     * Attaches node name
     * @param nameVer      name of the node
     */
    public void name(String nameVer){
        name=nameVer;
    }

    /**
     * Attaches x coordinate of node
     * @param xpos         x-coordiante
     */
    public void xCoor(int xpos){
        x=xpos;
    }

    /**
     * Attaches y coordinate of node
```

```
* @param ypos        y-coordiante
*/
public void yCoor(int ypos){
      y=ypos;
 }

 /**
  * Allows establishing source node in forwarding process
* @param vtx        node acting as source node in current broadcasting process
 */
 public void source(Vertex vtx){
      //source vertex updates only if current node
      //has not received a previous copy of message
      if (sourceVertex == null)
          sourceVertex = vtx;
 }

 /**
* Increments counter every time a node forwards a message
* @param vtx     node for which counters must be incremented
*/
 public void incFwdMessages(Vertex vtx){
      FwdMessages++;
 }

 /**
* Saves a copy of current message being forwarded
* @param mess     message to be broadcasted
*/
 public void sendMessage(int mess){
      message = mess;
 }

 /**
* Gets name of node
* @return       String containing node name
*/
 public String getName(){
      return name;
 }

 /**
  * Gets number of messages forwarded
* @return       number of messages forwarded
*/
 public int getFrwdMessages(){
      return FwdMessages;
 }

 /**
  * Allows knowing source node
  * @return       pointer to source node in broadcasting task
*/
 public Vertex getSourceNode(){
      return sourceVertex;
 }
```

```
    /**
     * Gets x-coordinate of node
     * @return        x-coordinate of node
     */
    public int getX() {
        return x;
    }

    /**
     * Gets y-coordinate of node
     * @return        y-coordinate of node
     */
    public int getY() {
        return y;
    }
}
```

```
/*
 * InternetMap.java
 */

import java.io.*;
import java.util.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import jdsl.graph.algo.*;
import jdsl.core.api.*;
import jdsl.core.ref.*;

/**
 * IntenetMap class contains all methods used to load into memory Graphs
 * generated by topology generators such as BRITE and Recursive.
 * In all cases, passed parameters need to establish source file from
 * which to load data and format of input data.
 * <p>
 * Once data has been loaded into memory, Graph is stored on an instance
 * variable of class InternetMap. InternetMap class also contains methods
 * to pass loaded Graph to other methods in different classes.
 * <p>
 * Although BRITE provides x-y coordiantes for each node, edges leght and
 * time delay for each connection, Recu rsive does not, so InternetMap
 * includes some methods to randomly generate x-y coordianates for each
 * node, delay between two nodes is assigned randomly in the range [0, 1]
 * to the edge joining them.
 *
 */
public class InternetMap {
    Graph internetGraph;
    int numNodes;
    int numEdges;
    String fileName;
    String fileType;
    BufferedReader inputFile;

    /**
     * Allows having access to Graph stored on instance variable of
    * <code>InternetMap</code>
    * @return   Graph containing Internet Map loaded by means of
    * <code>InternetMap</code> class
    */
    public Graph getGraph(){
        return internetGraph;
    }

    /**
     * Prepares input to be readed by further methods
     * @param file          name of file containing data to be loaded
     * @throws IOException   an exception if given file does not exists
     *
     */
    public InternetMap(String file) throws IOException{
        internetGraph=null;
        StringTokenizer fileToken;
        numNodes=0;
```

```
        if (!checkFile(file)) System.exit(1);
        fileName=file;
        // Prepare input and output files
        inputFile = new BufferedReader(new FileReader(fileName));
        File src = new File(fileName);
        fileToken = new StringTokenizer(src.getName(), ".");
        fileToken.nextToken();
        fileType = fileToken.nextToken();
        System.out.println("Internet Graph loaded from "+src.getName () +"...");
    }

    /**
     * Reads file generated by Recursive assigning a random x-y coordinate
     * to each node present on file.
     */
    public void distance()throws IOException{
        System.out.println("Generating Internet Grap h with distance...");
        getNumberNodes();
        insertNodesXY();
        insertEdges();
        inputFile.close();
        EdgeIterator elt = internetGraph.edges();
    }

    /**
     * Reads file generated by Recursive assigning a random delay in range
     * [0, 1] to each edge joining a pair of nodes.
     */
    public void delay() throws IOException{
        System.out.println("Generating Internet Graph with delay...");
        getNumberNodes();
        insertNodesDelay();
        insertEdges();
        inputFile.close();
    }

/**
     * Reads file generated by BRITE
     */
    public void briteXY() throws IOException{
        System.out.println("Generating Internet Graph with BRITE coordinates...");
        getNumberNodes();
        insertNodesBrite();
        insertEdges();
        inputFile.close();
        EdgeIterator elt = internetGraph.edges();
    }

    /**
     * Reads file generated by BRITE but without edges info in order to use it for
    * simulate a Wireless adhoc network
    */
    public void special() throws IOException{
        System.out.println("Generating Internet Graph with BRITE coordinates...");
        getNumberNodes();
        insertNodesBrite();
        defineEdges();
```

```
        inputFile.close();
        EdgeIterator eIt = internetGraph.edges();
    }

    /**
    * Allows knowing number of nodes present in input file, value is
    * stored on internal instance variable
    *
    */
    public void getNumberNodes() throws IOException {
        String lineFromFile;
        StringTokenizer st, fileToken;
        Integer x = new Integer(0);
        boolean exit;
        if (fileType.equalsIgnoreCase("nm")) {
            // Read file
            exit = false;
            while (((lineFromFile= inputFile.readLine()) != null)  && (!exit)) {
                if (lineFromFile.indexOf("NETWORK - GRAPH TOPOLOGY")!= -1) {
                    lineFromFile=inputFile.readLine();
                    st = new StringTokenizer(lineFromFile);
                    st.nextToken();
                    st.nextToken();
                    numNodes = x.parseInt(st.nextToken());
                    exit = true;
                }
            }
        } else if (fileType.equalsIgnoreCase("rec")) {
            exit = false;
            while (((lineFromFile= inputFile.readLine()) != null)  && (!exit)) {
                st = new StringTokenizer(lineFromFile);
                if ((st.nextToken()).compareTo("generating")==0)
                {
                    numNodes = x.parseInt(st.nextToken());
                    System.out.println("numNodes... "+numNodes);
                    st.nextToken();st.nextToken();
                    numEdges = x.parseInt(st.nextToken());
                    System.out.println("numEdges... "+numEdges);
                    exit = true;
                }
            }
        } else if (fileType.equalsIgnoreCase("brite")) {
            exit = false;
            while (((lineFromFile= inputFile.readLine()) != null)  && (!exit)) {
                st = new StringTokenizer(lineFromFile);
                st.nextToken();
                st.nextToken();
                numNodes = x.parseInt(st.nextToken());
                System.out.println("numNodes... "+numNodes);
                exit = true;
            }
        }
    }

    private void insertNodesXY() {
        int m = 100;                //space por generating graph 100x100
        int x;
```

```
        int y;
        VertexInfo vertex;
        //rnd is the random index for selecting nodes
        Random rnd = new Random();
        internetGraph  = new IncidenceListGraph();
        //this cycle for generating nodes' position
        for(int i=0; i<numNodes; i++){
            vertex = new VertexInfo();
            x=rnd.nextInt(m);
            y=rnd.nextInt(m);
            vertex.name(""+i);
            vertex.xCoor(x);
            vertex.yCoor(y);
            //inserts vertex selected into graph
            internetGraph.insertVertex(vertex);
        }
    }

    private void insertNodesDelay(){
        VertexInfo vertex;
        internetGraph  = new IncidenceListGraph();
        //this cycle for inserting nodes
        for(int i=0; i<numNodes; i++){
            vertex = new VertexInfo();
            vertex.name(""+i);
            //inserts vertex selected into graph
            internetGraph.insertVertex(vertex);
        }
    }

    private void insertNodesBrite() throws IOException{
        int nodeID=0, x, y;
        Integer NODEID, X, Y;
        Vertex tmpV;
        VertexInfo vertex;
        String lineFromFile;
        StringTokenizer st;
        ArrayList list;
        internetGraph  = new IncidenceListGraph();
        if (fileType.equalsIgnoreCase("brite")){
            while ((lineFromFile= inputFile.readLine()) != null){
                if (lineFromFile.indexOf("Nodes:")!= -1){
                    while (nodeID < numNodes-1){
                        lineFromFile = inputFile.readLine();
                        st = new StringTokenizer(lineFromFile);
                        NODEID = Integer.valueOf(st.nextToken());
                        X = Integer.valueOf(st.nextToken());
                        Y = Integer.valueOf(st.nextToken());
                        nodeID = NODEID.intValue();
                        x=X.intValue();
                        y=Y.intValue();
                        vertex = new VertexInfo();
                        vertex.name(""+nodeID);
                        vertex.xCoor(x);
                        vertex.yCoor(y);
                        list = new ArrayList();
```

```
                    vertex.FwdList(list);
                    tmpV = internetGraph.insertVertex(vertex);
                    vertex.addFwdVertex(tmpV);
                }
                break;
            }
        }
    } else {
        System.out.println("invalid format file");
    }
}

private void insertEdges()throws IOException{
    String lineFromFile;
    StringTokenizer st;
    Edge edge;
    Integer x = new Integer(0);
    String startNode;
    String endNode;
    Double DELAY;
    double delay;
    if (fileType.equalsIgnoreCase("nm")) {
        while ((lineFromFile= inputFile.readLine()) != null) {
            if (lineFromFile.indexOf("EDGES")!= -1) {
                while ((lineFromFile= inputFile.readLine()) != null) {
                    st = new StringTokenizer(lineFromFile);
                    startNode = st.nextToken();
                    endNode = st.nextToken();
                    Vertex star=getNode(startNode);
                    Vertex end=getNode(endNode);
                    createEdge(star, end);
                }
            }
        }
    } else if (fileType.equalsIgnoreCase("rec")) {
        for (int i=0; i<numEdges; i++) {
            lineFromFile= inputFile.readLine();
            st = new StringTokenizer(lineFromFile);
            startNode = st.nextToken();
            endNode = st.nextToken();
            Vertex star=getNode(startNode);
            Vertex end=getNode(endNode);
            createEdge(star, end);
        }
    } else if (fileType.equalsIgnoreCase("brite")) {
        while ((lineFromFile= inputFile.readLine()) != null) {
            if (lineFromFile.indexOf("Edges:")!= -1) {
                while ((lineFromFile= inputFile.readLine()) != null) {
                    st = new StringTokenizer(lineFromFile);
                    st.nextToken ();
                    startNode = st.nextToken();
                    endNode = st.nextToken();
                    st.nextToken ();
                    DELAY = Double.valueOf(st.nextToken());
                    delay = DELAY.doubleValue();
                    Vertex star=getNode(startNode);
                    Vertex end=getNode(endNode);
```

```
                    createEdge(start, end, delay);
                }
            }
        }
    }
}

private void defineEdges() {
    for (VertexIterator vtxIter_1 = internetGraph.vertices(); vtxIter_1.hasNext();) {
        Vertex start = vtxIter_1.nextVertex();
        for (VertexIterator vtxIter_2 = internetGraph.vertices(); vtxIter_2.hasNext();) {
            Vertex end = vtxIter_2.nextVertex();
            if (start != end) {
                if (!internetGraph.areAdjacent(start,end)) {
                    double lenght;
                    lenght=graphTools.lenghtEdge(end,start);
                    if (lenght <= 1000) {
                        EdgeInfo edgeInf = new EdgeInfo();
                        edgeInf.lenght=lenght;
                        edgeInf.delay = Math.random()*1000;
                        edgeInf.name(((VertexInfo)start.element()).getName()+"-"+
                                        ((VertexInfo)end.element()).getName());
                        internetGraph.insertEdge(start, end, edgeInf);
                    }
                }
            }
        }
    }
    plotNetwork plot = new plotNetwork(internetGraph,"Input Graph");
}

private Vertex getNode(String name) {
    boolean found;
    Vertex node=null;
    VertexIterator vIter= internetGraph.vertices();
    VertexInfo vInfo;
    found = false;
    while ((vIter.hasNext()) && (!found)) {
        vInfo = new VertexInfo();
        node = vIter.nextVertex();
        vInfo = (VertexInfo)node.element();
        if (vInfo.getName().equals(name)) {
            found = true;
        }
    }
    return node;
}

private void createEdge(Vertex a, Vertex b) {
    double lenght;
    EdgeInfo edgeInf = new EdgeInfo();
    VertexInfo A = (VertexInfo)a.element();
    VertexInfo B = (VertexInfo)b.element();
    if((isXYNode(a)) && (isXYNode(b))) {
        lenght=graphTools.lenghtEdge(a, b);
        edgeInf.setXYEdge();
        edgeInf.lenght=lenght;
```

```
                edgeInf.name(((VertexInfo)a.element()).getName()+"-"+
                        ((VertexInfo)b.element()).getName());
                internetGraph.insertEdge(a, b, edgeInf);
            }
            if((!isXYNode(a)) && (!isXYNode(b))){
                Random rnd = new Random();
                lenght = Math.random();
                edgeInf.lenght=lenght;
                edgeInf.name(((VertexInfo)a.element()).getName()+"-"+
                        ((VertexInfo)b.element()).getName());
                internetGraph.insertEdge(a, b, edgeInf);
            }
        }

        private void createEdge(Vertex a, Vertex b, double delay){
            double lenght;
            EdgeInfo edgeInf = new EdgeInfo();
            VertexInfo A = (VertexInfo)a.element();
            VertexInfo B = (VertexInfo)b.element();
            if((isXYNode(a)) && (isXYNode(b))){
                lenght=graphTools.lenghtEdge(a, b);
                edgeInf.setXYEdge();
                edgeInf.lenght=lenght;
                edgeInf.name(((VertexInfo)a.element()).getName()+"-"+
                        ((VertexInfo)b.element()).getName());
                edgeInf.delay(delay);
                internetGraph.insertEdge(a, b, edgeInf);
            }
        }

        /**
        * Allows to know if a node it is being used x-y coordinates or
        * delay as metric
        *
        */
        private boolean isXYNode(Vertex a){
            VertexInfo A = (VertexInfo)a.element();
            return((A.getX() != -1) && (A.getY() != -1));
        }

        //check file function
        private static boolean checkFile(String fileName) {
            File src = new File(fileName);
            if (src.exists()) {
                if (src.canRead()) {
                    if (src.isFile()) return(true);
                    else System.out.println("ERROR 3: File is a directory");
                }
                else System.out.println("ERROR 2: Access denied");
            }
            else System.out.println("ERROR 1: No such file "+src.getPath());
            return(false);
        }
}
```

```
/*
 * GraphTools.java
 */

import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import java.util.*;
import java.text.*;
import java.io.*;

/**
* Contains useful methods for working with graphs.
*
*/
public class graphTools {

    /**
     * gets distance between two nodes
     * @param a     node a
     * @param b     node b
     * @return      distance between a and b
     */
    public static double lenghtEdge(Vertex a, Vertex b){
        VertexInfo A = (VertexInfo)a.element();
        VertexInfo B = (VertexInfo)b.element();
        double dx=A.getX()-B.getX();
        double dy=A.getY()-B.getY();
        double lenght = java.lang.Math.sqrt(dx*dx+dy*dy);
        return lenght;
    }

    /**
     * Gets xy coordiantes from node name
     * @param vContent    node from which to obtain coordinates
     * @return            array containing [x,y] coordiantes
     */
    public static int[] getVertexXY(Vertex vContent){
        int[] xy=new int[2];
        String vName= vContent.toString();
        vName=vName.replace('',');
        String xyStr=vName.substring(20, vName.length());
        int finInX = xyStr.indexOf(',');
        String xStr=xyStr.substring(0,finInX);
        Integer xInt =new Integer(xStr);
        int xVal=xInt.parseInt(xStr);
        String yStr=xyStr.substring(finInX+1, xyStr.length());
        Integer yInt =new Integer(yStr);
        int yVal=yInt.parseInt(yStr);
        xy[0]=xVal;
        xy[1]=yVal;
        return xy;
    }

    /**
     * Allows moving back or forward through a vertexIterator,
     * thus going to a specific node
     * @param verIt      <code>VertexIterator</code> containing all vertices
```

```
     *                in the Graph
     * @param vertex     number ID of node to move to
     * @return           a <code>VertexIterator</code> forwarded to desired
     *                node
     */
    public static VertexIterator goToVertex(VertexIterator verIt, int vertex){
        verIt.reset();
        for (int i=0; i<vertex; i++){
            verIt.nextVertex();
        }
        return verIt;
    }

    /**
     * Makes copy of nodes in a Graph and puts them in a new Graph
     * @param inGraph    source node
     * @return           a graph containing only the nodes of input Graph
     */
    public static Graph copyNodesFrom(Graph inGraph){
        Graph outGraph = new IncidenceListGraph();
        VertexIterator vtxIter;
        for (vtxIter=inGraph.vertices(); vtxIter.hasNext();){
            VertexInfo vtxInfo = new VertexInfo();
            Vertex vertex = vtxIter.nextVertex();
            vtxInfo=(VertexInfo)vertex.element();
            outGraph.insertVertex(vtxInfo);
        }
        return outGraph;
    }

    /**
     * Copies a whole Graph
    * @param inputGraph    graph to be copied
    * @return              a copy of inputGraph
    */
    public static Graph copyGraph(Graph inputGraph){
        StringTokenizer st;
        String nameEndNode;
        String nameStarNode;
        Graph outputGraph = new IncidenceListGraph();
        VertexIterator vtxIter;
        EdgeIterator edgIter;
        for (vtxIter=inputGraph.vertices(); vtxIter.hasNext();){
            VertexInfo vtxInfo = new VertexInfo();
            Vertex vertex = vtxIter.nextVertex();
            vtxInfo=(VertexInfo)vertex.element();
            outputGraph.insertVertex(vtxInfo);
        }
        vtxIter = outputGraph.vertices();
        for (edgIter=inputGraph.edges(); edgIter.hasNext();){
            EdgeInfo edgInfo = new EdgeInfo();
            Edge edge = edgIter.nextEdge();
            edgInfo=(EdgeInfo)edge.element();
            st = new StringTokenizer(edgInfo.getName());
            nameStarNode =st.nextToken("-");
            nameEndNode = st.nextToken("-");
            Vertex starNode = getNodeWhichName(nameStarNode, outputGraph);
```

```
            Vertex endNode = getNodeWhichName(nameEndNode, outputGraph);
            Edge outEdge = outputGraph.insertEdge(startNode, endNode, edgInfo);
            ((EdgeInfo)outEdge.element()).name = edgInfo.getName();
        }
        return outputGraph;
    }

    /**
     * Looks for a specific node inside a VertexIterator using name of node
     * as key with a given vertex iterator allows look for a specific node
     * using its name node is returned for further use
     * @param name    name of the node to be searched
     * @param vtxIter  <code>VertexIterator</code> into which it looks for
     *                the node
     * @return       pointer to node found
     * @deprecated  Use instead public static Vertex getNodeWhichName(String name, Graph graph)
     */
    public static Vertex getNodeWhichName(String name, VertexIterator vtxIter){
        boolean exit;
        Vertex vertex;
        vertex = null;
        exit = false;
        while((vtxIter.hasNext()) || (!exit)){
            vertex = vtxIter.nextVertex();
            VertexInfo vtxInfo = new VertexInfo();
            vtxInfo=(VertexInfo)vertex.element();
            if(vtxInfo.getName().equals(name)){
                exit=true;
            }
        }
        vtxIter.reset();
        return vertex;
    }

    /**
    * Looks for a specific node inside a Graph using name of node as key
    * @param name    name of the node to be searched
    * @param vtxIter  <code>Graph</code> into which it looks for the node
    * @return       pointer to node found
    */
    public static Vertex getNodeWhichName(String name, Graph graph){
        Vertex vertex;
        VertexIterator vtxIter = graph.vertices();
        vertex = null;
        while(vtxIter.hasNext()){
            Vertex vtxWhichName = vtxIter.nextVertex();
            if((((VertexInfo)vtxWhichName.element()).getName()).compareTo(name)==0){
                vertex = vtxWhichName;
            }
        }
        return vertex;
    }

    /**
    * Compares number of edges and degree of two different graphs
    * @param gra1     graph1 to be compared
    * @param gra2     graph2 to be compared
```

```
 * @param output    name of file to write out comparison information
 * @exception  IOException   an exception if it is not possible to write in the file
 */
 public static void compare(Graph gra1, Graph gra2, FileWriter output)
     throws IOException{
     double avDeg1, avDeg2, savings;
     String pattern;
     pattern = "###.###";
     DecimalFormat formater = new DecimalFormat("###.##");
     System.out.println("\nResults:");
     output.write("\nResults:\n");
     avDeg1=(double)gra1.numEdges()*2/(double)gra1.numVertices();
     avDeg2=(double)gra2.numEdges()*2/(double)gra2.numVertices();
     savings = gra2.numEdges()*100/gra1.numEdges();
     System.out.println("Edges \tAv. Degree\tEdges (RNG)\tAv. Degree (RNG)\t%Cost");
     output.write("Edges\tAv. Degree\t\tEdges (RNG)\tAv. Degree (RNG)\t%Cost\n");
     System.out.println(gra1.numEdges()+"\t"+formater.format(avDeg1)+
                        "\t\t"+gra2.numEdges()+"\t\t"+
                        formater.format(avDeg2)+"\t\t\t"+savings);
     output.write(gra1.numEdges()+"\t"+formater.format(avDeg1)+"\t\t"+
                  gra2.numEdges()+"\t\t"+formater.format(avDeg2)+
                  "\t\t\t"+savings);
 }

 /**
  * Counts number of nodes in a VertexIterator
  *  @param vtxIter    contains nodes to be counted
  *  @return          number of nodes
 */
 public static int countVertices(VertexIterator vtxIter){
     int i=0;
     while(vtxIter.hasNext()){
         vtxIter.nextVertex();
         i++;
     }
     return i;
 }

 /**
 * Shows on screen messages forwarded by every node in a Graph
 * @param graph    graph to be analysed
 *
 */
 public static void showFw(Graph graph){
     VertexIterator iter = graph.vertices();
     while(iter.hasNext()){
         iter.nextVertex();
         int fw = ((VertexInfo)iter.element()).getFrwdMessages();
         System.out.println(((VertexInfo)iter.element()).getName()+": "+ fw);
     }
 }

 /**
 * Shows on screen source node of every node in the Graph
 * @param graph      graph to be analysed
 *
```

```
    */
    public static void showSource(Graph graph){
        VertexIterator iter = graph.vertices();
        while(iter.hasNext()){
            iter.nextVertex();
            Vertex source = ((VertexInfo)iter.element()).getSourceNode();
            System.out.println("source for: "+
                               ((VertexInfo)iter.element()).getName()+": "+
                               ((VertexInfo)source.element()).getName() );
        }
    }

    /**
    * Makes a copy of a node from Graph if a pattern node is found
    * @param paternNode     node to be compared to nodes in daughtherGraph
    * @param daughterGrph   Graph in which it looks for pattern node
    * @return               <code>codeNode</code> is a copy of node
    *                       contained in daughterGraphs which fits with name in patternNode
    */
    public static Vertex cloneNode(Vertex paternNode, Graph daughterGraph){
        VertexInfo geneticCode;
        VertexInfo sonCode;
        Vertex sonNode;
        boolean found;
        String sonName;
        VertexIterator vtxIter = daughterGraph.vertices();
        sonNode = null;
        found = false;
        while ((vtxIter.hasNext()) && (found != true)){
            sonNode = vtxIter.nextVertex();
            sonCode = (VertexInfo)sonNode.element();
            sonName = sonCode.getName();
            if (sonName.equalsIgnoreCase(((VertexInfo)paternNode.element()).getName()))
                found = true;
        }
        return sonNode;
    }
}
```

```
/*
 * ConnectivityTesterjava
 */

import jdsl.graph.algo.DFS;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;

/**
 * ConnectivityTester allows checking if a given graph is completely
 * connected. This class extends Depth-First Search algorithm implemented
 * on JDSL. The method <code>isConnected()</code> runs DFS algorithm starting at a
 * random node, afterwards, entire Graph is traversed to see if it exists
 * parent nodes besides start node, if such, returns <code>false</code>
 * indicating that Graph is not completely connected.
 *
 */
public class ConnectivityTester extends DFS{
    /**
     * Checks for connectivity of a Graph
     * @param g     graph to check connectivity
     * @return      <code>true</code> if graph is completely connected
     *              <code>false</code> any other case.
     *
     */
    public boolean isConnected(InspectableGraph g){
        Vertex star=g.aVertex();
        execute(g, star);
        for (VertexIterator vertices = g.vertices(); vertices.hasNext();){
            Vertex v= vertices.nextVertex();
        if (parent(v)==null && v!=star)
          return false;
      }
      return true;
    }
}
```