UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

FLOODING INTELIGENTE EN INTERNET

TESIS

QUE PARA OBTENER EL GRADO DE:
INGENIERO EN TELECOMUNICACIONES
P R E S E N T A :
ÓSCAR ESCALANTE MENDIETA

DIRECTOR: DR. JULIO SOLANO GONZÁLEZ
CO-DIRECTOR: DR. IVAN STOJMENOVIC.

MÉXICO, D.F. OCTUBRE 2002

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

A mi madre,

Por su entereza para salir adelante,
por su ejemplo de trabajo y dedicación
constantes, pero sobre todo,
por el amor que siempre me ha brindado.

À V. García et M. Moctezuma

Mes professeurs et conseillers
aux Departement des Télécommunications,
pour leur soutien inconditionnel
chaque fois que je l'avais besoin.

Resumen

Flooding, es el mecanismo clásico empleado para la propagación de información a través de Internet. Se utiliza cuando se debe enviar un mensaje a todos los nodos en la red, presentándose como el proceso natural para el *broadcasting* de actualizaciones de topología y de solicitudes de información, representando además, un método rápido y directo para difundir datos a través de un área entera de ruteo. Pero, a pesar de sus ventajas importantes, cuando el número de solicitudes se incrementa o cuando crece el área de ruteo, el *overhead* de comunicación por medio de *flooding* resulta prohibitivo, limitando la escalabilidad y disponibilidad de la red e incrementando los requerimientos de ancho de banda y de procesamiento.

En este trabajo, proponemos reducir el *overhead* de comunicación en el *broadcasting* por medio de *flooding* mediante el empleo del modelo uno-a-uno y empleando el concepto de Grafos de Vecindad Relativa (GVR) sobre topologías de Internet. El intercambio de mensajes entre nodos vecinos, el cual incluye entre otros, actualizaciones de localización, resulta suficiente para mantener estructuras como los GVR, y por lo tanto no aumentan el *overhead* del protocolo.

El *broadcasting* empleando los conceptos de GVR se ha propuesto anteriormente para redes inalámbricas *ad hoc*, por lo que una implementación del algoritmo para obtener GVR a partir de Grafos Unitarios Aleatorios se presenta para comparación con los resultados obtenidos en trabajos previos. Una vez que el algoritmo para obtener GVR se ha verificado, este mismo se emplea sobre topologías generadas sintéticamente con las características de Internet con el fin de cuantificar la reducción en la retransmisión de mensajes. Finalmente, se presenta una aplicación práctica del algoritmo de *broadcasting* empleando GVR, en el área de redes peer-to-peer al compararlo con el protocolo de *broadcasting* en Gnutella, una red completamente descentralizada empleada para compartir e intercambiar archivos, cuya topología ha sido muy bien estudiada recientemente.

Los resultados obtenidos en nuestros experimentos, reportan ahorros importantes tanto en topologías con las características de Internet como en la red Gnutella, ambos comparados con el empleo del mecanismo de *flooding* puro. Estos ahorros podrían ayudar a sobrellevar las limitaciones intrínsecas del *flooding* tales como la escalabilidad y el desperdicio de los recursos disponibles tanto en los nodos como en los enlaces de Internet.

Índice

AGRADECIMIENTOS 5

RESUMEN 6

INTRODUCCIÓN 8

- SOLUCIÓN PROPUESTA 8
- HERRAMIENTAS NECESARIAS 8

GRAFOS DE VECINDAD RELATIVA 9

- PROPIEDADES DE LOS GVR 9
- GENERACIÓN DE GUA 9
- *BROADCASTING* POR MEDIO DE GVR 10

GRAFOS DE INTERNET 11

- RETOS EN LA GENERACIÓN DE UNA TOPOLOGÍA DE INTERNET 11
- NIVELES EN LA TOPOLOGÍA DE INTERNET 11
- LEYES DE POTENCIA 12
- MODELOS DE RED UTILIZADOS 13
  - *Waxman [W]* *13*
  - *Palmer y Steffan [PaS]* *13*
  - *Barabàsi y Albert [BA]* *14*

GENERADOR DE TOPOLOGÍAS 16

- BRITE 16
  - *Funcionamiento de BRITE* *16*
  - *Formato de salida de BRITE* *17*

PROCESO DE SIMULACIÓN 19

- CLASES DEFINIDAS 19
- *BROADCASTING* SOBRE GUA EMPLEANDO GVR 20
- RESULTADOS DEL *BROADCASTING* SOBRE GRAFOS DE INTERNET 21

APLICACIÓN PRÁCTICA 25

- EL TRABAJO DE PORTMANN Y SENEVIRATNE EN LA RED GNUTELLA 25
  - *Algoritmo de PR* *25*
  - *Comparación entre los algoritmos PR y RNG* *27*

CONCLUSIONES 29

- TRABAJO FUTURO 30

REFERENCIAS 31

VERSIÓN EN INGLÉS 33

Introducción

En la tarea de *flooding*, un mensaje debe enviarse desde un nodo fuente a todos los otros nodos presentes en la red, para lo cual, cada nodo que reciba el mensaje, lo reenviará a todos sus vecinos, excepto a aquel del cual lo ha recibido. De acuerdo con Portamann y Seneviratne [PS], para *flooding* en Internet, la única solución conocida es el *broadcasting* directo.

El *flooding*, es el proceso empleado principalmente por protocolos de Internet de estado de enlace para difundir información nueva a través de una determinada área de ruteo [NS], después de presentarse cambios en el estado de algún enlace, con el fin de mantener actualizada la información de la topología o las solicitudes de ruteo. Sin embargo, este mecanismo se torna prohibitivo tan pronto como la frecuencia de los cambios presentados se incrementa o el área de cobertura crece. Puede decirse, por lo tanto, que las principales desventajas del *flooding* son debidas a sus características intrínsecas, que en términos generales originan:

- Incremento en el *overhead* del protocolo de enrutamiento.
- Reducción del ancho de banda disponible para otros procesos.
- Incremento en los costos de procesamiento.

Solución propuesta

Se base en algoritmos de *broadcasting* empleando el modelo uno-a-uno a través de Grafos de Vecindad Relativa (GVR) propuestos por [SSS] para broadcasting en redes *ad hoc* inalámbricas. Cuando un nodo *A* desea difundir un mensaje, el nodo *A* primero identifica cuáles nodos pertenecen a su GVR, y entonces envía el mensaje únicamente a eso nodos, y así subsecuentemente, con lo que la información se propaga a toda la red sin las desventajas del *flooding*.

Herramientas necesarias

- Implementación del algoritmo para obtener los GVR
- Generador de gráficas de Internet
- Definición de métricas
- Aplicación del algoritmo de GVR sobre gráficas de Internet.

Grafos de Vecindad Relativa

El GVR es un concepto teórico geométrico y gráfico propuesto por Toussaint [T] en 1981, en el cual se establece lo siguiente:

> *Un enlace (*u,v*) existe entre dos vértices* u *y* v *si la distancia entre ellos, 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* UV *existe si* $\forall w \neq u, v : d(u,v) \leq \max[d(u,w), d(v,w)]$

Propiedades de los GVR

Toussaint [T] 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.
- es un grafo plano.

El hecho de que se trate de una grafo plano, asegura que es un grafo esparcido. Lo cual resulta de suma utilidad, pues en un grafo plano con n vértices puede haber a lo más $3n-6$ aristas [SSS].

Generación de GUA

Las redes inalámbricas *ad hoc* se modelan adecuadamente por medio de grafos construidos de la siguiente forma: sea un nodo A que tiene un rango de transmisión $t(A)$. Dos nodos A y B en la red serán vecinos (por lo tanto, unidos por un enlace) si la distancia euclideana entre ellos es menor que el mínimo entre sus radios de transmisión (*i.e.*, $d(A,B) < min \{t(A), t(B)\}$). Si se tiene que todos los radios de transmisión son iguales, entonces el grafo correspondiente se conoce como el Grafo Unitario [SSS]. En este nivel de abstracción, un nodo en la red, está representado por un vértice en el grafo, mientras que un enlace está representado por una arista. Sin pérdida de formalidad, los términos nodo y vértice se usan indistintamente para referirse a la misma entidad, haciendo la consideración análoga para enlace y arista. El término aleatorio, en los grafos unitarios, se debe a que la localización de los nodos presentes en el grafo, se asigna de forma aleatoria. El algoritmo para la generación de un GUA es el siguiente:

- escoger n puntos de forma aleatoria de un plano de $[0,m] \times [0,m]$; donde n= 20, 50,100, 200, etc.
- seleccionar el grado promedio de los vértices $d = 2,3,4,5,\ldots$
- ordenar todos las $\frac{(n-1)n}{2}$ aristas en forma ascendente de acuerdo a su longitud.
- el radio para todos los nodos, será la longitud de la $R = \frac{nd}{2}$ *enésima* arista.
- el grafo se rechaza si está desconectado.

Broadcasting por medio de GVR

En este trabajo, nos concentramos únicamente en optimizar el número de mensajes cortos, de acuerdo al algoritmo de *flooding* definido en [HKB], en el que se establece que el número de mensajes reenviados es igual al número total de enlaces en la red. Por lo tanto, la propuesta de este trabajo, es reducir ese número mediante la aplicación de los conceptos de GVR.

Con el fin de estar seguros acerca de la correcta implementación del algoritmo para obtener un GVR a partir de un grafo de Internet, lo primero que se llevó a cabo fue el empleo de Grafos Unitarios Aleatorios (GUA) repitiendo los experimentos presentados en [SSS] para comparar los resultados.

Sean *U(S)* y *GVR(S)* el grafo unitario y el grafo de vecindad relativa definidos para un conjunto de nodos *S*. Por lo tanto si *U(S)* está conectado entonces $U(S) \cap GVR(S)$ también está conectado [SSS].

De lo anterior, el algoritmo de *broadcasting* basado en un subgrafo planar puede ser definido de la siguiente forma: Se construye un subgrafo planar (por ejemplo, un GVR) conteniendo todos los nodos se conjunto *S*. El número de mensajes cortos para el algoritmo de *broadcasting* es por lo tanto, igual al número total de enlaces en el subgrafo planar, el cual es a lo más de *3n-6* para un grafo de *n* nodos [SSS].

Grafos de Internet

En general, en los estudios y simulaciones de Internet, se asumen ciertas propiedades topológicas o se emplean topologías generadas de forma sintética. Si se pretende que tales estudios provean una guía correcta del comportamiento de Internet y por lo tanto de los protocolos y algoritmos estudiados, las topologías empleadas deben exhibir las propiedades fundamentales encontradas de forma empírica en la estructura actual de la red mundial, pues de otra forma, no se podría llegar a conclusiones adecuadas [MLBM]. Estudios recientes acerca de la topología de Internet han mostrado que algunos de sus parámetros poseen una distribución exponencial [FFF] por lo que distribuir aleatoriamente vértices y aristas en un grafo no representaría de forma adecuada su a la topología de Internet.

Retos en la generación de una topología de Internet

- Reflejar las características de una estructura enorme por medio de grafos relativamente pequeños.
- Tiempo de cómputo reducido para la generación de grafos.
- Cumplir con la distribución de leyes de potencia.
- Propiedades resultantes acordes con los resultados empíricos obtenidos

Niveles en la topología de Internet

Internet es la 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. Al nivel de ruteador, los nodos son los ruteadores y los enlaces son las conexiones físicas entre ellos. Al nivel de interdominio (o Sistema Autónomo) cada dominio, compuesto de centenares de ruteadores y computadoras, 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].

Debe mencionarse que el modelado de la topología de Internet se basa en la información disponible actualmente, 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]. Al nivel de sistemas autónomos, la información disponible es relativamente extensa por que ésta puede ser obtenida de las tablas empleadas en los BGP (Border Gateway Protocol) [NLAR, G]. En cambio es

mucho más difícil 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]. Por lo expuesto en [GT, CM y 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, NLAR], sin embargo, en la medida en que puedan conocerse las propiedades a este nivel, podrán desarrollarse protocolos que optimizen los recursos disponibles.

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.

Para el modelado de la topología de Internet, se emplearon tres modelos distintos, uno de los cuales se considera obsoleto actualmente, pero lo incluimos como referencia. Los otros dos cumplen con la distribución de ley de potencia del grado promedio de los nodos, la cual se considera como el principal parámetro en el modelado de la topología de Internet.

Leyes de potencia

Faloutsos *et al.* [FFF] han analizado Internet al nivel de sistema autónomo (o interdominio) y al de ruteadores, en ambos casos encontraron que la distribución de las conexiones sigue una ley de potencia. Adicionalmente han propuesto tres distribuciones adicionales, que junto con la de conexiones conforman lo que se ha denominado como las cuatro leyes de potencia de Internet, las cuales repetimos a continuación:

Ley de potencia 1: exponente de clasificación **R**. El grado de entrada, d_v, de un nodo v, es proporcional a la clasificación del nodo, r_v, elevada a una constante **R:** $d_v \propto r_v^R$. Esta ley de potencia se evalúa calculando el grado de salida para cada nodo, ordenando en forma descendente todos los valores calculados, asignando un índice a cada uno de estos valores, mismo que corresponde al rango r_v y trazando el grafo d_v *vs* r_v en una escala log-log. La información recolectada para topologías de interdominio y para las de ruteadores exhibe una distribución *cuasi* lineal, con una ligera desviación al final.

Ley de potencia 2: exponente de grado de salida **O**. La frecuencia, f_d, de un grado de salida, *d*, es proporcional al grado de salida elevado a una constante **O**: $f_d \propto d^O$

Ley de potencia 3: exponente de saltos **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**: $P(h) \propto h^H \quad h < \delta$, siendo δ el diámetro del grafo.

Ley de potencia 4: *eigen*-exponente **E**. Los *eigen*-valores, de un grafo son proporcionales al orden, *i*, elevado a una constante, **E**: $\lambda_i \propto i^E$.

[FFF] han sugerido el empleo de estas leyes de potencia para cuantificar el "realismo" de grafos de Internet generados de forma artificial. De cualquier forma, solo la primera ley es la que se evalúa actualmente en topologías generadas sintéticamente.

Modelos de red utilizados

Waxman [W]

El modelo de Waxman fue uno de los primeros que propuso distribuir los enlaces entre nodos de forma no aleatoria, al basar la conexión entre dos nodos en una función de probabilidad que toma en cuenta la distancia entre los mismos.

Los pasos para generar una topología de acuerdo al modelo de Waxman son:

- seleccionar *n* nodos de forma aleatoria, los cuales se encuentran distribuidos en un plano *xy*.
- un enlace entre un par de nodos *u*, *v* existe con una probabilidad que depende de la distancia entre ellos. Dicha probabilidad está dada por

$$P(\{u,v\}) = \beta^{\frac{-d(u,v)}{L\alpha}}$$

Donde α y β toman valores entre (0, 1) y L representa la máxima distancia internodal. Valores altos de β generan grafos con densidades altas, mientras que valores pequeños de α aumentan la existencia de enlaces cortos comparada con los grandes.

Palmer y Steffan [PaS]

El modelo de Palmer y Steffan está conformado por un algoritmo, al que llamaremos Recursivo, el cual consiste de dos etapas: uno, definición de una función de

probabilidad que selecciona de forma aleatoria un par de nodos y dos, la utilización de esta función de densidad de probabilidad para producir un grafo con características de distribución de conexiones que sigue una ley de potencia.

La función de densidad de probabilidad es una generalización de una distribución "80-20". En una distribución "80-20" en plano determinado, se divide en mitades, en una mitad de distribuyen 80% de los valores y en la otra el 20% restante. Para un arreglo bi-dimensional, se definen 4 cuadrantes, donde $(\alpha+\beta+\gamma+\varepsilon=1)$ corresponden al porcentaje de valores que se desea distribuir en cada subarreglo. Los pasos del algoritmo de generación son los siguientes:

- Asumir por simplicidad que el número de nodos N es potencia de 2, y que se desean M enlaces.
- Segmentar el grafo en dos conjuntos de nodos V_1 y V_2 cada uno de tamaño $N/2$. Hacer que la fracción p_{ij} de enlaces vaya de un nodo en Vi a algún nodo en Vj , $i; j = 1; 2$. $P = (p_{ij})$ son proporcionados como parámetros (α, β, γ, ε). La idea es favorecer enlaces internos al mantener p_{11} y p_{22} mayores que p_{12} o p_{21}, de aquí el nombre de "80-20".
- Empleando P, seleccionar uno de los cuatro posibles enlaces.
- Operar de forma recursiva entre la cuarta parte de los enlaces que cumplen con la selección arriba mencionada, y continuar con la recursividad con los mismos parámetros P hasta que un enlace específico se materialice (es decir, hasta que el número de nodos restantes sea igual a 1 ó 2
- Repetir hasta que M enlaces sean agregados al grafo.

Barabàsi y Albert [BA]

Barabàsi y Albert llegaron a la conclusión de la distribución de ley de potencia por medio de un tipo general de redes que han sido llamadas sin escalas, este tipo de redes describen sistemas complejos, y sus propiedades se aplican no solamente a la topología de Internet sino a una gran variedad de áreas que van desde la biología a las ciencias sociales o la química. El modelo de Barabàsi se basa en dos ideas principales: crecimiento en el tiempo y vinculación preferente.

Crecimiento y vinculación preferente:

- La mayoría de las redes reales crecen con el tiempo.

- La mayoría de las redes reales exhiben conectividad preferente. Un nuevo nodo incorporándose a la red se unirá con mayor probabilidad a un nodo muy conocido con alta interconexión.

Algoritmo para generar redes sin escalas. [BA2]

- Empezar con m_0 nodos sin enlaces.
- En cada iteración, realizar una de las siguientes operaciones:
  - Con probabilidad p agregar m ($m<m_0$) nuevos enlaces. Para ello, seleccionar de forma aleatoria un nodo como extremo inicial del nuevo enlace. El extremo final del nuevo enlace, en cambio es seleccionado con probabilidad

    $$\Pi(k_i) = \frac{k_i + 1}{\sum_j (k_j + 1)}$$

    incorporando el hecho de que los nuevos enlace apuntan de forma preferente a nodos populares con un gran número de conexiones. Este proceso se repite m veces.
  - Con probabilidad q volver a formar m enlaces. Para esto, seleccionar un nodo i y un enlace l_{ij} conectado a éste. A continuación, se elimina este enlace y se reemplaza con uno nuevo $l_{ij'}$ que conecta i con el nodo j' elegido con probabilidad $\Pi(k_{j'})$. Este proceso se repite m veces.
  - Con probabilidad $1-p-q$ agregar un nuevo nodo. El nuevo nodo tiene m nuevos enlaces que con probabilidad $\Pi(k_i)$ están conectados a nodos i ya presentes en la red.
- Después de t iteraciones, el modelo conduce a un grafo aleatorio con $N(t) = m_0 + (1-p-q)t$ vértices y $\sum_j k_j = (1-q)2mt - m$ aristas. Esta red evoluciona sobre un estado de escala invariante con la probabilidad de que un nodo tenga k enlaces, siguiendo una ley de potencia de tal forma que la distribución de las conexiones sigue una expresión de la forma:

$$P(k) \propto k^{-\gamma}$$

Generador de topologías

BRITE

BRITE es un generador universal de topologías diseñado por Medina *et al.* [MLMB] en la Universidad de Boston. En BRITE se implementan varios modelos de generación de topologías, entre ellos el de Waxman y el de Barabàsi, tanto a nivel de ruteador como a nivel de sistemas autónomos, además de lo anterior BRITE fue seleccionado para generar las topologías empleadas en nuestros experimentos debido principalmente a su:

- *Flexibilidad:* se pueden generar topologías de una gran variedad de tamaños.
- *Eficiencia:* ya que genera topologías de gran tamaño (*e.g.* número de nodos > 100, 000) con requerimientos de CPU y memoria reducidos.
- *Interfaz amigable*: se siguen los patrones estándar de generación de interfaces, con lo que solo se deben aprender una vez los mecanismos para generar topologías independientemente del modelo utilizado.

Funcionamiento de BRITE

Los detalles específicos en cuanto a cómo se genera un determinado tipo de topología dependen del modelo que se emplee, de cualquier forma, de manera esquemática, el proceso de generación puede dividirse en cuatro etapas principales:

- Colocación de los nodos en el plano
- Interconexión de nodos
- Asignación de atributos a los componentes de la topología (retardo y ancho de banda para los enlaces, identificadores para los nodos, etc.)
- Entrega de la topología con un formato específico

Como ya se mencionó, la anterior, no es una división estricta que se siga en todos los modelos, pero de forma conceptual refleja qué es lo que ocurre durante la generación de topologías. Adicionalmente, varios modelos pueden compartir procedimientos generales a lo largo del proceso de generación, mientras que otros difieren de forma significativa durante etapas específicas [MLMB].

Colocación de los nodos

- *Aleatoriamente:* cada nodo se coloca en una ubicación del plano seleccionada aleatoriamente.
- *Distribución de cola pesada:* el plano se divide en rectángulos (el tamaño del plano y de los rectángulos se controla por medio de parámetros de entrada). A cada uno de los rectángulos se le asigna un número de nodos, extraído de una distribución de cola pesada. Una vez que aquel valor es asignado, los nodos se colocan de forma aleatoria en el rectángulo.

Asignación de anchos de banda

Se ejecuta una vez que la topología se ha completado. BRITE asigna a los enlaces anchos de banda siguiendo una de las siguientes distribuciones seleccionadas:

- *Constante:* el valor especificado por *BW min* (el mismo para todos los enlaces en la topología).
- *Uniforme:* un valor extraído de una distribución uniforme en el rango *BW min* y *BW max*
- *Exponencial:* un valor distribuido exponencialmente con media *BW min*
- *Cola pesada:* un valor distribuido de cola pesada (Pareto con forma 1.2) con valores mínimo y máximo iguales a *BW max*

Formato de salida de BRITE

Un archivo de salida con formato BRITE está compuesto de tres secciones:

- *Información del modelo:* información acerca de la topología contenida en el archivo. Incluye número de nodos y enlaces e información específica del modelo empleado para generar la topología.
- *Nodos:* para cada nodo en el grafo, se escribe una línea en el archivo de salida la cual tiene el siguiente formato: `NodeId xpos ypos indegree outgdegree ASid type`
- *Enlaces:* para cada enlace el grafo, se escribe una línea en el archivo de salida la cual tiene el siguiente formato: `EdgeId from to length delay bandwidth ASfrom ASto type`

Las tablas que a continuación se muestran ofrecen una descripción de los campos presentes en un archivo de salida con el formato de BRITE.

NodeId	Identificador único para cada nodo
xpos	Coordenada en el eje x del plano
ypos	Coordenada en el eje y del plano
indegree	Grado de entrada del nodo
outdegree	Grado de salida del nodo
ASid	Identificador del sistema autónomo al que pertenece este nodo (si se trata de una topología jerárquica)
type	Tipo sistema al que pertenece el nodo (e.g. nivel de ruteador, sistema autónomo)

Significado de los campos en la sección de nodos

EdgeId	Identificador único para cada enlace
from	Identificador de nodo origen
to	Identificador de nodo destino
length	Distancia euclideana
delay	Retardo en la propagación
bandwidth	Ancho de banda
ASfrom	Si se trata de una topología jerárquica identificador del sistema autónomo del nodo origen
Asto	Si se trata de una topología jerárquica identificador del sistema autónomo del nodo destino
Type	Tipo asignado al enlace

Significado de los nodos en la sección enlaces

Proceso de simulación

Nuestro algoritmo de difusión se programó enteramente en JAVA, empleando el compilador JDK 1.3.4. Se eligió a Java por sus características de diseño orientado a objetos y por facilidad para desarrollar aplicaciones multiplataforma de manera rápida. Para la implementación de las estructuras de datos empleamos la Biblioteca Java de Estructuras de Datos, JDSL, por sus siglas en inglés. [JDSL].

Clases definidas

La tabla de abajo muestra las clases que se definieron, así como una breve descripción de ellas.

Clase	Función
`ConnectivityTester`	Verifica la conectividad en un grafo. Se emplea al generar los GUA.
`EdgeInfo`	Administra la información almacenada en los enlaces, como nombre, longitud, sistema de referencia.
`graphTools`	Implementa varias funciones útiles para manejo de grafos.
`InternetMap`	Carga una topología de Internet en memoria para usarla posteriormente.
`RNG`	Implementa el algoritmo de broadcasting basado en GVR.
`RUG`	Genera GUA.
`testIG`	Programa principal de prueba cuando se usan grafos de Internet.
`testRUG`	Programa principal de difusión cuando se emplean GUA.
`VertexInfo`	Administra las propiedades de los nodos, como nombre, sistema de referencia, ubicación.

Clases definidas en el programa de simulación.

Broadcasting sobre GUA empleando GVR

Esta simulación se realizó para repetir los experimentos presentados por [SSS] y de esta forma asegurar que al algoritmo para obtener los GUA estuviera trabajando adecuadamente. A continuación se presenta el pseudo código para la generación de GUA y para *broadcasting* cuando se construye el GVR.

Pseudo código para la generación de grafos unitarios aleatorios

```
RUG(m, n, d)
CHOOSE n points AT RANDOM FROM plane [0, m] x [0, m]
FOR_EACH pair of points IN Graph
    edge.lenght <-- distance(pair of points)
    Add edge.lenght in lenght list
SORT lenght list
Radius R= nd/ 2- th edge in sorted lenght list
IF graph disconnected
    RUG(m, n, d)
```

En un plano de [*0, m*] x [*0, m*] cada nodo tiene coordenadas (*x,y*) y la distancia entre nodos es la distancia euclideana calculada empleando las coordenadas de los nodos.

Pseudo código para obtener un GVR a partir de un grafo determinado

```
FOR_EACH node i
    FOR_EACH i.neighbor
        neighbors_list<-- i.neighbor
    FOR_EACH node j IN neighbors_list
        FOR_EACH j.neighbor
            IF j.neighbor IS IN neighbors_list
                k<--j.neighbor
                edgeij<--distance(i, j)
                edgeik<--distance(i, k)
                edgejk<--distance(j, k)
                dmax=dmax(edgeik, edgejk)
                if dmax<edgeij
                DELETE longer edgeij
```

La siguiente tabla muestra los resultados obtenidos para *broadcasting* en GVR, cuando se tienen al inicio un GUA. Nuestros resultados son muy similares a los

presentados en [SSS], por lo que podemos concluir que la implementación del algoritmo para obtener los GVR es correcta.

Enlaces	Grado promedio	Enlaces (GVR)	Grado promedio (GVR)	%Ahorro
204	4.08	123	2.46	40.0
304	6.08	124	2.48	60.0
404	8.08	128	2.56	69.0
501	10.02	122	2.44	76.0
751	15.02	125	2.5	84.0
1002	20.04	123	2.46	88.0
1252	25.04	121	2.42	91.0
1503	30.06	120	2.4	93.0
1753	35.06	121	2.42	94.0
2001	40.02	121	2.42	94.0

Resultados de la simulación para broadcasting sobre GUA.

Resultados del *broadcasting* sobre grafos de Internet

Los grafos que empleamos fueron creados mediante los modelos propuestos por Barabàsi. Palmer y Waxman. BRITE se utilizó como generador para Barabàsi y Waxman, mientras que el generador Recursivo de utilizo para el modelo de Palmer.

Empleando el modelo de Barabàsi

Como ya se mencionó Barabàsi ha propuesto un modelo general para la generación de topologías de redes complejas, que incluyen no solo a Internet. Este tipo de redes se denomina sin escalas. Para nuestros análisis establecimos los siguientes valores para los parámetros de entrada:

```
N=1000
p=0.25
q=0.08
m=1, 2 .. 10.
```

Estos valores fueron empleados con el fin de obtener topologías con propiedades similares a aquellas encontradas por Govindan *et al.* [GT] para Internet. Los valores que ellos encontraron fueron resultado de la exploración de aproximadamente 150 mil interfaces y cerca de 200 mil enlaces, lo que les permitió concluir que la distribución de conexiones en Internet sigue una ley de potencia. Cabe señalar que el trabajo de exploración que ellos realizaron es casi 50 veces más grande, en lo referente a nodos y enlaces, que el realizado por [FFF], con lo que se reafirman

las conclusiones de estos últimos investigadores en cuanto a la ley de potencia que sigue la distribución de las conexiones.

De acuerdo con [AB] los resultados experimentales obtenidos por [GT], pueden ser modelados por una ley de potencia de la forma $P(k) \propto k^{-\gamma}$ donde $\gamma = 2.4$ y en la que el grado promedio de los nodos es $<k>= 2.66$. Para nuestras simulaciones comenzamos empleando grafos con valores de $<k>=4.56$, mismos que fueron incrementando con el fin de observar los ahorros en la retransmisión de mensajes conforme la densidad de los grafos aumentaba.

Los resultados para el proceso de simulación se muestran en la tabla de abajo, la métrica empleada para obtener el GVR fue la longitud de los enlaces, misma que se calculó empleando las coordenadas (*x*,*y*) de los nodos conectados en los extremos de cada enlace y a partir de estas la distancia euclideana entre los nodos:

Enlaces	Grado promedio	Número de enlaces (GVR)	Grado promedio (GVR)	%Ahorros
2745	5.49	2281	4.56	17
5598	11.2	3772	7.54	33
8319	16.64	4572	9.14	46
10820	21.64	5339	10.68	51
13385	26.77	5840	11.68	57
20565	41.13	6766	13.53	68
27310	54.62	6875	13.75	75

Resultados empleando el modelo de Barabàsi

Resultados con el modelo de Waxman

El modelo de Waxman fue uno de los primeros que propuso establecer conexión entre los nodos de forma no aleatoria, al definir una función de probabilidad que toma en cuenta la distancia entre dos nodos. Debido a que este modelo no trataba de modelar la topología de Internet, no cumple con la ley de potencia para la distribución de las conexiones [MP, PaS], de cualquier forma, representa una referencia adecuada cuando se evalúan otros modelos. Los valores de los parámetros para este modelo fueron los siguientes:

```
alfa=0.2
beta=0.2
n=500
m=4, 6, 8, 10, 20, 30, 40
```

Según lo expuesto por [MP] valores de $\alpha = 0.2$ y $\beta = 0.2$ conducen a una distribución aproximada a la topología de Internet, para nuestros experimentos elegimos n=500, donde n representa el número de nodos presentes en la red, mientras que m, que representa el número de enlaces con lo que se puede controlar el grado promedio de los nodos.

Enlaces	Grado promedio	Enlaces en GVR	Grado promedio (GVR)	%Ahorros
500	2	500	2	0.0
1000	4	984	3.94	2.0
1500	6	1430	5.72	5.0
2000	8	1832	7.33	9.0
2500	10	2185	8.74	13.0
5000	20	3338	13.35	34.0
7500	30	3662	14.65	52.0
10000	40	3704	14.82	63.0

Resultados usando el modelo de Waxman

Modelo de Palmer y Stefan

En el trabajo de [PaS] se presentan dos modelos diferentes, nosotros usamos el modelo Recursivo ya que mediante la utilización de generador aleatorio "80-20" se puede obtener una topología que cumple muy bien con una distribución de ley de potencia [PaS, CJPP].

Los valores para nuestros experimentos fueron:

```
alfa=beta=gamma=epsilon=0.25
n=512
links=1000, 1500, 2000, 2500, 5000, 7500, 10000
```

alfa, beta, gama y *epsilon* tienen el mismo valor para poder generar una distribución uniforme en los cuatro subconjuntos en los que se divide el plano.

Como el algoritmo Recursivo no proporciona la posición de los nodos, ésta se asigna de forma aleatoria una vez que la topología se ha generado, con lo cual ya es posible calcular la distancia euclideana entre cada par de nodos conectados. Cuando se usa como métrica el retardo, lo único que cambia es que este valor se selecciona aleatoriamente de un intervalo $(0,1)$ y éste es a la longitud del enlace. De cualquier manera, los valores obtenidos son idénticos por lo que no se muestran en la tabla.

Enlaces	Grado promedio	Enlaces (GVR)	Grado promedio (RNG)	%Ahorros
1000	3.91	968	3.78	4.0
1500	5.86	1421	5.55	6.0
2000	7.81	1821	7.11	9.0
2500	9.77	2109	8.24	16.0
5000	19.53	2988	11.67	41.0
7500	29.3	3091	12.07	59.0
10000	39.06	2981	11.64	71.0

Resultados al utilizar el algoritmo Recursivo empleando como métrica la longitud del enlace.

Aplicación práctica

Flooding es el proceso empleado tradicionalmente en los protocolos de estado de enlace para propagar información a través de un área de ruteo, tan pronto como se presenta un cambio del estado de un enlace. Las ventajas intrínsecas del *flooding* no constituyen una limitante cuando los cambios en la topología son infrecuentes o cuando el área de ruteo es relativamente pequeña, en ambos casos comparándolos con el ancho de banda o con los recursos computacionales disponibles. Sin embargo, en los últimos meses ha habido un interés creciente en las llamadas redes peer-to-peer (P2P), desencadenado principalmente por la popularidad de aplicaciones para compartir archivos como Napster, Gnutella o Morpheus [PS]. Algunas de las características principales de un sistema P2P son que todos sus participantes poseen las mismas funciones, y la inexistencia de una entidad que haga el papel de administrador o controlador central.

El trabajo de Portmann y Seneviratne en la red Gnutella

Portmann y Seneviratne [PS] presentaron una alternativa para superar las limitaciones del *flooding* en términos de costo y escalabilidad para redes descentralizadas P2P. La propuesta consistió en el empleo de un protocolo de enrutamiento llamado propagación de rumores (PR) (también conocido como *Gossiping* o "chismeo") empleando como marco de trabajo el protocolo especificado para Gnutella [GPS].

Algoritmo de PR

Los protocolos de propagación de rumores o chismeo pertenecen a la clase de protocolos probabilísticos para envío de mensajes, debido a que los nodos 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 broadcasting
al enviar un mensaje m a B de sus vecinos
elegidos de forma aleatoria.

When (un nodo p recibe un mensaje m de un nodo q)
    If(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 ha visto al mensaje m.
```

El parámetro B especifica el número máximo de vecinos a los cuales un mensaje es 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 control las propiedades del protocolo. El costo del *broadcasting* en este algoritmo corresponde al número total de mensajes retransmitidos para difundir un mensaje en la red completa.

Un nodo p sabe si su vecino q ya ha visto el mensaje que le quiere enviar si y solo si p se lo envió anteriormente a q o si p recibió aquel mensaje de q. Cuando el número de vecinos válidos es menor que B, el mensaje solo se envía a este reducido número de vecinos.

El costo del *broadcasting* al emplear *flooding* tal y como es especificado en Gnutella puede ser calculado sabiendo que durante el *broadcasting* cada nodo recibe el mensaje por lo menos una vez. La primera vez que se recibe el mensaje, el nodo receptor lo reenvía a todos sus nodos, excepto aquel del cual le ha sido enviado. Los arribos subsecuentes del mismo mensaje son ignorados, por lo anterior, el número de mensajes que cada nodo debe enviar en cada *broadcast* se limita al número de vecinos que posee menos uno. Unicamente el nodo que inició el *broadcasting* envía su mensaje a todos sus vecinos, lo cual incrementa el costo total en uno. Lo anterior conduce a la expresión general para calcular el costo del *broadcasting* de un mensaje sobre la red entera.

$$c = 1 + N(d-1)$$

c: costo, dado por el número de mensajes reenviados

d: grado promedio de los nodos

N: número de nodos

La siguiente tabla, muestra los resultados obtenidos por [PS] para el desempeño de algoritmo de PR en comparación con el de *flooding*, al usar combinaciones distintas de valores en los parámetros B y F. Los resultados se obtuvieron a partir de topologías de Barabàsi con 1000 nodos y grado promedio de los nodos d de 4, 6 y 8. También se muestran los tres parámetros; costo, alcance y tiempo para el algoritmo de propagación de rumores, como figuras comparativas con los valores obtenidos por medio del *flooding*, que representa el 100%.

B	F	d=4		d=6		d=8	
		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%

Desempeño de algoritmo PR en comparación con flooding

Comparación entre los algoritmos PR y GVR

En el algoritmo de *flooding* tal y como está implementado en Gnutella [GPS], un enlace puede reenviar un mismo mensaje en ambas direcciones; del nodo *A* al nodo *B* y del nodo *B* al nodo *A* por ejemplo, dependiendo del instante en que un mensaje sea recibido, lo anterior debido a que un nodo debe reenviar un mensaje recibido a todos sus vecinos, excepto a aquel del cual provino, sin tomar en cuenta si alguno de los vecinos ha recibido previamente ese mismo mensaje por parte de algún otro nodo. En [PS] se presenta una alternativa al *flooding* llamada propagación de rumores, en la cual un nodo selecciona aleatoriamente *B* vecinos para retransmitirles el mensaje y en el que se define el número *F* de veces que un nodo puede retransmitir un mismo mensaje, esto origina que ocasionalmente un enlace puede ser utilizado en ambas direcciones, de acuerdo a los valores empleados para *B* y *F*. Llamaremos a estos dos modelos DS (Doble Sentido, debido a que un mensaje puede viajar en ambos sentidos).

Para nuestros experimentos, utilizamos el modelo expuesto por Heinzelman *et al.* en [HKB]. En el cual establece que el número de mensajes reenviados en una tarea de *broadcasting* es igual al número de aristas en el grafo. En este modelo, cada enlace puede ser utilizado solo en un sentido para enviar un mismo mensaje. Es decir, mientras *A* envía a *B*, *B* no envía a *A*, en vez de ello, recibe algunos mensajes, los procesa, y si el mismo mensaje es encontrado, cancela el intento de enviarlo a *A*. Llamaremos a este modelo SS (por que el mensaje puede viajar un Solo Sentido a través de un enlace).

Dado que el *broadcasting sobre* GVR garantiza el alcance del 100% de nodos, hemos comparado el número de mensajes reenviados con los resultados del algoritmo de PR cercanos al 100% para realizar una comparación más adecuada. Abajo se muestran los resultados al comparar GVR con PR, empleando topologías de Barabàsi con 500 nodos y promedio de grados de los nodos *d=4, 6* y *8*.

Grado promedio	Mensajes reenviados DS (100%)	Costo de PR	Costo de GVR	Costo RNG DS
4	2998	69.2%	63.82%	96.6%
6	4992	66.88%	53.96%	89.91%
8	6989	61.1%	50.88%	87.66%

Algoritmo de PR contra GVR

De lo mostrado anteriormente se puede decir que el algoritmo de *broadcasting* sobre GVR, tal y como se propuso originalmente proporciona mejor desempeño que el de PR, además de que los ahorros se incrementan conforme la densidad de la red aumenta. Incluimos en la última columna el número de mensajes reenviados cuando un enlace se emplea para enviar un mismo mensaje en ambas direcciones, mostrando el impacto negativo de esta acción.

Conclusiones

Este trabajo muestra que la búsqueda del modelo para evaluar el desempeño de nuestro algoritmo de *broadcasting* en Internet no necesariamente conduce a una respuesta única, debido principalmente a la existencia de varios modelos de generación y a la variedad de enfoques para reflejar de manera fehaciente las características de la topología de Internet. Si bien, los resultados para nuestro algoritmo no varían de forma drástica de un modelo a otro, consideramos que es importante emplear varios generadores, pues aún no existe un modelo unificado que refleje todas las características de Internet. No obstante, en todos los casos, los ahorros en la retransmisión de mensajes comienzan a ser notorios para grados promedio de los nodos mayores a 10 en donde éstos representan un 25% en promedio, respecto a la utilización de *flooding*. En general, mientras más redundante se vuelva una topología, mayores ahorros se obtienen al utilizar los GVR.

También se ha mostrado que en Internet el *broadcasting* empleando GVR puede ser considerado como una alternativa al *flooding* y al algoritmo de PR. Los resultados de las simulaciones indican que los GVR representan una opción más escalable pues reducen el costo del *broadcasting* por medio de *flooding* y del algoritmo de PR. Adicionalmente los GVR garantizan alcance del 100% de nodos de la red, mientras que, debido a su naturaleza probabilística, el protocolo de PR no alcanza este índice. Es decir, utilizar GVR reduce el costo del *broadcasting* sin pérdida en el alcance de nodos de la red.

Finalmente, de forma indirecta se ha mostrado la importancia en la elección del modelo de *broadcasting* y el impacto que esto último tiene en los resultados obtenidos. Cuando en la primera parte de este trabajo comparamos las GVR con *flooding* al emplear topologías de Barabàsi, los ahorros por encima del 40% se obtuvieron solo con redes cuyo grado promedio de los nodos no era menor a 16, mientras que al comparar las GVR con los resultados de [PS] empleando las mismas topologías que ellos usaron, se empezaron a obtener ahorros mayores al 35% para grados promedio de los nodos de apenas 4. El modelo de topología fue el mismo, además se emplearon los mismos valores utilizados en los experimentos de [PS]; la diferencia radicó en que el modelo de *broadcasting* era diferente.

Trabajo futuro

Queda para desarrollos posteriores la implementación del algoritmo de PR para poder evaluar si las GVR proporcionan todavía mejores resultados cuando se cambian algunas propiedades de las topologías empleadas. También dejamos para un trabajo posterior el análisis de la distribución de carga de mensajes entre los nodos, lo que nos permitiria saber si los GVR permiten la distribución de la tarea de reenvío entre todos los nodos participantes en la red.

Referencias

[AB] R. Albert, A. Barabàsi; Statistical Mechanics of Complex Networks. June 2001.

[BA] A. Barabàsi, R. Albert; Emerge of scaling in random networks. Science , 286. 1999.

[BA2] A. Barabàsi, R. Albert; Topology of evolving networks: local events and universality. USA, 2000.

[CJPP] A. Chakrabarti, M. Joshi, K. Punera, D. Pnnock; The structure of broad topics on the web. Honolulu, Hawaii, USA. 2002.

[CM] K.C. Claffy and D. McRobb. Measurement and Visualization of Internet Connectivity and Performance. http://www.caida.org/Tools/Skitter

[FFF] M. Faloutsos, P. Faloutsos, Christos Faloutsos, On Power-Law Relationships of the Internet Topology SIGCOMM (1999)

[G] L. Gao. On Inferring Autonomous System Relationships in the Internet. IEEE Global Internet, November 2000.

[Go] M. T. Goodrich; Data Structures and Algorithms in Java. EUA. 1998. Willey And Sons Inc.

[GPS] Gnutella Protocol Specification. http://www.clip2.com/GnutellaProtocol04.pdf

[GT] R. Govindan, H. Tangmunarunkit; Heuristics for Internet Map Discovery. Proc. IEEE Infocom 2000.

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

[JDSL] Data structures library in Java. http://www.jdsl.org/

[MLMB] A. Medina, A. Lakhina, I. Matta, J. Byers; BRITE: Universal Topology Generation from a User's perspective, April 2001, Boston University. EUA. http://www.cs.bu.edu/brite/

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

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

[NS] P. Narvaez, K. S. Siu; Fault-Tolerant Routing in the Internet without Flooding. MIT. 1999

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

[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.

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

[W] B. Waxman, Routing of Multipoint Connections. IEEE J. Select. Areas Commun., December 1988.

Anexo

Versión en Inglés

Index

ABSTRACT 36

THESIS INTRODUCTION 37

1.1 GOALS OF THIS THESIS 37
1.2 RESEARCH CONTRIBUTIONS 37
1.3 SIMULATIONS AND RESULTS 38
1.4 DOCUMENTATION OUTLINE 38

BACKGROUND 40

2.1 AD HOC WIRELESS NETWORKS 40
2.2 BROADCASTING TASK 41
2.3 INTERNET GRAPHS 42
2.4 PEER TO PEER NETWORKS 43

AD HOC WIRELESS NETWORKS 44

3.1 GRAPH MODEL USED FOR *AD HOC* NETWORKS 44
3.1.1 Unit Graphs *45*
3.2 GRAPH THEORETIC CONCEPTS 45
3.2.1 Dominating sets and internal nodes *45*
3.2.2 Relative Neighborhood Graph *47*
3.3 BROADCASTING ALGORITHMS 48
3.3.1 Classic Flooding *49*
3.3.2 Gossiping *50*
3.3.3 New Broadcasting algorithm *50*
3.4 BROADCASTING VIA RNG 50

INTERNET GRAPHS 52

4.1 NETWORK TOPOLOGIES 52
4.2 EMPIRICAL RESULTS FOR INTERNET TOPOLOGY 55
4.3 POWER LAWS 57
4.4 TOPOLOGY NETWORK MODELS 57
4.4.1 Waxman [W] *57*
4.4.2 Palmer and Steffan [PaS] *58*
4.4.3 Barabàsi and Albert [BA] *58*
4.5 TOPOLOGY GENERATOR 60
4.5.1 BRITE *60*

PEER-TO-PEER NETWORKS 63
5.1 DEFINITION OF PEER-TO-PEER 63
5.2 THE GNUTELLA NETWORK 66
5.3 PORTMAN AND SENEVIRATNE WORK ON GNUTELLA NETWORK 68
5.3.1 The Rumor Mongering algorithm 68
THE BROADCASTING SIMULATOR 70
6.1 IMPLEMENTATION OF ALGORITHMS 70
6.1.1 Defined classes 70
6.2 SIMULATION PROCESS 71
6.2.1 Implementation of RNG algorithm 72
6.3 AN OBSERVATION IN THE USE OF THE RNG TERM 73
SIMULATION RESULTS AND ANALYSIS 74
7.1 THE BROADCASTING OVER RUG VIA RNG 74
7.2 BROADCASTING OVER INTERNET GRAPHS 75
7.2.1 Using Barabàsi model 75
7.2.2 Using the Waxman model 76
7.2.3 Palmer and Stefan model 77
7.3 SOME VISUAL EXAMPLES OF INTERNET GRAPHS 78
7.4 GOSSIP VS. RNG ALGORITHM 80
CONCLUSIONS 82
8.1 CONCLUSIONS 82
8.2 FUTURE WORK 82
REFERENCES 83
APPENDIX A **BASICS ON GRAPH THEORY** 89
APPENDIX B **ADT FOR SIMULATOR CLASSES** 97
APPENDIX C **SOURCE CODE FOR GRAPHS PACKAGE** 109

Abstract

Flooding process is the classic mechanism for propagating information over the Internet. Used when a message must be delivered to all nodes in a network, arrives as a natural way to broadcast topology updates and request for information, representing a direct and fast method for disseminating data across an entire routing area. Despite its important advantages, when the number of requests increases or the routing area grows, communication overhead by means of flooding becomes prohibited, limiting the scalability and the availability of the network and increasing the bandwidth needed and the processing time consumption.

In this work we propose to reduce the communication overhead of broadcasting by means of flooding by using one-to-one model and applying the concept of Relative Neighborhood Graphs (RNG) over Internet-like topologies. Regular message exchanges between neighbors, which include location updates or signal strengths, suffice to maintain these structures, and therefore they do not impose additional communication overhead. Broadcasting using RNG concepts have been previously proposed for wireless *ad hoc* networks, thus an implementation of a RNG algorithm over Random Unit Graphs is presented for comparison with previous results. Once that the RNG algorithm has been tested, it is applied over Internet-like topologies to quantify the reduction in messages retransmission. Finally, a practical application of broadcasting applying the RNG algorithm is presented for the field of peer-to-peer networks when comparing to broadcasting protocol of Gnutella, a fully decentralized file sharing application, whose topology has been well studied recently.

Our results report important savings for broadcasting on both Internet-like topologies and the Gnutella network, compared with the use of pure flooding mechanism. This savings could lead to overcome intrinsic limitations of flooding such as the limitation of scalability or the waste of available resources on Internet nodes and links.

Chapter 1

Thesis Introduction

Although using a flooding mechanism for propagating information over Internet has always showed some disadvantages, these were no so evident before some applications became popular among Internet users. Flooding is widely used in link-state protocols like OSPF for broadcasting topology updates, but negative impact is mainly seen and undergone by Internet administrators and not by end users.

But nowadays the emergence of novel network applications intended to facilitate worldwide sharing of information jointly with the model used for its implementation has become in a revival of the original Internet spirit: a worldwide community sharing and interchanging information.

The success of new applications will depend on the ability to provide efficient and fast communication between an increasingly large number of autonomous hosts dispersed all over the Internet. Now we are implementing known solutions for broadcasting in *ad hoc* networks on what has been called peer-to-peer networks. In wireless networks research in broadcasting algorithms has been done extensively, we propose here to apply one of these concepts for broadcasting over an Internet topology.

1.1 Goals of this thesis

- To design an algorithm for Internet flooding applying the RNG concept.
- To evaluate the performance of the proposed algorithm and to compare the results with known solutions for Internet flooding.

1.2 Research Contributions

This thesis applies concepts of RNG for Internet broadcasting. Results from this work can be particularly useful in the field of peer-to-peer networks, which could be considered as the Internet analogy of wireless *ad hoc* networks due to its decentralized nature and dynamic behavior.

Peer-to-peer networking has been gaining attention mainly for the increasing popularity of file sharing applications like Gnutella, KaZaA or Morpheus but other systems with completely different applications such as SETI@home [SETI] can also be labeled as peer-to-peer.

The importance of designing alternatives to flooding are clear if we consider that bandwidth requirements for pure flooding may quickly become prohibited for relative small networks, limiting their scalability and availability.

1.3 Simulations and Results

A number of simulations are presented for broadcasting over RUG, which are those used to model *ad hoc* networks, and applying RNG over Internet-like topologies. It is shown how retransmissions savings increase as network density gets larger. Results show how RNG represents a more scalable option in terms of messages retransmission compared with the solution presented by [PS].

1.4 Documentation outline

Chapter 2 provides a brief description of the key concepts in this work: wireless *ad hoc* networks, broadcasting task over *ad hoc* networks, Internet graphs and peer-to-peer networks.

Chapter 3 presents an exhaustive description of the work presented by Stojmenovic *et al* [SSS], upon which the proposed routing algorithm for this work is based.

Chapter 4 discusses an actual scenario for the generation of Internet topologies, existing models and generators and known properties for graphs describing these topologies.

Chapter 5 gives a general description of peer-to-peer networks, focusing on Gnutella and its protocol for searching files and peers.

Chapter 6 describes the broadcasting simulator, the implemented classes and the RNG algorithm.

Chapter 7 presents and discuses the broadcasting simulation results when using RUG and Internet graphs.

Chapter 8 gives the general conclusions of this thesis and discusses some future work.

Appendix A presents a basic introduction to graph theory.

Appendix B provides ADT for used classes in our simulation processes.

Appendix C contains the full Java code for the implemented classes.

Chapter 2

Background

In this chapter we present a general explanation of the main topics of this thesis: *ad hoc* wireless networks, broadcasting task on *ad hoc* networks, Internet graphs, properties and generators and peer-to-peer networks. We attempt to offer a general description that will guide to more detailed explanations in subsequent chapters.

2.1 Ad hoc wireless networks

An *ad hoc* wireless network is a wireless network without centralized control where every node acts as a router, forwarding packets for other nodes as necessary. This makes it possible for this kind of networks to emerge whenever there is a need for it. As soon as two nodes are within range, a network connection can be established. This type of network has many advantages over traditional wired networks, for example that it is possible to use very small and versatile devices as nodes. Examples are PDAs (Personal Digital Assistants), laptops and cellular phones. But the greatest advantage is that there is no need for an existing infrastructure in order for a network to form. Setting up a mobile network is very fast, efficient and can be done practically anywhere.

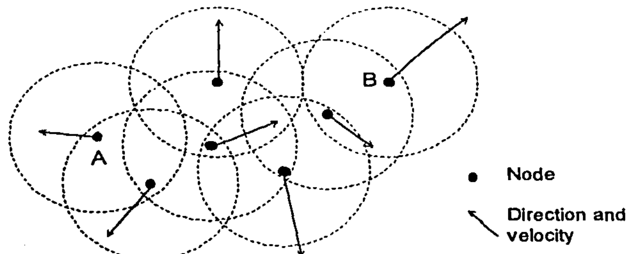


Figure 1.1: An ad hoc wireless network where nodes are moving in arbitrary directions and with arbitrary speed. Node A is communicating with node B via four other nodes. Dotted circles represent transmission radii.

A wireless *ad hoc* network is preferable in many situations when people wish to share information quickly, such as search-and-rescue operations, meetings or conferences, various military operations and police matters.

2.2 Broadcasting task

In one-to-all model, transmission by each node can reach all nodes that are within a radius distance from it, while in the one-to-one model, each transmission is directed toward only one neighbor. A one-to-one communication model emerges when narrow beam directional antennas are used, or when each node receives messages on its own frequency, and each node is aware of the frequency of each of its neighbors.

In a broadcasting task, a source node sends the same message to all the nodes in the network. Broadcasting in one-to-one model is studied in [BMSU, HKB, SK], and it is the one we will use in this work. Broadcasting is frequently refereed to as *flooding*. We will use this term to refer to broadcasting scheme in which the same message is retransmitted by *all* nodes that receive it, and on *all* outgoing edges (except the edges on which the same message was already received). Broadcasting applications include paging a particular host or sending an alarm signal. Flooding is also used for route discovery in any source-initiated routing algorithm. For example: the source S may initiate a destination search process by broadcasting a short message that contains the location of S and some control bits. When the destination search message reaches successfully the destination D, D applies any location based routing algorithm (in [BMSU], their algorithm guarantees delivery if location of destination, in this case S, is accurate) and reports back to S with a short message containing its location. The source S can then apply again the same routing algorithm [BMSU] (or use a path created in the previous step by D if that path was recorded in the process) to send the full message towards D. Note that a one-to-one communication model suffices for single-path routing algorithms, such as [BMJHJ, BMSU].

Ad hoc networks are best modeled by unit graphs which are constructed in the following way: Let A be a node having its transmission range $t(A)$. Two nodes A and B in the network are neighbors (thus joined by an edge) if the Euclidean distance between them is less than the minimum between their transmission radii (that is $d(A,B) < min\ \{t(A), t(B)\}$). If all transmission ranges are equal, the corresponding graph is known as the unit graph. We have used random unit graphs in our experiments.

Seddigh, Solano and Stojmenovic [SSS] proposed to reduce the number of message retransmissions in the broadcasting task by, among other concepts, limiting

the retransmissions to edges that belong to RNG. They assume that RNG maintenance is incorporated into location updates between neighboring nodes by means of a GPS (Global Position System) or another location method available to all the nodes in the network. The GPS provides the location information (latitude, longitude, height and time among others) to hosts in a wireless network using a satellite network. They also proposed that alternatively, nodes may measure signal strengths of incoming messages and determine the location of its neighbors by exchanging signal strength information with their neighbors; the signal strength determines the node distance, and distances from two neighboring nodes uniquely determine the position of a node [CHH].

The concept of localized algorithms was proposed in [EGHK] as distributed algorithms where simple local node behavior achieves a desired global objective. The RNG broadcasting algorithm is localized, since each node needs only the information from its neighbors, and possibly two hop neighbors.

2.3 Internet Graphs

At first sight it could be thought that the Internet host distribution follows a random behavior. In fact, this was the early approach that scientists follow to describe complex networks like Internet, but a more detailed analysis on hosts distribution lead to consider that there must be some kind of non random organization. Network studies often simulate some artificial network topologies to evaluate new ideas and schemes. There are several topology generation algorithms that are commonly used, but these are parameterized by average-based metrics and do not necessarily produce topologies that follow the observed power laws proposed by Faloutsos *et al.* [FFF]. It is therefore possible that the artificial networks used in simulation-based studies are not "realistic", and that the corresponding conclusions are inaccurate.

In recent years, many scientists have dedicated to discover laws that rule the behavior of complex networks, and to develop tools and measures to capture in quantitative terms the underlying organizing principles. As stated, recent studies have shown that for Internet certain topological properties follow a power-law [FFF] that is, certain graph metrics follow the distribution $y \propto x^{\alpha}$. This phenomenon has been observed in router topology, inter-domain topology [FFF], and the worldwide-web [AJB, BA, HA, KKRRT, KRRT]. Previous metrics used to characterize Internet graphs have focussed on averages: for example, average out-degree of routers. While these metrics are important, they do not capture higher-order properties such as the power-laws.

For the present work, we consider three different models for the Internet topology, two of them are implemented by BRITE [MLMB], a topology generator designed at Boston University, and the Recursive model that was implemented by Palmer and Steffan [PaS].

2.4 Peer to Peer Networks

The popularity of peer-to-peer file sharing applications such as Gnutella and Napster has created a flurry of recent research activity into the peer-to-peer architecture. Although the exact definition of "peer-to-peer" is debatable, these systems typically lack dedicated, centralized infrastructure, but rather depend on the voluntary participation of peers to contribute resources out of which the infrastructure is constructed. Membership in a peer-to-peer system is *ad hoc* and dynamic: as such, the challenge of such systems is to figure out a mechanism and architecture for organizing the peers in such a way that they can cooperate to provide a useful service to the community of users. For example, in a file sharing application, one challenge is organizing peers into a cooperative global index so that any peer can quickly and efficiently locate all content in the system. Additionally, the system must take into account the suitability of a given peer for a specific task before explicitly or implicitly delegating that task to the peer.

As shown by [SGG] there is a significant amount of heterogeneity in the Gnutella and Napster networks; bandwidth, latency, availability, and the degree of sharing vary between three and five orders of magnitude across the peers in the system. Thus, correct delegating of tasks to peers according to its own resources must be done in order to the entire system to work properly.

In this work we focus on a broadcasting algorithm to reduce the communication overhead between nodes in the network. This communication overhead limits scalability and in some extreme cases may cause network fragmentation as it has occurred in Gnutella network, due to its protocol specifications for finding peers and querying files by means of pure flooding.

Chapter 3

Ad Hoc Wireless Networks

A network is a set of hosts with network links between some subset of them. We can model such a structure as an undirected graph $G=(V, E)$ where $E \subseteq V \times V$ such that $(u,v) \in E \Rightarrow (v,u) \in E$. We further require that $(u,u) \notin E$ for all u (*i.e.*, there are no hosts that have an external link to themselves).

3.1 Graph model used for *ad hoc* networks

For a wireless network to be represented by a graph in the sense described above, a vertex in the graph represents a node while an edge represents a direct connection. Direct connection between nodes A and B is established in direction AB if node B is in the transmission range $t(A)$ of node A. Bi-directional connection may not always be established, since transmission radii may be different. Nevertheless, for our experiments, the transmission radius is equal for all nodes in the network since we have used unit graphs.

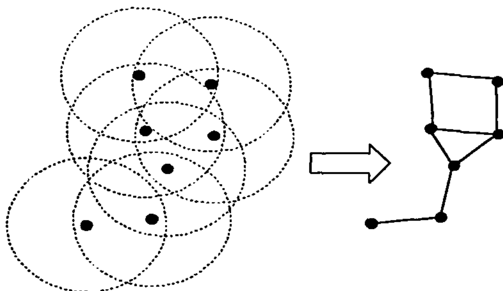


Figure 3.1: Graphic model of an ad hoc network. Nodes in the network are vertices in the graph and the links connecting the nodes are the edges in the graph.

3.1.1 Unit Graphs

We will use the unit graph concept as defined by Bose *et al.* [BMSU].

Let S be a set of points in the plane. Then the *unit graph* $U(S)$ is a geometric graph that contains a vertex for each element of S. An edge (u,v) is present in $U(S)$ if and only if $dist(x,y) \leq 1$, where $dist(x,y)$ denotes the normalized Euclidean distance between vertices x and y. In the remainder of this work we will refer to the elements of S alternately as host, nodes, or vertices. Unit graphs are a reasonable mathematical abstraction of wireless networks in which all nodes have equal broadcast ranges. In this work we describe some algorithms for broadcasting on unit graphs that do not require global information about $U(S)$. Each vertex $v \in U(S)$ represents a transmission station, and has no information about $U(S)$ except the set of nodes $N(v)$ to which it is adjacent. A packet that is stored at vertex v can be transmitted to any vertex in $N(v)$.

3.2 Graph theoretic concepts

3.2.1 Dominating sets and internal nodes

Let G be the graph that corresponds to a given wireless network. A set is dominating if all the nodes in G are either in the set or neighbors of nodes in the set. For example, the clusterheads and border nodes in a clustered structure define a dominating set. Nodes that belong to a dominating set will be called internal nodes for G (of course, a different definition for a dominating set leads to a different set of internal nodes). Routing based on a connected dominating set is a frequently used approach [WL], where the searching space for a route is reduced to the corresponding internal nodes. It is desirable to create dominating sets with minimal possible ratio of internal nodes. However, the savings in the communication overhead in maintaining routing tables shall not be overshadowed by the overhead imposed in maintaining the dominating set structure. In [WL] a literature review of several existing dominating set definitions is given, those definitions have significant overhead in maintaining the structure and do not produce better ratios of internal nodes than the simple definitions given in [WL].

Wu and Li [WL] proposed a simple and efficient distributed algorithm for calculating connected dominating sets in *ad hoc* wireless networks. They introduced the concept of an *intermediate* node. A node A is an *intermediate* node if there exist two neighbors B and C of A that are not direct neighbors themselves. Wu and Li [WL] introduced also two rules that considerably reduce the number of internal nodes in the network. Let $N(u)$ be the (open) set of all neighbors of node u, and let $N[u] = N(u) \cap \{u\}$ be the corresponding closed neighbor set, that is the set of all

neighbors and *u* itself. Suppose that each node has a unique *id* number (it may be obtained by generating a random number in [0,1], or their *x*-coordinate may serve the purpose). Let us define the *inter-gateway* nodes as the intermediate nodes that are not eliminated by Rule 1. Next, let the *gateway* nodes be those intermediate nodes that are not eliminated by both rules.

- Rule 1 [WL] is as follows:
  Consider two intermediate nodes *v* and *u*. If $N[v] \subseteq N[u]$ in *G* and $id(v) < id(u)$, then node *v* is not an *inter-gateway* node. In other words, if any neighbor of *v* is also a neighbor of *u*, and *v* is connected to *u* and has lower *id*, then a path via u can replace any path via v, thus node *v* is not needed as internal node. We may also say that node *v* is "covered" by node *u*. The number of internal nodes can be further reduced by applying Rule 2 [WL].
- Rule 2 [WL] is as follows:
  Assume that, after applying Rule 1, *u* and *w* are two inter-gateway neighbors of an inter-gateway node *v*. If $N[v] \subseteq N[u] \cup N[w]$ in *G* and $id(v) = min\ \{id(v), id(u), id(w)\}$, then node *v* is declared a non-gateway node. In other words, if each neighbor of *v* is a neighbor of *u* or *w*, where *u* and *w* are two connected neighbors of *v*, then *v* can be eliminated from the list of gateway nodes (when, in addition, *v* has lowest *id* among the three). The hop count between a source and destination node may increase by one in this process, since a segment *pvq* of the path between them is replaced by a segment *puwq* which is one hop longer.

If locations of nodes are available, each node can determine whether or not it is an intermediate, inter-gateway or gateway node in $O(k^2)$ computation time (where *k* is the number of its neighbors), and without any message exchanged with its neighbors for that purpose. Otherwise, the maintenance of internal node status requires each node to know the list of neighbors for each of its neighbors.

In [SSZ], [SSS] proposed to replace node *ids* with a record (*degree*, *x*, *y*), where *degree* is the number of neighbors of a node, and *x* and *y* are its two coordinates in the plane. In both rules from [WL], nodes compare first their degrees, and the node with higher degree has greater chances of remaining as an internal node. In case of ties, the *x*-coordinate is used to resolve (or an *id*, if location is not available). If the *x*-coordinates also happen to be the same, use the *y*-coordinate for the final decision. Such comparison rule will result in fewer remaining nodes in the graph. The example in Figure 3.2 has used so defined record instead of *ids*. The information about the

degree of neighboring nodes may be gathered together with information about their location.

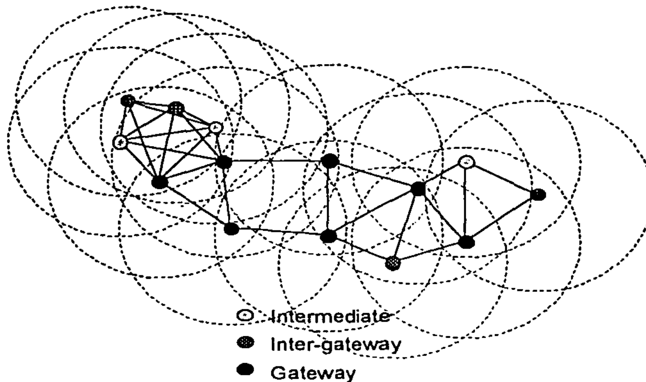


Figure 3.2: Representation of intermediate, inter-gateway and gateway nodes.

3.2.2 Relative Neighborhood Graph

The relative neighborhood graph (RNG) is a geometric and graph theoretic concept proposed by Toussaint [T]. The definition of RNG is as follows. An edge *(u,v)* exists between vertices *u* and *v* if the distance between them, *d(u,v)*, is less than or equal to the distance between every other vertex *w*, and whichever of *u* and *v* is father from *w*. In other words, $\forall w \neq u, v : d(u,v) \leq \max[d(u,w), d(v,w)]$. Thus, for an edge *(u,v)* to be included, the intersection of two circles centered at *u* and *v* and with diameter *uv* in Figure 3.2 (shaded area) should contain no vertex *w* from the set. In Figure 3.3 *uv* is not in RNG because of witness node *w*. Figure 3.4 shows the RNG on a set of six nodes (in this case the RNG is a spanning tree, which is not always the case).

Toussaint [T] proved several important properties of relative neighborhood graphs, which are necessary for their application in a broadcasting task. RNG is a connected graph, and is a planar graph. The planarity of the graph assures that it is a sparse graph. It is well known that a planar graph with *n* nodes may have at most *3n-6* edges [SSS].

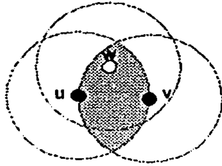


Figure 3.3: Edge (u,v)is not in RNG because of a witness w point. (Taken from [SSS])

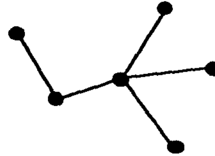


Figure 3.4: The Relative Neighborhood Graph of 6 points. (Taken from [SSS])

3.3 Broadcasting algorithms

In a one-to-all model, transmission by each node can reach all nodes that are within a radius distance from it, while in the one-to-one model, each transmission is directed toward only one neighbor. A one-to-one communication model emerges when narrow beam directional antennas are used, or when each node receives messages on its own frequency, and each node is aware of the frequency of each of its neighbors.

Broadcasting using the conventional mechanism of classic flooding gives different strengths and limitations. A flooding task starts with a source node sending its data to all of its neighbors. Upon receiving a packet of data, each node then stores it and sends a copy of it to all of its neighbors. This is therefore, a straightforward protocol requiring no protocol state at any node. It also disseminates information quickly in a network where the bandwidth is not a scarce and links are not loss-prone [HKB].

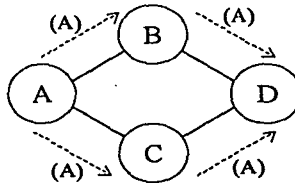


Figure 3.5: The implosion problem. Node A starts by flooding its data to all of its neighbors. Node D receives two copies of the data.

Two deficiencies of this simple approach render it inadequate as a protocol for *ad hoc* wireless networks, since in them, bandwidth is a limitation:

- *Implosion:* in classic flooding, a node always sends data to its neighbors, regardless of whether or not the neighbor has already received the data from another source. This leads to the implosion problem illustrated in Figure 3.5. Here, node *A* starts out by flooding data to its two neighbors. *B* and *C*. These nodes store the data from *A* and send a copy of it on to their neighbor *D*. The protocol thus wastes resources by sending two copies of the data to *D*. It is easy to see that implosion is linear in the degree of any node.
- *Resource blindness:* in classic flooding, nodes do not modify their activities based on the amount of CPU power or bandwidth available to them at a given time. A network of nodes can be "resource-aware" and adapt its computation and communication to the state of its available resources.

3.3.1 Classic Flooding

In classic flooding, a node wishing to disseminate a message across the network starts by sending a copy of this message to all its neighbors. Whenever a node receives new information makes a copy of it and sends it all its neighbors, except the node from which it just received it. The amount of time it takes a group of nodes to receive the same message and then forward it on to their neighbors is called *round*. The algorithm finishes, or *converges*, when all the nodes in the network have received a copy of the message. Flooding converges on *O(d)* rounds, where *d* is the diameter of the network, because it takes at most *d* rounds for a piece of data to travel from one end of the network to the other.

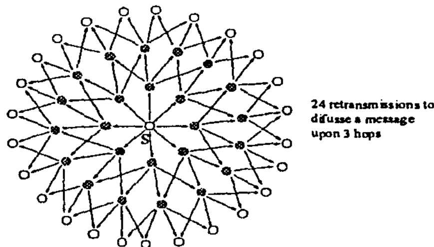


Figure 3.6: Diffusion of Broadcast message using pure flooding. Twenty-four messages are retransmitted when only 3 hops have been done. (Figure from [QVL])

TESIS CON FALLA DE ORIGEN

3.3.2 Gossiping

Gossiping [HHL] is an alternative to the classic flooding approach that uses *randomization* to conserve energy. Instead of indiscriminately forwarding data to all its neighbors, a gossiping node only forwards information on to one randomly selected neighbor. If a gossiping node receives data from a given neighbor, it can forward it back to that neighbor if it randomly selects that neighbor.

Whenever data travels to a node with high degree in a classic flooding network, more copies of the data start floating around the network. At some point, however, these copies may end up imploding. Gossip avoids such imploding because it only makes one copy of each message at any node. The fewer copies made, the lower the likelihood that any of these copies will ever implode.

While gossiping distributes information slowly, it dissipates energy at a slow rate as well. Consider the case where a single data source disseminates data using gossiping. Since the source sends to only one of its neighbors, and that neighbor sends to only one of its neighbors, the fastest rate at which gossiping distributes data is 1 node/round. Thus, if there are c data sources in the network, gossiping's pastes possible distribution rate is c nodes/round.

3.3.3 New Broadcasting algorithm

In some applications, such as route discovery or paging, message to be broadcast can be considered as "short" one. On the other hand, file sharing is an example of a "long" message. In [HKB], the concept of short and long messages was proposed. Each node will receive long message exactly once, which is optimal. Thus we will only try to optimize the number of short messages. In flooding algorithm [HKB], the number is equal to the total number of edges in the network. We propose to reduce that number by applying concepts of the RNGs.

3.4 Broadcasting via RNG

In the case of unit graphs, the test for a given edge (whether or not it belongs to the RNG) may be applied to neighboring nodes only. Since location update between neighboring nodes is performed regularly for other reasons (routing for example), the selection of edges for the planar graph is local and requires no additional communication overhead. Edge (u,v) belongs to the constructed planar subgraph if and only if none of the common neighbors of u and v is located inside the mentioned intersection of two circles. Let $U(S)$ and $RNG(S)$ be unit graph and relative

neighborhood graphs defined on the set of nodes S. We shall prove the following lemma.

Lemma 1. If $U(S)$ is connected then $U(S) \cap RNG(S)$ is connected.

Proof. It is well known that the minimum spanning tree $MST(S)$ is a subset of $RNG(S)$ [T]. Thus we only need to prove that $MST(S) \subseteq U(S)$ $U(S)$ is connected. The proof of this fact is given in [BMSU].

The planar subgraph based broadcast algorithm, referred to as RNG broadcast algorithm, can be defined as follows. It constructs planar subgraph (for instance, RNG) for all the nodes. The number of short messages in the algorithm is therefore equal to the total number of edges in the planar subgraph, which is at most $3n-6$ for a network with n nodes. We propose to use RNG here because it was the sparsest planar subgraph that can be locally defined, to the best of our knowledge. RNG is a subgraph of the Gabriel graph [BMSU] which is another planar locally defined subgraph which is more appropriate for use in routing with guaranteed delivery [BMSU], which prefers more edges. The Gabriel graph can be also used for broadcasting, but will result in fewer retransmission savings [SSS].

Chapter 4

Internet graphs

In the previous chapter we introduced the concept of RUG that are those graphs used to model wireless *ad hoc* networks. Since the main goal of this thesis is to provide an alternative to the Internet flooding, we face to the need of having a model to simulate the Internet topology. This chapter provides a literature review of this task, presents the models we have used in our simulations and the tools employed to generate them.

In general, Internet studies and simulations assume certain topological properties or use synthetically generated topologies. If such studies are to give accurate guidance as to Internet–wide behavior of the protocols and algorithms being studied, the chosen topologies must exhibit fundamental properties or invariants empirically found in the actual existing structure of the Internet. Otherwise, correct conclusions cannot be drawn [MLBM]. Studies of Internet topology have show that Internet graphs follow a power law distribution [FFF] so a simple random distribution would not accurately represent an Internet topology.

In strict terms, Internet topology refers to the physical (or logical) structure of Internet at a given time and Internet graph refers to the mathematical abstraction to represent that structure, although, without loss of formality, we will use the terms Internet graph and Internet topology indistinctly.

4.1 Network topologies

Complex weblike structures describe a wide variety of systems of high technological and intellectual importance. For example, the cell is best described as a complex network of chemicals connected by chemical reactions; the Internet is a complex network of routers and computers linked by various physical or wireless links; fads and ideas spread on the social network whose nodes are human beings and edges represent various social relationships; the Wold-Wide Web is an enormous virtual network of WebPages connected by hyperlinks. These systems represent just a few of the many examples that have recently prompted the scientific community to investigate the mechanisms that determine the topology of complex networks. The

desire to understand such interwoven systems has brought along significant challenges as well [AB]. Physics and mathematicians have developed an arsenal of tools to predict the behavior of a system as a whole from the properties of its constituents.

Traditionally the study of complex networks has been the territory of graph theory. While graph theory initially focused on regular graphs, since the 1950's large-scale networks with no apparent design principles were described as random graphs, proposed as the simplest and most straightforward realization of a complex network. Random graphs were first studied by the Hungarian mathematicians Paul Erdös and Alfréd Rényi. According to the Erdös-Rényi model, in this model we start with N nodes and connect every pair of nodes with probability p, creating a graph with approximately $pN(N-1)/2$ edges distributed randomly [AB]. This model has guided the thinking about complex networks for decades after its introduction, but the growing interest in complex systems prompted many scientists to reconsider this modeling paradigm and ask if real networks behind such diverse complex systems as the cell or the Internet are fundamentally random. The intuition could indicate that complex systems must display some organizing principles, which should be at some level encoded in their topology as well. But if the topology of these networks indeed deviates from a random graph, it is needed to develop tools and measures to capture in quantitative terms the underlying organizing principles.

In the past few years there have been numerous advances in this direction, prompted by several parallel developments. First, the computerization of data acquisition in all fields led to the emergence of large databases on the topology of various real networks. Second, the increased computing power allows researchers to investigate networks containing millions of nodes, exploring questions that could not be addressed before. Third, the slow but noticeable breakdown of boundaries between disciplines offered researchers access to diverse databases, allowing them to uncover the generic properties of complex networks. Finally, there is an increasingly voiced need to move beyond independent approaches and try to understand the behavior of the system as a whole. Along this route, understanding the topology of the interactions between the components, *i.e.* networks, is unavoidable [AB].

Motivated by these converging developments and circumstances, many quantities and measures have been proposed and investigated in depth in the past few years. However, three concepts occupy a prominent place in contemporary thinking about complex networks. These three concepts are briefly defined below:

Small worlds: The small world concept in simple terms describes the fact that despite their often large size, in most networks there is a relatively short path between any two nodes. The distance between two nodes is defined as the number of edges along the shortest path connecting them. The most popular manifestation of "small worlds" is the "six degrees of separation" concept, uncovered by the social psychologist Stanley Milgram [M], who concluded that there was a path of acquaintances with typical length about six between most pairs of people in the United States [K]. The small world property appears to characterize most complex networks: the actors in Hollywood are on average within three costars from each other, or the chemicals in a cell are separated typically by three reactions [AB]. The small world concept, while intriguing, is not an indication of a particular organizing principle. Indeed, as Erdös and Rényi have demonstrated, the typical distance between any two nodes in a random graph scales as the logarithm of the number of nodes. Thus random graphs are small worlds as well.

Clustering: A common property of social networks is that cliques form, representing circles of friends or acquaintances in which every member knows every other member. This inherent tendency to clustering is quantified by the clustering coefficient [WS]. Let us focus first on a selected node i in the network, having k_i edges which connect it to k_i other nodes. If the first neighbors of the original node were part of a clique, there would be $k_i(k_i-1)/2$ edges between them. The ratio between the number E_i of edges that actually exist between these k_i and the total number $k_i(k_i-1)/2$ gives the value of clustering coefficient of node i

$$C_i = \frac{2E_i}{k_i(k_i-1)}$$

the clustering coefficient of the whole network is the average of all individual C_i's.

In a random graph, since the edges are distributed randomly, the clustering coefficient is $C=p$. However, it was Watts and Strogatz [WS] who first pointed out that in most, if not all, real networks the clustering coefficient is typically much larger than it is in a random network of equal number of nodes and edges.

Degree distribution: Not all nodes in a network have the same number of edges. The spread in the number of edges a node has, or node degree, is characterized by a distribution function $P(k)$, which gives the probability that a randomly selected node has exactly k edges. Since in a random graph the edges are placed randomly, the majority of nodes have approximately the same degree, close to the average degree $\langle k \rangle$ of the network. The degree distribution of a random graph is a Poisson

distribution with a peak at $P(\langle k \rangle)$. On the other hand recent empirical results show that for most large networks the degree distribution significantly deviates from a Poisson distribution. In particular, for a large number of networks, including the World-Wide Web [AJB], Internet [FFF] or metabolic networks [JTAOB], the degree distribution has a power-law tail:

$$P(k) \sim k^{\gamma}$$

Such networks are called scale-free [BA]. While some networks display an exponential tail, often the functional form of $P(k)$ still deviates significantly from a Poisson distribution expected for a random graph.

These three concepts, small path length, clustering and scale-free degree distribution have initiated a revival of network modeling in the past few years, resulting in the introduction and study of three main classes of modeling paradigms. First, random graphs, which are variants of the Erdös-Rényi model, are still widely used in many fields, and serve as a benchmark for many modeling and empirical studies. Second, following the discovery of clustering, a class of models, collectively called small world models, has been proposed. These models interpolate between the highly clustered regular lattices and random graphs. Finally, the discovery of the power-law degree distribution has led to the construction of various scale-free models that, by focusing on the network dynamics, aim to explain the origin of the power-law tails and other non-Poisson degree distributions seen in real systems [AB].

4.2 Empirical results for Internet topology

The Internet is the network of physical links between computers and other telecommunication devices [AB]. The topology of Internet has been studied from different approaches according to its interconnection level. At the *router level* the nodes are the routers, and edges are the physical connections between them. At the *interdomain* (or Autonomous System) level each domain, composed of hundreds of routers and computers, is represented by a single node, and an edge is drawn between two domains if there is at least one route that connects them [AB].

But actually, modeling the topology of Internet, in order to more accurately represent it, is usually developed based on current topological information that is not completely accurate [MLMB]. Such lack of accuracy is mainly due to the fact that mapping the Internet topology is a very challenging task [GT, TR]. At the Autonomous System (AS) level, available information is richer because it can be obtained or inferred from Border Gateway Protocol tables [NLANR, G]. In contrast,

accurate router-level topological information is hard to obtain and until now inferring router-level connectivity has been done by using traceroute or traceroute-like probing mechanisms [GT, CM]. Identifying the actual fundamental properties of topologies at the router level is still an open research question [MMB].

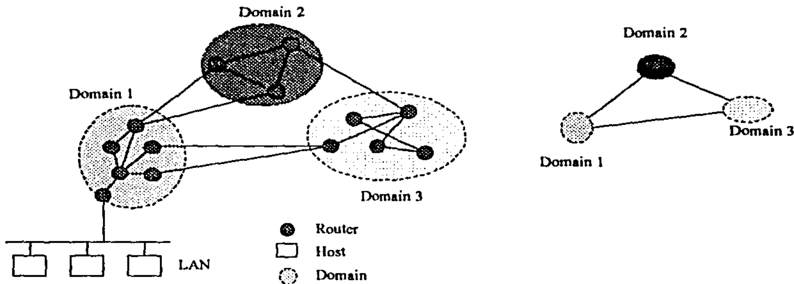


Figure 4.1: The structure of Internet at the router level (left) and at inter-domain level (right). Host connects to routers in LANs.

As shown in [MLMB], research in topology generation and modeling is a growing and dynamic area. For testing our algorithm we focus here on graphs that model the Internet at the router level, we will use three different models, one now obsolete but used at the early days of Internet topology modeling and used now as reference. The other two models comply with the power-law degree distribution used nowadays as the main parameter for Internet graphs.

Faloutsos *et al.* [FFF] have studied the Internet at both router and AS levels concluding that in both cases the degree distribution follows a power-law. The AS topology of the Internet, captured at three different dates between 1997 and 1998, resulted in degree exponents between $\gamma_I^{as} = 2.15$ and $\gamma_I^{as} = 2.2$ [AB]. The 1995 survey of the Internet topology at the router level, containing 3,888 nodes found $\gamma_I^r = 2.48$ [FFF]. More recently, Govindan and Tangmunarunkit [GT] mapped the connectivity of nearly 150,000 router interfaces and nearly 200,000 router adjacencies, confirming the power-law scaling with γ_I^r ; 2.48 [AB]. Additionally, the Internet as a network, displays clustering and small path length as well, [YJB, PV] studied the Internet at the domain level between 1997 and 1999 and found that its clustering coefficient ranged between 0.18 and 0.3 to be compared with $C_{rand} \approx 0.001$ for random networks of similar parameters. The average path length of the Internet at the domain level ranged

between 3.7 and 3.77 [YJB, PV] and at the router level was around 9 [YJB] indicating its small world character.

4.3 Power laws

Faloutsos *et al.* [FFF] have studied the Internet at both levels and concluded that in each case the degree distribution follows a power-law. They have additionally proposed three more power-laws, jointly with a degree distribution, known as the four power-laws, those are: rank exponent, out-degree exponent, hop-plot exponent, and *eigen* exponent. For convenience, we repeat the definitions here.

Power-Law 1: rank exponent **R**. The out-degree, d_v, of a node v, is proportional to the rank of the node, r_v, to the power of a constant, **R**: $d_v \propto r_v{}^R$. This power-law is evaluated by computing the out-degree for every node, sorting the out-degrees in descending order, and plotting in log-log space. Data for inter-domain graphs and router graphs show a near-linear distribution, with a slight deviation at either end.

Power-Law 2: out-degree exponent **O**. The frequency, f_d, of an out-degree, d, is proportional to the out degree to the power of a constant, **O**: $f_d \propto d^O$

Power-Law 3: hop-plot exponent **H**. The total number of pairs of nodes, *P(h)*, within h hops, is proportional to the number of hops to the power of a constant, **H**: $P(h) \propto h^H \quad h < \delta$, the diameter.

Power-Law 4: *eigen* exponent **E**. The *eigenvalues*, λ_i, of a graph are proportional to the order, i, to the power of a constant, **E**: $\lambda_i \propto i^E$.

[FFF] suggest using these power-laws to measure the realism of synthetic Internet Graphs.

4.4 Topology network models

4.4.1 Waxman [W]

The Waxman model was one of the earliest that proposed a non-random edge distribution but based on a probability function that regards the distance between two selected nodes.

Steps to generate a topology according to the Waxman model are:

- n nodes are randomly distributed over a rectangular coordinate grid.

- For each pair of nodes, a distance is chosen in (0, L) range from an uniform random distribution.
- Edges are introduced between pairs of nodes u, v with a probability that depends on the distance between them. The edge probability is given by $P(\{u,v\}) = \beta^{\frac{-d(u,v)}{L\alpha}}$
- Larger values of β result in graphs with higher edge densities , while small values of α increase the density of short edges relative to longer ones.

4.4.2 Palmer and Steffan [PaS]

This recursive algorithm consists of two parts, the definition of a probability distribution function that will randomly select a pair of nodes and the use of this probability density function to produce a network graph with real-valued edge weights.

The probability density function is a generalization of an 80-20 distribution. The steps for Recursive algorithm is as follows:

- Assume for simplicity that the number of nodes N is a power of 2, and M edges are desired.
- Partition the graph into two node sets V_1 and V_2 each of size $N/2$. Let a fraction p_{ij} of edges go from some node in V_i to some node in V_j, $i, j = 1; 2$. $P = (p_{ij})$ are provided as parameters. The idea is to favor intra-community links by having p_{11} and p_{22} larger than p_{12} or p_{21}, hence the name "80-20".
- Using P, pick one of the four possible types of edges.
- Recurse within the 1/4 of edges that are consistent with the above choice; continue recursing with the same parameters P until a specific edge is materialized (i.e., until the number of remaining nodes equals 1 or 2).
- Repeat until M edges are added to the graph.

4.4.3 Barabàsi and Albert [BA]

Barabàsi and Albert arrived to the conclusion of a power-law distribution via a more general kind of networks: scale free networks, its properties apply not only to Internet Topologies but to a variety of fields such as biology, social sciences or chemistry. The Barabàsi's model is based on two main ideas: growth over time and preferential attachment.

Growth and preferential attachment

- Most real network systems grow in time.
- Most real network exhibit preferential connectivity: a new node is most likely to be attached to an important existing node with large connectivity.

4.4.3.1 *Algorithm to generate scale free networks [BA2]*

- Start with m_0 nodes without links.
- At every time step, perform one of the following operations:
  - With probability p add m ($m < m_0$) new links. For this select randomly a node as the starting point of the new link. The other end of the link, however, is selected with probability

    $$\Pi(k_i) = \frac{k_i + 1}{\sum_j (k_j + 1)}$$

    incorporating the fact that new links preferentially point to popular nodes, with a high number of connections [BA]. This process is repeated m times.
  - With probability q rewire m links. For this we randomly select a node i and a link l_{ij} connected to it. Next we remove this link and replace it with a new link $l_{ij'}$ that connects i with node j' chosen with probability $\Pi(k_{j'})$. This process is repeated m times.
  - With probability $1-p-q$ add a new node. The new node has m new links that with probability $\Pi(k_i)$ are connected to nodes i already present in the system.
- After t time steps, the model leads to a random network with $N(t) = m_0 + (1-p-q)t$ vertices and $\sum_j k_j = (1-q)2mt - m$ edges. This network evolves onto a scale-invariant state with the probability that a vertex has k edges, following a power-law such that:

$$P(k) \propto k^{-\gamma}$$

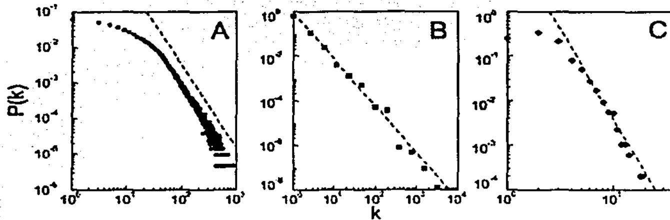


Figure 4.2: Barabàsi scale-free networks. The distribution function of connectivities for various large networks. (A) Actor collaboration graph with N=212,250 vertices and average connectivity <k>=28.78. (B) The WWW, N=325,729, <k>=5.46. (C) Power grid data N=4941, <k>=2.67. The dashed lines have slopes (A) γ_{actor}=2.3, (B) γ_{www}=2.1 and (C) γ_{power}=4. (Figure from [BA])

4.5 Topology Generator

4.5.1 BRITE

BRITE is an universal topology generator created by Medina, Lakhina, Matta and Byers at Boston university [MLMB]. BRITE implements various generation models for both Flat Router level and AS-level graphs. Waxman and Barabàsi models, among others, are implemented in BRITE. Selection of BRITE for generating employed topologies was because of its:

- *Flexibility*: Generates topologies over a wide range of sizes.
- *Efficiency*: Generates large topologies (e.g. number of nodes > 100; 000) with reasonable CPU and memory consumption.
- *User-friendliness*: Follows the usage principles of standard user interfaces. The user should learn the mechanics of the generation tool only once.

4.5.1.1 How BRITE works

The specific details regarding how a topology is generated depend on the specific generation model being used. We can think of the generation process as divided into a four-step process:

- Placing the nodes in the plane
- Interconnecting the nodes

- Assigning attributes to topological components (delay and bandwidth for links, AS *id* for nodes, etc.)
- Outputting the topology to a specific format.

This of course is not a clear-cut division that will fit every generation model but conceptually reflects what happens when a topology is being generated. Also, several models may share specific steps during the generation process, while other models differ significantly on the individual steps. We show below some possible options when placing the nodes and assigning bandwidths for topologies to be generated.

4.5.1.2 Placing the nodes

- Randomly: each node is placed in a randomly selected location of the plane.
- Heavy-tailed: divides the plane into squares (the size of the plane and the size of the squares is controlled by parameters passed to BRITE.). Each of these squares is assigned a number of nodes drawn from a heavy-tailed distribution. Once that value is assigned, then that many nodes are placed randomly in the square.

4.5.1.3 Assigning Bandwidths

Executed once the topology has been completely generated. BRITE assigns bandwidths to links according to one of four possible distributions.

- Constant: the value specified by *BW min* (equal for all links in the topology).
- Uniform: a value uniformly distributed between *BW min* and *BW max*
- Exponential: a value exponentially distributed with mean *BW min*
- Heavy-tailed: a value heavy-tailed distributed (Pareto distribution with shape 1.2) with minimum value and maximum value equal to *BW max*

4.5.1.4 BRITE output format

A BRITE-formatted output file contains three sections:

- Model information: information about the topology contained in the file. Includes the number of nodes and edges, and information specific to the model used to generate the topology.

- Nodes: for each node in the graph, a line is written into the output file with the following format: `NodeId xpos ypos indegree outgdegree ASid type`
- Edges: for each edge in the graph, a line with the following format is written in the output file: `EdgeId from to length delay bandwidth ASfrom ASto type`

NodeId	Unique id for each node
xpos	x-axis coordinate in the plane
ypos	y-axis coordinate in the plane
indegree	Indegree of the node
outdegree	Outdegree of the node
ASid	id of the AS this node belongs to (if hierarchical)
type	Type assigned to the node (e.g. router, AS)

Table 4.1. Nodes section Field Meaning

EdgeId	Unique id for each edge
from	node id of source
to	node id of destination
length	Euclidean length
delay	propagation delay
bandwidth	bandwidth (assigned by AssignBW method)
ASfrom	if hierarchical topology, AS id of source node
Asto	if hierarchical topology, AS id of destination node
Type	Type assigned to the edge by classification routine

Table 4.2. Edges section Field Meaning

Chapter 5

Peer-to-peer networks

The importance and impact of the savings in the message retransmissions can be better valued in the context of an application. In the following sections of this chapter we make a review of the communication protocols as implemented in the Gnutella network, a fully decentralized peer-to-peer network, showing that its current implementation can limit its scalability and availability. We show an alternative to flooding proposed in [PS] and in chapter 7 we will make a comparison of the obtained results when using the RNG algorithm proposed in this thesis.

Flooding is the process used traditionally by Internet link-state protocols for disseminating new information across an entire routing area after changes in any state link. The intrinsic disadvantages of flooding are not a limitation when topology changes are not so frequent or network size is small enough, both compared with the available bandwidth or the computational resources. Although, overcoming these disadvantages represents a challenge in a conventional Internet structure and organization, it has been faced with relative success [NaS]. Nowadays, there has been a growing interest in a kind of networks called peer-to-peer, sparked by the popularity of sharing applications such as Napster, Gnutella or Morpheus [PS]. Some typical characteristics of a peer-to-peer system are that all the conforming nodes are equal participants into a network without special entities with facilitating or administrative roles. In addition that network topology is dynamic since peers continuously join and leave the network.

5.1 Definition of Peer-to-Peer

As with many new technologies, there is no a single universally accepted definition for peer-to-peer (P2P). The recently formed Peer-to-Peer Working Group, a consortium lead by the industry giants such as Hewlett-Packard, Intel and IBM, defines peer computing as "sharing of computer resources by direct exchange". Indeed it is this notion of direct access to resources, instead of through a centralized server as with the traditional client-server model, which characterizes P2P. However, this

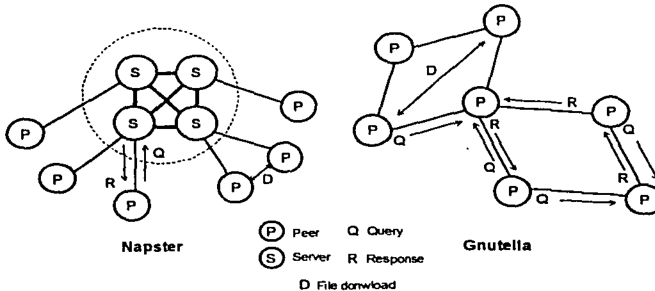
definition may be too general, as it would seem to include applications typically considered client-server, such as FTP and TELNET. According to [S],the two fundamental criteria that each P2P application must satisfy are:

- treating the variable connectivity and the temporary network addresses as the norm and,
- giving nodes at the edges of the network significant autonomy.

Using this definition, applications such as email are not P2P since addresses are not machine independent, while instant messaging applications such as ICQ and Jabber are P2P, because "they devolve connection management to the individual nodes" and dynamically map users to their IP addresses. However, the fundamental idea of having computers to act as peers is hardly new, some may even argue it has its root in the original design of the Internet, since it was part of the early ARPANET architecture. In fact, early network applications such as USENET and DNS were based on a peer-to-peer communication model and can be considered predecessors to modern P2P technologies. The true innovation of these technologies therefore lies not in their architecture design, but rather in their implementation and scale. In order for these applications to extend the scope of P2P computing beyond a single LAN, they needed to overcome serious technical challenges posed by technologies such as firewalls, dynamic IP, and NAT, designed to obstruct open communications between computers for reasons of security. They did so by mitigating the application complexity to the edges of the network, thereby creating a much more significant role for the Internet-connected PCs than previously offered by the traditional client-server model. This idea of transferring the complexity to the edges can be best explained in comparison with a telephone network. At first glance a telephone network may seem P2P, since communication occurs directly between two points in the network. However the crucial difference between a telephone network and P2P is that the former relies on an intelligent network for functions such as routing, and relatively "dumb" devices in the form of telephone sets. In contrast, P2P application like Gnutella relies on an existing, "dumb" network (the Internet) and incorporates all the application logic at endpoints. The main advantage to such design from a perspective of a researcher is that it enables rapid development and deployment of innovative technologies, which can perhaps serve as an explanation for such a large number of P2P applications we are seeing today.

Current network applications have embraced three forms of peer computing: sharing of information, sharing of computing power, and communication. This does not mean the P2P computing model is limited to these resources, but simply that a P2P

application for sharing other types of resources has not yet been designed. Applications such as SETI@Home [SETI] outline a clear relationship between P2P and another computing paradigm commonly referred to as distributed computing. These applications allow the computing power of thousands of Internet-connected PCs to be harnessed and used for performing computationally intensive tasks that would otherwise require the use of a supercomputer. Examples include processing radio signals from outer space in search for extraterrestrial intelligence [SETI] and simulating protein folding [F]. Perhaps the most popular form of peer computing on the Internet is instant messaging. Unlike email, where messages travel through centralized mail servers, instant messaging allows individuals to directly communicate with each other. To route messages between users across the entire Internet, applications such as AIM, ICQ, MSN, and Jabber rely on a centralized back-end server to dynamically map users to their IP addresses and in case of ICQ, buffer



messages when the user is offline.

Figure 5.1: File location in Napster and Gnutella. (Model taken from [SGG])

Ongoing work toward development of a generalized platform for building P2P applications [Gr] can be perhaps taken as an indication that the P2P model is here to stay. The main goal of Groove developers is abstracting away many common challenges to building P2P network applications, such as providing open PC-to-PC communication. The main obstacles are arising from the fact that the Internet architecture has been built for years around the prevalent client-server model. As a result, numerous technologies such as firewalls, dynamic IP, NAT, and asymmetric bandwidth connections have been deployed on the Internet, driven by the fundamental assumption that most Internet-connected PCs will only serve as clients. This underlying assumption is being strongly challenged by P2P applications such as

Gnutella, Napster, and Freenet, which strive to provide a fully distributed worldwide information sharing system. These applications require their users to serve both as consumers and producers of information in a large distributed information storage system. The idea behind peer-to-peer information sharing is that much of the desired content is stored on individual workstations and not behind some centralized server. Applications like Gnutella allow users to directly connect to each other for the purpose of exchanging information. From the perspective of this thesis, a common thread that ties all of these applications is that they all form highly dynamic networks of peers with a complex topology.

Membership in a peer-to-peer system is *ad hoc* and dynamic: as such the challenge of such system is to figure out a mechanism and architecture for organizing the peers in such a way so that they can cooperate to provide a useful service to the community of users. For example in a file sharing application, one challenge is organizing peers into a cooperative, global index so that any content can be quickly and efficiently located by any peer in the system [SGG]. Based on this, we propose an application for our broadcasting algorithm over a well studied network such as Gnutella; as shown in the following section, the Gnutella protocol make use of the flooding mechanism for query actions and for peer discovering. As known, flooding dictates that each host is to simply forward each received message to all of its neighbors, except the one from which the message was received. As such, flooding provides a simple and effective way of broadcasting messages in a dynamically changing network without requiring the use of routing tables or knowledge of the global network topology. However it clearly does not scale for Internet-wide applications, as it generates a large number of redundant messages and uses all available paths across the network [J]. This, among other factors, could lead to fragmentation of the network, as occurred to Gnutella users in summer of 2000 when the size of user community rapidly increased.

5.2 The Gnutella Network

We provide here an example of common connection when a new peer A wants to join the Gnutella network through peer B already in the network. The description is based in process as expressed in the Gnutella protocol specification [GPS] and the work presented by Portmann and Seneviratne [PS].

When a new node (node A) wants to connect to the Gnutella network, it needs to get the address of at least one node (node B) currently connected to the network. How this address is obtained is not specified by the Gnutella protocol. Then, node A

connects to node B via TCP and sends a GNUTELLA CONNECT request. If node B is willing to accept the connection it replies with a GNUTELLA OK message. At this point, node A is part of the Gnutella overlay network and it can do two things: to connect to additional nodes or start a file searching, one of the more important services Gnutella provides.

For more peers to connect, it can discover more peers on the network by sending out PING messages. These PING messages are forwarded via flooding to all the nodes within the reach of the sending node. Nodes that receive a PING, reply with a PONG message, containing the sending node address and information about the file it shares. The PONG is then routed back to the sender of the PING. With this information, node A then can decide to connect to additional nodes in the network, in the same way as it connected to the first node.

If a node wants to search for a specific file, it sends a QUERY message, containing a search string to all its neighbors. This message is then broadcasted to all the Gnutella participants. Each node looks in its list of shared files for a match with the query string. If such a match occurs, a QUERY_HIT message is routed back to the searching node. A QUERY_HIT message contains the IP address and port number where the file can be downloaded from. The searching node directly connects to the node that shares the requested file and downloads it via HTTP, bypassing the Gnutella overlay network. This requires that all the Gnutella nodes run a small HTTP server to serve these requests.

To implement its services, the Gnutella protocol uses the following 5 types of messages:

PING	Used to actively discover nodes on the Gnutella network. A node receiving a PING is expected to respond with a PONG message.
PONG	The response to a PING. It includes the address of the node and information regarding the amount of data it is making available on the network.
QUERY	The primary mechanism for searching in the network. A node receiving a QUERY message will respond with a QUERY HIT if a match is found against its local data set.
QUERY_HIT	The response to a QUERY. The message provides the recipient with enough information to acquire the data matching the corresponding QUERY.
PUSH	A mechanism that allows a node behind a firewall to share files with nodes on the other side of the firewall.

Table 5.1. Type of messages used by Gnutella protocol

TESIS CON
FALLA DE ORIGEN

Table 5.1. Type of messages used by Gnutella protocol

The general algorithm for Gnutella message routing is based on flooding, with some modifications. The complete set of rules is as follows:

- In general, a message received by a node is forwarded to all its neighbors, except for the one, where the message was received from.
- Each message has a Time To Live (TTL) field which is decremented by one at each visited node. If the TTL reaches zero, the message is no longer forwarded.
- Each message has a 16 byte ID to uniquely identify it on the network. Every node keeps a record of IDs of messages that it has seen in the recent past. If a node receives a message with the same ID and type as one that it has received before, the message is ignored and not forwarded.
- PONG and QUERY_HIT messages are routed a bit differently. They are only sent along the same path that carried the incoming PING or QUERY messages. For this, nodes only forward PONG or QUERY_HIT messages if they have previously seen the corresponding PING or QUERY message.

5.3 Portman and Seneviratne work on Gnutella network

Portmann and Seneviratne [PS] presented an alternative to flooding for overcoming the problems of cost and scalability in decentralized peer-to-peer networks. They proposed the use of a routing protocol called Rumor Mongering (also known as Gossip) and used the Gnutella protocol specification [GPS] as framework.

5.3.1 The Rumor Mongering algorithm

Rumor Mongering or Gossip protocols are a class of probabilistic protocols for message routing since the neighbors to whom messages are forwarded by each node are chosen randomly. [PS] evaluated one specific type of Gossip protocol, called ***blind counter Rumor Mongering***. This protocol determines how messages are routed at each node and has the following structure:

```
A node n initiates a broadcast by sending the message m to B
of its
neighbors, chosen at random.

When (node p receives a message m from node q)
    If(p has received m no more than F times)
       p sends m to B randomly chosen neighbors that p
       knows have not yet seen m.
```

The parameter B specifies the maximum number of neighbors a message is forwarded to. The parameter F determines the number of times a node forwards the same message m to B of its neighbors. The parameters F and B can be chosen to control the properties of the Rumor Mongering protocol. Broadcasting cost in Gossip protocol is equal to the total number of messages forwarded for transmitting a message over the entire network. In the Gossip protocol, a node p knows if its neighbor q has already seen the message m only if p has sent it to q previously, or if p received the message from q. In the case where the number of valid neighbors for a node is less than B, the message is only forwarded to this smaller number of neighbors.

The cost of broadcasting using flooding as it is implemented in Gnutella can be calculated knowing that during a broadcast, every node receives the message at least once. The first time the message is received, the receiving node forwards it to all its neighbors, except the one from where the message came from. Subsequent arrivals of the same message are ignored. Therefore, the number of messages that each node has to forward per broadcast is limited to the number of its neighbors, minus one. Only the node initiating the broadcast sends the message to all its neighbors, which results in an increase in the total cost by one. This guides to the following expression for calculating the cost of broadcasting a message over the entire network.

$$c = 1 + N(d-1)$$

c: cost in number of messages forwarded

d: average node degree

N: number of nodes

Table 5.2 shows the results obtained by [PS] for the performance of Rumor Mongering compared to flooding for different combinations of the parameters B and F. The results are shown for Barabàsi topologies of 1000 nodes with an average node degree d of 4, 6 and 8. The three parameters, cost, reach and time for Rumor Mongering are shown as comparative figures to the values obtained by flooding, representing 100%.

B	F	d=4		d=6		d=8	
		Cost	Reach	Cost	Reach	Cost	Reach
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%

Table 5.2. Results for Gossip protocol

Chapter 6

The broadcasting simulator

In this chapter we present a general description of the implemented tools used to apply the concepts of RUG to the Internet topologies. We begin with a brief description of the implemented algorithms and classes used in the simulation process; we then described the general steps that we followed in the broadcasting simulations comprising those made over RUGs. Finally, we include the pseudocode for generating a RUG and for obtaining the RNG from a given graph.

6.1 Implementation of algorithms

Our broadcasting algorithm was programmed completely in Java, using the JDK compiler version 1.3.4 [JDK]. Java was chosen due to its natural object-oriented structure and rapid development characteristics. For implementing the RUG and Internet topologies we made use of the Java Data Structure library (JDSL), a package developed at Brown University [JDSL]. JDSL provides an extensive collection of classes and interfaces that implement data structures and graph theory algorithms.

6.1.1 Defined classes

Table 6.1 contains the name of the classes provided in the *Graphs* Package and a brief function description. The Graphs package was the one allocating all the employed classes in our experiments. For information on the classes, methods and fields refer to Appendix B, where a complete description of the Abstract Data Types is included. Additionally, source code for all the implemented classes is included in Appendix C.

Class name	Function
`ConnectivityTester`	Checks for connectivity when generating RUG
`EdgeInfo`	Manage edge information like name, length and system reference.

graphTools	Implements various useful functions for Graphs.
InternetMap	Loads Internet topology from a given file into memory for further use.
RNG	Implements RNG based broadcasting algorithm.
RUG	Generates RUG.
TestIG	Main test program for broadcasting algorithms using Internet Graphs.
TestRUG	Main test program for broadcasting algorithms using RUG.
VertexInfo	Manages vertex properties such as name, location and reference system.

Table 6.1: Defined classes for simulation program

6.2 Simulation process

Before simulating the broadcasting process over both RUG an Internet graphs, it was needed to generate the topologies. In the case of the RUG, a Java class to generate them was implemented. For the Internet graphs we use the BRITE generator for the Barabàsi and the Waxman models and the Recursive generator for the Palmer model.

All simulations were performed on an Intel Pentium II 333 MHz machine running Mandrake Linux 8.0 and Microsoft Windows 2000. The Java code was developed indistinctly in the Borland JBuilder suite when using Windows and in the Sun NetBeans suite when using Linux. The simulation process could also be done in a Windows environment since Java is a platform independent language. All the Internet topologies, using BRITE or the Recursive generator were created under Linux, since BRITE does not provide complete information for compiling the Java source in Windows and Recursive makes use of some `gcc` compiler functions.

In the case of the RUGs it was needed to repeat the experiments presented in [SSS]. For generating the graphs we followed the value parameters in [SSS] for the number n of nodes in the graph, the dimension mxm of the grid and the average node degree d. Once the topology was generated we applied the RNG algorithm. A method for counting the number of edges in the graph before and after obtaining the RNG allowed us to take register of savings for every RUG generated. In order to obtain the

results presented in the following chapter, ten simulations were done for every reported value of *d*.

For broadcasting over the Internet graphs the process was a bit different in the first part. In order to obtain the RNG of a given topology it was first generated by using the BRITE or Recursive generators. These generators allow storing the generated topologies in a file that was used later by one class of the Graphs package.

In general terms the steps to obtaining the RNG of an Internet topology are as follow:

- The Internet topology is generated and stored in a file.
- The file is then given as an input parameter to the class `InternetMap`.
- `InternetMap` reads the file and passes the stored information to a graph data structured loaded into memory.
- The graph loaded into memory is passed to the `RNG` class, which applies the RNG algorithm and provides as output the number of edges in the graph and the average node degree before and after the application of the RNG algorithm. It also provides the savings in message retransmissions.

This process is repeated ten times for every row in the tables of results presented in the following chapter.

6.2.1 Implementation of RNG algorithm

This simulation is done in order to repeat the work in [SSS] and to prove that the algorithm for the Relative Neighbor Graphs is working properly. Below, we show the pseudocode for RUG generation and for broadcasting when applying RNG concepts.

Pseudo code for Random Unit Graph generation

```
RUG(m, n, d)
CHOOSE n points AT RANDOM FROM plane [0, m] x [0, m]
FOR_EACH pair of points IN Graph
    edge.lenght <-- distance(pair of points)
    Add edge.lenght in lenght_list
SORT lenght_list
Radius R= nd/ 2- th edge in sorted lenght_list
IF graph disconnected
    RUG(m, n, d)
```

In plane [*0, m*] x [*0, m*] each node has coordinates (*x,y*) and the distance between each pair of nodes is the Euclidean distance calculated using the nodes coordinates.

Pseudo code for getting Relative Neighbor Graph from a given graph

```
FOR_EACH node i
    FOR_EACH i.neighbor
       neighbors_list<-- i.neighbor
    FOR_EACH node j IN neighbors_list
       FOR_EACH j.neighbor
          IF j.neighbor IS IN neighbors_list
             k<--j.neighbor
             edgeij<--distance(i, j)
             edgeik<--distance(i, k)
             edgejk<--distance(j, k)
             dmax=dmax(edgeik, edgejk)
             if dmax<edgeij
             DELETE longer edgeij
```

6.3 An observation in the use of the RNG term

Before continuing to present the simulation results it is needed to make an observation to the use of the term RNG.

As defined in [T] the Relative Neighbor Graph is a connected graph and is a planar graph. Although the matter of planarity is true for the case of the RNG constructed from a RUG [SSS], this is not the case for the RNG constructed from an Internet graph. The planarity is because of the way in which a RUG is constructed as previously described in section 3.3.5.

Therefore when in this work we refer to the RNG of an Internet graph, the RNG is not as the one described in [T] but a graph constructed using the fundamental concept of neighboring nodes. Here, two nodes are said to be neighbors if they are connected by an edge regardless of the physical distance between them.

Chapter 7

Simulation Results and Analysis

In this chapter we focus on the analysis of the broadcasting simulation process results over Internet graphs when using the RNG algorithm. First, in order to prove the correct implementation of the RNG algorithm, we repeat the experiments presented in [SSS] where Random Unit Graphs (RUG) were used for modeling the *ad hoc* networks. Once the RNG algorithm has been tested, we change the RUGs to Internet Graphs, which are those modeling the Internet topology. We present results for the Waxman, Barabàsi and Palmer models used to generate them, and show that no significant difference in message retransmission savings exist among the employed models.

The first section in this chapter presents results of broadcasting when using RUG. The second one presents and analyses results when employing the Internet graphs. The third part provides some visual examples of how graphs look before and after applying the RNG algorithm. In the last section a practical application of the broadcasting via RNG is presented when comparing results to those provided in [PS] for evaluating the cost of application-level broadcasting in the Gnutella network.

7.1 The broadcasting over RUG via RNG

Results shown in Table 7.1 correspond to the average values obtained when repeating the broadcasting process ten times over RUG.

Edges	Av. Degree	Edges (RNG)	Av. Degree (RNG)	%Savings
204	4.08	123	2.46	40.0
304	6.08	124	2.48	60.0
404	8.08	128	2.56	69.0
501	10.02	122	2.44	76.0
751	15.02	125	2.5	84.0
1002	20.04	123	2.46	88.0
1252	25.04	121	2.42	91.0
1503	30.06	120	2.4	93.0
1753	35.06	121	2.42	94.0
2001	40.02	121	2.42	94.0

Table 7.1: Simulation results for broadcasting over RUG.

The numerical results shown in Table 7.1 are almost identical to those presented in [SSS]. This values are obtained when using RUGs with 100 nodes, distributed over a plane of 100 x 100 which remain constant through out the process. The average node degree d takes different values as seen in the table. These d values allow us to see how the savings in message retransmissions vary as the network density changes. It can be seen that the savings start being large even for networks with low d values. For example when $d=6$, the percentage of savings is 60%; and in the case of $d=40$, the percentage of savings is 94%. Thus, this validates our implementation of the RNG algorithm. As a next step in our work we apply the RNG concept when using Internet graphs as input. The results of this process are presented in the following section.

7.2 Broadcasting over Internet Graphs

The Internet graphs used in this work were generated using the Barabàsi [BA2], Palmer [PaS] and Waxman [W] models. BRITE [MLMB] was used as the topology generator for the Barabàsi and Waxman models and Recursive was used for the Palmer model.

7.2.1 Using Barabàsi model

Barabàsi proposed a general topology generation model, not only for internet-like topologies but for a more general kind of networks named *scale-free* [BA, BA2]. For our analysis we have established the model parameters to the following values:

```
N=1000
p=0.25
q=0.08
m=1, 2 .. 10.
```

These values were selected in order to comply with the Internet topology obtained in [GT]. They empirically found, after exploring nearly 150,000 interfaces and nearly 200,000 links that the degree distribution of the Internet topology at the router level follows a power law tail. Their Internet exploring work is almost 50 times larger than the work done by [FFF] used to formulate the power laws, confirming the power-law scaling.

According to [AB], the results presented in [GT] are modeled by a power-law $P(k) \propto k^{-\gamma}$ where $\gamma = 2.4$ and with an average degree $<k> = 2.66$. However, here we failed to produce an Internet topology with this $<k>$ as characteristic, due to known difficulties when trying to generate a connected graph with average degree less than 3. For that reason we have started with $<k>=5.49$, when m=2.

Results for the simulation process are given in Table 7.2. We used the distance between nodes as a metric for RNG, the distance was computed using *(x,y)* coordinates of every pair of connected nodes. These coordinates were also obtained by BRITE.

Edges	Average Degree	Edges (RNG)	Av. Degree (RNG)	%Savings
2745	5.49	2281	4.56	17
5598	11.2	3772	7.54	33
8319	16.64	4572	9.14	46
10820	21.64	5339	10.68	51
13385	26.77	5840	11.68	57
20565	41.13	6766	13.53	68
27310	54.62	6875	13.75	75

Table 7.2: Results using the Barabàsi model

Table 7.2 shows the percentages of retransmission savings in Internet graphs generated using the Barabàsi model [BA2] with $n=1000$ and $m=2, 4, 6, 8, 10, 20$ respectively. The savings values shown in the last column are those obtained when compared to flooding which represent 100%. From these results it is possible to see that savings in message retransmissions increases as the average node degree becomes larger. For example, a reduction of almost 50% is achieved for an average node degree of approximately 20, while in contrast, a reduction of 17% is presented in a network with an average node degree approximately 6.

7.2.2 Using the Waxman model

The Waxman model is among the earliest ones used for the generation of network topologies. Its innovation was to propose a non-random connection between nodes, defining a probability function that takes into account the separation between a pair of selected nodes. As it was not trying to specifically model the Internet topology, it fails to reproduce the power law properties [MP, PaS] discussed in [FFF], but it offers a good reference when evaluating other generation models [PaS].

The parameter values for generating the graphs used in our experiments were:

```
alfa=0.2
beta=0.2
n=500
m=4, 6, 8, 10, 20, 30, 40
```

According to [MP] and [ZCD] a combination of $\alpha = 0.2$ and β in the range [0.15, 0.2] leads to a topology with an adequate number of short edges and hop-diameter to be compared to other generation models. For comparison with the Barabàsi and Palmer models we selected $\alpha = 0.2$ and $\beta = 0.2$; *n=500* is the number of nodes and *m* allows control over the node average degree.

Edges	Average Degree	Edges (RNG)	Av. Degree (RNG)	%Savings
1000	4	984	3.94	2.0
1500	6	1430	5.72	5.0
2000	8	1832	7.33	9.0
2500	10	2185	8.74	13.0
5000	20	3338	13.35	34.0
7500	30	3662	14.65	52.0
10000	40	3704	14.82	63.0

Table 7.3: Results for broadcasting using the Waxman model

Table 7.3 shows broadcasting results when applying RNG algorithm to topologies generated by using the Waxman model. In this case, savings are smaller than those obtained when using the Barabàsi model. However, it is still possible to conclude that the savings increase as the node average degree increases.

7.2.3 Palmer and Stefan model

On [PaS] two different generation models are presented, we use the Recursive model for our experiments since it shows that a simple "80-20" random graph generator fits a power-law degree distribution. [PaS, CJPP].

The parameter values for our experiments were:

```
alpha=epsilon=0.4, beta=gamma=0.1
n=512
links=1000, 1500, 2000, 2500, 5000, 7500, 10000
```

Since the Recursive algorithm does not provide the position of the nodes, we have assigned to each node in the network a set of (x,y) coordinates, chosen randomly from a grid of $[0,m]$ x $[0,m]$. Once that position of nodes is available, we can calculate the Euclidean distance between every pair of connected nodes. In this model the parameters *alpha, beta, gamma* and *epsilon* control the percentage of links to be placed in each quadrant. In general, setting the four parameters equal to 0.25 gives a

uniform random graph. The main idea is to favor the formation of intra-community links [CJPP], since the Internet topology displays clustering organization [AB, YJB].

Edges	Average Degree	Edges (RNG)	Av. Degree (RNG)	%Savings
1000	3.91	968	3.78	4.0
1500	5.86	1421	5.55	6.0
2000	7.81	1821	7.11	9.0
2500	9.77	2109	8.24	16.0
5000	19.53	2988	11.67	41.0
7500	29.3	3091	12.07	59.0
10000	39.06	2981	11.64	71.0

Table 7.4: Results using Recursive algorithm for the length metric.

Table 7.4 shows the savings in message retransmission when using the Recursive model. This values are similar to those reached when using the Barabàsi model, again it is possible to say that the more dense a network becomes, the more savings that can be obtained when broadcasting via the RNG.

It is important to note that in the three employed models for generating the Internet graphs the savings were lower than those obtained when using the RUG. We consider that this can be explained in the following way: although the nodes in a RUG are placed randomly, the connection between nodes is restricted to a certain group of nodes that are in the range of a given node. It means, a node could be connected only to those nodes no so far away from it, which is not the case in an Internet graph, where the nodes have no restriction in the sense of the remoteness to the nodes they are connected.

The following section provides some visual examples of Internet graphs generated by BRITE when using the Barabàsi model. These pictures are only provided to show the results after applying the RNG algorithm, therefore we present graphs with a reduced number of nodes and edges to better understand the process.

7.3 Some visual examples of Internet graphs

Figure 7.1 (left) contains 30 nodes and 108 edges. It allows having a visual representation of how nodes are distributed and connected. We also show the topology after applying the RNG algorithm (right). Here, the number of edges was decreased to 48, representing a reduction of approximately 56%.

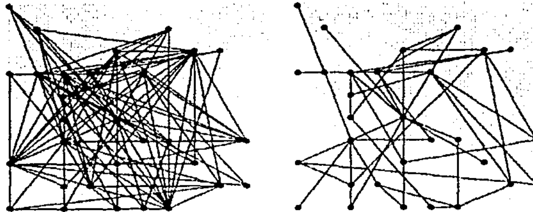


Figure 7.1: Graph generated using the Barabàsi model, it contains 30 nodes and 108 edges (left). Graph after applying RNG algorithm contains only 48 edges(right).

Figures 7.2 and 7.3 show similar graphs, but with a fewer number of nodes and edges where test by hand of the RNG algorithm can be done.

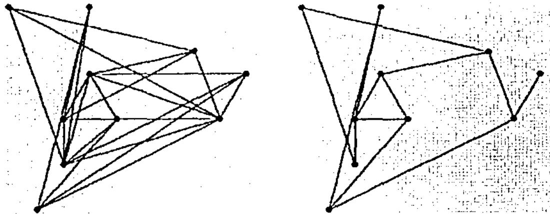


Figure 7.2: Graph before and after RNG algorithm.

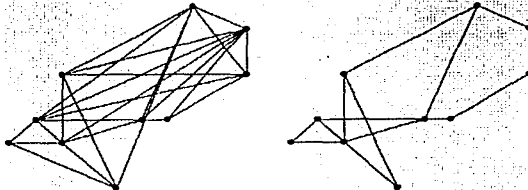


Figure 7.3: An additional example. It can be seen how after applying RNG redundant edges are eliminated.

ESTA TESIS NO SALE
DE LA BIBLIOTECA

7.4 Gossip *vs.* RNG algorithm

Once that the results of broadcasting over Internet graphs via the RNG have proven to provide important savings in message retransmissions, we present a practical application of the RNG algorithm for the field of peer-to-peer networks. It is focused on the Gnutella network where flooding is used for both Gnutella's services: searching files and peer discovery. A previous research in this topic is presented in [PS] where Rumor Mongering is proposed as an alternative to flooding. We make comparisons to the experiments presented in [PS] to see if the RNG algorithm could also represent an alternative to the Rumor Mongering protocol in terms of message savings.

As presented in [PS], in the flooding task, an edge can forward a message in two directions: from node A to B or from B to A for example, depending on the time of a message arrival; due to the fact that a node must sent a message to all its neighbors except the one the message came from, regardless if any of the neighbors has previously received the same message. Portamann *et al.* presented an alternative to flooding called Rumor Mongering in which a node randomly selects B neighbors for retransmission and which defines F times a node can retransmit the same message, occasionally also an edge can be used in both directions. We will refer to as DB for this model (DB stands for double broadcasting, because a same message can travel in both ways through an edge).

In our experiments we are using the model explained in Heinzelman *et al.* [HKB], which establishes that the number of forwarded messages is equal to the number of edges in the graph. In this model each edge can be used only once to send the same message. That is, while A sends to B, B does not send to A, instead it receives some messages, processes them, and if the same message is found, cancels the intended one to A. We will call this model SB (for single broadcasting for each edge).

Given that the RNG algorithm guarantees reaching the 100% of nodes, we compare a number of forwarded messages to results from Gossip closer to 100% in order to make a more appropriate comparison. We show the performance average results for RNG *vs.* Gossip for the Barabàsi topologies with 1000 nodes and d=4, 6 and 8, when running a process 10 times.

Average degree N=500	Messages forwarded DB (100%)	Cost Gossip	Cost RNG	Cost RNG DB
4	2998	69.2%	63.82%	96.6%
6	4992	66.88%	53.96%	89.91%
8	6989	61.1%	50.88%	87.66%

Table 7.5: Gossip vs. RNG algorithm

It can be said that the RNG algorithm, as originally proposed, provides better performance than the Gossip algorithm and the savings increase as the density of the network increases. In the three analyzed cases RNG showed better performance than Gossip, when $d=4$ the difference in savings between the two algorithms was only 5.3 percentual points, but for $d=8$ it was of 10.2 percentual points. This suggests that even for larger values of d the RNG algorithm provides larger savings compared to Gossip. Finally, we include in last column the number of message retransmissions when using an edge for sending in two directions to show the negative impact of sending a message even if any neighbor has already received the same message.

Through this chapter we have presented and analyzed the results of broadcasting via the RNG when using Internet graphs generated using the Barabàsi, Waxman and Palmer models. As stated in [BKMRRSTW, MLMB, PaS] the research in the area of Internet topology is still in its early days, thus research on this is still ongoing. With that in mind, we can say according to the results presented, that the RNG algorithm can be considered as an alternative to pure flooding when broadcasting over Internet graphs. This conclusion can also be extended to the Gnutella network, since Jovanovic [J] has studied its topology concluding that it exhibits "small-world" properties and also obey the four [FFF] power-laws.

As stated before, constructing the RNG does not impose additional communication overhead in the network. The only limitation is that nodes must have information about the distance to its neighboring nodes or must count with a method to obtain this information. In the case of Internet nodes, this could be achieved by sending ping messages to calculate delays in the edges that join each node to its neighbors.

Conclusions

8.1 Conclusions

In the work presented in this thesis, we have shown that designing an algorithm for use at topology Internet level is a challenged task due to the existence of several topology models and approaches to faithfully capture the main characteristics of the Internet topology. The results obtained in our approach do not vary drastically among the topologies generated by the three models used. Savings in message retransmissions start being notorious for average degree greater than 10 when savings are 25% in average. In general, the more redundant a topology becomes the more savings can be obtained using the broadcasting algorithm over its RNG.

We showed that broadcasting over RNG may represent an alternative to pure flooding because of the savings in retransmissions and also to the Rumor Mongering protocol since our algorithm also guarantees 100% of reachability, while due to its probabilistic nature, Rumor Mongering does not. That is, RNG reduces flooding cost without loosing the reachability of nodes in the network.

Finally, indirectly we have shown the importance of choosing a broadcasting model and its impact in the results obtained. When comparing RNG with flooding using the Barabàsi model, we obtained savings above 40% for networks with node average degree not smaller than 16, while when comparing RNG with results shown in [PS] we started obtaining savings larger than 35% for an average degree of 4. Although the topology model was the same, outstanding changes were due to the broadcasting model employed. In this case, we considered the same parameters used in the experiments presented in [PS].

8.2 Future work

The implementation of the Rumor Mongering algorithm was left for future work. This is in order to compare the performance of it to the broadcasting algorithm via the RNG when using topologies generated with different parameters. It also would be interesting to conduct an analysis of the load message distribution among nodes when broadcasting via the RNG, to prove whether it allows the distribution of forwarding tasks to all nodes in the network.

References

[AB] R. Albert, A. Barabàsi; Statistical Mechanics of Complex Networks. Reviews of Modern Physics 74, 47. 2002.

[ACL] William Aiello, Fan Chung, and Linyuan Lu; A random graph model for massive graphs, Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing. 171-180. 2000.

[AJB] R. Albert, H. Jeong, A. Barabàsi; The diameter of the World Wide Web. Nature 401, 130-131. 1999.

[B] B. Bollobas; Random Graphs. Academic Press, Inc., Orlando, Florida, 1985.

[BA] A. Barabàsi, R. Albert; Emerge of scaling in random networks. Science 286, 509-12. 1999.

[BA2] A. Barabàsi, R. Albert; Topology of evolving networks: local events and universality. Physical Review Letters 85, 5234. 2000.

[BG] J. Behrens, J. J. Garcia-Luna-Aceves; Distributed, Scalable Routing Based on Link-State Vectors. Proc. ACM SIGCOMM 94, London, UK, October 1994.

[BGI] R. Bar-Yehuda, O. Goldreich, A. Itai; On the time complexity of broadcast in radio networks: an exponential gap between determinism and randomization, Proc. ACM Symp. Principles of Distributed Computing, 1987, 98-108.

[BH] L. Briesemeister and G. Hommel; Role-based multicast in highly mobile but sparsely connected ad hoc networks, Proc. MobiHOC, August 2000, 45-50.

[BKMRRSTW] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener; Graph Structure in the web. Proc. 9th International World Wide Web Conference, pp. 309--320, 2000.

[BMJHJ] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, J. Jetcheva; A performance comparison of multi-hop wireless ad hoc network routing protocols, Proc. MOBICOM, 1998, 85-97.

[BMSU] P. Bose, P. Morin, I. Stojmenovic and J. Urrutia; Routing with guaranteed delivery in ad hoc wireless networks. 3rd Int. Workshop on Discrete Algorithms and methods for Mobile Computing and Communications, Seattle, August 20, 1999, 48-55.

[CGGP] B. S. Chlebus, L. Gasienec, A. Gibbons, A. Pelc; Deterministic broadcasting in unknown radio networks. Proceedings 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2000), pp. 861-870, 2000.

[CJPP] A. Chakrabarti, M. Joshi, K. Punera, D. Pnnock; The structure of broad topics on the web. Proc. 7th International Conference on the World Wide Web. Honolulu, Hawaii, USA. 2002.

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

[CHH] S. Capkun, M. Hamdi, J.P. Hubaux; GPS-free positioning in mobile ad-hoc networks, Proc. Hawaii Int. Conf. on System Sciences, January 2001.

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

[EWB] A. Ephremides, J.E. Wieselthier and D. J. Baker; A design concept for reliable mobile radio networks with frequency hoping signaling, Proc. IEEE 75, 1987, 56-73.

[F] Folding@home. Distributed computing. University of Stanford.

[FFF] M. Faloutsos, P. Faloutsos, Christos Faloutsos; On Power-Law Relationships of the Internet Topology. Proc. SIGCOMM (1999)

[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

[GeT] M. Gerla and J.T.C. Tsai; Multicluster, mobile, multimedia radio network, Wireless networks, 1, 1995, 255-265.

[GKP] M. Gerla, T.J. Kwon and G. Pei; On demand routing in large ad hoc wireless networks with passive clustering. Proc. IEEE WCNC, September 2000.

[Go] M. T. Goodrich; Data Structures and Algorithms in Java. EUA. 1998. Willey and Sons Inc. http://www.cs.brown.edu/courses/cs016/book/

[GPS] Gnutella Protocol Specification. http://www.clip2.com/GnutellaProtocol04.pdf

[Gr] Groove Networks, Inc. Introducing Groove. http://www.groove.net/products/

[GT] R. Govindan, H. Tangmunarunkit; Heuristics for Internet Map Discovery. Proc. IEEE INFOCOM 2000.

[GTT] The University of Saskatchewan, Graph Theory Tutorial. http://www.cs.usask.ca/resources/tutorials/csconcepts/graphs/

[HA] R. Huberman, L. Adamic; Evolutionary dynamics of the World Wide Web. Technical report. Xerox Palo Alto Research Center, California, Febraury 1999.

[HHL] Hedetniemi, S. Hedetniemi, S. Liestman; A survey of Gossiping and Broadcasting in Communication Networks. Networks 18, 1988.

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

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

[JA] Jacobson, Andree; Metrics in Ad Hoc Networks. Lulea University of Technology, Computer Science and Electrical Engineering Department. Sweden May 2001.

[JCJ] C. Jin, Q. Chen, and S. Jamin; Inet: Internet Topology Generator. Technical Report Research Report CSE-TR-433-00, University of Michigan at Ann Arbor, 2000.

[JDK] Java™ 2 Platform, Standard Edition (J2SE™). http://java.sun.com/j2se/

[JDSL] Data structures library in Java. http://www.jdsl.org/

[JTAOB] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai and A.-L. Barabàsi; The large-scale organization of metabolic networks. Nature 407, 651-654, 2000.

[K] M. Kochen; The Small World. Ablex, Norwood, NJ. 1989.

[KKRRT] J. Kleinberg, S. R. Kumar, P. Ragavan, S. Rajagopalan and A. Tomkins; The web as a graph: measurements, models and methods. Proceedings of International Conference on Combinatorics Computing, 1999.

[KRRT] S. R. Kumar, P. Ragavan, S. Rajagopalan and A. Tomkins; Extracting large scale knowledge bases from the web. IEEE International Conference on Very Large Databases (VLDB). Edinburg, Scotland, 1999.

[L] G. Lauer; Address servers in hierarchical networks, Proc. ICC, 1988, 443-451.

[LG] C.R. Lin and M. Gerla; Adaptive clustering for mobile wireless networks, IEEE J. Selected Areas in Communications, 15, 7, 1997, 1265-1275.

[Lo] S. Locke; Graph Theory Tutorial. http://www.math.fau.edu/locke/graphthe.htm

[M] S. Milgram; Psychology Today 2, 60. 1967.

[MLMB] A. Medina, A. Lakhina, I. Matta, J. Byers; BRITE: Universal Topology Generation from a User's perspective, April 2001, Boston University. EUA.

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

[NaS] P. Narvaez, K. S. Siu; Fault-Tolerant Routing in the Internet without Flooding. Proc. of the 1999 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, San Juan, Puerto Rico, April 1999.

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

[NoS] P. Norton, W. Steken; Peter Norton's Guide to Java Programming. Sams publishing, 1997.

[NTCS] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen and J.-P. Sheu; The broadcast storm problem in a mobile ad hoc network, Proc. MOBICOM, Seattle, Aug. 1999, pp. 151-162.

[P] A. K. Parekh; Selecting routers in ad hoc wireless networks. ITS, 1994.

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

[PF] V. Paxson and S. Floyd; Why We Don't Know How To Simulate The Internet. In Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, January 1997.

[PL] Wei Peng, Xi-Cheng Lu; On the reduction of broadcast redundancy in mobile ad hoc networks, Proc. First Annual Workshop on Mobile and Ad Hoc Networking and Computing, Boston, USA, August 11, 2000, 129-130.

[Po] Ponte Gomez, Federico; Arquitectura para Redes Inalámbricas Ad Hoc Híbridas. Universidad Politécnica de Valencia, Facultad de Informática. Valencia, España. Noviembre 2000.

[PR] E. Pagani and G. P. Rossi; Providing reliable and fault tolerant broadcast delivery in mobile ad-hoc networks, Mobile Networks and Applications, 4, 1999, 175-192.

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

[PV] R. Pastor-Satorras and A. Vespignani; Epidemic spreading in scale-free networks. Physical Review Letters, 86(14), pp. 3200--3203, April 2, 2001.

[QVL] A. Qayyum, L. Viennot, A. Laouiti; Multipoint relaying: An efficient technique for flooding in mobile wireless networks, RR-3898, INRIA, March 2000, www.inria.fr/RRRT/RR-3898.html

[S] C. Shirky; What is p2p... and what isn't? The O'Reilly Network, November 24,2000. www.openp2p.com/pub/a/p2p/2000/11/24/shirky1whatisp2p.html

[SETI] SETI@home web site: http://setiathome.ssl.berkeley.edu

[SFLYO] M. T. Sun, W.C. Feng, T.H. Lai, K. Yamada, H. Okada; GPS based message broadcast for adaptive inter-vehicle. Proc. IEEE VTC Fall 2000, pp. 2685-2692, Vol. 6, Boston, MA, September 2000.

[SGG] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble; A measurement study of peer-to-peer file sharing systems. In Proceedings of Multimedia Computing and Networking 2002.

[SK] L. Subramanian and R. H. Katz; An architecture for building self-configurable systems, Proc. First Annual Workshop on Mobile and Ad Hoc Networking and Computing, Boston, USA, August 11, 2000, 63-73.

[SS] I. Stojmenovic, M. Seddigh; Broadcasting algorithms in wireless networks, Proc. Int. Conf. on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet SSGRR, L'Aquila, Italy, July 31-Aug. 6, 2000.

[SSF] Scalable Simulation Framework. http://www.ssfnet.org

[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.

[SSZ] I. Stojmenovic, M. Seddigh, J. Zunic; Internal node based broadcasting in wireless networks, Proc. IEEE Int. Conf. on System Sciences, January 2001.

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

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

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

[WCD] 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, Dec. 1997, 770-785.

[We] D. B. West; Introduction to Graph Theory. EUA. Prentice Hall. 1996.

[Wei] Weisstein, E: Eric Weisstein's world of Mathematics. http://mathworld.wolfram.com/

[Wi] P.M. Williams, Data Structures notes. The School of Cognitive and Computing Sciences, University of Sussex.
http://www.cogs.susx.ac.uk/local/teach/dats/notes/html/notes.html

[WJ] J. Winick, S. Jamin; Inet 3.0: Internet Topology Generator. University Generator. University of Michigan. Technical Report CSE-TR-433-00.

[WL] J. Wu and H. Li; On calculating connected dominating set for efficient routing in ad hoc wireless networks, Proc. DIAL M, Seattle, Aug. 1999.

[WNE] J.E. Wieselthier, G.D. Nguyen and A. Ephremides, On the construction of energy-efficient broadcast and multicast trees in wireless networks, IEEE INFOCOM, March 2000.

[WS] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. Nature, pages 440-442, June 1998.

[YJB] S. H. Yook, H. Jeong, and A.-L. Barabàsi. Modeling the Internet 's large-scale topology. Technical Report cond-mat/0107417, Condensed Matter Archive, http://xxx.lanl.gov, July 2001.

[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. 1997.

Appendix A

Basics on Graph Theory

One of the first people to experiment with graph theory was a man by the name of Leonhard Euler (1707-1783). He attempted to solve the problem of crossing seven bridges onto an island without using any of them more than once [GTT].

From that point on, graphs study have had applications in a host of different domains, including mapping (in geographic information systems), transportation (in road and flight networks), electrical engineering (in circuits), and computer networking (in the connections of the Internet). Because applications for graphs are so widespread and diverse, people have developed a great deal of terminology to describe different components and properties of graphs [Go].

A.1 Some terminology

Graph: a graph consists of a nonempty set of points or vertices, and a set of edges that link together the vertices. A *simple graph* can be thought of as a triple $G=(V,E,I)$, where V and E are disjoint finite sets and I is an incidence relation such that every element of E is incident with exactly two distinct elements of V and no two elements of E are incident to the same pair of elements of V. We call V the *vertex set* and E the *edge set* of G [Lo].

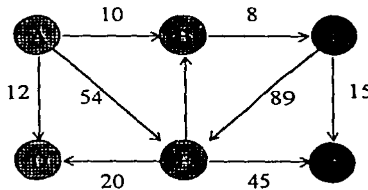


Figure A.1: A directed weighted graph with 6 vertices and 9 edges

Directed graph: If all the edges in a graph are directed, then we say the graph is an *directed graph* also called *digraph*. An edge (u,v), where $(u,v) \in V$, is said to be *directed* from u to v if the pair (u,v) is ordered, with u preceding v.

Undirected graph: is a graph whose edges are all undirected. An edge (u,v) is said to be undirected if the pair (u,v) is not ordered. A graph that has both directed and undirected edges is often called a mixed graph.

End vertices: the two vertices joined by an edge are called the *end vertices* of the edge. The end vertices of an edge are also known as the *endpoints* of that edge. If an edge is directed, its first endpoint is its *origin* and the other is the *destination* of the edge.

Adjacent vertices: two vertices are said to be *adjacent* if they are endpoints of the same edge.

Incident edge: an edge is said to be *incident* on a vertex if the vertex is one of the edge's endpoints.

Outgoing edge: an *outgoing edge* of a vertex is a directed edge whose origin is that vertex.

Incoming edge: an *incoming edge* of a vertex is a directed edge whose destination is that vertex.

Degree of a vertex: denoted *deg(v)*, is the number of incident edges of v, which is the same as the number of the adjacent vertices of v. The *in-degree* and *out-degree* of a vertex v are the number of the incoming and outgoing edges of v, and are denoted indeg(v) and outdeg(v), respectively [Go].

Neighbors of a vertex: the set of *neighbors*, *N(v)*, of a vertex v is the set of vertices which are adjacent to v. The degree of a vertex is also the cardinality of its neighbor set.

Walk: A *walk* or *path* is an alternating sequence of vertices and edges, with each edge being incident to the vertices immediately preceding and succeeding it in the sequence.

Trail: A *trail* is a walk with no repeated edges.

Simple path: A *simple path* is a walk with no repeated vertices. A walk is *closed* if the initial vertex is also the terminal vertex.

Cycle: is a closed trail with at least one edge and with no repeated vertices except that the initial vertex is the terminal vertex [Lo].

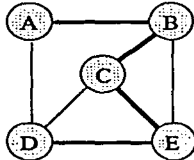


Figure A.2: A simple path ABCED

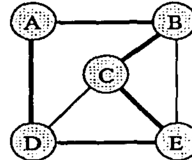


Figure A.3: A cycle ABCEDA

Lenght of a walk: the *length* of a walk is the number of edges in the sequence defining the walk. Thus, the length of a path or cycle is also the number of edges in the path or cycle.

Distance: If *u* and *v* are vertices, the *distance* from *u* to *v*, written *d(u,v)*, is the minimum length of any path from *u* to *v*. In an undirected graph, this is obviously a metric.

Eccentricity: The *eccentricity*, *e(u)*, of the vertex *u* is the maximum value of *d(u,v)*, where *v* is allowed to range over all of the vertices of the graph. The *radius* of the graph *G*, *rad(G)*, is the minimum value of *e(u)*, for any vertex *u*, and the *diameter*, *diam(G)*, is the corresponding maximum value. It should be obvious that $diam(G) <= 2rad(G)$ [Lo].

Clique: aA graph is *complete*, or a *clique*, if every pair of vertices are adjacent. We write K_m for the complete graph on *m* vertices [Lo].

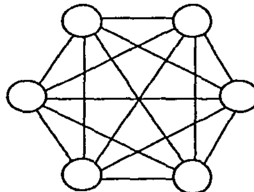


Figure A.4: A clique for a given graph containing 6 nodes.

Connected Graph: a non-null graph is *connected* if, for every pair of vertices, there is a walk whose ends are the given vertices.

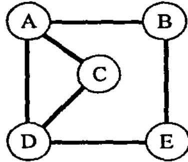


Figure A.5: A connected graph

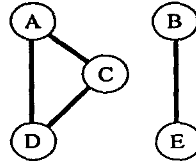


Figure A.6: A non connected graph

Tree: a tree is a graph G completely connected and containing no cycles. A Tree with *n* nodes has *n-1* edges. Conversely, a connected free loop graph with *n* nodes and *n-1* edges is a tree. In a tree, nodes are known as forks, edges as branches. Final edges in the nodes at their ends are called leaves.

Minimal Spanning Tree: a Minimal Spanning Tree (MST) of a graph G is a three of G containing all vertices of G and having the smallest possible total distance.

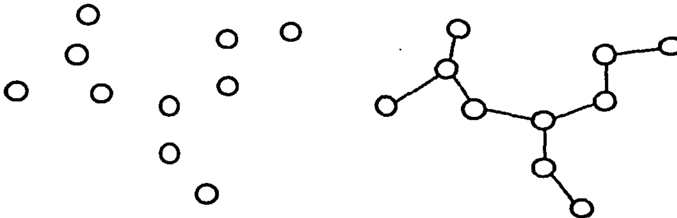


Figure A.8: A set of point in the plane (left) The Minimal Spanning Tree of that set of points (right).

A.2 Graph Representations

A.2.1 Adjacency matrix [Wi]

Some types of graph are defined by knowing, for each pair of nodes, whether or not there is an edge connecting them. In the case of directed graphs, it matters which way round we consider the pair of nodes. There may be an edge in one direction but not in the other.

Assuming a correspondence between the *n* nodes of a graph and the integers $0,\dots,n-1$, the graph can be defined by an $n \times n$ matrix of Boolean values. The (*i,j*)'th

entry of the matrix is true if there is an edge from node i to node j and false otherwise. This is referred to as an adjacency matrix representation.

For a weighted graph it will be necessary to provide storage for the weights, either in a parallel $n \times n$ numerical matrix, or by having a matrix of pairs of booleans and numbers or, possibly, by collapsing the representation into a single matrix by using a device such as indicating non-existence of paths by "infinite" weights.

The matrix representation can have advantages in terms of simplicity and time efficiency, but it also suffers from disadvantages. First, it can be extravagant in terms of storage space. The amount of storage required is $O(n^2)$. But for sparse graphs, much less is really needed. It is only necessary to store those edges and weights which exist. For example, in a graph with around 3000 nodes, almost 1,000,000 units of storage will be needed in the adjacency matrix. But if there are only about 6 or 7 edges originating from a given node on average, only 20,000 units of storage are really needed.

Secondly the adjacency matrix representation, in its simple form, is unable to represent graphs in which there may be more than one edge between a given pair of nodes.

A.2.2 Adjacency lists [Wi]

An alternative representation uses just the collection of edges in the graph. It is convenient to subdivide these into classes of edges originating from given nodes, and at this point we fix specifically on directed graphs.

Let us agree on some terminology. Consider the edge:

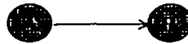


Figure A.9: Directed edge from A to B

which is directed from node A to node B. We shall say that node a is the tail of the edge and that node b is the head of the edge. The terminology comes from thinking of the head and tail of the arrow. We shall also say that node b is adjacent to node a when there is an edge having b as its head and a as its tail.

We can now organize the edges in a directed graph by listing, for each node in turn, the nodes which are adjacent to it. For the graph of *Figure A.1*, this gives

$A \rightarrow \{B, E\}$
$B \rightarrow \{C\}$
$C \rightarrow \{E, F\}$
$D \rightarrow \{A\}$
$E \rightarrow \{B, D, F\}$
$F \rightarrow \{\ \}$

Note the empty list for F. The idea is to represent the graph as a collection of nodes, each of which carries with it its list of adjacent nodes.

Note in passing that the dual representation

$A \leftarrow \{D\}$
$B \leftarrow \{A, E\}$
$C \leftarrow \{B\}$
$D \leftarrow \{E\}$
$E \leftarrow \{A, C\}$
$F \leftarrow \{C, E\}$

where we list the nodes to which a given node is adjacent, would serve equally well. But it is more usual and more intuitive, especially when considering shortest path algorithms, to think in a "forwards" direction. For some purposes, however, it may be convenient to make simultaneous use of both representations.

Appendix B

ADT for simulator classes

Class ConnectivityTester

`Graphs.ConnectivityTester`

public class **ConnectivityTester**

ConnectivityTester allows checking if a given graph is completely connected. This class extends Depth-First Search algorithm implemented on JDSL. `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.
This algorithm runs in O(V+E) time where V is the number of vertices in the graph, and E is the number of edges. It also uses O(V+E) additional space to store data about the vertices and edges during the computation. To this end, it is expected that the `Edges` and `Vertices` implement the `Decorable` interface.

Method Detail

isConnected

`public boolean isConnected(Graphs.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.

Class EdgeInfo

```
java.lang.Object
  |
  +--Graphs.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 and lenght

Field Detail

name

`protected java.lang.String name`

lenght

`protected double lenght`

xyEdge

`protected boolean xyEdge`

markForDeleting

`protected boolean markForDeleting`

Constructor Detail

0.1.1.1.1 EdgeInfo

```
public EdgeInfo(java.lang.String nameEdg,
                double lenghtEdg)
```

Stores name and lengh edge
Parameters:
`nameEdg` - string containing edge name
`lenghtEdg` - edge lenght in double format

EdgeInfo

`public EdgeInfo()`

Standard constructor, used when name and lenght 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:

`leghtEdge` - lengh 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 Retuns true if edge needs to be deleted

Class graphTools

```
java.lang.Object
  |
  +--Graphs.graphTools
```

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

Contains useful methods for working with graphs.

Method Detail

lenghtEdge

```
public static double lenghtEdge(Graphs.Vertex a,
                                Graphs.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(Graphs.Vertex vContent)
```

Gets xy coordiantes from node name

Parameters:

`vContent` - node from which to obtain coordinates

Returns:

array cointaining [x,y] coordiantes

goToVertex

```
public static Graphs.VertexIterator
goToVertex(Graphs.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 Graphs.Graph copyNodesFrom(Graphs.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 Graphs.Graph copyGraph(Graphs.Graph
                                     inputGraph)
```

Copies a whole Graph

Parameters:

`inputGraph` - graph to be copied

Returns:

a copy of inputGraph

getNodeWhichName

```
public static Graphs.Vertex
getNodeWhichName(java.lang.String name,
                 Graphs.VertexIterator vtxIter)
```

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

Parameters:

`name` - name of the node to be searched

`vtxIter` - `VertexIterator` into which it looks for the node

Returns:

pointer to node found

compare

```
public static void compare(Graphs.Graph gra1,
                           Graphs.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 conpared

`output` - name of file to write out comparison information

countVertices

```
public static int countVertices(Graphs.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(Graphs.Graph graph)
```

Shows on screen messages forwarded by every node in a Graph

Parameters:

`graph` - graph to be analysed

showSource

```
public static void showSource(Graphs.Graph graph)
```

Shows on screen source node of every node in the Graph

Parameters:

`graph` - graph to be analysed

cloneNode

```
public static Graphs.Vertex cloneNode(Graphs.Vertex
                                       paternNode,
                                       Graphs.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 daughtherGraph

`daughterGrph` - Graph in which it looks for pattern node

Returns:

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

Class InternetMap

```
java.lang.Object
   |
   +--Graphs.InternetMap
```

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

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.

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 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 Graphs.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.

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.

briteXY

```
public void briteXY()
             throws java.io.IOException
```

Reads file generated by BRITE

getNumberNodes

```
public void getNumberNodes()
                    throws java.io.IOException
```

Allows knowing number of nodes present in input file, value is stored on internal instance variable

isXYNode

```
public boolean isXYNode(Graphs.Vertex a)
```

Allows to know if in a node it is being used x-y coordinates or delay as metric

Class RNG

```
java.lang.Object
  |
  +--Graphs.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; The RNG of a finite planar set. 1980

Method Detail

getRNG

`public Graphs.Graph getRNG(Graphs.Graph inputGraph)`

Obtains RNG from a given graph it works only when input graph is a RUG due to intrinsec 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 Graphs.Graph applyRNG(Graphs.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 Graphs.Graph applyRNG2(Graphs.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

Class RUG

```
java.lang.Object
  |
  +--Graphs.RUG
```

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

RUG creates a Random Unit Graph in a plane of n x n. Nodes are randomly selected and asigned a pair of x-y coordinates. For generating a connected graph, once that RUG algorithm has been applyied function `isConnected` check if generated graph is copmpletely connected, if not, repeats generation.

Method Detail

execute

```
public Graphs.Graph execute(int mxm,
                            int vtxIterIJ,
                            int degree)
```

Obtains a RUG with average `degree` from a set of `mxm` points. `execute` only contains a loop that checks for connectivity. Actually, `makeGraph` implements RUG algorithm.

Parameters:

`mxm` - plane dimmension

`vtxIterIJ` - vertex iterator for covering entire plane

`degree` - the average degree for RUG

Returns:

a Graph containing generated RUG

makeGraph

```
public Graphs.Graph makeGraph(int mxm,
                              int vtxIterIJ,
                              int degree)
```

Contains main algorithm for RUG, only selects, sort edges length and make connections between node, thus connectivity of entire graph needs to be tested

Parameters:

`mxm` - plane dimmension

`vtxIterIJ` - vertex iterator for covering entire plane

`degree` - the average degree for RUG

Returns:

a Graph containing generated RUG

Class VertexInfo

```
java.lang.Object
  |
  +--Graphs.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

Method Detail

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(Graphs.Vertex vtx)`

Allows establishing source node n forwarding process

Parameters:

`vtx` - node acting as source node in current broadcasting process

incFwdMessages

`public void incFwdMessages(Graphs.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 Graphs.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

Appendix C

Source code for Graphs Package

ConnectivityTester.java
Created with JBuilder

```
package Graphs;

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. isConnected runs DFS algorithm starting at
 * 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.
 *

 * This algorithm runs in O(V+E) time where V is the number of vertices
 * in the graph, and E is the number of edges. It also uses O(V+E)
 * additional space to store data about the vertices and edges during the
 * computation. To this end, it is expected that the Edges
 * and Vertices implement the Decorable
 * interface.
 *
 *
 */

public class ConnectivityTester extends DFS{
  /**
   * Checks for connectivity of a Graph
   * @param g      graph to check connectivity
   * @return       true if graph is completely connected
   *               false 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;
```

```
    }
}


EdgeInfo.java
Created with JBuilder

package Graphs;

import java.io.*;
import java.lang.*;

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

public class EdgeInfo {
   protected String name;
   protected double lenght;
   protected boolean xyEdge=false;
   protected boolean markForDeleting;

  /**
   * Stores name and lengh 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 and lenght have not been
   * calculated
   */
   public EdgeInfo(){
      lenght=0;
      markForDeleting = false;
   }
   /**
    * Attaches edge name
    * @param nameEdge     name of current edge
    */
   public void name(String nameEdge){
      name=nameEdge;
   }

   /**
    * Attaches lenght of current edge
    * @param leghtEdge    lengh of edge in double format
    */
   public void lenght(double lenghtEdge){
      lenght=lenghtEdge;
   }
```

```
    /**
     * Marks edge to be deleted
     * @param mark         contains true if edge must be
     *                     deleted otherwise contains false
     */
    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
    * Retuns true if edge needs to be deleted
    */
    public boolean isMarked(){
        return markForDeleting;
    }

}
```

```
graphTools.java
Created with JBuilder

package Graphs;

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 cointaining [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; //xVal;
    }

   /**
    * Allows moving back or forward through a vertexIterator,
    * thus going to a specific node
    * @param verIt        VertexIterator containing all vertices
    *                     in the Graph
    * @param vertex       number ID of node to move to
    * @return             a VertexIterator 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
    */
    //allows making a copy of nodes from a given 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, vtxIter);
          Vertex endNode = getNodeWhichName(nameEndNode, vtxIter);
          outputGraph.insertEdge(starNode, endNode, edgInfo);
       }
       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  VertexIterator into which it looks for
     *                 the node
     * @return         pointer to node found
     */
    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;
    }

    /**
     * Compares number of edges and degree of two different graphs
     * @param gra1       graph1 to be compared
     * @param gra2       graph2 to be conpared
     * @param output     name of file to write out comparison information
     *
     */
    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 = 100-gra2.numEdges()*100/gra1.numEdges();
        System.out.println("Edges\tAv. Degree\t\tEdges (RNG)\tAv. Degree
(RNG)\t%Savings" );
        output.write("Edges\tAv. Degree\t\tEdges (RNG)\tAv. Degree
(RNG)\t%Savings\n" );
        System.out.println(gra1.numEdges()+"\t"+formater.format(avDeg1)+
                           "\t\t"+gra2.numEdges()+"\t\t"+
                           formater.format(avDeg2)+"\t\t"+savings);
        output.write(gra1.numEdges()+"\t"+formater.format(avDeg1)+"\t\t"+
                           gra2.numEdges()+"\t\t"+formater.format(avDeg2)+
                           "\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                 codeNode 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;
   }
}
```

InternetMap.java
Created with JBuilder

```
package Graphs;

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.
 *

 * 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.
 *
 */
public class InternetMap {
   Graph internetGraph;
```

```
    int numNodes;
    String fileName;
    String fileType;
    BufferedReader inputFile;

   /**
    * Allows having access to Graph stored on instance variable of
    * InternetMap
    * @return    Graph containing Internet Map loaded by means of
    * InternetMap 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 Graph with distance...");
        getNumberNodes();
        insertNodesXY();
        insertEdges();
        inputFile.close();

        EdgeIterator eIt = 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 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")) {

          lineFromFile=inputFile.readLine();
           numNodes = x.parseInt(lineFromFile);

       } 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());
            exit = true;
         }
      }
   }



   private void insertNodesXY(){
      int m = 100;                    //space por genrating 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;

      VertexInfo vertex;
      String lineFromFile;
      StringTokenizer st;
```

```
        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();
//                     System.out.println(nodeID+ " "+ numNodes);
                    x=X.intValue();
                    y=Y.intValue();
                    vertex = new VertexInfo();
                    vertex.name(""+nodeID);
                    vertex.xCoor(x);
                    vertex.yCoor(y);
                    //inserts vertex selected into gra
                    internetGraph.insertVertex(vertex);
                 }
                 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 starNode;
        String endNode;

        if (fileType.equalsIgnoreCase("nm")){
           while ((lineFromFile= inputFile.readLine()) != null){
              if (lineFromFile.indexOf("EDGES")!= -1){
                 while ((lineFromFile= inputFile.readLine()) != null){
                    st = new StringTokenizer(lineFromFile);
                    starNode = st.nextToken();
                    endNode = st.nextToken();
                    Vertex star=getNode(starNode);
                    Vertex end=getNode(endNode);
                    createEdge(star, end);
//                  VertexInfo vinf = (VertexInfo)star.element();
//                  System.out.println(vinf.getName());
                 }
              }
           }
        } else if (fileType.equalsIgnoreCase("rec")){
            int i =0;
           while ((lineFromFile= inputFile.readLine()) != null){
              st = new StringTokenizer(lineFromFile);
```

```
            starNode = st.nextToken();
            endNode = st.nextToken();
            Vertex star=getNode(starNode);
            Vertex end=getNode(endNode);
             i++;
            createEdge(star, end);
//                VertexInfo vinf = (VertexInfo)star.element();
//                System.out.println(vinf.getName());
          }
          System.out.println(i);
      } 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 ();
                   starNode = st.nextToken();
                   endNode = st.nextToken();
                   Vertex star=getNode(starNode);
                   Vertex end=getNode(endNode);
                   createEdge(star, end);
//                VertexInfo vinf = (VertexInfo)star.element();
//                System.out.println(vinf.getName());
                }
             }
          }
       }
    }

    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);
        }
    }


    /**
     * Allows to know if in a node it is being used x-y coordinates or
     * delay as metric
     *
     */
    public 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);
//        System.out.println(src.getAbsolutePath());
        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);
    }
}

/**
 * There are 2 cases for this problem
 * 1) generate Graph using x,y coordinates for getting lenght parameter
 * 2) generate Graph using delay as lenght parameter
 *
 * Using X,Y
 * ================
 * GraphCreation(file_name)
 *    initialization
 *    insertNodes
 *    insertEdges
 *
 * initialization
 *    OPEN file_name
```

```
 *    graph<-null
 *    numberNodes<-0
 *
 * insertNodes
 *    GET numberNodes
 *    FOR_EACH node
 *       Randomly assign X and Y
 *       insert node IN graph
 *
 * insertEdges
 *    while !EOF
 *       GET starNode
 *       GET endNode
 *       CALCULATE lenght
 *       CREATE edge
 *       INSERT edge IN graph
 *    CLOSE file
 *
 *
 * Using delay
 * ================
 * GraphCreation(file_name)
 *    initialization
 *    insertNodes
 *    insertEdges
 * initialization
 *    OPEN file_name
 *    graph<-null
 *    numberNodes<-0
 *
 * insertNodes
 *    GET numberNodes
 *    FOR_EACH node
 *       insert node IN graph
 *
 * insertEdges
 *    while !EOF
 *       GET starNode
 *       GET endNode
 *       Randomly assign delay
 *       CREATE edge
 *       INSERT edge IN graph
 *    CLOSE file
 *
 */
```

```
RNG.java
Created with JBuilder

package Graphs;

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.
 *


 * 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;
 * The RNG of a finite planar set. 1980
 *
 * @author Oscar Escalante
 */

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;

        long startTime = System.currentTimeMillis();

        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++;
//          System.out.print("\r"+ 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);
           }
        }

        long endTime = System.currentTimeMillis();
        long totalTime = endTime - startTime;
        System.out.println("total time: " +totalTime);
        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++;
//           System.out.print("\r"+ 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 but 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;
    }
}

/**
 * Algorithm for finding RNG
 * ===========================
 *
 * FOR_EACH pair of vertex (vi, vj)
 *     COMPUTE distance(vi, vj) i,j=1,..,n i!=j
 *
 * FOR_EACH pair of vertex (vi, vj)
 *     COMPUTE dkmax=max{d(vk, vi), d(vk, vj)} k=1,...n, k!=i, k!=j.
```

```
 *
 * FOR_EACH pair of vertex (vi, vj)
 *     IF_NOT dkmax *        INSERT edge between vi and vj
 *
 * Alternative algorithm for RNG
 * ================================
 *
 * FOR_EACH pair(i, j) IN graph
 *     CALCULATE distance(i, j)
 *     FOR_EACH vertex(k) IN graph
 *        CALCULATE distance(k, i), distance(k, j)
 *        dkmax <--- max[distance(k, i), distance(k, j)]
 *     IF_NOT dkmax *        INSERT edge between i, j
 *
 * Algorithm for finding RNG over a given graph
 *
 * FOR_EACH node i
 *     FOR_EACH i.neighbor
 *        neighbors_list<-- i.neighbor
 *     FOR_EACH node IN neighbors_list
 *        FOR_EACH node.neighbor
 *           IF node.neighbor IS IN neighbors_list
 *              k<--node.neighbor
 *              edgeij<--distance(i, j)
 *              edgeik<--distance(i, k)
 *              edgejk<--distance(j, k)
 *              dmax=dmax(edgeik, edgejk)
 *              if dmax *              DELETE longer edgeij
*
 */
```

RUG.java
Created with JBuilder

```
package Graphs;

import java.io.*;
import java.lang.*;
import java.util.Random;
import java.util.Vector;
import java.util.Collections;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;
import jdsl.graph.algo.*;
import jdsl.core.api.*;
import jdsl.core.ref.*;

/**
 * RUG creates a Random Unit Graph in a plane of n x n.
 * Nodes are randomly selected and asigned a pair of x-y coordinates.
 * For generating a connected graph, once that RUG algorithm has been
 * applyied function isConnected check if generated graph is
 * copmpletely connected, if not, repeats generation.
 *
 */

public class RUG {
```


TESIS CON
FALLA DE ORIGEN

```
/**
 * Obtains a RUG with average degree from a set of
 * mxm points. execute only contains a loop
 * that checks for connectivity. Actually, makeGraph
 * implements RUG algorithm.
 *
 *
 * @param mxm            plane dimmension
 * @param vtxIterIJ      vertex iterator for covering entire plane
 * @param degree         the average degree for RUG
 * @return               a Graph containing generated RUG
 *
 */
public Graph execute (int mxm, int vtxIterIJ, int degree){
    Graph outGraph = new IncidenceListGraph();

    //creates a RUG
    outGraph=makeGraph(mxm, vtxIterIJ, degree);

    //Tests for connectivity
    ConnectivityTester connTester = new ConnectivityTester();
    while (!connTester.isConnected(outGraph)){
       outGraph=makeGraph(mxm, vtxIterIJ, degree);
    }

    //returns graph only if it is completely connected
    return outGraph;
}

/**
 * Contains main algorithm for RUG, only selects, sort edges lenght
 * and make connections between node, thus connectivity of entire
 * graph needs to be tested
 *
 * @param mxm            plane dimmension
 * @param vtxIterIJ      vertex iterator for covering entire plane
 * @param degree         the average degree for RUG
 * @return               a Graph containing generated RUG
 */
public Graph makeGraph(int mxm, int vtxIterIJ, int degree){
    Double Ratio;
    Vector vLenghts;
    VertexIterator vtxIter;
    Vertex vInicial, vFinal;
    int x, y, i;

    //rnd is the random index for selecting nodes
    Random rnd = new Random();
    Graph gra = new IncidenceListGraph();
    VertexInfo vtxInfo;
    EdgeInfo edgInfo;

    //this cycle for random selecting n nodes
    for(i=0; i<vtxIterIJ; i++){
       vtxInfo = new VertexInfo();
       x=rnd.nextInt(mxm);
       y=rnd.nextInt(mxm);
       vtxInfo.xCoor(x);
```

```
            vtxInfo.yCoor(y);
            vtxInfo.name(""+i);
            gra.insertVertex(vtxInfo);     //inserts vertex selected into
gra
        }

        //this cycle for calculating lenghts of all potential edges
        vtxIter = gra.vertices();
        vLenghts=new Vector();

        //covers all potential edges
        for (i=0; i<vtxIterIJ-1; i++){

            //forwards vtxIter to desired index i
            vtxIter=graphTools.goToVertex(vtxIter,i);
            vInicial=vtxIter.nextVertex();     //vInicial equals to index i
            while (vtxIter.hasNext()){
                vFinal=vtxIter.nextVertex();

                //calculates lenght from vInicial to vFinal
                Double lenEdg=new Double(graphTools.lenghtEdge(vInicial,
vFinal));
                vLenghts.addElement(lenEdg);    //stores lenght edge in
vector
            }
        }
        Collections.sort(vLenghts);             //sorts lenghts vector

        //takes ratio as the lenght of degree*vtxIterIJ/2-th edge
        Ratio =(Double)vLenghts.elementAt((int)degree*vtxIterIJ/2);

        //cycle connecting nodes if distance is equal or less than
        //calculated Ratio
        vtxIter.reset();
        for (i=0; i<vtxIterIJ-1; i++){
            vtxIter=graphTools.goToVertex(vtxIter,i);
            vInicial=vtxIter.nextVertex();
            while (vtxIter.hasNext()){
                vFinal=vtxIter.nextVertex();
                Double lenEdg=new Double(graphTools.lenghtEdge(vInicial,
vFinal));
                if (lenEdg.compareTo(Ratio)<=0){
                    edgInfo = new EdgeInfo();
                    edgInfo.lenght(lenEdg.doubleValue());
                    edgInfo.name(((VertexInfo)vInicial.element()).getName()+
                    "-"+((VertexInfo)vFinal.element()).getName());
                    gra.insertEdge(vInicial, vFinal, edgInfo);
                }
            }
        }
        return gra;
    }
}

/**
 * RUG(m, n, d)
 * CHOOSE n points AT RANDOM FROM [0, m] x [0, m]
 * FOR_EACH pair of points IN Graph
```

```
 *    edge.lenght <-- distance(pair of points)
 *    Add edge.lenght in lenght list
 * SORT lenght list
 * Radius R= nd/ 2- th edge in sorted lenght list
 * IF graph disconnected
 *    RUG(m, n, d)
 *
 */
```

VertexInfo.java
Created with JBuilder

```
package Graphs;

import java.io.*;
import java.lang.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;

/**
 * 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
 *
 *
 */

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;

  /**
   * 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 n 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;
    }
}
```