UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ingeniería

"Dinámica De Agentes De Cómputo Móvil"

TESIS

Que para obtener el título de

Ingeniero en Computación

Presenta

Francisco López Ramírez

Director de Tesis:
Dr. Octavio Miramontes Vidal

Agosto 2010

A mi madre, por haberme enseñado que el amor es paciente e infinito.

A mi padre, en profundo agradecimiento porque cada año que pasamos

sin él más entiendo cuánto nos quiso y cuánto hizo por nosotros.

AGRADECIMIENTOS

Quisiera agradecer de manera especial al Dr. Octavio Miramontes por haberme brindado la oportunidad de involucrarme en un proyecto tan interesante y estimulante como este, por su disposición a resolver cualquier duda o inquietud y por la gran cordialidad que me mostró cada vez que toqué a su puerta. Al Ing. Javer Martínez Mendoza por haber sido mi mentor en la computación y por haberme apoyado tanto en mi desarrollo profesional. A la Dra. Tetyana Baydyk cuyas minuciosas revisiones enriquecieron el presente trabajo. A Nicolás Palma por ayudarme a desentrañar dudas referentes a sistemas complejos. A Carlos López Nataren y Alex Juárez del equipo de cómputo del IFUNAM por sus recomendaciones y consejos. A Valeria Hansberg por sus oportunas correcciones. Al programa DGAPA-PAPIIT por apoyarme con una beca durante la primera parte de este proyecto.

A Timothy Heyman, Ramse Gutiérrez y Arturo Garmendia de Heyman y Asociados por permitirme concluir este trabajo a pesar de que significó muchas horas de ausencia en la oficina.

A la familia muégano (Arcelia, Beatriz, Enrique y Manuel) que con la misma devoción con la que el ratón Macías alguna vez dijo "todo se lo debo a mi Manager", yo les digo "todo se lo debo a ustedes", gracias por confiar en mí y por apoyarme incondicionalmente. A mi queridísimo hermano Rafo por enseñarme tantas cosas y ser tan buena onda. A mis amigos que son familia: Al Negro (gracias por 15 años de matrimonio), al general de la treceaceava legión Ignacio López, a Héctor, Alejandro, Sandino, Adriana, Fernando, Olmo, Moye, Adolfo, Javo, Pamela, Tulio y Angel. Gracias por hacerme la vida ligera. A O. girl por inflarme el corazón.

A los malos maestros de la facultad de Ingeniería por no destruir por completo mi creatividad, a los buenos por compartir su pasión y conocimientos. Gracias a la UNAM por ser un espacio abierto de conocimiento y creatividad.

Índice general

Capítulo 1

Introducción

1.1. Redes con tolerancia al retraso

En el campo de la arquitectura de redes de cómputo, las redes con tolerancia al retraso, DTN por sus siglas en inglés ("*Delay-Tolerant network*"), se refieren al estudio de las complicaciones que surgen al implementar redes informáticas en sitios donde la conectividad de los elementos que forman la red puede verse interrumpida. Ejemplos de este tipo de redes son aquellas que operan en entornos móviles u hostiles en donde la interrupción de la conectividad puede ser causada por múltiples factores incluyendo un rango limitado de señal inalámbrica, cambios en la posición de los nodos, fuentes de energía insuficiente, daños en los canales de comunicación o ruido.

Aunque los primeros estudios sobre el ruteo de información entre dos computadoras sin posición fija comenzaron en los años setentas, la consolidación del estudio de las redes móviles ocurre hasta principios de los años noventas con el desarrollo de las tecnologías de comunicación inalámbrica. Las redes ad-hoc móviles y la redes ad-hoc vehiculares fueron los primeros modelos en aparecer. Su principal característica radica en establecer tablas dinámicas de ruteo de información de acuerdo a la posición y disponibilidad de los nodos de una red. Paralelamente al desarrollo de estas redes, la agencia estadounidense de investigación en proyectos de defensa (DARPA) inicia el proyecto "*Interplanetary Internet*" (IPN) cuyo objetivo era diseñar un esquema de red que pudiera soportar los grandes retrasos y la corrupción de paquetes de información en las comunicaciones espaciales. En 2002, Kevin Fall comienza a adaptar muchos de los conceptos de IPN en redes terrestres y en un artículo publicado en la SEGCOMM en 2003 [Fall 03] acuñe el término genérico "*Delay-Tolerant network*" para referirse a cualquier arquitectura de red capaz de funcionar eficientemente a pesar de los retrasos en el envío y recepción de información causados por un ambiente hostil. En los años que siguieron, el interés por combinar los trabajos de redes ad-hoc móviles y DTN aumentó considerablemente [Basagni 04].

Aplicaciones de redes con tolerancia al retraso En la actualidad, es posible encontrar un gran número aplicaciones de redes con tolerancia al retraso en contextos muy variados, intentaremos mencionar las más representativas:

MESH_Networking En esta arquitectura de red, cada nodo actúa como un ruteador independiente, de manera que, si la trayectoria de comunicación se ve interrumpida o algún nodo de la red pierde conectividad, cada nodo es capaz de establecer una nueva ruta de comunicación con los nodos disponibles [Hossain 07]. Como resultado de esto, se dice que las redes MESH tienen la capacidad de "curarse a sí mismas", lo que significa que son capaces de reconfigurar las trayectorias de comunicación en caso de que un nodo falle (Figura 1.1).

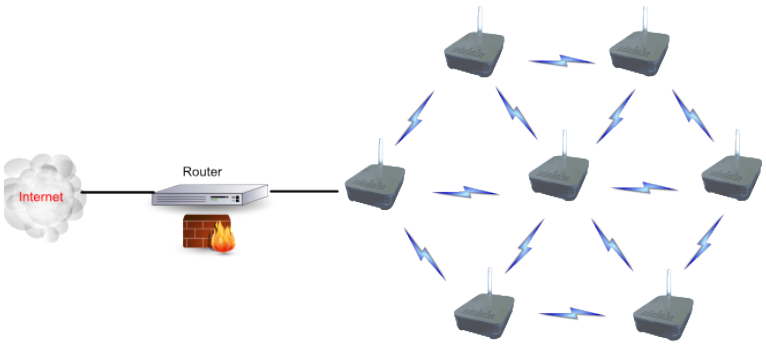

Figura 1.1: Topología de una red inalámbrica MESH.

Como ejemplo de una implementación de redes MESH podemos mencionar las computadoras OLPC XO-1 (figura 1.2), que forman parte del proyecto OLPC ("*One Laptop Per Child*"). Estas computadoras, diseñadas especialmente para niños que asisten a escuelas en entornos rurales o zonas marginadas, utilizan una red MESH para crear una infraestructura de comunicación robusta y económica. Las conexiones instantáneas entre estas computadoras permiten ampliar el área de cobertura de Internet sin depender de recursos externos ya que, al contar con un solo nodo con acceso a Internet, este es capaz de compartir la conexión con el resto de la red. En Abril de 2010, la Organización de las Naciones Unidas entregó a 2,100 niños palestinos de la franja de Gaza una computadora XO-1 con el fin de establecer una red informática educativa que les permita tener acceso a Internet y a otras tecnologías de información [Sayer 10].

Figura 1.2: Computadora OLPC XO-1 [OlpcNews 10]

MANETS Las redes MANETS ("*mobile ad hoc network*") están formadas por nodos móviles que, a través de tablas dinámicas de ruteo, establecen una trayectoria óptima de acuerdo a la posición de los nodos y a la conectividad de la red [Basagni 04]. En el caso de comunicaciones vehiculares, las MANETS han sido utilizadas para mantener una constante comunicación entre vehículos circundantes [Eichler 06] (Figura 1.3), de esta manera, al conocer en todo momento la posición y la velocidad de los vehículos vecinos, es posible reducir el tráfico y prevenir accidentes. Como otro ejemplo notable, las redes iMANETS ("*Internet based mobile ad hoc networks*") ligan dispositivos móviles con gateways fijos de Internet [Corson 99]. De esta manera, se pueden crear grandes redes inalámbricas con acceso a Internet alimentadas por el flujo de estos nodos móviles. Podemos pensar incluso en una red cuyos nodos móviles sean teléfonos celulares portados por humanos, en tal caso, es posible configurar una red inalámbrica de gran cobertura con el simple flujo de personas en un área determinada.

Figura 1.3: Red MANET Vehicular [Popescu 10].

WiMAX "*Worldwide interoperability for microwave access*" es una interesante red no convencional que ha comenzado a tener una gran difusión. Está basada en el protocolo de comunicaciones inalámbricas IEEE 802.16m y es capaz de alcanzar un ancho de banda de 1 Gbit/s. Aunque la arquitectura básica de WiMAX es muy similar al de las redes inalámbricas estándar (WiFi), esta nueva tecnología incorpora protocolos modernos que multiplican la cobertura y eficiencia de las comunicaciones [Fazel 08]. Las tecnologías inalámbricas por microondas están basadas en puntos de acceso que envían y reciben información a dispositivos que se encuentran a un radio de distancia que podríamos considerar pequeño (32m). Una vez que la información llega al punto de acceso, esta viaja por Internet de manera convencional [Choi 08]. Así, la transmisión inalámbrica de datos sólo ocurre entre los puntos de acceso y los dispositivos receptores, el resto se efectúa de manera alámbrica por medio de ruteadores y proxys que pueden estar conectados por fibra óptica, cables ethernet o coaxiales. WiMAX utiliza torres de transmisión con áreas de cobertura mucho mayores que las redes WiFi, estas torres conectadas directamente a Internet con un cable con ancho de banda alto, son capaces de comunicarse con otras torres de transmisión conocidas como "*backhauls*" (Figura 1.4). Al sumar la cobertura de cada torre (aproximadamente $8km^2$), una red WiMAX, compuesta de varios *backhauls*, es capaz de cubrir amplísimas zonas rurales sin necesidad de cables.

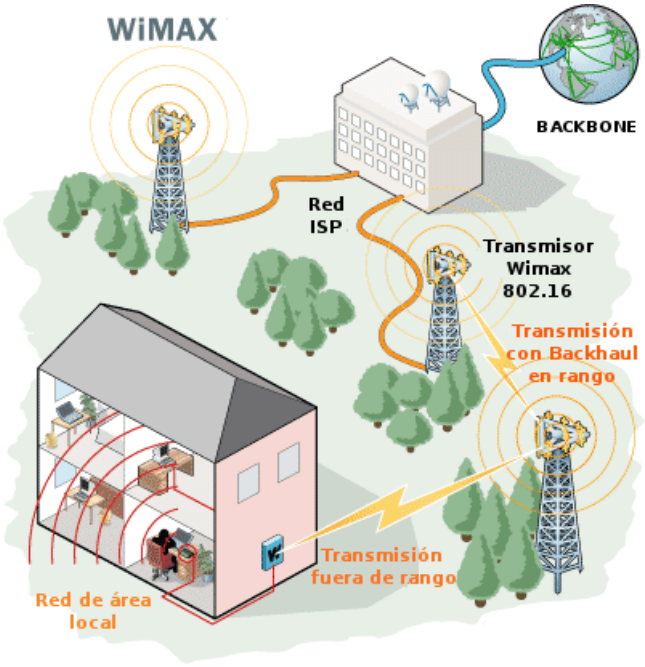

Figura 1.4: Arquitectura de una red WiMAX [Sprint 10]

Para que una red WiMAX pueda alternar la comunicación entre antenas de transmisión y usuarios finales, utiliza dos tipos de servicios:

- Comunicación "Fuera de Rango" (*non-line-of-sight*): muy parecida a la tecnología WiFi, establece comunicación por microondas de baja frecuencia (de 2 a 11 GHz) entre la computadora del usuario final y la torre de transmisión. Las microondas de baja frecuencia tienen mayor capacidad de difracción y doblamiento, lo que les permite bordear mejor los objetos.

- Comunicación "En Rango" (*line-of-sight*): ocurre cuando una antena de transmisión apunta directamente a otra. La comunicación en rango genera un enlace más fuerte y más estable, lo que permite enviar más información con menores probabilidades de error. Este servicio usa frecuencias mucho más altas (66 GHz), lo que permite incrementar el área de cobertura y reducir la interferencia (Figura 1.4) .

Para redes de cómputo móvil, WiMAX incorpora una extensión del protocolo conocida como la enmienda 802.16e, que permite que un dispositivo móvil cambie de celda de cobertura sin perder conexión con la red [Ergen 09], de forma similar a como trabajan los teléfonos celulares (figura 1.5).

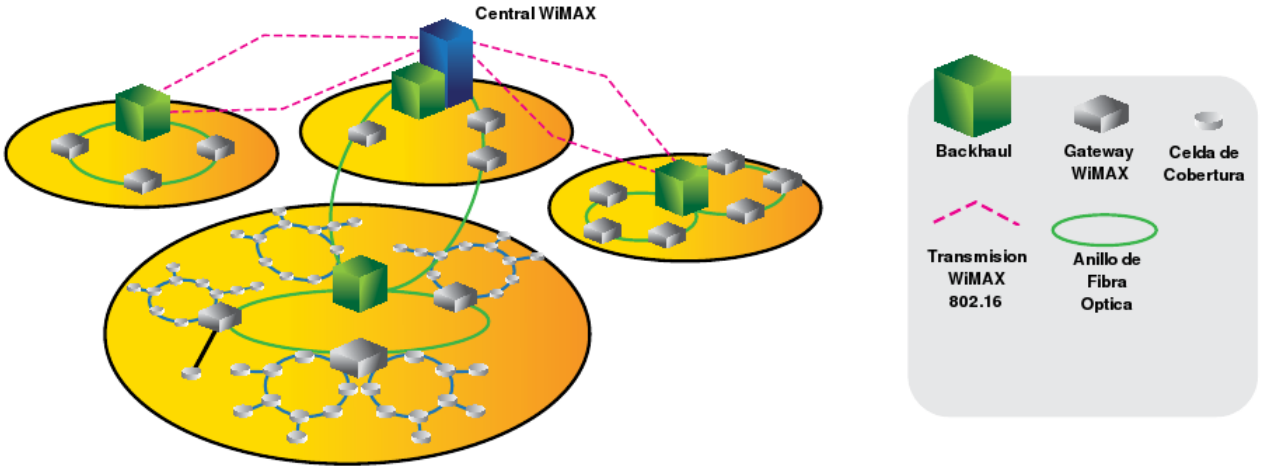

Figura 1.5: Diagrama de WiMAX móvil [Sprint 10].

WiMAX combina una poderosa tecnología inalámbrica con un protocolo de ruteo eficiente que permite intercomunicar dispositivos en constante movimiento, utilizando elementos estáticos como torres de transmisión y gateways fijos. La modularidad de este tipo de redes las hace altamente expandibles y fácilmente reparables. En países como la República Democrática del Congo, WiMAX ha sido ampliamente usada para expandir la cobertura de Internet en zonas rurales a un costo muy bajo.

1.1.1. Ruteo de paquetes en redes con tolerancia al retraso

Como hemos visto, existe una gran variedad de DTN que resuelven de distintas formas las complicaciones impuestas por el medio en el que operan. Todas, además de contar con tecnologías que permitan establecer comunicaciones estables, deben contar con lógicas de coordinación entre dispositivos que permitan aprovechar lo mejor posible los recursos de la red. Dichas lógicas, materializadas como protocolos de ruteo, han probado ser uno de los aspectos más desafiantes en la implementación de DTN.

Los protocolos de ruteo de información en redes convencionales se sustentan en que, al haber constante conectividad entre los nodos de la red, existe en todo momento una trayectoria definida entre un nodo y otro antes del envío de la información. En las DTN, al no existir una conectividad permanente, no es posible trazar una trayectoria definida entre nodos antes del envío de información y esta tiene que ser enviada y almacenada en algún otro punto de la red con la esperanza de que llegue a su destino exitosamente [Kupier 08]. Una técnica comúnmente empleada para aumentar las probabilidades de que la información sea transmitida exitosamente es almacenar varias copias del mensaje en distintos nodos, esperando que alguna de estas logre llegar a su destino. Esto es posible si los nodos de la red cuentan con suficiente capacidad de almacenamiento de información y si el ancho de banda entre los nodos es suficientemente alto con respecto al tráfico esperado. Los algoritmos de ruteo, encargados de establecer las reglas de intercambio de información entre nodos, juegan un papel crucial en el desempeño global de la red pues de ellos depende la dinámica con la que los paquetes de información

se desplazarán en la red. Una dinámica de intercambio de información eficiente puede traducirse en menores requerimientos tecnológicos o una mayor velocidad en el intercambio de información.

1.2. Sistemas complejos

En su forma más general, los sistemas complejos se definen como sistemas dinámicos compuestos de partes interconectadas que exhiben propiedades globales que no pueden ser explicadas a partir de las propiedades individuales de los elementos que lo forman [Bar 97]. Es decir que las interrelaciones de los elementos que componen un sistema complejo contienen información que determina en gran medida las propiedades globales que exhibirá el sistema, imposibilitando así que este pueda ser entendido como "la suma de sus partes". Dichas propiedades reciben el nombre de "propiedades emergentes" y, aunque resultan ser un denominador común en el estudio de los sistemas complejos, distan de ser la única característica de lo que entendemos por "complejidad". La no linealidad en los sistemas dinámicos, la presencia de caos en modelos deterministas y los sistemas adaptativos completan el cuadro esencial del estudio de la complejidad.

1.2.1. Sistemas no lineales

Matemáticamente, una función es lineal si cumple con la propiedad de aditividad:

$$f(x+y) = f(x) + f(y) \tag{1.1}$$

y la propiedad de homogeneidad:

$$f(ax) = af(x) \tag{1.2}$$

De manera que una función como $f(x) = x^2 + 5$ resulta ser no lineal ya que no cumple con ninguna de las condiciones anteriores. Una ecuación tan sencilla como la última no supone una alta complejidad, esta surge al nivel de ecuaciones diferenciales no lineales que representan sistemas dinámicos pues surgen dos principales complicaciones. Por un lado, al no cumplirse la propiedad de aditividad, una ecuación diferencial no lineal no puede ser expresada como la suma de ecuaciones diferenciales más sencillas, complicando enormemente su resolución. Por otro, al no cumplirse la propiedad de homogeneidad, los cambios en las condiciones iniciales provocan cambios desproporcionados en el resultado de la ecuación. Esto puede ser observado en el clima terrestre, un sistema altamente no lineal en donde un ligero cambio en la temperatura, presión o humedad puede desatar resultados tan catastróficos como un huracán.

La linealidad de los sistemas constituye un estudio fundamental en la física ya que la gran mayoría de los sistemas encontrados en la naturaleza se comportan como sistemas no lineales y por tanto son difícilmente predecibles.

1.2.2. Caos en sistemas deterministas

Un sistema determinista, en oposición a un sistema probabilístico, es aquel en el que el azar no interviene en los estados futuros del sistema. Conociendo con exactitud el estado actual del sistema y dada la ecuación que lo representa es posible conocer su estado futuro. Por ejemplo, consideremos la siguiente ecuación poblacional:

$$s_{t+1} = rs_t(1 - s_t), \tag{1.3}$$

en donde s_{t+1} representa la población de individuos al año siguiente, r representa la tasa de crecimiento y s_t representa la población actual medida entre 0 (total extinción) y 1(máxima población posible). De un primer análisis podríamos concluir que es una ecuación determinista sencilla, ya que al conocer la población actual y la tasa de crecimiento podemos conocer la población del siguiente año. Analizando un poco más la ecuación podríamos sugerir que a mayor tasa de crecimiento existirá una mayor población el siguiente año y que esta tendencia permanecerá así durante toda la evolución de la población, pero, como descubriría el biólogo Robert May en la década de los 70 [May 76], esto sólo se cumple para ciertos valores. Tomemos por ejemplo un valor $r = 2$ y una población inicial de 0.2, si estudiamos la historia dinámica de la población observaremos:

$$s_{t+1} = 2 * 0.2(1 - 0.2) = 0.32$$

$$s_{t+1} = 2 * 0.32(1 - 0.32) = 0.4352$$

$$s_{t+1} = 2 * 0.4352(1 - 0.4352) = 0.4916$$

$$s_{t+1} = 2 * 0.4916 * (1 - 0.4916) = 0.4998$$

$$s_{t+1} = 2 * 0.4998 * (1 - 0.4998) = 0.5$$

$$s_{t+1} = 2 * 0.5 * (1 - 0.5) = 0.5.$$

Que representa una estabilización en la población pues en los años subsiguientes la población se conserva en un valor de 0.5 [Braun 96]. Lo mas sensato sería esperar que al aumentar la tasa de crecimiento, la población se estabilizará a un valor más alto que 0.5. En realidad, al superar una tasa de crecimiento de 3, la población comienza a mostrar un comportamiento extraño, al cabo de 10 años comienza a oscilar entre dos valores, 0.764 y 0.558. Hasta aquí, nuestra ecuación determinista ya no es tan predecible como en el caso anterior pero aún conserva un alto grado de predictibilidad pues pasados los 10 años, sabemos que si un año hubo una población de 0.764 el siguiente la población será de 0.558.

Conforme aumentamos el valor de r, la ecuación empieza a oscilar, en lugar de entre dos valores, entre cuatro, después entre 8 y así sucesivamente. Al graficar los números entre los cuales oscila nuestra ecuación conforme aumentamos el valor de r obtenemos la Figura 1.6, en donde podemos observar que al llegar a un valor de r cercano al 4 es imposible predecir el tamaño de la población para el año siguiente, pues este puede caer en un infinidad de posibles valores. A esta condición se le conoce como caos determinista [Braun 96].

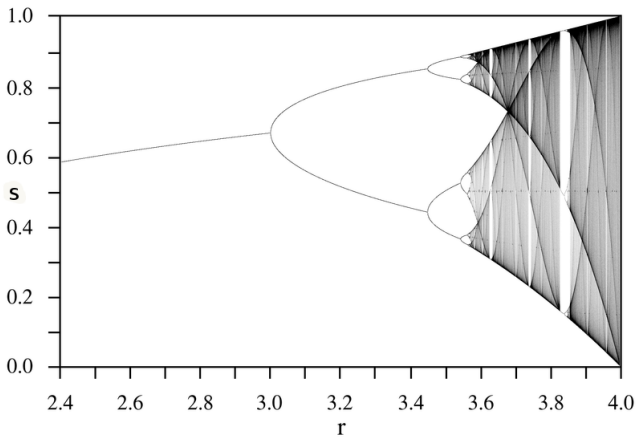

Figura 1.6: Diagrama de bifurcación para la ecuación de poblaciones [Wk 05].

En las ciencias exactas, una de las principales funciones de modelar matemáticamente un sistema físico es la capacidad de predecir el comportamiento del mismo. En el siglo XVII, el filósofo Immanuel Kant sostenía que en la ciencia existen dos tipos de modelos, "explicativos" y "predictivos", los primeros, como la teoría de la evolución de Darwin, son capaces de explicar el origen de cierta fenomenología pero incapaces de predecirla: Darwin fue capaz de explicar el origen de las especies pero incapaz de predecir las especies futuras. Los modelos predictivos, como la teoría de gravitación universal de Newton son su contraparte: Newton no fue capaz de explicar la gravedad como propiedad de la masa pero sí de predecir la atracción gravitatoria entre cualquier par de cuerpos dadas sus masas. Kant suponía que una teoría científica es predictiva en tanto que es "matematizable", lo que

quiere decir que si es posible formular la naturaleza de un fenómeno como una ecuación matemática, entonces este fenómeno puede ser predecible; en la resolución de la ecuación que lo modela radica la predictibilidad y el entendimiento último del mismo [Kant 86].

Gran parte de las abstracciones matemáticas realizadas para modelar sistemas físicos se sustentan en el argumento de un que el efecto de un acontecimiento menor y distante puede ser ignorado sin perder exactitud o afectar significativamente la naturaleza del fenómeno, incluso, en muchos casos es posible ignorar aspectos tan tangibles como la fricción o la masa para privilegiar la sencillez de un modelo sin comprometer su naturaleza. El avance en el estudio de los sistemas complejos ha permitido romper con los paradigmas anteriores. El caos en sistemas deterministas muestra que en algunos sistemas, a pesar de poder "matematizar" un fenómeno, es imposible predecirlo. La no linealidad muestra que algunos sistemas son extraordinariamente sensibles a fenómenos que pueden ser considerados insignificantes y por lo tanto no pueden ser entendidos como sistemas aislados. Las propiedades emergentes muestran la existencia de sistemas en donde la relación entre sus elementos es determinante para la evolución del mismo.

Como hemos visto, los sistemas complejos representan fenómenos cuya predictibilidad se ve significativamente reducida pero, al mismo tiempo, aportan importantes propiedades como una mayor riqueza en los estados dinámicos o el estado de borde de caos. Estas propiedades, medibles gracias a conceptos de teoría de la información, constituyen motores de complejidad que son indispensables en muchos fenómenos naturales. Por ejemplo, hoy sabemos que un corazón sano exhibe una mayor complejidad en sus latidos que uno enfermo [Aziz 07], que un gas que exhibe una dispersividad cercana al caos se expande con mayor eficiencia [Landsberg 98], que el movimiento ocular de los mamíferos muestra comportamientos complejos [Low 08], etc.

Una de las áreas en donde se han aplicado con gran éxito los modelos de sistemas complejos es en la representación de sistemas biológicos ya que parecen abstraer las propiedades de auto-organización y comportamiento cooperativo que caracterizan a estos sistemas [Bonabeu 91]. En sociedades de hormigas, por ejemplo, se ha logrado modelar un estado de transición entre el orden y el desorden del hormiguero que muestra propiedades emergentes que emulan un comportamiento colectivo óptimo [Solé 93]. La capacidad de exhibir este tipo de propiedades hacen de los modelos de sistemas complejos candidatos atractivos para ser aplicados en aspectos de ingeniería y ciencias de la computación, ya que su eficiencia ya ha sido observada en fenómenos naturales.

1.3. Redes y complejidad

Dadas las características de las DTN (conjunto de nodos con conectividad intermitente gobernados por una dinámica global), es posible abordar el problema del ruteo de información con un enfoque de sistemas complejos. De esta manera, el problema de la intermitencia en la conectividad de los nodos puede solucionarse si se encuentra una dinámica de conexión lo suficientemente "compleja" para garantizar que la conectividad global en la red no se vea afectada, aún cuando haya un gran numero de nodos desconectados en la misma en un instante de tiempo. Como hemos visto, la complejidad es un concepto amplio y ambiguo, en nuestro caso, la complejidad en la intermitencia de la conectividad de los nodos puede ser medida a través de conceptos de teoría de la información [Li 08], específicamente a través de la entropía informática de Shannon y la complejidad algorítmica de Kolmogorov, que definiremos más adelante (sección 2.3).

Para analizar la complejidad de un modelo de ruteo de información es imprescindible conocer o estimar la forma en la que se desplazan los nodos en la red. Una dinámica de ruteo puede ser eficiente para ciertos tipos de movimientos e ineficiente para otros. El tipo de desplazamiento conocido como "Vuelos de Lévy" resulta ser un tipo de movimiento interesante, ya que presenta dos importantes características: a) se ha comprobado que es una distribución espacial con alta dispersividad, que la hace una óptima estrategia de búsqueda aleatoria cuando el terreno es desconocido [Viswanathan 00] y b) es una distribución ampliamente observada en la naturaleza, especialmente en estrategias de exploración de animales [Ramos 04, Viswanathan 96] y en la forma en la que los seres humanos nos desplazamos [Rhee 08]. Aunque existen trabajos en DTN cuyos elementos se desplazan por Vuelos de Lévy [Lee 08, Sheen 08], pocos se enfocan en estudiar de que manera afecta el tipo de movimiento la dinámica de la red.

El objetivo de este trabajo es investigar la complejidad de una dinámica de ruteo e intercambio de información de una red de dispositivos móviles que utiliza una ecuación de estados empleada para modelar oscilaciones de actividad en sociedades de hormigas. Nuestro interés se centra en conocer la variación de la complejidad de la red

(medida a través de la entropía de Shannon y la complejidad de Kolmogorov), de acuerdo a la forma en la que los dispositivos se desplazan. Para ello analizaremos tres tipos de desplazamiento incluyendo el desplazamiento por vuelos de Lévy. En un marco más general, nuestra intención es aplicar a un problema de ingeniería un modelo de sistemas complejos bio-inspirados con el afán de aprovechar el potencial de estos últimos. Nuestra herramienta de investigación será la simulación computacional directa, programando los modelos antes mencionados en Python y lenguaje C, ya que estos constituyen una buena combinación entre versatilidad sintáctica y rapidez de ejecución.

Capítulo 2

Marco Teórico

2.1. Autómatas celulares

Los autómatas celulares (AC) son un tipo de sistemas matemáticos deterministas, discretos en tiempo y espacio caracterizados por tener interacciones locales y por su inherente evolución paralela [Ilachinsky 01]. Descritos por primera vez en la década de los 50s por John von Neumann para representar modelos simples de autoreplicación biológica, los autómatas celulares son modelos prototípicos de sistemas complejos y de procesos compuestos por un gran número de elementos idénticos, simples y que interactúan localmente. El estudio de estos sistemas ha generado gran interés a través de los años debido a la habilidad que poseen de generar un amplio espectro de patrones de comportamiento complejo a partir de reglas sencillas.

Aunque la mayoría de los estudios teóricos realizados sobre AC competen a la matemática y a la ciencia computacional, existen numerosas aplicaciones de estos modelos en ciencias como la física, la biología, la química, la bioquímica y la geología entre otras. Algunos ejemplos específicos de fenómenos que han sido modelados utilizando AC incluyen fluidos y turbulencia química [d'Hum 86][Gerhardt 89], desarrollo de plantas [Linde 89] y crecimiento dendrítico de cristales [Kess 90], teoría ecológica [Phip 92], evolución del ADN, propagación de enfermedades infecciosas [Segel 99], dinámicas sociales [Axtell 96], patrones de actividad eléctrica en redes neuronales [Fran 92], etc.

A pesar de que existe una gran variedad de modelos particulares de AC, cada uno diseñado para cumplir los requerimientos específicos de un sistema dado, podemos describir cinco características que poseen la gran mayoría de ellos:

Matriz discreta de celdas: El sustrato del sistema consta de una matriz celular de una, dos o tres dimensiones.

Homogeneidad: Todas las celdas son idénticas y equivalentes.

Estados discretos: Cada celda toma un solo valor de un número finito de valores discretos.

Interacciones locales: Cada celda interactúa sólo con celdas que se encuentran en su vecindad.

Dinámica discreta: Para cada unidad discreta de tiempo, cada celda actualiza su estado de acuerdo a una regla de transición tomando en cuenta los estados de las celdas vecinas.

Al ser temporal y espacialmente discretos, los modelos representados por AC resultan ser computables con absoluta exactitud. No existen problemáticas como la acumulación de errores por redondeo o truncamiento y cada característica dinámica observada en el modelo adquiere la rigurosidad de un teorema [Toffoli 77]. Así, la herramienta indispensable para estudiar un AC es una computadora lo suficientemente robusta para satisfacer los requerimientos del modelo. Simular un AC en un programa de computadora equivale a utilizar las reglas que lo gobiernan para hacer que los arreglos de celdas evolucionen a partir de un estado inicial, de esta forma, al producirse configuraciones sucesivas de celdas, los AC procesan información de forma paralela.

Veamos un ejemplo extraído del popular AC conocido como "El juego de la Vida" [Gardner 70]: en este AC sólo existen dos posibles estados, "vivo", representado por una celda negra, y "muerto", representado por una celda blanca. Cada celda toma en cuenta sus 8 vecinos inmediatos (Vecindad de Moore) y sigue las siguientes reglas:

Nacimiento: reemplazar una celda previamente muerta por una viva si 3 de sus vecinos inmediatos están vivos.

Muerte: reemplazar una celda previamente viva por una muerta si (a) la celda viva no tiene más de un vecino inmediato vivo (muerte por aislamiento), o (b) la celda viva tiene más de tres vecinos inmediatos vivos (muerte por sobrepoblación).

Supervivencia: mantener una celda viva y si ésta tiene dos o tres vecinos vivos.

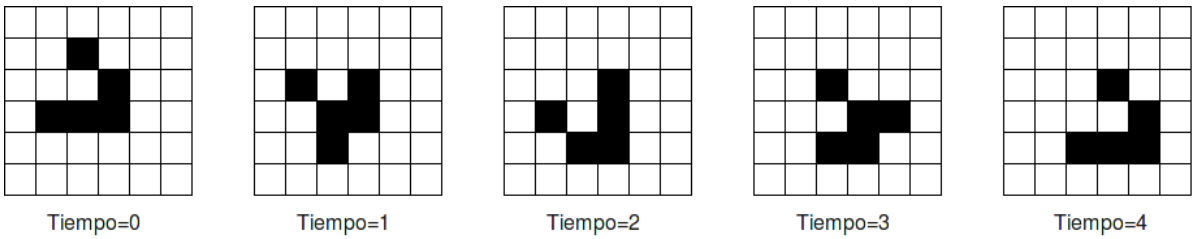

Figura 2.1: Sucesión de configuraciones en el "juego de la vida"

Considerando el estado inicial y los estados subsecuentes de la figura 2.1, podemos apreciar la evolución paralela de las celdas, ya que en cada instante de tiempo se evalúan las condiciones de cada celda para calcular la configuración de la siguiente generación. En este caso podemos observar una estructura cíclica que se repite cada 4 unidades de tiempo con un desplazamiento de una unidad diagonal. Esta estructura es conocida como "deslizador" y es una de las estructuras oscilatorias más notables del juego de la vida.

2.1.1. Cómputo universal y autómatas celulares

El autómata bidimensional autoreplicativo de von Neumann fue el primer modelo discreto de computación paralela en cumplir por completo las características de una máquina de cómputo universal (Maquina de Turing) [Ilachinsky 01]. Veinte años después, John Conway introdujo las reglas del ya mencionado "Juego de la Vida" que continúa siendo uno de los sistemas de cómputo universal, rigurosamente probados, más simples conocidos [Gardner 70].

Universalidad computacional es la habilidad de los sistemas de implementar cualquier algoritmo finito y por tanto evaluar cualquier función computable. Esta propiedad puede ser demostrada ya sea probando que un sistema dado es formalmente equivalente a un sistema de cómputo universal conocido o identificando directamente la dinámica de las estructuras intrínsecamente generadas con los estados de transición de una máquina de Turing.

La propiedad de cómputo universal otorga, por un lado, el mismo poder computacional que una computadora digital convencional, pero por otro restringe severamente cualquier predicción teórica acerca del comportamiento del sistema [Ilachinsky 01]. En los términos más estrictos posibles, el comportamiento a largo plazo de un sistema dinámico que se comporte como máquina de cómputo universal sólo puede ser obtenido mediante simulación directa, no existe un procedimiento general predictivo. Esto implica que para sistemas como el autómata de von Neumann, y en general para cualquier AC, no puede haber una expresión analítica que describa su comportamiento asintótico ni una ecuación que defina el comportamiento a largo plazo que pueda ser resuelta en menos tiempo de lo que tomaría al autómata evolucionar hasta llegar a dicho plazo. Todos estos sistemas, computacionalmente irreductibles, comparten la propiedad de que su propia evolución resulta ser la más eficaz simulación de su comportamiento.

2.1.2. Definición de autómata celular

Un autómata celular puede ser definido a partir de una la tupla $\{Z, S, \{k'\}, f\}$ donde:

- Z es una matriz finita o infinita que representa el espacio "físico" en el que actúa el autómata
- S es un conjunto finito de estados o valores que puede tomar cada celda de la matriz Z
- $\{k'\}$ es un conjunto de celdas vecinas

- f es una función de transición local definida por una regla o tabla de transiciones

La matriz Z actúa como una cuadrícula regular finita o infinita de celdas. Cada celda está definida por su posición en la matriz y un valor asociado que se extrae del conjunto S, que contiene todos los posibles valores que puede tomar una celda.

El tiempo, como ya mencionamos, es discreto y el estado futuro de una celda en el tiempo $t+1$ es función del estado presente de la misma y del estado presente de las celdas que la rodean (conjunto N). Así, la función de transferencia $f : S^t \rightarrow S^{t+1}$ permite calcular el estado futuro de una celda $S(k;t)$ a partir del estado de las celdas que la circundan en el estado presente. La función de transferencia puede ser expresada como una función dinámica del tipo:

$$S(k,t) = F_\mu \left[\sum_{k'} \alpha(k) S(k',t) \right], \tag{2.1}$$

en donde k representa una posición en la matriz, F_μ el conjunto de reglas de estado, $\{k'\}$ el conjunto de vecinos más cercanos de k y $\alpha(k)$, un conjunto de coeficientes. Así, el estado de cada celda es función del tiempo t y la posición k. El hecho de que cada celda actualice su estado de acuerdo al estado de las celdas vecinas es el origen del comportamiento cooperativo que exhiben los AC.

2.1.3. Autómatas celulares móviles

Los autómatas celulares móviles (ACM) son un tipo de AC utilizados para representar sistemas cuyos elementos pueden desplazarse en un área específica, de esta manera, las reglas de los ACM emulan las restricciones propias del desplazamiento espacial, por ejemplo, la imposibilidad de que dos celdas ocupen la misma posición al mismo tiempo [Wolf 87].

Resulta común encontrar modelos bidimensionales en donde existe una condición de movilidad representada por una variable local m_i. De acuerdo a las reglas del ACM esta variable determina si un elemento es visto como activo y puede desplazarse o si permanece inmóvil. De esta manera, el conjunto de celdas Z de un ACM se divide en tres subconjuntos:

- Z_1 conjunto de elementos activos y por lo tanto móviles.
- Z_2 conjunto de elementos inactivos y por lo tanto inmóviles (su estado no se actualiza).
- Z_3 conjunto de celda vacías o "disponibles".

Si el número de elementos (activos e inactivos) iguala al numero de celdas, se dice que el sistema está saturado (densidad=1) y los elementos no se moverán debido a la falta de celdas disponibles. Si esta situación permanece, el ACM bidimensional será equivalente a un AC "convencional" en donde todas las celdas activas se actualizan sin movimiento y en donde el número y tipo de conexiones entre celdas se mantiene invariante durante toda la evolución del sistema.

2.1.4. Redes neuronales

Un tipo especialmente importante de AC son las redes neuronales formales, cuyo objetivo es modelar el comportamiento cooperativo del sistema nervioso animal para estudiar sus propiedades estadísticas y computacionales. Típicamente, los elementos que constituyen las redes neuronales (neuronas) son celdas binarias que pueden estar activas o inactivas, dependiendo del estado de otros elementos de la red.

Los sistemas de Hopfield son un modelo especial de red neuronal que ha atraído un gran interés debido a su propiedad de recursividad y a la gran gama de propiedades emergentes que pueden presentar. El modelo de Hopfield está definido por un conjunto de elementos de la forma

$$\Xi(n,t) = \{S_i(t), \ldots, S_n(t)\}, \tag{2.2}$$

donde $S_i(t)$ el estado de la neurona i en el tiempo t [Hopfield 82]. En este tipo de sistemas $S_i \in \mathbb{R}$ y la dinámica de la red está definida por la regla determinística

$$S_i(t+1) = \Phi \left[g \sum_{j=1}^{n} J_{ij} S_j(t) \right]. \tag{2.3}$$

Por lo general, en las redes de Hopfield $\Phi(z) = tanh(z)$ o cualquier otra función sigmoidal. La ecuación 2.3 permite actualizar el estado de la i-ésima neurona a partir del estado de las neuronas vecinas. El llamado parámetro de ganancia g es una medida del grado de no-linealidad en la interacciones entre neuronas, para $g \to \infty$ la función Φ converge a una función escalón convencional. La figura 2.2 muestra la función $\Phi(z) = tanh(gz)$ para diferentes valores de g

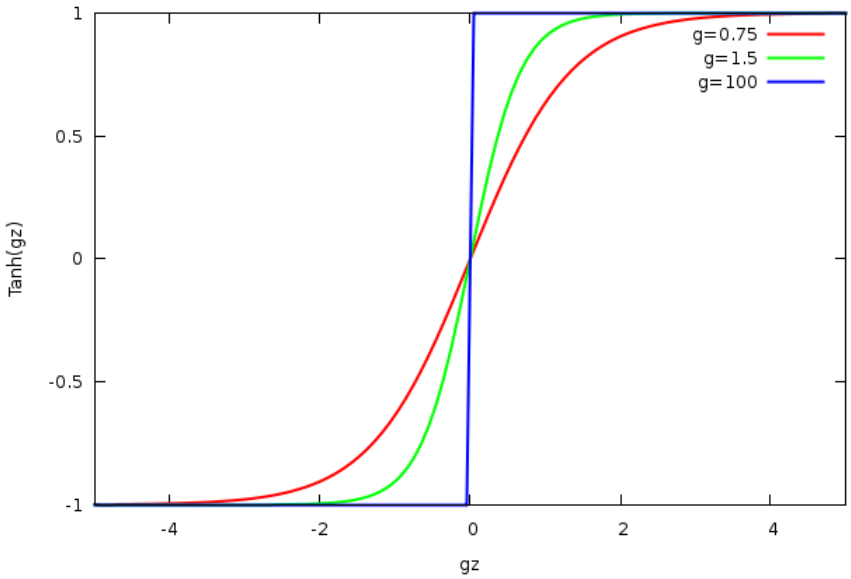

Figura 2.2: Gráfica $\Phi(z) = tanh(gz)$ para tres valores de g.

En la ecuación 2.3, J_{ij} representa la matriz de acoplamiento entre neuronas, que refleja el tipo de interacción entre cada par (i, j) de neuronas. En el modelo de Hopfield, la matriz es asumida como simétrica $J_{ij} = J_{ji}$ y no existe auto-interacción de neuronas $J_{ii} = 0$.

El modelo de Hopfield presenta una amplísima gama de características incluyendo memoria asociativa y patrones caóticos [Skarda 87]. Estas características hacen de las redes de Hopfield modelos ideales para representar sistemas neuronales biológicos [Chua 88].

2.2. Vuelos de Lévy

Los vuelos de Lévy, nombrados así en honor del matemático francés Paul Pierre Lévy, son un tipo de caminata aleatoria en donde los incrementos del desplazamiento están distribuidos de acuerdo a una ley de potencias del tipo

$$P(l_j) \sim l_j^{-\mu}, \tag{2.4}$$

donde $1 < \mu \leq 3$ y l_j representa la longitud del desplazamiento o "vuelo" [Barth 08] . Al graficar la ecuación 2.4 para $\mu = 1$ obtenemos la figura 2.3, de donde podemos observar que entre más largo es el vuelo, menos frecuente resulta. La distribución de Lévy es un tipo de distribución probabilística de "larga estela" ya que no está limitada en su dominio y por lo tanto su varianza es infinita.

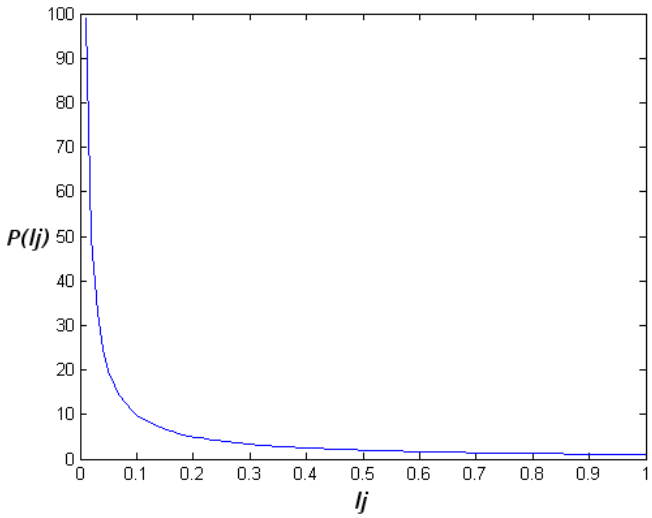

Figura 2.3: Gráfica de longitudes de Lévy para $\mu = 1$

La presencia de vuelos de Lévy bidimensionales en fenómenos naturales así como sus propiedades fractales fueron descritas por Benoît Mandelbrot en 1982 en su trabajo "La geometría fractal de la Naturaleza". Debido a su escalamiento exponencial, los vuelos de Lévy bidimensionales presentan la característica fractal de autosimilitud (figura 2.4), es decir que una porción de la trayectoria (marcada en la figura como recuadros rojos) conserva las mismas proporciones de ocurrencia de longitud de vuelo que la trayectoria completa.

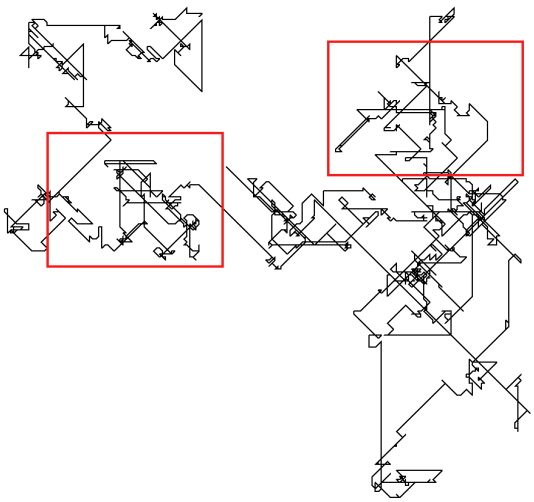

Figura 2.4: Trayectoria bidimensional de 1200 pasos por vuelos de Lévy con $\mu = 1$.

En relación con otros tipos de caminatas aleatorias como el movimiento browniano o la caminata completamente aleatoria, los vuelos de Lévy muestran mayor dispersividad al cubrir mayor área de exploración en menor tiempo. Esto hace de los vuelos de Lévy protocolos de búsqueda sumamente eficientes cuando se desconoce por completo el terreno. Se sabe que el parámetro μ de la distribución de Lévy tiene una relación directa con la eficiencia en la búsqueda [Viswanathan 00] y aunque algunos trabajos difieren ligeramente acerca del valor óptimo de dicho parámetro, la gran mayoría coinciden en un valor cercano a $\mu = 2$.

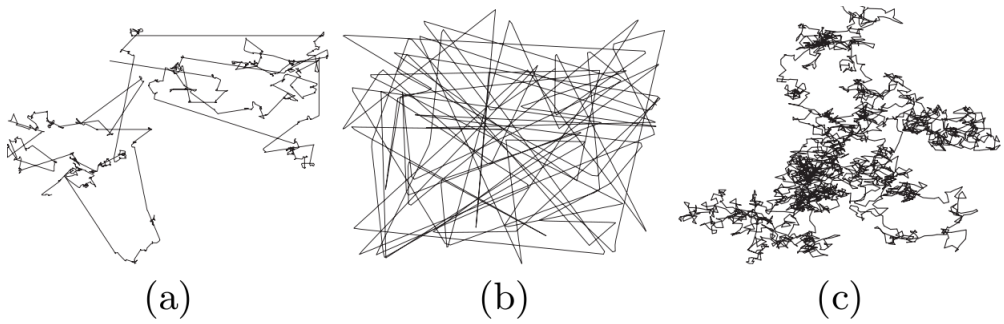

Figura 2.5: Comparación entre tres tipos de caminatas aleatorias (a) Vuelo de Lévy (b) Caminata completamente Aleatoria (c) Movimiento Browniano [Rhee 08]

En la naturaleza, los vuelos de Lévy pueden ser observados en fenómenos tan variados como las estrategias de búsqueda de animales como albatros, chacales, monos araña, abejorros, hormigas y saltamontes, en el movimiento de bacterias y fitoplancton, en el patrón de esparcimiento de las moléculas de algunos gases, el movimiento ocular en mamíferos e incluso en la forma en la que los humanos nos desplazamos [Shles 94, Rhee 08].

2.3. Complejidad y teoría de la información

La consolidación de la física de los sistemas complejos a mediados del siglo pasado trajo consigo el replanteamiento y la ampliación de los fenómenos que atañen a la física como ciencia. El descubrimiento del caos en sistemas determinísticos, las estructuras auto-similares y la ley de Zipf dotaron a la física de herramientas para estudiar fenómenos, antes incomprendidos, como el análisis de turbulencias, las redes neuronales, el flujo de los mercados financieros o las estructuras disipativas [Braun 96]. La complejidad, antes percibida como un fenómeno limitante y preferiblemente evitable, pasó a tener un papel protagónico en el estudio de los sistemas dinámicos. Por ejemplo, hoy sabemos que existen sistemas que al operar en un estado limítrofe entre el orden y el caos liberan una mayor cantidad de energía [Landsberg 98], o que un corazón que exhibe una mayor complejidad en sus latidos es un corazón más sano [Aziz 07]. Así, en la actualidad, la complejidad de un sistema es un parámetro de gran relevancia que no sólo representa qué tan intrincado, irrealizable o carente de control es un sistema sino qué tan rico en información es o cuántos posibles estados produce.

La complejidad, en sí misma, es un término abstracto e indefinible. Para cada rama de las ciencias cobra un significado diferente y aunque en física existen muchas medidas propuestas para su cuantificación hay pocos acuerdos sobre su utilidad y universalidad.

En general, podemos hablar de 4 tipos de complejidad [Mitchel 09]:

Comportamiento complejo El comportamiento de un sistema complejo comúnmente se asocia a propiedades emergentes y de auto-organización. La teoría del caos investiga la sensibilidad de los sistemas a la variación de sus condiciones iniciales como una causa del comportamiento complejo.

Mecanismos complejos Vida artificial, computación evolutiva, algoritmos genéticos y teoría de juegos son ejemplos de mecanismos complejos, cuya principal característica es la de exhibir capacidades adaptativas.

Sistemas complejos Los sistemas complejos son típicamente sistemas dinámicos biológicos, económicos, sociales, químicos o físicos que tienden a ser multidimensionales, altamente no lineales, caóticos y difíciles de modelar.

Complejidad en la información Gran parte de los estudios en teoría de la información y teoría algorítmica de la información se centran en el estudio de la predictibilidad de las cadenas de datos.

Al centrarnos en la complejidad en la información, surgen dos medidas que son particularmente útiles y especialmente importantes para este trabajo: la entropía de Shannon y la complejidad algorítmica de Kolmogorov, que, a pesar de ser medidas definidas para representar exclusivamente la complejidad de cadenas binarias, resultan ser herramientas muy útiles para caracterizar sistemas dinámicos ya que es posible representar estos últimos como cadenas binarias y entonces aplicar ambas medidas para conocer su complejidad.

2.3.1. Entropía de Shannon

La entropía de Shannon o entropía informática es una cuantificación de la incertidumbre que puede arrojar una cadena de datos de acuerdo al lenguaje en la que está escrita [Cover 06].

Al arrojar un dado al aire se tienen $N = 6$ posibles resultados igualmente probables, $p_i = 1/6$ para $i = 1, \ldots, 6$. El resultado de arrojar el dado puede ser interpretado como recibir un mensaje y el lenguaje o alfabeto utilizado serán los números del uno al seis, cada uno con la misma probabilidad. Al aumentar N, aumenta la incertidumbre del sistema porque la gama de posibilidades en el resultado también aumenta.

Una medida de la incertidumbre o ganancia informática del sistema I, sólo puede ser obtenida si resulta aditiva para eventos independientes, es decir, que si existen dos conjuntos de posibles resultados N_1 y N_2, el número total de resultados es $N = N_1 \cdot N_2$ y para eso es necesario que

$$I(N_1 \cdot N_2) = I(N_1) + I(N_2). \tag{2.5}$$

Condición que puede ser satisfecha únicamente por:

$$I = c \log N, \tag{2.6}$$

donde c es una constante arbitraria convencionalmente fijada a 1 para cadenas binarias. Así, la ecuación anterior puede expresarse como

$$I = \log_2 N \qquad (bits). \tag{2.7}$$

Para el caso del dado arriba mencionado la entropía informática será de $log_2 6 = 2{,}58$ bits.

Ahora bien, si pensamos en eventos cuyos resultados tienen diferentes probabilidades de ocurrencia, la ecuación cambiará. Una moneda arrojada al aire tendrá una entropía de 1 bit pues sólo existen dos posibles resultados con la misma probabilidad, es decir $N = 2$, pero si la moneda estuviera cargada, digamos un 70 % hacia cara, la incertidumbre del sistema disminuiría.

Para dos conjuntos de resultados independientes N_1 y N_2 el número total de posibles resultados es $\nu = N!/(N_1!N_2!)$ donde $N = N_1 + N_2$. Ahora, para obtener la entropía del sistema es necesario promediar la entropía de los posibles resultados entre el número de posibles resultados, es decir:

$$\Lambda = \frac{I}{N} = \frac{\log_2 \nu}{N} \tag{2.8}$$

Que al aplicar la aproximación de Stirling para factoriales: $\log n! \approx n \log n - n$ se transforma en

$$\Lambda^{(n)}(p_1, p_2 \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i \qquad (bits/simbolos) \tag{2.9}$$

donde $p_i = \frac{N_i}{N}$ es la probabilidad individual de que un símbolo o conjunto de símbolos de un sistema ocurra en la cadena.

Así, la ecuación 2.9 se conoce como la entropía de Shannon para cualquier sistema o cadena de símbolos en donde N representa los posibles símbolos o resultados del sistema y p_i la probabilidad individual de que dichos símbolos ocurran.

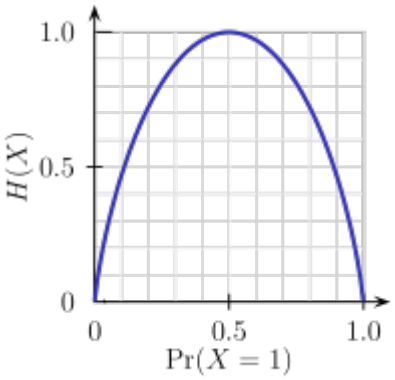

Figura 2.6: Entropía de Shannon de una moneda arrojada al aire.

La entropía de Shannon se sustenta en un esquema probabilístico de ocurrencia de símbolos, es decir que cuantifica los cambios o variaciones que un sistema puede producir y, al ser la información, en esencia, un conjunto de cambios asociados a algún significado, la entropía informática permite medir:

- La cantidad de información contenida en un mensaje o la capacidad informática de un sistema. Por ejemplo, un sistema que sólo genera cadenas largas de ceros tiene entropía cero ya que el siguiente caracter siempre será cero y por lo tanto no contiene información. Entre más entropía tiene un mensaje o sistema, mayor riqueza informática contiene.
- La cantidad promedio de bits que necesita una fuente de información para codificar una cadena de datos dada.
- La capacidad mínima de un canal de transmisión para enviar confiablemente un mensaje codificado en un lenguaje específico.

2.3.2. Complejidad algorítmica de Kolmogorov

2.3.2.1. Historia

Andrei Kolmogorov propone en 1965 una interesante definición acerca de la complejidad de una cadena de datos en la que sostiene que, al calcular el tamaño en bits del programa más corto capaz de generar dicha cadena, se obtiene una medida que refleja la complejidad de la misma [Kolmogorov 65]. Con esta medida, conocida como la complejidad algorítmica de Kolmogorov (CAK), es posible conocer la complejidad de una cadena de bits si se conoce la cantidad de recursos computacionales necesarios para su fabricación.

Un año antes, Ray Solomonoff había demostrado que es imposible que exista un algoritmo general capaz de calcular la CAK tal como Kolmogorov la define [Solomonoff 64], por lo que muchos teóricos de la información centraron sus esfuerzos en obtener medidas indirectas algorítmicamente realizables. La más notable, sin duda, es la propuesta por Jacob Ziv y Abraham Lempel en 1976 [Lempel 76] que, aunque no mide la longitud del programa necesario para generar una cadena de longitud n, permite calcular un número $c(n)$ que es una medida útil en el calculo de dicha longitud. Las justificaciones matemáticas que acercan la medida propuesta por Lempel y Ziv a la complejidad de Kolmogorov pueden encontrarse en [Lempel 76], en esta sección, nos limitaremos a describir $c(n)$ y el algoritmo que permite su obtención.

2.3.2.2. Definición

El cálculo de $c(n)$ se obtiene de la siguiente manera: Supongamos que una cadena $s_1 s_2 \cdots s_n$ ha sido reconstruida por un programa hasta el dígito s_r y que s_r ha sido insertado por primera ocasión, es decir que no puede ser obtenido copiándolo de $s_1 s_2 \cdots s_{r-1}$. La cadena hasta el dígito s_r será denotada por $S = s_1 s_2 \cdots s_r \cdot$, en donde el punto indica que s_r ha sido insertado por primera ocasión. Para verificar que el resto de S, por

ejemplo, $s_{r+1} \cdots s_n$ puede ser reconstruido copiando alguna subcadena de S o si se debe introducir un nuevo dígito, Lempel y Ziv proceden con los siguientes pasos [Lempel 76]. Primero se debe tomar un termino $Q \equiv s_{r+1}$ y verificar si está contenido en el conjunto de subcadenas de S (vocabulario) para saber si Q puede ser obtenido simplemente copiando dicha subcadena al final de S. Esto es equivalente a preguntar si Q está contenido en el vocabulario $v(SQ\pi)$ de $SQ\pi$ donde $SQ\pi$ denota la cadena compuesta por S y Q , π representa que el último dígito tiene que ser eliminado (aquí, $SQ\pi = S$). Esta última pregunta puede ser generalizada a situaciones en las que Q también contiene dos o mas elementos, por ejemplo $Q = s_{r+1}s_{r+2}$. Asumamos, por ejemplo, que s_{r+1} puede ser copiado del vocabulario de S . Entonces, usando la proposición de arriba, preguntamos si $Q = s_{r+1}s_{r+2}$ esta contenido en el vocabulario de $SQ\pi$ que es igual a $v(Ss_{r+1})$, y así sucesivamente hasta que Q sea tan largo que no pueda ser obtenido copiando una palabra de $v(SQ\pi)$ y se tenga que introducir un nuevo dígito. El número de pasos de producción c necesarios para crear una cadena, es decir, el número de dígitos insertados por primera ocasión (mas uno si el último paso de copiado no continúa con la inserción de un dígito), es usado por Lempel y Ziv como medida de la CAK de una cadena dada.

Algunos ejemplos:

Para una cadena que solo contiene ceros podríamos decir intuitivamente que debe tener la menor complejidad posible. De hecho, solo es necesario insertar el primer cero y reconstruir el resto de la cadena copiando este dígito:

$$00000\cdots \rightarrow 0 \cdot 0000\cdots$$

la CAK de la cadena es $c = 2$.

De manera similar, una secuencia compuesta de unidades 01

$$010101\cdots \rightarrow 0 \cdot 1 \cdot 01\cdots$$

tendrá un valor $c = 3$.

La CAK de una secuencia 0010 puede ser determinada mediante los siguientes pasos.

1. El primer dígito siempre tiene que ser introducido $\rightarrow 0 \cdot$.
2. $S = 0$, $Q = 0$, $SQ = 00$,$SQ\pi = 0$, $Q \in v(SQ\pi) \rightarrow 0 \cdot 0$.
3. $S = 0$, $Q = 01$, $SQ = 001$,$SQ\pi = 00$, $Q \notin v(SQ\pi) \rightarrow 0 \cdot 01$.
4. $S = 001$, $Q = 0$, $SQ = 0010$,$SQ\pi = 001$, $Q \in v(SQ\pi) \rightarrow 0 \cdot 01 \cdot 0$.

Ahora c es igual al número de partes de la cadena separadas por puntos, $c = 3$. La figura 2.7 muestra el diagrama de flujo del algoritmo que determina $c(n)$ para una cadena de longitud n.

Lempel y Ziv también demuestran que el valor $c(n)$ de una cadena binaria completamente aleatoria de n elementos casi siempre tiende al mismo valor [Kaspar 86]:

$$\lim_{n\rightarrow\infty} c(n) = b(n) \equiv n/\log_2 n \tag{2.10}$$

Por lo que es posible parametrizar $c(n)$ de una cadena con respecto a una cadena completamente aleatoria dividiendo entre el factor $b(n)$ es decir:

$$z(n) = \frac{c(n)}{b(n)}, \tag{2.11}$$

donde $z(n)$ representa la CAK normalizada.

Como podemos observar, el algoritmo de Lempel y Ziv nos permite conocer la complejidad de cualquier cadena S calculando el tamaño del vocabulario que utiliza. En teoría de la información, $c(n)$ cobra una especial relevancia pues permite conocer el número de palabras que utiliza una cadena de datos y, a partir de ello, codificar y comprimir dicha cadena. En sistemas dinámicos, $c(n)$ nos permite, entre otras cosas, conocer la complejidad de la secuencia de estados de un sistema dado o las condiciones bajo las cuales un sistema dinámico genera nueva información.

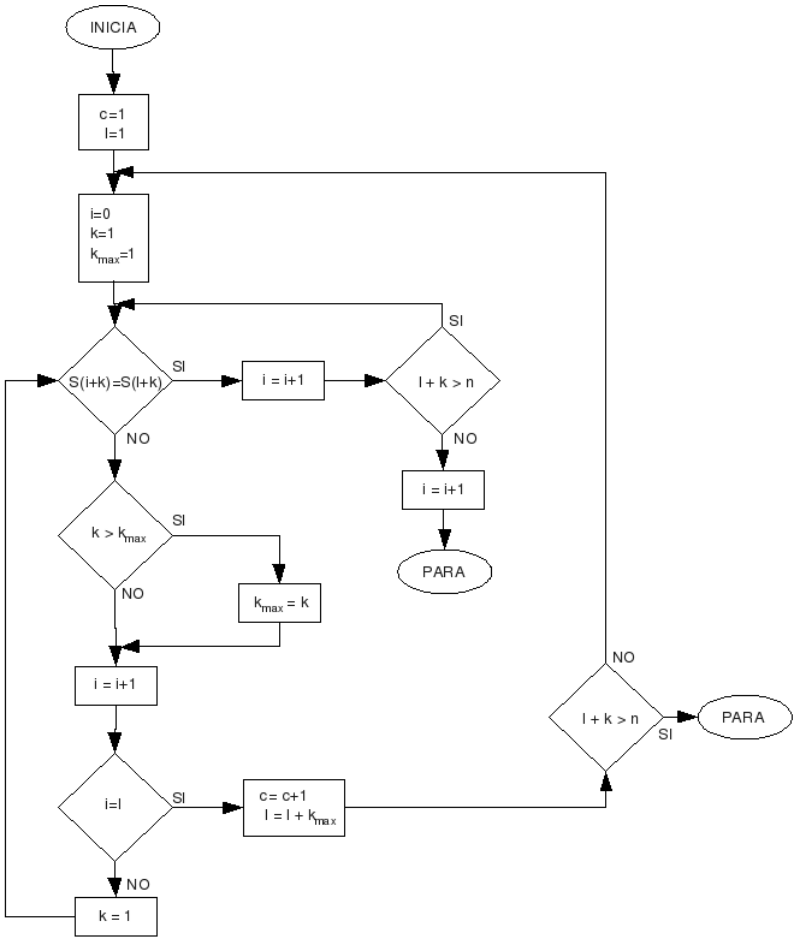

Figura 2.7: Diagrama de flujo del algoritmo que permite calcular $c(n)$ para una cadena de longitud n [Kaspar 86]

2.4. Arquitecturas de red

2.4.1. Arquitectura de red en Internet

Gran parte del extraordinario éxito que ha conseguido Internet en interconectar dispositivos en cualquier parte del mundo se debe al uso de protocolos de comunicación homogéneos conocidos como la suite de protocolos TCP/IP [Hauben 98]. Todos los dispositivos en las cientos de miles de subredes que conforman el Internet utilizan estos protocolos para rutear información y asegurar la integridad del intercambio de datos.

Aunque hoy en día existe una gran variedad de tecnologías inalámbricas y satelitales, la conectividad en Internet depende principalmente de enlaces alámbricos incluyendo redes telefónicas. Estos enlaces están conectados ininterrumpidamente de extremo a extremo en trayectorias con muy bajo retraso entre orígenes y destinos. Tienen una baja tasa de errores y las tasas bidireccionales de intercambio de información son prácticamente simétricas.

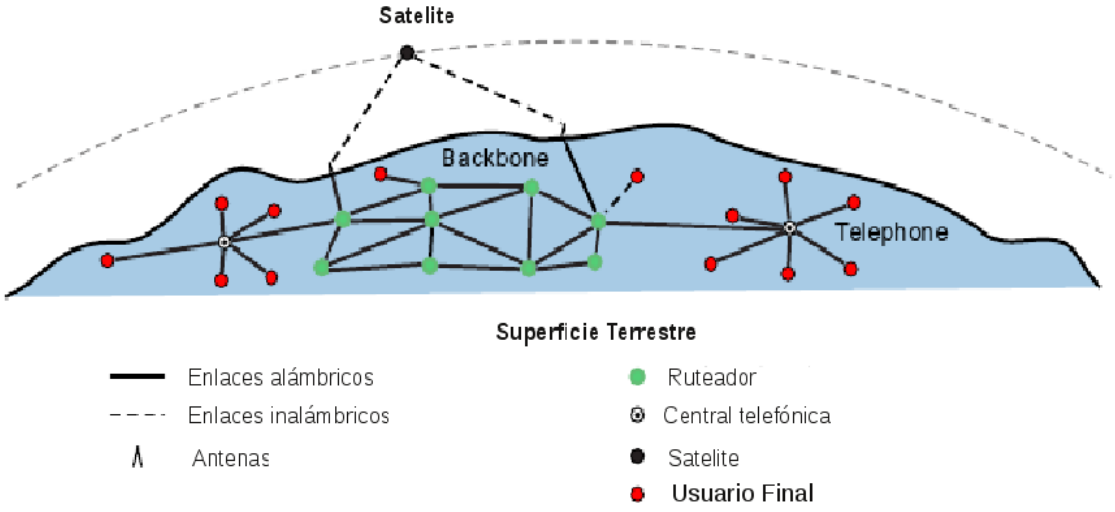

Figura 2.8: Diagrama de las conexiones terrestres que conforman Internet [Warth 03].

En cuanto a la forma en la que se envía la información, Internet utiliza principalmente el modelo de intercambio de paquetes o "*Packet Switching*" en donde un mensaje completo se divide en paquetes de datos que viajan independientemente a través de los nodos de una red. Cada uno de los paquetes, que constituyen un mensaje completo, toma un camino diferente en la red, si un enlace se pierde, los paquetes escogen otro disponible. Cada paquete contiene información propia del mensaje e información de control conocida como encabezado. El encabezado contiene la dirección del dispositivo destino y otra información relevante que indica a los ruteadores como debe ser conducido dicho paquete en la red ("*switching*"). Los paquetes pueden llegar en desorden pero el sistema de transporte del dispositivo destino es capaz de ensamblarlos en el orden correcto.

Para que un mensaje pueda llegar a su destino, debe atravesar las etapas o "capas" del protocolo que se esté utilizando. Como mínimo, un protocolo de red requiere cinco capas [Farrel 04]:

Capa de aplicación Genera o consume información del usuario (mensajes).

Capa de transporte Divide el mensaje en paquetes de información o ensambla paquetes de información según sea el caso. Se encarga también de ejecutar los algoritmos de paridad para prevenir errores y controlar el flujo de datos. Un ejemplo de un protocolo de esta capa es el protocolo TCP antes mencionado.

Capa de red Se encarga del ruteo de los paquetes a través de nodos intermedios de la red. Internet utiliza para esta capa el protocolo IP.

Capa de enlace Transmite y recibe paquetes de información de dispositivo a dispositivo con controles de error. Por ejemplo: Ethernet para redes de área local.

Capa física Transmite y recibe paquetes de bits. Como ejemplo podemos mencionar los cables UTP, coaxiales o de fibra óptica.

Para que un mensaje atraviese las capas del protocolo de comunicación y llegue a su destino, depende absolutamente de equipos ruteadores que le indiquen hacia dónde debe dirigirse (Figura 2.9).

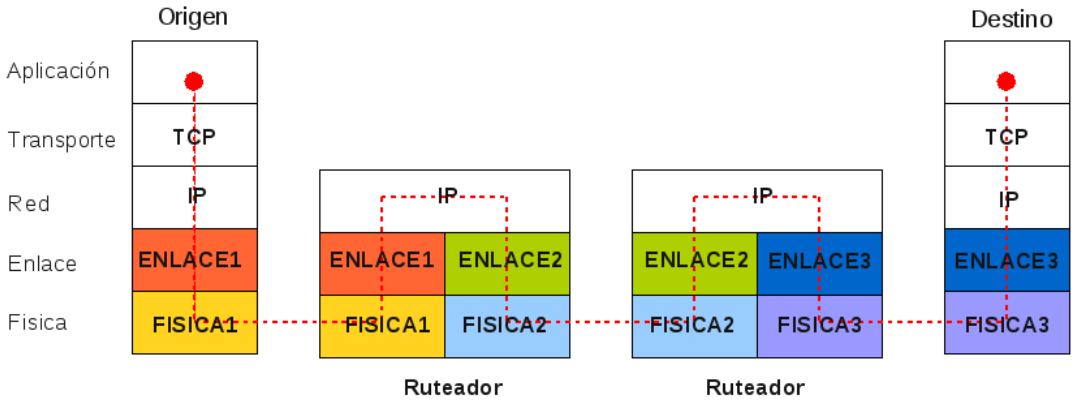

Figura 2.9: Trayectoria de un paquete de información en Internet a través de las capas del protocolo.

El conjunto de protocolos TCP/IP, el esquema de capas en dichos protocolos y el ruteo de información en Internet fueron construidos bajo cuatro principales suposiciones:

Trayectorias bidireccionales continuas entre nodos La conexión entre los nodos de una red debe ser continua y bidireccional para que exista interacción entre los nodos.

Intercambio de datos de corta duración Debe existir un pequeño y relativamente consistente retraso entre el envío de un paquete de información y su respectivo reconocimiento de recepción.

Velocidades simétricas La velocidades de envió entre fuente y destino deben ser similares.

Baja tasa de incidencia de errores Debe existir muy poca corrupción de datos.

2.4.2. Arquitectura de red en redes con tolerancia al retraso

Las redes que operan en entornos en donde estas cuatro suposiciones no se cumplen, no pueden operar bajo el mismo esquema que Internet y necesitan definir sus propios protocolos de ruteo e intercambio de información de manera que cumplan con las exigencias del entorno [Taniar 09]. Aunque las características de cada una de estas redes son muy distintas entre sí, es común encontrarlas con el nombre genérico de DTN, ya que cualquier incumplimiento en las suposiciones arriba mencionadas se traduce en un retraso en la red. Un ejemplo muy estudiado de este tipo de redes son las redes de cómputo móvil en donde la posición de cada nodo varía y con ella la conectividad de la red. Para este tipo de redes, es necesario implementar protocolos que se ajusten tanto a la forma en la que se desplazan los dispositivos como al tipo de conectividad disponible.

Un tipo de protocolo ampliamente utilizado por las DTN para sortear el problema de la conectividad intermitente es el método de almacenamiento y reenvio ("*store and forward switching*"). Bajo este esquema, el mensaje original se divide en bloques de información que son desplazados y almacenados temporalmente en nodos vecinos hasta que dichos paquetes alcancen el nodo destino [Jain 04]. De esta forma, aunque la conectividad de la red se vea interrumpida, el mensaje queda almacenado en distintos punto de la red hasta que la conectividad sea restituida y el mensaje pueda transmitirse con éxito.

Otra herramienta útil, en la que se utilizan los nodos de una red de cómputo móvil, son las conexiones por contactos oportunos. Estos ocurren cuando el emisor y el receptor se encuentran en posibilidades de intercambio de información de manera no programada. Personas o vehículos que se encuentren fortuitamente en un rango en donde es posible intercambiar información es un ejemplo de conexión por contacto oportuno. Normalmente, en una DTN cada nodo es programado para buscar persistentemente el nodo destino del mensaje que carga para así aprovechar los contactos oportunos . En contraparte, si se conoce a priori la trayectoria de los nodos, es posible programar y sincronizar el intercambio de información entre ellos pues se conoce con exactitud la trayectoria de los dispositivos y el instante en el que se encontrarán a una distancia suficiente para intercambiar datos (Figura 2.10).

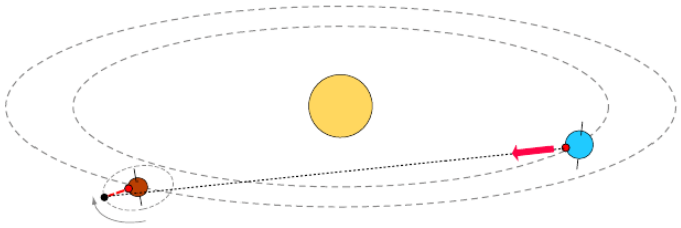

Figura 2.10: Ejemplo de una red móvil con trayectoria predecible [Warth 03]

Las capas del protocolo de red para una DTN también son distintas a las del Internet convencional y en muchas redes no existe distinción entre nodos y ruteadores ya que cada nodo debe realizar las funciones de un ruteador.

La forma en la que los nodos se desplazan juega un papel fundamental en el desempeño de la red. El tipo de movilidad puede servir como punto de partida para diseñar un protocolo de ruteo a modo o, si el protocolo ya está definido, la movilidad de los nodos influye directamente en la frecuencia de contactos oportunos y, en consecuencia, en el desempeño del protocolo[Sung 09]. En cuanto al gasto de energía en una DTN, la movilidad también juega un papel crucial consideremos el escenario de la figura 2.11:

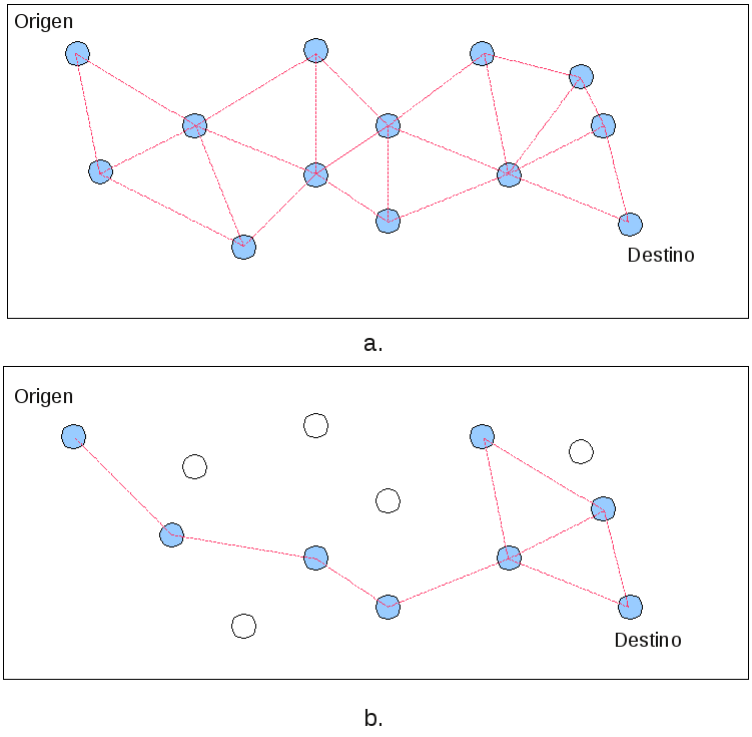

Figura 2.11: a) Ejemplo en el que la movilidad y densidad de una DTN permiten una gran cantidad de contactos oportunos. b) Es posible conservar la conectividad de la red aún cuando muchos dispositivos estén inactivos.

Dado que la movilidad y la densidad de la red de nodos permite una gran cantidad de contactos oportunos, es posible que los dispositivos se activen intermitentemente sin interrumpir la conectividad de la red. De esta forma es posible ahorrar energía en los dispositivos.

Capítulo 3

Definición del Modelo

Como puede advertirse en el Marco Teórico, el modelo que usaremos para simular nuestra red de agentes de cómputo móvil es un Autómata Celular Móvil que, como cualquier ACM convencional consta de tres elementos constitutivos: la definición espacial, la dinámica de movimiento y la ecuación de estados o regla de transición.

3.1. Definición espacial

Para representar espacialmente la red de cómputo móvil utilizaremos una matriz bidimensional Z, en donde cada dispositivo puede ocupar una celda. El tamaño de la matriz representa el área de cobertura de la red a simular y cada celda de dicha matriz puede ser ocupada por un dispositivo que forme parte de la red. La comunicación directa entre dispositivos sólo podrá realizarse si éstos ocupan celdas contiguas. Para nuestro modelo hemos considerado que los dispositivos están confinados en la matriz de simulación, si la posición del siguiente estado de un dispositivo excede los limites de la matriz, dicho dispositivo deberá recalcular la ubicación hasta encontrar una posición libre al interior de la matriz. Para el presente trabajo hemos considerado sólo matrices cuadradas $Z_{n \times n}$

3.2. Dinámica de movimiento

Como mencionamos en la introducción, uno de nuestros principales objetivos es medir la complejidad de una DTN de acuerdo al tipo de movimiento que presentan sus elementos. Para ello hemos decidido analizar tres casos de movilidad:

Movimiento Moore Este tipo de movimiento, también llamado movimiento de vecinos más cercanos, consiste en el desplazamiento aleatorio hacia cualquiera de las ocho celdas contiguas a la celda ocupada en el tiempo t:

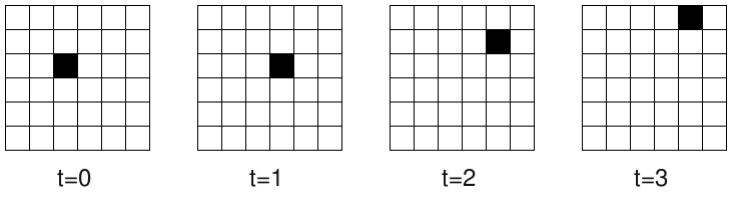

Figura 3.1: Secuencia de movimiento Moore de un dispositivo en la red.

Movimiento mixing Cuando los elementos de un ACM pueden desplazarse aleatoriamente hacia cualquier celda, se dice que tienen una movilidad de "total mixing". Este caso, aunque plantea un movimiento de muy alta difusividad es poco realista, ya que casi nada en el mundo físico se mueve de esta forma. En contraparte, existe el tipo de movimiento "mixing" parcial, más acorde a un escenario real, que combina movimientos aleatorios hacia cualquier celda con movimientos aleatorios hacia las celdas más cercanas

(movimiento Moore). La frecuencia con la que se realiza cada uno de estos dos movimientos está definida por un umbral de ocurrencia que en nuestro caso corresponde al 50 %.

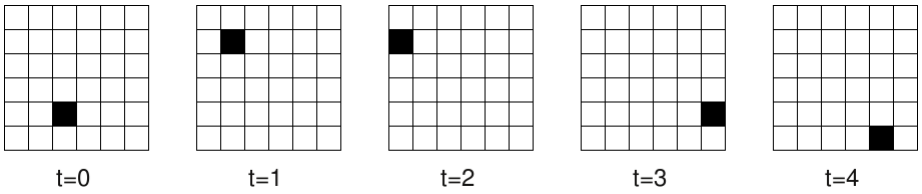

Figura 3.2: Secuencia de Movimiento Mixing de un dispositivo en la red con un umbral de ocurrencia de 50 %, la mitad de las veces el dispositivo se moverá por movimiento Moore y la mitad por movimiento "total mixing".

Movimiento por vuelos de Lévy De acuerdo a las definiciones hechas en la sección 2.2, en este tipo de movimiento las longitudes de desplazamiento de los dispositivos corresponden a una distribución de Lévy. Para simular este tipo de movimiento, los dispositivos de nuestro programa primero determinarán una longitud acorde a dicha distribución y después determinarán aleatoriamente una de las 8 posibles direcciones que pueden tomar.

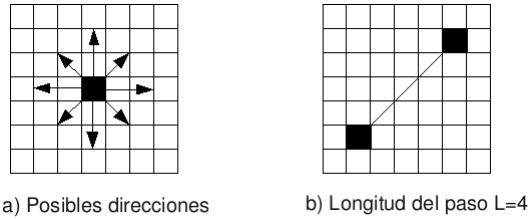

Figura 3.3: a) Representación de las posibles direcciones que puede tomar un dispositivo que se mueve por vuelos de Lévy. b) Representación de un paso de 4 unidades en dirección Noreste.

Como la representación de la matriz está hecha digitalmente, las unidades de desplazamiento tienen la misma longitud que una celda de la matriz, así, un desplazamiento de 4 unidades al norte, representa un desplazamiento de 4 celdas al norte. Esto ocasiona, naturalmente, que los desplazamientos en diagonal abarquen una mayor longitud real. Aunque esto resulta muy notorio en matrices pequeñas, en matrices grandes este efecto tiende a ser casi imperceptible y por lo tanto despreciable.

3.3. Ecuación de estados

La ecuación de estados que usaremos en nuestro experimento está basada en el trabajo de Miramontes, Goodwin y Solé acerca de las oscilaciones de actividad en sociedades de hormigas [Solé 93]. En dicho trabajo, los autores utilizan un AC para representar el intercambio de información entre hormigas de la especie *Leptothorax*. Aquí, tomaremos la misma dinámica de intercambio de información y la aplicaremos en un entorno en el que la movilidad de los elementos de la red cubren los tres casos mencionados en la sección anterior.

Comenzaremos definiendo la actividad de un dispositivo:

La actividad $e \in \mathbb{R}$ de un dispositivo presente en la red es un valor del rango [0,1] que indica que un dispositivo está en condiciones de intercambiar información con cualquier vecino contiguo. En nuestro modelo, la actividad de los dispositivos se refiere sólo a su capacidad de intercambio de información ya que tanto un dispositivo activo como inactivo presentan movilidad. Así, un dispositivo activo es aquel que tiene una actividad $e > 0$ y un elemento inactivo el que tiene una actividad $e = 0$.

La ecuación de transición de actividad de los dispositivos está dada por:

$$S_i^{t+1} = tanh\left\{gh_k(t)\right\}, \tag{3.1}$$

que es una ecuación típica de un sistema Hopfield (Véase Sección 2.1.4) que actualiza el estado del i-ésimo dispositivo a partir del estado de los dispositivos vecinos $h_k(t)$ para un tiempo t. $h_k(t)$ puede expresarse como

$$\sum_{j=1}^{k} J_{ij} S_j^t + J_{ii} S_i^t, \tag{3.2}$$

en donde J_{ij} representa la matriz de acoplamiento, $\sum_{j=1}^{k} J_{ij} S_j^t$ la suma de los estados de los dispositivos vecinos y $J_{ii} S_i^t$ el valor actual del elemento multiplicado por la matriz auto-interacción.

Al sustituir la ecuación 3.2 en 3.1 obtenemos:

$$S_i^{t+1} = tanh \left\{ g \left(\sum_{j=1}^{k} J_{ij} S_j^t + J_{ii} S_i^t \right) \right\}. \tag{3.3}$$

Al ser nulo el valor de interacción entre dispositivos ($J_{ii} = 0$) y al considerar la matriz de acoplamiento unitaria ($J_{ij} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$), la ecuación 3.3 se transforma en

$$S_i^{t+1} = tanh \left\{ g \left(S_i^t + \sum_{j=1}^{k} S_j^t \right) \right\}, \tag{3.4}$$

que es la ecuación que usaremos de aquí en adelante para actualizar el estado de los dispositivos en la red.

El parámetro de no-linealidad o ganancia g, ha sido experimentalmente fijado en 0.05, ya que en este valor ha mostrado tener una interesante transición entre el orden y el caos de acuerdo a la densidad de elementos en la red [Solé 95] (Figura 3.4).

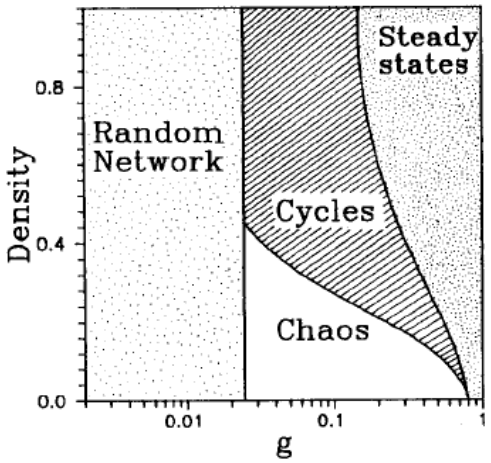

Figura 3.4: Estados dinámicos del sistema (ecuación 3.4) de acuerdo a la variación del parámetro de ganancia g y la densidad de elementos en la red [Solé 95].

De acuerdo a la ecuación 3.4, un individuo con actividad cero puede ser activado al entrar en contacto con un individuo activo que se encuentre en su vecindad, a este tipo de activaciones se le conoce como activaciones inducidas.

El modelo de Miramontes *et al.* contempla la propiedad de actividad espontánea, misma que ha probado ser crucial en la dinámica del modelo [Solé 93]. Esta propiedad contempla que un elemento de la red puede activarse espontáneamente con una probabilidad fija de 0.01.

Al fijar el coeficiente de no linealidad g en 0.05 y comparar diferentes densidades en la red, Miramontes encuentra oscilaciones en la actividad del hormiguero a pesar de que este último no está compuesto por elementos oscilantes, es decir, las oscilaciones en el hormiguero son una propiedad emergente del autómata celular que modela al hormiguero. Así, es posible observar una transición entre el caos y el orden en estas oscilaciones conforme aumenta la densidad de la red (Figura 3.5).

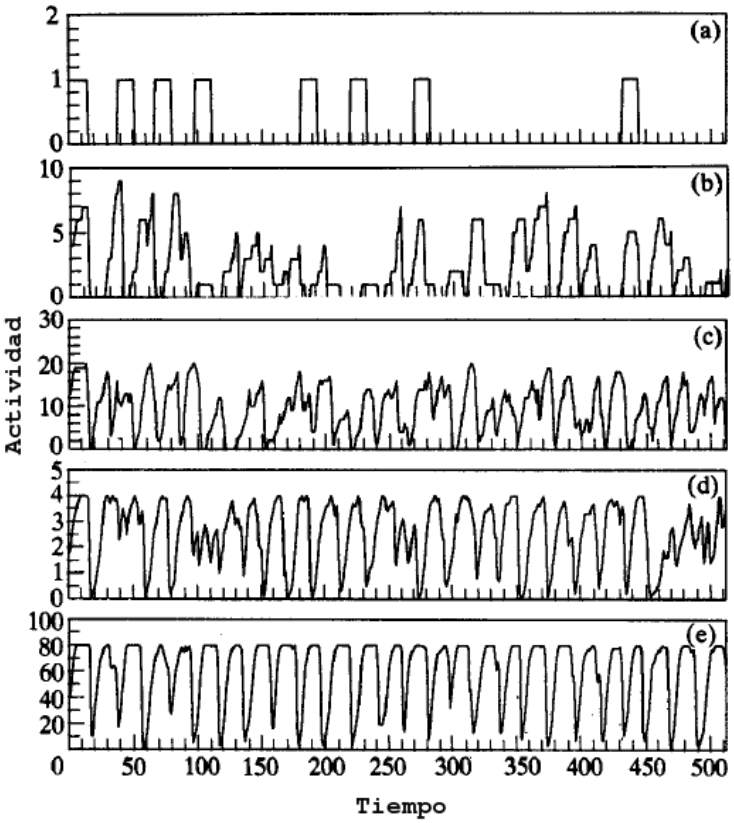

Figura 3.5: Oscilaciones coherentes obtenidas incrementado el numero de hormigas incluidas en una matriz de 10x10. Mostramos en a) la actividad de N=1(una sola hormiga); b) N=10; c) N=20; d) N=40 y e) N=80. Conforme aumenta N, y con ella la densidad de la colonia, el comportamiento colectivo periódico aumenta [Solé 95] .

En un artículo posterior [Miramontes 95], Miramontes encuentra que el punto de mayor entropía en el modelo está cerca del valor de densidad de 20 %. En este valor, la colonia alcanza la máxima expresión del comportamiento cooperativo.

Capítulo 4

Desarrollo

La implementación del autómata celular que modela la red móvil consta de dos partes: la primera, programada en Python, cuenta con una interfaz visual que permita al usuario modificar fácilmente los parámetros de entrada, visualizar la actividad de los dispositivos en la red y muestra una animación de la matriz de estados que alberga los dispositivos de la red. La segunda, programada en lenguaje C, carece de interfaz visual pero, al ser un lenguaje compilado, contará con la rapidez y estabilidad de ejecución necesarias para simular grandes redes.

Al ser grandes ensambles de elementos idénticos, los autómatas celulares resultan ser sencillos de programar y modelar bajo un esquema orientado a objetos. En este trabajo utilizaremos este esquema sólo en la implementación en Python ya que ofrece una sintaxis altamente versátil para este tipo de programación. En la implementación en C utilizaremos un enfoque eminentemente estructurado para conservar la sencillez y claridad en el código.

4.1. Implementación en Python

4.1.1. Núcleo del programa

4.1.1.1. Definición de clases

La forma más sencilla de esquematizar el modelo es suponer dos clases: (a) la clase "Nodo" que almacena las propiedades de cada dispositivo (posición y actividad) y (b) la clase "Red" que almacena las reglas del autómata celular, los tipos de movimiento de cada nodo y las funciones necesarias para medir los parámetros de complejidad. Una vez construidas estas clases, que conforman en núcleo del programa, es posible programar una interfaz visual que simplemente muestre la operaciones y los estado producidos por el núcleo.

Comenzaremos definiendo la clase Nodo (Figura 4.1).

```
 8  class Nodo:
 9  »   def __init__(self, x=0, y=0, e=0):
10  »   »   self.x=x; self.y=y; self.e=e
11  »   »
12  »   def __str__(self):
13  »   »   return '('+ str(self.x) + ',' + str(self.y) + ',' + str(self.e) + ','  + ')'
```

Figura 4.1: Clase Nodo.

Esta sencilla clase nos permite definir una posición (x,y) y un valor de actividad (e) para cada nodo de la red. El método __str__ muestra los atributos de los nodos y será muy útil para conocer el estado de los nodos en la red.

La clase Red contiene como atributos la matriz bidimensional y los parámetros definidos en el modelo:

```
class Red:
    def __init__(self, maxx, maxy, num): # tamaño de la matriz y numero de nodos
        self.maxx=maxx; self.maxy=maxy; self.num=num;
        self.pac=0.01       #probabilidad de activacion espontanea
        self.g=0.05         #parametro de no-linealidad
        self.umbral_cero=1e-16 #valor practico de 0
        self.umbral=0.5     #este es el umbral para movimiento moore y mixing
        self.conact=1e-6;   #valor inicial de actividad en los nodos
        self.nodos=[]       #arreglo de elementos de la clase Nodo
        self.newstate=[]    #arreglo auxiliar de estados actualizados
        self.maxxmu=self.maxx+2 ; self.maxymu=self.maxy+2;
        self.grid=[[0 for col in range(self.maxxmu)] for row in range(self.maxymu)]
        for i in range(self.maxxmu):
            for j in range(self.maxymu):
                self.grid[i][j]=-2.0
        for i in range(self.num):
            while True:
                tx=int(random.random()*self.maxx)+1
                ty=int(random.random()*self.maxy)+1
                if self.grid[tx][ty]==-2.0: break
            self.grid[tx][ty]=self.conact
            self.nodos.append(Nodo(tx,ty,self.conact))
            self.newstate.append(1)
```

Figura 4.2: Método “init” de la clase Red.

De la figura 4.2 podemos observar que “grid” es el arreglo bidimensional que representa la matriz del AC. En el primer ciclo “for” se inicializan todos los elementos de la matriz a -2.0, indicando ausencia de dispositivos. El segundo ciclo coloca aleatoriamente los dispositivos en la red con un valor inicial de activación igual a la variable “conact”. El arreglo “nodos” almacena los objetos de la clase Nodo y permite almacenar tanto la posición como el valor de actividad de los nodos de la red.

4.1.1.2. Movimiento de los nodos

Los tres tipos de movimiento contemplados para los nodos de la red se encuentran implementados como métodos de la clase Red. Estos métodos reciben como parámetros un entero i que representa el índice del Nodo en el arreglo “nodos” y actualizan la posición de este en la matriz, de acuerdo al tipo de movimiento que se trate.

```
    def transporta(self, i):
        if self.nodos[i].e > self.umbral_cero:
            j=0
            while True:
                j=j+1
                nouveaux=random.randint(1,self.maxx)
                nouveauy=random.randint(1,self.maxy)
                if(self.grid[nouveaux][nouveauy] == -2.0) or (j==6): break
            if j<6:
                self.grid[self.nodos[i].x] [self.nodos[i].y]= -2.0
                self.nodos[i].x=nouveaux
                self.nodos[i].y=nouveauy
                self.grid[self.nodos[i].x][self.nodos[i].y]= self.nodos[i].e
```

Figura 4.3: Método “transporta".

En la figura 4.3 se muestra el método “transporta”, que contiene las reglas para ejecutar el movimiento “total mixing” definido en la sección 3.2. En esencia, este algoritmo escoge un par de coordenadas aleatorias entre el 0 y el tamaño de la matriz, si dichas coordenadas se encuentran libres asigna esta nueva posición al nodo i, de lo contrario, busca otras coordenadas aleatorias disponibles un máximo de 6 ocasiones. El tipo de movimiento Moore es muy parecido y conserva la misma estructura, la diferencia radica en que en este último los dispositivos sólo pueden moverse a celdas contiguas. El caso del movimiento de Lévy es un tanto distinto ya que antes de definir la dirección del desplazamiento es necesario calcular la longitud del mismo, para ello se utiliza un algoritmo estándar que regresa un valor de longitud acorde a la distribución de Lévy (Figura 4.4).

```
90      def vlevy(self, vsize):  #esta funcion determina la longitud del vuelo de levy
91          b=2                  #dada una distancia especifica
92          c=pe=0
93          p=[]
94          for i in range(1,vsize+1):
95              p.append(0)
96          for i in range(1,vsize+1):
97              c = c+i**(-b)
98          for i in range(1,vsize+1):
99              pe = pe+((i**(-b))/c);
100             p[i-1] = pe
101         ran=random.random()
102         k=0
103         while True:
104             k=k+1
105             if ran <= p[k-1]: break
106         return k
```

Figura 4.4: Algoritmo estándar para calcular la longitud de un paso de acuerdo a la proporción de Lévy.

4.1.1.3. Ecuación de estados

La ecuación de estados se encuentra implementada en el método "calcampo" que recibe un entero i que indica el índice del arreglo "nodos" en donde se encuentra el nodo a actualizar (Figura 4.5).

```
73      def calcampo(self,i):
74          cx=int(self.nodos[i].x)
75          cy=int(self.nodos[i].y)
76          campo=0
77          for j in range(cx-1,cx+2):
78              for k in range(cy-1,cy+2):
79                  if self.grid[j][k] >= -1:
80                      campo = campo + self.grid[j][k]
81          self.newstate[i]=math.tanh(campo*self.g)
82          if self.newstate[i] < self.umbral_cero:
83              self.newstate[i]=0
84          rand=random.random()
85          if rand <= self.pac:
86              if self.newstate[i]<=0:
87                  self.newstate[i]=self.conact
```

Figura 4.5: Método calcampo. Aplica la ecuación 3.4 a partir de los estados de los nodos vecinos.

La variable "campo", que se encuentra dentro del ciclo "for" anidado, suma las actividades de los elementos vecinos. Posteriormente, el resultado se multiplica por el factor g y se calcula la tangente hiperbólica. Este resultado se almacena en el arreglo "newstate" con el mismo índice que en "nodos".

4.1.1.4. Iteraciones

Por cada instante de tiempo o iteración de nuestra simulación es necesario actualizar el estado de actividad y posición de todos los dispositivos de acuerdo al tipo de movilidad escogida, para ello, implementamos métodos de la clase "Red" que ejecutan todas las funciones necesarias para ejecutar una iteración completa en la simulación. En esta sección, sólo mostraremos el método "activamoore" (Figura 4.6) aunque existen sus contrapartes "activamixing" y "activalevy".

```
150     def activamoore(self):
151         for i in range(self.num):
152             self.calcampo(i)
153         for i in range(self.num):
154             self.nodos[i].e=self.newstate[i]
155             self.grid[self.nodos[i].x][self.nodos[i].y]=self.newstate[i]
156             self.mueve(i)
157
```

Figura 4.6: Algoritmo que ejecuta una iteración completa de la simulación con movimiento Moore.

En este método, primero se calcula el estado de actividad siguiente de todos los dispositivos de la red utilizando el método "calcampo". Después se desplaza a todos los dispositivos utilizando el método "mueve" que utiliza el método de desplazamiento Moore.

4.1.1.5. Algoritmos de complejidad

Como mencionamos en la introducción, nuestro interés se centra en medir dos parámetros de complejidad en el modelo de red: la entropía de Shannon y la complejidad algorítmica de Kolmogorov.

Complejidad algorítmica de Kolmogorov La CAK está definida a partir de cadenas binarias, lo que significa que es necesario expresar la actividad de los nodos en la red como una sucesión de unos y ceros. Para ello sólo es necesario medir si más de la mitad de los dispositivos están activos en un instante de tiempo. Si es el caso, expresaremos la actividad global de la red con 1, de lo contrario la actividad será 0. Una vez expresada la actividad de la red como elementos binarios, es posible aplicar el algoritmo definido por Lempel y Ziv (véase sección 2.3.2) para calcular CAK.

```
192    def complejidad(self, s):
193        t=s
194        control=self.num/2
195        print 'control ',control
196        for i in range(len(t)):
197            if t[i]>control:
198                t[i]=0
199            else:
200                t[i]=1
201        c=1
202        l=1
203        while True:
204            i=0
205            k=1
206            kmax=1
207            while True:
208                while t[i+k-1]==t[l+k-1]:
209                    k=k+1
210                    if l+k<len(t):
211                        continue
212                    else:
213                        c=c+1
214                        return c
215                if k>kmax:
216                    kmax=k
217                i=i+1
218                if i==l:
219                    break
220                k=1
221            c=c+1
222            l=l+kmax
223            if l+1>len(t):
224                return c
```

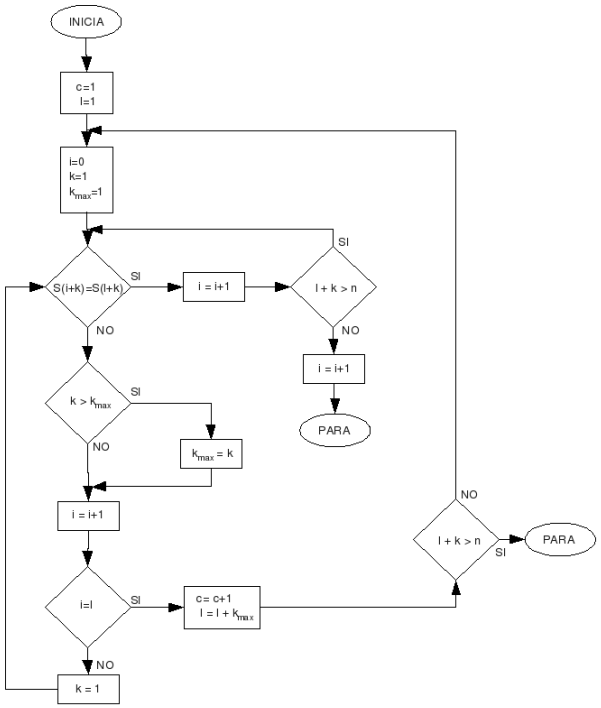

Figura 4.7: Diagrama de flujo del algoritmo de Lempel y Ziv y su correspondiente implementación en Python.

El método "complejidad" recibe una cadena "s" de enteros y entrega la variable "c" como resultado del cálculo de CAK. La cadena "s" contiene el número de dispositivos activos para cada instante de tiempo que duró la simulación. La primera parte de este método transforma la cadena "s" en valores binarios de acuerdo a los criterios arriba mencionados. El algoritmo de complejidad comienza propiamente en la línea 201(Figura 4.7). Es posible apreciar que es un algoritmo completamente estructurado, es decir que no utiliza funciones que alteren la secuencia de ejecución. También es importante señalar que la implementación del algoritmo requiere de ciclos del tipo "do/while". En Python, estos ciclos no existen propiamente, son implementados a partir de ciclos "while" del tipo "while True" como puede apreciarse en las líneas 203 y 207. El valor "c" que regresa este método corresponde a la CAK no normalizada (véase sección 2.3.2.2).

Entropía de Shannon Para calcular la entropía de Shannon es necesario conocer las probabilidades con la que n dispositivos estuvieron activos durante toda la simulación. Para ello requerimos el arreglo "s" que almacenó el número de dispositivos activos por cada iteración. Basta sumar todas las veces en las que n dispositivos estuvieron activos y después dividirlas entre el total de iteraciones para obtener el arreglo de probablilidades p. Veamos un ejemplo: sea $s = [\ 3 \quad 1 \quad 3 \quad 2\ ]$ una matriz de dispositivos activos. Representa que en el instante

$t = 1$ hubo 3 dispositivos activos, en el instante $t = 2$ hubo un elemento activo y así sucesivamente. Para obtener las probabilidades p_i de cada número de dispositivos activos basta contar la cantidad de veces en las que hubo n dispositivos activos y dividirla entre el total de iteraciones (en nuestro ejemplo cuatro iteraciones) así:

$p_1 = 1/4$ pues hubo sólo una ocasión en la que hubo un dispositivo activo.

$p_2 = 1/4$ pues hubo sólo una ocasión en la que hubo dos dispositivos activos.

$p_3 = 2/4$ porque en dos ocasiones hubo 3 dispositivos activos.

Una vez obtenidas todas la probabilidades, se aplica la ecuación $-\sum_{i=1}^{n} p_i \log_2 p_i$ (véase sección 2.3.1) para calcular la entropía de Shannon. En el ejemplo de arriba, la entropía será de: $\Lambda^{(3)}(p_1, p_2, p_3) = -(2(\frac{1}{4}\log_2\frac{1}{4})+\frac{1}{2}\log_2\frac{1}{2})=1.5$ bits.

En el código de la figura 4.8 podemos observar que el método "entropía" recibe el arreglo "s" como entrada. El primer ciclo "for" permite contar las veces en las que n dispositivos estuvieron activos y almacena la información en el arreglo "hist". El segundo ciclo calcula las probabilidades p_i a partir del arreglo "hist" y calcula la entropía.

```
226    def entropia(self, s):
227        entropia=0
228        size=max(s)
229        hist=[0.0 for p in range(size+1)]
230        for i in range(len(s)):
231            hist[s[i]-1]=hist[s[i]-1]+1
232
233        for i in range(size+1):
234            if hist[i]!=0:
235                pi=hist[i]/len(e)
236                entropia=entropia+pi*math.log(pi)
237        entropia=-entropia
238
239        return entropia
```

Figura 4.8: Algoritmo para calcular la entropía de Shannon.

Los algoritmos anteriores completan el conjunto de operaciones necesarias para realizar la simulación completa. En la siguiente sección explicaremos el funcionamiento de la interfaz visual y no será necesario analizar el código con tanto detenimiento pues la interfaz actúa sólo como una envoltura que permite recabar los datos que ejecuta el núcleo del programa.

4.1.2. Interfaz Visual

4.1.2.1. wxPython

Como conjunto de herramientas visuales hemos escogido wxPython por la sencillez con la que pueden construirse interfaces robustas [Rappin 06]. En realidad, wxPython es un conjunto de referencias ("*binding*") al popular kit visual wxWidgets originalmente escrito en C++ [Rappin 06]. Este kit permite crear "*widgets*" (elementos visuales) en un entorno completamente orientado a objetos y con la capacidad de correr en múltiples plataformas. Cuenta también con la librería "plot" que permite realizar gráficas de forma sencilla y con muchas opciones de configuración. wxPython hereda también la estabilidad en el manejo de eventos característica de wxWidgets y, al ser un entorno orientado a objetos, resulta muy sencillo incorporar una gran cantidad de eventos a cada widget.

4.1.2.2. Panel principal

La figura 4.9 nos proporciona una idea general de lo que nuestro programa debe realizar.

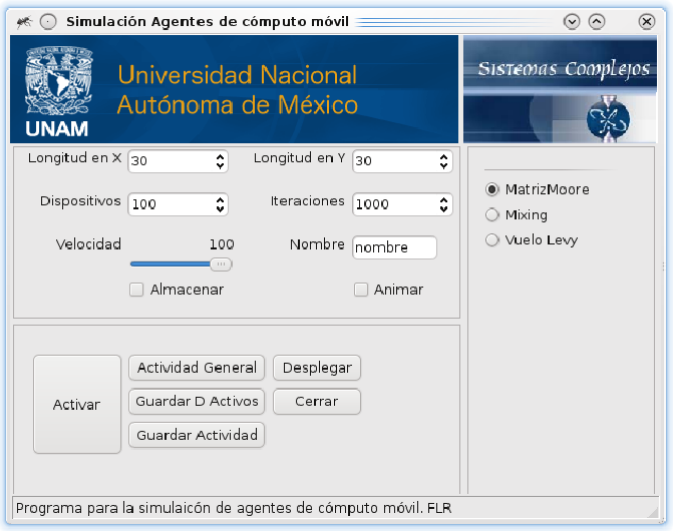

Figura 4.9: Panel principal del programa de simulación.

Panel de inserción de datos El primer panel en la esquina superior izquierda nos permite introducir parámetros necesarios para la simulación como el tamaño de la matriz, el número de dispositivos a colocar y el número de iteraciones. El primer *checkbox* marcado con la etiqueta "almacenar" nos permitirá guardar en el disco duro todas las imágenes de la simulación a realizar, esto resulta muy útil cuando simulamos matrices muy grandes. En estos casos, mostrar el estado actual de la red tardaría demasiado tiempo debido a la gran cantidad de operaciones a realizar y la animación sería poco fluida. Pero, si almacenamos cada estado de la red como una secuencia de imágenes, es posible ver la animación fluidamente si primero almacenamos todos los cuadros de la simulación y después los desplegamos a la velocidad deseada. El campo "nombre" permite agrupar todas las imágenes correspondientes a una simulación bajo el mismo nombre.

El segundo *checkbox* habilita la opción "animar" que despliega el estado de la matriz en tiempo real. La barra de velocidad controla la rapidez con la que se desea desplegar los cuadros que representan el estado de la red. Si la matriz es pequeña, las operaciones serán pocas y la animación de la red puede ser demasiado rápida para apreciar la evolución de la misma, esta barra permite modular la velocidad en estos casos. El panel derecho nos permitirá escoger, por medio de un *radio button*, el tipo de movimiento a simular.

Panel de acciones En el panel inferior derecho se encuentran los botones que controlan las acciones necesarias, "activar" es el botón más importante pues con él comienza la simulación a partir de los datos introducidos en el panel superior. "Activar" mostrará una gráfica en tiempo real del número de dispositivos activos en cada instante, desplegará una animación de la red en tiempo real y, al final de la simulación, entregará los resultados obtenidos de complejidad y entropía.

El botón "Actividad General" desplegará una gráfica secundaria en donde, en lugar de mostrar el número de dispositivos activos en cada instante, muestra la suma total de actividad en la red, es decir, suma la actividad individual de todos los dispositivos en la red por cada instante de tiempo.

El botón "Desplegar" nos permitirá animar una simulación previamente realizada. Esto es: si se realizó una simulación con el checkbox "almacenar" activado y con el nombre "prueba", al colocar el nombre "prueba" en el campo "Nombre" y activar el botón "Desplegar", el programa mostrará una animación de todas las imágenes almacenadas de la simulación "prueba" a la velocidad seleccionada en la barra. En la parte inferior se encuentran los botones "Guardar D Activos" y "Guardar Actividad", ambos permiten almacenar en un archivo de texto los datos de la simulación.

Gran parte del código que requiere la interfaz visual corresponde a funciones convencionales de wxPython que no vale la pena reproducir íntegramente (el código completo se encuentra en la parte de anexos), así que nos limitaremos a explicar en esta sección sólo el funcionamiento de los botones "activar" y "desplegar" ya que constituyen la parte esencial de la simulación.

4.1.2.3. Acción "Activar"

```
132     def AlActivar(self, event):
133         if not self.tc1.GetValue() or not self.tc2.GetValue():
134             return
135         self.data=[];self.data2=[];self.data3=[];self.data4=[]
136         maxxp=int(self.tc1.GetValue())
137         maxyp=int(self.tc2.GetValue())
138         nump=int(self.tc3.GetValue())
139         maxitp=int(self.tc4.GetValue())
140         almacenar=self.tc7.GetValue()
141         animar=self.tc8.GetValue()
142         modalidad=self.radiobox1.GetSelection()
143         self.red=Red(maxxp,maxyp,nump)
144         accion=[lambda: self.red.activamoore(), lambda: self.red.activamixto(),
145         lambda: self.red.activalevy()]
146         frm = wx.Frame(self, -1, 'Actividad', pos=(460,0),size=(750,300))
147         frm.SetIcon(wx.Icon('antgris1.ico', wx.BITMAP_TYPE_ICO))
148         client = plot.PlotCanvas(frm)
```

Figura 4.10: Primera parte del código asociado al botón "Activar".

En la primera línea del método "AlActivar" (Figura 4.10), corroboramos que se hayan introducido por lo menos los dos primeros valores que corresponden al tamaño de la matriz. En la siguiente línea se declaran 4 arreglos necesarios para la simulación. Enseguida obtenemos los datos necesarios: "maxxp" y "maxyp" representan el tamaño de la matriz, "nump" el número de dispositivos a colocar, "maxitp" el número de iteraciones, "almacenar" y "animar" son las variables booleanas de almacenamiento y animación. "modalidad" es el tipo de movimiento escogido. En la siguiente línea se crea el objeto "red" a partir de la clase "Red", definida en la sección anterior. "accion" es un arreglo de métodos que necesitamos asociar de acuerdo al tipo de movimiento escogido, podemos observar que el arreglo está definido a partir de lambdas para poder llamar métodos de la clase Red anónimamente. En la parte inferior creamos un "frame" de wxPython en el que graficaremos la actividad de la red. "client" es un objeto creado por la librería "plot", necesario para realizar gráficas (Figura 4.11).

```
149         if maxxp < 80 or maxyp < 80:
150             self.sizex=maxxp*10+10
151             self.sizey=maxyp*10+10
152             mult=10
153             radio=5
154         elif maxxp >= 80 and maxxp <= 250:
155             self.sizex=maxxp*4+2
156             self.sizey=maxyp*4+2
157             mult=4
158             radio=2
159         else:
160             self.sizex=maxxp+1
161             self.sizey=maxyp+1
162             mult=1
163             radio=1
164         self.frh = wx.Frame(self, -1, 'Animacion', pos=(0,350),
165         size=(self.sizex,self.sizey))
166         self.frh.SetBackgroundColour('Black')
167 
168         wx.EVT_PAINT(self.frh, self.OnPaint)
169         for j in range(maxitp):
170             accion[modalidad]()
171             suma=self.red.Hactivas()
172             suma2=self.red.actividad()
173             self.data.append((j,suma))
174             self.data2.append((j,suma2))
```

Figura 4.11: Continuación del método AlActivar.

En las primeras lineas de esta última figura se hace un escalamiento de pixeles de acuerdo al tamaño de la

matriz solicitada. Si la matriz es menor de 80x80, entonces el tamaño de la matriz será multiplicada por un factor de 10 pixeles para ser representada gráficamente. Los dispositivos serán representados por círculos de 5 pixeles de radio. Si la longitud de la matriz excede las 80 unidades pero es menor que 250, el factor de pixeles es 4 y el radio de los dispositivos será de 2 pixeles. En cualquier otro caso, el factor de escalamiento y radio valen 1, que es el mínimo valor con el que se puede representar un dispositivo en pantalla.

Las siguientes líneas corresponden a la creación del "frame" en donde se dibujará la matriz de dispositivos. Se asigna el color negro como fondo del cuadro y en la línea 166 se asocia el evento "wx.EVT_PAINT" al cuadro "frh", esto significa que cualquier evento de dibujo que ocurra en el programa debe ser realizado sobre el cuadro "frh". En la siguiente línea comienza el ciclo principal de la simulación, este se repetirá tantas veces como el usuario lo halla indicado en el campo "Iteraciones" del panel principal. El primer y más importante paso del ciclo es activar la red, la línea "accion[modalidad]()" es una llamada a un método mediante el arreglo "accion". La variable "modalidad" corresponde a la activación de la red con el tipo de movimiento solicitado, ya sea "activamoore", "activamixto" o "activalevy". Vale la pena recordar que la activación de la red es la suma de dos procesos: a) el cálculo del estado de los dispositivos utilizando el método "calcampo" y b) el desplazamiento de los dispositivos de acuerdo a la modalidad escogida. Así, esta línea resulta ser en sí misma la simulación, el resto del código sirve meramente para extraer información de la misma.

Una vez terminada la activación de la red, los dispositivos se encuentran en una nueva distribución de donde es necesario extraer información del estado de la red. La variable "suma" nos permite obtener el número de dispositivo activos en la nueva distribución mediante el método "Hactivas", la variable "suma2" almacena la suma de las actividadades individuales utilizando el método "actividad". Estos valores son almacenamos en los arreglos "data" y "data2" para poder calcular, al final de la simulación, la complejidad y la entropía.

```
174                self.data4.append(suma)
175                line = plot.PolyLine(self.data, legend='legend', colour='black', width=1)
176                gc = plot.PlotGraphics([line], 'Dispositivos Activos', 'Iteracion'
177                , '# de agentes activos')
178                client.Draw(gc, xAxis=(0, maxitp), yAxis=(0,nump+0.1*nump))
179                self.show_bmp = wx.StaticBitmap(self.frh)
180                self.draw_image(nump, mult, radio)
181                if almacenar==True:
182                    self.save_image(j)
183                if animar==True:
184                    self.OnPaint()
185                    show=True
186                else:
187                    show=False
188
189                frm.Show(True)
190                self.frh.Show(show)
191            p=self.red.complejidad(self.data3)
192            p=p/(maxitp/math.log(maxitp,2))
193            q=self.red.entropia(self.data4)
194            #Comienza despliegue de entropia y complejidad
195            frmcomplex = ComplexFrame(p,q)
196            frmcomplex.Show(True)
197
```

Figura 4.12: Tercera y última parte del método AlActivar.

La variable "line" es un objeto del tipo "PolyLine" que almacena los datos a graficar, utiliza como primer parámetro "data" ya que es el arreglo que contiene la información de los dispositivos activos. Las líneas 177 y 178 de la figura 4.12 se encargan de desplegar la gráfica en el cuadro "frm". Como estas instrucciones se encuentran dentro del ciclo principal de simulación, la gráfica de los dispositivos activos por unidad de tiempo se actualiza en cada iteración, mostrando una gráfica en tempo real. La línea 179 crea un mapa de bits en el cuadro "frh", esto prepara el cuadro para recibir pixeles coordenados de dibujo y así poder graficar la matriz. "draw_image" es el método de graficación más importante, recibe como parámetros de entrada el número de dispositivos en la red, el parámetro de multiplicación de pixeles "mult" y el radio en pixeles que debe tener cada dispositivo. Con estos datos, "draw_image" se encarga de representar la matriz de la red como un cuadrado y cada dispositivo como un circulo (los activos de color rojo y los inactivos de color blanco) en su respectiva posición.

Si la opción "almacenar" está activada se ejecuta el método "save_image", análogo a "draw_image" que, en lugar de dibujar sobre el cuadro "frh", obtiene el mapa de bits creado en la línea 179 y lo almacena en memoria como una imagen tipo "tif". El nombre de cada imagen se construye con la concatenación del nombre de la simulación y el índice de la iteración en la que se creó. Así, si el nombre de la simulación es "prueba" y se

solicitaron 50 iteraciones, la lista de imágenes creadas será: “prueba1.tif, prueba2.tif, ... , prueba50.tif”. De esta manera será muy fácil recuperar la secuencia de las imágenes y mostrarlas a la velocidad deseada.

Si la opción “animar” está activa, la variable “show” toma el valor “Verdadero” y se muestra el cuadro “frh” al final de cada iteración, de lo contrario la animación se desactiva y sólo se muestra la gráfica de dispositivos activos.

Una vez terminado el ciclo principal, se cuenta con la información completa de la simulación y es posible calcular la entropía y la complejidad utilizando los arreglos “data3” y “data4” que almacenaron la cantidad de agentes activos en cada instante de tiempo. “p” almacena la complejidad de la simulación llamando al método “complejidad”, contenido en nuestra instancia de la clase “Red”. Enseguida se divide el resultado entre el parámetro de normalización de CAK (véase sección 2.3.2.2 en la página 22) para poder compararla con una cadena binaria completamente aleatoria. Análogamente, obtenemos la entropía de Shannon utilizando el método “entropía”. Por último creamos un cuadro de despliegue llamado “frmcomplex” para mostrar los resultados de CAK y entropía.

Una vez concluida la programación del método “AlActivar” y sus funciones asociadas, es posible realizar cualquier simulación. La figura 4.13 muestra un ejemplo de una simulación completa de una matriz de 30x30 con 100 dispositivos, 1000 iteraciones y tipo de movimiento Mixing.

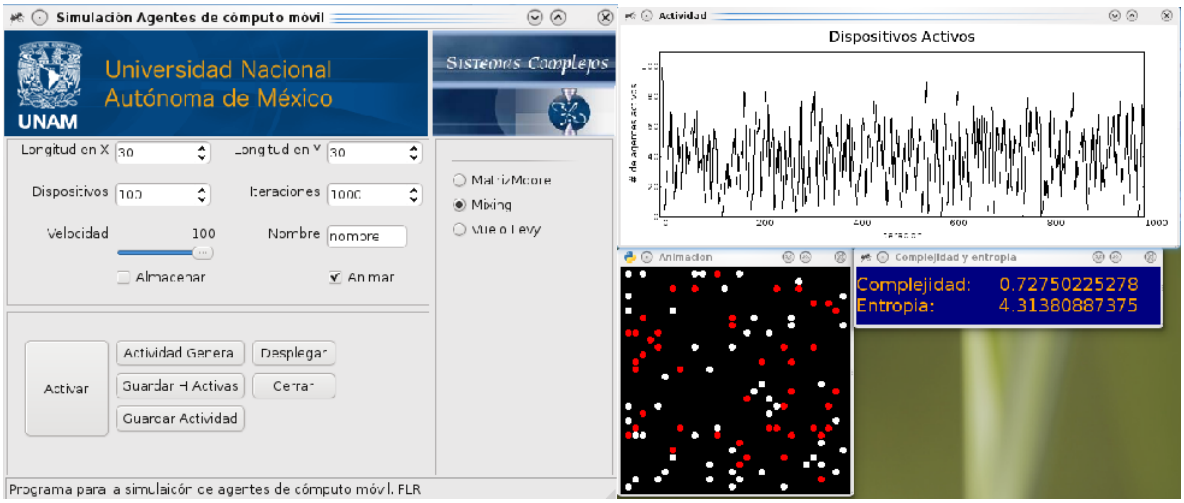

Figura 4.13: a) Panel de la Primera simulación b) Resultados de la simulación.

La simulación convencional cuenta entonces con tres cuadros, la gráfica de dispositivos activos por unidad de tiempo, el panel de animación y el recuadro que muestra, al final de la simulación, la complejidad y la entropía. En el cuadro de animación de la figura es posible observar como cada dispositivo está representado por un circulo de 5 pixeles de radio, rojo si está activo, blanco si está inactivo. Si la matriz fuera mayor de 80 unidades, el programa ajustaría el tamaño de la matriz y de los dispositivos para poder ser desplegados en pantalla.

En caso de que se desee conocer la suma de actividades de los dispositivos, basta con apretar el botón “Actividad General” al término de la simulación y el programa desplegará un cuadro que contiene dicha gráfica (Figura 4.14)

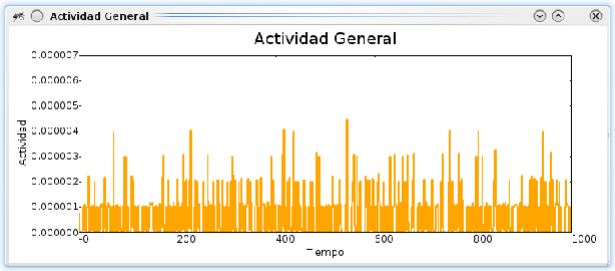

Figura 4.14: Cuadro que despliega la gráfica de actividad general.

4.1.2.4. Acción "Desplegar"

Existen dos funciones que intervienen en la acción de desplegar imágenes previamente almacenadas. "Al Desplegar" obtiene el tamaño y número de imágenes a desplegar del panel de inserción de datos, crea el "wxFrame" necesario para mostrar la animación y llama a la segunda función "OnDisplay", que actúa como puente entre las imágenes almacenadas y el "wxFrame" creado para reproducir la animación (Figura 4.15).

```
253     def AlDesplegar(self, event):
254         self.i=0
255         sizx=self.tc1.GetValue()
256         sizy=self.tc1.GetValue()
257         num=int(self.tc4.GetValue())
258         if sizx < 80 or sizy < 80:
259             sizex=sizx*10+10
260             sizey=sizy*10+10
261         elif sizx >=80 and sizx <250:
262             sizex=sizx*4+4
263             sizey=sizy*4+4
264         else:
265             sizex=sizx
266             sizey=sizy
267         self.frh = wx.Frame(self, -1, 'Animacion', pos=(260,0),size=(sizex,sizey))
268         wx.EVT_PAINT(self.frh, self.OnDisplay)
269         for self.i in range(num):
270             self.OnDisplay()
271             self.frh.Show(True)
```

Figura 4.15: Método "AlDesplegar".

En la primera parte del código de la figura 4.15 podemos apreciar el mismo escalamiento de pixeles hecho en el método "AlActivar" para representar adecuadamente los dispositivos de acuerdo al tamaño de la matriz. Al término del escalamiento, en la línea 267 comienza la creación del "wxFrame" que servirá como ventana de animación. En las últimas líneas se ejecuta un ciclo "for" que llama a la función "OnDisplay" tantas veces como iteraciónes se hayan introducido en el panel principal. Es importante señalar que esta función toma todos los datos necesarios del panel principal por lo que es importante conocer los parámetros con los que se hizo la simulación original (Figura 4.16).

```
243     def OnDisplay(self, event=None):
244         vel=int(self.tc5.GetValue())
245         dc = wx.BufferedDC(wx.ClientDC(self.frh))
246         cuadro = 'renderframes/%s%s.tif' % (self.tc6.GetValue(),self.i)
247         image = wx.Image(cuadro)
248         image = image.ConvertToBitmap()
249         dc.Clear()
250         dc.DrawBitmap(image,0,0, False)
251         temps=(100-vel)*0.01
252         time.sleep(temps)
```

Figura 4.16: Método "OnDisplay".

La primera línea de este código corresponde a la obtención del valor de velocidad escogido por el usuario. Se diseñó una barra de velocidad con escala del 0 al 100 donde 100 representa la mayor velocidad y 0 la menor. En la línea 245 se crea un objeto del tipo "wx.BufferedDC". El sufijo DC significa "device context", que es el instrumento que utiliza wx.Python para pintar texto y gráficos en la pantalla. El tipo "Buffered" se refiere a la creación de un "device context" que espera a que todos los pixeles de una imagen se completen en la memoria antes de ser desplegados, esto evita efectos indeseables como imágenes borrosas o con defectos y permite una animación fluida. En la misma línea, se observa como la creación del "wxBufferdDC" está anclada al cuadro "frh" que creamos en el método anterior.

Una vez creado el "device context" es necesario indicar la localización de las imágenes a desplegar, "cuadro" es una cadena que indica la ubicación y el nombre de las imágenes almacenadas utilizando el valor del campo "nombre" y la variable "i", controlada por el método "AlDesplegar". Con la ruta definida, se crea una imagen tipo "wxImage" que, al ser convertida en un mapa de bits, se transmite al "device context" creado previamente. En

las últimas líneas se pide al programa esperar un tiempo proporcional a la velocidad solicitada con el comando "time.sleep".

4.2. Implementación en C

La implementación en Python nos permitió contar con una interfaz visual sencilla e intuitiva que nos muestra una animación de la red en tiempo real. Además, con la versátil sintaxis de Python nos fue posible programar todo esto con una mínima cantidad de código. Sin embargo, al ser un lenguaje interpretado, Python impone serias restricciones de velocidad al simular matrices mayores a 100x100. Como mencionamos en la introducción, nuestro objetivo es medir la manera en la que cambian la complejidad y la entropía de la red de acuerdo a la densidad de la misma, esto significa que dada una matriz de tamaño NxN, es necesario simular el comportamiento de la red para cada valor de densidad. Aunado a esto, para cada valor de densidad que se desee simular, es preciso simularlo por lo menos unas 10 veces ya que en nuestro sistema interviene considerablemente el azar y es necesario obtener medidas promedio de la complejidad y la entropía. Esto significa que para conocer la variación de la complejidad en una red con un tamaño determinado en variaciones de densidad de 0.1 en 0.1, se requieren cerca de 1000 simulaciones para obtener el espectro total de densidad. Esto hace indispensable el uso de un lenguaje compilado ya que cualquier lenguaje interpretado tardaría demasiado en obtener estos resultados. Por esta razón hemos escogido C pues es uno de los lenguajes más rápidos que existen.

En la implementación en Python utilizamos un enfoque orientado a objetos por que el programa en sí se adecuaba a este enfoque y porque la sencilla definición de clases y métodos en Python nos permiten tener un código reducido. En el caso de C decidimos mantener un enfoque estructurado, con funciones en lugar de métodos y variables globales en lugar de atributos, con el fin de obtener un código simplificado.

La gran mayoría de las funciones necesarias en la implementación en C son simples traducciones de los métodos del núcleo de la implementación en Python, por lo que no vale la pena reproducirlos aquí, en lugar de ello, mostraremos la función "main" del programa para entender cómo funciona y apreciar los ciclos que permiten obtener los datos de una simulación de densidad completa (Figura 4.17).

```
39  /*------------------Programa principal-----------------------*/
40  main(){
41      int complex, ppmuestra, w, z;
42      float entropia, inter, coef, complexnorm;
43
44      FILE *complexPtr;
45      FILE *entropiaPtr;
46
47      if ((complexPtr = fopen("complex.dat", "w"))==NULL ||
48          (entropiaPtr = fopen("entropia.dat", "w"))==NULL){
49          printf("Alguno de los archivos no pudo ser abierto.\n");}
50      else{
51          srand(time(NULL));
52          printf("Working hard...\n");
53
54  /**************Parametros de entrada*****************************/
55          maxx=100;
56          maxy=100;
57          maxit=1000;
58          ppmuestra=10;
59          inter=0.01;
60          g=0.05;
61          pac=0.01;
```

Figura 4.17: Variables locales, apuntadores y parámetros de entrada de la implementación en C.

La primera parte de la función corresponde a la declaración de las variables locales. Es importante mencionar que no aparece la declaración de variables como el tamaño de la matriz, el número de iteraciones por simulación o los arreglos de dispositivos, ya que estos fueron declarados fuera de la función "main" para que actuaran como variables globales y así poder ser accesadas por cualquier función del programa. En la siguiente sección, se definen dos apuntadores que son necesarios para almacenar los datos de la simulación en los archivos "complex.dat" y "entropia.dat", una vez abiertos estos archivos en el sistema, comienza la definición de parámetros de la simulación:

maxx y maxy Tamaño de la matriz.

maxit Número de iteraciones por simulación.

ppmuestra Número de simulaciones a realizar por cada intervalo de densidad.

inter Intervalo de densidad.

g Parámetro de no-linealidad del modelo.

pac Probabilidad de activación espontánea en el modelo.

En la figura 4.18, por ejemplo, se desea simular la red en una matriz de 100x100, con 1000 iteraciones por simulación, a un intervalo de densidad de 0.1 en 0.1 hasta llegar a 1, simulando 10 veces cada valor de densidad.

```
63  /*************Comienza Simulación*******************************/
64      coef=inter;
65      while (coef<=1){
66          num=coef*maxx*maxy;
67          coef+=inter;
68          for (w=0;w<ppmuestra;w++){
69              inicia();
70              for (numit=0; numit<maxit;numit++){
71                  activalevy();
72                  suma=Hactivas();
73                  cadenab[numit]=suma/((num/2+1));
74                  actividad[numit]=suma;
75                  }
76              complex=Kolmogorov();
77              complexnorm=complex/(maxit/log2(maxit));
78              entropia=Shannon();
79              fprintf(complexPtr,"%4.2f\t %f\n",coef-inter,complexnorm);
80              fprintf(entropiaPtr,"%4.2f\t %f\n",coef-inter,entropia);
81              }
82          }
83          fclose(complexPtr);
84          fclose(entropiaPtr);
85      }
86  }
```

Figura 4.18: Segunda parte de la función “main” de la implementación en C.

En la primera línea de este figura se iguala el valor de “coef” con “inter”, ya que el valor inicial de densidad equivale al intervalo de densidad deseado. El ciclo “while” controla que la simulación llegue hasta un valor de densidad de 1. Dada la densidad de simulación, es necesario conocer el número de dispositivos presentes en la red, esto se calcula en la línea 66 multiplicando el valor de densidad “coef” por el área de la matriz. Enseguida, incrementamos el valor de “coef” de acuerdo al “inter” introducido. El siguiente ciclo, anidado en el ciclo anterior, se ejecuta tantas veces como simulaciones se deseen por valor de densidad (en este caso 10). El primer paso, controlado por la función “inicia()” vacía la matriz de simulación y coloca el número de dispositivos aleatoriamente. El segundo ciclo anidado contiene propiamente la simulación de la red y se ejecuta tantas veces como iteraciones por simulación se deseen. Los pasos de este ciclo son idénticos a los del botón “Activar” en la implementación en Python, es decir, existe una función de activación que, de acuerdo al tipo de movimiento escogido, desplaza los dispositivos de la red y actualiza su estado. Una vez terminada esta simulación se calculan la complejidad con la función “Kolmogorov” y la entropía con la función “Shannon”. Al final, estos valores se almacenan en dos archivos: “complex.dat” y “entropia.dat”, que contienen la información que nos interesa, es decir, el cambio de la complejidad y la entropia conforme cambia la densidad de la red.

Capítulo 5

Resultados

5.1. Entropía y complejidad algorítmica en la red

Los archivos generados por la implementación en C, en los cuales se muestra la entropia y la CAK de la red conforme aumenta la densidad, constituyen la principal herramienta de investigación en nuestro experimento. Al graficar dichos archivos para una matriz de 30x30 con dispositivos con desplazamiento Moore obtenemos la figura 5.1.

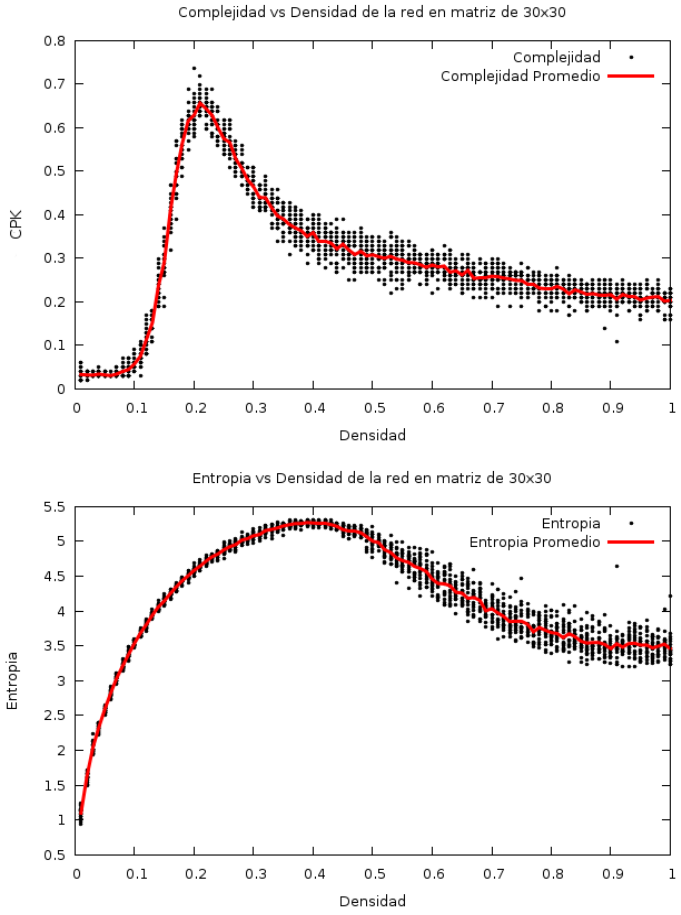

Figura 5.1: Gráficas de CAK vs densidad y entropia vs densidad de una simulación de red completa para una matriz de 30x30.

En estas gráficas, para cada valor de densidad se realizaron 20 simulaciones, representadas por los puntos

negros. Cada simulación realiza 1000 iteraciones o pasos del AC. La línea roja indica el promedio de las 20 simulaciones para cada valor de densidad. La probabilidad de activación espontánea es de de 0.01 y el parámetro de no-linealidad es de 0.05.

En la gráfica de CAK, es posible apreciar que el máximo ocurre cerca del 0.2 de densidad, lo cual es congruente con el modelo de Miramontes *et al.* En el caso de la entropía, se observa un máximo menos pronunciado y alrededor del 0.4 de densidad. Es sabido que la complejidad algorítmica de Kolmogorov y la entropía informática de Shannon guardan similitudes teóricas [Kolmogorov 68], mismas que pueden ser apreciadas en las gráficas de la figura 5.1, ya que ambas alcanzan un máximo antes del 0.5 y, a partir de este punto, son monotónicamente decrecientes.

En las gráficas subsecuentes (Figura 5.2 y 5.3) mostramos la comparación entre los tres tipos de movimiento para diferentes tamaños de matriz, conservando el formato de la figura anterior; es decir, el promedio de 20 simulaciones, de 1000 iteraciones cada una en intervalos de densidad de 0.1 hasta completar 1.

De la figura 5.2 es posible observar, para cada uno de los movimientos, la variación de la entropia y de CAK conforme aumenta la densidad de la red. En el caso de la complejidad algorítmica, para todos los tipos de movimiento, la red alcanza un máximo de complejidad para después disminuir monotónicamente. Antes de este punto, el sistema se encuentra en estado caótico y el número de activaciones inducidas aumenta. Al pasar el punto de máxima complejidad, las activaciones disminuyen y, conforme aumenta la densidad de la red, el sistema entra en la etapa oscilatoria, provocando una disminución en la complejidad. De lo tres tipos de movimiento estudiados, el movimiento "mixing" alcanzó mayor CAK, seguido del movimiento por vuelos de Lévy. Esto puede explicarse a partir de la aleatoriedad de los tipos de movimiento: entre más aleatoria sea la forma en la que los dispositivos se dispersan en la red, mayor será el número de activaciones inducidas que generen y por lo tanto mayor será la CAK alcanzada. También es importante resaltar que la densidad crítica δ_c, a la cual se expresa la máxima CAK, también difiere en los tres casos, siendo el tipo de movimiento Moore el que se expresa a mayor densidad ($\delta_c \approx 0{,}2$).

En cuanto a la variación de la entropía, es posible advertir ciertas diferencias. Aunque la forma de las gráficas es similar, la máxima expresión de la entropía es mucho menos pronunciada que en el caso de la complejidad algorítmica (se aprecia cómo, en el caso del movimiento Moore es aún más alargada que en los otros dos tipos de movimiento). Para tamaños pequeños de red (Figura 5.2) es posible ver que la entropía es claramente mayor en el caso del movimiento Moore, pero conforme aumenta el tamaño de la matriz, esta diferencia tiende a disminuir (Figura 5.3).

Cuando la red alcanza el estado de saturación (densidad=1), todos los tipos de movimiento tienden al mismo valor de complejidad y entropía ya que todas las celdas del AC se encuentran ocupadas por nodos y estos no pueden moverse.

Al aumentar el tamaño de la red, más allá de una matriz de 80x80 (Figura 5.3) se observan variaciones importantes. En primer lugar, aunque el valor máximo de CAK alcanzado por cada tipo de movimiento no cambió, puede verse una acentuación en la cresta de la curva, reduciendo el número de densidades a las cuales el modelo alcanza una CAK significativamente mayor. En el caso de los movimientos Mixing y Moore, se aprecia un segundo pico de complejidad que comienza a apreciarse desde la matriz de 80x80 pero que, conforme aumenta el tamaño de la matriz, se acentúa aún más. Las oscilaciones en la complejidad hacia el final de la gráfica que mostraron los movimientos Mixing y Moore, también marcan una notable diferencia a los casos de matrices pequeñas, en donde la complejidad es monotónicamente decreciente.

En cuanto a la entropía de la red, las diferencias resultaron más sutiles al aumentar el tamaño. Aunque en todos los casos el movimiento Moore presentó mayor CAK (seguido de los vuelos de Lévy), la diferencia entre los valores máximos alcanzados por los tres tipos de movimiento se redujo hasta casi desaparecer: en la matriz de 150x150 las entropías máximas fueron $e_cMoore = 6{,}7$ dits $e_cMixing = 6{,}59$ dits y $e_cLevy = 6{,}68$ dits. Las diferencias en la forma de las gráficas de entropía mantuvieron la misma tónica que en el caso de la figura 5.2: Mixing y Lévy alcanzaron un máximo de entropía en diferentes valores de densidad para después descender y Moore conservó una forma significativamente diferente, sin máximos acentuados.

5.2. Oscilaciones en la complejidad algorítmica de Kolmogorov

Las oscilaciones presentes en la gráfica de CAK (Figuras 5.2-e, 5.3-g, 5.3-i y 5.3-k) suponen un resultado inesperado ya que por un lado rompen con la decreciente monotonía observada en matrices mas pequeñas y por otro, resulta desconcertante la aparición de estas en el caso CAK y no en el caso de la entropía. Al graficar el número de dispositivos activos por iteración para dos diferentes etapas de las crestas de oscilación de CAK

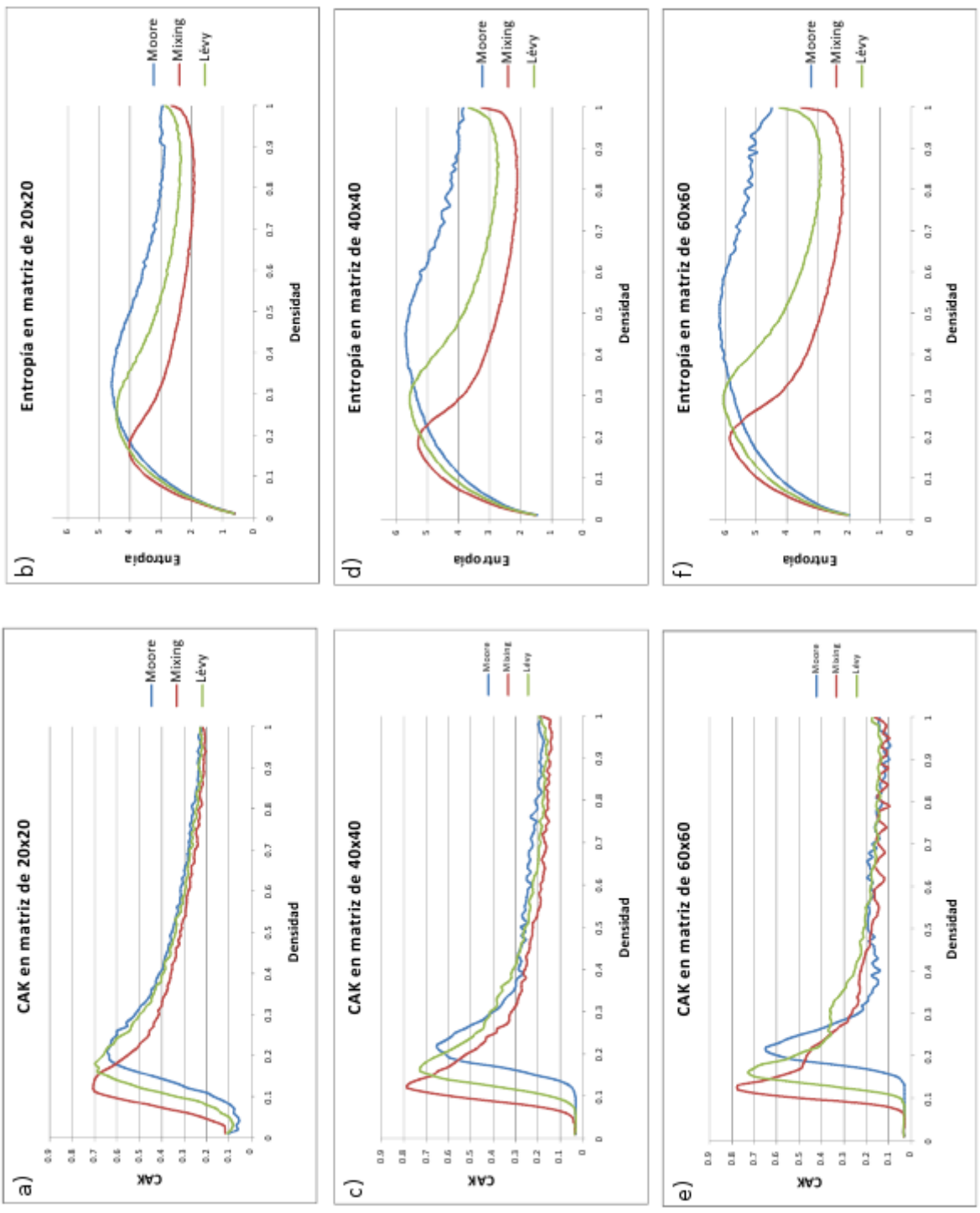

Figura 5.2: Primera serie de resultados: se muestran la entropía y la CAK de la red de acuerdo al tipo de movimiento para matrices de 20x20, 40x40, y 60x60.

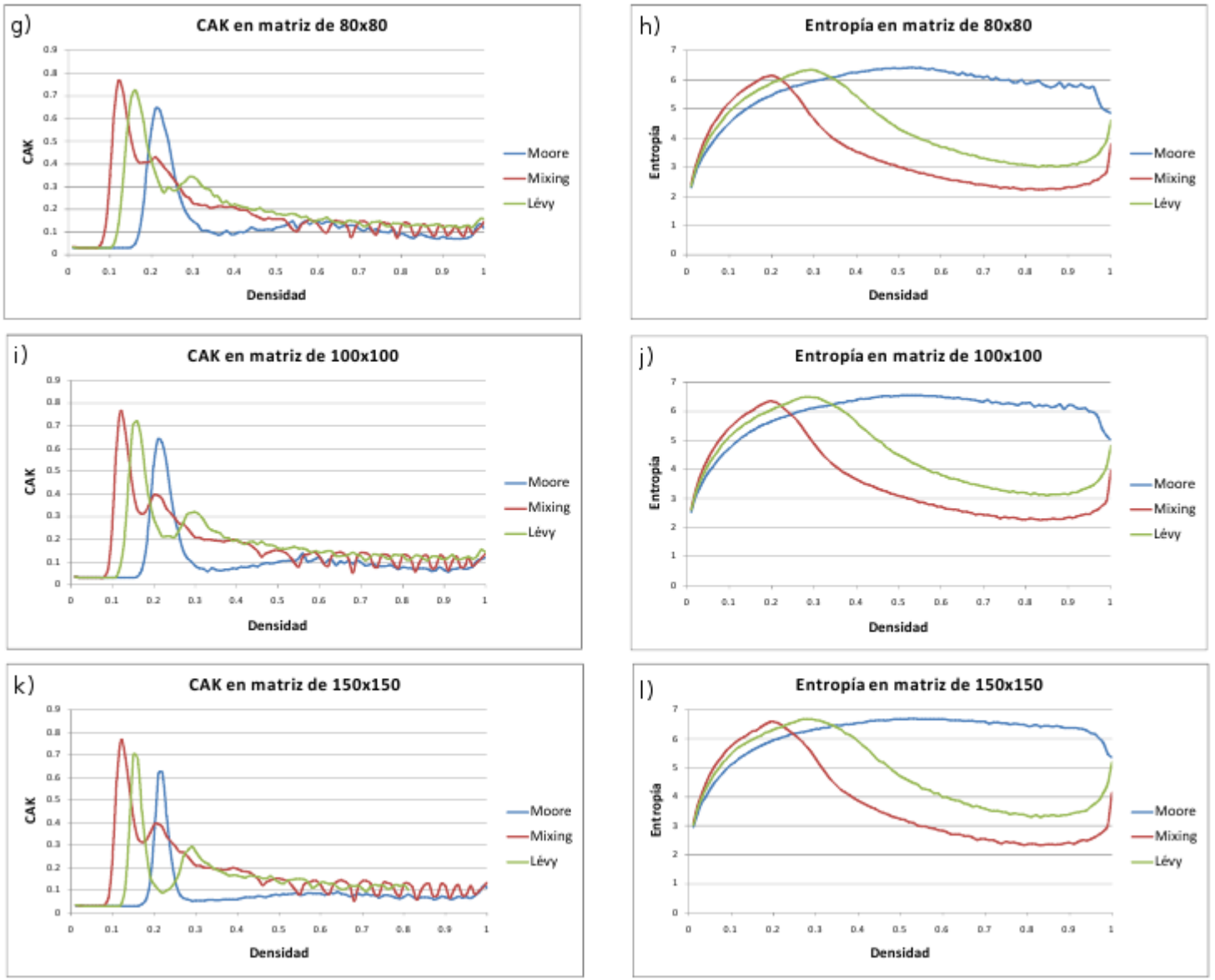

Figura 5.3: Segunda serie de resultados: se muestran la entropía y la CAK de la red de acuerdo al tiempo de movimiento para matrices de 80x80, 100x100 y 150x150

obtenemos la figuras 5.4 y 5.5. En la figura 5.4 observamos el número de nodos activos en la red cuando esta trabaja a 0.64 de densidad, que representa un máximo en una cresta de oscilación de CAK. A pesar de que el sistema se encuentra en estado oscilatorio, no se aprecia un patrón claro en la secuencia de nodos activos entre cada oscilación de actividad, lo cual aporta complejidad al sistema. Al graficar el mínimo de una cresta de oscilación de CAK, en la figura 5.5, se observa que, además de que el sistema continúa en estado oscilatorio, el patrón de nodos activos entre cada cresta de actividad es bastante regular, lo cual se refleja en una considerable disminución de la CAK de la red. No así de la entropía pues esta última no es sensible a la secuencia de estados sino a la probabilidad de que un estado específico ocurra, lo cual no resulta significativamente distinto en las dos etapas de la cresta de oscilación analizadas.

Aunque esto explica porqué las oscilaciones sólo se observan en CAK y no en la entropía, no basta para entender el origen de las mismas en marices grandes. En la siguiente sección trataremos de dar algunas hipótesis al respecto.

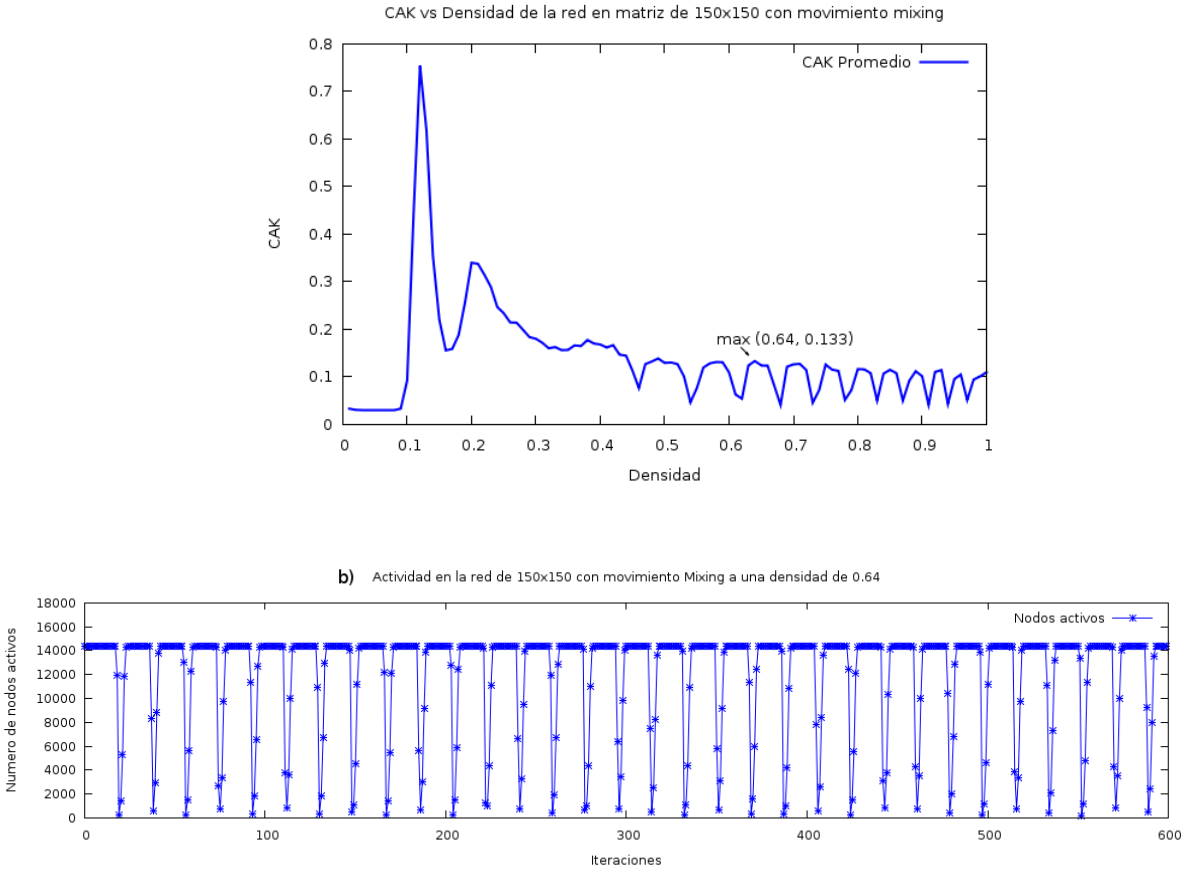

Figura 5.4: a) Gráfica de CAK para matriz de 150x150 donde se muestra un punto máximo de las crestas de oscilación (0.64, 0.133) b) Gráfica de actividad en la red cuando trabaja al 0.64 de densidad en la que puede verse la periodicidad en la actividad.

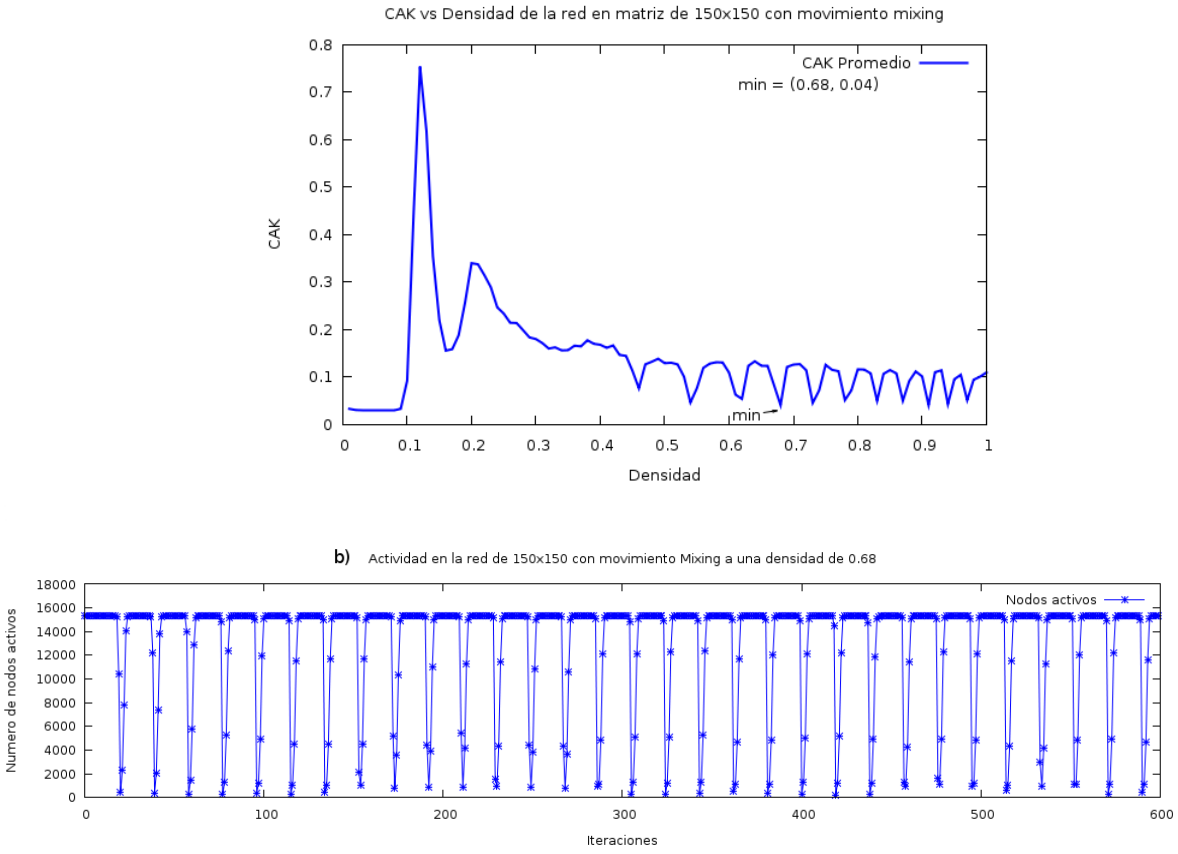

Figura 5.5: a) Gráfica de CAK para la red de 150x150 donde se muestra un punto mínimo de las crestas de oscilación (0.68, 0.04) b) Gráfica de actividad en la red cuando trabaja al 0.68 de densidad. Además de que el sistema se encuentra en estado oscilatorio, es posible observar un patrón claro de nodos activos entre cada cresta de actividad.

Capítulo 6

Conclusiones

El modelo de actividad en hormigas *Leptothorax* de Miramontes *et al.* ha mostrado ser un modelo de comportamiento cooperativo, robusto y dinámicamente interesante por el estado de transición entre orden y caos que presenta al aumentar la densidad del hormiguero. En el presente trabajo nos propusimos aplicar este modelo a una red de cómputo móvil y analizar sus características para tres diferentes tipos de movimiento. Como herramienta de medición escogimos la complejidad algorítmica de Kolmogorov y la entropia informática de Shannon ya que ambas reflejan el estado dinámico de un sistema. De esta manera, la principal conclusión a la que llegamos es que el tipo de movimiento que presentan los nodos en la red tiene un impacto directo en las métricas de contacto de los nodos de la red y por ende en la complejidad del sistema.

En cuanto a la complejidad algorítmica de Kolmogorov, los resultados muestran que entre más aleatoria es la forma de desplazamiento de los nodos, mayor CAK exhiben. Esto resulta relevante ya que una mayor complejidad en el sistema puede traducirse en mayor desempeño en los protocolos de ruteo de una DTN.

En el caso de la entropía de Shannon, el valor de los máximos alcanzados por cada tipo de movimiento no difieren tanto como en el caso anterior. En los tipos de movimiento Mixing y Moore las densidades críticas de entropía y CAK se encuentran muy próximas, lo cual permite definir una zona de operación óptima, que abarca del máximo de CAK al máximo de entropía, en donde ambos valores se mantienen altos. En matrices grandes (>80x80) el máximo de entropía coincide con una disminución significativa de la complejidad algorítmica (figura 5.3), por lo que la zona de operación óptima se ve reducida.

El aumento en el tamaño de la matriz del sistema trajo consigo tres principales modificaciones en el comportamiento de la complejidad algorítmica: a) la cresta de CAK máxima disminuyó considerablemente en longitud b) apareció un segundo pico de CAK aproximadamente a 0.1 unidades de la densidad crítica y c) surgieron oscilaciones en la CAK del sistema en valores mayores al 0.5 de densidad. Estas características pueden ser interpretadas como una desestabilización en el comportamiento de la complejidad del sistema, ya que no muestran la regularidad observada en matrices pequeñas y, aunque bien pueden tratarse de propiedades emergentes que exhibe el AC en matrices grandes, sugerimos, a manera de hipótesis, que es posible que las ondas de activación generadas por los nodos de la red pierdan sincronía al encontrarse en espacios grandes.

Debido a la gran cantidad de recursos necesarios para implementar una DTN cuyos nodos cuenten con movimiento mixing, los vuelos de Lévy se perfilan como el caso más eficiente entre los tres estudiados, demandan una menor cantidad de recursos y alcanzan un alto valor de complejidad. De esta forma, podemos concluir que cualquier protocolo de ruteo, basado en contactos oportunos, que se aplique a una DTN cuyos nodos se desplacen por Vuelos de Lévy, exhibirán un mayor desempeño en comparación con otros tipos de movimiento.

Hemos visto que la ecuación del AC que gobierna los estados de los dispositivos no exige que todos los nodos se encuentren activos en todo momento, incluso, es posible que más de la mitad de los dispositivos en la red se encuentren inactivos en un instante de tiempo sin sacrificar la complejidad de la red. Esto podría traducirse en un significativo ahorro de energía en los dispositivos móviles. De igual forma, al contar con un protocolo de ruteo eficiente es posible disminuir la capacidad de almacenamiento necesario en cada dispositivo móvil para garantizar el envío eficiente de paquetes de información.

Al aplicar un modelo de sistemas biológicos en un contexto de ingeniería de redes, mostramos que un sistema de comunicación puede aumentar significativamente su desempeño al incorporar dinámicas complejas observadas en la naturaleza como el comportamiento cooperativo o el estado de borde de caos.

Anexos

Anexo A

Implementación en Python

Núcleo del Programa (Red.py)

```
# -*- coding: utf-8 -*-
#10 de Septiembre de 2009
#Autor FRANCISCO LOPEZ RAMIREZ

import random
import math

class Nodo:
        def __init__(self, x=0, y=0, e=0):
                self.x=x; self.y=y; self.e=e

        def __str__(self):
                return '('+ str(self.x) + ',' + str(self.y) + ',' + str(self.e) + ',' + ')'

class Red:
        def __init__(self, maxx, maxy, num): # tamaño de la matriz y numero de nodos
                self.maxx=maxx; self.maxy=maxy; self.num=num;
                self.pac=0.01               #probabilidad de activacion espontanea
                self.g=0.05                       #parametro de no-linealidad
                self.umbral_cero=1e-16  #valor practico de 0
                self.umbral=0.5         #este es el umbral para movimiento moore y mixing
                self.conact=1e-6;       #valor inicial de actividad en los nodos
                self.nodos=[]           #arreglo de elementos de la clase Nodo
                self.newstate=[]        #arreglo auxiliar de estados actualizados
                self.maxxmu=self.maxx+2 ; self.maxymu=self.maxy+2;
                self.grid=[[0 for col in range(self.maxxmu)] for row in range(self.maxymu)]
                for i in range(self.maxxmu):
                        for j in range(self.maxymu):
                                self.grid[i][j]=-2.0
                for i in range(self.num):
                        while True:
                                tx=int(random.random()*self.maxx)+1
                                ty=int(random.random()*self.maxy)+1
                                if self.grid[tx][ty]==-2.0: break
                        self.grid[tx][ty]=self.conact
                        self.nodos.append(Nodo(tx,ty,self.conact))
                        self.newstate.append(1)

        def mueve(self,i):
                if self.nodos[i].e > self.umbral_cero:
                        j=0
                        while True:
                                j=j+1
                                while True:
                                        rand=random.random()*3
                                        movx=int(rand)-1
                                        if (self.nodos[i].x + movx > 0) and (self.nodos[i].x +
                                                movx <= self.maxx): break
                                while True:
```

```
                                rand=random.random()*3
                                movy=int(rand)-1
                                if (self.nodos[i].y + movy > 0) and (self.nodos[i].y +
                                    movy <= self.maxy): break
                        if (self.grid[self.nodos[i].x + movx][self.nodos[i].y + movy]
                            == -2.0) or (j==6): break
                if j<6:
                        self.grid[self.nodos[i].x] [self.nodos[i].y]= -2.0
                        self.nodos[i].x=self.nodos[i].x + movx
                        self.nodos[i].y=self.nodos[i].y + movy
                        self.grid[self.nodos[i].x][self.nodos[i].y]=self.nodos[i].e

def transporta(self, i):
        if self.nodos[i].e > self.umbral_cero:
                j=0
                while True:
                        j=j+1
                        nouveaux=random.randint(1,self.maxx)
                        nouveauy=random.randint(1,self.maxy)
                        if(self.grid[nouveaux][nouveauy] == -2.0) or (j==6): break
                if j<6:
                        self.grid[self.nodos[i].x] [self.nodos[i].y]= -2.0
                        self.nodos[i].x=nouveaux
                        self.nodos[i].y=nouveauy
                        self.grid[self.nodos[i].x][self.nodos[i].y]= self.nodos[i].e

def calcampo(self,i):
        cx=int(self.nodos[i].x)
        cy=int(self.nodos[i].y)
        campo=0
        for j in range(cx-1,cx+2):
                for k in range(cy-1,cy+2):
                        if self.grid[j][k] >= -1:
                                campo = campo + self.grid[j][k]
        self.newstate[i]=math.tanh(campo*self.g)
        if self.newstate[i] < self.umbral_cero:
                self.newstate[i]=0
        rand=random.random()
        if rand <= self.pac:
                if self.newstate[i]<=0:
                        self.newstate[i]=self.conact

def vlevy(self, vsize):  #esta funcion determina la longitud del vuelo de levy
        b=2                                     #dada una distancia especifica
        c=pe=0
        p=[]
        for i in range(1,vsize+1):
                p.append(0)
        for i in range(1,vsize+1):
                c = c+i**(-b)
        for i in range(1,vsize+1):
                pe = pe+((i**(-b))/c);
                p[i-1] = pe
        ran=random.random()
        k=0
        while True:
                k=k+1
                if ran <= p[k-1]: break
        return k

def vuelalevy(self,i): #Esta funcion ejecuta un vuelo de levy para la hormiga nodos[i]
        if self.nodos[i].e > self.umbral_cero:
                j=0                     #Es decir, para cada hormiga
                while True:
                        j=j+1 #j es el contador de intentos para saltar
                        while True:
                                paso=self.vlevy(self.maxx)
                                #print 'Este es el paso de levy escogido: ', paso
                                dir=int(random.random()*8) #numeros aleatorios del 1
                                    al 7 inclusive para definir direccion
```

```
                                #print 'Esta es la direccion escogida', dir
                                if dir==0:
                                        movx=0
                                        movy=paso
                                elif dir==1:
                                        movx=movy=paso
                                elif dir==2:
                                        movx=paso
                                        movy=0
                                elif dir==3:
                                        movx=paso
                                        movy=-paso
                                elif dir==4:
                                        movx=0
                                        movy=-paso
                                elif dir==5:
                                        movx=movy=-paso
                                elif dir==6:
                                        movx=-paso
                                        movy=0
                                elif dir==7:
                                        movx=-paso
                                        movy=paso
                                else:
                                        print 'Algo Terrible ha ocurrido'
                                if (self.nodos[i].x + movx > 0) and (self.nodos[i].x +
                                     movx <= self.maxx) and (self.nodos[i].y + movy >
                                    0) and (self.nodos[i].y + movy <= self.maxy):
                                    break
                        if (self.grid[self.nodos[i].x + movx][self.nodos[i].y + movy]
                            == -2.0) or (j==6): break #esto garantiza que solo se
                             mueve si la matriz esta vacia
                if j<6:
                        self.grid[self.nodos[i].x] [self.nodos[i].y]= -2.0
                                #putpixel
                        self.nodos[i].x=self.nodos[i].x + movx
                        self.nodos[i].y=self.nodos[i].y + movy
                        self.grid[self.nodos[i].x][self.nodos[i].y]=self.nodos[i].e

def activamoore(self):
        for i in range(self.num):
                self.calcampo(i)
        for i in range(self.num):
                        self.nodos[i].e=self.newstate[i]
                        self.grid[self.nodos[i].x][self.nodos[i].y]=self.newstate[i]
                        self.mueve(i)


def activamixto(self):
        for i in range(self.num):
                self.calcampo(i)
        for i in range(self.num):
                        self.nodos[i].e=self.newstate[i]
                        self.grid[self.nodos[i].x][self.nodos[i].y]=self.newstate[i]
                        rand=random.random()
                        if rand < self.umbral:
                                self.mueve(i)
                        else :
                                self.transporta(i)

def activalevy(self):
        for i in range(self.num):
                self.calcampo(i)
        for i in range(self.num):
                        self.nodos[i].e=self.newstate[i]
                        self.grid[self.nodos[i].x][self.nodos[i].y]=self.newstate[i]
                        self.vuelalevy(i)

def Hactivas(self):
        hactivas=0
```

```
            for i in range(self.num):
                    if self.nodos[i].e > self.umbral_cero:
                            hactivas=hactivas+1
            return hactivas

    def actividad(self):
            actividad=0
            for i in range(self.num):
                    actividad=actividad+self.nodos[i].e
            return actividad

    def complejidad(self, s):
            t=s
            control=self.num/2
            print 'control ',control
            for i in range(len(t)):
                    if t[i]>control:
                            t[i]=0
                    else:
                            t[i]=1
            c=1
            l=1
            while True:
                    i=0
                    k=1
                    kmax=1
                    while True:
                            while t[i+k-1]==t[l+k-1]:
                                    k=k+1
                                    if l+k<len(t):
                                            continue
                                    else:
                                            c=c+1
                                            return c
                            if k>kmax:
                                    kmax=k
                            i=i+1
                            if i==l:
                                    break
                            k=1
                    c=c+1
                    l=l+kmax
                    if l+1>len(t):
                            return c

    def entropia(self, s):
            entropia=0
            size=max(s)
            hist=[0.0 for p in range(size+1)]
            for i in range(len(s)):
                    hist[s[i]-1]=hist[s[i]-1]+1

            for i in range(size+1):
                    if hist[i]!=0:
                            pi=hist[i]/len(s)
                            entropia=entropia+pi*math.log(pi)
            entropia=-entropia

            return entropia
```

Interfaz visual (Interfaz.py)

```
# -*- coding: utf-8 -*-
#18 de Septiembre de 2009
#Autor FRANCISCO LOPEZ


import wx, os
import wx.lib.plot as plot
```

```
import random
import time
import math
from Red import Red
class Interfaz(wx.Frame):

        def __init__(self, parent, id, title):
                wx.Frame.__init__(self, parent, id, title, size=(610,460), pos
                    =(0,0))
                self.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)
                self.data=[]
                hbox  = wx.GridBagSizer()
                vbox1 = wx.BoxSizer(wx.VERTICAL)
                vbox3 = wx.GridSizer(5,4,5,5)
                vbox4 = wx.GridBagSizer(5,5)
                vbox6= wx.BoxSizer(wx.VERTICAL)
                pnl1 = wx.Panel(self, -1, style=wx.SUNKEN_BORDER)
                pnl2 = wx.Panel(self,-1, style=wx.SUNKEN_BORDER)
                pnl3 = wx.Panel(self, -1, style=wx.SUNKEN_BORDER)
                pnl1.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)
                pnl1.SetFocus()
                jpg1 = wx.Image("unam.gif", wx.BITMAP_TYPE_ANY).ConvertToBitmap
                    ()
                sb = wx.StaticBitmap(self, -1, jpg1, (jpg1.GetWidth(), 0), (
                    jpg1.GetWidth(), jpg1.GetHeight()))
                jpg2 = wx.Image("complejos.png", wx.BITMAP_TYPE_ANY).
                    ConvertToBitmap()
                sb2 = wx.StaticBitmap(self, -1, jpg2, (jpg2.GetWidth(), 0), (
                    jpg2.GetWidth(), jpg1.GetHeight()))
                vbox1.Add(pnl1, 1, wx.EXPAND | wx.ALL, 3)
                vbox1.Add(pnl2, 1, wx.EXPAND | wx.ALL, 3)
                vbox6.Add(pnl3, 1, wx.EXPAND | wx.ALL, 3)
                self.tc1 = wx.SpinCtrl(pnl1, -1, initial=30, max=100000)
                self.tc2 = wx.SpinCtrl(pnl1, -1, initial=30, max=100000)
                self.tc3 = wx.SpinCtrl(pnl1, -1, initial=100, max=10000000)
                self.tc4 = wx.SpinCtrl(pnl1, -1, initial=1000, max=100000)
                self.tc4.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)
                self.tc5 = wx.Slider(pnl1,-1, 100, 1, 100, size=(100, -1),
                style=wx.SL_HORIZONTAL | wx.SL_AUTOTICKS | wx.SL_LABELS, name="
                    Velocidad" )
                self.tc5.SetTickFreq(20,1)
                self.tc5.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)
                self.tc6 = wx.TextCtrl(pnl1, -1, "nombre")
                self.tc7=wx.CheckBox(pnl1,-1,"Almacenar")
                self.tc8=wx.CheckBox(pnl1,-1,"Animar")
                self.radiobox1=wx.RadioBox(pnl3, -1, "", (10, 10), (130,100),
                                                    ['MatrizMoore', 'Mixing', 'Vuelo
                                                        Levy'], 1, wx.
                                                        RA_SPECIFY_COLS)
                #self.Bind(wx.EVT_RADIOBOX, self.OnRadio, self.radiobox1)
                vbox3.AddMany([ (wx.StaticText(pnl1, -1, 'Longitud en X'),0, wx.
                    ALIGN_RIGHT),
                                                    (self.tc1, 0,),
                                                    (wx.StaticText(pnl1, -1, '
                                                        Longitud en Y'),0, wx.
                                                        ALIGN_RIGHT),
```

```
                                        (self.tc2,0,),
                                        (wx.StaticText(pnl1, -1, '
                                            Dispositivos'),0, wx.
                                            ALIGN_RIGHT),
                                        (self.tc3,0,),
                                        (wx.StaticText(pnl1, -1, '
                                            Iteraciones'), 0, wx.
                                            ALIGN_RIGHT),
                                        (self.tc4, 0,),
                                        (wx.StaticText(pnl1, -1, '
                                            Velocidad'), 0, wx.
                                            ALIGN_RIGHT),
                                        (self.tc5,0,),
                                        (wx.StaticText(pnl1, -1, 'Nombre
                                            '), 0 , wx.ALIGN_RIGHT),
                                        (self.tc6,0,),
                                        (wx.StaticText(pnl1, -1, ''), 0,
                                             wx.ALIGN_RIGHT),
                                        (self.tc7,0,),
                                        (wx.StaticText(pnl1, -1, ''), 0,
                                             wx.ALIGN_RIGHT),
                                        (self.tc8,0),
                                        ])
                pnl1.SetSizer(vbox3)
                vbox4.Add(wx.Button(pnl2, 10, 'Activar'), pos=(1,1), span=(4,1),
                     flag=wx.EXPAND)
                vbox4.Add(wx.Button(pnl2, 11, 'Actividad␣General'), (1,2),(1,1),
                     wx.EXPAND)
                vbox4.Add(wx.Button(pnl2, 12, 'Guardar␣D␣Activos'),(2,2), (1,1),
                    wx.EXPAND)
                vbox4.Add(wx.Button(pnl2, 13, 'Guardar␣Actividad'), (3,2), (1,1)
                    ,wx.EXPAND)
                vbox4.Add(wx.Button(pnl2,  15, 'Desplegar'), (1,3), (1,1),wx.
                    EXPAND)
                vbox4.Add(wx.Button(pnl2, 18, 'Cerrar'), (2,3), (1,1),wx.EXPAND)
                pnl2.SetSizer(vbox4)
                self.Bind(wx.EVT_BUTTON, self.AlActivar, id=10)
                self.Bind(wx.EVT_BUTTON, self.AlGeneral, id=11)
                self.Bind(wx.EVT_BUTTON, self.AlGuardarH, id=12)
                self.Bind(wx.EVT_BUTTON, self.AlGuardarA, id=13)
                self.Bind(wx.EVT_BUTTON, self.AlDesplegar, id=15)
                self.Bind(wx.EVT_BUTTON, self.AlCerrar, id=18)

                hbox.Add(sb, pos=(0,0), span=(1,1), flag=wx.EXPAND)
                hbox.Add(sb2, pos=(0,1), span=(1,1), flag=wx.EXPAND)
                hbox.Add(vbox1, pos=(1,0), span=(1,1), flag=wx.EXPAND)
                hbox.Add(vbox6, pos=(1,1), span=(1,1), flag=wx.EXPAND)
                self.SetSizer(hbox)

        def save_image(self, frameNumber):
                nombre = self.tc6.GetValue()
                circles_image = self.show_bmp.GetBitmap()
                frameFilename = '%s/%s%s.%s' %("renderframes", nombre,
                    frameNumber, "tif")
                circles_image.SaveFile(frameFilename, wx.BITMAP_TYPE_TIF)
```

```
def draw_image(self, nump, mult, radio):
        draw_bmp = wx.EmptyBitmap(self.sizex, self.sizey)
        self.canvas = wx.MemoryDC(draw_bmp)
        fondo=wx.Brush('black')
        self.canvas.SetBackground(fondo)
        self.canvas.Clear()
        self.canvas.SetPen(wx.Pen('black', 1))
        for i in range(nump):
                if self.red.nodos[i].e==0:
                        color="white"
                elif self.red.nodos[i].e <= 0.1 and self.red.nodos[i].e
                    > 0:
                        color="red"
                else: color="blue"
                self.canvas.SetBrush(wx.Brush(color))
                self.canvas.DrawCircle(self.red.nodos[i].y*mult,self.red
                    .nodos[i].x*mult,radio)
        self.show_bmp.SetBitmap(draw_bmp)


def OnKeyDown(self, event):
        keycode = event.GetKeyCode()
        if keycode == wx.WXK_RETURN:
                self.AlActivar(wx.EVT_KEY_DOWN)
        event.Skip()


def OnPaint(self, event=None):
        dc = wx.BufferedDC(wx.ClientDC(self.frh))
        dc.Blit(0,0,self.sizex,self.sizey, self.canvas, 0, 0)
        temps=(100-int(self.tc5.GetValue()))*0.01
        time.sleep(temps)

def AlActivar(self, event):
        if not self.tc1.GetValue() or not self.tc2.GetValue():
                return
        self.data=[]; self.data2=[]; self.data3=[]; self.data4=[]
        maxxp=int(self.tc1.GetValue())
        maxyp=int(self.tc2.GetValue())
        nump=int(self.tc3.GetValue())
        maxitp=int(self.tc4.GetValue())
        almacenar=self.tc7.GetValue()
        animar=self.tc8.GetValue()
        modalidad=self.radiobox1.GetSelection()
        self.red=Red(maxxp,maxyp,nump)
        accion=[lambda: self.red.activamoore(), lambda: self.red.
            activamixto(),
        lambda: self.red.activalevy()]
        frm = wx.Frame(self, -1, 'Actividad', pos=(460,0),size=(750,300)
            )
        frm.SetIcon(wx.Icon('antgris1.ico', wx.BITMAP_TYPE_ICO))
        client = plot.PlotCanvas(frm)
        if maxxp < 80 or maxyp < 80:
                self.sizex=maxxp*10+10
                self.sizey=maxyp*10+10
```

```
                mult=10
                radio=5
        elif maxxp >= 80 and maxxp <= 250:
                self.sizex=maxxp*4+2
                self.sizey=maxyp*4+2
                mult=4
                radio=2
        else:
                self.sizex=maxxp+1
                self.sizey=maxyp+1
                mult=1
                radio=1
        self.frh = wx.Frame(self, -1, 'Animacion', pos=(0,350),
        size=(self.sizex,self.sizey))
        self.frh.SetBackgroundColour('Black')

        wx.EVT_PAINT(self.frh, self.OnPaint)
        for j in range(maxitp):
                accion[modalidad]()
                suma=self.red.Hactivas()
                suma2=self.red.actividad()
                self.data.append((j,suma))
                self.data2.append((j,suma2))
                self.data3.append(suma)
                self.data4.append(suma)
                line = plot.PolyLine(self.data, legend='legend', colour=
                    'black', width=1)
                gc = plot.PlotGraphics([line], 'Dispositivos Activos', '
                    Iteracion'
                , '# de agentes activos')
                client.Draw(gc, xAxis=(0, maxitp), yAxis=(0,nump+0.1*
                    nump))
                self.show_bmp = wx.StaticBitmap(self.frh)
                self.draw_image(nump, mult, radio)
                if almacenar==True:
                        self.save_image(j)
                if animar==True:
                        self.OnPaint()
                        show=True
                else:
                        show=False

                frm.Show(True)
                self.frh.Show(show)
        p=self.red.complejidad(self.data3)
        p=p/(maxitp/math.log(maxitp,2))
        q=self.red.entropia(self.data4)
        #Comienza despliegue de entropia y complejidad
        frmcomplex = ComplexFrame(p,q)
        frmcomplex.Show(True)



def AlGeneral(self, event):
        nump=int(self.tc3.GetValue())
```

```
        maxitp=int(self.tc4.GetValue())
        frm = wx.Frame(self, -1, 'Actividad General', size = (750,300))
        frm.SetIcon(wx.Icon('antgris1.ico', wx.BITMAP_TYPE_ICO))
        client = plot.PlotCanvas(frm)
        line = plot.PolyLine(self.data2, legend='', colour='orange',
            width=2)
        gc = plot.PlotGraphics([line], 'Actividad General', 'Tiempo', '
            Actividad')
        client.Draw(gc, xAxis=(0, maxitp), yAxis=(0, nump*0.00000007))
        frm.Show(True)


    def AlGuardarH(self, event):
        s=''
        for i in range(len(self.data)):
                a=str(self.data[i])
                s=s+'%s\n'%(a[1:-1])
                s=s.replace(',','\t')
        dlg = wx.FileDialog(self, "Choose a file", os.getcwd(), "", "*.*
            ", wx.SAVE)
        if dlg.ShowModal() == wx.ID_OK:
                path = dlg.GetPath()
                mypath = os.path.basename(path)
                self.SetStatusText("You selected: %s" % mypath)
                f=open(mypath,"w")
                f.write(s)
        dlg.Destroy()

    def AlGuardarA(self, event):
        s=''
        for i in range(len(self.data2)):
                a=str(self.data2[i])
                s=s+'%s\n'%(a[1:-1])
                s=s.replace(',','\t')
        dlg = wx.FileDialog(self, "Choose a file", os.getcwd(), "", "*.*
            ", wx.SAVE)
        if dlg.ShowModal() == wx.ID_OK:
                path = dlg.GetPath()
                mypath = os.path.basename(path)
                self.SetStatusText("You selected: %s" % mypath)
                f=open(mypath,"w")
                f.write(s)
        dlg.Destroy()

    def OnDisplay(self, event=None):
        vel=int(self.tc5.GetValue())
        dc = wx.BufferedDC(wx.ClientDC(self.frh))
        cuadro = 'renderframes/%s%s.tif' %(self.tc6.GetValue(),self.i)
        image = wx.Image(cuadro)
        image = image.ConvertToBitmap()
        dc.Clear()
        dc.DrawBitmap(image,0,0, False)
        temps=(100-vel)*0.01
        time.sleep(temps)

    def AlDesplegar(self, event):
```

```
                self.i=0
                sizx=self.tc1.GetValue()
                sizy=self.tc1.GetValue()
                num=int(self.tc4.GetValue())
                if sizx < 80 or sizy < 80:
                        sizex=sizx*10+10
                        sizey=sizy*10+10
                elif sizx >=80 and sizx <250:
                        sizex=sizx*4+4
                        sizey=sizy*4+4
                else:
                        sizex=sizx
                        sizey=sizy
                self.frh = wx.Frame(self, -1, 'Animacion', pos=(260,0),size=(
                    sizex,sizey))
                wx.EVT_PAINT(self.frh, self.OnDisplay)
                for self.i in range(num):
                        self.OnDisplay()
                        self.frh.Show(True)

        def AlCerrar(self, event):
                self.Close()

#Clase para desplegar entropía y complejidad en un Frame
class ComplexFrame(wx.Frame):
        def __init__(self, complex, entropia):
                wx.Frame.__init__(self, None, -1, 'Complejidad y entropia', pos
                    =(800,350), size=(410, 80))
                #Viewing basic static text
                panel = wx.Panel(self, -1)
                panel.SetBackgroundColour('navy')
                string = "Complejidad: \t" + str(complex) + "\n" + "Entropia: \t
                    \t" + str(entropia)
                text = wx.StaticText(panel, -1, string, (0, 10))
                text.SetForegroundColour('orange')
                font = wx.Font(18, wx.DECORATIVE,wx.BOLD, wx.NORMAL)
                text.SetFont(font)
                self.SetIcon(wx.Icon('antgris1.ico', wx.BITMAP_TYPE_ICO))


class MyApp(wx.App):
        def OnInit(self):
                dia = Interfaz(None, -1, 'Simulación Agentes de cómputo móvil')
                dia.SetIcon(wx.Icon('antgris1.ico', wx.BITMAP_TYPE_ICO))
                dia.CreateStatusBar()
                dia.SetStatusText("Programa para la simulaicón de agentes de
                    cómputo móvil. FLR")
                dia.Show(True)
                return True

app = MyApp(0)
app.MainLoop()
```

Anexo B

Implementación en C

Red.c

```
//25 de Octubre de 2009
//Autor: FRANCISCO LOPEZ RAMIREZ


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

typedef struct nodo{
        int x;
        int y;
        double e;
        } Nodo;

const double conact = 1E-06;
const double umbral_cero = 1E-16;

Nodo nodo[1000000];
double grid[1000][1000];
double newstate[1000000];
int cadenab[2000];
int actividad[20000];
int numit, num, maxx, maxy, maxit;
int suma;
float g, pac;
//**********Declaración de Funciones**************************//
//**********Funciones aritméticas*****************************//
void calcampo(int);
int Dactivos(void);
int Kolmogorov(void);
float Shannon(void);
int pasolevy(int);
/***********Procedimientos*************************************/
void inicia(void);
void mueve(int);
void transporta(int);
void vuelalevy(int);
/***********Activaciones***************************************/
void activamoore(void);
void activamixto(void);
void activalevy(void);
/*--------------------Programa principal-------------------------*/
main(){
        int complex, ppmuestra, w, z;
        float entropia, inter, coef, complexnorm;

        FILE *complexPtr;
        FILE *entropiaPtr;

        if ((complexPtr = fopen("complex.dat", "w"))==NULL ||
```

```
                (entropiaPtr = fopen("entropia.dat", "w"))==NULL){
        printf("Alguno de los archivos no pudo ser abierto.\n");}
    else{
                srand(time(NULL));
        printf("Working hard...\n");

/*************Parametros de entrada*****************************/
                maxx=10;
                maxy=10;
                maxit=100;
                ppmuestra=2;
                inter=0.01;
                g=0.1;
                pac=0.01;

/*************Comienza Simulación*******************************/
                coef=inter;
                while (coef<=1){
                        num=coef*maxx*maxy;
                        coef+=inter;
                        for (w=0;w<ppmuestra;w++){
                                inicia();
                                for (numit=0; numit<maxit;numit++){
                                        activamoore();
                                        suma=Dactivos();
                                        cadenab[numit]=suma/((num/2+1));
                                        actividad[numit]=suma;
                                        }
                                complex=Kolmogorov();
                                complexnorm=complex/(maxit/log2(maxit));
                                entropia=Shannon();
                                fprintf(complexPtr,"%4.2f\t %f\n",coef-inter,
                                    complexnorm);
                                fprintf(entropiaPtr,"%4.2f\t %f\n",coef-inter,
                                    entropia);
                                }
                        }
                        fclose(complexPtr);
                        fclose(entropiaPtr);
                }
        }

/*----------------------Inicialización de los puntos de la red----------------------
   */

void inicia(void){
        int i,j;
        float ran;

        for(i=0;i<(maxx+1);i++){
                for(j=0;j<(maxy+1);j++){
                        grid[i][j]=-2.0;}}


        for(i=0;i<num;i++){
                do{
```

```
                        ant[i].x=rand() %(maxx);
                        ant[i].y=rand() %(maxy);
                        } while(grid[ant[i].x][ant[i].y]!=-2.0);

                ran=rand() %100;
                ran=ran/100;
                if (ran < 0.5)
                        ant[i].e=conact;
                else
                        ant[i].e=umbral_cero;
                grid[ant[i].x][ant[i].y] = ant[i].e;
                }
        }

/*----------------------Movimiento de las hormigas activas
   ----------------------------*/
void mueve(int i){
        int j, movx, movy;
        if (ant[i].e > umbral_cero){
                j=0;
                do{
                        j=j+1;
                        do{
                                movx=(rand() %3)-1;
                                } while((ant[i].x + movx <= 0) || (ant[i].x +
                                    movx > maxx-1));
                        do{
                                movy=(rand() %3)-1;
                                } while((ant[i].y + movy <= 0) || (ant[i].y +
                                    movy > maxy-1));
                        }while((grid[ant[i].x + movx][ant[i].y+movy] != -2.0) &&
                             (j<6));
                        if (j < 6){
                                grid[ant[i].x][ant[i].y] = -2.0;
                                ant[i].x = ant[i].x + movx;
                                ant[i].y = ant[i].y + movy;
                                grid[ant[i].x][ant[i].y] = ant[i].e;
                        }
                }
        }//mueve

/*--------------------Calculo del campo Promedio-----------------------*/
void calcampo(int i){
        int coorx, coory;
        double field, ran;
        int j,k;

        coorx=ant[i].x;
        coory=ant[i].y;
        field=0;
        for(j=(coorx-1);j<=(coorx+1);j++){
                for(k=(coory-1);k<=(coory+1);k++){
                        if (grid[j][k] >= -1){
                                field=field+grid[j][k];}}}
        newstate[i]=tanh(field*g);
```

```
        ran=rand() %(10000);
        ran=ran/10000;
        if (newstate[i] <= umbral_cero){
                if (ran <= pac){
                        newstate[i]=conact;}
                else{
                        newstate[i]=umbral_cero;}}
        }//calcampo

/*----------Función que activa la red con movimiento moore
   ----------*/
void activamoore(void){
        int i;
        for (i=0;i<num;i++){
                calcampo(i);
                }
        for (i=0;i<num;i++){
                ant[i].e=newstate[i];
                grid[ant[i].x][ant[i].y]=newstate[i];
                mueve(i);
                }
        }
/*----------Calcula el numero de nodos activos----------
   */
int Dactivos(void){
        int dactivos=0;
        int i;
        for(i=0;i<num;i++){
                if (ant[i].e > umbral_cero)
                        dactivos++;
                }
        return dactivos;
        }

/*----------Funcion que calcula la complejidad de Kolmogorov
   ----------*/
int Kolmogorov(void){
        int c,l,k,kmax,i,size;
        c=1;
        l=1;
        size=maxit;
        while(1==1){
                i=0;
                k=1;
                kmax=1;
                while(1==1){
                        while (cadenab[i+k-1]==cadenab[l+k-1]){
                                k=k+1;
                                if (l+k<size)
                                        continue;
                                else{
                                        c++;
                                        return c;}}
                        if (k>kmax)
                                kmax=k;
                        i++;
```

```
                        if (i==l)
                                break;
                        k=1;}
                c++;
                l=l+kmax;
                if(l+1>size)
                        return c;}
        }
/*----------------------Funcion que calcula la entropia de Shannon
   -------------------------*/
float Shannon(void){
        int x, w;
        float pi, ent;
        float hist[num];
        ent=0;

        for(x=0;x<num;x++)
                hist[x]=0;

        for(x=0;x<maxit;x++)
                hist[actividad[x]]+=1;

        for (x=0;x<num;x++){
                if (hist[x]!=0){
                        pi=hist[x]/maxit;
                        ent=ent+pi*log(pi);
                        }
                }
        ent=-ent;

        return ent;
}
/*----------------------Funcion que realiza Mixing Total en el movimiento
   -------------------------*/
void transporta(int i){
        float ran;
        int nx, ny, y;

        if (ant[i].e > umbral_cero){
                y=0;
                while (1==1){
                                y++;
                                nx=(rand()%maxx)+1;
                                ny=(rand()%maxy)+1;
                                if((grid[nx][ny] == -2.0) || (y==6))
                                        break;
                                }
                        if (y<6){
                                grid[ant[i].x][ant[i].y]= -2.0;
                                ant[i].x=nx;
                                ant[i].y=ny;
                                grid[ant[i].x][ant[i].y]= ant[i].e;
                                }
                }
        }
```

```
/*----------Función que activa la red con mitad moore, mitad mixing
   ----*/
void activamixto(void){
        int i;
        float ran;
        float umbral=0.5;
        for (i=0;i<num;i++)
                calcampo(i);

        for (i=0;i<num; i++){
                ant[i].e=newstate[i];
                grid[ant[i].x][ant[i].y]=newstate[i];
                ran=rand()%100;
                ran=ran/100;
                if (ran < umbral)
                        mueve(i);
                else
                        transporta(i);
                }
        }

/*----------Función que calcula la distancia del paso de levy dada una
     distancia maxima-------*/

int pasolevy(int vsize){
        int x,k;
        int b=2;
        float c,pe,ran;
        float p[vsize+1];

        c=pe=0;
        for (x=1;x<vsize+1;x++)
                p[x]=0;

        for (x=1;x<vsize+1;x++)
                c+=pow(x,-b);

        for (x=1;x<vsize-1;x++){
                pe+=pow(x,-b)/c;
                p[x-1]=pe;
                }
        ran=rand()%100;
        ran=ran/100;
        k=0;
        while(1==1){
                k++;
                if (ran<=p[k-1])
                        break;
                }
        return k;
        }

/*----------Función que ejecuta un vuelo de levy en dirección azarosa
    por cada nodo----------*/
void vuelalevy(int x){
        int y, dir, paso, movx, movy;
```

```
if (ant[x].e > umbral_cero){
        y=0;                    //Es decir, para cada hormiga
        while (1==1){
                y++; //y es el contador de intentos para saltar
                while (1==1){
                        paso=pasolevy(maxx);
                        //print 'Este es el paso de levy escogido: ',
                            paso
                        dir=rand()%8; //numeros aleatorios del 0 al 7
                            inclusive para definir direccion

                        //printf("Esta es la direccion escogida %d\n",
                            dir);
                        switch (dir){
                                case 0:
                                        movx=0;
                                        movy=paso;
                                        break;
                                case 1:
                                        movx=movy=paso;
                                        break;
                                case 2:
                                        movx=paso;
                                        movy=0;
                                        break;
                                case 3:
                                        movx=paso;
                                        movy=-paso;
                                        break;
                                case 4:
                                        movx=0;
                                        movy=-paso;
                                        break;
                                case 5:
                                        movx=movy=-paso;
                                        break;
                                case 6:
                                        movx=-paso;
                                        movy=0;
                                        break;
                                case 7:
                                        movx=-paso;
                                        movy=paso;
                                        break;
                                default:
                                        printf("Algo Terrible ha
                                            ocurrido\n");
                                }
                        if ((ant[x].x + movx > 0) && (ant[x].x + movx <=
                            maxx) && (ant[x].y + movy > 0) && (ant[x].y
                            + movy <= maxy))
                                break;
                        }
                        if ((grid[ant[x].x + movx][ant[x].y + movy] ==
                            -2.0) || (y==6))
```

```
                                        break;
                                        } //esto garantiza que solo se mueve si la
                                            matriz esta vacia
                                if (y<6){
                                        grid[ant[x].x][ant[x].y]= -2.0;
                                        ant[x].x=ant[x].x + movx;
                                        ant[x].y=ant[x].y + movy;
                                        grid[ant[x].x][ant[x].y]=ant[x].e;
                                        }
                                }
                        }

/*--------------------Función que activa la red con vuelos de Levy
    ------------------------------------------------*/
void activalevy(void){
        int x;
                for (x=0;x<num;x++)
                        calcampo(x);

                for (x=0;x<num;x++){
                                        ant[x].e=newstate[x];
                                        grid[ant[x].x][ant[x].y]=newstate[x];
                                        vuelalevy(x);
                                        }
        }
```

Bibliografía

[Aziz 07] Aziz W. and Arif M., "Complexity Analysis of Heart Beat Time series by Threshold based Symbolic Entropy", **IFMBE Proceedings**, Volume 15, (2007).

[Axtell 96] Axtell R. and Epstein J. M., "Growing Artificial Societies: Social Science from the Bottom Up", ***MIT Press***, (1996).

[Bar 97] Bar-Yam Y., "Dynamics of Complex Systems" ***Addison Wesley*** (1997).

[Basagni 04] Basagni S. *et. al.* "Mobile Ad Hoc Networking" ***IEEE Press-Wiley*** (2004).

[Barth 08] Barthelemy P., Bertolotti J. and Wiersma D.S., "A lévy flight for light", ***Nature,*** May 22 495-498, (2008).

[Bonabeu 91] Bonabeu E. *et. al.*, "Is Animal Behaviour Chaotic? Evidence from activity of ants" ***Proc. R. Soc. Lond. B,*** 244, (1991).

[Braun 96] Braun E., "Caos, fractales y cosas raras" ***FCE colección: la ciencia para todos,*** (1996).

[Choi 08] Choi S. and Lee B. G., "Broadband Wireless Access & Local Networks: Mobile Wimax an WiFi" ***Artech House Publishers,*** (2008).

[Chua 88] Chua L.O. and Yang L., "Cellular Neural Networks: Theory". ***IEEE Trans. Circuits Syst.*** *35*, 1257-1272 (1988).

[Corson 99] Corson M.S., Macker J.P. and Cirincione G.H., "Internet-based Mobile Ad Hoc Networking", ***IEEE Internet Computing Magazine,*** Jul/Aug, (1999).

[Cover 06] Cover T.M. and Thomas J.A., "Elements of Information Theory" ***Wiley-Interscience,*** 13-15, (2006) .

[d'Hum 86] d'Hummieres D. and Lallemand P., "Lattice gas cellular automata, a new experimental tool for hydrodynamics", ***Physica 140A***, 326, (1986).

[Eichler 06] Eichler S., Schorth C. and Eberspächer J., "Car-to-Car Communication" ***VDE-Kongress - Innovations for Europe CY - Aachen PB - VDE Verlag,*** (2006).

[Ergen 09] Ergen M., "Mobile Broadband- Including WiMAX and LTE", ***Springer NY***, (2009).

[Fall 03] Fall K., "A Delay-Tolerant Network Architecture for Challenged Internets", ***SIGCOMM*** (2003).

[Farrel 04] Farrel A., "The Internet and Its Protocols: A Comparative Approach" ***ELSEVIER*** (2004).

[Fazel 08] Fazel K. and Kaiser S., "Multi-Carrier and Strpead Spectrum Systems: From OFDM and MC-CDMA ot LTE and WiMAX", ***John Wiley & Sons,*** (2008).

[Fran 92] Franceschetti D.R., Campbell B.W. and Hamneken J.W., "Hamming nets, Ising sets, cellular automata, neural nets and random walks", ***American Journal of Physics,*** 61 50-53, (1992).

[Gardner 70] Gardner M., "The fantastic combinations of John Conway's new solitaire game "life" ", ***Scientific American***,223: 120-123, October, (1970).

[Gerhardt 89] Gerhardt M. and Schuster H., "A cellular automaton describing the formation of spatially ordered structures in chemical systems", ***Physica 36D,*** 209-221, (1989).

[Hauben 98] Hauben R., "From de ARPANET to the Internet", ***TCP Digest,*** (UUCP), (1998).

[Hopfield 82] Hopfield J.K., "Neural networks and physical systems with emergent collective computational abilities", ***Proc. natn. Acad. Sci.***, U.S.A 79, 2554-2557, (1982).

[Hossain 07] Hossain E. and Leung K., "Wireless Mesh Networks: Architectures and Protocols", ***Spinger***, (2007).

[Ilachinsky 01] Ilachinsky A., "Cellular Automata, a discrete universe", ***World Scientific Publishing,*** (2001).

[Jain 04] Jain S., Fall K. and Patra R., "Routing In A Delay Tolerant Network", **Proceedings of ACM SIGCOMM**, Portland (2004).

[Kant 86] Kant I., "Metaphysical Foundations of Natural Science", (4:470), (1786).

[Kaspar 86] Kaspar F. and Shuster H.G., "Easily Calculable Measure For The Complexity Of Spatiotemporal Patterns". ***Physical Review***, Volume 36, number 2, 842-848, (1986).

[Kess 90] Kessler D.A., Lovine H. and Reynolds W.N., "Coupled-map lattice model for crystal growth," ***Phys. Rev. 42A***, 6125, (1990).

[Kolmogorov 65] Kolmogorov A.N., "Three approaches to the definition of the concept "quantity of information"", ***Probl. Inf. Transmission,*** 1, 1 (1965).

[Kolmogorov 68] Kolmogorov A.N., "Logical Basis For Information Theory And Probability Theory". ***IEEE Trans. Inform. Theory***, IT-14(5):662–664, (1968).

[Kupier 08] Kupier E., "Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles", ***Linköping Studies in Science and Techology***, Thesis No. 1356, (2008).

[Landsberg 98] Landsberg P.T. and Shiner J.S, "Disorder And Complexity In An Ideal Non-equilibrium Fermi Gas", ***Physics Letters A***, Volume 245, Issues 3-4, (1998).

[Lee 08] Lee U. *et al.*, "Bio-inspired Multi-agent Data Harvesting In a Proactive Urban Monitoring Environment", ***J. Ad Hoc Networks***, ELSEVIER, (2009).

[Lempel 76] Lempel A. and Ziv J., "A Universal Algorithm for Sequential Data Compression", ***IEEE Trans. Inf. Theory***, IT-22, 75, (1976).

[Li 08] Li M. and Vitányi P., "An Introduction to Kolmogorov Complexity and Its Applications", ***Springer***, pp. 601-605, (2008).

[Linde 89] Lindemayer A. and Prusinkiewicz P., "Developmental models of multicellular organisms: a computer graphics perspective" ***Proc. Inter. Workshop on the Synthesis and Simulation of Living Systems***, Los Alamos, NM, USA, pp. 221-249, September, (1989).

[Low 08] Low P.S. *et. al.*, "Mammalian-like Features Of Sleep Structure In Zebra Finches", ***PNAS***, 105(26), 9081-9086, (2008).

[May 76] May R., "Simple Mathematical Models With Very Complicated Dynamics", ***Nature***, Vol. 261, June, (1976).

[Miramontes 95] Miramontes O., "Order-disorder Transitions In The Behavior Of Ant Societies" ***COMPLEXITY 1***, 56-60, (1995).

[Mitchel 09] Mitchel M., "Complexity: A Guided Tour", ***Oxford Univerity Press***, (2009).

[OlpcNews 10] www.olpcnews.com (2010).

[Phip 92] Phipps M.J., "From local to global: the lesson of cellular automata" pages 165-187 in Individual-Based Models and Approaches in Ecology: Populations, Communities and Ecosystems, edited by Deangelis D.L. and Gross L.J., ***Chapman and Hall,*** (1992).

[Popescu 10] Popescu-Zeletin R., Radusch I. and Rigani M. A., "Vihicular-2-x Communication", ***Springer***, (2010).

[Ramos 04] Ramos-Fernandez G. *et al.,* "Levy walk patterns in the foraging movements of spider monkeys (*ateles geoffroyi*)", ***Behavioural Ecology and Sociobiology***, (2004)

[Rappin 06] Rappin N. and Dunn R., "wxPython In Action", ***Manning***, (2006).

[Rhee 08] Rhee I. *et al.,* "On the Levy-walk Nature of Human Mobility", ***Proceedings of the 27th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)***,(2008)

[Sayer 10] Sayer P., "UN to buy 500,000 OLPC laptosps for Palestinian children", ***Bloomberg Buisnessweek***, IDG, April 29, (2010).

[Segel 99] Segel L. and Bar-Or R. L., "Immunology viewed as the study of an autonomous decentralized system" pages 65-88 in Artificial Immunes Systems and Their Applications, edited by D.Dasgupta, ***Springer-Verlag***, (1999).

[Sheen 08] Shin M. *et al.*, "DTN Routing Strategies using Optimal Search Patterns", ***SIGMOBILE***, ACM, (2008).

[Shles 94] Shlesinger M.F., Zaslavasky G.M. and Fisch U. (Eds.) "Lévy Flights and Related Topics in Physics" ***Proceedings of the International Workshop***, Nice France, June, (1994).

[Skarda 87] Skarda C.A. and Freeman W.J., "How Brains Make Chaos In Order To Make Sense Of The Workd", ***Brain Behav. Sci.****, 10*, 161-195, (1987).

[Solé 93] Solé R., Miramontes O. and Goodwin B., "Oscilations an Chaos in Ant Societies", ***J. theor Biol.***, (1993).

[Solé 95] Solé R. and Miramontes O., "Information At The Edge Of Chaos In Fluid Neural Networks". ***Physica D***, 80, 171-180, (1995)

[Solomonoff 64] Solomonoff R., "A Formal Theory Of Inductive Inference", ***Information and Control***, Volume 7, No 1, Pp.1-22 (1964).

[Sprint 10] (www.sprint.com/Mobile_WiMAX_) "Mobile WiMAX: The 4G Revolution Has Begun" (2010).

[Sung 09] Sungwon K. *et. al.*, "Super Diffusive Behavior Of Mobile Nodes And Its Impact In Routing Protocol Performance", ***NCSU***, (2009).

[Taniar 09] Taniar D., "Mobile Computing: Concepts, Methodologies, Tools and Applications", ***Information Science Reference***, (2009).

[Toffoli 77] Toffoli T., "Cellular Automata Mechanics", ***Comp. Comm. Sci. Dept.***, Technical Report 208, University of Michigan, (1977).

[Viswanathan 96] Viswanathan G. M. *et al.,* "Levy Flights Search Patterns Of Wandering Albatrosses", **Nature**, 381:413-415, (1996).

[Viswanathan 00] Viswanathan G. M. *et al.*, "Optimizing the success of random searches", **Nature**, 408, (2000).

[Warth 03] Warthman F., "Delay-Tolerant Networks (DTNs): A Tutorial", **IPNSIG**, (2003).

[Wk 05] (http://en.wikipedia.org/wiki/File:LogisticMap_BifurcationDiagram.png) "LogisticMap Bifurcation-Diagram.png " (2005).

[Wolf 87] Wolfram S. (ed.), "Theory and Applications of Cellular Automata", **World Scientific**, Singapore, (1987).