INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

E S C O M

Trabajo Terminal

"Computación basada en reacción de partículas en un autómata celular hexagonal"

20080040

Que para cumplir con la opción de titulación curricular en la carrera de

"Ingeniería en Sistemas Computacionales"

Presenta

Basurto Flores Rogelio
León Hernández Paulina Anaid

Directores

M. en C. Miguel Olvera Aldana **Dr. Genaro Juárez Martínez**

México D.F. a Junio de 2009

Computación basada en reacción de partículas en un autómata celular hexagonal
Trabajo terminal No. 20080040

Rogelio Basurto Flores
&
Paulina Anaid León Hernández

June 29, 2009

ADVERTENCIA

> "Este documento contiene información desarrollada por la Escuela Superior de Cómputo del Instituto Politécnico Nacional a partir de datos y documentos con derecho de propiedad y por lo tanto su uso queda restringido a las aplicaciones que explícitamente se convengan".

La aplicación no convenida exime a la escuela de su responsabilidad técnica y da lugar a las consecuencias legales que para tal efecto se determinen.

Información adicional sobre este reporte técnico podrá obtenerse en:

El Departamento de Diseño Curricular de la Escuela Superior de Cómputo del Instituto Politécnico Nacional, situada en Av. Juan de Dios Bátiz s/n esquina Miguel Othón de Mendizabal. Unidad Profesional Adolfo López Mateos. Teléfono: 5729-6000 Extensión 52021.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Hoy en día la computación se ha expandido a lo largo y ancho del mundo mediante computadoras que pueden ser usadas por niños, jóvenes y adultos por igual. Los años que se han invertido para que éstas sean usadas por gente fuera de los círculos de la ciencia han costado mucho en el sentido de una búsqueda de nuevas y diversas formas de hacer computación. La importancia de esta búsqueda radica en la imposibilidad de resolver muchos problemas de índole científica a través de los medios actuales.

Los diferentes enfoques en los que puede verse inmersa la computación han estado, en algunos casos, presentes desde los orígenes de las computadoras que conocemos actualmente; un perfecto ejemplo de esto es el hecho de que el creador de una de las arquitecturas de computadoras que se utilizan hoy en día, John von Neumann, fue también el autor de un modelo que ha ido creciendo a través de los años y con el que también se ha realizado computación, éste modelo es llamado: autómatas celulares.

Los autómatas celulares son un modelo que nos permite observar de una manera más sencilla sistemas de la naturaleza que de otra manera serían muy difíciles de estudiar. Con el paso del tiempo los autómatas celulares han sido estudiados, un punto decisivo en la historia del modelo se dio en los años 70's, cuando John Conway presentó el autómata celular llamado "El juego de la vida". Conway demostró entre otras cosas que era posible implementar computación en un autómata celular, mediante la construcción de compuertas lógicas. El juego de la vida abrió un camino que no se sospechaba y el cual se ha seguido estudiando, no sólo dentro de éste autómata celular, sino que se han encontrado otros autómatas que permiten la construcción de compuertas lógicas.

En el 2005 Andrew Adamatzky y Andrew Wuensche mostraron un autómata celular hexagonal llamado "Spiral", mismo que presentaba condiciones deseables para implementar computación de manera similar a como se realizó con el juego de la vida. Aún con ésto, las herramientas con las que se disponían no permitían un fácil y rápido estudio, para la posterior construcción de compuertas lógicas; por estas razones el presente trabajo tiene como objetivo la búsqueda de computación dentro de un autómata celular hexagonal y para ello se realizará el análisis, diseño y codificación de un simulador que permita la modificación de estados del autómata celular.

Para poder comprender el contenido del presente trabajo es necesario tener conocimiento de los siguientes temas:

- Teoría de conjuntos

- Teoría de grafos
- Relaciones
- Algebra de Boole

Por otra parte, se abordará un básico estudio de la teoría de la computación, tocando temas como:

- *Teoría de autómatas.* En este capítulo se estudiarán los conceptos de computación, proceso computable y computabilidad, posteriormente algunas de las diversas máquinas existentes con el fin de tener herramientas para la asimilación de los conceptos previos y obtener bases para el entendimiento de la teoría de autómatas celulares.
- *Computación no-convencional.* La sección esta dedicada a la explicación general de los diferentes paradígmas y medios por los cuales se busca implementar computación.
- *Computación basada en compuertas lógicas.* A lo largo de la sección se mostrarán las bases de las compuertas lógicas y su implementación actual, así como posibles implementaciones en diversos medios con el fin de ejemplificar que las compuertas lógicas son un modelo abstracto.

Posteriormente, es necesaria la asimilación de la teoría de autómata celular, por lo cual serán vistos los siguientes temas:

- *Etapas importantes.* Se da un recorrido por los puntos cruciales de la teoría de autómata celular; las explicaciones son de manera general, buscando la comprensión del marco en el que se desarrolló la teoría misma; queda complementado con las secciones siguientes.
- *Definiciones básicas.* Se dan los conceptos básicos y necesarios, de la misma forma se presenta una explicación minuciosa y formal de la teoría de autómata celular.
- *El juego de la vida.* Esta sección esta dedicada al estudio de un autómata celular en específico, con el fin de que se comprendan los conceptos previamente vistos; así como para mostrar la computación basada en compuertas lógicas vista a través de la teoría de autómatas celulares.
- *Autómata celular hexagonal.* Se presentan las condiciones que enmarcan la aparición e importancia de los autómatas celulares hexagonales y su uso en el presente trabajo.

Antes de poder analizar la regla se buscaron las herramientas necesarias para la observación y análisis de autómatas celulares hexagonales, al no encontrar una herramienta que se adaptara a dichas necesidades, se propuso una herramienta, la cual será detallada a lo largo del capítulo, de la siguiente manera:

- *Definición de problemas y causas.*Se realiza el estudio de la herramienta existente, se exponen las limitantes que tiene y se hace el planteamiento de las características requeridas por el simulador para poder realizar el estudio de la regla Spiral.

- *Requerimientos de la herramienta.* Se describen las características necesarias para el buen estudio de la regla Spiral.

- *Propuesta de solución.* Se desglozan las características que tiene la herramienta propuesta para el análisis.

- *Análisis y diseño de la solución.* Se muestran el modelado de la herramienta propuesta para su implementación.

Finalmente, se analiza la regla en la que se centra el trabajo, con la finalidad de implementar compuertas lógicas. El capítulo esta desarrollado como sigue:

- *Descripción de la regla.* Se desarrolla una descripción minuciosa de la regla para poder comprender plenamente su funcionamiento.

- *Dinámica y partículas básicas.* Se explica el comportamiento general y las particularidades de interés en la regla para fines computacionales, al tiempo que se muestran las partículas básicas y sus principales características.

- *Computación en la regla Spiral.* Se dan las pautas y condiciones necesarias en las partículas a utilizar en la búsqueda de computación en la regla Spiral. Se explica la analogía necesaria para la comprensión del funcionamiento de una compuerta lógica sobre un autómata celular. Son presentados los resultados obtenidos.

- *Otras configuraciones y partículas interesantes.* Se muestran partículas, como gliders y glider-guns, encontradas en las pruebas del simulador y a lo largo del análisis de la regla.

2. TEORÍA DE LA COMPUTACIÓN

2.1 Historia

La idea de construir una computadora autónoma se ha difundido desde mediados del siglo XVII con el calculador de Blas Pascal, en 1801 con el Telar de J. Jacquard, en 1821 con la máquina de diferencias de Babbage y más tarde con su máquina analítica. Sin embargo, no fue hasta el año de 1900 cuando se plantearon las preguntas correctas para el desarrollo de una máquina con cierta autonomía. Estas preguntas se consideran ahora los fundamentos de la teoría de la computación y fueron expuestas por el matemático alemán David Hilbert [11].

Hilbert compiló estas peguntas y algunas otras (23 en total), durante el transcurso del Congreso Internacional de Matemáticas, aquellas con un particular interés para el presente trabajo son:

1. ¿Son completas las matemáticas, en el sentido de que pueda probarse o no cada aseveración?

2. ¿Son las matemáticas consistentes, en el sentido de que no pueda probarse simultáneamente una aseveración y su negación?

3. ¿Son las matemáticas decidibles, en el sentido de que exista un método definido que se pueda aplicar a cualquier aseveración, y que determine si dicha aseveración es cierta o no?

La meta de Hilbert era crear un sistema matemático formal "completo" y "consistente", en el que todas las aseveraciones pudieran plantearse con precisión. Su idea era encontrar un algoritmo que determinara la verdad o falsedad de cualquier proposición en el sistema formal. A este problema se le llama el problema de decibilidad o *Entscheidungs problem*. Si Hilbert hubiera podido cumplir su objetivo, cualquier problema que estuviera bien definido se resolvería simplemente al ejecutar dicho algoritmo [10].

En la década de 1930 se produjeron una serie de investigaciones que mostraron que esto no era posible. Las primeras noticias en contra de esta idea surgieron en 1931 cuando K. Gödel publicó su famoso Teorema de Incompletitud. Con ello Gödel deja en claro que la idea de Hilbert no es posible, dado que ningún sistema deductivo que contenga los teoremas de la aritmética, y con los axiomas recursivamente enumerables, puede ser consistente y completo a la vez.

En el mejor de los casos, puede ser consistente e incompleto, o exclusivamente completo pero inconsistente. En caso de que una teoría pueda ser consistente, su consistencia no puede ser probada dentro de la teoría misma. Por lo tanto cualquier prueba de falsedad o verdad conduce a una contradicción. Gödel dejó abierta la pregunta de si podría haber un "proceso mecánico" con el que proposiciones indemostrables puedan ser demostradas.

A partir de ahí muchos matématicos se dieron a la tarea de buscar dicho proceso. Uno de ellos fue Church quien en su artículo de 1936, hace un esquema de la demostración de la equivalencia entre las funciones λ-definibles y las funciones recursivas de Herbrand-Gödel; y se aventura a decir que estas iban a ser las únicas funciones calculables por medio de un algoritmo [11].

De igual manera Alan Turing utilizó su concepto de máquina para demostrar que existen funciones que no son calculables por un método definido. Con el tiempo se hizo la demostración de que la tesis de Church y de Turing son equivalentes.

El identificar los problemas que son computables o de aquellos que no lo son, tiene un considerable interés, pues indica el alcance y los límites de la computabilidad; el desarrollo de la teoría de la computabilidad ha estado íntimamente ligado al desarrollo de la lógica matemática, esto ha sido así porque la decibilidad de los distintos sistemas lógicos es una cuestión fundamental; como ejemplo, se puede ver que el cálculo proposicional es decidible. Church y Turing demostraron en 1936 que el cálculo de predicados no era decidible. Por otro lado, para cada una de las distintas teorías se ha ido estudiando su posible decibilidad.

Con todo lo anterior se puede decir que el propósito inicial de la teoría de la computabilidad es hacer precisa la noción intuitiva de función calculable; esto es, una función cuyos valores pueden ser calculados de forma automática o efectiva mediante un algoritmo. Así se puede obtener una comprensión más clara de esta idea; y sólo de esta forma se puede explorar matemáticamente el concepto de computabilidad y los conceptos relacionadas con ella, como la decibilidad. Surge así una teoría que producirá resultados positivos y negativos (se esta pensando en resultados de no-computabilidad o de indecibilidad).

La teoría de la computabilidad puede caracterizarse, desde el punto de vista de las computadoras, como la búsqueda de respuestas para las siguientes preguntas:

1. ¿Qué pueden hacer las computadoras (sin restricciones de espacio, tiempo o dinero)?
2. ¿Cuáles son las limitaciones inherentes a los métodos automáticos de cálculo?

Es mediante el estudio de los diferentes modelos de computación tales como

la máquina de Turing, teoría de autómatas y lenguajes formales, entre otros, que se resuelven estas preguntas. Algunos de estos modelos serán estudiados más adelante, así se tratará de conocer los límites que tiene una computadora. Estos límites solo los marca la posibilidad de solución de los problemas, pero ¿qué hay si el tiempo necesario para realizar el cálculo es tan grande que tomaría una vida el realizarlo?, este caso ya no es muy factible. Otro caso interesante sería que con un modelo se tardase un tiempo muy grande y con otro modelo diferente el tiempo sea menor. En este punto surge la pregunta: ¿Cómo es posible saber cuando es más eficaz un modelo en comparación con otro? Por ejemplo, que sería más fácil utilizar, ¿una máquina de Turing o un autómata finito?. La respuesta esta dada, entre otras cosas, por las características del problema planteado.
La *teoría de la complejidad* es el área de estudio dentro de la teoría de la computación que busca la manera de responder estas interrogantes, para ello toma como medidas principales el tiempo y el espacio requeridos para la computación. Estos dos parámetros se toman de la siguiente manera:

- **Tiempo**. Mediante una aproximación al número de pasos de ejecución de un algoritmo para resolver un problema teniendo en cuenta que cada paso se realiza en un tiempo determinado.
- **Espacio**. Mediante una aproximación a la cantidad de memoria utilizada para resolver un problema.

Cabe destacar que las mediciones realizadas son para los diferentes modelos utilizados, y para aquellos entre los que existe una equivalencia conocida, pues el objetivo es saber que modelo es más conveniente utilizar para que los cálculos no tarden eones en proceso. Además de tiempo y espacio existen otros parámetros menos utilizados como por ejemplo la aleatoriedad, el número de alteraciones y el tamaño del circuito.

La teoría de la complejidad se alimentó de los trabajos realizados en la primera mitad del siglo XX por matemáticos como Kleene, Post, Gödel, Turing, entre otros. Pero no fue hasta 1965 en que vio la luz un trabajo intimamente relacionado con la complejidad, "On the Computational Complexity of Algorithms", por Juris Hartmanis y Richard Stearns. A pesar de esto, el documento solo reflejaba trabajos realizados con una máquina de Turing multicintas [12]. Hoy en día, con la tecnología que se tiene y los nuevos modelos que comienzan a tomar auge, esta rama de la teoría de la computación tiene una vital importancia para el futuro.

2.2 Teoría de autómatas

Como todas las teorías motivadas por las necesidades de la ciencia y la ingeniería, la teoría de autómatas de estados finitos concierne con modelos matemáticos de

cierta aproximación a fenómenos abstractos o físicos. Sin embargo, no está relacionada con un área en específico.

Uno puede encontrar las ideas y las técnicas de las máquinas de estados finitos empleadas en problemas no relacionados entre sí, como la investigación de la actividad nerviosa, el análisis del idioma inglés y el diseño de computadoras.

Muchos de los problemas encontrados en las investigaciones científicas y de ingeniería caen dentro de dos categorías:

- Problemas de análisis, donde se desea predecir el comportamiento de un sistema específico.
- Problemas de síntesis, donde se desea construir un modelo con un comportamiento específico.

Desde ambos puntos de vista, es conveniente separar las variables que caracterizan al sistema en:

1. Variables de exitación: Representan el estímulo generado por otro sistema diferente al que esta bajo investigación, el cual influye en el comportamiento del sistema.
2. Variables de respuesta: Simboliza aquellos aspectos del sistema que le interesan al investigador.
3. Variables intermedias: Son aquellas que no son, ni de exitación ni de respuesta.

Esquemáticamente, un sistema puede ser representado por una "caja negra", con un número finito de "terminales", como lo muestra la figura 2.1.

Las *terminales de entrada* representan las variables de exitación y son identificadas por las flechas que apuntan hacia adentro de la caja.
Las *terminales de salida* representan las variables de respuesta y son representadas por las flechas que apuntan hacia afuera de la caja.

Todos los sistemas, como las máquinas finitas, pueden ser controlados por una fuente sincronizadora. Esta fuente, se da gracias a un evento llamado *señal de sincronización*; esta señal se da cada determinado tiempo, a ese tiempo se le llama, *tiempo de muestreo o sampling*, cada señal de sincronización se le conocerá como el tiempo discreto v y cada una de ellas deberá tener el mismo intervalo. Así cualquiera que sea el intervalo, cualquiera que sea el sistema de variación, los valores dependen del número de señales de muestreo.

Se le llama *alfabeto* al conjunto finito de distintos valores que puede tener las variables de la máquina. En general, estos conjuntos se dividen en el alfabeto

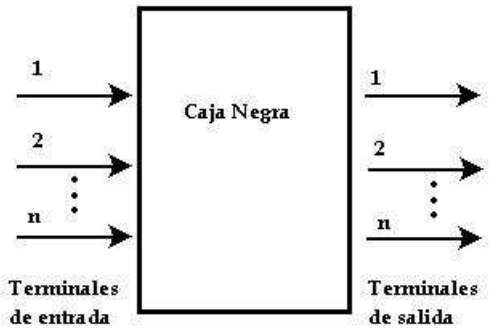

Fig. 2.1: Esquema representativo de un sistema.

de entrada, el alfabeto de salida, y cada elemento del alfabeto se llama *símbolo*.

Las variables intermedias, no pueden ser estudiadas, dado que no son del conocimiento del investigador, como las variables de entrada y de salida, sin embargo, son muy importantes ya que influyen en el sistema, y causan un "efecto", este efecto se le llama *estado*. El total de efectos que puede ser mostrado por un sistema se le llama *conjunto de estados*.

Al intentar definir lo que es un estado, se puede ver como una característica específica y que puede ser descrita por lo siguiente:

- El símbolo de salida en un tiempo t es únicamente determinado por un símbolo de entrada en el mismo tiempo t.
- El estado del tiempo $t+1$ esta dado por el símbolo de entrada y el estado del tiempo t.

A continuación se describen los ejemplos más relevantes de la teoría de máquinas.

2.2.1 Máquina de Turing

En 1936 Turing dio vida a la idea intuitiva que hasta la actualidad se tiene de algoritmo, formalizándola en lo que se denomina hoy en día como máquina de Turing (MT). Este modelo matemático sentó las bases de las ciencias de la computación y tiene como característica principal ser el autómata o máquina abstracta más general.

La MT puede ser descrita intuitivamente como un dispositivo compuesto de:

1. Una *unidad de control* que puede adoptar un número finito de estados $q_0, q_1, q_2, \ldots, q_{n-1}$ y almacena el programa de la transformación a realizar, el cual determina el conjunto de estados Q de la máquina.

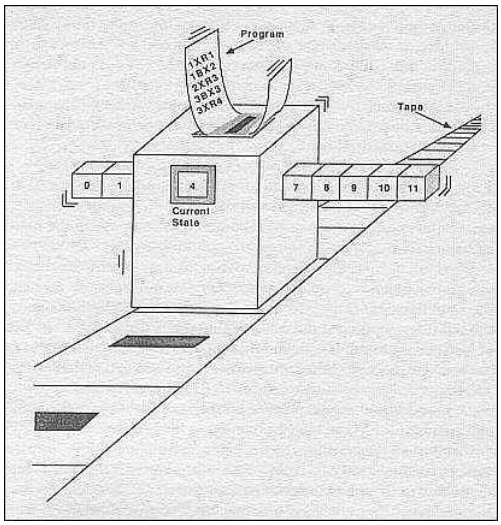

Fig. 2.2: Máquina de Turing [1].

2. Una cinta de entrada dividida en celdas, infinita por ambos extremos, cada celda tiene un símbolo de un conjunto de símbolos $V = \{s_1, s_2, s_3, \ldots, s_m\}$, llamado *alfabeto* de la MT. A este alfabeto se le añade el símbolo "espacio en blanco" (B).

3. Un *cabezal lector-escritor* con movimientos discretos sobre el tiempo t, donde cada intervalo t se posiciona en una celda de la cinta gracias a un movimiento, ya sea a la derecha o izquierda, para poder hacer una lectura o escritura.

En cada tiempo t la máquina puede adoptar uno de los estados del conjunto de estados Q, puede leer o escribir de la cinta mediante el cabezal, y en función de los valores del estado y el símbolo, puede realizar alguna de las siguientes acciones:

1. Escribe un nuevo símbolo en la cinta, en la misma casilla donde acaba de leer.

2. Desplaza el cabezal una única posición a lo largo de la cinta, hacia la derecha o hacia la izquierda, o bien puede dejarlo apuntando a la misma casilla.

3. Cambia de estado.

Existen diferentes variantes de la MT, por ejemplo, si se define una que analice palabras sucederá lo siguiente: al inicio del proceso la maǵuina tiene el cabezal

[1] Imagen tomada de static.flickr.com/93

posado sobre el primer símbolo de la palabra a analizar; la máquina se moverá y escribirá según las reglas de transición o acciones definidas para esa máquina; en caso de no tener ninguna acción para la configuración en la que se encuentre la máquina se detiene.
La definicón de la MT se da por la septupla:

$$M = (\Sigma, \Gamma, B, Q, q_0, F, \delta) \tag{2.1}$$

Donde:

- Σ es el alfabeto de entrada finito de las palabras a procesar.
- Γ es el alfabeto de la cinta, que contiene todos los símbolos que la máquina puede leer o escribir en la cinta. Siempre se cumple que $\Sigma \subset \Gamma$.
- $B \in \Gamma$ representa el símbolo en blanco o espacio en blanco.
- Q es el conjunto finito de estados.
- q_0 es el estado inicial.
- F es el conjunto de estados finales.
- $\delta : (Q \times \Gamma) \rightarrow (Q \times \Gamma \times +, -, =)$ es una función parcial, llamada *función de transición*

A partir del estado actual y del símbolo leído en la cinta, se recibe de la función δ el estado al que pasa la máquina, el símbolo que escribe en la cinta y en que dirección se mueve el cabezal. Los símbolos $+, -$ e $=$ representan, respectivamente un movimiento, a derecha, izquierda o que el cabezal permanezca en la misma celda.

Dado que Q y Γ son conjuntos finitos, la función δ puede representarse mediante una tabla en la que cada fila corresponde a un estado y cada columna a un símbolo de entrada, y cada celda muestra el valor de δ.

Un ejemplo de una MT es:

$$\mathcal{M} = (\Sigma, \Gamma, B, Q, q_0, F, \delta) \tag{2.2}$$

Donde:

$\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, B\}$, $Q = \{q_0, q_1, q_F\}$, $F = \{q_F\}$

Con δ definida como se muestra en la tabla siguiente:

δ	0	1	B
$\rightarrow q_0$	$(q_0, 0, +)$	$(q_1, 1, +)$	$(q_F, 0, =)$
q_1	$(q_1, 0, +)$	$(q_0, 1, +)$	$(q_F, 1, =)$
$*q_F$			

Se puede observar que la máquina acepta cualquier cadena $\omega \ \forall \ \omega \in \Sigma^*$, donde Σ^* es el conjunto de todas las palabras formadas por la concatenación de los miembros de Σ^*, además el cabezal debe estar posicionado en el primer símbolo a la izquierda de la cinta y en estado q_0. Un aspecto importante de la máquina descrita anteriormente es el hecho de que, aunado a que acepta cualquier cadena de $1's$ o $0's$, cambia el primer símbolo B dependiendo del estado en que se encuentre al momento de leer la celda con B, por un 1 si esta en estado q_1 o un 0 si estaba en el estado q_0. Es decir, si se ingresa la cadena $\omega = 0101100BBB\ldots$, se obtiene la cadena de salida $\alpha = 01011001BB\ldots$

Es importante hacer notar que el modelo estudiado es una máquina que acepta lenguajes, y a partir de ahí puede modificarlos o simplemente aceptarlos. El uso de la máquina de Turing no está restringido solamente a ese tipo de aplicaciones, por ejemplo, puede hacer operaciones aritméticas básicas. La forma de representar una suma a través de una MT es por medio de la siguiente abstracción: se pueden representar los números enteros con una cadena de símbolos, con la cantidad de 1's del número que queramos representar; así, el número "3" decimal se representaría como "111" y el "5" como "11111", y podemos asignar algún símbolo intermedio para la operación requerida o bien una cantidad de ceros predeterminada [22].

Hasta el momento se ha planteado sólo un tipo de MT, sin embargo, existen variaciones como una máquina multicintas, no-determinística o multidimensionales, entre otras. Para una mayor profundización en el tema puede consultarse [22]. Es importante remarcar el hecho de que todas estas máquinas poseen el mismo poder de cálculo que la máquina aqui presentada.

2.2.2 Universalidad y el problema del paro

La MT es la máquina abstracta más general, esto se debe a que puede resolverse cualquier algoritmo con su uso. Lo que implica que tiene el mayor poder de cálculo, por encima de otras máquinas. Las computadoras convencionales están basadas en el modelo descrito anteriormente, lo cual demuestra el gran poder que tiene. Sin embargo, también es posible que una MT simule el comportamiento de cualquier otra MT, eso es lo que se conoce como *Máquina de Turing Universal*. La máquina debe recibir como entrada la descripción de la MT a simular y una cadena (ω) de entrada a analizar; la simulación se dará de transición a transición hasta que la cadena ω sea aceptada llegando a un estado final, en caso de no ser aceptada se llegará a un estado no final o bien a que la máquina nunca se detenga.

Es una realidad que la MT puede resolver cualquier algoritmo, no obstante no puede resolver cualquier problema. A este respecto la computabilidad y la complejidad, ramas de la teoría de la computación, estudian los problemas que pueden ser resueltos y la dificultad que conlleva hacerlos, respectivamente. En este caso se haría con referencia a una MT pero es posible tomar como referencias otro tipo de máquinas como los sistemas L.

Dentro de los problemas que no puede resolver una MT se hace casi necesario abordar uno de los problemas más estudiados: el problema del paro.

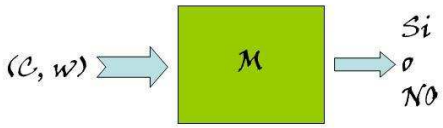

Fig. 2.3: Máquina de Turing $\mathcal{M}$.

Este problema plantea la existencia de una MT $\mathcal{C}$, con la secuencia de entrada ω, aquí la pregunta es: ¿Parará la máquina $\mathcal{C}$ en algún momento después de darle la secuencia ω, o permanecerá trabajando por siempre? La pregunta por si misma es bastante natural y he ahí lo interesante de este problema.

La solución de dicho problema es la existencia de una máquina que pueda decir si la máquina $\mathcal{C}$ con la entrada ω, parará en algún momento. La máquina con las características antes mencionadas se muestra en la figura 2.3. Donde ($\mathcal{C}$, ω) representan la descripción y entrada de la máquina $\mathcal{C}$ respectivamente, mismos que son los valores de entrada de la máquina $\mathcal{M}$; y las salidas posibles son Si, para cuando la máquina $\mathcal{C}$ pare y No, para cuando entre en un ciclo. La forma de comprobar su existencia es por medio de una contradicción, para ello se necesita otra MT para simular la máquina $\mathcal{M}$, esta sería una máquina universal de Turing, la máquina $\mathcal{Z}$. La máquina $\mathcal{M}$ tiene la propiedad de recibir ($\mathcal{C}$, ω), y pasárselo a $\mathcal{Z}$, por lo que $\mathcal{Z}$ llegará a un estado final si $\mathcal{M}$ no termina, y no finalizará nunca si $\mathcal{M}$ si termina. En la figura 2.4 se expone mejor.

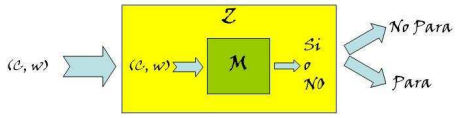

Fig. 2.4: Máquina de Turing $\mathcal{Z}$.

De esta manera, al hacer que $\mathcal{Z}$ tenga como entrada a si misma, es decir, que ($\mathcal{C}$, ω) sea remplazado por ($\mathcal{Z}$, ω) podremos observar la contradicción en el sentido de que cuando $\mathcal{Z}$ pare, la máquina $\mathcal{M}$ entregará como salida un

"Si", lo que llevará a que la máquina $\mathcal{Z}$ no pare cuando se suponía esta había finalizado. Por lo tanto, la existencia de la máquina $\mathcal{M}$ es imposible y de esta manera podemos ver que una máquina de Turing puede ser muy útil en muchas circunstancias, no obstante, existen problemas que no podemos solucionar por este medio.

2.2.3 Máquinas finitas

La MT es la máquina más poderosa, no obstante, no es la única dado que existen otros modelos que se desprenden de la teoría de máquinas, tales como, los autómatas finitos, autómatas de pila, las máquinas de Moore y Mealy. Estos modelos no tienen la misma capacidad de computación que la MT, sin embargo, son modelos muy utilizados debido a la naturalidad de su concepto y la fácil implementación.

Aquí se presentan los autómatas finitos como un ejemplo de una máquina sencilla con un bajo poder de computación y una gran utilidad.

Los autómatas finitos, en adelante AF, son un modelo matemático con el que se puede representar sistemas con entradas y salidas discretas. Como ejemplo, se puede imaginar un expendedor automático de refrescos; el comportamiento total del expendedor puede ser representado con un AF. Como parte de las entradas se tiene la elección del cliente y el dinero insertado, y como salida el producto y el cambio en caso de existir. Para fines didácticos se reduce el ejemplo a analizar únicamente la parte referente a la inserción de dinero y determinar si es la cantidad correcta. Para esto se debe definir el costo del producto y la denominación a aceptar por la máquina expendedora, por ejemplo, el costo puede ser de 5 pesos y las monedas que puede aceptar son de 1, 2 y 5 pesos; además de definir las entradas se debe puntualizar cual será la salida del sistema, en este caso una delimitación puede ser si el dinero es suficiente para el refresco o no. De esta manera se pueden buscar muchos ejemplos en la vida cotidiana en donde los AF tienen aplicación, ya sea en el control de un elevador o en el analizador léxico de nuestro lenguaje de programación favorito.

Dentro de los AF existen dos tipos, uno son los AF determinísticos o bien AFD y los AF no determinísticos, o AFN. A pesar de ser diferentes estas dos vertientes son de igual capacidad de computación, o lo que es lo mismo, si existe un AFD que represente un sistema, existe de igual manera su correspondiente AFN para el mismo sistema.

La definición formal de un AFD se da como la quíntupla:

$$(Q, \Sigma, \delta, q_0, F) \tag{2.3}$$

Donde:

- Q es el conjunto de estados posibles para el autómata.

- Σ es el alfabeto de entrada.
- δ es la función de transición.
- q_0 es el estado inicial del autómata.
- F es el conjunto de estados finales.

Para visualizar de una manera más sencilla un AF suele representarse mediante grafos dirigidos, donde sus vértices representan los estados del autómata, y las flechas sus transiciones de un estado a otro, en la figura 2.5, se presena un autómata de dos estados, A y B, donde se marca el estado inicial a través de una pequeña flecha, en este caso en A. La flecha que va de A a B representa una posible transición mediante el símbolo 1.

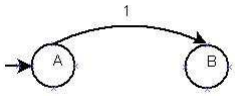

Fig. 2.5: Autómata finito.

Esta manera de ver los AF se denomina diagrama de transiciones y es la más utilizada debido a su practicidad. Para resolver el ejemplo de la inserción de monedas en la máquina expendedora el autómata inicia con un estado arbitrario y posteriormente se iran agregando estados conforme sean necesarios, para los fines del problema basta con tener los estados necesarios para que se pueda saber cuando ya se han ingresado 5 pesos. En la figura 2.6 tenemos la primera parte del autómata donde a partir de un estado A, pasa al estado B siempre y cuando se ingrese una moneda de 1 peso; en caso de que la moneda ingresada sea de 2 pesos, se pasa al estado C y si es de 5 pesos la transición se efectua al estado D.

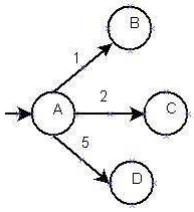

Fig. 2.6: Autómata finito de la inserción de la primer moneda.

De acuerdo a la premisa dada en el planteamiento del problema, el producto cuesta 5 pesos. En la transicion de 5 pesos ya no es necesario ingresar más monedas, es por eso que no existen más transiciones después del estado D. Para

continuar con la creación del autómata debe pensarse detenidamente que monedas aceptará la máquina después de la inserción una moneda de 1 y 2 pesos. El precio total es de 5 pesos y no se contempló la devolución de cambio, no es necesario colocar la transición de 5 pesos después de los estados B y C, pero si son necesarias las de 1 y 2 pesos para ir acumulando las monedas y al final tener las necesarias para sumar 5 pesos. El autómata hasta este paso se muestra en la figura 2.7

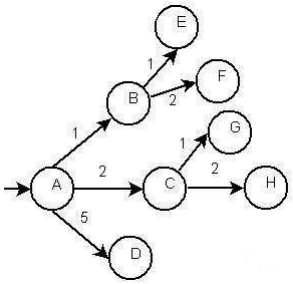

Fig. 2.7: Autómata finito de la segunda moneda.

El autómata termina cuando se hayan acabado todas las posibles transiciones. En la figura 2.8 se puede observar el autómata terminado, se puede ver que solamente cuando existen 5 pesos exactos se da por finalizado el autómata. De acuerdo con la figura los estados finales son todos aquellos donde ya se tiene el resultado deseado. Para el ejemplo, el conjunto de estados finales F es: F=L, N, O, P, R, S, T, U, D. Entonces el conjunto de estados Q son todos los estados que se crearon, es decir, Q=A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U. El alfabeto utilizado son las denominaciones de las monedas: $\Sigma = 1, 2, 5$. El estado inicial es A y finalmente se puede hacer una tabla, como la utilizada para la MT, para saber las transiciones de un estado a otro, véase la tabla 2.1, sin embargo, no es común su uso.

	→A	B	C	∗D	E	F	G	H	I	J	K	∗L	M	∗N	∗O	∗P	Q	∗R	∗S	∗T	∗U
1	B	E	G	-	I	M	Q	**U**	K	**O**	**L**	-	**P**	-	-	-	**S**	-	-	-	-
2	C	F	H	-	J	**R**	**T**	-	**N**	-	-	-	-	-	-	-	-	-	-	-	-
5	**D**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tab. 2.1: Tabla de transiciones de él AFD.

Por convención se utiliza una flecha (→) para indicar el estado inicial y una estrella (∗) para los estados finales; los estados remarcados en negrita dentro de la tabla son todos los estados finales. De la tabla se puede observar que no se definen todas las combinaciones de estado y símbolo en el diagrama de transiciones. Sólo se han definido aquellos que llegan a un estado final. Se

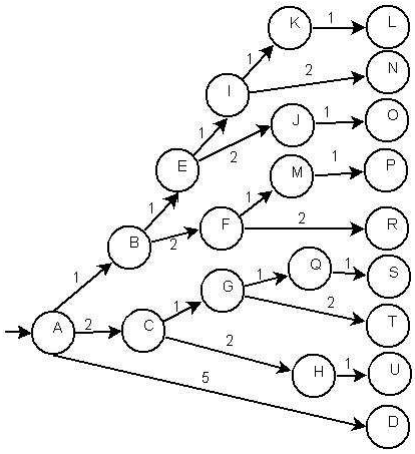

Fig. 2.8: Autómata finito completo.

puede imaginar un estado de no-aceptación para todos aquellos estados que no pueden existir en el autómata, de esta manera se tienen definidos todos los posibles estados a los que puede llegar el autómata; dicho estado de no aceptación puede representar un mensaje para el cliente, como por ejemplo, que aún no ha ingresado el dinero suficiente.
El AF mostrado un AFD, y tiene como característica principal que sólo existe una transición con un símbolo para pasar de un estado a otro. Esta característica es la que hace la diferencia entre los AFD y los AFN. Un AFN puede tener una transición definida con más de un símbolo o bien con el símbolo ϵ, que representa la cadena vacía. La definición formal de un AFN es igual que la del AFD sólo cambia la función de transición, dicho cambio radica en que el AFN puede tener desde la cadena vacía hasta 2^Q símbolos para una misma transición. Como ejemplo de este tipo de AF podemos hacer un autómata que genere números con o sin punto decimal, por ejemplo, 45.8, 19, 34.987, 5. En la figura 2.9 se muestra el autómata que genera estos números.

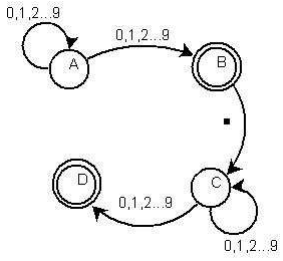

Fig. 2.9: AFN para los números reales positivos.

Del diagrama de transiciones cabe resaltar que los estados remarcados son estados finales, y de la misma manera que con los AFD una pequeña flecha indica el estado inicial. La forma de saber si un número es aceptado por este autómata es ingresarlo caracter a caracter, es decir, de un número cualquiera se toma el caracter que tienen más a la izquierda, se observa si existe una transición para dicho símbolo en el estado inicial, si existe se continua con el siguiente caracter a la derecha pero ahora en el estado que devolvió la transición anterior. Por ejemplo, para verificar la cadena "15", primero se toma el símbolo "1" y se observa si existe una transición para el caracter "1" en el estado inical, el estado A. Al existir 2 transiciones con el estado A se debe comprobar ambas para saber si con alguna de las dos es posible hacer que la cadena sea aceptada. Si se elige la transición de "A" a "B" con el símbolo "1" el autómata rechazará la cadena de entrada pues no existe una transición en el estado B que acepte el símbolo "5"; todo lo contrario si se elige la transición que va de "A" a "A" con el símbolo "1", pues también existe una transición que lleva a un estado final que acepta el segundo símbolo de la cadena, el 5.

Al ingresar cualquiera de los números antes expuestos se puede comprobar que el autómata funciona y además, es importante destacar que tiene dos transiciones con los mismos símbolos; se puede pasar, por ejemplo, del estado "A" al estado "A" con el símbolo "3" y además con ese mismo símbolo es posible pasar del estado "A" al estado "B". Este ejemplo puede expandirse para que acepte no solo números con punto decimal sino también con signo, de esta forma el autómata aceptaría todo el conjunto de los números reales.

Para finalizar el estudio de las máquinas es importante mencionar la existencia de las máquinas de Moore y Mealy o máquinas secuenciales, que tienen su principal aplicación en la electrónica. Estas máquinas precedieron a los autómatas finitos y es por eso que básicamente son un autómata finito determinista. La diferencia radica en que no tienen estados finales; en su lugar estas máquinas entregan símbolos de salida. Su funcionamiento general no es complicado, la máquina de Moore tiene la salida de sus símbolos asociados con el estado actual y la máquina de Mealy es a través de la transición que ha llevado. Un estudio más a fondo de estas máquinas puede ser visto en [22] y [14]

2.3 Computación no-convencional

Como se ha visto a lo largo de la primera parte de este trabajo, la computación es un proceso por el cual los datos son manipulados: adquiriéndolos (entrada), tranformándolos (haciendo cálculos) y transfiriéndolos (salida), es decir, cualquier forma de procesamiento ocurrida, ya sea de manera natural o manipulada por algún medio, es una computación [29]. La instancia básica es:

1. Medir una cantidad física.
2. Ejecutar una operación aritmética o lógica.
3. Darle valor a una cantidad física.

Como se puede observar de la sección 2.2.1 esto se cumple en la máquina de Turing y en la unidad aritmética lógica de una computadora convencional; por lo que si una computadora, es sin duda, un objeto bien conocido; se llamará una computadora "no-convencional" si su medio físico o su arquitectura salen significativamente de dicha norma [31].

Es decir, al tomar la definición de lo que es la computación, no se restringe a una representación específica, lo que hace posible pensar en nuevos modelos y conceptos de los procesamientos computacionales basados en elementos más palpables, por ejemplo: las reacciones químicas, dispositivos mecánico-cuánticos; o elementos abstractos, entre los que se pueden encontrar los autómatas celulares y las poblaciones genéticas artificiales. Por ejemplo, si los bits de la computación convencional se representan como la carga o la descarga de corriente en un capacitor, se podrían representar como átomos A y B en una cadena molecular o en una reacción química específica [24].

A simple vista y en las necesidades de la vida cotidiana lo anterior no tendría utilidad alguna, sin embargo, en las ciencias computacionales aún existen muchos problemas que aunque son computables, tienen una muy alta complejidad para poderlos resolver a través de una computadora convencional, debido a que la cantidad de datos de entradas a procesar es muy grande y una computación que se lleva a cabo en serie tardaría mucho tiempo, un ejemplo de ello son los problemas no-polinomiales o la simulación de sistemas físicos, en la que debido a que el procesador realiza una operación y luego otra y otra, hasta terminar con todo el conjunto de partículas que se quieren estudiar se desperdicia la capacidad de procesamiento [31]; aun así, existe una forma de evadir el desperdicio de tiempo computacional para este tipo de problemas llamada *"computación paralela"* [31], [24], [29].

La computación no-convencional se auxilia del paralelismo masivo inherente en [29]:

- Computadoras de reacción difusión.
- Algoritmos genéticos.
- Autómatas celulares.
- Computadoras cuánticas.

Para este tipo de procesamiento de información se puede ver como si se tuviera un conjunto de "miniprocesadores" los cuales pueden realizar operaciones lógicas o aritméticas muy simples, pero que lo hacen en la misma unidad de tiempo; es decir, en lugar de tener un procesador que realiza operaciones seridas en varias unidades de tiempo, hará muchas operaciones en una sola unidad de tiempo, el número de operaciones que se realizarán será el número de "miniprocesadores" que se encuentran trabajando [31], [24].

En la computación no-convencional, los miniprocesadores no tienen la misma velocidad de procesamiento que la de un procesador convencional actual (100 millones de instrucciones por segundo), sin embargo, si se pueden hacer muchas operaciones en una sola unidad de tiempo, el resultado se obtendrá en menos unidades de tiempo lo que hace que el procesamiento sea más rápido [24].

A continuación se dará una breve introducción de los ejemplos de computación no-convencional que se han venido desarrollando e investigando.

Lo que concierne a los modelos abstractos en los inicios de la computación artificial fue la expectacularidad de que se pudiera explotar el paradigma de la evolución biológica para propósitos computacionales, similares a las ideas que tuvo von Neumann de que la computación se podía hacer como un proceso en paralelo.

En años más recientes se han revivido esas ideas y se han transformado en herramientas computacionales, uno de estos paradigmas se llama *"redes neuronales"* [24].

Las redes neuronales son circuitos que consisten en un número grande de elementos simples, diseñados de tal forma que el aspecto más explotado es el comportamiento de cada elemento; históricamente se propusieron como una altenativa de hardware, y se ven diseñados como las redes neuronales de los animales; sin embargo, su diseño es un poco menos complejo; los circuitos que simulan las redes neuronales tienen muchas entradas, de hecho, una de las aplicaciones es en la ayuda de problemas de decisión donde existen muchas entradas.

Otro de los paradigmas es la forma biológica computacional llamada evolución de la que se han desprendido muchas áreas de aplicación como los algoritmos genéticos, programación evolutiva, programación genética; sin embargo, no tiene bien establecidas las bases teóricas debido a la enorme dificultad para entender los algoritmos que emiten la evolución.

Otro de los modelos son los autómatas celulares que son sistemas descentralizados espacialmente consistente de un número grande de componentes identicos simples con funciones locales; tienen la capacidad de simular sistemas que pueden ejecutar computaciones complejas con un alto grado de eficacia y robustes, así como los modelos de los sistemas complejos en la naturaleza. Por ello, los autómatas celulares y arquitecturas similares han sido estudiados exhaustivamente en la física y en la química, además de ser considerados como objetos matemáticos sobre los cuales propiedades formales pueden ser provistos; también han sido usados como dispositivos de computación paralela en simulaciones de modelos científicos y tareas computacionales, por ejemplo, procesamiento de imágenes; los autómatas celulares son incluidos dentro de la clasificación de las "redes iterativas" o "redes autónomas".

Otros modelos más "reales" como los sistemas biológicos realizan la computación apoyándose de estructuras biológicas; si se toma en cuenta que cada célula lee información de una memoria, reescribe, y recibe datos los cuales son los que proporcionan el estado de su medio ambiente; entones, ésta comenzará a actuar, es decir, tendrá una reacción, como resultado de una "computación".

El problema con las computaciones biológicas es que existen otros resultados y otras variables que afectan la computación, es decir, una célula viva no usa ningún dispositivo con el cual se reciban los datos de entrada para realizar una operación; esto podría ser un problema o un motivante para estudiar los algoritmos que se puedan crear para solucionar problemas específicos de la biología y de la biología química.
El modelo computacional biológico más conocido es el que realiza el ADN, que es usado como medio de almacenamiento de información, además de que el RNA mensajero es usado como bus para transportar la información necesaria para que las proteínas, que por momentos pueden actuar como señales receptivas, compuertas lógicas o señales tranductoras entre diferentes formas de señalización, como luz (fenómenos de luminicencia), o sistemas mensajeros químicos (función de RNA mensajero en los seres vivos)[31], [24].

Otro de los modelos físicos es la computación cuántica que es usada en el diseño de sistemas de comunicación y dispositivos semiconductores; ésto nos muestra que no existen diferencias entre los resultados de las operaciones lógicas con respecto a los dispositivos convencionales; sin embargo, estas variables se encuentran en un estado cuántico y un paso computacional es el resultado de un operador evolucionado unitario que actúa en ese estado; esto se debe a que los qbits se comportan de manera diferente de los bits dado que los estados de los sistemas cuáticos se caracterizan por una función de onda compleja, que representa una superposición de los estados, es decir, que existen diferentes niveles en los que se encuentra la información. La teoría de la computación cuántica ha sido estudiada para encontrar soluciones a problemas polinomiales, dada la relación que existe con los sistemas digitales convencionales, además de tener un amplio estudio en el área de la criptografía.

La computación no-convencional para poder realizar las operaciones, necesita, al igual que la computación convencional, algún medio en el cual se desarrollen los procesos computacionales; sin embargo, a diferencia de la computación convencional, la no-convencional utiliza el medio físico como parte importante para la realización de los cálculos [31].

Por ejemplo, la computación convencional utiliza los procesos de amplificación de señales, regeneración de las señales, así como la conversión de tokens. Debido a esto se da una diferencia entre las compuertas lógicas y las compuertas lógico-conservativas, la cual radica en que las convencionales permiten que la salida de una operación sea la entrada de otra, mientras que las conservativas no, debido a que las señales copiadas son hechas por los significados de las com-

puertas, mas no por la transferencia de datos a través de un cable, además de producir resultados *"basura"* y el resultado deseado; un ejemplo de compuertas lógico-conservativas es la compuerta de Fredkin.
Dejando más en claro lo que es la lógica conservativa se dice que es un esquema de computación basado en operaciones discretas (eventos), u objetos discretos (señales) y que estos esquemas satisfacen tres leyes de conservación llamadas:

- *Conservación del número de hilos*: Se refiere a que cada evento tiene muchas entradas y salidas y que la relación existente entre ellas es de uno a uno, es decir, que por cada entrada existe una salida, así cada "hilo" tendrá un estado en un evento, de tal manera que los estados de cada hilo podrán cambiar con respecto al tiempo, pero el número de hilos se conserva.

- *Conservación del número de token*: A la representación de los valores 0 y 1 se le llama token; Los tokens son "cargados" en los hilos que existen en un evento, los tokens pueden cambiar de valor, pero el número de tipos de tokens son invariantes, es decir, que si para representar el 1 se usa un token y para el 0 otro, solo existirán esos tipos.

- *Conservación de la información*: La lógica conservativa es reversible, cada evento establece una relación uno a uno entre los estados colectivos de las señales de entrada y sus señales de salida; como consecuencia, el estado general actual podría conocer únicamente su pasado o el futuro del sistema. Si se conoce el estado inicial del sistema que es expresado como una distribución estadística, entonces esta distribución cambiará como proceso de la computación pero su entropía no.

Dadas estas características una computadora lógico-conservativa podría ser vista como un tapete espacio-temporal en el cual habrá hilos que serán los responsables de hacerlo cambiar de color, pero que todo lo que lo compone (el material, el color y la información) obedecen a una disciplina que es impuesta por las leyes de Kirchhoff.

Ejemplos claros de lo que es la computación lógico-conservativa, son las basadas en colisiones (la computación en bolas de billar), o computadoras de carga y descarga, entre muchas más.

Como se puede apreciar, la computación no-convencional, es un paradigma que sigue las leyes de la computación convencional y que muchos de los modelos no-convencionales pueden ejecutarse en computadoras convencionales, pero que tiene sus propios paradigmas, ventajas y desventajas, para ser usada para la solución de problemas de todas las áreas científicas y de ingeniería.

2.4 Computación basada en compuertas lógicas

Actualmente cuando se escucha hablar de compuertas lógicas inmediatamente se imaginan los símbolos de las compuertas lógicas universales, mostrados en la figura 2.10, y se hace una relación con circuitos electrónicos; esto es debido al gran auge de la electrónica, no obstante, las compuertas lógicas son un modelo abstracto.

En 1854, George Bool escribió un libro titulado *The Laws of Thought*, donde exponía toda una teoría lógica que utilizaba símbolos en lugar de palabras. Posteriormente C. E. Shanon observó que el álgebra de bool se podia aplicar al análisis de circuitos eléctricos, de esta manera nacieron en el área de le electrónica las ahora llamadas compuertas lógicas; aun con esto, así como Shanon aplicó la teroría de Bool en la electrónica, se puede aplicar a diferentes áreas de la ciencia donde sea posible hacer una distinción lógica, como falso y verdadero, o '0' y '1' [28].

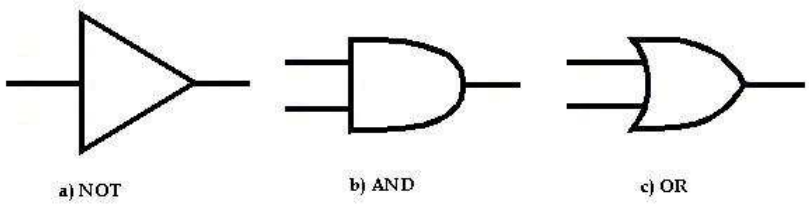

Fig. 2.10: Símbolos de las compuertas lógicas universales.

El comportamieno de las compuertas lógicas es representado mediante las llamadas *tablas de verdad*, que son tablas donde se indican los valores de entrada de la compuerta y sus valores de salida para la combinación de entrada respectiva. Las tablas de verdad para las compuertas lógicas mostradas en la figura 2.10 son expuestas a continuación.

Entrada	Salida
0	1
1	0

Tab. 2.2: Tabla de verdad de la compuerta NOT.

Bases de la implementación clásica

Una de las maneras de representar este modelo ha sido mediante los circuitos electrónicos hechos con germanio y silicio, dos materiales semiconductores, lo

Entrada A	Entrada B	Salida
0	0	0
0	1	1
1	0	1
1	1	1

Tab. 2.3: Tabla de verdad de la compuerta OR.

Entrada A	Entrada B	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Tab. 2.4: Tabla de verdad de la compuerta AND.

que significa que pueden o no conducir la corriente eléctrica dadas algunas circunstancias específicas. El diodo es el dispositivo más simple diseñado y que sirve para ejemplificar la construcción de compuertas lógicas. El diodo es fabricado mediante la unión de dos tipos de materiales denominados "P" y "N", que hacen referencia a su carga, es decir, positivo y negativo respectivamente. En la figura 2.11 se muestra la estructura de materiales del diodo y su símbolo, donde "A" representa el anodo o parte positiva y "K" representa el cátodo o parte negativa.

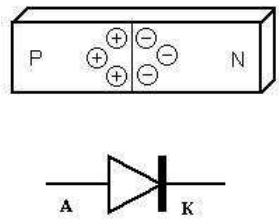

Fig. 2.11: Diodo semiconductor.

El funcionamiento del diodo es el siguiente: permite el paso de corriente siempre y cuando tenga una carga positiva en el anodo y una carga negativa en el cátodo, de lo contrario se comportará como un aislante, es decir, obstruirá el paso de la corriente. Esto se debe a las propiedades de los semiconductores; al "contaminarse" un semiconductor como el silicio con otros materiales como el arsénico o el indio obtiene las características de alguno de los dos tipos de materiales: 'P' o 'N'.

Si se une con arsénico el material gana electrones y surge el material tipo N; por otro lado, si se contamina con indio en el material surgen "huecos" para

que pueda dar lugar a una obtención de electrones, todo esto en la última capa del enlace de valencia del átomo de cada material. El lugar donde se unen los materiales se denomina *juntura p-n*, y es donde, al colocar cargas en los extremos del diodo puede hacer que el dispositivo sirva como conductor o como no-conductor al seguir las leyes de las cargas. De esta manera se crean los diodos.

La manera de representar una compuerta lógica con el diodo se puede observar si se conecta el diodo con dos entradas: una X y otra Y; una caída de voltaje que puede ser una resistencia, y una salida S. El circuito antes descrito se muestra en la figura 2.12.

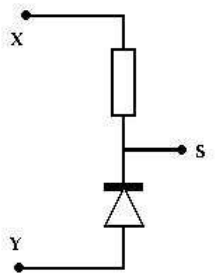

Fig. 2.12: Compuerta lógica AND mediante un diodo semiconductor.

De la figura 2.12, cuando el diodo conduce, pasa corriente a través de él y de la resistencia de carga conectada en serie, estableciendo una caída de voltaje en la resistencia. Esta caída de voltaje separa el voltaje de entrada X de la salida S, que se toma de la unión de la resistencia de carga y del anodo del diodo. En contraste, la caída de voltaje en el diodo se puede tomar prácticamente como cero. De esta manera, cuando existe conducción en el diodo la salida S está conectada por el diodo directamente al voltaje de entrada Y, por lo tanto es igual a Y. Por el contrario, cuando el diodo no conduce, no hay paso de corriente ni caída de voltaje en la resistencia de carga. El voltaje de entrada Y, se encuentra "desconectado" por medio del diodo no-conductor por lo tanto el voltaje de salida será igual a X.

De esta manera si se representa la presencia de voltaje con un uno '1' y la ausencia de voltaje con un cero '0' se puede observar las combinaciones de entrada como se muestra en la tabla 2.5. Si se observa con cuidado se verá que la tabla 2.5 es idéntica a la tabla 2.4, que es la tabla de verdad de la compuerta AND, por lo tanto se ha construido mediante circuitos electrónicos una compuerta lógica. Sin embargo, no es el único camino para la representación física de las compuertas lógicas, aunque si la más utilizada debido a los beneficios que brinda actualmente. Cabe destacar que las compuertas lógicas no utilizan el circuito presentado aqui, sino uno un poco más complejo, pues se hace necesario

tener la certeza del resultado y con el presente circuito aun existen posibles fallas.

X	Y	S
0	0	0
0	1	0
1	0	0
1	1	1

Tab. 2.5: Tabla de verdad del circuito para la compuerta lógica AND.

Otras formas de implementación

Las investigaciones para la implementación de compuertas lógicas continúa. En este marco de investigación en el 2004 vio la luz el trabajo titulado *"Magnetic logic gate for binary computing"* [2] de S. Anisul Haque *et. al.*, en el cual se expone una forma diferente de represenar las compuertas lógicas [9].

En dicho trabajo se manifiesta la existencia de "nanodots" ferromagnéticos, que después de los estudios realizados se vió que la mejor opción eran aquellos con una forma elíptica. Otra propiedad de los nanodots son sus vectores magnéticos; estos vectores son interpretados de la siguiente manera para poder hacer la computación binaria: si los vectores magnéticos estan orientados hacia $+x$ entonces representa un '1' lógico, y un '0' lógico se interpreta cuando los vectores apunten hacia $-x$. Esquemáticamente aparecen como flechas que indican su dirección en el plano cartesiano, es decir, a la derecha para $+x$ y a la izquierda para $-x$.

Con estas bases se muestra la forma en que se pueden construir las compuertas NAND y NOR, que son las negaciones de las compuertas AND y OR, respectivamente. La manera de construir la compuerta se puede ver en la figura 2.13, donde se tienen 4 nanodots, 2 de entrada, uno de salida y otro para la elección de la compuerta.

De la figura 2.13 se puede observar que el nanodot que se debe "leer" para saber el resultado es el Z; los nanodots B y C son las entradas que alimentan el sistema y por último el nanodot A se utiliza para elegir entre la compuerta NAND y NOR, de tal manera que si A vale '0' la compuerta elegida es la NOR y si es un '1' se elige la compuerta NAND. Un ejemplo del funcionamiento de la compuerta se muestra en la figura 2.14, donde se exhiben los cuatro nanodots, en color rojo los que tienen un valor '1' y en verde los que valen '0'.

Como se puede apreciar de la figura 2.14 la compuerta elegida es la NAND, los valores ingresados son '0' y '1', que dan como resultado un 1. También se

[2] Las imágenes mostradas sobre compuertas lógicas magnéticas están basadas en las presentadas en [9]

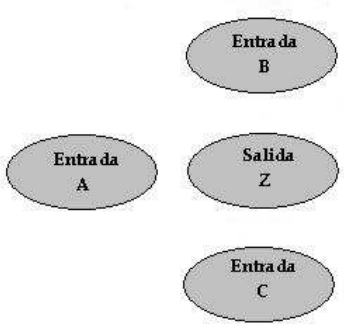

Fig. 2.13: Orden de nanodots para construcción de compuerta lógica.

muestra la velocidad de procesamiento que se tuvo al ejecutar esa operación, que en este caso fue de 2.0 ns; la velocidad de procesamiento es una característica importante a destacar de este trabajo, que por lo mismo promete ser una buena vía para la computación, no obstante aún hay mucho que investigar en esta área, así como realizar la construcción de circuitos más complejos.

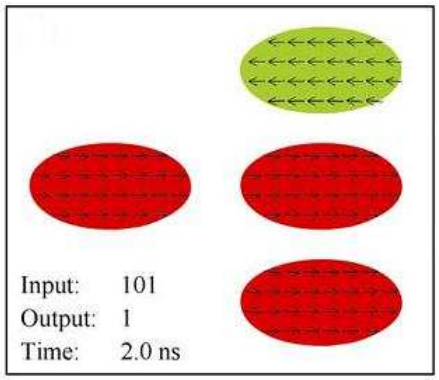

Fig. 2.14: Compuerta lógica magnética.

A lo largo de la sección se han presentado dos ejemplos de implementaciones de computación basada en compuertas lógicas. En el presente trabajo se utilizarán las compuertas lógicas como medio para desarrollar la computación y además centra su interés en un modelo perteneciente a la computación basada en autómatas celulares para llevar a cabo dicha computación; esto gracias a que se ha mostrado que los autómatas celulares pueden soportar la computación basada en compuertas lógicas, como por ejemplo, la regla Life que será presentada en 3.3.

Del capitulo "Computacion no-convencional", es importante recordar que la computación puede ser paralela y seriada. Las compuertas lógicas pueden

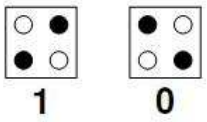

Fig. 2.15: Representación de bits en computación cuántica.

ser implementadas de diversas maneras, y la computación paralela tiene como ejemplo la compuerta *majority* (mayoritaria), donde se pueden hacer multiples operaciones lógicas en los mismos tiempos de reloj, una gran ventaja que las implementaciones clásicas no tienen.

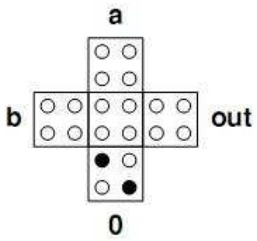

Fig. 2.16: Compuerta lógica AND cuántica.

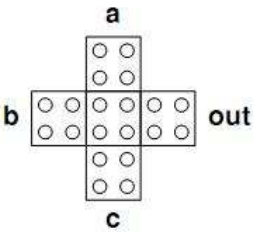

Fig. 2.17: Compuerta lógica mayoritaria.

La compuerta majority básica realiza una operación con tres entradas, y los tiempos de procesamiento son los mismos que cuando se procesan 2. Como ejemplo de la compuerta existe el trabajo realizado por *Whitney J. Townsend and Jacob A. Abraham* titulado "Complex Gate Implementations for Quantum Dot Cellular Automata"[3]. En este trabajo la manera de representar los bit '0' y '1' se muestra en la figura 2.15. Y en la figura 2.16 se muestra como se implementa la compuerta AND. Como puede observarse solo se utilizan dos de las tres posibles entradas, la tercera es utilizada para cambiar entre la compuerta

[3] Las imágenes mostradas son tomadas de [34]

que se desea elegir, es decir, entre AND y OR. Sin embargo, al utilizar esta entrada como una variable más es posible procesar las tres entradas en el mismo tiempo. Es importante este comportamiento debido a que se hace de manera paralela. Una profundización en el tema y construcciones más complejas son presentadas en [34].

Los autómatas celulares son inherentemente paralelos, por lo mismo es posible la computación paralela, de manera similar que con la compuerta mayoritaria, de esta manera el procesamiento se realiza con mayor rapidez. Ésta característica de los autómatas celulares permite ver como prometedora la computación sobre este modelo. Un ejemplo de la computación paralela en los autómatas celulares es presentado en el A.

3. TEORÍA DE AUTÓMATA CELULAR

3.1 Etapas importantes

3.1.1 Precursor John von Neumann

A finales de los 40's John von Neumann desarrolló trabajos donde dejaba ver su interés por autómatas complicados, esto se refleja en su escrito "Theory and Organization of Complicated Automata", que unido a su posterior trabajo "The Theory of Automata: Construction, Reproduction, Homogeneity" brindan las bases de lo que hoy llamamos *teoría de autómta celular*. El objetivo de von Neumann era representar sistemas naturales tales como el sistema nervioso humano. Bajo esta premisa definió un espacio dividido en celdas donde cada una podría estar en un estado de un conjunto finito de estados; el sistema evolucionaría en tiempos discretos, así cada celda calcularía su siguiente estado por sí misma, la forma de calcular el siguiente estado es a través de una función de transición, que depende de las celdas aledañas a la celda estudiada, que sería la misma para todas las celdas del espacio [33].

La primera replica de autómata propuesto por von Neumann fue compuesta por un arreglo bidimensional y la estructura de la replica fue hecha por miles de células elementales. Cada una de las células obtiene veintinueve estados posibles. La regla de evolución requiere conocer los estados de cada una de las células, además, de las células vecinas localizadas al Norte, Sur, Este y Oeste.

El trabajo realizado por von Neumann fue inconcluso debido a su repentino fallecimiento, dejando sólo las bases de algo que con el curso de los años ha crecido y se ha fundamentado en diversas áreas de la ciencia.

3.1.2 Life de John Horton Conway

En 1970, el matemático John Conway propuso, el ahora famoso autómta celular, "Game of Life" (Juego de la vida), coloquialmente llamado Life. Su motivación fue el encontrar una regla de autómata celular que fuese simple pero a la vez tuviese una dinámica compleja. Esto lo logró a través de un autómata bidimensional donde cada célula podría estar en alguno de los dos estados definidos: vida o muerte. La manera en que evoluciona este autómata es por medio de tres simples reglas. Para comprobar la segunda condición, la dinámica compleja, Conway estudió lo que resultaba de evolucionar el sistema dadas configuraciones iniciales aleatorias [13].

Al hacer esto se puede observar que la regla es impredecible, es decir, que su comportamiento no puede ser conocido con anterioridad y gracias a esto se dice

que Life es complejo. Con el tiempo y el constante estudio sobre la regla se han descubierto diversas partículas que interactúan para la generación de patrones singulares así como para la implementación de computación.

A pesar de la popularidad del autómata éste no paso en su momento de una simple curiosidad matemática, no obstante con el tiempo se vío la importancia que tenía.

3.1.3 AC en una dimensión por Stephen Wolfram

Llegados los años 80's el estudio de autómatas celulares se había extendido, aun así, no se tenía un estudio completo y estructurado, mucho menos una clasificación describiendo el comportamiento de los autómatas celulares. Wolfram propuso en esta década una clasificación que con el tiempo llevaría su nombre.

Wolfram se dio cuenta de que los autómatas celulares eran algo demaciado grande para ser estudiado por completo, es por eso que enfocó su estudio a lo que llamó el autómata básico o más sencillo. Tomó los autómatas de una dimensión, con 2 estados (cero y uno). Con estas bases definió todas las reglas posibles, utilizando como vecindad solamente la célula central y sus dos células adyacentes. Así tuvo un total de 256 reglas que estudió y clasificó y gracias a las cuales llegó a la ahora conocida clasificación de Wolfram [30]; esta clasificación tiene cuatro grupos:

1. El comportamiento es simple y todas las condiciones iniciales llegarán a un estado homogéneo.

2. Existen muchos posibles estados finales, pero todos ellos pertenecen a un cierto conjunto de estructuras simples que o son los mismos siempre, o se repiten en pocas iteraciones.

3. El comportamiento es más complicado, y se puede ver en varios aspectos aleatorio. Ésta clase se dice que tiene comportamiento caótico

4. Es la unión de los tres puntos anteriores, es decir, se observan partes en caos, ciclos y homogeneidad al mismo tiempo.

Como se deja ver de lo anterior la clasificación esta basada en el comportamiento que tiene el autómata al evolucionar. Esto permite vislumbrar un poco más las posibilades de estos sistemas; por ejemplo, la regla 30 estudiada por Wolfram se encuentra dentro de su clasificación en el tercer grupo, es decir, tiene un comportamiento caótico; esta característica permite que pueda ser utilizado para la generación de números aleatorios.

Esta clasificación se puede extender para los autómatas de otras dimensiones y con un mayor número de estados; cabe destacar que al no haber un estudio específico de las demás dimensiones y estados no puede saberse a ciencia cierta si la regla que estemos estudiando pertence a la clasificación de Wolfram, aun así esta clasificación se ha venido convirtiendo en un punto de referencia al momento de querer nombrar el comportamiento de los autómatas celulares.

3.2 Definiciones básicas

Los autómatas celulares son modelos de sistemas complejos en los cuales el tiempo y espacio son determinísticos, y evolucionan a través de una regla local isotrópica.

La dinámica de los autómatas celulares son basadas en las siguientes observaciones de los sistemas físicos:

- La información viaja a través de una distancia finita en un tiempo infinito.
- Las leyes de la física son independientes del observador.

A esta lista von Neumann agregó el simplificar con tiempos y espacios discretos. Esta definición se puede extender hasta cualquier número de dimensiones [21].

Con esto se quiere decir que los autómatas celulares se conforman de tres partes [1]:

- Una lattice (malla) discreta, o un arreglo finito de celdas (donde estarán las células) de una porción de un espacio d-dimensional.
- Una vecindad mediante la que se determina el estado de cada célula en un tiempo t.
- Una regla de evolución o una tabla de transición de estados, la cual especifica la forma de evolución de los estados.

A continuación se definen con mayor detalle las partes antes mencionadas.

Regla de evolución. Una regla de evolución es aquella en la cual tomando en cuenta los estados de las células vecinas, la célula central "evolucionará" a un estado en el siguiente tiempo $t + 1$.

Vecindad. Conocemos que una regla de evolución es local por definición. La "evolución" de una célula dada requiere conocer el estado de las células que estan a su alrededor. Por lo que, la región espacial en el cual una célula necesita conocer sus estados se le llama "vecindad".

En principio no existen restricciones para el tamaño y forma de una vecindad, excepto que tiene que ser la misma para cada célula, y el radio de una vecindad se define como la cantidad máxima de células que se toman en cuenta a partir de la célula central hacia alguna de las direcciones que pueda tomar el espacio; es decir, si se propone un número de vecinos de radio 1 significa que una vecindad en un espacio unidimensional, de una célula la conprenderán una célula a su izquierda, y una a la derecha. Pero como la forma puede variar, el radio de, por ejemplo, una vecindad en forma de L, que tome dos células a la derecha y tres hacia abajo, se dirá que es una *subvecindad* de radio 3 [1].

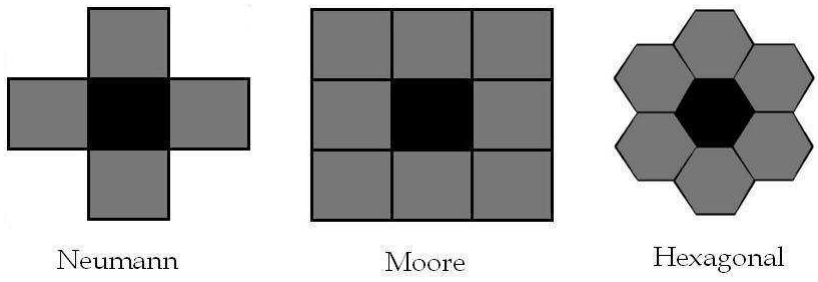

Fig. 3.1: Tipos de vecindades en 2-dimensiones.

En la figura 3.1 se muestran tres tipos de vecindades diferentes para autómatas celulares en dos dimensiones, a continuación se explican detalladamente dichas vecindades.

Vencidad de von Neumann. Consiste en una célula central y considera sus 4 vecinos geográficos más próximos (norte, sur, este y oeste).

Vecindad de Moore. Moore toma la vecindad de von Neumann y agrega los vecinos del noreste, noroeste, sureste y suroeste y en total la vecindad tiene ocho células.

Vecindad hexagonal. En esta vecindad las células tienen una forma hexagonal y en ella se toman en cuenta las seis células adyacentes a la célula central.

Todo lo descrito anteriormente se puede definir formalmente como que un atuómata celular es determinado por una cuatrupla $A = (S, d, N, f)$ donde:

- S es un conjunto finito de estados.
- d es un entero positivo que representa la dimensión.
- $N \subset \mathbb{Z}^d$ es la configuración inicial de las células, es decir, un conjunto finito de estados para cada célula en $t = 0$.
- $f: S^N \to S$ es una función de transición local.

Por lo que la función global de transición se representa de la siguiente manera: $G_f: S^L \to S^L$ está definida como $G_f(c)_v = f(c_{v+N})$.

Los autómatas celulares son un mundo tan grande que su clasificación ha presentado todo un reto a lo largo de su historia. Una de las formas mas sencillas de clasificarlos es mediante la dimensión en la que evolucionan. En la definición

formal dada anteriormente se mencionó un entero positivo que representa la dimensión. La dimensión en la que puede evolucionar un autómata celular va desde la primer dimensión (lineal) hasta la dimensión n, siendo $n \in \mathbb{Z}$. En la figura 3.2 se muestran las lattices para la primer y segunda dimensión, y en la figura 3.3 se muestra un ejemplo de un autómata celular en 3 dimensiones.

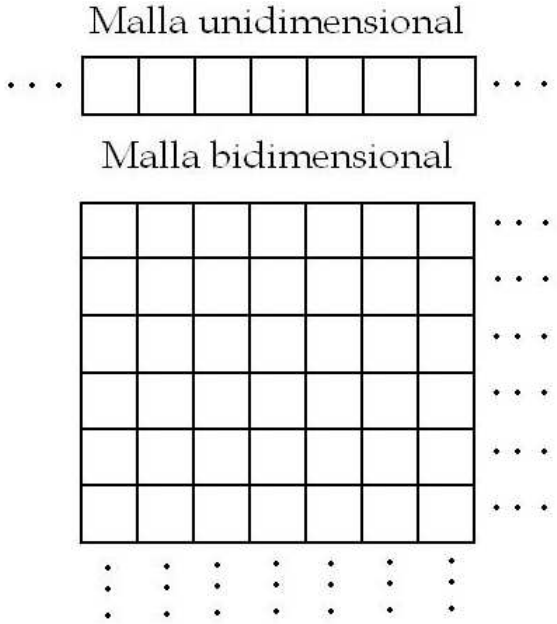

Fig. 3.2: Autómata celular en 1 y 2 dimensiones.

3.3 El Juego de la vida

Anteriormente se ha dado una breve introducción al trabajo realizado por Conway, en esta sección se detallará un poco más sus resultados. [1]

3.3.1 Dinámica de la regla

Conway imaginó un arreglo de dos dimensiones, como un tablero de ajedrez en el cual cada célula puede estar viva (color negro), o puede estar muerta (color blanco). Utilizando la vecindad de Moore para células cuadradas definió la regla de evolución de la manera siguiente:

[1] Las imagenes mostradas sobre partículas para la ejemplificación de Life fueron realizadas en el simulador *Golly* de Andrew Trevorrow y Tomas Rokicki.

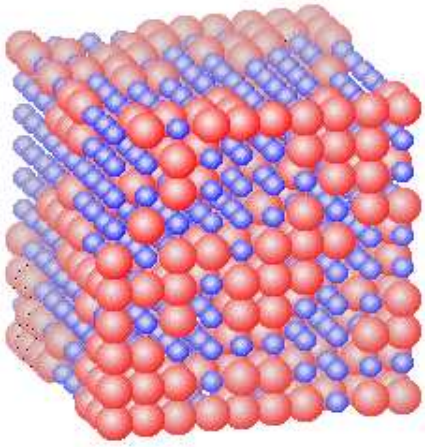

Fig. 3.3: Autómata celular en 3 dimensiones.[2]

1. Una célula muerta, vivirá si y solo si tiene 3 células vivas a su alrededor.
2. Si una célula viva tiene menos de 2 o más de 3 células vivas a su alrededor, muere por aislamiento o sobrepoblación, respectivamente.
3. Si una célula viva tiene 2 o 3 células vivas a su alrededor continuará con vida, es decir, sobrevivirá.

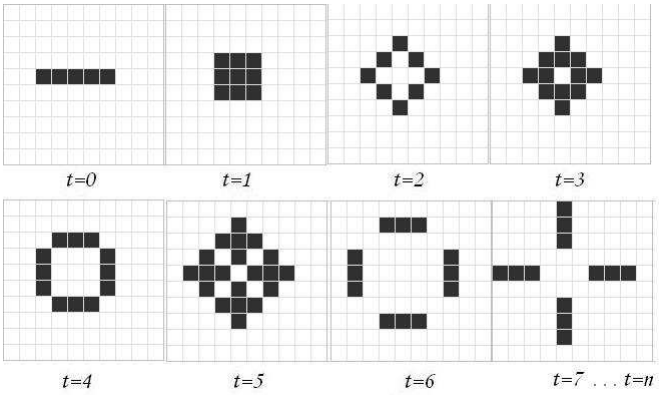

Fig. 3.4: Osciladores en Life.

[2] Imagen tomada de http://cellular.ci.ulsa.mx/miscelanea/viewforum.php?f=16.

3.3.2 Ilustrando patrones o construcciones complejas

Estas simples reglas son las que lograron el objetivo del comportamiento complejo; parte de la complejidad que tiene esta regla de autómata celular se debe a las "partículas" que soporta en su medio. Estas partículas han sido estudiadas y clasificadas. En la figura 3.4 se muestra el ejemplo más sencillo de una partícula oscilando, esta partícula recibe el nombre de "blinker". Si se observan cuidadosamente las evoluciones, en cada tiempo se realiza el cálculo para cada célula dentro de la lattice, así al final se tiene una partícula que cambiará entre dos "estados", y este cambio se realizará hasta el tiempo n.

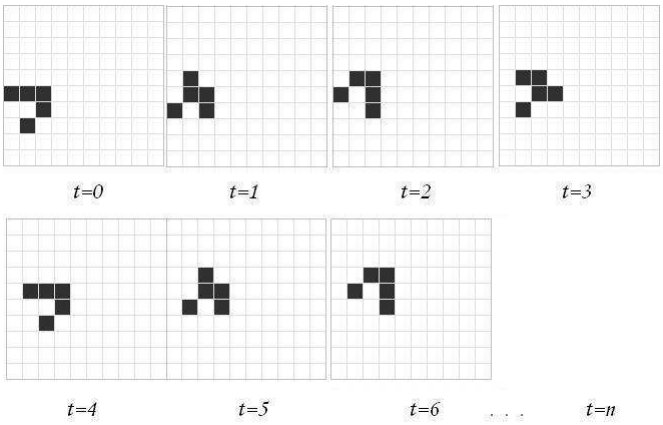

Fig. 3.5: Un glider en Life.

Existen otras partículas de un mayor interés en Life, como lo son los "gliders" y los "gliders-gun". Un glider es una partícula que se puede mover a través del espacio de evoluciones. En Life los gliders se mueven en líneas de 45^o, se muestra un ejemplo de su evolución en la figura 3.5. Con el conocimiento de lo que es un glider surge la pregunta: ¿Cómo es que se generan los gliders? La respuesta son los gliders-gun, una partícula más compleja que después de un determinado número de evoluciones lanza un glider. Se puede observar un glider-gun en la figura 3.6; cabe aclarar que los gliders son lanzados con una inclinación de 45^o.

3.3.3 Life es universal

Como las partículas antes vistas existen muchas otras, sin embargo la importancia de estas dos últimas se debe a que son las partículas que Conway utilizó para la creación de las compuertas lógicas básicas, mostrando así la universalidad lógica de la regla. Para comprender como es que puede representar una compuerta lógica, por ejemplo la compuerta NOT por medio de gliders y gliders-gun es necesario hacer uso de una pequeña abstracción.

Se puede imaginar que un flujo de gliders representa una señal, ya sea de

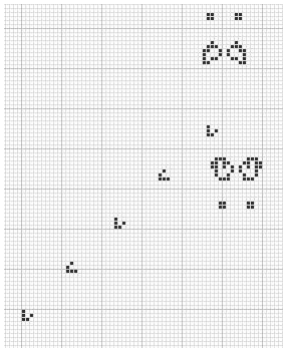

Fig. 3.6: Un glider-gun en Life.

entrada o de salida. Esta señal tiene un valor alto o "1" cuando existe un glider, y bajo o "0" en su ausencia; en la figura 3.7 se puede ver el flujo de gliders, note que la sección del flujo donde no existe glider es el pulso bajo o cero de la señal. De esta manera la señal que representa el flujo de gliders de la figura 3.7 es: 11011011.

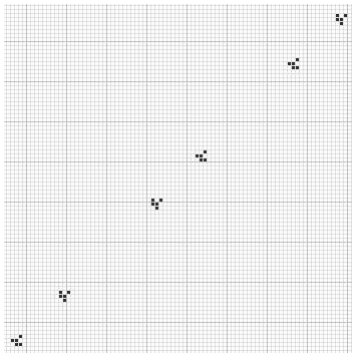

Fig. 3.7: Flujo de gliders.

Utilizando el glider-gun mostrado en la figura 3.6 y el flujo de gliders de la figura 3.7 se pueden manipular para que colisionen ambos flujos, el resultado de tal colisión será un nuevo flujo, una señal de salida. Esta señal de salida será contraria a la señal de entrada. De igual manera se pueden manipular los gliders y gliders-gun para la creación de las otras compuertas lógicas. Las figuras 3.8 y 3.9 representan una compuerta NOT, el antes y después de la señal de entrada, el resultado de la operación es el flujo de gliders que tiene la misma dirección que los gliders que lanza el glider-gun. Las células vivas entre los flujos de gliders representan la ausencia de un glider dentro del flujo y no forman parte de la operación ni influyen en su resultado dado que desaparecen inmediatamente al siguiente tiempo.

Entrada	Glider-gun	Salida
1	1	0
0	1	1
1	1	0
1	1	0
0	1	1
1	1	0

Tab. 3.1: Flujos necesarios para la compuerta NOT.

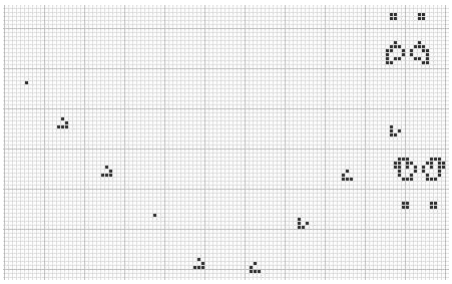

Fig. 3.8: Compuerta NOT; antes de la reacción.

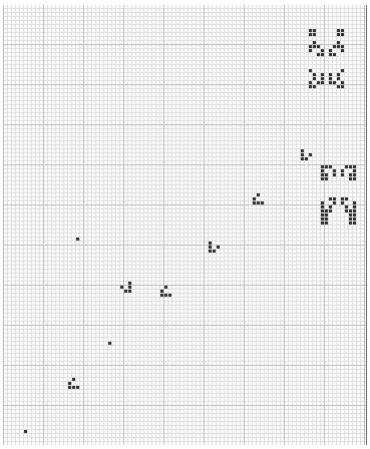

Fig. 3.9: Compuerta NOT; después de la colisión.

3.4 Autómata celular hexagonal

En el año 2005 Andrew Wuensche descubrió un autómata celular hexagonal, que representa un sistema de reacción-difusión basado en una reacción química de tres sustacias: activador, inhibidor y sustrato; el autómata celular fue bien

recibido dado que presentaba gliders, partículas que pueden viajar por el espacio de evoluciones; gracias a estas partículas y a las colisiones entre ellas se logró llegar a la construcción de las compuertas lógicas básicas, mostrando así su universalidad lógica; estos resultados fueron presentados en [3]; debido al comportamiento que presentó se le da el nombre de regla Beehive. Su estudio demostró que a pesar de que presentaba una lógica universal aún tenía detalles a superar, como la ausencia de "glider-guns" estáticos, partículas básicas para la computación en autómatas célulares [2] estos resultados fueron un trabajo presentado en [4] que presenta una nueva regla derivada de Beehive. Esta nueva regla es llamada "spiral rule" (regla espiral), en adelante regla Spiral, y tiene la ventaja de presentar glider-guns estacionarios y móviles, gracias a esto y al antecedente de la regla Beehive se sospecha que la regla Spiral podría soportar una lógica universal a través de compuertas lógicas.

En particular, éste autómata celular hexagonal presenta tres estados derivados de las sustancias: 0, 1 y 2; donde el estado '0' hace referencia al sustrato o base, el activador estado '1', e inhibidor estado '0'; las células reaccionarán mediante la vecindad que esta formada por células hexagonales; se puede ver esta vecindad en la figura 5.1, y donde se toman en cuenta solo a los 6 vecinos inmediatos.

4. SIMULADOR PARA AUTÓMATA CELULAR HEXAGONAL

Para poder realizar el estudio de la regla *Spiral* de los autómatas celulares hexagonales, se vio la necesidad de buscar una herramienta que permitiese visualizar las evoluciones de la regla para poder observar las colisiones y la interacción existente entre cada célula, y por consiguiente, entre las partículas que se forman.

La primer herramienta que se encontró para poder estudiar la regla fue el *Discret Dynamic Laboratory*, conocido como el *DDLab*, el cual se desarrolló por Andrew Wuensche de la facultad de *Computación, Ingeniería y Ciencias Matemáticas* de la Universidad del Este de Inglaterra.

DDLab es un software interactivo para la investigación de redes dinámicas discretas, relevantes para el estudio de fenómenos complejos, redes neuronales y biomoleculares; además es aplicada a la investigación y educación en universidades científicas para la búsqueda de biotecnología. Una red dinámica discreta puede tener conecciones arbitrarias y reglas heterogéneas, así como la presencia de autómatas celulares y redes booleanas aleatorias. La lattice puede ser unidimensional, bidimensional, en forma rectangular o hexagonal, y tridimensional.

La herramienta fue implementada en C, y corre en las plataformas Unix, Irix y Windows, en las cuales se puede manejar un total de 65025 células, además durante la simulación se puede cambiar el estado de las células y la escala.

No obstante, todas estas bondades no fueron suficientes para poder análizar la regla de manera sencilla, debido a que el acceso a algunas de estas opciones no era intuitiva. Además, el número de células que puede procesar es pequeño y limita para la implementación de construcciones complejas.

Para resolver este problema se propuso realizar una herramienta la cual tuviera una interfaz gráfica para que no se necesite de un amplio conocimiento en el manejo de la computadora, y que permitiese manipular un número de células mayor.

4.1 Definición del problema y causas

El DDLab es una herramienta muy eficiente para la visualización de los autómatas celulares; sin embargo, la limitante de tener espacios de evolución pequeños acotaba la realización de construcciones complejas; además, para poder crear una configuración inical y simular la regla es necesario introducir una serie de parámetros lo que hace muy tardado la creación y manipulación de la regla y por consiguiente su estudio.

Para el pleno cumplimiento del objetivo del trabajo, era necesaria una herramienta que permitiese manejar grandes espacios de evolución con la finalidad de realizar construcciones complejas las cuales requieren que la herramienta tenga la capacidad de realizar operaciones en tiempos aceptables de procesamiento y visualización; además de ser indispensable la manipulación de los estados de las células para poder crear construcciones libremente de una manera rápida y sencilla. Para poder conservar los resultados obtenido en el simulador, es necesario poder manejar archivos, es decir, que se pueda abrir y guardar la información.

4.2 Requerimientos de la herramienta

Después de un análisis se definió que la herramienta deberá cumplir con las siguientes características, como mínimo.

Se requiere que la herramienta visualice la evolución de las células en un tiempo t, este cambio se dará gracias a la función de transición especificada en la regla Spiral.

Cada célula del autómata celular tendrá una forma hexagonal, así como cada una podrá tener uno de tres estados dado un tiempo t y serán representados por colores predefinidos. El espacio de evoluciones es donde el autómata celular hexagonal evolucionará en tiempos discretos, dicho espacio tendrá un número determinado de células. Este espacio de evoluciones será presentado en una ventana que deberá tener la capacidad de mostrar lattices de 80×80, 160×160, 240×240, 500×500 y 1000×1000 células.

Para poder crear un nuevo espacio de evoluciones debe ser posible realizarlo de las dos siguientes maneras:

- Configuraciones aleatorias
- Configuración editable

En ambos casos el usuario podrá escoger el tamaño del espacio de evoluciones, así como también los colores que representarán a los tres estados.

Las configuraciones iniciales aleatorias le asignarán uno de los tres estados a cada célula al azar. La configuración editable deberá mostrar las células en

estado cero. Una vez que las muestra el usuario podrá cambiar el estado de cada célula al darle clic con el apuntador del mouse, la transición de los estados será del estado cero al uno, del uno al dos y del dos al cero.

Una vez que ya se ha establecido la configuración inicial se deberá poder controlar las evoluciones; para lo cual deberán existir botones que permitan que evolucione paso a paso, evolucione continuamente y además pueda pausar esa evolución, así como permitir regresar a la configuración inicial previamente establecida. Dado que todo se da en tiempos discretos, se requiere que el simulador muestre en que tiempo se encuentra la evolución, es decir, el número de la evolución actual.

Dado que es necesario almacenar las configuraciones que se realicen y que sean de utilidad para los fines del trabajo se requiere el manejo de archivos, en el cual pueda guardar la configuración inicial para después poder abrirla y mostrarla en el espacio de evoluciones.

El simulador debe tener la capacidad de permitir editar una configuración una vez que ya se ha dado una configuración inicial. Para facilidad de visualización se solicita que se pueda cambiar la velocidad y el tamaño de las células.

4.3 Propuesta de solución

Para poder darle una solución a lo requerido se propuso hacer una herramienta, la cual será llamada *Spiral Simulator*, de la que se describirá en esta sección sus características.

El paradigma que se utilizó para poder desarrollar esta herramienta es la orientada a objetos; ésto nos ayuda para que en un futuro esta herramienta pueda ser ampliada gracias a la robustes del diseño y que es más sencillo ver toda la composición del autómata como un conjunto de objetos que interactúan entre sí.

La herramienta fue desarrollada en el lenguaje de programación C#, de la plataforma .NET de Microsoft, se escogió este lenguaje gracias a que está orientado a objetos, y a eventos, lo cual facilita la implementación de la parte gráfica del simulador; además de la velocidad que tiene para ejecutar operaciones, lo cual hace más eficiente el cálculo de las transiciones de las células en cada tiempo.

La herramienta que se propone tendrá la capacidad de soportar espacios de evoluciones de hasta 500×500 en una máquina con un solo procesador, y de 1000×1000 en una máquina con dos procesadores, en los cuales el proceso será dado en tiempos aceptables. El resultado de la transición de estados se verá reflejado en un *bitmap*, en el cual las células tendran forma hexagonal, y se podrá cambiar el tamaño.

El objetivo principal de crear una nueva herramienta es, poder manipular fácil e intuitivamente la herramienta, donde no es necesario introducir una serie de paramétros complicados para poder obtener una configuración inicial, como lo es en el caso del DDLab; es decir, que el usuario no deberá tener un gran conocimiento sobre el uso de la computadora para poder establecer una configuración inicial aleatoria o para poderla editar; así como también deberá tener una facilidad para manipular los parametros de visualización.

4.4 Análisis y diseño de la solución

En el diagrama de casos de usos de la figura 4.1 muestra que el usuario podrá establecer configuraciones iniciales, para poder manipular la simulaciones; además, se tiene la opción de manipular archivos, esto se puede visualizar en la figura 4.1.

El detalle del caso de uso de establecer configuraciones, se observa en la figura 4.2 que se pueden establecer configuraciones aleatorias o editables, además, ambos tipos de configuraciones pueden ser editados por el usuario.

En la figura 4.3 se muestra el caso de usos del detalle de la visualización, en el cual se observa que el usuario puede modificar los colores de estados de las células, el tamaño de la visualización de la células.

Finalmente se puede observar en la figura 4.4 el detalle del caso de uso de la manipulación de archivos, en la cual se puede abrir y guardar las configuraciones en un archivo, el cual poseerá una extensión *.spr*.

El diseño del sistema para implementar la herramienta se puede observar en la figura 4.5; el sistema se compondrá de dos módulos principales; la primera se compone por dos clases; *Celula*: posee las características de una célula, como son, su estado y la vecindad a la que pertenece; y *Lattice*: en la cual se determina el tamaño del espacio de evoluciones, el estado inicial de la configuración y el estado de las células en un tiempo t; y se encargará de realizar todas las operaciones necesarias para la evolución.

Para poder visualizarlo se utilizan dos clases, una de ellas es la de *hexagono* que es la encargada de calcular los puntos que poseerá un hexágono dentro de un espacio de dibujado, las cuales serán requeridas por la clase *DibujadordeLattice* la cual hará todas las operaciones necesarias para que los cambios según las evoluciones sean visualizadas por el usuario.

Ambos módulos serán unificados en el programa principal del sistema, por lo que no se muestra la interacción entre ellos. Así mismo, fue necesario crear una clase estática encargada de realizar todas las operaciones matemáticas que se requieran llamada *mate*.

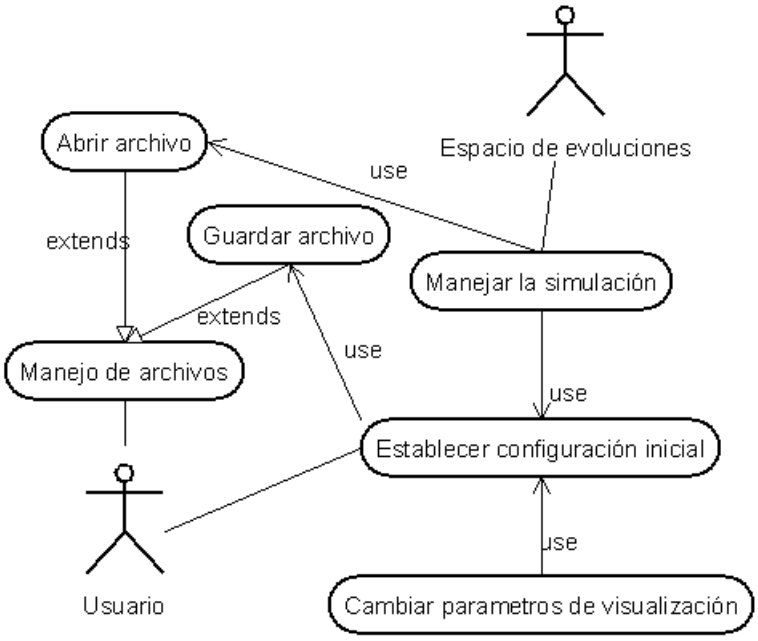

Fig. 4.1: Diagrama de casos de uso.

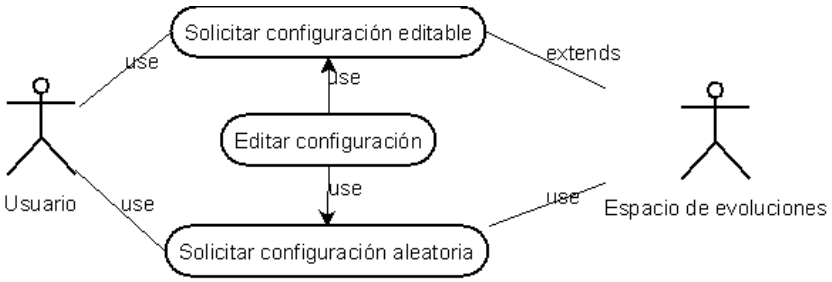

Fig. 4.2: Diagrama de Casos de uso de la configuración inicial

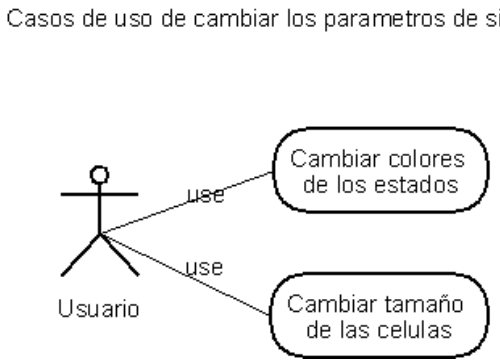

Fig. 4.3: Diagrama de Casos de uso de la visualización

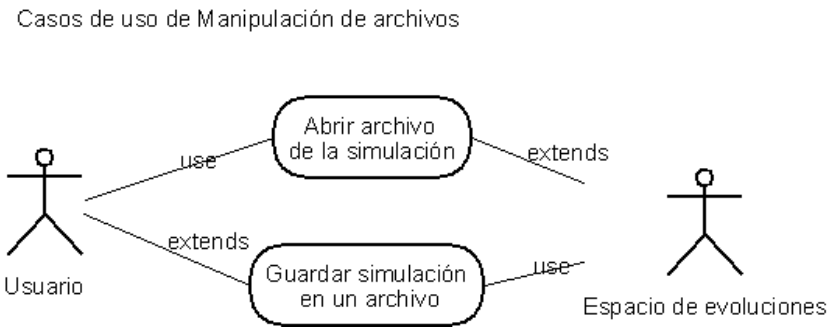

Fig. 4.4: Diagrama de Casos de uso de la manipulación de archivos

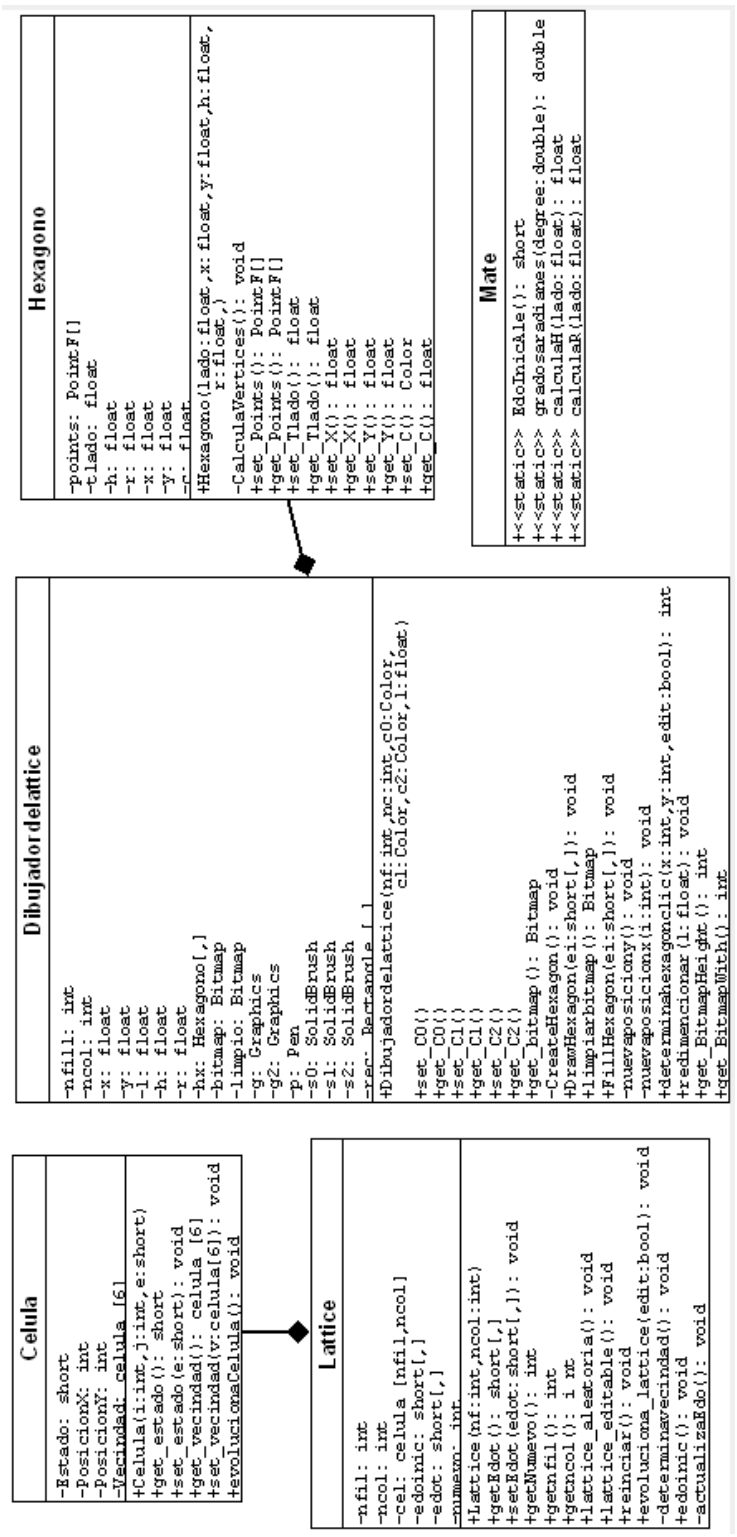

Fig. 4.5: Diagrama de clases.

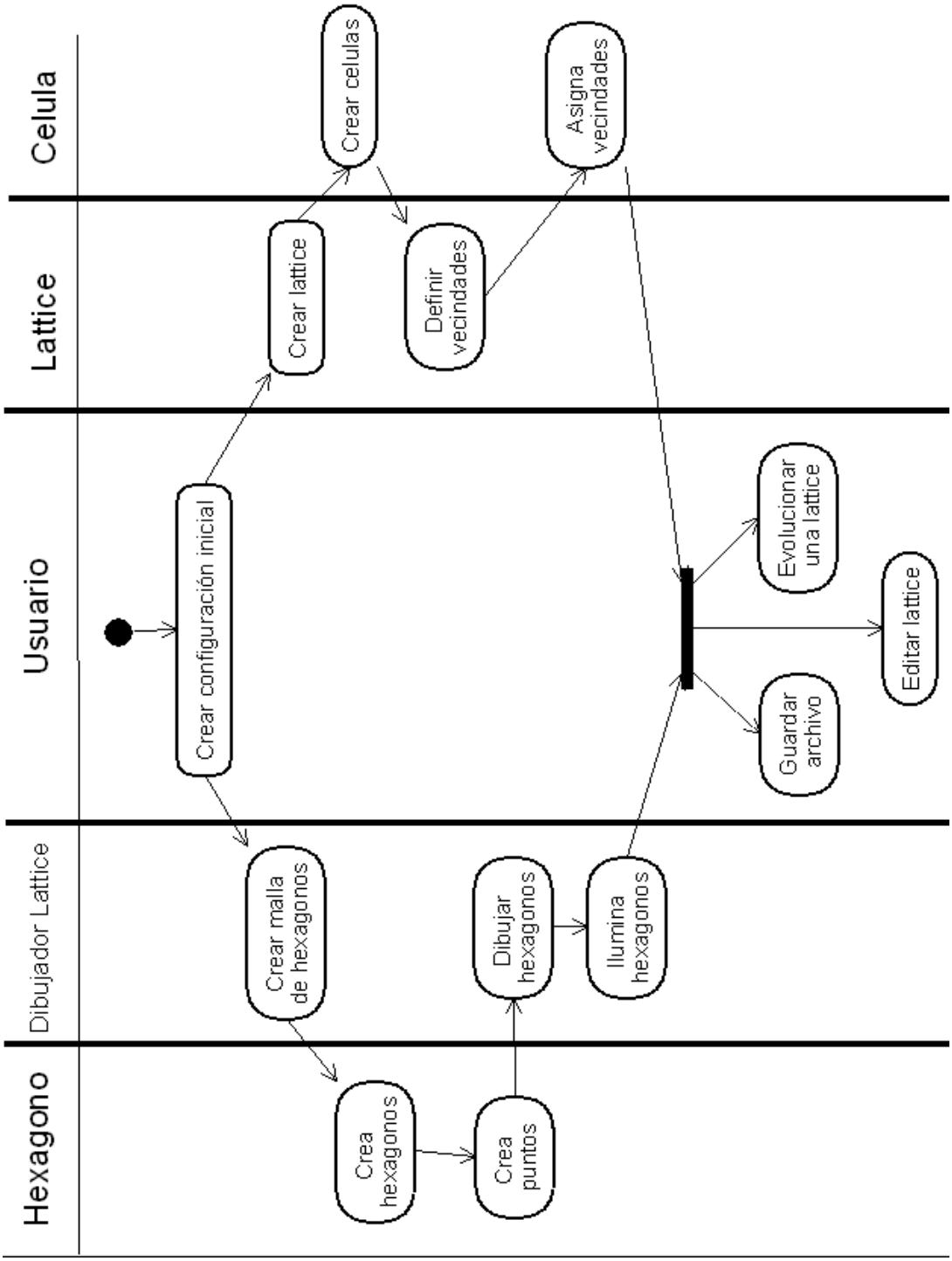

Fig. 4.6: Diagrama de actividades.

5. SPIRAL RULE

La regla Spiral es una regla de AC de 2 dimensiones donde cada célula tiene forma de hexágono; cada célula tiene 6 vecinos inmediatos y puede tener uno de tres estados $\{0, 1, 2\}$; la vecindad es mostrada en la figura 5.1. La regla Spiral es totalística, lo que quiere decir que para que una célula en un tiempo t, evolucione al tiempo $t + 1$, dependerá de su propio estado así como el de sus vecinos.

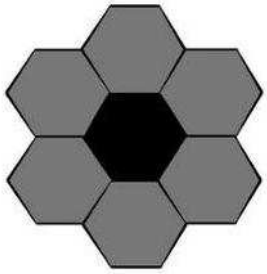

Fig. 5.1: Vecindad Hexagonal.

En las siguientes secciones se explicará el funcionamiento de la regla, las partículas y la dinámica de cada una de ellas.

5.1 Descripción de la regla

Al tener la vecindad 7 células y como cada célula puede tener uno de 3 estados se obtiene un total de 36 posibles vecindades diferentes y para cada una está definido un estado para el tiempo siguiente, de esta manera es posible analizar los cambios de estado de una célula que esté rodeadea de una cantidad de vecinos en cierto estado. Vista de esta manera, la regla puede ser modificada para tener 'mutaciones' de la regla original al cambiar el estado al que pasa dada una de las 36 configuraciones. Es importante resaltar que para la regla sólo se toma en cuenta el número de células dado un estado, gracias a esto, existe una forma alternativa de visualizar la regla, a través de una matriz de transición de estados, en la tabla 5.1 es mostrada dicha matriz de trasición. Las columnas (j) representan la cantidad de células en estado 1 y las filas (i) la cantidad de células en estado 2. La cantidad de células en estado 0 puede ser calculado mediante

		j							
		0	1	2	3	4	5	6	7
	0	0	1	2	1	2	2	2	2
	1	0	2	2	1	2	2	2	
	2	0	0	2	1	2	2		
i	3	0	2	2	1	2			
	4	0	0	2	1				
	5	0	0	2					
	6	0	0						
	7	0							

Tab. 5.1: Matriz de evolución de la regla Spiral

una sencilla operación, $7 - (i + j)$, donde 7 es el número total de células en la vecindad.

La manera de interpretar la matriz de transición es la siguiente: las filas representan la cantidad de células que se tienen en la vecindad a analizar en el estado 2; las columnas representan la cantidad de células en estado 1; de esta manera se observa la vecindad a evolucionar y teniendo la cantidad de células de cada estado en la vecindad el tiempo t se busca la intersección entre las filas y las columnas y el valor que se obtenga será el estado al que pasará la célula central al tiempo $t + 1$.

Lo anterior se ve directamente reflejado al hacer una evolución paso a paso de una configuración inicial sencilla como la mostrada en la figura 5.2, donde se presenta unicamente una célula en estado 1.

Al evaluar las 6 células adyacentes se observa que cada una tiene como vecino a la célula en estado 1, por lo tanto la vecindad de las células adyacentes, como de la ćelula central, es la misma: una ćelula en estado 1 y cero células en estado 2. Entonces, de la tabla 5.1 se puede obtener el estado siguiente de las células adyacentes así como de la célula central, es decir el estado 1. De esta manera se pueden evaluar las células en cada tiempo, obteniendo la evolución mostrada en la figura 5.2.

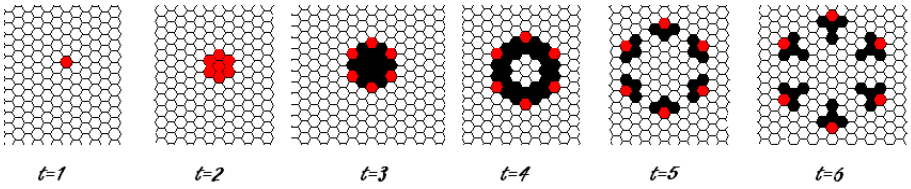

Fig. 5.2: Evolución de una configuración inicial con sólo una célula en estado 1.

El resultado de la interacción entre las células a través del tiempo es la generación de 6 partículas que tienen la característica de viajar a través del espacio de evoluciones llamadas gliders, dichas partículas son estudiadas más a fondo en la sección **??**.

Al colocar como estado inicial células en estado 1 alineadas en el espacio de evoluciones se obtiene un comportamiento realmente interesante, pues su interacción genera diversos gliders, no solo de tipo básico, sino de tipos más complejos como se muestra en las figuras 5.3 y 5.4.

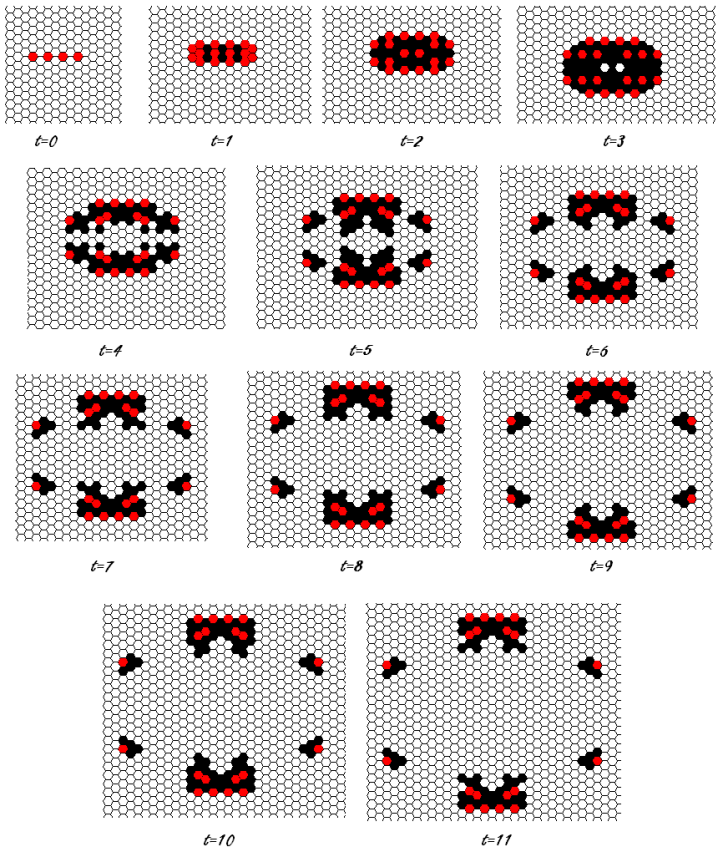

Fig. 5.3: Evolución de una configuración inicial con 4 células alineadas en estado 1.

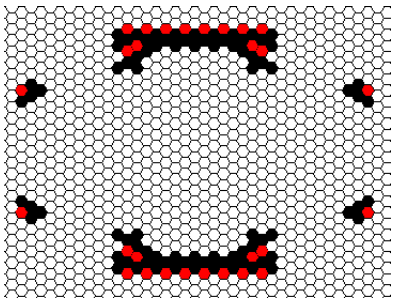

Fig. 5.4: Evolucion número 11 de una configuración inicial de 8 células alineadas en estado 1. Se puede observar la presencia de 2 gliders complejos en la parte superior e inferior.

5.2 Dinámica y partículas básicas

La regla de evolución, representada con la matriz 5.1 permite observar la interacción entre células, lo que a través de las evoluciones hará que se formen partículas de forma inherente. Un ejemplo de la regla Spiral donde se pueden apreciar algunas de las parículas que pueden formarse es mostrada en la figura 5.5, cabe destacar que muestra una configuración inicial aleatoria en la evolución 191 con una lattice de 80×80 células. También se puede observar la característica principal por la cual la regla es de sumo interés, es decir, la presencia de gliders y glider-guns. Estas partículas que en la regla Life eran complejas y difíciles de crear, en la regla Spiral se crean por sí mismas con gran rapidez.

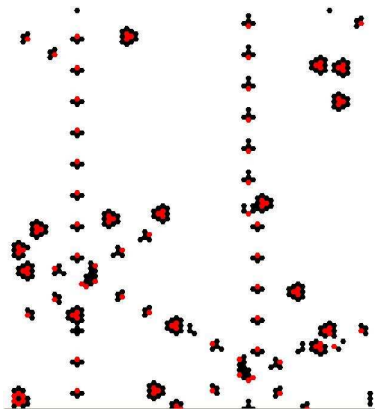

Fig. 5.5: Evolución 191 de una configuración inicial aleatoria en la regla Spiral.

De vital importancia es identificar las diversas partículas que presenta la

regla con la finalidad de analizar aquellas que servirán en la búsqueda de computación. En la figura 5.6 se muestran los tipos básicos de gliders que presenta el autómata, de ellos se podrán obtener diferentes datos relevantes al momento de utilizarlos con fines computacionales:

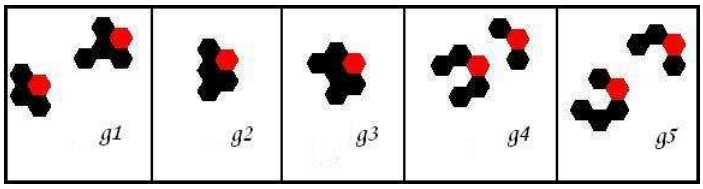

Fig. 5.6: Tipos de gliders.

- *Peso:* se refiere a la cantidad de células que forman una partícula; de tener más de una fase (o forma) se toma la mayor como su peso.
- *período:* es el número de evoluciones necesarias para que una partícula vuelva a tener su estructura inicial.
- *Desplazamiento:* se refiere al número de células que avanza el glider por período, se define su dirección en base al plano cartesiano.
- *Velocidad:* esta determinada como P/D, es decir, Período/Desplazamiento.

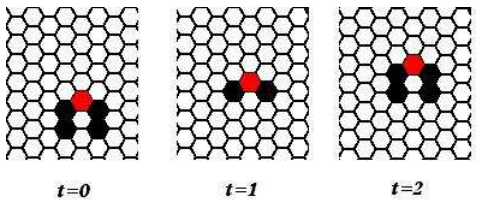

Fig. 5.7: Glider g_4.

Para explicar estos conceptos se puede tomar como ejemplo el glider g_4. Su peso es 5, pues tiene 5 células que lo conforman en su fase más grande; su período es 2, esto puede apreciarse de la figura 5.7, donde se puede ver que le toma 2 tiempos regresar a su forma inicial. En la misma imagen se observa el desplazamiento que tiene el glider, donde se desplaza 1 célula cada tiempo, y, sí para completar un período necesita de dos tiempos, entonces tiene un desplazamiento igual a 2; finalmente, la velocidad del glider g_4 se obtiene dividiendo su período entre su desplazamiento, es decir, $\frac{2}{2} = 1$. De esta manera se han analizado los 5 gliders, la información obtenida se muestra en la tabla 5.2.

Partícula	Peso	Velocidad	período	Desplazamiento
g_1	5	1	2	2
g_2	5	1	1	1
g_3	6	1	1	1
g_4	5	1	2	2
g_5	5	1	2	2

Tab. 5.2: Características de gliders.

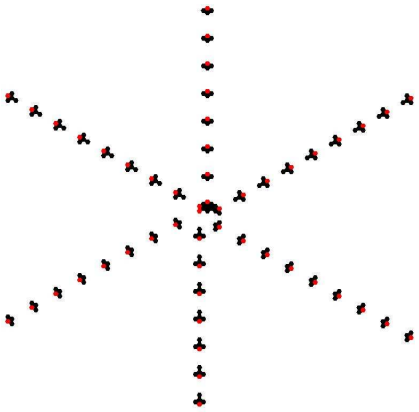

Fig. 5.8: Glider-gun G_1.

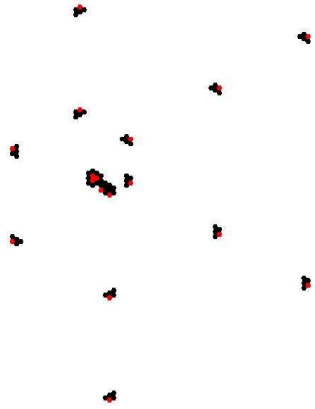

Fig. 5.9: Glider-gun G_2.

La creación de gliders es muy importante, y para ello la regla Spiral tiene dos tipos de gliders-gun básicos, también llamados guns, estos son mostrados en las figuras 5.8 y 5.9. La información que se puede obtener de estos guns está condensada en la tabla 5.3. De la tabla, se define la frecuencia como el número de gliders que produce en un período, y el período se define de la misma manera que para los gliders.

Glider Gun	Glider que produce	período	Frecuencia
G_1	g_1	6	6
G_2	g_2	22	6

Tab. 5.3: Características de glider-guns.

Otra partícula importante es aquella capaz de eliminar los gliders, se le denomina eater. Existen dos tipos básicos de eaters en la regla Spiral, estos se presentan en la figura 5.10. Sin importanr cuantas veces y en que ángulo choque un glider con un eater, este será destruido. Es interesante observar el comportamiento cuando un glider no choca, sino sólo roza el eater pues con base en esto es que se modifican los tipos de gliders, es decir, si tenemos un glider g_1

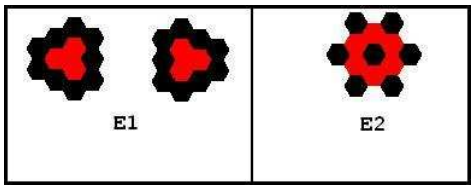

Fig. 5.10: Tipos de eaters.

que pasa por un lado de un eater E_1, se transformara en un glider g_4 como se muestra en la figura 5.11.

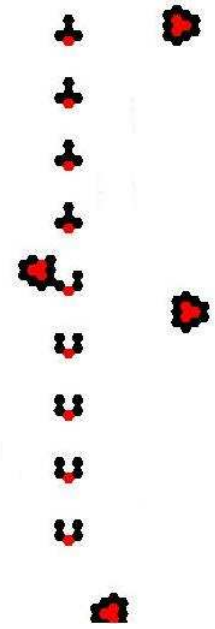

Fig. 5.11: Transformación de gliders por medio de un eater.

5.3 Computación en la regla Spiral

Mediante la manipulación de las partículas básicas que presenta la regla Spiral es posible implementar computación, y se demuestra mediante compuertas lógicas.

La característica de los glider-guns de este autómata que les permite lanzar gliders en 6 direcciones puede llegar a ser una ventaja, pues se podrían procesar 6 señales al mismo tiempo, no obstante, por ahora sólo se considerará un solo flujo; los flujos de gliders que no se utilicen serán eliminados mediante un eater

E_1. Un glider-gun limitado en 5 de sus 6 flujos se puede apreciar en la figura 5.12. Esto mismo se puede extender para el glider-gun G_2.

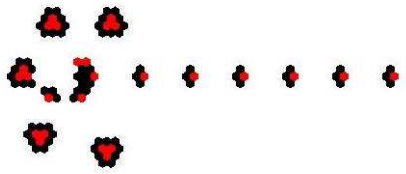

Fig. 5.12: Glider-gun G_1 con 5 flujos eliminados.

De la misma manera que en Life, en la regla Spiral se representan unos lógicos, '1', con la presencia de gliders y ceros lógicos, '0', con la ausencia de los mismos. Así, utilizando el glider-gun G_1 se puede representar una cadena constante de información con 1's. La manera de cambiar esta cadena y poder hacerla más diversa es mediante el glider-gun G_2. Al lanzar los gliders a una frecuencia más baja es posible modificar el flujo de gliders del G_1 para generar cadenas de unos y ceros, un ejemplo de esto se muestra en la figura 5.13.

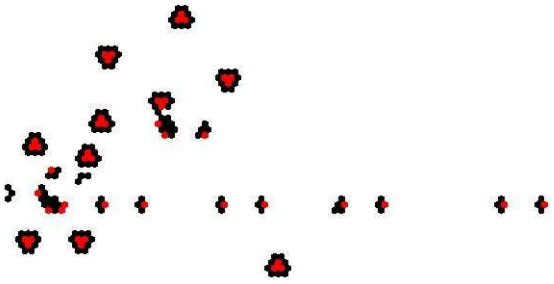

Fig. 5.13: Flujo de gliders modificado.

La sincronización entre glider-guns es una de las báses primordiales para la creación de compuertas lógicas, no solo en la regla spiral, sino en cualquier autómata, pues la computación esta basada en las colisiones entre las partículas y la reacción que estas colisiones producen. Despues de observar la regla Spiral se notó una similitud entre las colisiones existentes entre gliders, dichas colisiones se muestran en la figura 5.14. En la imágen se observan tres colisiones diferentes, la colisión del inciso A tiene como reacción el cambiar de dirección el glider proveniente del suroeste; en el inciso B se observa la aniquilación de ambos gliders; el resultado de la colisión del inciso C es la eliminación de un solo glider, mientras el otro sigue su curso normalmente. Éstas y otras colisiones se utilizan como función para la construcción de las compuertas lógicas.

Otra característica más a considerar son las partículas "excedentes", es decir, gliders que se generan y no son útiles para la computación propuesta, estos gliders son eliminados mediante eaters. Finalmente, para construir una compuerta lógica y que su resultado sea fácilmente verificable es necesario construir una flujo de entrada que contenga los bits necesarios para comprobar la tabla de verdad de la compuerta implementada.

Las compuertas implementadas mediante la regla Spiral son: AND, OR y NOT; es decir, las compuertas lógicas básicas; con estas compuertas la regla Spiral posee una lógica universal. A lo largo de la búsqueda se logró encontrar otras compuertas más, estas son: NOR, XOR y XNOR.

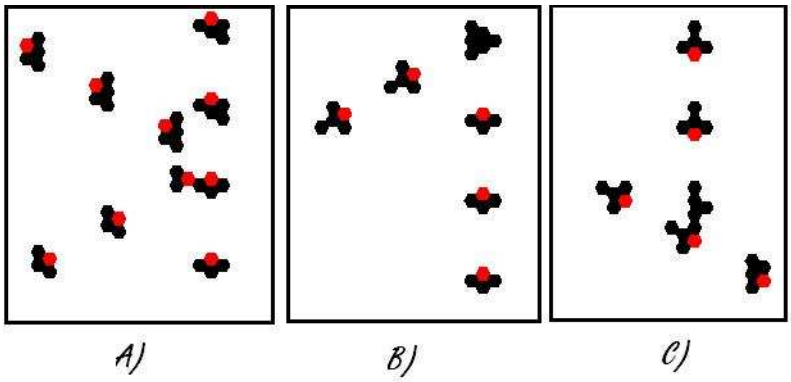

Fig. 5.14: Tipos de colisiones entre gliders.

A	S
0	1
1	0

Tab. 5.4: Tabla de verdad de la compuerta NOT.

Entradas		Salidas				
A	B	AND	OR	NOR	XOR	XNOR
0	0	0	0	1	0	1
0	1	0	1	0	1	0
1	0	0	1	0	1	0
1	1	1	1	0	0	1

Tab. 5.5: Tablas de verdad para las compuertas AND, OR, NOR, XOR y XNOR.

EL orden cronológico de creación es el siguiente: NOT, AND, NOR y XNOR. Debido a que se tenían las compuertas NOR y XNOR se hizo uso de la compuerta NOT para de esta manera crear las compuertas OR y XOR. La representación en el autómata celular de las compuertas lógicas se presenta a continuación:

- La compuerta NOT esta formada por dos glider-gun G_1 y un glider-gun G_2, este último se utiliza para modificar la señal de entrada de la compuerta; se observa la compuerta en la figura 5.15, donde A es el glider-gun que tiene el flujo de entrada y S es el flujo de salida de la compuerta. La señal de entrada es: 1100110; por lo que su flujo de salida es: 0011001.

- En la compuerta AND, que se puede ver en la figura 5.16, se utiliza un glider-gun G_1 por cada señal de entrada, de igual manera, para modificar el flujo de gliders se requiere un glider-gun G_2 por cada flujo de entrada; el flujo A es: 1111010; para la entrada B se utiliza: 1101100; el resultado de aplicar la operación AND se muestra con la salida S y es: 1101000.

- Para construir la compuerta NOR se requieren 3 glider-guns G_1, dos para las entrads A y B, y uno que forme parte del proceso de transformación de los gliders para generar el resultado; también se utilizan 4 G_2 para modificar los flujos de entrada, dos por cada flujo. Las cadenas de bits que representan los flujo de los gliders de entrada son: 1101000 para la entrada A y 1100100 para la entrada B; el resultado es: 0010011. En la figura 5.18 se puede observar la compuerta NOR.

- Finalmente, en la compuerta XNOR, se ocupan 3 glider-gun G_1, dos para las entradas y uno que forma parte del proceso de transformación de los datos. Para modificar las señales de entrada se requirierón de 2 glider-guns G_2, uno para cada entrada; las señales de entrada de la compuerta son: para A 0111101 y 1101100 para B; la salida de la compuerta es: 0101110. En la figura 5.20 se presenta la compuerta lógica implementada.

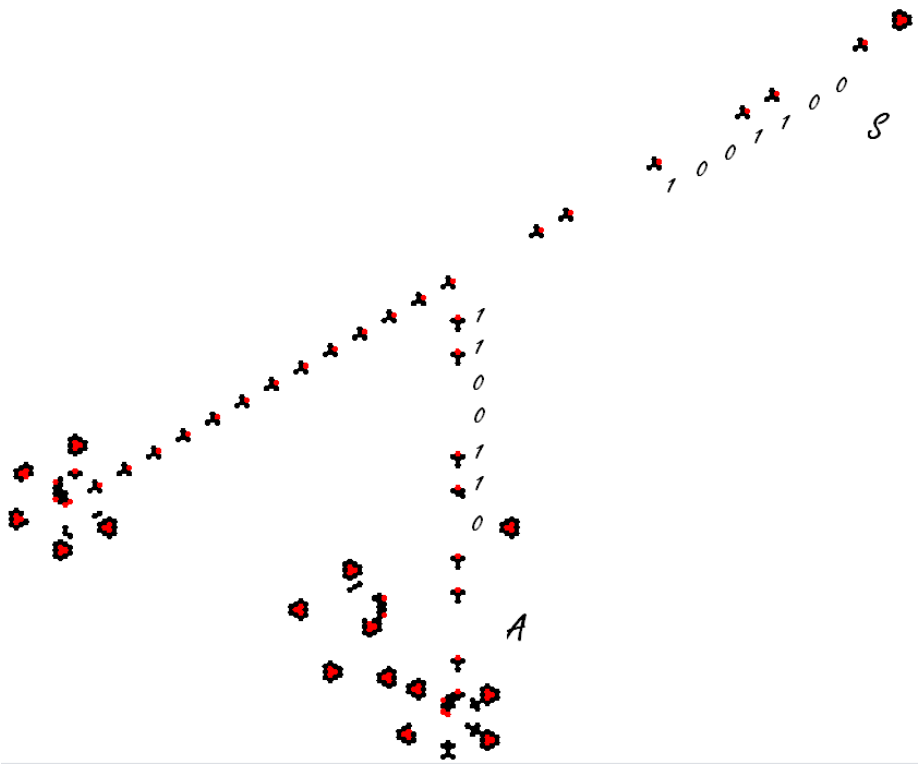

Fig. 5.15: Compuerta NOT en la regla spiral.

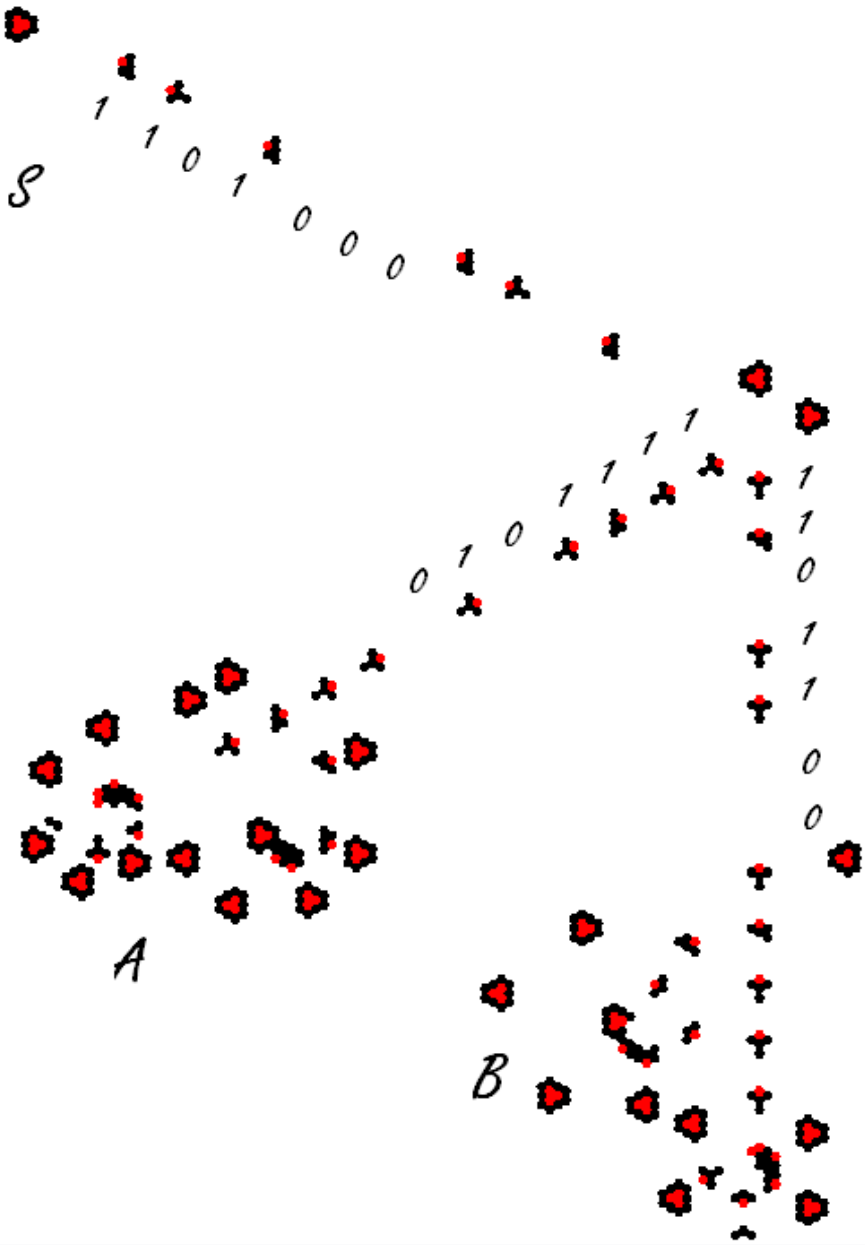

Fig. 5.16: Compuerta AND en la regla spiral.

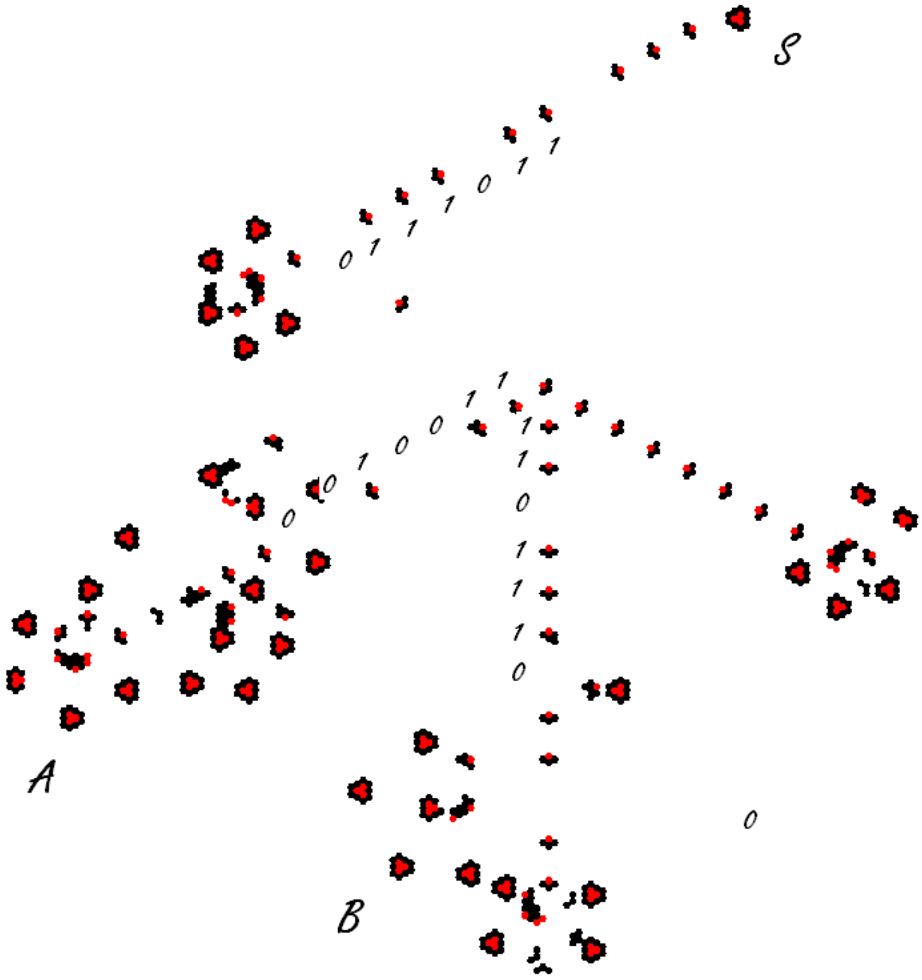

Fig. 5.17: Compuerta OR en la regla spiral.

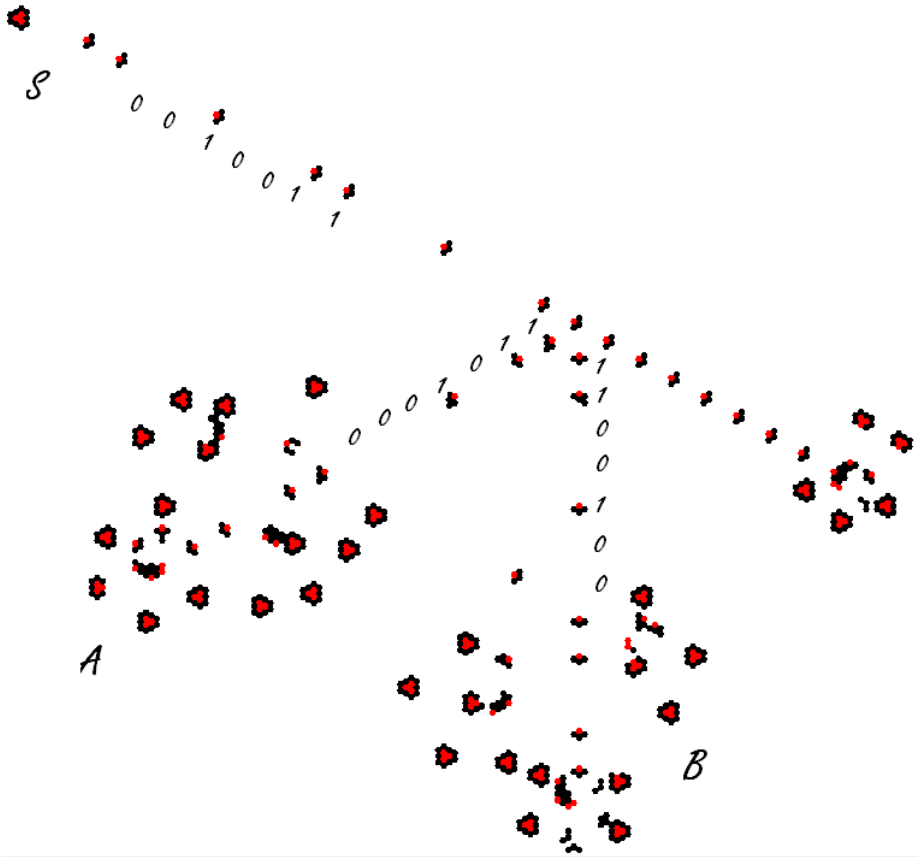

Fig. 5.18: Compuerta NOR en la regla spiral.

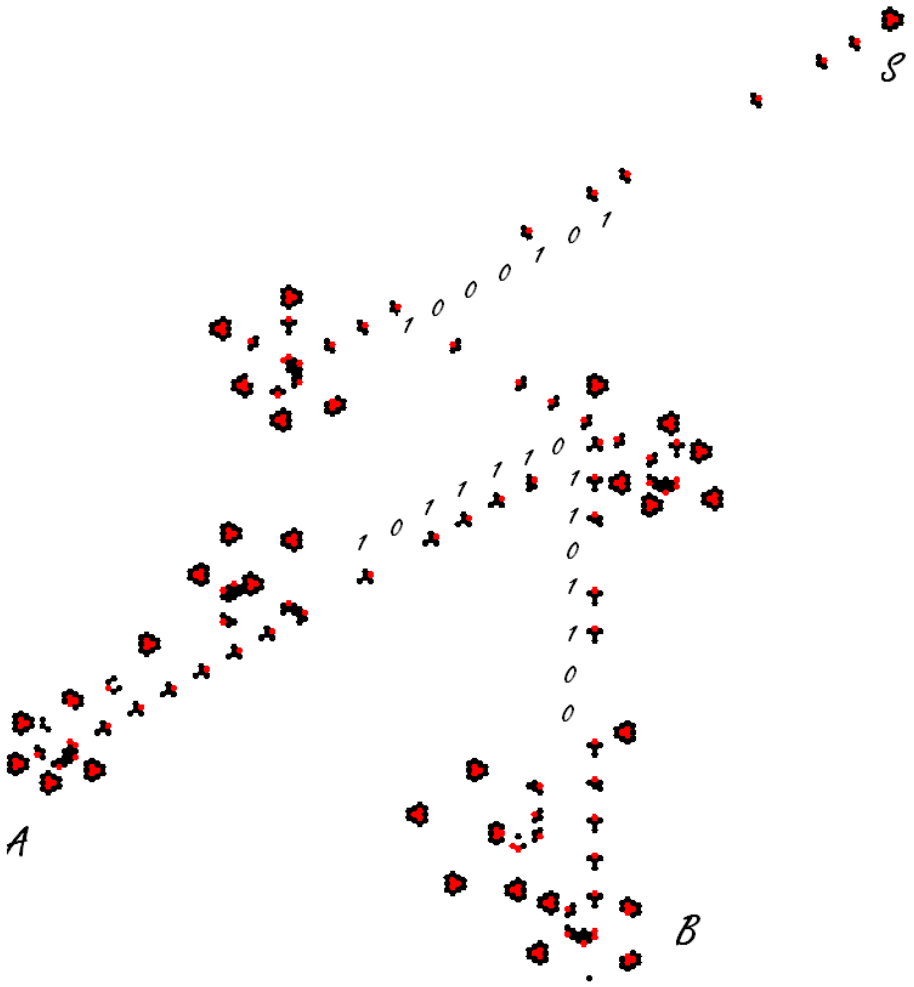

Fig. 5.19: Compuerta XOR en la regla spiral.

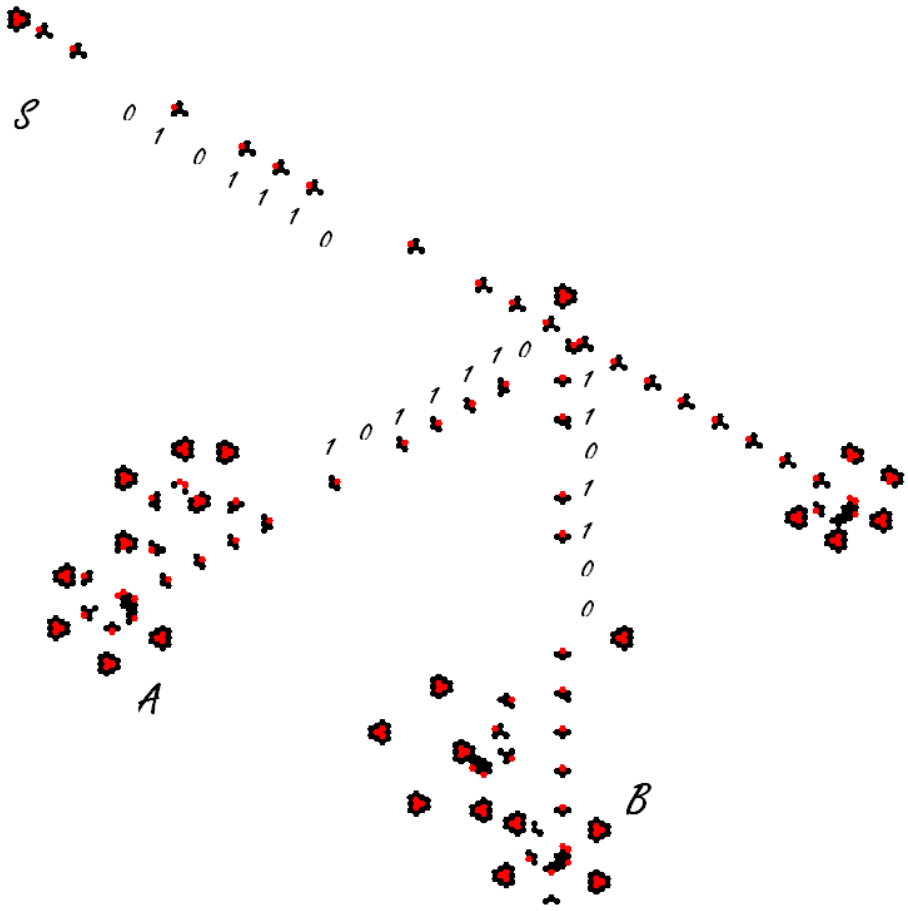

Fig. 5.20: Compuerta XNOR en la regla spiral.

5.4 Otras configuraciones y partículas interesantes

Al realizar el estudio de la regla Spiral se observó que los gliders antes presentados no son los únicos existentes dentro de la regla, sino que existe una diversa fauna de partículas. Es interesante observar que todos los gliders tienen una velocidad de 1, es decir, avanzan una célula en cada evolución. A continuación se presentan algunas de las partículas que se encontraron a lo largo de la búsqueda de computación.

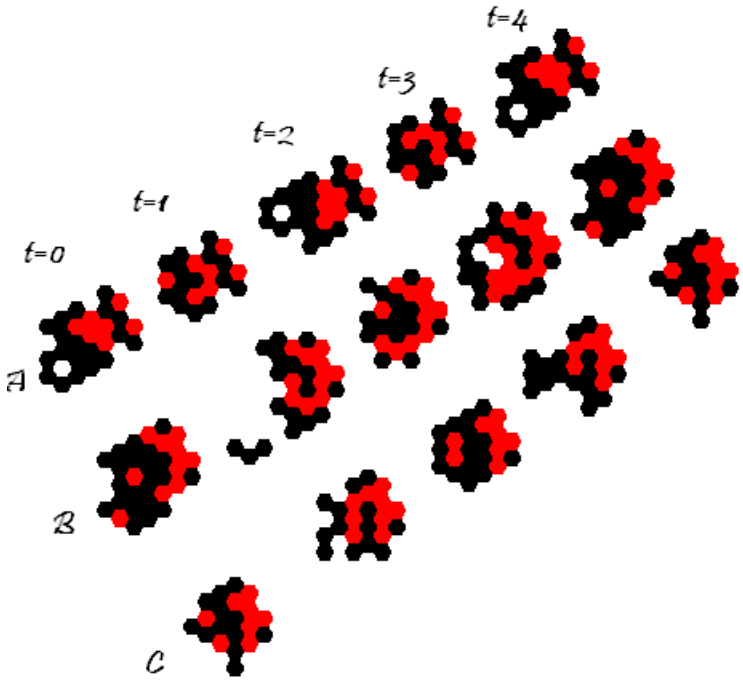

Fig. 5.21: Gliders complejos en la regla Spiral de período 4.

En la figura 5.24 se puede observar una configuración inicial, inciso A, y en el inciso B se presenta dicha configuración después de 27 evoluciones; en el inciso B se puede apreciar la generación de dos glider-gun movibles derivados de la configuración de la imagen del inciso A.

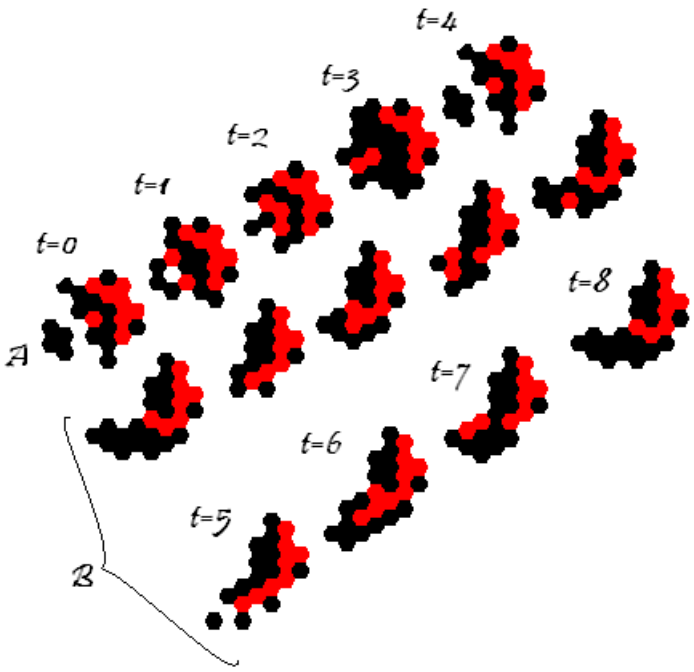

Fig. 5.22: Gliders complejos en la regla Spiral. El glider A posee un período 4; el glider B tiene un período 8.

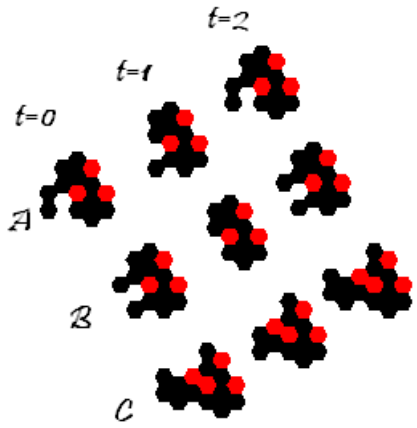

Fig. 5.23: Gliders complejos en la regla Spiral de período 2.

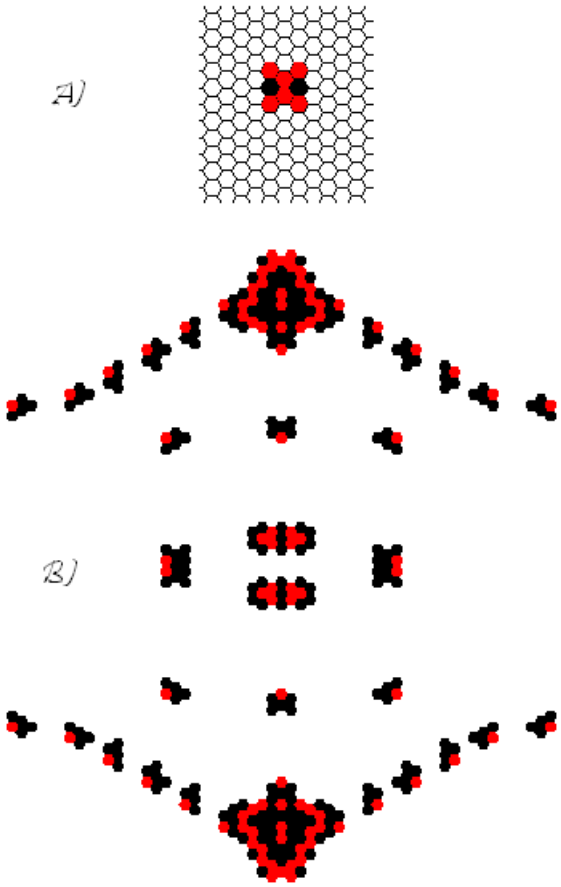

Fig. 5.24: Glider-gun movible. En el inciso *A* se muuestra la configuración inicial que da paso a la generación de los dos glider-gun presentados en el inciso *B*

6. DISCUSIÓN, RESULTADOS Y TRABAJO A FUTURO

Se ha analizado la regla Spiral y las colisiones entre partículas, así como sus reacciones. Por otro lado, se ha programado un simulador a través del cual dicho análisis ha sido más rápido y detallado, y gracias a esto se han podido construir compuertas lógicas en la regla Spiral de autómata celular hexagonal.

Gracias a estas construcciones se ha demostrado que es posible implementar computación dentro de la regla Spiral para autómatas celulares hexagonales; también, se ha demostrado la lógica universal de la regla al tener las tres compuertas lógicas básicas: AND, OR y NOT. Derivado de éstas construcciones se logró implemetar otras compuertas que serán de ayuda en la búsqueda de construcciones más complejas. Por ejemplo, un objetivo siguiente puede ser la construcción de un medio sumador utilizando la compuerta XOR y AND que se tienen actualmente. Además de utilizarlas para construir dispositivos más complejos, simular una función computable completa o algún otro sistema activador, inhibidor y refractario; incluso cualquier otro sistema no-lineal con dicha dinámica

Dentro de la regla Spiral existen partículas que aún no han sido utilizadas con fines computacionales, en este sentido queda abierta la posibilidad de encontrar más partículas, o tomar algunas de las ya encontradas para utilizarlas en nuevas construcciones, o alguna construcción derivada de las ya existentes.

Los espacios de evolución utilizados durante las pruebas y construcciones mostradas en el presente trabajo son de 160×160 y 240×240 células, lo que hace pensar que al realizar contrucciones derivadas de las actuales, será necesario utilizar espacios de evoluciones más amplios, lo que conlleva un mayor procesamiento y una visualización menos agradable; es por eso, que se propone como trabajo a futuro migrar el simulador a un cluster de visualización.

A través de las pruebas realizadas y la experiencia de las construcciones hechas se observó la necesidad de herramientas de edición, como un draw&drop, o una herramienta que permita saber la dirección exacta de los flujos de gliders generados por un glider-gun antes de ser incertado al espacio de evoluciones.

Finalmente, resta pensar en los diferentes tipos de análisis estadísticos que pueden ser mostrados por el simulador; por ejemplo, un análisis de densidad de estados, o de partículas “vivas” dentro del espacio de evoluciones.

BIBLIOGRAFÍA

[1] Andrew Adamatzky (1994), *Identification of Cellular Automta*, Taylor & Francis.

[2] Andrew Adamatzky (Ed.) (2002), *Collision-Based Computing*, Springer.

[3] Andrew Adamatzky, Andrew Wuensche and B. De Lacy Costello (2005), "Glider-based computing in reaction diffusion hexagonal cellular automata", *Journal Chaos, Solitons & Fractals.*

[4] Andrew Adamatzky and Andrew Wuensche (2006), "Computing in Spiral Rule Reaction-Diffusion Hexagonal Cellular Automaton," *Complex Systems* **16 (4)**.

[5] Andrew Adamatzky and Christof Teuscher, *From Utopian to Genuine Unconventional Computers*, Luniver Press, 2006.

[6] Andrew Ilachinski, *Cellular Automata: A Discrete Universe*, World Scientific Press, 2001.

[7] Andrew Wuensche (2004), "Self-reproduction by glider collisions; the beehive rule," *International Journal Pollack et. al*, editor, Alife9 Proceedings, MIT Press.

[8] Andrew Wuensche (2005), "Glider Dynamics in 3-Value Hexagonal Cellular Automata: The Beehive Rule," *International Journal of Unconventional Computing.*

[9] Anisul Haque, Masahiko Yamamoto y Ryoichi Nakatani, Yasushi Endo (2003), Japon, Science and Technology of Advanced Materials.

[10] Bell, E T. (1940), *"Historia de las Matemáticas"*, Mexico, Fondo de Cultura Económica.

[11] Boyer, Carl B., *"A History of Mathematics"*, US, 2° edition, Jhon Wiley & Sons Inc.

[12] Dexter C. Kozen, "Theory of Computation", Springer.

[13] Elwyn R. Berlekamp, John H. Conway y Richard K. Guy, *"Winning Ways for Your Mathematical Plays"* Volume 4, Second Edition, **927-961**, A K Peters.

[14] Enrique Alfonseca Cubero, Manuel Alfonseca Moreno and Roberto Moriyón Salomon (2007), *Teoría de Autómatas y Lenguajes Formales*, Mc Graw Hill.

[15] Genaro J. Martínez, Andrew Adamatzky, Harold V. McIntosh, and Ben De Lacy Costello, "Computation by competing patterns: Life rule B2/S2345678" desde *Automata 2008: Theory and Applications of Cellular Automata* páginas 356–366, Luniver Press, 2008.

[16] Genaro J. Martínez, Adriana M. Méndez, y Miriam M. Zambrano, Un subconjunto de autómata celular con comportamiento complejo en dos dimensiones, Escuela Superior de Cómputo, Instituto Politécnico Nacional, México, 2005,http://uncomp.uwe.ac.uk/genaro/Papers/Papers_on_CA.html.

[17] Genaro J. Martínez, Andrew Adamatzky, and Ben De Lacy Costello,(2008) "On logical gates in precipitating medium: cellular automaton model" Physics Letters A 1 (48), 1-5.

[18] Genaro J. Martínez, Harold V. McIntosh, Juan C. S. T. Mora, and Sergio V. C. Vergara (2007), "Rule 110 objects and other constructions based-collisions", Journal of Cellular Automata 2 (3), 219-242.

[19] Genaro J. Martínez, Andrew Adamatzky, and Harold V. McIntosh (2006), "Phenomenology of glider collisions in cellular automaton Rule 54 and associated logical gates", Chaos, Fractals and Solitons 28, 100-111.

[20] Harold V. McIntosh, *One Dimensional Cellular Automata*, Luniver Press, 2009.

[21] Howard Gutowitz (1991), *Cellular Automata, Theory and Experiment*, The MIT Press.

[22] John E. Hopcroft and J. D. Ullman (1993), *Introducción a la teoría de autómatas, lenguajes y computación*, Addison-Wesley.

[23] Luciano García Garrido, *Cibernética Matemática*, México, IPN.

[24] Manfred Schroeder, *Non-standard computation: molecular computation-cellular automata-evolutionary algorithms-quantum computers.*

[25] Manrique Mata-Montero y Virginia Berrón Lara, *¿Qué es una computadora?*, División de Posgrado de la Universidad Tecnológica de la Mixteca.

[26] Massimo Macucci, *Quantum cellular automata: theory, experimentation and prospects*, Imperial College Press, 2006.

[27] Moshe Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, Springer, 1997.

[28] Richard Johnsonbaugh (1988), *Matemáticas Discretas*, México, Grupo Editorial Iberoamérica.

[29] Selim G. Akl (2006), *Conventional or unconvetional: Is any computer universal?*, from "Utopian to Genuine Unconventional Computers", Luniver Press.

[30] Stephen Wolfram (2002), *A New Kind of Science*, Champaign, Illinois, Wolfram Media Inc.

[31] Tommaso Toffoli, *Non-Conventional Computers*, ECE Department, Boston University.

[32] Tommaso Toffoli and Norman Margolus (1987), *Cellular Automata Machines*, The MIT Press, Cambridge, Massachusetts.

[33] von Neumann, John (1966), *"Theory of self-reproducting automata"*, Urbana and London, University of Illinois.

[34] Whitney J. Townsend y Jacob A. Abraham (2004), *"Complex Gate Implementations for Quantum Dot Cellular Automata"*, University of Texas, Austin.

APÉNDICE

A. NUEVAS FORMAS DE VIDA

Una manera diferente de represenar la regla Life es mediante la siguiente notación: $R(X_{min}, X_{max}, Y_{min}, Y_{max})$. Donde X representa el número de células para la supervivencia, este valor puede oscilar entre un minimo y un máximo; de manera similar para Y, que representa el rango de células necesarias para un nacimiento. Estos rangos estan limitados por el número de células vecinas, es decir, que si usamos una vecindad de Moore nuestro máximo será obligatoriamente 8, y 4 si es que se usa la vecindad de Neumann. De esta manera podemos ahora representar la regla Life así: R(2, 3, 3, 3).

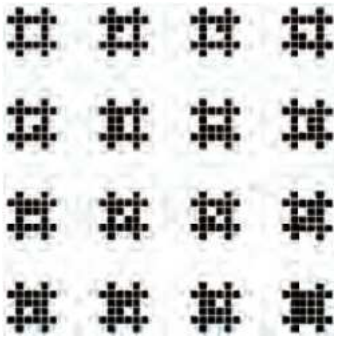

Fig. A.1: Partícula indestructible en la regla Life B2/S2345678.

En la regla Life B2/S2345678[1] presentada en [15], se observan partículas de un interés especial para la computación via AC. Esta partícula, mostrada en la figura A.1, tiene la característica de ser indestructible, por lo cual es posible la creación de "tuberías" por las que puede enviarse un flujo de células vivas con un patrón especifico, esta otra característica, el generar flujos con patrones reconocibles, es gracias al comportamiento que tiene el estado "vida", un comportamiento de absorción.

La manera de realizar la computación es mediante las compuertas lógicas, construidas a través de una tubería en forma de T; en los extremos superiores entran los flujos de células y al colisionar se deja el paso sólo a un tipo de flujo, un ejemplo de esto es mostrado en la figura A.2. De esta manera es posible la

[1] Las imagenes mostradas sobre la regla fuerón tomadas de [15]

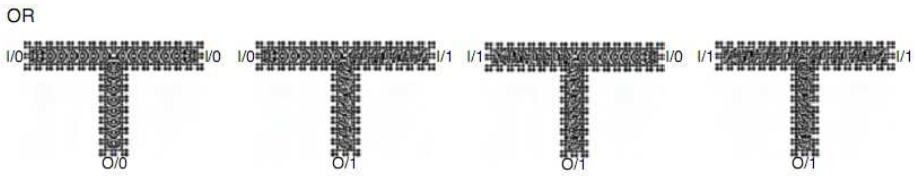

Fig. A.2: Compuerta lógica OR usando la regla Life B2/S2345678.

creación de sistemas más complejos, por ejemplo un medio-sumador, como el mostrado en la figura A.3.

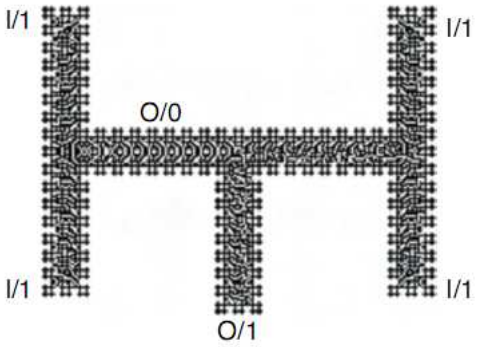

Fig. A.3: Medio-sumador, con la operación 1+1.

B. MANUAL TÉCNICO

Inicialmente se describirán las clases necesarias para el desarrollo del sistema.

Clase célula. En ella se describen las características que poseen las células, que son posición en x, y dentro de la lattice, el estado en que se encuentra, la vecindad a la pertenece; dentro de las funciones que realiza ésta clase son:

- Asignar las vecindades, y
- Realizar la evolución de una célula.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace simuv10
7 {
8     [Serializable()] //
9     class celula
10    {
11        short estado;//stado de la celula, para spiral 0 ,1, 2
12        celula[] vecindad; //vecinos de la celula
13        int i, j;
14        //  Este es el constructor
15        //estado debe ser el estado inicial de la celula
16        public celula(int i, int j,short estado)
17        {
18            this.i = i;
19            this.j = j;
20            this.estado = estado;
21            vecindad = new celula[7];
22        }
23
24
25        public short Estado
26        {
27            set
28            {
29                estado = value;
30            }
31
32            get
33            {
34                return estado;
35            }
36        }
37
38        public celula[] Vecindad
39        {
40            set
41            {
```

```
42                  vecindad = value;
43              }
44              get
45              {
46                  return vecindad;
47              }
48          }
49
50
51          public short evolucionacelula()
52          {
53              int vedo1 = 0;
54              int vedo2 = 0;
55              short edoaux;
56              for (int k = 0; k < 7; k++)
57              {
58                  if (vecindad[k].Estado == 1)
59                      vedo1++;
60                  else if (vecindad[k].Estado == 2)
61                      vedo2++;
62              }
63
64              switch (vedo1)
65              {
66                  case 0:
67                      edoaux = 0;
68                      break;
69                  case 1:
70                      if (vedo2 == 0)
71                          edoaux = 1;
72                      else if (vedo2 == 1 || vedo2 == 3)
73                          edoaux = 2;
74                      else
75                          edoaux = 0;
76                      break;
77                  case 2:
78                      edoaux = 2;
79                     break;
80                  case 3:
81                     edoaux = 1;
82                      break;
83                  default:
84                      edoaux = 2;
85                      break;
86              }
87              return edoaux;
88          }
89      }
90  }
```

Clase lattice. La clase lattice se utiliza para poder crear nuestros espacios de evoluciones, es importante que todos los parámetros que se le mandan son por referencia, dado que son las condiciones que proporciona el usuario, como por ejemplo, el tamaño de la lattice y los estados iniciales.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.IO;
6 using System.Runtime.Serialization.Formatters.Binary ;
7
8
9 namespace simuv10
10 {
11     [Serializable()] // Para guardarlo como objeto
12     class Lattice
```

```
13    {
14        int nfil;//filas de la lattice
15        int ncol; //filas de la columna
16        celula[,] cel;//celulas
17        short[,] edoinic;
18        short[,] edot; // estado celula en el tiempo t
19        int numevo; // numero de evoluciones
20
21        public Lattice(int nfil, int ncol)
22        {
23            this.nfil = nfil;
24            this.ncol = ncol;
25            cel = new celula [nfil,ncol];
26            edoinic = new short[nfil,ncol];
27            edot = new short[nfil,ncol];
28            numevo = 0;
29        }
30
31        public short[,] Edot
32        {
33            get { return edot; }
34            set { edot = value; }
35        }
36
37        public short[,] Edotini
38        {
39            get { return edoinic; }
40            set { edoinic = value; }
41        }
42
43        public int Numevo
44        {
45            get { return numevo; }
46        }
47
48        public int Fil
49        {
50            set { nfil = value; }
51            get { return nfil; }
52        }
53
54        public int Col
55        {
56            set { ncol = value; }
57            get { return ncol; }
58        }
59
60        //Estable los estadps iniciales de una configuraci\'on aleatoria
61        public void lattice_aleatoria()
62        {
63            for(int i =0; i< nfil ; i++)
64                for (int j = 0; j < ncol; j++)
65                {
66                    cel[i, j] = new celula(i,j, mate.edoInicAlea());
67                    edot[i, j] = cel[i, j].Estado;
68                }
69            for (int i = 0; i < nfil; i++)
70                for (int j = 0; j < ncol; j++)
71                    determinarvecindad(i, j);
72
73            numevo = 0;
74        }
75
76        //Crea la lattice con celulas en estado cero
77        public void lattice_editable()
78        {
79            for (int i =0; i< nfil;i++)
80                for (int j = 0; j < ncol; j++)
```

```
81                  {
82                      cel[i, j] = new celula(i,j,0);
83                      edot[i, j] = cel[i, j].Estado;
84                  }
85              for (int i = 0; i < nfil; i++)
86                  for (int j = 0; j < ncol; j++)
87                      determinarvecindad(i, j);
88
89              numevo = 0;
90          }
91
92
93          public void reiniciar()
94          {
95              edoinicial(false);
96              actualiza_estado();
97              numevo = 0;
98          }
99
100         //Realiza la evolucion de toda la lattice
101         public void evoluciona_lattice(bool edit)
102         {
103             if(numevo ==0)
104                 edoinicial(edit);
105             actualiza_estado();
106             for (int i = 0; i < nfil; i++)
107                 for (int j = 0; j < ncol; j++)
108                     edot[i, j] = cel[i, j].evolucionacelula();
109             numevo++;
110         }
111
112         //Determina la vecindad de todas las celulas de la lattice
113         private void determinarvecindad(int ci, int cj)
114         {
115             celula[] vecindad = new celula[7];
116             vecindad[0] = cel[ci, cj];
117             if ((cj % 2) == 0)
118             {
119                 if (ci == 0)
120                 {
121                     vecindad[1] = cel[(nfil - 1), cj];
122                     vecindad[2] = cel[(nfil - 1), cj + 1];
123                     vecindad[3] = cel[ci, (cj + 1)];
124                     vecindad[4] = cel[(ci + 1), cj];
125                     if (cj == 0)
126                     {
127                         vecindad[5] = cel[ci, ncol - 1];
128                         vecindad[6] = cel[(nfil - 1), (ncol - 1)];
129                     }
130                     else
131                     {
132                         vecindad[5] = cel[ci, cj - 1];
133                         vecindad[6] = cel[(nfil - 1), (cj - 1)];
134                     }
135                 }
136                 else
137                 {
138                     vecindad[1] = cel[ci - 1, cj];
139                     vecindad[2] = cel[(ci - 1), (cj + 1)%ncol];
140                     vecindad[3] = cel[ci, (cj + 1)%ncol];
141                     vecindad[4] = cel[(ci + 1)%nfil, cj];
142                     if (cj == 0)
143                     {
144                         vecindad[5] = cel[ci, ncol - 1];
145                         vecindad[6] = cel[ci - 1, ncol - 1];
146                     }
147                     else
148                     {
```

```
149                         vecindad[5] = cel[ci, cj - 1];
150                         vecindad[6] = cel[ci - 1, cj - 1];
151                     }
152                 }
153             }
154             else
155             {
156                 if (ci == 0)
157                 {
158                     vecindad[1] = cel[(nfil - 1)%nfil, cj];
159                     vecindad[2] = cel[ci, (cj + 1) % ncol];
160                     vecindad[3] = cel[ci + 1, (cj + 1) % ncol];
161                     vecindad[4] = cel[ci + 1, cj];
162                     vecindad[5] = cel[ci + 1, cj - 1];
163                     vecindad[6] = cel[ci, cj - 1];
164                 }
165                 else
166                 {
167                     if (ci == 0)
168                     {
169                         vecindad[1] = cel[(nfil - 1), cj];
170                         vecindad[2] = cel[ci, (cj + 1)%ncol];
171                         vecindad[3] = cel[(ci + 1), (cj + 1)%ncol];
172                         vecindad[4] = cel[ci + 1, cj];
173                         vecindad[5] = cel[ci + 1, (cj - 1)];
174                         vecindad[6] = cel[ci, cj - 1];
175                     }
176                     else
177                     {
178                         vecindad[1] = cel[(ci - 1), cj];
179                         vecindad[2] = cel[ci, (cj + 1)%ncol];
180                         vecindad[6] = cel[ci, cj - 1];
181                         vecindad[3] = cel[(ci + 1)%nfil,(cj+1)%ncol];
182                         vecindad[4] = cel[(ci + 1)%nfil, cj];
183                         vecindad[5] = cel[(ci + 1)%nfil, (cj-1)];
184                     }
185                 }
186             }
187             cel[ci, cj].Vecindad = vecindad;
188         }
189 
190         public void edoinicial(bool edit)
191         {
192 
193                 for (int i = 0; i < nfil; i++)
194                     for (int j = 0; j < ncol; j++)
195                         edoinic[i, j] = edot[i, j];
196         }
197 
198         private void actualiza_estado()
199         {
200             for (int i = 0; i < nfil; i++)
201                 for (int j = 0; j < ncol; j++)
202                     cel[i, j].Estado = edot[i,j];
203         }
204 
205         public void reestablecerInicial(short [,]reic)
206         {
207             for (int i = 0; i < nfil; i++)
208                 for (int j = 0; j < ncol; j++)
209                     edoinic[i, j] = reic[i, j];
210         }
211     }
212 }
```

Clase Hexagono. Se ocupan clases aparte para el dibujado de la lattice dentro de un bitmap; la clase *hexagono* es la equivalente a la clase *célula*, pero para la

visualización de la misma; además que en ella se realizan todos los cálculos de los puntos que tiene cada vértice de cada hexágono.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Drawing;
6
7 namespace simuv10
8 {
9     class Hexagono
10    {
11        private System.Drawing.PointF[] points;
12        private float tlado;
13        private float h;
14        private float r;
15        private float x;//posiscion en x
16        private float y;//posicion en y
17        private Color c;//color de estado
18
19
20        public Hexagono(float lado, float x, float y, float h, float r)
21        {
22            tlado = lado;
23            this.h = h;
24            this.r = r;
25            this.x = x;
26            this.y = y;
27            CalculaVertices();
28            c = Color.Beige;
29        }
30
31        private void CalculaVertices()
32        {
33            points = new PointF[6];
34            points[0] = new PointF(x, y);
35            points[1] = new PointF(x + tlado, y);
36            points[2] = new PointF(x + tlado + h, y + r);
37            points[3] = new PointF(x + tlado, y + r + r);
38            points[4] = new PointF(x, y + r + r);
39            points[5] = new PointF(x - h, y + r);
40        }
41
42        public PointF[] Points
43        {
44            get
45            {
46                return points;
47            }
48            set
49            { }
50        }
51
52        public float Tlado
53        {
54            get
55            {
56                return tlado;
57            }
58            set
59            { tlado = value; }
60        }
61        public float X
62        {
63            get
64            {
65                return x;
```

```
66              }
67              set { }
68          }
69
70          public float Y
71          {
72              get
73              {
74                  return y;
75              }
76              set
77              { }
78          }
79
80          public Color C
81          {
82              get
83              {
84                  return c;
85              }
86              set
87              {
88                  c = value;
89              }
90
91          }
92
93      }
94  }
```

Clase DibujadorLattice. En esta clase se encuentra todo lo que se necesita para poder visualizar las transiciones que van ocurriendo en cada evolución.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Drawing;
6
7 namespace simuv10
8 {
9     class Dibujadordelattice
10    {
11        int nfil;
12        int ncol;
13        float x, y;//posicion de pixel en x,y
14        float l;
15        float h, r;
16        Hexagono[,] hx;
17        Bitmap bitmap;
18        Bitmap limpio;
19        Graphics g;
20        Graphics g2;
21        Pen p;
22        SolidBrush s0, s1, s2;
23        Rectangle [,]rec;
24
25        //Este es el constructor
26        public Dibujadordelattice(int nf,int nc,Color c0,Color c1,Color c2,float l)
27        {
28            this.nfil = nf;
29            this.ncol = nc;
30            x = 0;
31            y = 0;
32            h = mate.calculaH(l);
33            r = mate.calcularR(l);
34            hx = new Hexagono[nfil, ncol];
35            rec = new Rectangle[nfil, ncol];
```

```
36              bitmap=new Bitmap(Convert.ToInt16(nfil*(7+mate.calculaH(7))+7+mate.calculaH(7)),
37                  Convert.ToInt16((ncol*((2 * mate.calcularR(7))) + mate.calcularR(7))));
38              limpio = new Bitmap(Convert.ToInt16(nfil * (l + h) + l + h),
39                  Convert.ToInt16((ncol * ((2 * r)) + r)));
40
41              g  = Graphics.FromImage(bitmap);
42              g2 = Graphics.FromImage(limpio);
43              p = new Pen(Color.Black, 1);
44              s0 = new SolidBrush(c0);
45              s1 = new SolidBrush(c1);
46              s2 = new SolidBrush(c2);
47               this.l = l;
48              CreateHexagon();
49          }
50
51
52          public Color C0
53          {
54              set { s0.Color = value; }
55              get { return  s0.Color; }
56          }
57
58          public Color C1
59          {
60              set { s1.Color = value; }
61              get { return s1.Color; }
62          }
63
64          public Color C2
65          {
66              set { s2.Color = value; }
67              get { return s2.Color; }
68          }
69
70          public Bitmap Bmp
71          { get { return bitmap; } }
72
73
74          private void CreateHexagon()
75          {
76              x = h;
77              y = r;
78              for (int j = 0; j < nfil ; j++)
79              {
80                  for (int i = 0; i < ncol; i++)
81                  {
82                      nuevaposicionx(i);
83                      hx[j,i] = new Hexagono( l, x, y, h, r);
84                      rec[j,i] = new Rectangle(Convert.ToInt16(x),Convert.ToInt16(y),
85                      Convert.ToInt16(l), Convert.ToInt16(2 * r));
86
87
88                  }
89                  nuevaposiciony();
90              }
91
92          }
93
94          //Dibuja la lattice y muestra el borde de los hexagonos
95          public void DrawHexagon(short[,] ei)
96          {
97              g.FillRectangle(s0, 0, 0, Convert.ToInt16((nfil * (l +h))+h+h),
98                  Convert.ToInt16(((ncol * ((2 * r)))+r)));
99              for (int i = 0; i < hx.GetLength(0); i++)
100             {
101                 for (int j = 0; j < hx.GetLength(1); j++)
102                 {
103                     g.DrawPolygon(p, hx[i, j].Points);
```

```
104                 // rec[i, j] = new Rectangle(Convert.ToInt16(x), Convert.ToInt16(y),
105                 Convert.ToInt16(l),Convert.ToInt16(2 * r));
106                  switch (ei[i, j])
107                  {
108                      case 1:
109                          g.FillPolygon((Brush)s1, hx[i, j].Points);
110                          hx[i, j].C = s1.Color;
111                          break;
112                      case 2:
113                          g.FillPolygon((Brush)s2, hx[i, j].Points);
114                          hx[i, j].C = s2.Color;
115                          break;
116                  }
117              }
118          }
119           g.DrawImage(bitmap, new Point(0,0));
120
121      }
122
123      public Bitmap  limpiarbitmap()
124      {
125          g2.FillRectangle(s0, 0, 0, Convert.ToInt16(nfil * (l + h) + h),
126              Convert.ToInt16((ncol * ((2 * r) + r))));
127          g2.DrawImage(limpio , new Point(0, 0));
128          return limpio;
129      }
130
131      //Muestra los hexagonos pero solo con los colores del relleno,
132      public void FillHexagon(short[,] ei)
133      {
134          g.FillRectangle(s0, 0, 0, Convert.ToInt16(nfil * (l + h)+l +h+h),
135              Convert.ToInt16((ncol * ((2 * r)+r+r))));
136          for (int j = 0; j < hx.GetLength(0); j++)
137              for (int i = 0; i < hx.GetLength(1); i++)
138              {
139                  switch (ei[j,i])
140                  {
141                      case 1:
142                          g.FillPolygon((Brush)s1, hx[j,i].Points);
143                          hx[j, i].C = s1.Color;
144                          break;
145                      case 2:
146                          g.FillPolygon((Brush)s2, hx[j,i].Points);
147                          hx[j, i].C = s2.Color;
148                          break;
149
150                  }
151              }
152           g.DrawImage(bitmap, new Point(0, 0));
153      }
154
155
156      private void nuevaposiciony()
157      {
158          x = h;
159          y = y + (2 * r);
160
161      }
162
163      private void nuevaposicionx(int i)
164      {
165          if (i % 2 == 0)
166          {
167              x = x + l + h;
168              y = y - r;
169          }
170          else
171          {
```

```
172                 x = x + l + h;
173                 y = y + r;
174             }
175         }
176 
177         //Esta parte sirve para determinar a que hexagono de la lattice
178         // se le dio click y por lo tanto se le puede modificar su estado
179         public int determinahexagonclic(int x, int y, bool edit)
180         {
181             int a = 0;
182             bool encontrada = false;
183             if (edit)
184             {
185 
186                 for (int i = 0; i < rec.GetLength(0); i++)
187                 {
188                     for (int j = 0; j < rec.GetLength(1); j++)
189                     {
190                         if (rec[i, j].Contains(x, y) == true)
191                         {
192                             encontrada = true;
193                             break;
194                         }
195                         else
196                             a++;
197                     }
198                     if (encontrada == true)
199                         break;
200                 }
201             }
202             else
203                 a= (nfil * ncol) +1;
204             return a;
205         }
206 
207         //Realiza el redimencionado de los hexagonos
208         public void redimencionar(float l)
209         {
210             if (hx != null)
211             {
212                 hx.Initialize();
213                 this.l = l;
214                 h = mate.calculaH(l);
215                 r = mate.calcularR(l);
216                 CreateHexagon();
217             }
218         }
219 
220         public int BitmapHeight
221         {
222             get { return Convert.ToInt16((ncol * ((2 * r) +r))); }
223         }
224 
225         public int BitmapWith
226         {
227             get { return Convert.ToInt16(nfil * (l + h) + l+ h+h ); }
228 
229         }
230     }
231 }
```

También se tiene una clase estática, la clase *mate*, en la cual se realizan todas las operaciones matemáticas para el desarrollo del sistema.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
```

```
5
6 namespace simuv10
7 {
8     class mate
9     {
10         private static Random r = new Random();
11         //Esta se realiza para la asignaci\'on de las configuraciones aleatorias
12         public static short edoInicAlea()
13         {
14
15             short ea = (short)(r.Next(90) \% 3);
16
17             return ea;
18         }
19
20         //Se utilizan para los c\'alculos de los vertices de los hexagonos
21         public static double gradosaradianes(double degrees)
22         {
23             return degrees * System.Math.PI / 180;
24         }
25
26         public static float calculaH(float tlado)
27         {
28             return (float)(System.Math.Sin(gradosaradianes(30)) * tlado);
29         }
30
31         public static float calcularR(float tlado)
32         {
33             return (float)(System.Math.Cos(gradosaradianes(30)) * tlado);
34         }
35     }
36 }
```

Una vez que teniendo lo anterior se procede a crear todos los botones y toda la interfaz que permita ver los resultados esperados.

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9 using System.IO;
10 using System.Runtime.Serialization.Formatters.Binary;
11
12 namespace simuv10
13 {
14     public partial class Form1 : Form
15     {
16         Lattice l;
17         Dibujadordelattice dl;
18         Graphics g;
19         short [,]aux;
20         short[,] aux2;
21         bool edit;
22
23         public Form1()
24         {
25
26             g = CreateGraphics();
27             InitializeComponent();
28         }
29
30         private void Form1_Load(object sender, EventArgs e)
31         {
32             this.panel1.Height = this.Height - 100;
```

```
33              this.panel1.Width = this.Width-10;
34          }
35
36          //Es en esta seccion donde el usuario da todos los parametros
37          //para la creacion de una nueva configuracion inicial.
38
39          private void newLatticeToolStripMenuItem_Click(object sender, EventArgs e)
40          {
41              New_configuration nc = new New_configuration();
42
43              if (nc.ShowDialog() == DialogResult.OK)
44              {
45                  if (l != null)
46                      l = null;
47                  if (dl != null)
48                      dl = null;
49                  edit = true;
50                  l = new Lattice(nc.Tamano_lattice(), nc.Tamano_lattice());
51                  aux = new short[l.Fil,l.Col];
52                  dl = new Dibujadordelattice(nc.Tamano_lattice(),
53                                  nc.Tamano_lattice(), nc.C1, nc.C2, nc.C3, 5);
54                  if (nc.LAleatoria())
55                  {
56                      l.lattice_aleatoria();
57                      dl = new Dibujadordelattice(nc.Tamano_lattice(),
58                                  nc.Tamano_lattice(), nc.C1, nc.C2, nc.C3, 3);
59                      dl.FillHexagon(l.Edot);
60                  }
61                  else
62                  {
63                      l.lattice_editable();
64                      dl = new Dibujadordelattice(nc.Tamano_lattice(),
65                           nc.Tamano_lattice(), nc.C1, nc.C2, nc.C3, 6);
66                      dl.DrawHexagon(l.Edot);
67                  }
68                  toolStripButton1.Enabled = true;
69                  toolStripButton2.Enabled = true;
70                  toolStripButton3.Enabled = true;
71                  toolStripButton4.Enabled = true;
72                  drawpicturebox();
73                  toolStripButton5.Enabled = true;
74              }
75          }
76
77          //Es lo que realiza el boton que permite
78          //ver la evolucion indefinidamente
79          private void toolStripButton2_Click(object sender, EventArgs e)
80          {
81              toolStrip3.Visible = false;
82              fileToolStripMenuItem.Enabled = false;
83              editToolStripMenuItem.Enabled = false;
84
85              toolStripButton4.Enabled = true;
86              timer1.Enabled = true;
87
88              edit = false;
89               timer1_Tick(sender, e);
90          }
91
92          //El evento Timer nos permitio controlar la velocidad
93          //con la que se generaba la evolucion indefinida
94           private void timer1_Tick(object sender, EventArgs e)
95          {
96              l.evoluciona_lattice(edit);
97              edit = false;
98              dl.FillHexagon(l.Edot);
99              drawpicturebox();
100             toolStripLabel2.Text = l.Numevo.ToString();
```

```
101
102         }
103
104         //Este es el codigo del boton identificar
105         private void toolStripButton4_Click(object sender, EventArgs e)
106         {
107             timer1.Stop();
108             timer1.Enabled = false;
109             fileToolStripMenuItem.Enabled = true ;
110             editToolStripMenuItem.Enabled = true ;
111             optionsToolStripMenuItem.Enabled = true ;
112
113             toolStripButton4.Enabled = false;
114             toolStripButton5.Enabled = true;
115         }
116
117         //Esto es lo que se utilizo para cambiar los colores de las celulas
118         private void statesColorsToolStripMenuItem_Click(object sender, EventArgs e)
119         {
120
121             SetColors sc = new SetColors(dl.C0, dl.C1, dl.C2);
122             if (sc.ShowDialog() == DialogResult.OK)
123             {
124                 dl.C0 = sc.C0;
125                 dl.C1 = sc.C1;
126                 dl.C2 = sc.C2;
127                 dl.FillHexagon(l.Edot);
128                 drawpicturebox();
129
130             }
131         }
132
133         //Para regresar a la configuracion inicial
134         private void toolStripButton1_Click(object sender, EventArgs e)
135         {
136             l.reiniciar();
137             dl.FillHexagon(l.Edotini);
138             pictureBox1.Image = dl.Bmp;
139             toolStripLabel2.Text = l.Numevo.ToString();
140         }
141
142         //Es el que nos permitia saber a que celula se esta cmabiando
143         private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
144         {
145             if (l != null)
146             {
147                 aux = l.Edot;
148                 if (dl.determinahexagonclic(e.X, e.Y, edit) < (l.Fil * l.Col))
149                 {
150
151                     aux[(dl.determinahexagonclic(e.X, e.Y, edit) / l.Fil),
152                       dl.determinahexagonclic(e.X, e.Y, edit)%l.Fil]=
153                       (short)((aux[(dl.determinahexagonclic(e.X,e.Y,edit)/l.Fil),
154                       (dl.determinahexagonclic(e.X, e.Y, edit)%l.Fil)]+1)%3);
155                     l.Edot = aux;
156                     dl.DrawHexagon(l.Edot);
157                     drawpicturebox();
158                 }
159             }
160         }
161 //Permite la evolucion paso a paso
162         private void toolStripButton3_Click(object sender, EventArgs e)
163         {
164             l.evoluciona_lattice(edit);
165             dl.FillHexagon(l.Edot);
166             drawpicturebox();
167             toolStripLabel2.Text = l.Numevo.ToString();
168         }
```

```
169
170  //ESto se necsito para el guardado de los archivos
171         private void saveLatticeToolStripMenuItem_Click(object sender, EventArgs e)
172         {
173             String ruta;
174             saveFileDialog1.Filter = "ficherospiral (*.spr)|*.spr";
175
176             if (saveFileDialog1.ShowDialog() == DialogResult.OK)
177             {
178                 ruta = saveFileDialog1.FileName;
179                 Stream fd = File.Create(ruta);
180                 BinaryFormatter seriador = new BinaryFormatter();
181                 seriador.Serialize(fd, l);
182                 fd.Close();
183             }
184          }
185
186         private void openLatticeToolStripMenuItem_Click(object sender, EventArgs e)
187         {
188             String ruta;
189
190             if (l != null)
191                 l = null;
192             if (dl != null)
193                 dl = null;
194             openFileDialog1.Filter = "ficherospiral (*.spr)|*.spr";
195             if(openFileDialog1.ShowDialog()== DialogResult.OK)
196             {
197                 ruta = openFileDialog1.FileName;
198                  if (File.Exists(ruta))
199                  {
200                      Stream fs = File.OpenRead(ruta);
201                      BinaryFormatter deseriador = new BinaryFormatter();
202                      l = (Lattice)deseriador.Deserialize(fs);
203
204                      dl = new Dibujadordelattice(l.Fil,l.Col,Color.White,
205                                                  Color.Red,Color.Black,3);
206                      dl.FillHexagon(l.Edot);
207                      drawpicturebox();
208                      fs.Close();
209                      toolStripButton1.Enabled = toolStripButton2.Enabled =
210
211
212
  true;
213                  }
214                  else
215                  {
216                       MessageBox.Show("This file doesn't exist");
217                  }
218             }
219         }
220
221
222         private void exitToolStripMenuItem_Click(object sender, EventArgs e)
223         {
224             this.Close();
225         }
226
227         private void fastToolStripMenuItem_Click(object sender, EventArgs e)
228         {
229             timer1.Interval = 100;
230         }
231
232         private void mediumToolStripMenuItem_Click(object sender, EventArgs e)
233         {
234             timer1.Interval = 400;
235         }
```

```
236
237         //CAmbia la velocidad de evolucion entre mas grande
238         //sea el numero, mas lento evoluciona
239         private void slowToolStripMenuItem_Click(object sender, EventArgs e)
240         {
241             timer1.Interval = 800;
242         }
243
244
245         private void toolStripButton5_Click(object sender, EventArgs e)
246         {
247             aux2 = l.Edot;
248             edit = true;
249             toolStripButton6.Enabled = true;
250             dl.DrawHexagon(l.Edot);
251             drawpicturebox();
252
253         }
254
255
256         private void toolStripButton6_Click(object sender, EventArgs e)
257         {
258             l.reestablecerInicial(aux);
259             toolStripButton6.Enabled = false;
260         }
261
262
263         //Limpia el picturebox y lo ajusta al tamano del picturebox al del bitmap
264         private void drawpicturebox()
265         {
266             pictureBox1.Image = dl.limpiarbitmap();
267             pictureBox1.Height = dl.BitmapHeight;
268             pictureBox1.Width = dl.BitmapWith;
269             pictureBox1.Image = dl.Bmp;
270         }
271
272         //Redimenciona el hexagono
273         private void smallToolStripMenuItem_Click(object sender, EventArgs e)
274         {
275             dl.redimencionar(2);
276             if (edit)
277                 dl.DrawHexagon(l.Edot);
278             else
279                 dl.FillHexagon(l.Edot);
280             drawpicturebox();
281         }
282
283         //Redimenciona el hexagono
284         private void mediumToolStripMenuItem1_Click(object sender, EventArgs e)
285         {
286             dl.redimencionar(4);
287             if (edit)
288                 dl.DrawHexagon(l.Edot);
289             else
290                 dl.FillHexagon(l.Edot);
291             drawpicturebox();
292         }
293         //Redimenciona el hexagono
294         private void bigToolStripMenuItem_Click(object sender, EventArgs e)
295         {
296             dl.redimencionar(6);
297             if (edit)
298                 dl.DrawHexagon(l.Edot);
299             else
300                 dl.FillHexagon(l.Edot);
301             drawpicturebox();
302         }
303
```

```
304
305         private void velocityToolStripMenuItem_Click(object sender, EventArgs e)
306         {
307             aux2 = l.Edot;
308             edit = true;
309             toolStripButton6.Enabled = true;
310             //toolStrip3.Visible = true;
311
312             dl.DrawHexagon(l.Edot);
313             drawpicturebox();
314
315         }
316
317         private void editToolStripMenuItem_Click(object sender, EventArgs e)
318         {
319             if(edit)
320             editToolStripMenuItem.Checked = true;
321         }
322     }
323 }
```

Además de la clase *Form* para el desarrollo, existieron dos forms más que fueron manipuladas como cuadros de diálogo; los cuales se implementaron de la siguiente forma:

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace simuv10
11 {
12     public partial class New_configuration : Form
13     {
14         public New_configuration()
15         {
16             InitializeComponent();
17         }
18
19         private void button1_Click(object sender, EventArgs e)
20         {
21             if (colorDialog1.ShowDialog() == DialogResult.OK)
22             {
23                 button1.BackColor = colorDialog1.Color;
24             }
25         }
26
27         private void button2_Click(object sender, EventArgs e)
28         {
29             if (colorDialog1.ShowDialog() == DialogResult.OK)
30                 button2.BackColor = colorDialog1.Color;
31         }
32
33         private void button3_Click(object sender, EventArgs e)
34         {
35             if (colorDialog1.ShowDialog() == DialogResult.OK)
36                 button3.BackColor = colorDialog1.Color;
37         }
38
39         public Color C1
40         {
41             get { return button1.BackColor; }
42         }
```

```
43
44          public Color C2
45          {
46              get { return button2.BackColor; }
47          }
48
49          public Color C3
50          {
51              get { return button3.BackColor; }
52          }
53
54          public int Tamano_lattice()
55          {
56              if (radioButton3.Checked)
57                  return 80;
58              else if (radioButton5.Checked)
59                  return 160;
60              else if (radioButton6.Checked)
61                  return 240;
62              else if (radioButton7.Checked)
63                  return 500;
64              else
65                  return 1000;
66          }
67
68          public bool LAleatoria()
69          {
70              if (radioButton1.Checked)
71                  return true;
72              else
73                  return false;
74          }
75
76      }
77  }
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace simuv10
11 {
12      public partial class SetColors : Form
13      {
14          public SetColors( Color c1, Color c2, Color c3 )
15          {
16
17              InitializeComponent();
18              button1.BackColor = c1;
19              button2.BackColor = c2;
20              button3.BackColor = c3;
21          }
22
23
24          private void button1_Click(object sender, EventArgs e)
25          {
26              if (colorDialog1.ShowDialog() == DialogResult.OK)
27              {
28                  button1.BackColor = colorDialog1.Color;
29              }
30          }
31
32          private void button2_Click(object sender, EventArgs e)
```

```
33          {
34              if (colorDialog1.ShowDialog() == DialogResult.OK)
35              {
36                  button2.BackColor = colorDialog1.Color;
37              }
38          }
39
40          private void button3_Click(object sender, EventArgs e)
41          {
42              if(colorDialog1.ShowDialog() == DialogResult.OK )
43                  button3.BackColor = colorDialog1.Color;
44          }
45
46          public Color C0
47          {
48              get { return button1.BackColor; }
49              set { }
50          }
51
52          public Color C1
53          {
54              get { return button2.BackColor; }
55              set { }
56          }
57
58          public Color C2
59          {
60              get { return button3.BackColor; }
61              set { }
62          }
63      }
64  }
```

C. MANUAL DE USUARIO

C.1 Introducción

Spiral Simulator es un software diseñado para la simulación de una regla de autómata celular hexagonal llamada "Spiral". A través del simulador es posible la creación de configuraciones iniciales, tanto aleatorias como creadas por el usuario; así mismo, es posible la edición de los estados de las células. El presente manual tiene como objetivo que el usuario inexperto obtenga los conocimientos suficientes para manejar de manera correcta y eficaz dicho simulador.

C.1.1 Requerimientos del sistema

Para el buen funcionamiento del sistema es necesario tener las siguientes características de software:

- Una computadora con sistema operativo Windows XP o posterior.
- Tener instalado el framework de la plataforma .NET versión 3.5 de Microsoft.

Las características de hardware mínimas necesarias son las siguientes:

- Procesador *Pentium* 4 a 1.7 GHz o similar.
- 512 MB en memoria RAM.
- 10 MB de disco duro.

C.2 Generalidades

La primer pantalla que se presenta al ejecutar el simulador, se puede ver en la figura C.1; en ella se puede observar que existe una barra de menús, la barra de manipulación de la simulación y el espacio de evoluciones en el cual se verán reflejadas las evoluciones.

C.3 Descripción de menús

El simulador cuenta con tres menús para su uso. La descripción de cada uno de ellos se dará en esta sección.

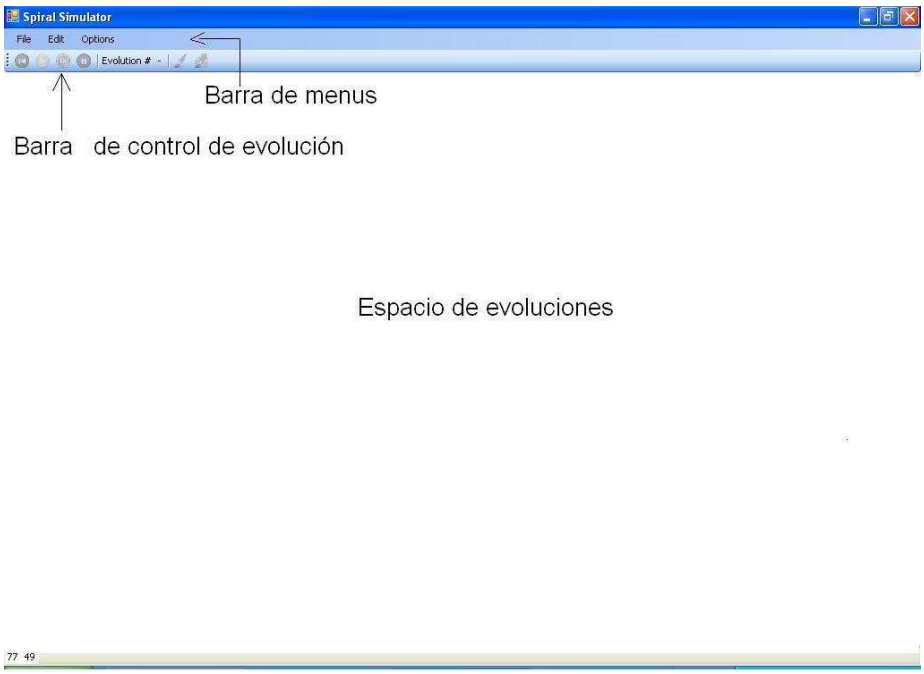

Fig. C.1: Pantalla principal del simulador.

C.3.1 File

Este meú contine las opciones de:

- Crear configuraciones iniciales.
- Guardar un archivo.
- Abrir un archivo.
- Salir.

Para poder crear una nueva configuración se tiene que acceder al menú *File*, y seleccionar la opción *New configuration*. Después aparecerá un cuadro de diálogo, mismo que se muestra en la figura C.3. Se le llamará nueva configuración a la creación de un espacio de evoluciones. Estos pueden ser aleatorios o estar listos para ser editados por el usuario.

La pantalla de la nueva configuración da la posibilidad de seleccionar el tipo de nueva configuración que se desea obtener, el tamaño de la lattice y los colores que representarán cada estado. Para las primeras dos opciones basta con seleccionar la opción correspondiente, sin embargo, para cambiar los colores de los estados se deberá dar clic al botón que se encuentra a lado del nombre del estado al que le pertenece ese color y seleccionar uno de la paleta de colores que

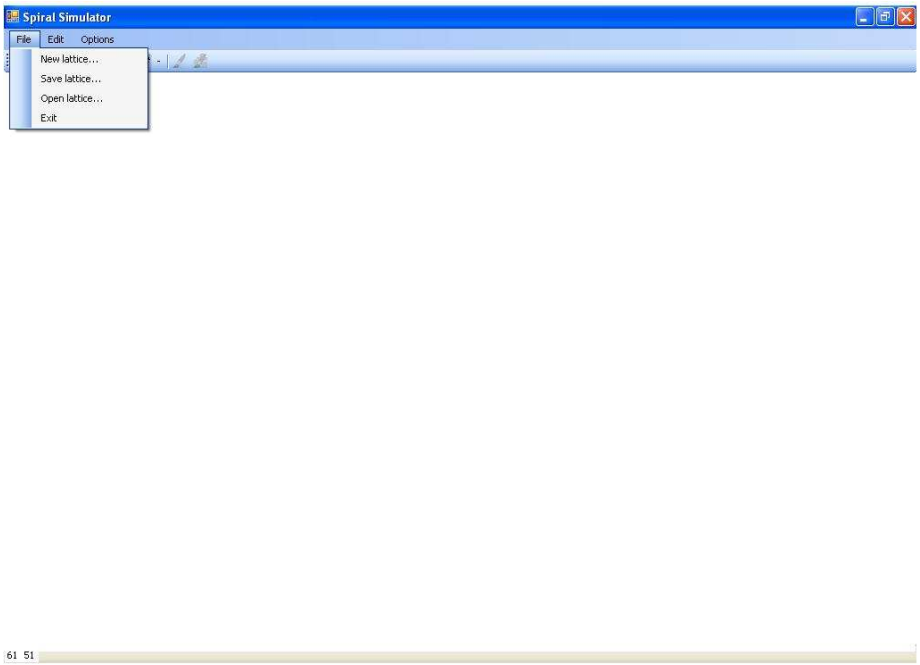

Fig. C.2: Menú *File*.

aparecerá, esta pantalla puede ser observada en la figura C.4.

Editar estados

Si la configuración escogida fue aleatoria, se mostrará en el espacio de evoluciones la configuración. Si se escogio que fuese editable aparecerán los hexagonos y para poder modificar su estado bastará con darle clic al hexágono deseado. Los cambios de estados son: del cero al uno, del uno al dos, y del dos al cero; esta opción se puede apreciar en la figura C.5.

Guardar archivo

El simulador puede guardar las configuraciones, cabe mencionar que lo que guarda es el estado inicial de la evolución, y todas sus características; para poder guardar se debe ir al menú *File*, y seleccionar *Save lattice...*. Aparecerá la pantalla de la figura C.6, se le coloca el nombre y se selecciona el lugar donde se desea guardar; se guardará un archivo con extensión .spr.

Abrir archivo

Para poder visualizar configuraciones anteriormente creadas y previamente guardadas, en el menú *File* se selecciona la opción *Open lattice...*, y se busca el archivo deseado, cabe mencionar que debe tener la extensión .spr. Mostrará la configuración de la última evolución antes de guardar. La figura C.7 muestra la

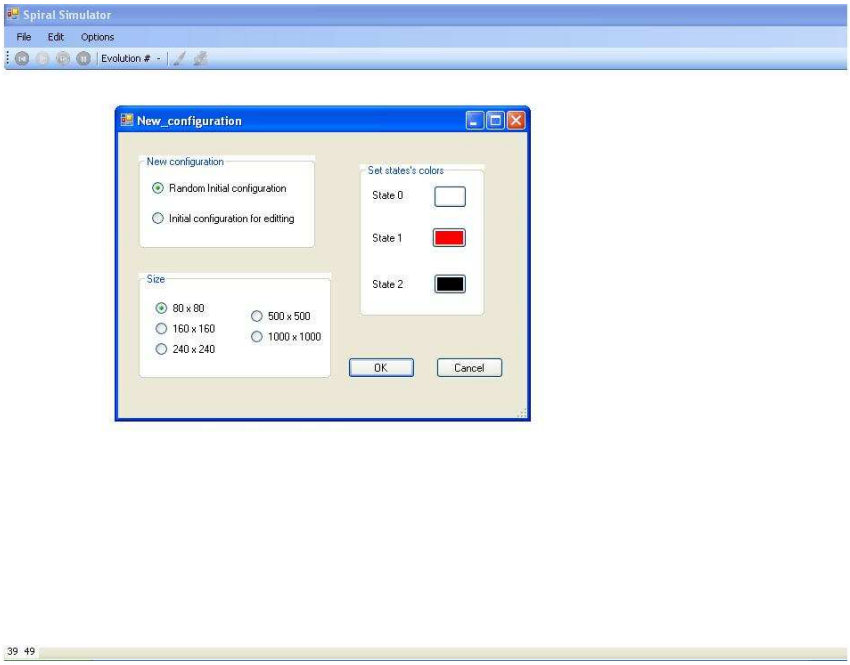

Fig. C.3: Nueva configuración.

localización de la opción mencionada.

C.3.2 Edit

En este menú se encuentra la opción de *Edit Lattice*, la cual tiene las mismas propiedades y características descritas en la sección configuración inicial editable; ver figura C.8.

También se pueden cambiar los colores de los estados; una vez seleccionando esa opción aparecerá un cuadro de diálogo con los tres estados. Para su cambio se hace el procedimiento descrito en la sección nueva configuración, ver figura C.9.

C.3.3 Options

En este menú se puede cambiar la velocidad y el zoom del espacio de evoluciones.

Cambio de velocidad

Existen tres velocidades: slow, medium y fast, que corresponden a tres velocidades predefinidas. De igual manera se tiene predefinido el zoom de los hexagonos: small, medium, big. El zoom se ejemplifica en la figura C.10.

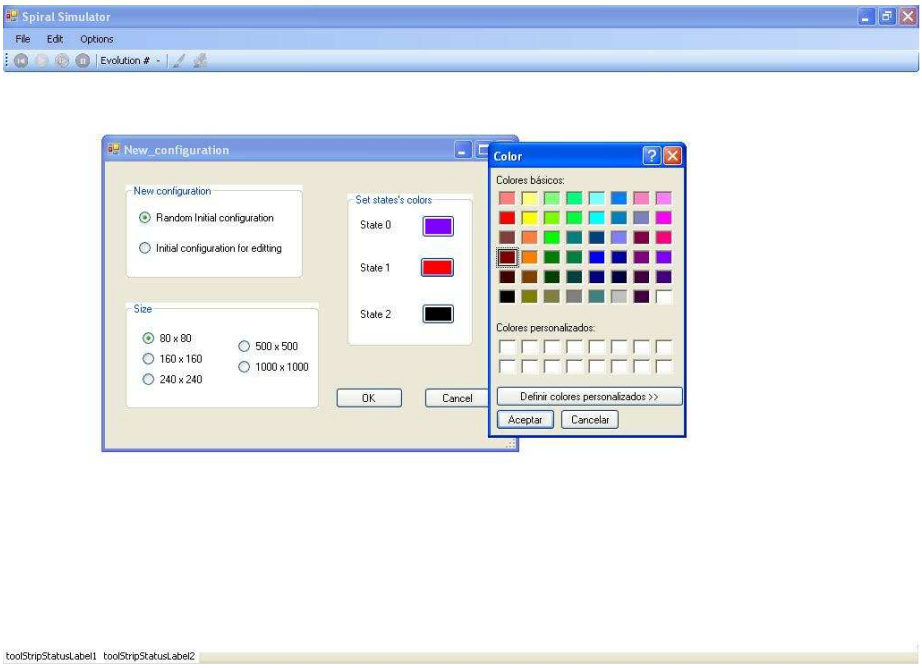

Fig. C.4: Cambiar colores iniciales.

Se selecciona la opción deseada y el zoom o la velocidad cambiarán inmediatamente.

C.4 Controlar la simulación

En la barra controladora de simulación, mostrada en la figura C.11, se puede observar que tiene varios botones, describiendolos de izquierda a derecha, realizan lo siguiente.

- *Reinciar.* Regresa a la configuración inicial.
- *Play.* Comienza la simulación.
- *Step by step.* Avanza un tiempo de la simulación.
- *Pause.* Detiene la simulación.

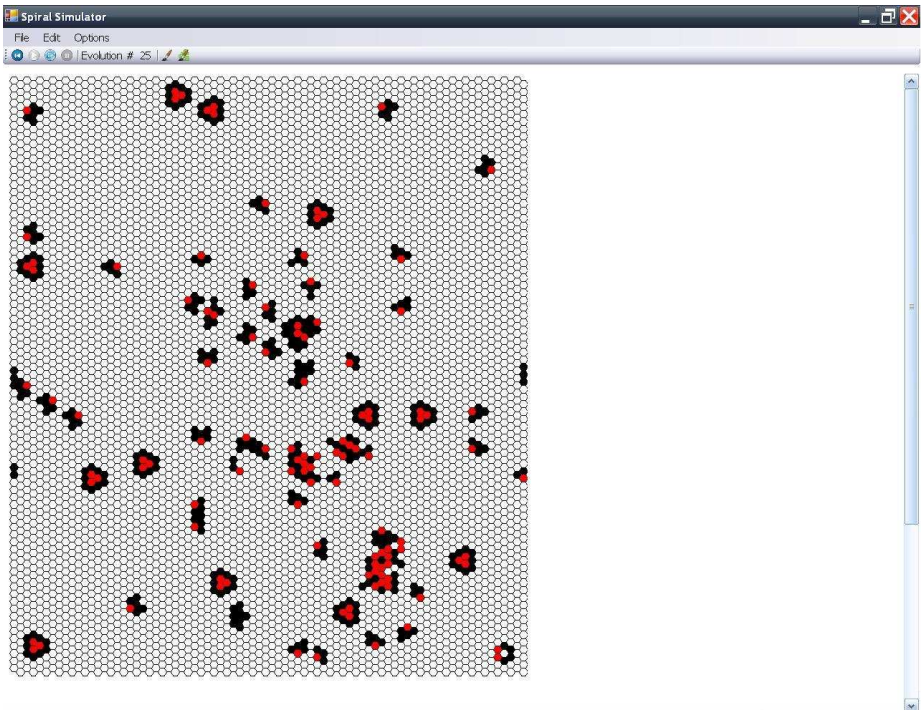

Fig. C.5: Edición de estados.

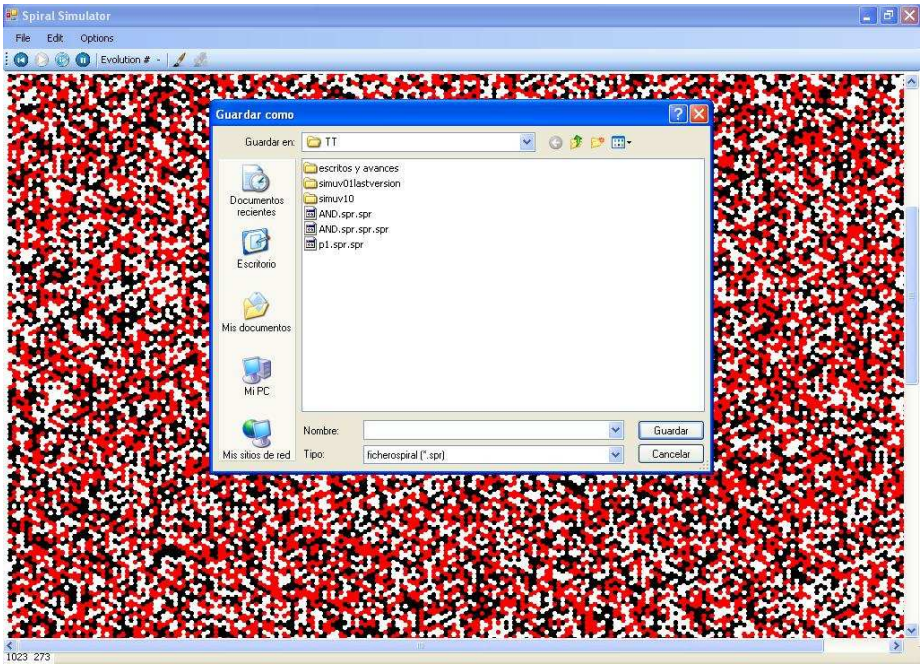

Fig. C.6: Guardar.

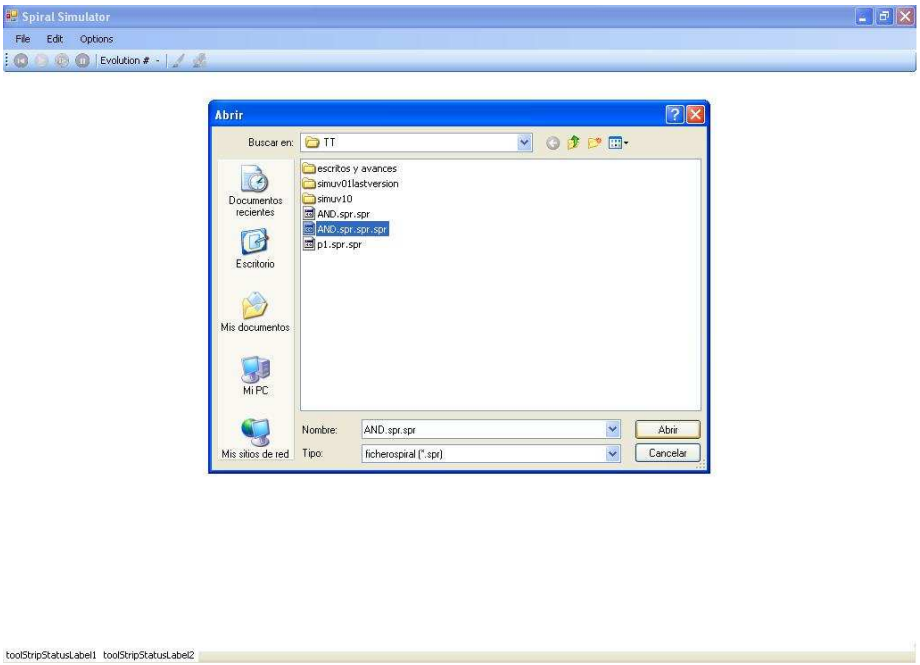
Spiral Simulator
File Edit Options
Evolution #
Abrir
Buscar en:
TT
escritos y avances
simuv01lastversion
simuv10
AND.spr.spr
AND.spr.spr.spr
p1.spr.spr
Documentos recientes
Escritorio
Mis documentos
Mi PC
Mis sitios de red
Nombre:
AND.spr.spr
Tipo:
ficherospiral (*.spr)
Abrir
Cancelar
toolStripStatusLabel1 toolStripStatusLabel2

Fig. C.7: Abrir.

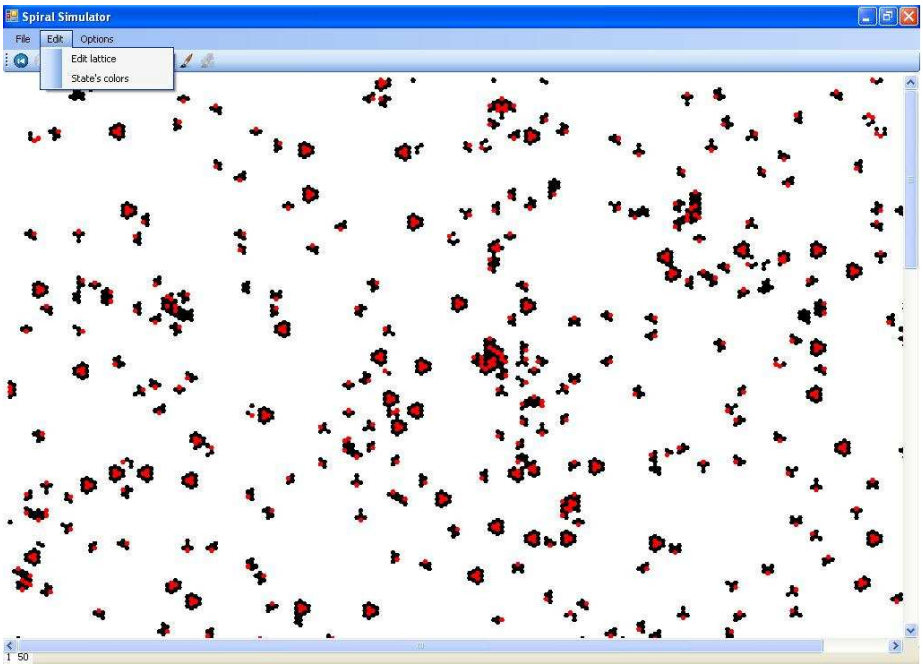

Fig. C.8: Menú *Edit*.

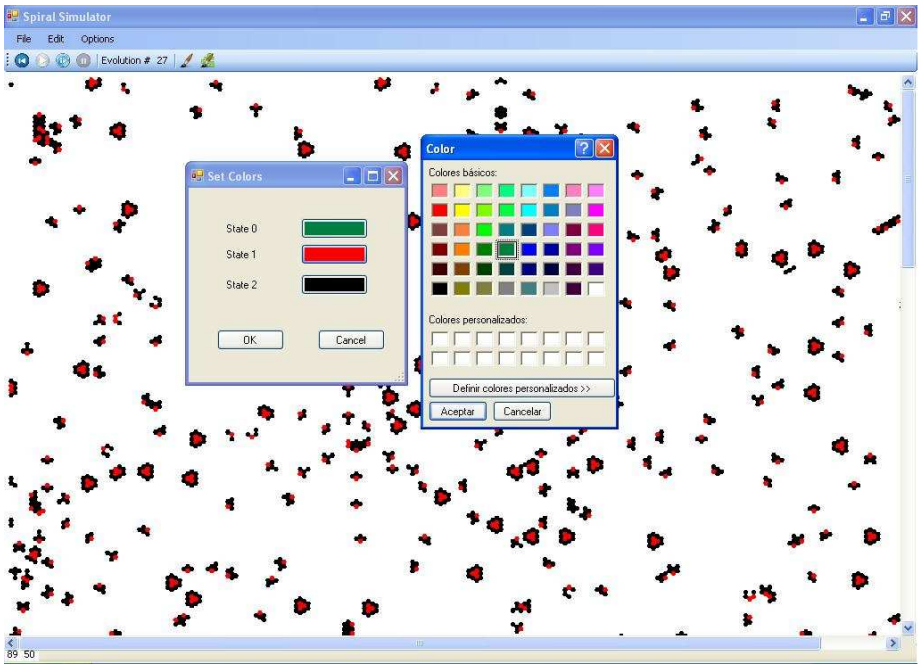

Fig. C.9: Edición de estados.

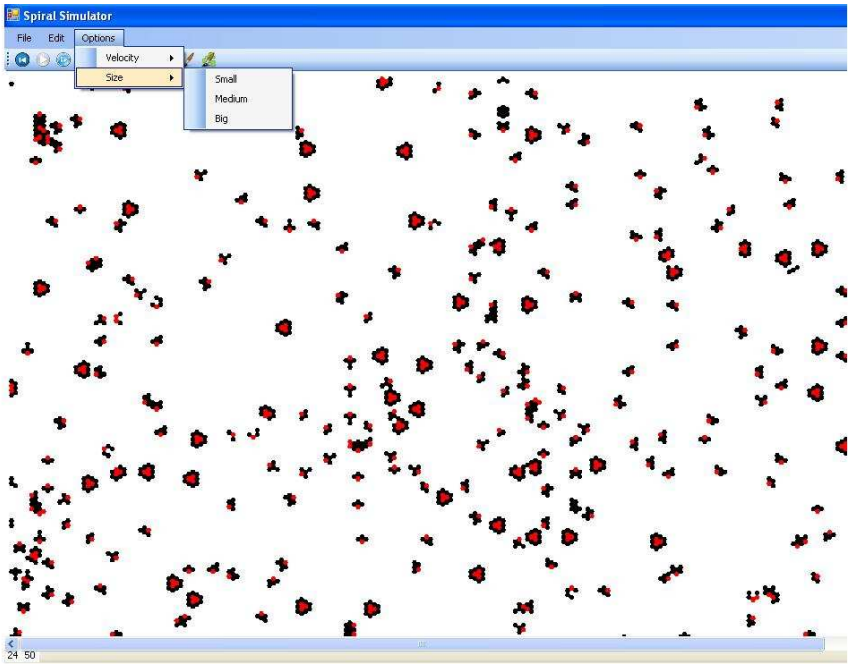

Fig. C.10: Edición de zoom.

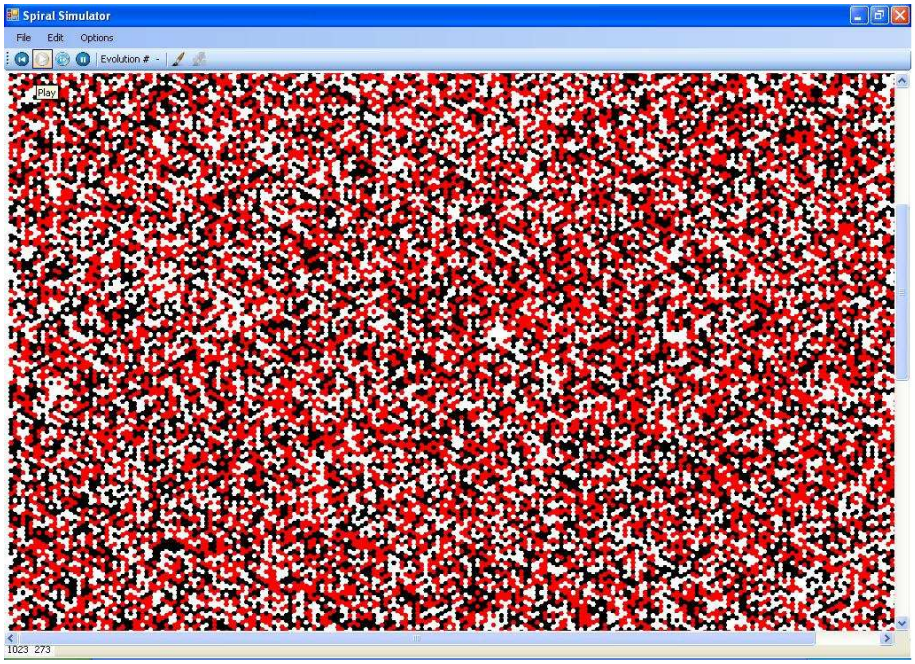

Fig. C.11: Controles de simulación.