Sistemas Informáticos
Curso 2003-04

Estudio de la evolución de vida artificial: ALiS (Artificial Life Simulation)

Realizado por:

Silvia García Díaz
Carmen Gutiérrez Ramírez
Raquel Hervás Ballesteros

Dirigido por:

Lourdes Araujo Serna

Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

RESUMEN

El proyecto ALiS surgió a partir de las ideas desarrolladas en el sistema de evolución digital Tierra, creado por Thomas S. Ray. Su creación fue motivada por el deseo de observar el proceso evolutivo en un medio distinto de la química del carbono, ya que nuestro conocimiento actual sobre la vida y la evolución se basa en un único ejemplo: la vida en la Tierra. Un entorno especialmente adecuado para ello es el medio digital.

ALiS es la simulación de un computador virtual donde la memoria principal está ocupada por programas escritos en un lenguaje ensamblador particular para este computador, cada uno de los cuales es considerado como una criatura del sistema. Estas criaturas proceden todas de una criatura ancestral cuya ejecución en el computador la lleva, simplemente, a copiarse a sí misma en memoria una y otra vez. Añadiendo alteraciones aleatorias sobre el código de esta criatura podrán aparecer en el sistema criaturas diferentes de la primitiva, es decir, evolucionadas.

Nuestro trabajo se ha fundamentado en conseguir en primer lugar simular el computador virtual básico y sobre él ajustar las mutaciones y cambios en el sistema necesarios para lograr la evolución del mismo. Conseguidos estos objetivos, hemos realizado un análisis exhaustivo de la evolución obtenida en él.

The ALiS project originated from the ideas of the digital evolution system Tierra, created by Thomas S. Ray. It was motivated by the possibility of studying the evolutionary process in an environment not based on the carbon chemistry, since our current knowledge about life and evolution is based on a single example: life on Earth. An environment which is specially appropriate for this is the digital one.

ALiS is the simulation of a virtual computer where the memory is filled with programs written in an assembler language that is particular for this computer, each one of them considered as a system creature. All these creatures come from an ancestral one whose execution in the computer makes copies of itself in memory time and again. Adding random changes in the code of this ancestor, there can appear creatures that are different from the ancestor, in other words, evolved creatures.

Our work has consisted of performing a simulation of the basic virtual computer, and fixing it with the mutations and changes needed to obtain evolution in it. Then, we have made an exhaustive analysis of the evolution obtained in the system.

Palabras clave: vida artificial, evolución, máquina virtual, genética, simulación de vida.

ÍNDICE

1.PRÓLOGO

El proyecto ALiS: Artificial Life Simulation (Simulación de Vida Artificial) surgió a partir de las ideas desarrolladas en el sistema de evolución digital llamado Tierra, cuyo creador es Thomas S. Ray. Dentro de la asignatura de Programación Evolutiva tuvimos un primer conocimiento de este sistema como un ejemplo muy particular de proyectos que hoy en día se están llevando a cabo dentro de este campo.

Como breve resumen podemos decir que Tierra es la simulación de un computador virtual donde la memoria principal está ocupada por programas, en un lenguaje ensamblador particular para ese computador, cada uno de los cuales es considerado como una criatura del sistema. Estas criaturas (programas) proceden todas de una criatura ancestral cuya ejecución en el computador la lleva, simplemente, a copiarse a sí misma en memoria una y otra vez. Añadiendo alteraciones aleatorias sobre el código de esta criatura podrán aparecer en el sistema criaturas diferentes a la primitiva, con código distinto, es decir, evolucionadas.

Por tanto, podemos obtener un medio evolutivo digital donde poder estudiar por una lado mecanismos de evolución generales que podrían estar presentes en cualquier medio, cumpliendo leyes estudiadas dentro de la Ciencia Biológica y, por otro, estudiarlos en un campo muy particular como es un computador virtual donde se pueden obtener resultados aplicados a la Ciencia Informática.

Partiendo de la documentación disponible para el sistema Tierra en www.his.atr.jp/~ray/tierra/ , hemos creado nuestro computador virtual con el objetivo de llegar a conseguir la evolución que describe el autor del sistema referenciado. Nuestro trabajo se ha fundamentado en conseguir en primer lugar simular el computador virtual básico y sobre él ajustar las mutaciones necesarias, cambios en el sistema, para lograr la evolución del mismo. Para ello, ha sido necesario comprender el funcionamiento de este medio para incluir en él las modificaciones aleatorias convenientes de forma que los cambios no le llevaran al completo caos. Conseguido este objetivo, hemos realizado un análisis exhaustivo de la evolución obtenida en él.

2.INTRODUCCIÓN

2.1. ¿DE DÓNDE VENIMOS? EL PROYECTO TIERRA

2.1.1. EVOLUCIÓN EN UN MEDIO DIGITAL

El proyecto Tierra (Ray [20,21,22]) surgió de la mente del ecologista tropical Thomas S. Ray a finales de 1989. Su creación fue motivada por el deseo de observar el proceso evolutivo en un medio distinto de la química del carbono. Si encontráramos este proceso evolutivo, sería otra instancia de la evolución, completamente independiente de la evolución que conocemos. En este caso se ampliaría nuestro conocimiento y entendimiento tanto de la vida como del proceso evolutivo que la genera, ya que nuestro conocimiento actual sobre la vida y la evolución se basa en un único ejemplo: la vida en la Tierra.

Con estas expectativas, se ha descubierto que podemos crear instancias de ciertos procesos vitales en un medio artificial, como por ejemplo un computador. Estas nuevas instancias de procesos vitales son conocidas como *Vida Artificial*, y el proyecto Tierra es un ejemplo de instancia de la evolución en un medio digital. Concretamente, Tierra es la simulación de un computador virtual con características especiales en cuanto a instrucciones máquina y gestión a nivel de sistema operativo, orientado al estudio de la evolución digital.

El proyecto Tierra se basa en la siguiente idea: si todas las formas de vida que existen hoy en día sobre la Tierra nos parecen descender de un organismo único muy simple, tal vez una molécula, que se habría replicado a sí misma, resulta que todas estas formas tienen rasgos comunes. Pero la evolución de las criaturas orgánicas es de una lentitud extrema, puesto que dura desde hace cuatro mil millones de años, y es entonces difícil observarla mientras se produce. Sin embargo, hay un medio para estudiar la evolución sin desbordar los límites de nuestra vida. Con las herramientas que tenemos hoy en día, basta con inocular el proceso de evolución en un ambiente artificial, es decir, creado por nosotros, y observar lo que se produce. De esta forma se podrían abstraer ciertas leyes generales que subyacen a cualquier proceso evolutivo, se dé éste en el medio que sea.

Quizás el más importante resultado del experimento Tierra fuera la activa dinámica co-evolutiva. Sistemas anteriores de evolución artificial no estaban construidos sobre entidades autorreplicantes, y por tanto tenían funciones de adaptación predefinidas (para la optimización de algoritmos genéticos, por ejemplo). En Tierra no hay funciones de adaptación predefinidas. Así la adaptación surge como resultado de la relativa habilidad de las variantes genéticas para sobrevivir y reproducirse. Significativamente, los organismos replicantes en el sistema forman una parte importante del entorno, conduciendo la evolución de adaptaciones a la presencia de otros organismos. Por tanto la naturaleza adaptativa del entorno evoluciona con los

organismos, al igual que ocurre en la evolución de la diversidad y complejidad de la biosfera terrestre.

Otro resultado interesante del experimento Tierra fue la demostración de que es posible evolucionar código máquina. En su momento, éste fue un resultado inesperado. Aparte de las adaptaciones ecológicas que aparecieron, los organismos también evolucionaron a importantes optimizaciones: organismos de aproximadamente un cuarto de tamaño del original que eran capaces de replicarse unas seis veces más rápido, e incluso optimizaciones más complejas, como el "desenrollado de bucles". Las criaturas iniciales introducidas en el sistema, para la copia de su código, se escribieron de forma que realizaban un bucle donde en cada iteración copiaban una de sus instrucciones. En la evolución del sistema se obtuvieron criaturas que presentaban esta la técnica de optimización de desenrollado: copiaban tres instrucciones en cada vuelta del bucle en vez de una sola, con lo que se realizan muchas menos iteraciones y se gana en tiempo de reproducción.

A lo largo de los años surgieron un considerable número de proyectos que pretendían emular o ampliar Tierra (Holland [15], Farmer et al. [13], Langton [16], Rasmussen et al.[19] y Bagley et al. [2]). Sin embargo, ninguno de estos proyectos pudo aportar una mejora significativa, y de hecho la mayoría presentaba una actividad evolutiva que era peor que la de Tierra.

Seguramente la mayoría de estos procesos fallaron debido a la falta de comprensión del medio digital, unido a una inapropiada tendencia a forzar que el medio inmaterial emulara al mundo natural. Cuando construimos sistemas evolutivos, naturalmente tomamos prestadas ideas de la evolución orgánica y las incorporamos al medio digital. El problema de qué propiedades tomar de la evolución orgánica e implementar en la evolución digital es crítico, y requiere ser extremadamente cuidadoso, ya que malas soluciones a esta cuestión son probablemente las que provocan los fracasos mencionados anteriormente.

El objetivo de los sistemas semejantes a Tierra es crear una instancia de la evolución por selección natural en un sistema artificial. Sin embargo, estos sistemas no deben ser simulaciones de la evolución orgánica. Si tomamos prestadas características de la evolución orgánica que nos obligan a crear una simulación de la misma, habremos fallado. Un ejemplo de un fallo repetido con frecuencia ha sido intentar redimensionar la memoria RAM del computador en un espacio bi o tridimensional, en la creencia de que esto permitiría mayor riqueza de interacciones.

Crear un buen sistema evolutivo en el medio digital es difícil por una sencilla razón. Debido a que estamos familiarizados con sólo una instancia de la vida y la evolución, la vida en la Tierra, nuestras mentes caen en algunas preconcepciones sobre la naturaleza del proceso. Muchas de estas ideas preconcebidas nos llevan a caminos equivocados cuando nos guían en el proceso de crear sistemas evolutivos digitales. Así que el primer paso para crear una instancia de evolución digital es liberar nuestra mente y permitirnos ver el medio digital como lo que es, no usarlo como un sistema para emular organismos naturales.

2.1.2. ORIGEN Y EVOLUCIÓN DE TIERRA

Cuando Thomas S. Ray se encontraba realizando estudios de postgrado en Harvard, llegó a su conocimiento la existencia de programas informáticos que se copian a sí mismos, y de ahí surgió la idea de Tierra: si comienzas haciendo un programa que se copia a sí mismo y le agregas mutación, estás obteniendo evolución.

Durante años se dedicó a la investigación en el campo de la informática para encontrar la manera de realizar la idea que tenía en mente. Si el lenguaje genético es tan robusto como para resistir mutaciones y recombinaciones, el lenguaje de los computadores también debería soportar el efecto de las mismas. La cuestión era averiguar si existían diferencias entre ambos lenguajes que pudieran provocar que uno pudiera evolucionar y el otro no. Thomas S. Ray se centró principalmente en dos ideas.

La primera a propósito del tamaño del ``juego de instrucciones'' dadas a un computador. El código genético se basa en un alfabeto de cuatro caracteres: los ácidos nucleicos. Estos ácidos, agrupados de tres en tres, son el código que define un aminoácido. Estos tres ácidos nucleicos pueden combinarse de 64 maneras diferentes (64 "codones"), pero no hay más que 20 aminoácidos, pues muchos "codones" designan el mismo aminoácido. El código genético es pues muy redundante. Las mutaciones implican el reemplazo, la inserción o la eliminación de un ácido nucleico. Resultan, pues, de cambios entre los 64 "codones", o más exactamente entre los 20 aminoácidos ensamblados en proteínas.

Si uno aplica el mismo análisis al lenguaje de los computadores, uno se percata de que en este lenguaje las mutaciones se ejercen sobre un número muy considerable de objetos posibles. En la más reciente generación del computador RISC (reduced instruction set) por ejemplo, cada unidad de información consta de 32 bits. Uno encuentra entonces, en el código binario de la máquina, 2^{32} objetos, es decir, más de 4 mil millones de objetos, sobre los cuales puede ejercerse una mutación. La intuición decía que la probabilidad de encontrar algo útil al efectuar 4 mil millones de combinaciones debía ser mucho más débil que al efectuar combinaciones entre 20 objetos solamente.

Thomas S. Ray se decidió entonces a reducir el número de objetos de información a 5 bits, creando un lenguaje de máquina que comprendiera 32 (es decir 2^5) instrucciones distintas. Podría pensarse que al reducir el número de objetos de información de 4 mil millones a 32 iba a mutilar la máquina. Pero no fue así. El número de operaciones reales que realizan los computadores (sumar, restar, multiplicar, etcétera) es relativamente reducido; en efecto, hay menos de cien. Ello es porque la mayor parte de los bits de la unidad de información de 32 bits no sirve para determinar el tipo de operación a realizar sino para designar los números que deben procesarse (los operandos). Así pues, se eliminaron del código los operandos numéricos, para que en el nuevo lenguaje máquina las instrucciones se aplicaran a los números que se encontraran en la unidad central de proceso, no en el código mismo. Esta modificación redujo automáticamente el tamaño del juego de instrucciones dado a la máquina. Se seleccionaron entonces 32 instrucciones para lanzar la primera experiencia.

Esta astucia provocó sin embargo un nuevo problema, que Thomas S. Ray resolvió con la segunda idea. Los programas informáticos no solamente afectan a los datos que se les suministra. A veces se afectan a sí mismos. De ordinario, el computador ejecuta las instrucciones máquina en secuencia lineal, siguiéndolas en el orden en que figuran en la memoria. Pero sucede también que a veces los programas se ramifican o hacen bucles, saltando de un lado a otro de la memoria. Los elementos de un programa deben a veces integrarse con otros elementos que pueden perfectamente encontrarse en lugares de la memoria muy alejados. En los computadores, estos elementos pueden llevar con ellos "la dirección" de los elementos con que deben reunirse. ¿Cómo hacen esto las moléculas de una célula viva? No llevan direcciones consigo, sino que presentan simplemente una superficie sobre la cual otra molécula puede fijarse o no, como la llave entra o no en una cerradura. Las moléculas se lanzan unas contra las otras por difusión en el organismo y, según sus formas sean o no complementarias, se integran o no. Según esto se podría utilizar un método del mismo género en el lenguaje máquina. Se desarrolló, entonces, un mecanismo para que el código, en lugar de especificar directamente adonde debía ir, lo incitara a ramificarse o a hacer un bucle cada vez que encontrara tal o cual configuración particular.

En breve, se integraron en el lenguaje de máquina estas dos ideas importadas directamente de la Biología: un número muy limitado de objetos de información y un sistema de direccionamiento por complementariedad de ciertas formas. Sobre estas bases, se diseñó un computador nuevo.

Así pues, Thomas S. Ray escribió un programa que simulaba el nuevo computador que tenía en la cabeza, el "computador virtual". Lo nombró con la palabra española Tierra. Pero para hacerla funcionar y probar su diseño, hacía falta todavía un programa virtual, evidentemente un programa que se copiara a sí mismo. Como ya existía un programa que se copiaba en un computador real, bastó con traducirlo a Tierran, el lenguaje de máquina del computador virtual Tierra.

La noche del 3 de enero de 1990, Thomas S. Ray tuvo éxito por primera vez en hacer funcionar el programa autorreplicante en el computador virtual. El rasgo más sorprendente de este universo digital fue que en el curso de su evolución, las criaturas descubrían sin cesar nuevas maneras de explotar a sus vecinos y de defenderse contra las tentativas de explotación lanzadas por ellos. La evolución es un proceso fundamentalmente egoísta, donde cada individuo se preocupa de sí mismo y mide su éxito según el número de genes propios que logra trasmitir a las siguientes generaciones. Para tener éxito en esta trasmisión, el organismo en evolución se muestra extraordinariamente inventivo, y se aprovecha automáticamente de todo lo que encuentra provechoso en su ambiente. Fueron muchos los datos que se obtuvieron esa noche, datos que le llevaron a iniciar una investigación sobre el proceso evolutivo del sistema.

Una vez que el ambiente se llena de criaturas, estas criaturas representan el recurso principal. Esa primera noche, muchas de las criaturas descubrieron que su ambiente era tan rico de información que ellos mismos no necesitaban llevarla toda consigo; para sobrevivir, les bastaba mirar en torno y servirse: apenas habían nacido y ya se convertían en parásitos. Otras

criaturas, seguramente más inocentes, se entretenían obstinadamente en duplicar todas las informaciones que ellos necesitaban. Los parásitos, pequeños y rápidos, llegaron a perfeccionar la técnica que les permitía utilizar la información de sus vecinos, y muy pronto dominaron la sopa biológica al replicarse en gran número.

Sin embargo, esta dominación no duró largo tiempo, pues los parásitos llegaron pronto a ser víctimas de su propio éxito. Apenas hubieron terminado de llenar su ambiente, comenzaron a tener trabajo en encontrar la información que necesitaban para sobrevivir. Privados de ella, comenzaron a morir.

La situación, no obstante, terminó por alcanzar una cierta estabilidad. A medida que los parásitos morían, sus huéspedes continuaron copiando para ellos mismos, laboriosamente, la información que a aquellos les hacía falta. Así los parásitos se salvaron de la extinción, y ellos y sus huéspedes entraron en una relación oscilante, los primeros reproduciéndose a expensas de los segundos, los segundos retomando importancia a medida que los primeros morían. Este tipo de oscilación de poblaciones de depredadores y sus presas (de parásitos y huéspedes) son bien conocidas en el mundo biológico; no son sino una de las numerosas y misteriosas maneras en que la evolución del universo digital reproduce la del universo orgánico.

Huéspedes y parásitos no se contentaban con crecer y decrecer siguiendo ciclos opuestos, sino que se lanzaban a una “carrera armamentista”. Los huéspedes desarrollaron mecanismos para defenderse contra los parásitos, y los parásitos técnicas para circunvalar las defensas de sus huéspedes. En este punto, no obstante, los huéspedes desarrollaron mecanismos que sacaban ventaja del hecho de estar siendo parasitados, para utilizar la energía de los parásitos en su propia reproducción y librarse así de su enemigo. En la práctica, los huéspedes dejaban que los parásitos los atacaran y se reprodujeran una vez, y enseguida les proporcionaban información falsa, la cual conducía a que los parásitos hicieran copias de sus huéspedes. Los huéspedes se convirtieron en una suerte de parásitos energéticos, de criaturas que, al principio, habían jugado contra ellos el rol de parásitos de información.

Al tener éxito en incitar a los parásitos a multiplicar el genoma de sus huéspedes y no el suyo propio, los huéspedes habían instalado en realidad un sistema de defensa absoluta, habían llegado a ser intocables, los verdaderos amos del mundo. O aún más: no había más que ellos en el mundo, nadie podía invadirlos. Siendo así, entonces, que evolucionaban en un mundo en que todas las criaturas eran parte de la misma familia, los huéspedes se hicieron confiados y comenzaron a cooperar unos con otros. Los huéspedes llegaron a ser criaturas sociales, interdependientes, que no podían reproducirse excepto si formaban grupos. Pero esta cooperación implicaba confianza, y la confianza puede ser traicionada. De hecho, poco después de que las criaturas se hicieron sociales, pero mucho tiempo después de que los parásitos originales hubieron desaparecido, una nueva especie de parásitos invadió la comunidad. Estos nuevos parásitos, llamados "los tramposos", se deslizaron entre las criaturas que vivían en familias que cooperaban entre sí; una vez que obtuvieron su confianza, los traicionaron. ¿Cómo? Jugándoles a sus confiadas víctimas la misma treta que estos les habían una vez jugado a los primeros parásitos a los cuales habían proporcionado falsas

informaciones, para llevarlos a duplicar el genoma de sus huéspedes y no el propio. Nueva etapa de la “carrera armamentista”.

Pero hubo un gran adelanto: en el curso del camino las criaturas habían descubierto el sexo. Cuando Thomas S. Ray suprimió las mutaciones para detener su evolución las criaturas siguieron evolucionando de todos modos. A fuerza de retomar la experiencia y observar su desarrollo, Thomas S. Ray comprendió que en el momento de replicarse las criaturas mezclaban sus genes, de suerte que su descendencia era diferente de cada uno de sus padres. Habían tomado su destino en sus propias manos.

Al fin se podía presenciar la evolución en el momento de suceder, en vez de simplemente observar sus resultados.

Después de estos primeros estudios Thomas S. Ray ha continuado todos estos años con la investigación de este sistema: probando distintos conjuntos de instrucciones, incluyendo operaciones genéticas, añadiendo diploidismo y desarrollando la versión en red.

2.2. Otros Sistemas de Simulación de Vida Artificial

Avida

Avida es un sistema auto-adaptativo diseñado para usarlo como plataforma en la investigación de la Vida Artificial o Digital. El sistema Avida está basado en conceptos similares a aquellos empleados por el sistema Tierra desarrollado por Thomas S. Ray. En términos generales, Avida es un mundo digital en el cual programas informáticos simples mutan y evolucionan. La población se adapta a la combinación de una aptitud intrínseca (autorreplicación) y una función de aptitud impuesta externamente (extrínseca) dada por el investigador.

Estudiando este sistema, se puede examinar la evolución de la adaptación, los rasgos generales de la vida en los sistemas (como la auto-organización) y otras cuestiones relacionadas con lo teórico y lo evolutivo de la Biología y los sistemas dinámicos. El punto fuerte de Avida es que proporciona un sistema digital controlable en el cual estudiar las teorías de la evolución biológica. A menudo, podemos estudiar elementos de la teoría de la evolución que son difíciles o imposibles de estudiar en sistemas biológicos.

Para más información sobre este sistema: http://dllab.caltech.edu/avida/

PHYSIS

Herramienta para estudiar los programas en código máquina autorreplicantes. Soporta distintas arquitecturas de máquina virtuales y diferentes conjuntos de instrucciones.

El sistema está implementado en Java y el genoma de la criatura ancestral tiene un menor número de instrucciones, con lo que se pretende ganar en velocidad de reproducción. El conjunto básico de instrucciones contiene 44 instrucciones.

El formato de las instrucciones es el siguiente:

I instruction [operand…]

[instruction[operand…]]…

Por ejemplo: I move 0 1 copia el IP al primer elemento estructural, guardando la dirección de la instrucción que está siendo ejecutada. Cada número entero identifica únicamente a un elemento estructural, a una instrucción o a un operando dependiendo del contexto.

Además de estudiar cómo varía la velocidad autorreplicante cambiando el genoma de la criatura ancestral con otro tipo de instrucciones, estudian el aprendizaje evolutivo de tareas computacionales llegando a la conclusión de que no aprende un organismo individualmente, sino a través de su descendencia.

Para más información sobre este sistema: http://physis.sourceforge.net/

HELIX4

El programa comienza cargando un organismo inicial desde un archivo. Cada criatura simplemente intenta hacer copias de sí misma. Cuanto más rápido se ejecute, mayor número de copias de sí misma podrá hacer. Cuando un organismo se ejecuta, comete errores. En tal caso su descendencia será representada en distinto color. Esa descendencia puede ser más o menos eficiente, por lo que sobreviven las mejores según el darwinismo. Cuando la memoria se llena los organismos más viejos son eliminados (así que desaparecen de la interfaz gráfica) y dejarán espacio para nuevos nacimientos.

Para más información sobre este sistema: http://www.necrobones.com/alife/helix.htm

COREWORLD

Coreworld es un mundo abstracto habitado por programas en lenguaje ensamblador. Este lenguaje (una versión extendida del Redcode Corewar) permite a los programas usar operaciones quánticas en bits quánticos (qubits).

Por una parte, el resultado más intrigante de esta investigación es la posibilidad de que las formas de vida en el Coreworld puedan ayudarnos a reconocer tal comportamiento en el mundo real.

Las operaciones permitidas en el mundo clásico y en el basado en el quanto (especialmente las operaciones que podemos modelar en un computador digital) son sólo sutilmente diferentes. El Quantum Coreworld es un modelo específico en el cual estas diferencias pueden ser estudiadas; esta exploración puede conducir a una comprensión mejor del mundo tal y como es.

Para más información sobre este sistema:
http://genetics.med.harvard.edu/~await/qtaas/index.html

AMOEBA

Amoeba modela la sopa primitiva como una porción de memoria. Cada dirección de memoria puede ser ocupada por un único organismo digital o "célula". Una célula es por sí misma un computador, aunque extremadamente simple: se comparan cuatro registros de carga, un contador de programa y una secuencia de instrucciones que conforman el código genético. Una célula vive cuando ejecuta sus instrucciones.

La principal diferencia entre Amoeba y Tierra consiste en que en Tierra las formas de vida deben ser introducidas manualmente en la sopa; en Amoeba, la sopa es sembrada con instrucciones aleatorias, y las formas de vida aparecen espontáneamente, como por "arte de magia".

Una razón para que se dé esta fecundidad es que las formas de vida en Amoeba tienen una estructura muy simple. El código genético de cada célula es una secuencia de instrucciones de cómo mucho 30 instrucciones, elegidas de un repertorio de 16 posibles instrucciones. Éstas están basadas en operaciones estándar como "saltar a una dirección","copiar el contenido de la posición de memoria A en la posición B", y "comparar registros y saltar la próxima instrucción". El autor de este sistema, Pergellis, ha diseñado cuidadosamente su conjunto de instrucciones de manera que sea posible crear una célula autorreplicante con tan sólo 5 instrucciones.

Al comienzo de una simulación, el 70% de la sopa es sembrada con células que contienen de 1 a 25 instrucciones elegidas aleatoriamente. El resto de la sopa permanece vacía. Amoeba va visitando cada célula y ejecuta sus instrucciones. En este momento, la mayoría de las células contendrán secuencias de instrucciones sin significado, y sus ejecuciones no tendrán efecto en la sopa. Unas pocas células, sin embargo, tendrán instrucciones que acceden a otras posiciones de memoria, copiando instrucciones en ellas o exhibiendo un comportamiento vírico invadiendo otra célula y ejecutando su código. Siempre que 1000 posiciones en la sopa contengan código, Amoeba invoca a la rutina del reaper, el cual eliminará el 30% de las células e introducirá unas cuantas células nuevas aleatorias en la sopa. Entonces es cuando una nueva generación comienza.

Pargellis encontró que en media, una secuencia de instrucciones autorreplicante emergería espontáneamente después de unas 400 generaciones. Inicialmente, eran innecesariamente complejas y contenían porciones de código que no realizaban ninguna tarea útil.

Debido a que las células compiten por los recursos del tiempo de procesador y el espacio de memoria, es por lo que Amoeba modela la evolución y la selección natural. Siempre que una célula se reproduce, hay un 10% de probabilidad de que una de sus instrucciones mute por otra instrucción. La mayoría de las mutaciones estropea la secuencia de instrucciones e inhibe la autorreplicación. En un 3% de los casos, sin embargo, Pargellis observó que la mutación permitía que la autorreplicación continuara y permitía a las células despojarse de su código redundante, vivir por más tiempo y reproducirse más eficientemente.

2.3. ¿ADÓNDE VAMOS? SISTEMA ALIS

Partiendo de la descripción básica de computador virtual dada por Thomas S. Ray, nuestro objetivo ha sido implementar un sistema de simulación de vida artificial, nuestro propio computador virtual, para estudiar la capacidad evolutiva de un sistema de este tipo.

Tomando la especificación del computador virtual de Thomas S. Ray, hemos construido nuestro computador determinista. En este computador se puede ejecutar cualquier programa escrito en el lenguaje ensamblador empleado en ALiS, en especial el programa que autorreplica su contenido en memoria obteniendo una población de criaturas (programas) exactamente iguales que habitan en la sopa (memoria).

Estos programas no pueden ser completamente deterministas, sino que deben permitir características, como la mutación, para dar lugar a los procesos evolutivos del medio. En Tierra se definen una serie de mutaciones que hemos aplicado y adaptado a nuestro propio computador, donde además hemos definido otras alteraciones. Para ello, partiendo del sistema determinista, hemos introducido distintos parámetros y cambios aleatorios que producirán cambios en el comportamiento determinista de éste, provocando la aparición de las formas de vida digital objeto de nuestro estudio. Estos mecanismos que dan lugar al indeterminismo serán distintos tipos de mutaciones, tanto a nivel de comportamiento como a nivel de código genético. Aunque la idea básica de las mutaciones partió también del sistema Tierra, ha sido el propio estudio del comportamiento del computador el que nos ha llevado a implementar y ajustar la serie de mutaciones que se pueden ejecutar en ALiS.

Así, una vez que hemos ajustado los cambios aleatorios en el sistema hemos podido estudiar las formas de evolución que aparecen en el medio digital recreado por ALiS.

3.ALiS: SIMULACIÓN DE VIDA ARTIFICIAL

La intención de este proyecto es sintetizar más que simular la vida. Esta aproximación comienza con un conjunto de organismos capaces de reproducirse y la posibilidad de una evolución infinita, y pretende generar diversidad y complejidad de manera semejante a la explosión del Cámbrico. Establecer esta meta lleva a problemas semánticos, porque la vida debe ser definida de una manera que no se limite a las formas de vida basadas en el carbono. No es probable que pudiera haber un consenso respecto a tal definición, ni incluso respecto a si la vida no necesita basarse en el carbono. Por tanto, tomaremos la vida en su sentido más general. Un sistema estará vivo si puede reproducirse y es capaz de una evolución sin fin. Por tanto, la vida sintética deberá ser autorreplicante y evolucionar en estructuras y procesos que no hayan sido diseñados o preconcebidos por el creador (Pattee [18]; Cariani [3]).

Los virus informáticos y gusanos (Cohen [4]; Dewdney [8,9,11,12]; Denning [7]; Rheingold [23]; Spafford et al. [24]) son capaces de autorreplicación, pero afortunadamente no de evolución. Por tanto no se les puede considerar organismos vivos, porque seguramente no serán capaces de evolucionar. Por otro lado, la mayoría de las evoluciones simuladas no son infinitas. Su potencial es limitado por la estructura del modelo, que generalmente asigna a cada individuo un genoma consistente en un conjunto de genes predefinidos, cada uno de los cuales debe existir en un conjunto de alelos (Holland [14]; Dewdney [10]; Dawkins [5,6]; Packard [17]; Ackley & Littman [1]). El objeto que evoluciona es generalmente una estructura de datos representando el genoma, que el programa simulador muta y/o recombina, selecciona y copia según cierto criterio designado en el simulador. Las estructuras de datos no contienen el mecanismo de la reproducción, simplemente son copiadas por el simulador si sobreviven a la fase de selección.

La autorreplicación es crítica para sintetizar vida, porque sin ella, los mecanismos de selección deberán ser también predeterminados por el simulador. Esta selección artificial jamás podrá ser tan creativa como la selección natural. Los organismos no son libres de inventar sus propias funciones de adaptación. Sin embargo, las criaturas con libre evolución descubrirán medios de explotación mutua y sus funciones de adaptación asociadas, que nunca habríamos imaginado.

La aproximación presentada aquí no tiene estas restricciones. Aunque el modelo está limitado a la evolución de criaturas basadas en secuencias de instrucciones máquina, puede tener un potencial comparable a la evolución basada en secuencias de moléculas orgánicas. Los organismos digitales han sido sintetizados basándose en una metáfora de la vida orgánica. En ella el tiempo de cpu es la "energía" y la memoria es la "materia". La memoria está organizada en patrones "genéticos" que explotan el tiempo de cpu para autorreplicarse. La mutación genera nuevas formas, y la evolución ocurre por selección natural cuando distintos "genotipos" compiten por el tiempo de cpu y el espacio en memoria. Además, aparecen nuevos

genotipos que explotan a otras "criaturas" para conseguir de ellas recursos energéticos y de información.

3.1. ¿EN QUÉ CONSISTE?

La vida orgánica se puede entender como la utilización de energía, la mayoría procedente del sol, para organizar materia. Análogamente, la vida digital se puede considerar como el uso de tiempo de cpu para organizar memoria. La vida orgánica evoluciona según la selección natural cuando los individuos compiten por los recursos (luz, comida, espacio, ...) de manera que los genotipos que dejan mayor descendencia incrementan su frecuencia. La vida digital evoluciona siguiendo el mismo proceso, según los algoritmos replicantes compiten por el tiempo de cpu y el espacio en la memoria, y los organismos evolucionan hacia estrategias para explotar a otros.

La memoria, la cpu y el sistema operativo del computador son considerados como elementos del medio ambiente físico. Una “criatura” es diseñada por tanto para adaptarse específicamente a las características del medio computacional. Así la criatura consiste en un programa autorreplicante escrito en un lenguaje ensamblador especial. Un bloque de la memoria RAM del computador es designada como la "sopa" que puede ser inoculada con criaturas. El “genoma” de las criaturas consiste en una secuencia de instrucciones máquina que forman el algoritmo autorreproductor de la criatura. El prototipo de criatura consiste en 80 instrucciones máquina, de manera que el tamaño de su genoma es 80 instrucciones, y su “genotipo” es el nombre que se le asigna a este genoma. A esta criatura se la llama “criatura ancestral”.

3.2. EL COMPUTADOR VIRTUAL

Los computadores actuales son capaces de emular a través de software el comportamiento de otros computadores que se podrían construir. Utilizaremos esta flexibilidad para diseñar un computador que sea especialmente habitable por la vida sintética que pretendemos crear.

Hay muchas razones por las que no es adecuado intentar sintetizar organismos digitales que exploten los códigos máquina y los sistemas operativos de máquinas reales. Una es la posibilidad de que la evolución natural de los códigos máquina lleve a la aparición de virus o gusanos que puedan ser difíciles de erradicar debido a sus genotipos cambiantes. Esta amenaza potencial aconseja crear exclusivamente programas que funcionen en computadores virtuales con su correspondiente sistema operativo virtual. Estos programas no serán nada más que datos en el computador real, y por tanto no presentarán más amenaza que un dato en una base de datos o un fichero de texto en un procesador de textos.

Otra razón para evitar desarrollar los organismos digitales en código máquina de un computador real, es que el sistema artificial estará atado al hardware de la máquina, y quedará obsoleto tan rápidamente como la máquina para la que se diseñó. En cambio, un sistema

artificial desarrollado en una máquina virtual podrá ser portado fácilmente a nuevas máquinas reales según vayan estando disponibles.

Un tercer hecho, el principal, es que el lenguaje máquina de los computadores reales no está diseñado para evolucionar, y de hecho no soporta evolución significativa: cualquier mutación o recombinación en el código de una máquina real producirá con seguridad un programa no funcional. Este problema puede ser mitigado diseñando un computador virtual cuyo código máquina sea diseñando teniendo en mente la evolución.

El trabajo descrito aquí tiene lugar en un computador virtual llamado ALiS (Artificial Life Simulation). ALiS es un computador paralelo de tipo MIMD (multiple instruction, multiple data), con un procesador (cpu) para cada criatura. El paralelismo es simulado, permitiendo a cada cpu ejecutar un pequeño periodo de tiempo por turno.

Cada cpu de este computador virtual contiene elementos típicos de unidad de proceso de cualquier computador. En concreto, la cpu de cada criatura contiene dos registros de direcciones (ax y bx), dos registros de datos (cx y dx), una pila y puntero a su cima (SP) y el puntero de instrucción (IP) que contendrá la dirección de memoria a ejecutar.

El conjunto de instrucciones de una cpu lleva a cabo operaciones aritméticas o manipulación de bits, dentro del pequeño conjunto de registros que contiene la cpu. Algunas instrucciones mueven datos entre los registros de la cpu, o entre los registros de la cpu y la sopa. Otras instrucciones controlan la localización y movimiento del puntero de instrucción (IP). El IP indica una posición de la sopa, donde se encuentra el código máquina del programa en ejecución (en este caso el organismo digital).

La cpu lleva a cabo perpetuamente un ciclo carga-decodificación-ejecución-incremento del IP: la instrucción máquina que se encuentra en la dirección apuntada por el IP es llevada a la cpu, su patrón de bits es descodificado para determinar de qué instrucción se trata, y la instrucción es ejecutada. Después el IP es incrementado para apuntar a la siguiente posición en la sopa, desde donde la siguiente instrucción será leída. Sin embargo, algunas instrucciones de salto o de llamadas a procedimientos manipulan directamente el IP, causando que la ejecución salte a otra secuencia de instrucciones en la memoria.

3.3. El Sistema Operativo

El computador virtual ALiS necesita un sistema operativo virtual que sea habitable para los organismos digitales. El sistema operativo determinará los mecanismos de comunicación entre procesos, tratamiento de memoria, y repartición del tiempo de cpu entre los procesos competidores. Los algoritmos evolucionarán para explotar estas características en su propio beneficio. Más que ser un mero aspecto del medio ambiente, el sistema operativo junto con el conjunto de instrucciones determinará la topología de las posibles interacciones entre individuos, tales como las relaciones depredador-presa o parásito-huésped.

3.3.1. GESTIÓN DE MEMORIA

El computador de ALiS opera sobre un bloque de memoria RAM del computador real que se ha reservado para este propósito. Este bloque de memoria se denomina "sopa". En este proyecto la sopa está formada por 60.000 bytes[1], que pueden alojar el mismo número de instrucciones máquina, a cada una de las cuales se le ha asignado un carácter de forma unívoca. Es con estos caracteres con los que trabaja el sistema.

Las criaturas de ALiS son consideradas celulares en el sentido de que están protegidas por una membrana semipermeable de localizaciones en memoria y es el sistema operativo de ALiS el que proporciona servicios de localización en la sopa. El tamaño de la criatura es justo el tamaño de su bloque de memoria, que usualmente corresponde con el tamaño de su genoma. La membrana de las criaturas es descrita como semipermeable porque aunque los privilegios de escritura están protegidos (cada criatura tiene privilegios exclusivos de escritura dentro de su bloque de memoria), los de lectura y ejecución no lo están. Así una criatura puede examinar el código de otra, e incluso ejecutarlo, pero no podrá escribir en él. Cada criatura tendrá privilegios exclusivos de escritura en dos bloques de memoria: el bloque en el que nació, al que nos referiremos como “célula madre”, y un segundo bloque que obtendrá mediante la ejecución de la instrucción correspondiente para solicitar memoria. Este segundo bloque, al que nos referiremos como “célula hija”, será usado para reproducirse. Cuando una criatura se divide, la célula madre pierde sus privilegios de escritura sobre la célula hija, pero a cambio puede solicitar un nuevo bloque de memoria para reproducirse. En el momento de la división, a la célula hija se le da su propia cpu, y puede solicitar su segundo bloque de memoria.

La sopa no guarda ningún distintivo de las posiciones de inicio o final de las criaturas, sino que serán las propias criaturas las que conocerán sus direcciones de inicio y final, y leerán directamente de la memoria si es necesario. Así la sopa se simplifica en una sucesión de caracteres, cada uno de los cuales representa una instrucción determinada.

Para la gestión de la memoria libre se guarda una lista de espacio libre, que indica las posiciones de inicio y final de los trozos de la sopa no ocupados por ninguna criatura. Sin embargo, ya que al eliminar una criatura no se borra su código de la memoria, el espacio libre de la sopa contendrá instrucciones que han dejado allí criaturas eliminadas. Este código puede se ejecutado y leído por las criaturas vivas del sistema, y al ser memoria libre, reescrito si el espacio es reservado por alguna criatura para reproducirse.

[1] El tamaño de la memoria viene dado por el parámetro del sistema TAM_MEM. Por defecto su valor es 60.000. [Ver apartado 3.8. Parámetros del sistema]

3.3.2. DISTRIBUCIÓN DEL TIEMPO – EL SLICER

El sistema operativo de ALiS debe ser multi-tarea (o paralelo) de manera que una comunidad de criaturas individuales pueda vivir en la sopa simultáneamente. El sistema proporciona pequeños tiempos de cpu a cada criatura de la sopa por turno. Para ello existe una cola circular denominada "slicer". Cuando una criatura nace, una cpu virtual se crea para ella, y además entra en el slicer justo delante de su madre, que es la criatura activa en ese momento. Así la recién nacida será la última criatura en la sopa en conseguir su tiempo de ejecución en la cpu, y la madre será la siguiente después de la hija. Como el tamaño de los turnos asignados a cada criatura es pequeño en relación con el tiempo de generación de las criaturas, este sistema de distribución del tiempo de cpu se aproxima al paralelismo. Según esto, el sistema tiene una población de cpu's virtuales, cada una de las cuales toma un trozo de tiempo de la cpu real cuando llega su turno en la cola slicer.

3.3.3. MORTALIDAD – EL REAPER

Las criaturas autorreplicantes en un tamaño de sopa constante rápidamente llenarían la sopa y bloquearían el sistema. Para evitar esto, es necesario incluir mortalidad. El sistema operativo de ALiS incluye un "reaper" que empieza a matar criaturas de la cola cuando la memoria se llena hasta cierto nivel (por ejemplo, el 80%)[2]. Las criaturas son eliminadas desasignando su bloque de memoria y quitándolas de las colas slicer y reaper. Su código “muerto” no se retira de la sopa.

El reaper usa una cola lineal. Cuando una criatura nace entra al final de la cola. El reaper siempre mata a la primera criatura de la cola. Sin embargo, los individuos se pueden mover arriba y abajo en el reaper de acuerdo con su éxito o fracaso al ejecutar ciertas instrucciones. Cuando una criatura ejecuta una instrucción que genera una condición de error, se mueve una posición arriba en la cola, siempre que el individuo por delante de él no tenga acumulados un número mayor de errores. También hay otras instrucciones que es especialmente complicado ejecutar sin generar ningún error, por lo que la ejecución exitosa de las mismas moverá a la criatura abajo una posición en el reaper, siempre que no tenga acumulados más errores que la criatura por debajo de ella.

El efecto del reaper es que algoritmos que están fundamentalmente dañados suban a la cima de la cola y mueran. Los algoritmos vigorosos tienen mayor longevidad, pero en general la probabilidad de muerte aumenta con la edad.

[2] La proporción de sopa que se mantendrá libre vendrá determinado por el parámetro PROP_MEM_LIBRE. [Ver apartado 3.8. Parámetros del sistema}

3.4. EL LENGUAJE “LALiS”

Antes de intentar implementar un sistema de vida sintética, se debe pensar cuidadosamente en cómo la representación de un lenguaje de programación afecta a su adaptabilidad en el sentido de ser robusto a operaciones genéticas como la mutación y la recombinación. La naturaleza del computador virtual es determinada en gran parte por el conjunto de instrucciones de su lenguaje máquina. Lo que se ha pretendido en este trabajo es dar mayor flexibilidad al código máquina, creando un sistema computacional híbrido entre los procesos biológicos y los clásicos procesos de von Neumann. Dos características han sido tomadas del mundo biológico que se consideran críticas para la capacidad de evolución del lenguaje LALiS.

Primero, el conjunto de instrucciones ha sido definido para ser de un tamaño del mismo orden de magnitud que el código genético. La información es codificada en el DNA usando 64 codones, que son traducidos en 20 aminoácidos. En nuestro caso, LALiS consiste en 32 instrucciones, que pueden ser representadas con 5 bits, operandos incluidos. Para hacer un código máquina con un conjunto de instrucciones realmente pequeño, debemos prescindir de los operandos numéricos. Esto se puede llevar a cabo haciendo que los registros de la cpu y la pila sean los únicos operandos del conjunto de instrucciones. Cuando necesitemos codificar un entero por alguna razón, bastará crear un registro numérico usando manipulaciones en los bits: intercambiando el último bit y desplazando a la izquierda, por ejemplo.

La segunda característica que ha sido tomada de la biología molecular en el diseño del lenguaje LALiS es el modo de direccionamiento, que llamaremos “direccionamiento por plantilla”. En muchos códigos máquina, cuando un dato es referenciado, o el IP salta a otro trozo de código, la dirección numérica exacta del dato o del código destino es especificada en el código máquina. Hay que considerar que los sistemas biológicos funcionan al contrario, de manera que cuando una proteína A en el citoplasma de una célula interactúa con una proteína B, no especifica las coordenadas exactas de B. En su lugar, la molécula A presenta una plantilla en su superficie que es complementaria a cierta superficie de B. La difusión une ambas plantillas, y las formaciones complementarias les permiten interactuar. El direccionamiento por plantillas se puede ver en el comportamiento de instrucciones como las de salto. Las plantillas en nuestro sistema están formadas por secuencias de instrucciones no-operación (*nop_0* y *nop_1*). A continuación de instrucciones de bifurcación se situará una plantilla formada por dichas instrucciones. Al llegar a esta instrucción se bifurcará a la dirección de memoria más próxima (la búsqueda podrá ser ascendente, descendente o bidireccional según la instrucción) que esté a continuación de una plantilla complementaria a la que acompaña a la instrucción ejecutada. Como es lógico, la complementariedad se dará transformando las instrucciones *nop_0* en *nop_1* y viceversa. En el siguiente apartado se podrá ver este funcionamiento en detalle.

3.4.1. CONJUNTO DE INSTRUCCIONES

En general, los registros ax y bx se usan para almacenar direcciones de memoria de la sopa. Los valores contenidos en ambos se mantienen como enteros positivos, módulo el tamaño de la sopa. Los registros cx y dx se usan generalmente para guardar números, que pueden ser tanto positivos como negativos. Así, se fuerza a que los valores de ambos permanezcan en el rango -TAM_MEM y TAM_MEM[3]. Cualquier operación que cause que el valor de cx o dx se salga de este rango, hará que el valor del registro se ponga a 0.

Muchas de las instrucciones fallarán bajo ciertas condiciones, que especificaremos más adelante. Cuando una instrucción falla, no hace nada más que incrementar el puntero de instrucción y aumentar el número de fallos de la criatura a la que pertenece.

A continuación sigue una descripción detallada del comportamiento usual de las 32 instrucciones del lenguaje. Sin embargo, las instrucciones de LALiS no son deterministas, y ciertas mutaciones pueden provocar que se comporten de forma distinta a la habitual. Estas mutaciones se expondrán más adelante.

Por otro lado, hay que tener en cuenta que en general las instrucciones no modifican el puntero de instrucción, sino que este incremento se hace posteriormente.

NOP0, NOP1:

Estas dos instrucciones son no-operaciones, no hacen nada. Se usan como componentes de las plantillas. No hay ninguna condición bajo la que estas instrucciones fallen.

OR1:

Esta instrucción cambia el bit de menor orden del registro cx. No hay ninguna condición bajo la que esta instrucción falle.

SHL:

Esta instrucción desplaza todos los bits del registro cx una posición hacia la izquierda, poniendo un 0 en el bit menos significativo del registro. A nivel binario esto es equivalente a multiplicar por 2.

No hay ninguna condición bajo la que esta instrucción falle.

ZERO:

Esta instrucción pone a 0 el registro cx. No hay ninguna condición bajo la que esta instrucción falle.

[3] El tamaño de la memoria viene dado por el parámetro del sistema TAM_MEM. Por defecto su valor es 60.000. [Ver apartado 3.8. Parámetros del sistema]

IF_CZ:

Esta instrucción sólo incrementa el puntero de instrucción en 1 si el valor del registro cx es distinto de 0. Esto significa que la instrucción siguiente a la actual será ejecutada sólo si el valor del registro cx es 0.

No hay ninguna condición bajo la que esta instrucción falle.

SUB_AB:

Esta instrucción resta el valor contenido en el registro bx al valor contenido en el registro ax, y almacena el resultado en el registro cx.

No hay ninguna condición bajo la que esta instrucción falle.

SUB_Ac:

Esta instrucción resta el valor contenido en el registro cx al valor contenido en el registro ax, y almacena el resultado en el registro ax.

No hay ninguna condición bajo la que esta instrucción falle.

INC_A:

Esta instrucción incrementa en uno el valor del registro ax, volviendo a almacenar el resultado en ax.

No hay ninguna condición bajo la que esta instrucción falle.

INC_B:

Esta instrucción incrementa en uno el valor del registro bx, volviendo a almacenar el resultado en bx.

No hay ninguna condición bajo la que esta instrucción falle.

INC_C:

Esta instrucción incrementa en uno el valor del registro cx, volviendo a almacenar el resultado en cx.

No hay ninguna condición bajo la que esta instrucción falle.

DEC_C:

Esta instrucción decrementa en uno el valor del registro cx, volviendo a almacenar el resultado en cx.

No hay ninguna condición bajo la que esta instrucción falle.

PUSH_AX:

Esta instrucción apila el valor que contiene el registro ax en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA[4]).

No hay ninguna condición bajo la que esta instrucción falle.

PUSH_BX:

Esta instrucción apila el valor que contiene el registro bx en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA).

No hay ninguna condición bajo la que esta instrucción falle.

PUSH_CX:

Esta instrucción apila el valor que contiene el registro cx en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA).

No hay ninguna condición bajo la que esta instrucción falle.

PUSH_DX:

Esta instrucción apila el valor que contiene el registro dx en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA).

No hay ninguna condición bajo la que esta instrucción falle.

POP_AX:

Esta instrucción desapila la cima de la pila y almacena su valor en el registro ax. El puntero de pila es decrementado en uno (módulo TAM_PILA).

No hay ninguna condición bajo la que esta instrucción falle.

POP_BX:

Esta instrucción desapila la cima de la pila y almacena su valor en el registro bx. El puntero de pila es decrementado en uno (módulo TAM_PILA).

No hay ninguna condición bajo la que esta instrucción falle.

POP_CX:

Esta instrucción desapila la cima de la pila y almacena su valor en el registro cx. El puntero de pila es decrementado en uno (módulo TAM_PILA).

No hay ninguna condición bajo la que esta instrucción falle.

POP_DX:

Esta instrucción desapila la cima de la pila y almacena su valor en el registro dx. El puntero de pila es decrementado en uno (módulo TAM_PILA).

[4] El tamaño de la pila viene dado por el parámetro del sistema TAM_PILA. Por defecto su valor es 10. [Ver apartado 3.8. Parámetros del sistema]

No hay ninguna condición bajo la que esta instrucción falle.

MOV_AB:

Esta instrucción copia el contenido del registro ax en el registro bx dejando intacto el registro ax.

No hay ninguna condición bajo la que esta instrucción falle.

MOV_CD:

Esta instrucción copia el contenido del registro cx en el registro dx dejando intacto el registro cx.

No hay ninguna condición bajo la que esta instrucción falle.

JMP:

Esta instrucción hace que el puntero de instrucción IP apunte a la última instrucción de la plantilla más cercana complementaria a la que sigue a la instrucción *jmp.*

La plantilla es un grupo de instrucciones de no-operación consecutivas (*nop_0* o *nop_1*) que siguen a la instrucción *jmp*, y su tamaño es el número de *nops* consecutivos que siguen a la instrucción. La plantilla a buscar puede ser una subplantilla de una plantilla de longitud mayor a la que buscamos. La búsqueda es bidireccional, es decir, se alternan la búsqueda hacia adelante y hacia atrás comenzando siempre por la búsqueda hacia adelante.

Para la búsqueda vamos a usar dos variables: a, que guardará la posición de la primera instrucción de la plantilla que hay después del *jmp*, y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma, la búsqueda hacia delante comienza en la dirección a + s + 1, y la búsqueda hacia atrás comienza en la dirección a - s - 1.

La búsqueda empieza hacia adelante, y alternativamente busca hacia delante y hacia atrás una posición más hasta que la plantilla complementaria es encontrada, o se sobrepasa el límite de búsqueda. La búsqueda de las plantillas se realiza circularmente alrededor de la sopa.

La instrucción falla si el tamaño de la plantilla es menor que MIN_TEMPL_SIZE[5] o si no se encuentra la plantilla dentro del LIMITE_BUSQ[6] . En caso de fallo, el puntero de instrucción apuntará a la última instrucción de la plantilla que va después del *jmp*, y se incrementará el número de fallos de la criatura.

Si la instrucción tiene éxito, se coloca el puntero de instrucción en la última instrucción de la plantilla encontrada.

[5] El tamaño mínimo de la plantilla viene determinado por el parámetro MIN_TEMPL_SIZE. Por defecto su valor es 1. [Ver apartado 3.8. Parámetros del sistema]

[6] El límite de búsqueda de una plantilla viene determinado por el parámetro LIMITE_BUSQ. Por defecto su valor es 300. [Ver apartado 3.8. Parámetros del sistema]

JMPB:

Esta instrucción hace que el puntero de instrucción IP apunte a la última instrucción de la plantilla más cercana hacia atrás complementaria a la que sigue a la instrucción *jmpb.*

La plantilla es un grupo de instrucciones de no-operación consecutivas (*nop_0* o *nop_1*) que siguen a la instrucción *jmpb*, y su tamaño es el número de *nops* consecutivos que siguen a la instrucción. La plantilla a buscar puede ser una subplantilla de una plantilla de longitud mayor a la que buscamos. La búsqueda es unidireccional hacia atrás.

Para la búsqueda vamos a usar dos variables: a, que guardará la posición de la primera instrucción de la plantilla que hay después del *jmpb*, y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma, la búsqueda hacia atrás comienza en la dirección a - s - 1.

La búsqueda sigue hasta que la plantilla complementaria es encontrada, o se sobrepasa el límite de búsqueda. La búsqueda de las plantillas se realiza circularmente alrededor de la sopa.

La instrucción falla si el tamaño de la plantilla es menor que MIN_TEMPL_SIZE o si no se encuentra la plantilla dentro del LIMITE_BUSQ. En caso de fallo, el puntero de instrucción apuntará a la última instrucción de la plantilla que va después del *jmpb*.

Si la instrucción tiene éxito, se coloca el puntero de instrucción en la última instrucción de la plantilla encontrada.

CALL:

Esta instrucción se comporta igual que la instrucción *jmp* con la diferencia de que la dirección de la última instrucción de la plantilla que sigue a la instrucción *call* se almacena en la pila. Después se realiza el salto. Esta instrucción es equivalente a una llamada a procedimiento.

Los casos de error son los que siguen. Si la plantilla complementaria no se encuentra dentro del LIMITE_BUSQ, el IP apuntará a la última instrucción de la plantilla que sigue a *call* y no se guardará ningún valor en la pila. Si la instrucción *call* va seguida de una plantilla de tamaño menor que MIN_TEMPL_SIZE, el IP apuntará a la última instrucción de esta plantilla.

RET:

Esta instrucción desapila una dirección de la cima de la pila y la almacena en el puntero de instrucción IP. El puntero de pila se decrementa en uno módulo TAM_PILA.

No hay ninguna condición bajo la que esta instrucción falle.

MOV_IAB:

Copia una instrucción de la sopa a otro lugar de la misma. La instrucción origen está en la dirección contenida en el registro bx, y la instrucción destino en la dirección contenida en el registro ax.

Esta instrucción puede fallar en las siguientes circunstancias:

- Si la dirección origen y destino son iguales.
- Si la dirección destino donde se va a escribir está protegida para escritura. Las zonas de memoria que no están protegidas para escritura para la célula que está en ejecución son la propia de la célula, y si tiene una hija de la que no haya hecho *divide*, también podrá escribir en la zona de memoria de la hija.
- Si ambas direcciones origen y destino están fuera de la memoria.

ADR:

Esta instrucción guarda en el registro ax la dirección de la última instrucción de la plantilla más cercana complementaria a la que sigue a la instrucción *adr.*

La plantilla es un grupo de instrucciones de no-operación consecutivas (*nop_0* o *nop_1*) que siguen a la instrucción *adr*, y su tamaño es el número de *nops* consecutivos que siguen a la instrucción. La plantilla a buscar puede ser una subplantilla de una plantilla de longitud mayor a la que buscamos. La búsqueda es bidireccional, es decir, se alternan la búsqueda hacia adelante y hacia atrás comenzando siempre por la búsqueda hacia adelante.

Para la búsqueda vamos a usar dos variables: a, que guardará la posición de la primera instrucción de la plantilla que hay después del *adr*, y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma, la búsqueda hacia delante comienza en la dirección a + s + 1, y la búsqueda hacia atrás comienza en la dirección a - s - 1.

La búsqueda empieza hacia delante, y alternativamente busca hacia delante y hacia atrás una posición más hasta que la plantilla complementaria es encontrada, o se sobrepasa el límite de búsqueda. La búsqueda de las plantillas se realiza circularmente alrededor de la sopa.

La instrucción falla si el tamaño de la plantilla es menor que MIN_TEMPL_SIZE o si no se encuentra la plantilla dentro del LIMITE_BUSQ. En caso de fallo, el puntero de instrucción apuntará a la última instrucción de la plantilla que va después del *adr*, y se incrementará el número de fallos de la criatura.

Si la instrucción tiene éxito, se coloca en ax la dirección de la última instrucción de la plantilla encontrada.

ADRB:

Esta instrucción guarda en el registro ax la dirección de la última instrucción de la plantilla más cercana hacia atrás complementaria a la que sigue a la instrucción *adrb.*

La plantilla es un grupo de instrucciones de no-operación consecutivas (*nop_0* o *nop_1*) que siguen a la instrucción *adrb*, y su tamaño es el número de *nops* consecutivos que

siguen a la instrucción. La plantilla a buscar puede ser una subplantilla de una plantilla de longitud mayor a la que buscamos. La búsqueda es unidireccional hacia atrás.

Para la búsqueda vamos a usar dos variables: a, que guardará la posición de la primera instrucción de la plantilla que hay después del *adrb*, y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma, la búsqueda hacia atrás comienza en la dirección a - s - 1.

La búsqueda sigue hasta que la plantilla complementaria es encontrada, o se sobrepasa el límite de búsqueda. La búsqueda de las plantillas se realiza circularmente alrededor de la sopa.

La instrucción falla si el tamaño de la plantilla es menor que MIN_TEMPL_SIZE o si no se encuentra la plantilla dentro del LIMITE_BUSQ. En caso de fallo, el puntero de instrucción apuntará a la última instrucción de la plantilla que va después del *adrb*, y se incrementará el número de fallos de la criatura.

Si la instrucción tiene éxito, se coloca en ax la dirección de la última instrucción de la plantilla encontrada.

ADRF:

Esta instrucción guarda en el registro ax la dirección de la última instrucción de la plantilla más cercana hacia delante complementaria a la que sigue a la instrucción *adrf.*

La plantilla es un grupo de instrucciones de no-operación consecutivas (*nop_0* o *nop_1*) que siguen a la instrucción *adrf*, y su tamaño es el número de *nops* consecutivos que siguen a la instrucción. La plantilla a buscar puede ser una subplantilla de una plantilla de longitud mayor a la que buscamos. La búsqueda es unidireccional hacia atrás.

Para la búsqueda vamos a usar dos variables: a, que guardará la posición de la primera instrucción de la plantilla que hay después del *adrf*, y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma, la búsqueda hacia delante comienza en la dirección a + s + 1.

La búsqueda sigue hasta que la plantilla complementaria es encontrada, o se sobrepasa el límite de búsqueda. La búsqueda de las plantillas se realiza circularmente alrededor de la sopa.

La instrucción falla si el tamaño de la plantilla es menor que MIN_TEMPL_SIZE o si no se encuentra la plantilla dentro del LIMITE_BUSQ. En caso de fallo, el puntero de instrucción apuntará a la última instrucción de la plantilla que va después del *adrb*, y se incrementará el número de fallos de la criatura.

Si la instrucción tiene éxito, se coloca en ax la dirección de la última instrucción de la plantilla encontrada.

MAL:

Esta instrucción solicita al sistema operativo un bloque de memoria en la sopa. La cantidad de espacio requerida viene dada por el valor del registro cx, y la dirección de inicio del bloque concedido se almacena en el registro ax. El espacio asignado pertenece a la criatura que se está ejecutando en ese momento.

El sistema operativo busca un bloque libre de memoria del tamaño requerido, y si no lo hubiera llamará al reaper para que se encargue de eliminar células de la memoria y dejar espacio libre, pero teniendo en cuenta que nunca se podrá matar a la célula que está en ejecución. El bloque de memoria se obtiene buscando el primer bloque libre de tamaño mayor o igual al solicitado empezando por la posición de la memoria donde nos habíamos quedado en la solicitud anterior. El bloque solicitado no dará la vuelta a la memoria, es decir, no puede ocupar las últimas y las primeras posiciones de memoria.

Esta instrucción puede fallar en las siguientes circunstancias:

- El espacio de memoria solicitado es menor o igual que cero.
- Si la criatura ya tiene otro bloque de memoria, es decir, la criatura hace dos llamadas a *mal* solicitando memoria sin llamar entre ambas a *divide.*

Si la instrucción se ejecuta con éxito se decrementa en uno el número de fallos de la criatura.

DIVIDE:

Esta instrucción hace que la célula madre pierda sus privilegios en el espacio que ocupa la célula hija, y que se introduzca la célula hija en las colas del reaper y del slicer.

La hija entra en el slicer justo delante de la madre, y la última en el reaper. De esta forma la hija va a ser la última célula en ser eliminada y la última en ejecutarse.

La instrucción falla si la madre no ha ejecutado ninguna instrucción *mal* antes de intentar el *divide*. En este caso, el número de fallos de la criatura se incrementa.

Si la instrucción se ejecuta con éxito se decrementa en uno el número de fallos de la criatura.

3.5. La Criatura Ancestral

Para realizar las ejecuciones del sistema es necesario que haya inicialmente en la sopa al menos una criatura con capacidad de reproducción. Con este fin se ha escrito una criatura de 80 instrucciones a la que denominaremos "ancestro" o "criatura ancestral"[7]. Su nombre de genotipo es 80aaa, el que le corresponde por ser la primera criatura de su tamaño. No se le ha asignado ninguna funcionalidad aparte de la habilidad de autorreplicación, y tampoco ningún diseño específico que sea potencialmente evolutivo.

[7] No es necesario que ésta sea la primera o única criatura en la sopa al iniciar una ejecución. Sin embargo, dada su sencillez y funcionalidad limitada, se ha usado en la mayoría de las pruebas del sistema.

La criatura ancestral se examina a sí misma en memoria para encontrar su principio y su final. Su comienzo está marcado por una plantilla de cuatro no-operaciones: 1111, y su final por la plantilla 1110. El ancestro localiza su inicio con cinco instrucciones: *adrb*, *nop_0*, *nop_0*, *nop_0*, *nop_0*. Esta serie de instrucciones hace que el sistema busque hacia atrás desde el *adrb* la plantilla complementaria a los cuatro *nop_0*, y que guarde la dirección de dicha plantilla complementaria en el registro ax de la cpu. Análogamente se localiza el final.

Una vez que conoce sus direcciones de inicio y final, la criatura las resta para calcular su tamaño, y solicita un bloque de memoria del mismo tamaño para su célula hija. Es entonces cuando llama al procedimiento de copia que copia el genoma entero de la célula madre en el espacio reservado para la célula hija, una instrucción cada vez. El inicio del procedimiento de copia está marcado con la plantilla 1100. Por tanto, la llamada al procedimiento se realiza con las siguientes cinco instrucciones: *call*, *nop_0, nop_0, nop_1, nop_1.*

Cuando el genoma ha sido copiado, el ancestro ejecuta la instrucción *divide*, que hace que pierda los privilegios de escritura sobre la célula hija y la introduzca en el slicer y el reaper. Después de esta primera reproducción la célula no se examina a sí misma de nuevo, sino que directamente pide espacio para una nueva hija y ejecuta el procedimiento de copia y la división celular en un ciclo sin fin.

Cuarenta y ocho de las ochenta instrucciones de la criatura ancestral son no-operaciones. Grupos de cuatro instrucciones *nop* se usan como plantillas complementarias para marcar doce sitios para el direccionamiento interno, de manera que la criatura pueda encontrar su principio y final, llamar al procedimiento de copia, y señalar direcciones de bucles y saltos en el código, entre otras.

El código de la criatura ancestral es el que sigue:

```
genotype: 0080aaa  parent genotype: 0666god
1st_daughter:  flags: 0  inst: 827  mov_daught: 80
2nd_daughter:  flags: 0  inst: 809  mov_daught: 80
comments: the ancestor, written by a human, mother of all other creatures.

nop1    ; 110 01   0 beginning marker
nop1    ; 110 01   1 beginning marker
nop1    ; 110 01   2 beginning marker
nop1    ; 110 01   3 beginning marker
zero    ; 110 04   4 put zero in cx
not0    ; 110 02   5 put 1 in first bit of cx
shl     ; 110 03   6 shift left cx (cx = 2)
shl     ; 110 03   7 shift left cx (cx = 4)
movDC   ; 110 18   8 move cx to dx (dx = 4)
adrb    ; 110 1c   9 get (backward) address of beginning marker -> ax
nop0    ; 100 00  10 complement to beginning marker
nop0    ; 100 00  11 complement to beginning marker
nop0    ; 100 00  12 complement to beginning marker
nop0    ; 100 00  13 complement to beginning marker
```

```
subAAC  ; 110 07  14 subtract cx from ax, result in ax
movBA   ; 110 19  15 move ax to bx,bx now contains start address of mother
adrf    ; 110 1d  16 get (forward) address of end marker -> ax
nop0    ; 100 00  17 complement to end marker
nop0    ; 100 00  18 complement to end marker
nop0    ; 100 00  19 complement to end marker
nop1    ; 100 01  20 complement to end marker
incA    ; 110 08  21 increment ax, to include dummy instruction at end
subCAB  ; 110 06  22 subtract bx from ax to get size, result in cx
nop1    ; 110 01  23 reproduction loop marker
nop1    ; 110 01  24 reproduction loop marker
nop0    ; 110 00  25 reproduction loop marker
nop1    ; 110 01  26 reproduction loop marker
mal     ; 110 1e  27 allocate space (cx) for daughter, address to ax
call    ; 110 16  28 call template below (copy procedure)
nop0    ; 100 00  29 copy procedure complement
nop0    ; 100 00  30 copy procedure complement
nop1    ; 100 01  31 copy procedure complement
nop1    ; 100 01  32 copy procedure complement
divide  ; 110 1f  33 create independent daughter cell
jmpo    ; 110 14  34 jump to template below (reproduction loop)
nop0    ; 100 00  35 reproduction loop complement
nop0    ; 100 00  36 reproduction loop complement
nop1    ; 100 01  37 reproduction loop complement
nop0    ; 100 00  38 reproduction loop complement
ifz     ; 000 05  39 dummy instruction to separate templates
nop1    ; 110 01  40 copy procedure template
nop1    ; 110 01  41 copy procedure template
nop0    ; 110 00  42 copy procedure template
nop0    ; 110 00  43 copy procedure template
pushA   ; 110 0c  44 push ax onto stack
pushB   ; 110 0d  45 push bx onto stack
pushC   ; 110 0e  46 push cx onto stack
nop1    ; 110 01  47 copy loop template
nop0    ; 110 00  48 copy loop template
nop1    ; 110 01  49 copy loop template
nop0    ; 110 00  50 copy loop template
movii   ; 110 1a  51 move contents of [bx] to [ax] (copy one instruction)
decC    ; 110 0a  52 decrement cx (size)
ifz     ; 110 05  53 if cx==0 perform next instruction,otherwise skip it
jmpo    ; 110 14  54 jump to template below (copy procedure exit)
nop0    ; 110 00  55 copy procedure exit complement
nop1    ; 110 01  56 copy procedure exit complement
nop0    ; 110 00  57 copy procedure exit complement
nop0    ; 110 00  58 copy procedure exit complement
incA    ; 110 08  59 increment ax (address in daughter to copy to)
```

```
incB    ; 110 09  60 increment bx (address in mother to copy from)
jmpo    ; 110 14  61 bidirectional jump to template below (copy loop)
nop0    ; 100 00  62 copy loop complement
nop1    ; 100 01  63 copy loop complement
nop0    ; 100 00  64 copy loop complement
nop1    ; 100 01  65 copy loop complement
ifz     ; 000 05  66 this is a dummy instruction to separate templates
nop1    ; 110 01  67 copy procedure exit template
nop0    ; 110 00  68 copy procedure exit template
nop1    ; 110 01  69 copy procedure exit template
nop1    ; 110 01  70 copy procedure exit template
popC    ; 110 12  71 pop cx off stack (size)
popB    ; 110 11  72 pop bx off stack (start address of mother)
popA    ; 110 10  73 pop ax off stack (start address of daughter)
ret     ; 110 17  74 return from copy procedure
nop1    ; 100 01  75 end template
nop1    ; 100 01  76 end template
nop1    ; 100 01  77 end template
nop0    ; 100 00  78 end template
ifz     ; 000 05  79 dummy instruction to separate creature
```

3.6. MUTACIONES

Para que sea posible la evolución, debe haber cambios en el genoma de las criaturas. Éstos deben ocurrir durante la vida útil de un individuo, o habrá errores al copiar el genoma a la descendencia. Para asegurar que haya cambios genéticos, el sistema operativo de ALiS cambia, bajo ciertas condiciones, unas instrucciones por otras, y el comportamiento de las instrucciones es ejecutado incorrectamente. En ALiS hay cuatro tipos de mutaciones: de fondo, de copia, de plantilla y de comportamiento.

En la mutación de fondo se cambian instrucciones de la sopa por otras aleatoriamente. Esta mutación es análoga a las mutaciones causadas por los rayos cósmicos, y tiene el efecto de prevenir que una criatura sea inmortal, ya que en algún momento puede mutar de forma que su comportamiento cause errores y muera al llegar a la cima del reaper. La frecuencia para la mutación de fondo ha sido escogida de una instrucción mutada cada MUT_FONDO instrucciones ejecutadas por el sistema[8].

Además, cuando se están copiando instrucciones de las células madre a las hijas, puede haber mutaciones de copia. En esta mutación la instrucción que se iba a copiar muta, copiándose en la hija una instrucción escogida aleatoriamente del conjunto de instrucciones. De esta forma da lugar a errores en la reproducción. La frecuencia de la mutación de copia ha

[8] La proporción de mutación de fondo viene dada por el parámetro del sistema MUT_FONDO. Por defecto su valor es 10.000. [Ver apartado 3.8. Parámetros del sistema]

sido escogida de una instrucción mutada cada MUT_COPIA instrucciones copiadas en el sistema[9].

También durante la reproducción se produce el tercer tipo de mutación, la de plantilla. Cada MUT_PLANT instrucciones *divide* que se ejecuten satisfactoriamente, se producirá una mutación en una de las plantillas de la criatura hija, de forma que un *nop_0* cambiará a un *nop_1*, y viceversa[10].

Finalmente, la ejecución de las instrucciones de LALiS no es determinista. Para la mayoría de ellas, el comportamiento no es el esperado con una pequeña frecuencia. Es lo que se denomina mutación de comportamiento, que se da en una proporción por defecto del MUT_COMP tanto por ciento de los casos[11]. A continuación se detallan las mutaciones de comportamiento de las instrucciones del sistema que se comportan de esta forma.

OR1:

Por defecto esta instrucción cambia el bit de menor orden del registro cx. En el caso de que haya mutación, el bit no se cambia y cx permanece intacto.

SHL:

Por defecto esta instrucción desplaza todos los bits del registro cx una posición hacia la izquierda, poniendo un 0 en el bit menos significativo del registro. Si hay mutación, el contenido de cx no se desplaza.

ZERO:

Por defecto esta instrucción pone a 0 el registro cx. Si hay mutación, el valor de cx no cambia.

IF_CZ:

Por defecto esta instrucción sólo incrementa el puntero de instrucción en 1 si el valor del registro cx es distinto de 0. Esto significa que la instrucción siguiente a la actual será ejecutada sólo si el valor del registro cx es 0. Si hay mutación, en el caso de que el valor de cx sea distinto de 0, el puntero de instrucción no se incrementa, con lo que se ejecutará la instrucción siguiente a la actual.

INC_A:

Por defecto esta instrucción incrementa en uno el valor del registro ax, volviendo a almacenar el resultado en ax. Si hay mutación, el valor de ax no se incrementa.

---

[9] La proporción de mutación de copia viene dada por el parámetro del sistema MUT_COPIA. Por defecto su valor es 1.000. [Ver apartado 3.8. Parámetros del sistema]

[10] La proporción de mutación de plantilla viene dada por el parámetro del sistema MUT_PLANT. Por defecto su valor es 2. [Ver apartado 3.8. Parámetros del sistema]

[11] La proporción de mutación de comportamiento viene dada por el parámetro del sistema MUT_COMP. Por defecto su valor es 1. [Ver apartado 3.8. Parámetros del sistema]

INC_B:

Por defecto esta instrucción incrementa en uno el valor del registro bx, volviendo a almacenar el resultado en bx. Si hay mutación, el valor de bx no se incrementa.

INC_C:

Por defecto esta instrucción incrementa en uno el valor del registro cx, volviendo a almacenar el resultado en cx. Si hay mutación, el valor de cx no se incrementa.

DEC_C:

Por defecto esta instrucción decrementa en uno el valor del registro cx, volviendo a almacenar el resultado en cx. Si hay mutación, el valor de cx no se decrementa.

PUSH_AX:

Por defecto esta instrucción apila el valor que contiene el registro ax en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA). Si hay mutación, el valor de ax no se apila.

PUSH_BX:

Por defecto esta instrucción apila el valor que contiene el registro bx en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA). Si hay mutación, el valor de bx no se apila.

PUSH_CX:

Por defecto esta instrucción apila el valor que contiene el registro cx en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA). Si hay mutación, el valor de no se apila.

PUSH_DX:

Por defecto esta instrucción apila el valor que contiene el registro dx en la pila, e incrementa el puntero de pila en uno (módulo TAM_PILA). Si hay mutación, el valor de no se apila.

3.7. EL BANCO DE GENES

La introducción de las mutaciones en el sistema provoca que aparezcan en la sopa criaturas diferentes en tamaño y/o lista de instrucciones que las componen (genoma de la criatura). Para identificar las criaturas se les dará un nombre único en la ejecución (un genotipo). El sistema necesitará guardar toda la información de los genotipos y genomas de las criaturas, y lo hará en el banco de genes.

Cuando aparece una criatura con un genotipo nuevo, a la criatura se le da un nombre único y se guarda su genoma, junto con otros datos, en el banco de genes. Cada nombre de genotipo tiene dos partes, un número y un código de tres letras. El número representa el número de instrucciones del genoma de la criatura. El código de tres letras es usado como un sistema de numeración en base 26 para asignar una etiqueta única a cada genotipo de un tamaño determinado. Al primer genotipo que aparezca de un tamaño concreto se le asigna la etiqueta aaa, al segundo la aab, y así sucesivamente. De esta forma, la criatura ancestral se nombra con 80aaa, y el primer mutante de tamaño 80 será 80aab. La primera criatura de tamaño 45 será 45aaa.

El banco de genes mantendrá información actualizada de la presencia de genotipos en cada instante de ejecución del sistema. Mantendrá datos de los tamaños de criaturas existentes, genotipos de cada tamaño y número de criaturas de cada genotipo. La información se actualizará cada vez que en el sistema nazca o muera alguna criatura. Esta información se podrá resumir de forma que sea fácil la visualización de la misma, ya sea en formato textual o como diagrama (método empleado en la interfaz gráfica que hemos aplicado a ALiS).

Aparte de mantener la información actualizada de genotipos presentes en el sistema, el banco de genes también realiza el seguimiento de los nacimientos y muertes actualizando la información sobre ellos en un fichero de texto: RegBirthDeath.txt que servirá posteriormente para el análisis de la evolución del sistema [12].

Hay que tener presente que dado que el comportamiento del sistema es completamente probabilístico, el nombre asignado a un genotipo en una ejecución no tiene por qué coincidir con el mismo genotipo en otra, ya que en cada ejecución las criaturas evolucionarán de forma distinta, y no siempre aparecerán los mismos genomas en el mismo orden. A pesar de este hecho, disponemos de un método para asignar siempre el mismo nombre (genotipo) a una criatura con una determinada secuencia de instrucciones (genoma). El banco de genes irá generando a lo largo de la ejecución unos ficheros con extensión .gen, un fichero distinto por cada tamaño de criatura: 80.gen, 14.gen,.... donde se van recogiendo aquellas criaturas más abundantes en la ejecución del sistema, existiendo un parámetro (LIM_SAVE_DISK[13]) que impone el límite de ocurrencia. Así, los genotipos de las criaturas cuya presencia en la sopa en algún instante superen el valor de LIM_SAVE_DISK, serán guardados en el correspondiente fichero .gen. Cuando aparecen nuevas criaturas en una ejecución, no presentes en la sopa, antes de asignarles un nuevo nombre (genotipo) se consultarán estos ficheros .gen para ver si su genoma ya tenía un genotipo asignado. Conservando estos ficheros .gen de una ejecución a otra obtendremos que para criaturas con el mismo genoma les asignaremos un nombre igual.

[12] Existe una aplicación para análisis de los resultados [Ver apartado 6. Analizador Semántico de Alto Nivel].

[13] Cuando existan mayor número de criaturas que las especificadas por el parámetro LIM_SAVE_DISK, se guardará en los ficheros .gen el genoma y genotipo de esa criatura. Por defecto, el valor del parámetro es 2. [Ver apartado 3.8. Parámetros del sistema]

3.8. PARÁMETROS DEL SISTEMA

El sistema ALiS es un sistema de simulación de vida artificial sujeto a numerosos parámetros, que pueden producir formas muy distintas de evolución según sus valores.

3.8.1. PARÁMETROS DE LA CPU

Son los parámetros que configuran la cpu que tiene cada una de las criaturas. Son los siguientes:

TAM_PILA:

Valor que indica la capacidad de la pila de la cpu. Su valor por defecto es 10.

3.8.2. PARÁMETROS DE LA MEMORIA

Son los parámetros que configuran la memoria o sopa donde se encuentran las criaturas. Tratan tanto las dimensiones de la memoria como otros parámetros necesarios para las búsquedas. Son los siguientes:

TAM_MEM:

Valor que indica la capacidad de la memoria, las posiciones válidas que contiene. Su valor por defecto es 60.000.

LIMITE_BUSQ:

Valor que indica el límite de búsqueda de plantillas complementarias. Sólo se busca dentro de este rango de instrucciones alrededor de la instrucción que solicita la búsqueda. Su valor por defecto es 300.

PROP_MEM_LIBRE:

Valor que indica la proporción de memoria que se debe mantener libre en todo momento. Si el espacio libre es inferior a este porcentaje, el sistema usará el reaper para eliminar las criaturas que corresponda. Su valor por defecto es 0.2, lo que significa que se puede ocupar como máximo un 80% de la memoria.

MIN_TEMPL_SIZE:

Valor que indica el tamaño mínimo de plantilla válida en número de instrucciones no-operación. Su valor por defecto es 1.

MAX_TEMPL_SIZE:

Valor que indica el tamaño máximo de plantilla válida en número de instrucciones no-operación. Su valor por defecto es 32, el máximo que puede alcanzar.

3.8.3. Parámetros del Sistema Operativo

Son los parámetros que configuran las distintas opciones del sistema operativo, desde parámetros de tiempo hasta parámetros sobre el tamaño de las criaturas. Son los siguientes:

TIEMPO_VIDA:

Valor que indica el número de instrucciones totales que ejecutará el sistema en una ejecución. No hay valor por defecto, ya que generalmente se querrá que el sistema ejecute ininterrumpidamente. Este parámetro es útil durante las pruebas del sistema.

VIDA_CELULA:

Valor que indica el número de instrucciones que ejecuta cada criatura en su turno de ejecución. Su valor por defecto es 100.

NUM_INST:

Valor que indica el número de instrucciones que forman el conjunto de instrucciones que se está utilizando. Su valor por defecto es 32.

TAM_MIN_CRIA:

Valor que indica el tamaño mínimo de criatura, en número de instrucciones. Su valor por defecto es 12.

TAM_MAX_CRIA:

Valor que indica el tamaño máximo de criatura, en número de instrucciones. Su valor por defecto es 100.

PROP_MOV:

Valor que indica el porcentaje mínimo de una criatura que debe haber sido rellenado por la madre antes de realizar el divide correspondiente.

FACTOR_SLICE:

Valor que indica el factor de influencia del tamaño de la criatura a la hora de asignarle un fragmento de tiempo de cpu. Se sigue la siguiente regla:

$$time\_slice = VIDA\_CELULA * tamaño^{FACTOR\_SLICE}$$

donde los distintos parámetros tienen el siguiente significado:

- `time_slice`. Número de instrucciones que se asignarán a la criatura en el fragmento de tiempo.
- `VIDA_CELULA`. Parámetro del sistema que indica el número de instrucciones que se asignan a una criatura en el caso neutral.
- `tamaño`. Tamaño de la criatura que se va a ejecutar.
- `FACTOR_SLICE`. Parámetro del sistema introducido en este apartado.

3.8.4. PARÁMETROS DE LAS MUTACIONES

Son los parámetros que configuran las distintas opciones de las mutaciones, desde si se tienen que realizar hasta la probabilidad de las mismas. Son los siguientes:

MUT_FONDO:

Valor que indica la frecuencia de la mutación de fondo. Su valor por defecto es 10.000, lo que quiere decir que en promedio una de cada 10.000 instrucciones ejecutadas mutará.

MUT_COPIA:

Valor que indica la frecuencia de la mutación de copia. Su valor por defecto es 1.000, lo que quiere decir que en promedio una de cada 1.000 instrucciones copiadas mutará.

MUT_COMP:

Valor que indica la proporción de mutación de comportamiento. Su valor por defecto es 0.5, lo que quiere decir que en el 0.5% de las ocasiones la instrucción que se esté ejecutando lo hará incorrectamente.

MUT_PLANT:

Valor que indica la frecuencia de mutación de plantilla. Su valor por defecto es 2, lo que quiere decir que cada 2 divides ejecutados satisfactoriamente se produce la mutación de plantilla en la criatura recién nacida.

HAY_MUT_FONDO:

Valor que indica si la mutación de fondo está activa o no. Un valor de 1 significa que lo está, y un valor de 0 que no.

HAY_MUT_COPIA:

Valor que indica si la mutación de copia está activa o no. Un valor de 1 significa que lo está, y un valor de 0 que no.

HAY_MUT_COMP:

Valor que indica si la mutación de comportamiento está activa o no. Un valor de 1 significa que lo está, y un valor de 0 que no.

HAY_MUT_PLANT:

Valor que indica si la mutación de plantilla está activa o no. Un valor de 1 significa que lo está, y un valor de 0 que no.

3.8.5. Parámetros del Banco de Genes

Son los parámetros que configuran las distintas opciones del banco de genes. Son los siguientes:

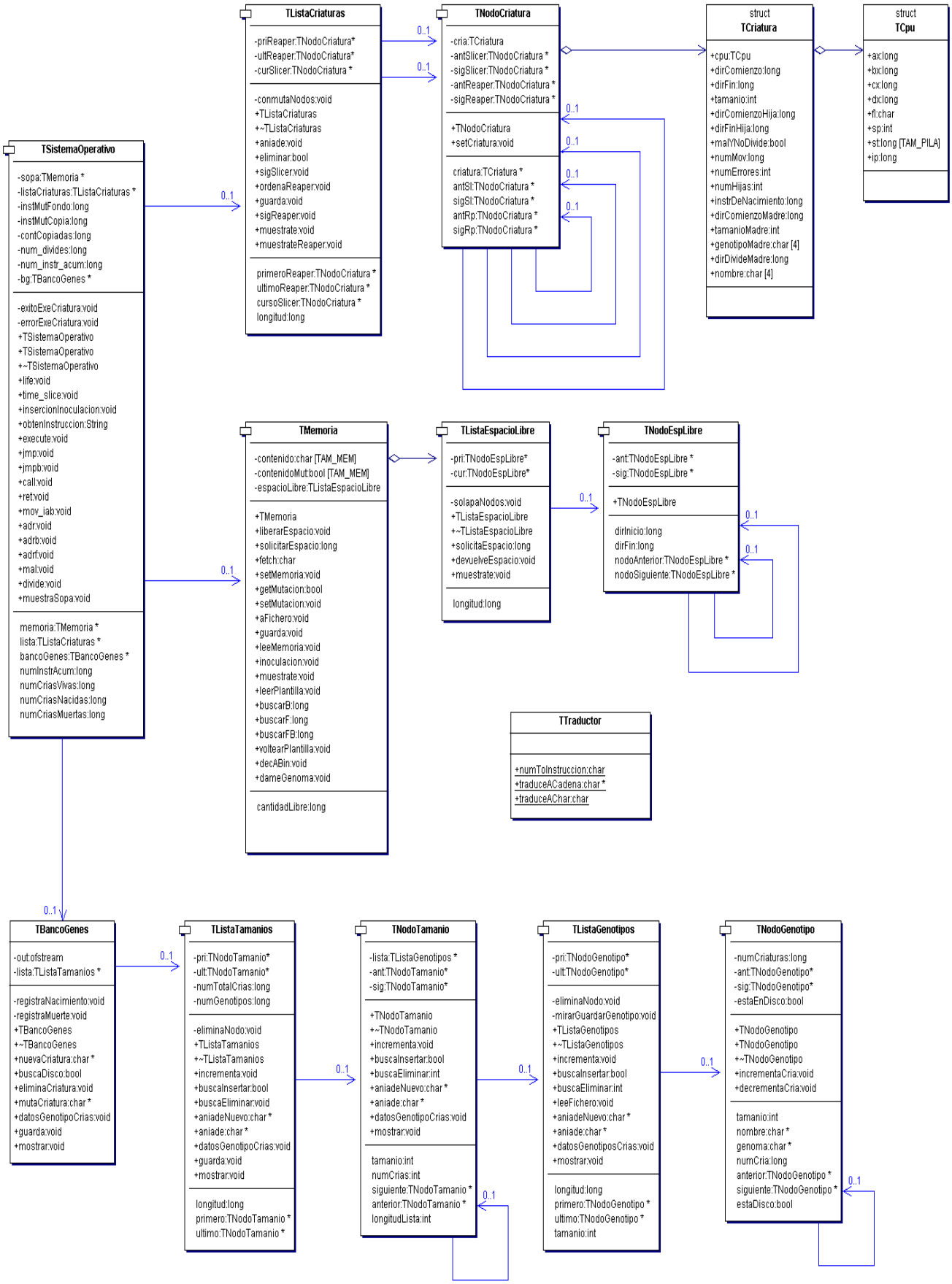
LIM_SAVE_DISK:

Valor que indica el número de criaturas que debe haber de un genotipo para que éste se guarde en fichero. Su valor por defecto es 2. Se ha escogido este valor ya que en las ejecuciones del sistema aparecen muchos genotipos de los que solamente existe una criatura. Estas criaturas no se consideran casos de estudio ya que si no aparecen más del mismo genotipo es porque son resultado de una mutación en su madre y ellas no son funcionalmente correctas ni capaces de reproducir su mismo genoma.

4.Arquitectura del Sistema ALiS

Hemos implementado el sistema ALiS empleando el lenguaje de programación C++. Decimos emplear este lenguaje para obtener, por un lado, un diseño modular que nos permitiera organizar el código en clases y módulos, y por otro, conseguir tiempos de ejecución rápidos, característica primordial en sistemas de simulación como éste.

Partiendo de la descripción de computador virtual dada en Tierra construimos el diseño de las clases y estructuras que podemos ver en el siguiente diagrama UML:


TListaCriaturas
-priReaper:TNodoCriatura*
-ultReaper:TNodoCriatura*
-curSlicer:TNodoCriatura *
-conmutaNodos:void
+TListaCriaturas
+~TListaCriaturas
+aniade:void
+eliminar:bool
+sigSlicer:void
+ordenaReaper:void
+guarda:void
+sigReaper:void
+muestrate:void
+muestrateReaper:void
primeroReaper:TNodoCriatura *
ultimoReaper:TNodoCriatura *
cursoSlicer:TNodoCriatura *
longitud:long
TNodoCriatura
-cria:TCriatura
-antSlicer:TNodoCriatura *
-sigSlicer:TNodoCriatura *
-antReaper:TNodoCriatura *
-sigReaper:TNodoCriatura *
+TNodoCriatura
+setCriatura:void
criatura:TCriatura *
antSl:TNodoCriatura *
sigSl:TNodoCriatura *
antRp:TNodoCriatura *
sigRp:TNodoCriatura *
struct
TCriatura
+cpu:TCpu
+dirComienzo:long
+dirFin:long
+tamanio:int
+dirComienzoHija:long
+dirFinHija:long
+malYNoDivide:bool
+numMov:long
+numErrores:int
+numHijas:int
+instrDeNacimiento:long
+dirComienzoMadre:long
+tamanioMadre:int
+genotipoMadre:char [4]
+dirDivideMadre:long
+nombre:char [4]
struct
TCpu
+ax:long
+bx:long
+cx:long
+dx:long
+fl:char
+sp:int
+st:long [TAM_PILA]
+ip:long
TSistemaOperativo
-sopa:TMemoria *
-listaCriaturas:TListaCriaturas *
-instMutFondo:long
-instMutCopia:long
-contCopiadas:long
-num_divides:long
-num_instr_acum:long
-bg:TBancoGenes *
-exitoExeCriatura:void
-errorExeCriatura:void
+TSistemaOperativo
+TSistemaOperativo
+~TSistemaOperativo
+life:void
+time_slice:void
+insercionInoculacion:void
+obtenInstruccion:String
+execute:void
+jmp:void
+jmpb:void
+call:void
+ret:void
+mov_iab:void
+adr:void
+adrb:void
+adrf:void
+mal:void
+divide:void
+muestraSopa:void
memoria:TMemoria *
lista:TListaCriaturas *
bancoGenes:TBancoGenes *
numInstrAcum:long
numCriasVivas:long
numCriasNacidas:long
numCriasMuertas:long
TMemoria
-contenido:char [TAM_MEM]
-contenidoMut:bool [TAM_MEM]
-espacioLibre:TListaEspacioLibre
+TMemoria
+liberarEspacio:void
+solicitarEspacio:long
+fetch:char
+setMemoria:void
+getMutacion:bool
+setMutacion:void
+aFichero:void
+guarda:void
+leeMemoria:void
+inoculacion:void
+muestrate:void
+leerPlantilla:void
+buscarB:long
+buscarF:long
+buscarFB:long
+voltearPlantilla:void
+decABin:void
+dameGenoma:void
cantidadLibre:long
TListaEspacioLibre
-pri:TNodoEspLibre*
-cur:TNodoEspLibre*
-solapaNodos:void
+TListaEspacioLibre
+~TListaEspacioLibre
+solicitaEspacio:long
+devuelveEspacio:void
+muestrate:void
longitud:long
TNodoEspLibre
-ant:TNodoEspLibre *
-sig:TNodoEspLibre *
+TNodoEspLibre
dirInicio:long
dirFin:long
nodoAnterior:TNodoEspLibre *
nodoSiguiente:TNodoEspLibre *
TTraductor
+numToInstruccion:char
+traduceACadena:char *
+traduceAChar:char
TBancoGenes
-out:ofstream
-lista:TListaTamanios *
-registraNacimiento:void
-registraMuerte:void
+TBancoGenes
+~TBancoGenes
+nuevaCriatura:char *
+buscaDisco:bool
+eliminaCriatura:void
+mutaCriatura:char *
+datosGenotipoCrias:void
+guarda:void
+mostrar:void
TListaTamanios
-pri:TNodoTamanio*
-ult:TNodoTamanio*
-numTotalCrias:long
-numGenotipos:long
-eliminaNodo:void
+TListaTamanios
+~TListaTamanios
+incrementa:void
+buscaInsertar:bool
+buscaEliminar:void
+aniadeNuevo:char *
+aniade:char *
+datosGenotipoCrias:void
+guarda:void
+mostrar:void
longitud:long
primero:TNodoTamanio *
ultimo:TNodoTamanio *
TNodoTamanio
-lista:TListaGenotipos *
-ant:TNodoTamanio*
-sig:TNodoTamanio*
+TNodoTamanio
+~TNodoTamanio
+incrementa:void
+buscaInsertar:bool
+buscaEliminar:int
+aniadeNuevo:char *
+aniade:char *
+datosGenotipoCrias:void
+mostrar:void
tamanio:int
numCrias:int
siguiente:TNodoTamanio *
anterior:TNodoTamanio *
longitudLista:int
TListaGenotipos
-pri:TNodoGenotipo*
-ult:TNodoGenotipo*
-eliminaNodo:void
-mirarGuardarGenotipo:void
+TListaGenotipos
+~TListaGenotipos
+incrementa:void
+buscaInsertar:bool
+buscaEliminar:int
+leeFichero:void
+aniadeNuevo:char *
+aniade:char *
+datosGenotiposCrias:void
+mostrar:void
longitud:long
primero:TNodoGenotipo *
ultimo:TNodoGenotipo *
tamanio:int
TNodoGenotipo
-numCriaturas:long
-ant:TNodoGenotipo*
-sig:TNodoGenotipo*
-estaEnDisco:bool
+TNodoGenotipo
+TNodoGenotipo
+~TNodoGenotipo
+incrementaCria:void
+decrementaCria:void
tamanio:int
nombre:char *
genoma:char *
numCria:long
anterior:TNodoGenotipo *
siguiente:TNodoGenotipo *
estaDisco:bool
0..1

Como podemos observar, las clases implementadas son claras componentes de un computador, en este caso, del computador virtual que hemos descrito anteriormente[14]: una memoria, una lista de criaturas (programas), un sistema operativo.

Algunas observaciones respecto al diseño son las siguientes:

- La clase principal del sistema es TSistemaOperativo, donde se encuentra el método *life*[15] que será el que simule todo el funcionamiento del sistema.
- Hemos implementado nuestras propias listas enlazadas ya que las características que necesitábamos eran muy particulares. Destacamos la lista de criaturas (TListaCriaturas) la cual es una lista doblemente enlazada. Así tenemos representadas en la misma estructura de nodos las dos listas necesarias para el sistema: el reaper y el slicer. También, por otro lado, al no emplear plantillas C++ obtenemos mejor rendimiento en la ejecución.
- Para determinadas estructuras del sistema, como la cpu y la criatura (TCpu y TCriatura) hemos empleado tipos *struct* en lugar de clases C++. El empleo de este tipo de datos mejora notablemente el tiempo de ejecución de nuestro programa. Nuestras primeras implementaciones estaban construidas con un tipo de criatura (TCriatura) construido como clase. Tras cambiarlo a una estructura (struct) obtuvimos una mejora en tiempo de ejecución de 1 minuto cada millón de instrucciones del sistema ejecutadas.
- El conjunto de instrucciones del lenguaje ensamblador descrito [16] se ha dividido en dos subconjuntos cada uno de los cuales ha sido implementado en un módulo:
    - Instrucciones aritméticas y operaciones sobre registros (*sub_ac, zero*,...): han sido implementadas para el tipo TCpu ya que operan directamente con estructuras de la cpu de cada criatura.
    - Instrucciones de salto (*jmp,adrb*,...), reserva de memoria (*mal*) y creación de hija independiente (*divide*): han sido implementadas como métodos de la clase TSistemaOperativo. El motivo de recoger su código en esta clase es que estas operaciones necesitan operar sobre todas las estructuras del computador virtual ALiS, referenciadas en el sistema operativo. Así, los saltos necesitan acceso a la memoria, la reserva de espacio en memoria requiere acceso a la misma y a la lista de criaturas (cuando se necesita eliminar criaturas para obtener el espacio solicitado) y la obtención de una criatura hija necesita modificar todas las estructuras para considerar la nueva criatura en el sistema.

En los siguientes puntos detallamos la implementación de cada una de las clases y estructuras de la aplicación.

[14] [Ver apartado 3.2. El Computador Virtual]
[15] [Ver apartado 4.1. TSistemaOperativo]
[16] [Ver apartado 3.4.1. Conjunto de Instrucciones]

4.1. TSISTEMAOPERATIVO

El sistema operativo determina los mecanismos de comunicación entre los módulos del sistema: la gestión de la memoria, la gestión del tiempo de cpu en el que se ejecutan las criaturas, la mortalidad de las criaturas, el registro de las criaturas nacidas y muertas y otras características como las mutaciones.

Por ello la clase que lo representa tiene como atributos:

-La memoria, que es un puntero a TMemoria para la solicitud de espacio, carga de datos, etc.

-La lista de criaturas, que es un puntero a TListaCriaturas, para la gestión de las colas, para el reparto de tiempo de cpu (slicer) y mortalidad de las criaturas (reaper)

-El banco de genes, que es un puntero a TBancoGenes, para registrar las criaturas del sistema.

-Un contador para llevar la cuenta del número de instrucciones de copia *mov_ia* en todo el sistema (*contCopiadas*) y una variable que señalará el número de instrucción de copia en la que hay que mutar (*instMutCopia*) para poder realizar la mutación de copia.

-Una variable que nos señalará dónde se aplica la mutación de fondo (*instMutFondo*)

-Un contador (*num_divides*) que lleva la cuenta del número de instrucciones *divide* en todo el sistema para poder realizar la mutación de plantilla .

-Un contador (*num_instr_acum*) para llevar la suma total de instrucciones ejecutadas en el sistema.

Tendremos también unas funciones auxiliares *exitoExeCriatura* y *errorExeCriatura*, para el uso de todas las funciones que ejecutan instrucciones del sistema que, según se ejecuten correcta o incorrectamente, deben modificar el número de errores de la criatura. Así, estas funciones se encargan de decrementar e incrementar respectivamente el número de fallos de las criaturas.

LIFE:

En este procedimiento se ejecuta un bucle en el que se da un determinado tiempo a cada criatura para ejecutarse. Este tiempo se mide en número de instrucciones que ejecuta la criatura en curso[17].

Tras dar el tiempo de ejecución a la criatura se reorganiza el reaper según el número de errores que haya acumulado en dicha ejecución. Una vez reordenado el reaper seleccionamos la siguiente criatura a la que le toca ejecutarse. Ésta será la primera en el orden de la cola del slicer.

Por último, nos encargaremos de que en la sopa quede siempre un espacio libre correspondiente a un tanto por ciento, indicado mediante el parámetro PROP_MEM_LIBRE[18],

[17] Se determinará según la fórmula en el apartado 3.8.3. Parámetros del Sistema Operativo, FACTOR_SLICE
[18] [Ver el apartado 3.8. Parámetros del sistema]

del tamaño total de la memoria. Para ello eliminamos las criaturas según el orden del reaper. Si al eliminar una criatura, ésta estaba teniendo una hija, es decir, había ejecutado un *mal*, pero no había llegado a ejecutar un *divide*, también eliminaremos el espacio correspondiente de la hija.

El bucle principal terminará de ejecutarse cuando el número de instrucciones ejecutadas sea igual al parámetro TIEMPO_VIDA[19] o cuando haya un error en la vida del sistema. Esto último ocurrirá cuando no se pueda liberar en la memoria la proporción de espacio necesaria para el sistema indicada por parámetro PROP_MEM_LIBRE.

TIME_SLICE:

En este procedimiento se realiza un bucle que se interrumpirá una vez que la criatura correspondiente haya ejecutado un número de instrucciones igual al que se le ha asignado para su ejecución.

En cada vuelta del bucle se buscará en la sopa la instrucción apuntada por el puntero de instrucción de la cpu de la criatura y la mandaremos ejecutar. Después de ejecutarla incrementaremos el puntero de instrucción en uno para que apunte a la siguiente instrucción del código de la criatura.

Aquí también se hace la mutación de fondo ya que se lleva la cuenta del número de instrucciones ejecutadas y se calcula la instrucción que va a ser reemplazada en la sopa según la mutación de fondo. Si el número de instrucciones ejecutadas es igual a la instrucción a mutar se llevará a cabo la sustitución de esta instrucción por otra instrucción del conjunto de instrucciones.

Para la mutación de copia, se lleva la cuenta del número de instrucciones de copia ejecutadas y se calcula la instrucción de copia que debe ser mutada. En el destino de copia se introducirá una instrucción aleatoria del conjunto de instrucciones.

Por otra parte se lleva la cuenta del número de *divide* ejecutados para poder realizar la mutación de plantilla. Así cada vez que la instrucción a ejecutar sea un *divide* se incrementará un contador que después usará la función encargada de llevar a cabo esta mutación.

EXECUTE:

Este procedimiento se encarga de llamar al procedimiento correspondiente que ejecute la instrucción que recibe por parámetro.

JMP:

Este procedimiento es el que ejecuta la instrucción *jm*p. Consiste en leer la plantilla que va justo después del *jmp* y buscar tanto hacia adelante como hacia atrás la plantilla complementaria a la leída. En caso de encontrarse se pondrá el puntero de instrucción de la criatura que ejecuta esta instrucción apuntando a la última instrucción de la plantilla

[19] [Ver el apartado 3.8. Parámetros del sistema]

complementaria. La plantilla complementaria tendrá el mismo tamaño que la leída. A continuación vamos a detallar los pasos que se llevan a cabo.

Inicialmente, se llama a la función *leerPlantilla*[20] pasándole como parámetro la dirección de comienzo de la plantilla. Para ello se le suma uno a la dirección actual del puntero de instrucción que está apuntando a la instrucción *jmp*. Esta función devolverá un número decimal correspondiente a la plantilla complementaria a la leída, el tamaño de la plantilla y una variable booleana indicando si ha habido error en la lectura . Habrá error en los casos en que no haya una plantilla después del *jmp* o si la plantilla que hubiera es menor que el parámetro MIN_TEMPL_SIZE[21].

Si no ha habido error en la lectura se procede a la búsqueda de la plantilla complementaria. Para ello se llama a la función *buscarFB*[22] pasándole como parámetros la dirección de la primera instrucción de la plantilla que está a continuación de la instrucción *jmp*, el número decimal correspondiente a la plantilla complementaria a la leída después del *jmp* y el tamaño de la misma. La función devolverá una variable booleana indicando si la plantilla complementaria se ha encontrado o no. En caso de que se haya encontrado también devolverá la dirección de la última instrucción de la plantilla encontrada y actualizaremos el puntero de instrucción de la criatura a dicha posición. Si no se ha encontrado lo que devuelve como dirección no nos interesa.

Si no se ha encontrado la plantilla complementaria se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno y actualizaremos el puntero de instrucción de la criatura a la posición de la última instrucción de la plantilla que sigue al *jmp*. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si se ha encontrado la plantilla complementaria se actualizará el puntero de instrucción de la criatura para que apunte a la última instrucción de la plantilla complementaria encontrada. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si ha habido error en la lectura de la plantilla se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno. A continuación actualizaremos el puntero de instrucción de la criatura sumándole el tamaño de la plantilla que sigue al *jmp*. Si esta plantilla no existía su tamaño será cero, con lo cual al sumárselo al puntero de instrucción, éste seguirá apuntando a la misma posición, es decir, al *jmp*. Si esta plantilla era menor que el mínimo tamaño requerido para las plantillas se le sumará al puntero de instrucción de tal manera que quede apuntando a la última posición de la plantilla. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en

[20] [Ver apartado 4.2. TMemoria]
[21] [Ver apartado 3.8. Parámetros del Sistema]
[22] [Ver apartado 4.2. TMemoria]

uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

JMPB:

Este procedimiento es el que ejecuta la instrucción *jmpb*. Consiste en leer la plantilla que va justo después del *jmpb* y buscar hacia atrás la plantilla complementaria a la leída. A continuación vamos a detallar los pasos que se siguen.

Inicialmente, se llama a la función *leerPlantilla*[23] pasándole como parámetro la dirección de comienzo de la plantilla. Para ello se le suma uno a la dirección actual del puntero de instrucción que está apuntando a la instrucción *jmpb*. Esta función devolverá un número decimal correspondiente a la plantilla complementaria a la leída, el tamaño de la plantilla y una variable booleana indicando si ha habido error en la lectura. Habrá error en los casos en que no haya una plantilla después del *jmpb* o si la plantilla que hubiera es menor que el parámetro MIN_TEMPL_SIZE[24].

Si no ha habido error en la lectura se procede a la búsqueda de la plantilla complementaria. Para ello se llama a la función *buscarB*[25] pasándole como parámetros la dirección de la primera instrucción de la plantilla que está a continuación de la instrucción *jmpb*, el número decimal correspondiente a la plantilla complementaria a la leída después del *jmpb* y el tamaño de la misma. La función devolverá una variable booleana indicando si la plantilla complementaria se ha encontrado o no. En caso de que se haya encontrado también devolverá la dirección de la última instrucción de la plantilla encontrada. Si no se ha encontrado lo que devuelve como dirección no nos interesa.

Si no se ha encontrado la plantilla complementaria se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno y actualizaremos el puntero de instrucción de la criatura a la posición de la última instrucción de la plantilla que sigue al *jmpb*. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si se ha encontrado la plantilla complementaria se actualizará el puntero de instrucción de la criatura para que apunte a la última instrucción de la plantilla complementaria encontrada. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si ha habido error en la lectura de la plantilla se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno. A continuación actualizaremos el puntero de instrucción de la criatura sumándole el tamaño de la plantilla que sigue al *jmpb*. Si esta plantilla no existía su tamaño será cero, con lo cual al sumárselo al

[23] [Ver apartado 4.2. TMemoria]
[24] [Ver apartado 3.8 Parámetros del Sistema]
[25] [Ver apartado 4.2. TMemoria]

puntero de instrucción, éste seguirá apuntando a la misma posición, es decir, al *jmpb*. Si esta plantilla era menor que el mínimo tamaño requerido para las plantillas se le sumará al puntero de instrucción de tal manera que quede apuntando a la última posición de la plantilla. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

ADR:

Este procedimiento es el que ejecuta la instrucción *adr*. Consiste en leer la plantilla que va justo después del *adr* y buscar tanto hacia delante como hacia atrás la plantilla complementaria a la leída. En caso de encontrarse, la dirección que sigue a la plantilla que sigue a la instrucción *adr* se almacena en el registro ax y el tamaño de la plantilla a buscar se guarda en el registro cx de la cpu de la criatura.

La plantilla complementaria tendrá el mismo tamaño que la leída. A continuación vamos a detallar los pasos que se llevan a cabo.

Se llama a la función *leerPlantilla*[26] pasándole como parámetro la dirección de comienzo de la plantilla. Para ello se le suma uno a la dirección actual del puntero de instrucción que está apuntando a la instrucción *adr*. Esta función devolverá un número decimal correspondiente a la plantilla complementaria a la leída, el tamaño de la plantilla y una variable booleana indicando si ha habido error en la lectura. Habrá error en los casos en que no haya una plantilla después del *adr* o si la plantilla que hubiera es menor que el parámetro MIN_TEMPL_SIZE[27].

Si no ha habido error en la lectura se procede a la búsqueda de la plantilla complementaria. Para ello se llama a la función *buscarFB*[28] pasándole como parámetros la dirección de la primera instrucción de la plantilla que está a continuación de la instrucción *adr*, el número decimal correspondiente a la plantilla complementaria a la leída después del *adr* y el tamaño de la misma. La función devolverá una variable booleana indicando si la plantilla complementaria se ha encontrado o no.

Si no se ha encontrado la plantilla complementaria se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno.

Si se ha encontrado la plantilla complementaria se guarda la dirección que sigue a la plantilla que sigue a la instrucción *adr* en el registro ax y el tamaño de la plantilla a buscar se guarda en el registro cx de la cpu de la criatura.

Se encuentre o no la plantilla actualizaremos el puntero de instrucción de la criatura a la posición de la última instrucción de la plantilla que sigue al *adr*. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

---

[26] [Ver apartado 4.2. TMemoria]
[27] [Ver apartado 3.8. Parámetros del Sistema]
[28] [Ver apartado 4.2. TMemoria]

Si ha habido error en la lectura de la plantilla se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno. A continuación actualizaremos el puntero de instrucción de la criatura sumándole el tamaño de la plantilla que sigue al *adr*. Si esta plantilla no existía su tamaño será cero, con lo cual al sumárselo al puntero de instrucción, éste seguirá apuntando a la misma posición, es decir, al *adr*. Si esta plantilla era menor que el mínimo tamaño requerido para las plantillas se le sumará al puntero de instrucción de tal manera que quede apuntando a la última posición de la plantilla. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

ADRB:

Este procedimiento es el que ejecuta la instrucción *adrb*. Consiste en leer la plantilla que va justo después del *adrb* y buscar hacia atrás la plantilla complementaria a la leída. En caso de encontrarse, la dirección que sigue a la plantilla que sigue a la instrucción *adrb* se almacena en el registro ax y el tamaño de la plantilla a buscar se guarda en el registro cx de la cpu de la criatura.

La plantilla complementaria tendrá el mismo tamaño que la leída. A continuación vamos a detallar los pasos que se llevan a cabo.

Se llama a la función *leerPlantilla*[29] pasándole como parámetro la dirección de comienzo de la plantilla. Para ello se le suma uno a la dirección actual del puntero de instrucción que está apuntando a la instrucción *adrb*. Esta función devolverá un número decimal correspondiente a la plantilla complementaria a la leída, el tamaño de la plantilla y una variable booleana indicando si ha habido error en la lectura. Habrá error en los casos en que no haya una plantilla después del *adrb* o si la plantilla que hubiera es menor que el parámetro MIN_TEMPL_SIZE[30].

Si no ha habido error en la lectura de la plantilla se procede a la búsqueda de la plantilla complementaria. Para ello se llama a la función *buscarB*[31] pasándole como parámetros la dirección de la primera instrucción de la plantilla que está a continuación de la instrucción *adrb*, el número decimal correspondiente a la plantilla complementaria a la leída después del *adrb* y el tamaño de la misma. La función devolverá una variable booleana indicando si la plantilla complementaria se ha encontrado o no.

Si no se ha encontrado la plantilla complementaria se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno.

Si se ha encontrado la plantilla complementaria se guarda la dirección que sigue a la plantilla que sigue a la instrucción *adrb* en el registro ax y el tamaño de la plantilla a buscar se guarda en el registro cx de la cpu de la criatura.

---

[29] [Ver apartado 4.2. TMemoria]
[30] [Ver apartado 3.8. Parámetros del Sistema]
[31] [Ver apartado 4.2. TMemoria]

Se encuentre o no la plantilla actualizaremos el puntero de instrucción de la criatura a la posición de la última instrucción de la plantilla que sigue al *adrb*. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si ha habido error en la lectura de la plantilla se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno. A continuación actualizaremos el puntero de instrucción de la criatura sumándole el tamaño de la plantilla que sigue al *adrb*. Si esta plantilla no existía su tamaño será cero, con lo cual al sumárselo al puntero de instrucción, éste seguirá apuntando a la misma posición, es decir, al *adrb*. Si esta plantilla era menor que el mínimo tamaño requerido para las plantillas se le sumará al puntero de instrucción de tal manera que quede apuntando a la última posición de la plantilla. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

ADRF:

Este procedimiento es el que ejecuta la instrucción *adrf*. Consiste en leer la plantilla que va justo después del *adrf* y buscar hacia adelante la plantilla complementaria a la leída. En caso de encontrarse, la dirección que sigue a la plantilla que sigue a la instrucción *adrf* se almacena en el registro ax y el tamaño de la plantilla a buscar se guarda en el registro cx de la cpu de la criatura.

La plantilla complementaria tendrá el mismo tamaño que la leída. A continuación vamos a detallar los pasos que se llevan a cabo.

Se llama a la función *leerPlantilla*[32] pasándole como parámetro la dirección de comienzo de la plantilla. Para ello se le suma uno a la dirección actual del puntero de instrucción que está apuntando a la instrucción *adrf*. Esta función devolverá un número decimal correspondiente a la plantilla complementaria a la leída, el tamaño de la plantilla y una variable booleana indicando si ha habido error en la lectura. Habrá error en los casos en que no haya una plantilla después del *adrf* o si la plantilla que hubiera es menor que el parámetro MIN_TEMPL_SIZE[33].

Si no ha habido error en la lectura de la plantilla se procede a la búsqueda de la plantilla complementaria. Para ello se llama a la función *buscarF*[34] pasándole como parámetros la dirección de la primera instrucción de la plantilla que está a continuación de la instrucción *adrf*, el número decimal correspondiente a la plantilla complementaria a la leída después del *adrf* y el tamaño de la misma. La función devolverá una variable booleana indicando si la plantilla complementaria se ha encontrado o no.

[32] [Ver apartado 4.2. TMemoria]
[33] [Ver apartado 3.8. Parámetros del Sistema]
[34] [Ver apartado 4.2. TMemoria]

Si no se ha encontrado la plantilla complementaria se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno.

Si se ha encontrado la plantilla complementaria se guarda la dirección que sigue a la plantilla que sigue a la instrucción *adrf* en el registro ax y el tamaño de la plantilla a buscar se guarda en el registro cx de la cpu de la criatura.

Se encuentre o no la plantilla actualizaremos el puntero de instrucción de la criatura a la posición de la última instrucción de la plantilla que sigue al *adrf*. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si ha habido error en la lectura de la plantilla se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno. A continuación actualizaremos el puntero de instrucción de la criatura sumándole el tamaño de la plantilla que sigue al *adrf*. Si esta plantilla no existía su tamaño será cero, con lo cual al sumárselo al puntero de instrucción, éste seguirá apuntando a la misma posición, es decir, al *adrf*. Si esta plantilla era menor que el mínimo tamaño requerido para las plantillas se le sumará al puntero de instrucción de tal manera que quede apuntando a la última posición de la plantilla. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

CALL:

Este procedimiento es el que ejecuta la instrucción *call*. Esta instrucción es equivalente a una llamada a un procedimiento.

Se comporta igual que la instrucción *jmp* con la diferencia de que la dirección de la última instrucción de la plantilla que sigue a la instrucción *call* se almacena en la pila. De esta manera cuando se ejecute la instrucción *ret*, que es la que realiza el retorno del procedimiento, el puntero de instrucción apuntará justo a la instrucción anterior a la que se tiene que ejecutar a continuación, y así al incrementar el puntero de instrucción de la criatura después de ejecutar la instrucción *ret* se estará apuntando a la instrucción correcta.

Este procedimiento es el que ejecuta la instrucción *call*. Esta instrucción es equivalente a una llamada a un procedimiento. Consiste en leer la plantilla que va justo después del *call* y buscar tanto hacia adelante como hacia atrás la plantilla complementaria a la leída. La plantilla complementaria tendrá el mismo tamaño que la leída. A continuación vamos a detallar los pasos que se siguen.

Se llama a la función *lee*rPlantilla[35] pasándole como parámetro la dirección de comienzo de la plantilla. Para ello se le suma uno a la dirección actual del puntero de instrucción que está apuntando a la instrucción *call*. Esta función devolverá un número decimal correspondiente a la plantilla complementaria a la leída, el tamaño de la plantilla y una variable booleana indicando si ha habido error en la lectura. Habrá error en los casos en que no haya

[35] [Ver apartado 4.2. TMemoria]

una plantilla después del *call* o si la plantilla que hubiera es menor que el parámetro MIN_TEMPL_SIZE[36].

Si no ha habido error en la lectura de la plantilla se procede a la búsqueda de la plantilla complementaria. Para ello se llama a la función *buscarFB*[37] pasándole como parámetros la dirección de la primera instrucción de la plantilla que está a continuación de la instrucción *call*, el número decimal correspondiente a la plantilla complementaria a la leída después del *call* y el tamaño de la misma. La función devolverá una variable booleana indicando si la plantilla complementaria se ha encontrado o no.

Si no se ha encontrado la plantilla complementaria se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno y actualizaremos el puntero de instrucción de la criatura a la posición de la última instrucción de la plantilla que sigue al *call*. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

En caso de encontrarse, se apila la dirección de la última instrucción de la plantilla que sigue a la instrucción *call* para que cuando se llame a la instrucción *ret* se recupere esta instrucción y después de ejecutar *ret* al incrementarse el puntero de instrucción, se apunte a una instrucción significativa. Además se actualizará el puntero de instrucción de la criatura para que apunte a la última instrucción de la plantilla complementaria encontrada. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

Si ha habido error en la lectura de la plantilla se llamará al procedimiento *errorExeCriatura* que incrementará el número de errores de la criatura en uno. A continuación actualizaremos el puntero de instrucción de la criatura sumándole el tamaño de la plantilla que sigue al *call*. Si esta plantilla no existía su tamaño será cero, con lo cual al sumárselo al puntero de instrucción, éste seguirá apuntando a la misma posición, es decir, al *call*. Si esta plantilla era menor que el mínimo tamaño requerido para las plantillas se le sumará al puntero de instrucción de tal manera que quede apuntando a la última posición de la plantilla. De esta forma, cuando se salga de esta instrucción, en el procedimiento *time_slice* se incrementará en uno el puntero de instrucción y se apuntará a la siguiente instrucción que ha de ejecutar la criatura.

RET:

Este procedimiento es el que ejecuta la instrucción *ret*. Consiste en desapilar una dirección de la cima de la pila y guardarla como actual puntero de instrucción de la criatura que ejecuta esta instrucción. Para desapilar llamaremos a la función *desapilar*[38] de la clase TCpu.

Este procedimiento no da lugar a fallos.

---

[36] [Ver apartado 3.8. Parámetros del Sistema]
[37] [Ver apartado 4.2. TMemoria]
[38] [Ver apartado 4.4.TCpu]

MAL:

Este procedimiento es el que ejecuta la instrucción *mal*. Cuando una criatura ejecuta esta instrucción es para solicitar un bloque de memoria para su hija.

El tamaño del bloque de memoria a solicitar se toma del registro cx de la cpu de la criatura que solicita el espacio. Si este tamaño es menor que TAM_MIN_CRIA o mayor que TAM_MAX_CRIA[39] no se concede el bloque de memoria y se llama a la función *errorExeCriatura* para incrementar el número de errores de la criatura.

Por otra parte hay que comprobar si la criatura previamente había solicitado un bloque de memoria y no ha hecho aún un *divide*. En este caso tampoco se le concede el bloque de memoria y también se incrementa el número de fallos.

Una vez superadas estas condiciones llamamos a la función *solicitarEspacio*[40] con el tamaño del bloque requerido como parámetro. Si el espacio se obtiene sin problemas esta función devuelve la dirección de inicio de este bloque, que será el asignado a la criatura hija, que será guardada en el registro ax de la criatura madre. Si no hay espacio suficiente en la memoria, devolverá -1 y entraremos en un bucle en el que nos dispondremos a eliminar a criaturas según el orden del reaper para encontrar espacio para la nueva hija. Para ello llamamos a la función *eliminar*[41] de la clase TListaCriaturas. De esta forma eliminamos de la lista de criaturas a una criatura y después llamamos a la función *liberarEspacio*[42] para liberar ese espacio en el contenido de la memoria y que pase a formar parte de la lista de nodos libres. Después de quitarla de memoria actualizaremos su contador en el banco de genes llamando a la función *eliminaCriatura*[43] de la clase TBancoGenes.

Una vez encontrado el espacio necesario se actualizaran los parámetros de la futura criatura y se llama a la función *exitoExeCriatura* que decrementará el número de fallos de la criatura que ejecutó esta instrucción.

Si no se encontrase el espacio necesario se llamaría a la función *errorExeCriatura* y se incrementaría en uno el número de fallos de la criatura.

DIVIDE:

Este procedimiento es el que ejecuta la instrucción *divide*. Cuando una criatura tiene una hija primero ha de llamar a la instrucción *mal* para solicitar espacio de memoria para su hija, después se entra en un bucle de copia de instrucciones de la madre a la hija y por último se ejecuta el *divide* que es la encargada de darle la vida a la nueva criatura. Lo que se hace fundamentalmente es meter esta nueva criatura en las colas del slicer y del reaper.

[39] [Ver apartado 3.8. Parámetros del Sistema]
[40] [Ver apartado 4.2. TMemoria]
[41] [Ver apartado 4.7. TListaCriaturas]
[42] [Ver apartado 4.2. TMemoria]
[43] [Ver apartado 4.7. TListaCriaturas]

En la cola del reaper entra al final de la misma con lo cual será la última criatura en ser eliminada. En la cola del slicer entra justo antes de la madre con lo cual será la última criatura en obtener su turno de ejecución.

Lo primero que tenemos que hacer es comprobar que previamente a ejecutar esta instrucción ejecutamos una instrucción *mal* para solicitar espacio para una hija. En caso de que no hubiese sido así saldremos del procedimiento llamando antes a la función *errorExeCriatura* para incrementar el número de errores.

Si sí habíamos ejecutado previamente una instrucción *mal* los pasos que se siguen son los siguientes.

Comprobamos si toca o no hacer la mutación de plantilla. En el caso en que haya que hacerla recorremos el genoma de la criatura contando el número de instrucciones *nop_0* y *nop_1* que hay. Creamos un vector de tamaño el número de *nops* que hay y recorremos de nuevo el genoma de la criatura guardando en el vector las direcciones de los *nops* que vayamos encontrando. A continuación calculamos una posición aleatoria del vector sobre la que hacer la mutación y cambiaremos esa dirección de memoria en la que había un *nop* por su *nop* complementario.

El siguiente paso es dar de alta esta nueva criatura en el banco de genes. Para ello recopilamos los datos necesarios de la criatura como son las direcciones de inicio y fin o el genoma de su madre y el suyo propio y se llama a la función *nuevaCriatura*[44] de la clase TBancoGenes, que devuelve el genotipo de esta criatura.

Por último se añade esta nueva criatura a la lista de criaturas con el procedimiento *aniade*[45] de la clase TListaCriaturas, el cual se encarga de ponerla en su sitio correspondiente en el slicer y en el reaper. Actualizamos el atributo de la criatura *malYNoDivide*[46] a falso para indicar que la criatura ya puede volver a ejecutar un *mal* para tener otra hija y no puede ejecutar otro *divide* antes de haber hecho un *mal*. A continuación llamamos al procedimiento *exitoExeCriatura* para decrementar el número de errores de la criatura al haberse esta instrucción ejecutado con éxito.

MOV_IAB:

Este procedimiento es el que ejecuta la instrucción *mov_iab*. Consiste en copiar una instrucción del contenido de la memoria en otra dirección de la memoria. La instrucción origen se encuentra en la posición dada por el registro bx de la cpu de la criatura y la dirección destino se encuentra en el registro ax de la cpu de la criatura.

Esta instrucción está sujeta a algunas restricciones. Se puede leer cualquier instrucción de la memoria, pero sólo se podrá escribir en la propia criatura o en su hija si la criatura ha ejecutado un *mal* pero no un *divide*. Por tanto no se podrá escribir en los genomas de las demás criaturas de la sopa. Por otro lado controlaremos que no nos salgamos de los límites de

[44] [Ver apartado 4.10. TBancoGenes]
[45] [Ver apartado 4.7. TListaCriaturas]
[46] [Ver apartado 4.6. TCriatura]

la memoria ni por el comienzo ni por el final de la misma. Si se da alguna de estas condiciones llamaremos al procedimiento *errorExeCriatura* e incrementaremos el número de errores de la criatura. En otro caso realizaremos el intercambio de instrucciones.

INSERCIONINOCULACION:

Este procedimiento se encarga de insertar una criatura, que se ha inoculado en cualquier momento de la ejecución del sistema, en la lista de criaturas y en el banco de genes.

Se procederá a liberar el espacio de memoria necesario para que la nueva criatura pueda introducirse en la sopa. Una vez realizado esto se solicita espacio en memoria para obtener una dirección en la que insertar la nueva criatura. A partir de un fichero de texto proporcionado se introducirá el contenido en la memoria en el espacio proporcionado antes. Se introducirá la nueva criatura en todas las estructuras del sistema operativo. Este funcionamiento es el mismo que para la instrucción *divide*, exceptuando la inserción del código de la criatura a partir de un fichero de texto.

4.2. TMEMORIA

La memoria se encarga de almacenar el código de las criaturas del sistema, y de la gestión del espacio libre de la misma.

El contenido de la memoria es un array de caracteres, donde cada carácter corresponde a una instrucción, y un array de flags, cada uno de los cuales indica si la posición de memoria correspondiente ha sufrido mutación o no en la ejecución. La memoria no tiene ningún tipo de señalización para el inicio y final de las criaturas, sino que esta información será directamente almacenada por las mismas.

La traducción entre los caracteres representados en la memoria y las instrucciones con los nombres descritos en los apartados anteriores vendrá dada por la clase TTraductor[47] que proporciona métodos estáticos para la obtención del nombre a partir del carácter, así como del carácter a partir del nombre.

En la constructora de la clase hemos contemplado varias formas de inicializar el relleno del array de caracteres que representa el contenido de la memoria. Puede comenzar “vacío”, es decir, sin introducir ningún carácter determinado. O bien, relleno con caracteres correctos, refiriéndonos con correctos a caracteres que tienen traducción a una instrucción del conjunto disponible en el lenguaje ensamblador. También se podrá inicializar rellena de “criaturas ancestrales”, de forma que hay repartidos a lo largo de toda la memoria un gran número de bucles de copia correctos disponibles para lectura de cualquier criatura.

[47] [Ver apartado 4.3. TTraductor]

Para la gestión de espacio libre tiene una lista de tipo TListaEspacioLibre[48], y una variable numérica que indica la cantidad de espacio libre de la memoria.

Todos los atributos de esta clase tienen métodos para acceder a ellos o modificarlos.

LIBERARESPACIO:

Este procedimiento recibe por parámetro la dirección de comienzo y la cantidad de espacio a liberar. Actualiza el atributo *cantidadLibre* de la memoria sumándole la cantidad a liberar al espacio libre del que ya disponíamos y se llama al procedimiento *devuelveEspacio*[49] con la dirección de comienzo del espacio a liberar y la dirección del final. Este procedimiento se encarga de liberar el espacio y actualizar la lista de espacio libre.

SOLICITARESPACIO:

Esta función recibe por parámetro la cantidad de espacio libre que se solicita y devuelve la dirección de comienzo del mismo si se realiza con éxito o –1 en caso de error.

Para ello se llama a la función *solicitaEspacio*[50] de la clase TListaEspacioLibre con la cantidad de espacio requerido. Si el espacio libre de que disponemos es menor que el que solicitamos esta función devolverá –1. Si no, devolverá la dirección de comienzo del bloque libre solicitado y actualizaremos el atributo *cantidadLibre* restándole el tamaño que acabamos de pedir.

FETCH:

Esta función devuelve el contenido de una posición de memoria que se le pasa por parámetro. Esta instrucción no puede fallar ya que se ha de llamar siempre con posiciones de memoria válidas.

AFICHERO:

Este procedimiento se encarga de escribir en un archivo de texto llamado “memoria.txt” el contenido de la memoria instrucción a instrucción.

Si la instrucción a escribir ha sido una instrucción mutada se escribirá un 1 antes del contenido de la posición de memoria para señalarlo. De lo contrario se escribirá un 0.

Las instrucciones del contenido de la memoria están representadas por medio de caracteres, así que antes de escribirlas a fichero llamaremos a la función *traduceACadena*[51] de la clase TTraductor, que dado un carácter que representa a una instrucción devuelve la cadena de caracteres correspondiente a la instrucción.

---

[48] [Ver apartado 4.9. TListaEspacioLibre]
[49] [Ver apartado 4.9. TListaEspacioLibre]
[50] [Ver apartado 4.9. TListaEspacioLibre]
[51] [Ver apartado 4.3. TTraductor]

LEEMEMORIA:

Este procedimiento lee el archivo de texto “entrada.txt” cuyo contenido es el genoma de la criatura inicial y lo copia en la memoria. En este archivo las instrucciones de la criatura ancestral están representadas mediante cadenas de caracteres (su nombre completo), así que para escribirlas en la memoria habrá que traducirlas a un carácter. Para esto llamamos a la función *traduceACaracter* [52].

INOCULACION:

Este procedimiento se encarga de leer un archivo del tipo .inoc que contiene el tamaño de una criatura a inocular y el genoma de la misma. Lo lee y lo va copiando en la memoria a partir de la dirección indicada por parámetro para la nueva criatura.

LEERPLANTILLA:

Este procedimiento se encarga de leer una plantilla de la memoria. Como parámetro de entrada tiene la dirección de comienzo de la plantilla que se va a leer. Como parámetros de salida tiene el decimal correspondiente a la plantilla complementaria a la que se va a leer, el tamaño de la plantilla que se ha leído y una variable booleana que nos indicará si ha habido algún fallo o no.

Lo primero que se hace es calcular el tamaño de la plantilla. Para ello vamos leyendo posiciones de memoria mientras lo que estemos leyendo sean *nops*. A la vez que vamos leyendo y calculando el tamaño aprovecharemos para calcular el decimal complementario a la plantilla que estamos leyendo. Por ejemplo una plantilla como ésta *nop_0*, *nop_0*, *nop_0* , *nop_1* será representada por el decimal “1”. Su plantilla complementaria será *nop_1, nop_1,* nop_1, nop_0 que será representada por el decimal “14”. Al leer la plantilla original iremos tratando los *nop_0* como si fueran *nop_1* para calcular el decimal complementario, y así al leer la plantilla *nop_0, nop_0, nop_0, nop_1* obtendremos el decimal “14”.

A continuación comprobamos si la plantilla leída tiene un tamaño adecuado. Si es menor que MIN_TEMPL_SIZE o mayor que MAX_TEMPL_SIZE[53] devolveremos un error y el tamaño correspondiente de la plantilla leída. Este tamaño nos servirá después para saber cuántas instrucciones correspondientes a la plantilla tenemos que saltarnos para situar el puntero de instrucción de la criatura en la posición de la siguiente instrucción válida. Si está dentro del rango previsto se indicará que no ha habido error, se devolverá el tamaño de la plantilla y el decimal correspondiente a la plantilla complementaria.

El procedimiento desarrollado para obtener el decimal equivalente a la plantilla complementaria a la obtenida en vez de obtener un vector con la representación binaria, ha sido desarrollado de este modo ya que nos permite calcular este número y el tamaño de la

[52] [Ver apartado 4.3. TTraductor]
[53] [Ver apartado 3.8. Parámetros del sistema]

plantilla en el mismo bucle. En otro caso necesitamos reservar espacio, o bien, a posteriori de calcular el tamaño de la plantilla, o bien, reservar un espacio máximo.

DECABIN:

Este procedimiento se encarga de pasar un decimal a un binario que estará representado por un vector de ceros y unos.

Este procedimiento recibirá como parámetros el decimal que representa a la plantilla complementaria leída por el procedimiento *leerPlantilla* y el tamaño de la misma. Devolverá un puntero a un vector de ceros y unos que representará a la plantilla complementaria de la plantilla de entrada, entendiendo que cada cero representa un *nop_0* y cada uno representa un *nop_1*. La primera posición del vector contendrá el bit más significativo de la plantilla y la última posición contendrá el bit menos significativo de la plantilla.

BUSCARB:

Esta función realiza la búsqueda hacia atrás de un plantilla en el contenido de la memoria. Por lo tanto esta función será llamada en los procedimientos que implementan las instrucciones de *jmpb* y *adrb*. Estas instrucciones estarán seguidas normalmente por una plantilla que llamaremos “plantilla origen” y llamarán a *buscarB* para encontrar la plantilla complementaria a la plantilla origen y que llamaremos “plantilla destino”.

Para ello esta función recibirá como parámetros la dirección de la primera instrucción de la plantilla origen; el decimal correspondiente a la plantilla a buscar, es decir, la plantilla destino; el tamaño de la plantilla a buscar (o de la plantilla leída, tanto la plantilla origen como la plantilla destino tendrán el mismo tamaño).

Si encuentra la plantilla buscada lo indicará poniendo la variable booleana *encontrada* a cierto y devolverá la dirección de la última instrucción de la misma. En caso de no encontrarla pondrá *encontrada* a falso y no nos importará la dirección que devuelva.

A la hora de buscar la plantilla hemos de tener en cuenta dos situaciones; que ésta puede encontrarse incluida en una plantilla de mayor tamaño o que puede encontrarse en las primeras y últimas posiciones de la memoria. En el segundo caso, si hemos encontrado una parte de la plantilla al principio de la memoria, podremos seguir buscando el resto de la plantilla en las últimas posiciones de la misma.

Lo primero que hace esta función es llamar al procedimiento *decABin* que transformará el decimal que representa a la plantilla destino en su binario correspondiente y que almacenará en el vector llamado *plantilla* el cual usaremos para realizar la búsqueda.

Para la búsqueda usaremos dos variables: a, que guardará la posición de la primera instrucción de la plantilla origen y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma la búsqueda hacia atrás comienza en la dirección a – s – 1 que almacenaremos en la variable dirBusqueda. Como la memoria tiene un comienzo y un final tendremos que controlar que no nos salimos de sus límites y para ello cada vez que actualicemos cualquier variable que

recorra las posiciones de la memoria haremos el módulo de su valor con el tamaño de la memoria.

Por otra parte la búsqueda está limitada por el parámetro LIMITE_BUSQ[54]. Esto significa que sólo buscaremos hacia atrás hasta un número de instrucciones dadas por este parámetro.

Una vez situados s instrucciones antes de la instrucción *jmpb* o *adrb* con la variable dirBusqueda buscaremos la plantilla en bloques de instrucciones de tamaño s. Analizaremos el bloque instrucción a instrucción comenzando por dirBusqueda y siguiendo con dirBusqueda+1, dirBusqueda+2,..., dirBusqueda+s-1. Si coincide hemos acabado; si no, haremos que dirBusqueda apunte a una posición anterior y repetiremos la búsqueda instrucción a instrucción.

BUSCARF:

Esta función realiza la búsqueda hacia adelante de un plantilla en el contenido de la memoria. Por lo tanto esta función será llamada en el procedimiento que implementa la instrucción de *adrf*. Esta instrucción estará seguida normalmente por una plantilla que llamaremos "plantilla origen" y llamará a *buscarF* para encontrar la plantilla complementaria a la que sigue a esta instrucción y que llamaremos "plantilla destino".

Para ello esta función recibirá como parámetros la dirección de la primera instrucción de la plantilla origen; el decimal correspondiente a la plantilla a buscar, es decir, la plantilla destino; el tamaño de la plantilla a buscar (o de la plantilla leída, tanto la plantilla origen como la plantilla destino tendrán el mismo tamaño).

Si encuentra la plantilla buscada lo indicará poniendo la variable booleana *encontrada* a cierto y devolverá la dirección de la última instrucción de la misma. En caso de no encontrarla pondrá *encontrada* a falso y no nos importará la dirección que devuelva.

A la hora de buscar la plantilla hemos de tener en cuenta dos situaciones; que ésta puede encontrarse incluida en una plantilla de mayor tamaño o que puede encontrarse en las últimas y primeras posiciones de la memoria. En el segundo caso, si hemos encontrado una parte de la plantilla al final de la memoria, podremos seguir buscando el resto de la plantilla en las primeras posiciones de la misma.

Lo primero que hace esta función es llamar al procedimiento *decABin* que transformará el decimal que representa a la plantilla destino en su binario correspondiente y que almacenará en el vector llamado *plantilla* el cual usaremos para realizar la búsqueda.

Para la búsqueda usaremos dos variables: a, que guardará la posición de la primera instrucción de la "plantilla origen" y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma la búsqueda hacia adelante comienza en la dirección a + s + 1 que almacenaremos en la variable dirBusqueda. Como la memoria tiene un comienzo y un final tendremos que controlar que no nos salimos de sus límites y para ello cada vez que actualicemos cualquier variable que

[54] [Ver apartado 3.8. Parámetros del sistema]

recorra las posiciones de la memoria haremos el módulo de su valor con el tamaño de la memoria.

Por otra parte la búsqueda está limitada por el parámetro LIMITE_BUSQ[55]. Esto significa que sólo buscaremos hacia adelante hasta un número de instrucciones dadas por este parámetro.

Una vez situados s instrucciones después de la instrucción *adrf* con la variable dirBusqueda buscaremos la plantilla en bloques de instrucciones de tamaño s. Analizaremos el bloque instrucción a instrucción comenzando por dirBusqueda y siguiendo con dirBusqueda+1, dirBusqueda+2,..., dirBusqueda+s-1. Si coincide hemos acabado; si no, haremos que dirBusqueda apunte a la posición siguiente y repetiremos la búsqueda instrucción a instrucción.

BUSCARFB:

Esta función realiza la búsqueda hacia adelante y hacia atrás alternativamente, comenzando hacia delante, de un plantilla en el contenido de la memoria. Por lo tanto esta función será llamada en los procedimientos que implementan las instrucciones de *jmp* y *adr*. Estas instrucciones estarán seguidas normalmente por una plantilla que llamaremos "plantilla origen" y llamarán a *buscarFB* para encontrar la plantilla complementaria a la que sigue a estas instrucciones y que llamaremos "plantilla destino".

Para ello esta función recibirá como parámetros la dirección de la primera instrucción de la plantilla origen; el decimal correspondiente a la plantilla a buscar, es decir, la plantilla destino; el tamaño de la plantilla a buscar (o de la plantilla leída, tanto la plantilla origen como la plantilla destino tendrán el mismo tamaño).

Si encuentra la plantilla buscada lo indicará poniendo la variable booleana *encontrada* a cierto y devolverá la dirección de la última instrucción de la misma. En caso de no encontrarla pondrá *encontrada* a falso y no nos importará la dirección que devuelva.

A la hora de buscar la plantilla hemos de tener en cuenta dos situaciones; que ésta puede encontrarse incluida en una plantilla de mayor tamaño o que puede encontrarse en las últimas y primeras posiciones de la memoria. En el segundo caso, si hemos encontrado una parte de la plantilla al final o al principio de la memoria, podremos seguir buscando el resto de la plantilla en las primeras o últimas posiciones de la misma respectivamente.

Lo primero que hace esta función es llamar al procedimiento *decABin* que transformará el decimal que representa a la plantilla destino en su binario correspondiente y que almacenará en el vector llamado *plantilla* el cual usaremos para realizar la búsqueda.

Para la búsqueda usaremos dos variables: a, que guardará la posición de la primera instrucción de la plantilla origen y s, que tendrá un valor igual al tamaño de la plantilla. De esta forma la búsqueda hacia adelante comienza en la dirección a + s + 1 que almacenaremos en la variable dirBusquedaF y la búsqueda hacia atrás comienza en la dirección a - s - 1 que almacenaremos en la variable dirBusquedaB. Como la memoria tiene un comienzo y un final

[55] [Ver apartado 3.8. Parámetros del sistema]

tendremos que controlar que no nos salimos de sus límites y para ello cada vez que actualicemos cualquier variable que recorra las posiciones de la memoria haremos el módulo de su valor con el tamaño de la memoria.

Por otra parte la búsqueda está limitada por el parámetro LIMITE_BUSQ[56]. Esto significa que sólo buscaremos hacia adelante hasta un número de instrucciones dadas por este parámetro y lo mismo hacia atrás, quedando un rango de búsqueda de 2*LIMITE_BUSQ.

El método de búsqueda será igual que en *buscarF* y *buscarB*.

DAMEGENOMA:

Este procedimiento se encarga de leer del contenido de la memoria una secuencia de instrucciones que conforman un genoma. Para ello se le pasa como parámetros la dirección de comienzo del genoma y la longitud del mismo y se devuelve un array de caracteres.

4.3. TTRADUCTOR

Esta clase proporciona una serie de métodos estáticos de traducción entre las cadenas de caracteres de cada instrucción (legibles para el usuario) y la representación en memoria de las mismas: cada instrucción será un carácter en memoria. Se empleará tanto en TMemoria como TSistemaOperativo: construcción del sistema con una criatura inicial cuyo contenido ha sido leido de fichero, inoculación de criaturas a partir de un fichero con su código,etc.[57]

TRADUCEACADENA:

Devuelve la cadena de caracteres correspondiente a una instrucción representada por un carácter.

TRADUCEACHAR:

Esta función se encarga de devolver el carácter correspondiente a una instrucción representada con una cadena de caracteres.

Una muestra de la equivalencia de caracteres e instrucciones es:

nop_0 ↔ ‘a’

nop_1 ↔ ‘b’

or1 ↔ ‘c’

[56] [Ver apartado 3.8. Parámetros del sistema]

[57] Las operaciones donde se consulta a TTraductor han sido descritas en los respectivos apartados. [Ver apartado 4.1. TSistemaOperativo y 4.2. TMemoria]

NUMTOINSTRUCCION:

A partir de un entero obtenemos una instrucción, el carácter que la representa en memoria. Este método servirá para inicializaciones aleatorias de memoria

4.4. TCPU

La cpu contiene elementos que almacenan parte del estado de la criatura a la que pertenece. Para ello consta de dos registros de dirección (ax y bx), dos registros de datos (cx y dx), una pila ST de tamaño TAM_PILA[58] con su puntero de cima SP correspondiente, y un puntero de instrucción IP.

El valor de los registros de dirección se mantendrá siempre dentro de los límites de la memoria (entre 0 y TAM_MEM[59]), y el de los registros de datos entre -TAM_MEM Y TAM_MEM.

Todos los atributos de esta estructura tienen métodos para acceder a ellos o modificarlos.

INICIACPU:

Inicializa los atributos de la cpu todos a 0, excepto el puntero de instrucción que se le pasa por parámetro.

SETAX:

Asigna un valor al registro ax. Al guardar direcciones este registro, su valor se mantendrá positivo módulo TAM_MEM.

SETBX:

Asigna un valor al registro bx. Al guardar direcciones este registro, su valor se mantendrá positivo módulo TAM_MEM.

SETCX:

Asigna un valor al registro cx. Su valor debe estar entre -TAM_MEM y TAM_MEM. Si no, se pone el registro a 0.

SETDX:

Asigna un valor al registro dx. Su valor debe estar entre -TAM_MEM y TAM_MEM. Si no, se pone el registro a 0.

[58] [Ver apartado 3.8. Parámetros del sistema]
[59] [Ver apartado 3.8. Parámetros del sistema]

APILAR:

Este procedimiento apila el valor que se le pasa como parámetro y luego incrementa el puntero de pila módulo TAM_PILA[60]. El puntero de pila siempre apunta a la posición por encima del último dato introducido en la pila. Esto es porque al empezar el puntero de pila se inicializa al valor 0 que es la primera posición vacía.

DESAPILAR:

Este procedimiento decrementa el puntero de pila módulo TAM_PILA y luego desapila el valor que se le pasa como parámetro. Si el puntero de pila apuntaba a la posición 0 la posición que se despilará será la TAM_PILA – 1; esto es debido a que la pila es circular.

SETIP:

Esta función asigna el valor dado por parámetro, módulo TAM_MEM, al puntero de instrucción.

INCREMENTARIP:

Este procedimiento incrementa en uno, módulo TAM_MEM, el valor del puntero de instrucción.

NOP0:

Este procedimiento es el que ejecuta la instrucción *nop_0*. No hace nada.

NOP1:

Este procedimiento es el que ejecuta la instrucción *nop_1*. No hace nada.

OR1:

Este procedimiento es el que ejecuta la instrucción *or1*. Para ello cambia el bit de menor orden del registro cx:

-Si el número es par, en binario acaba en 0, así que le sumamos 1.

-Si el número es impar, en binario acaba en 1, así que le restamos 1.

SHL:

Este procedimiento es el que ejecuta la instrucción *shl*. Para ello desplaza todos los bits del registro cx una posición a la izquierda poniendo un 0 en el bit menos significativo del registro. A nivel binario esto es equivalente a multiplicar por 2.

[60] [Ver apartado 3.8. Parámetros del sistema]

ZERO:

Este procedimiento es el que ejecuta la instrucción *zero*. Para ello pone a 0 el registro cx.

IF_CZ:

Este procedimiento es el que ejecuta la instrucción *if_cz*. Para ello incrementa el puntero de instrucción en 1 sólo si el valor del registro cx es distinto de 0. Esto significa que la instrucción siguiente a la actual será ejecutada sólo si el valor del registro cx es 0.

SUB_AB:

Este procedimiento es el que ejecuta la instrucción *sub_ab*. Para ello resta el valor contenido en el registro bx al valor contenido en el registro ax almacenando el resultado en el registro cx.

SUB_AC:

Este procedimiento es el que ejecuta la instrucción *sub_ac*. Para ello resta el valor contenido en el registro cx al valor contenido en el registro ax almacenando el resultado en el registro ax.

INC_A:

Este procedimiento es el que ejecuta la instrucción *inc_a*. Para ello incrementa en 1 el valor del registro ax volviendo a almacenar el resultado en ax.

INC_B:

Este procedimiento es el que ejecuta la instrucción *inc_b*. Para ello incrementa en 1 el valor del registro bx volviendo a almacenar el resultado en bx.

INC_C :

Este procedimiento es el que ejecuta la instrucción *inc_c*. Para ello incrementa en 1 el valor del registro cx volviendo a almacenar el resultado en cx.

DEC_C :

Este procedimiento es el que ejecuta la instrucción *dec_c*. Para ello decrementa en 1 el valor del registro cx volviendo a almacenar el resultado en cx.

PUSH_AX:

Este procedimiento es el que ejecuta la instrucción *push_ax*. Para ello apila el valor que contiene el registro ax en la pila e incrementa el puntero de pila en uno, módulo TAM_PILA[61].

PUSH_BX:

Este procedimiento es el que ejecuta la instrucción *push_bx*. Para ello apila el valor que contiene el registro bx en la pila e incrementa el puntero de pila en uno, módulo TAM_PILA.

PUSH_CX:

Este procedimiento es el que ejecuta la instrucción *push_cx*. Para ello apila el valor que contiene el registro cx en la pila e incrementa el puntero de pila en uno, módulo TAM_PILA.

PUSH_DX:

Este procedimiento es el que ejecuta la instrucción *push_dx*. Para ello apila el valor que contiene el registro dx en la pila e incrementa el puntero de pila en uno, módulo TAM_PILA.

POP_AX:

Este procedimiento es el que ejecuta la instrucción *pop_ax*. Para ello desapila la cima de la pila y almacena su valor en el registro ax y decrementa el puntero de pila en uno, módulo TAM_PILA.

POP_BX:

Este procedimiento es el que ejecuta la instrucción *pop_bx*. Para ello desapila la cima de la pila y almacena su valor en el registro bx y decrementa el puntero de pila en uno, módulo TAM_PILA.

POP_CX:

Este procedimiento es el que ejecuta la instrucción *pop_cx*. Para ello desapila la cima de la pila y almacena su valor en el registro cx y decrementa el puntero de pila en uno, módulo TAM_PILA.

POP_DX:

Este procedimiento es el que ejecuta la instrucción *pop_dx*. Para ello desapila la cima de la pila y almacena su valor en el registro dx y decrementa el puntero de pila en uno, módulo TAM_PILA.

[61] [Ver apartado 3.8. Parámetros del sistema]

MOV_AB:

Este procedimiento es el que ejecuta la instrucción *mov_ab*. Para ello copia el contenido del registro ax en el registro bx dejando intacto el registro ax.

MOV_CD:

Este procedimiento es el que ejecuta la instrucción *mov_cd*. Para ello copia el contenido del registro cx en el registro dx dejando intacto el registro cx.

4.5. TCRIATURA

Cada criatura almacena datos relativos a su posición en la memoria y tamaño, así como datos relativos a la hija que está produciendo (para la que ha reservado espacio usando la instrucción *mal*, pero no ha dado a luz usando la instrucción *divide*). También guarda el número de errores que lleva acumulados, y su nombre de genotipo.

Dado que una criatura que ha ejecutado una instrucción *mal* no puede volver a hacerlo hasta que no haya ejecutado el *divide*, se guarda también una variable booleana que indica si esta criatura ha ejecutado la instrucción de reserva de memoria pero aún no ha llegado a la instrucción para crear la criatura independiente. Esta variable será de utilidad a la hora de comprobar el error en la ejecución del bucle de copia de la criatura.

Para contemplar la restricción del sistema consistente en que no se podrá crear una hija independiente con la instrucción *divide* si no se ha copiado un porcentaje mínimo de instrucciones desde que se realizó la reserva de espacio con *mal*, cada criatura llevará un contador para sumar, precisamente, el número de instrucciones *mov_ia* que lleva realizadas. Si al llegar a *divide* el contador no lleva acumulado un determinado porcentaje de instrucciones de copia respecto al propio tamaño de la criatura, la instrucción *divide* fallará. El parámetro que determina ese porcentaje mínimo es PROP_MOV [62] Finalmente, a cada criatura se le asigna una cpu de tipo TCpu[63].

Todos los atributos de esta estructura tienen métodos para acceder a ellos o modificarlos.

4.6. TNODOCRIATURA

Éstos son los nodos que forman la lista de criaturas. Dado que se trata de una lista enlazada, y que se puede recorrer de dos formas (en el orden del reaper y en el orden del slicer), cada nodo contiene referencias a los nodos anterior y posterior para los dos órdenes.

Todos los atributos de esta clase tienen métodos para acceder a ellos o modificarlos.

[62] [Ver apartado 3.8. Parámetros del sistema]
[63] [Ver apartado 4.4. TCpu]

4.7. TLISTACRIATURAS

La lista de criaturas se ha implementado como una lista enlazada de nodos TNodoCriatura con dos recorridos diferentes: el del reaper y el del slicer. De esta forma se guardan las dos listas de criaturas con que trabaja el sistema en una sola.

En el caso del reaper sólo necesitamos conocer su primer elemento, para eliminarlo cuando corresponda, y su último elemento, para insertar nuevos nodos al final. Para ello se guardan los punteros priReaper y ultReaper.

En el caso del slicer sólo es necesario conocer la criatura que está ejecutándose en cada momento, y para ello se tienen el puntero curSlicer.

Además se guarda en un parámetro numérico la longitud de la lista, es decir, el número de criaturas del sistema.

Todos los atributos de esta clase tienen métodos para acceder a ellos o modificarlos.

ANIADE:

Este procedimiento es el encargado de insertar una nueva criatura en la lista de criaturas. Para ello deberá situarlo antes del nodo en curso del slicer y después del último nodo del reaper.

ELIMINAR:

Esta función elimina la primera criatura del reaper devolviendo la dirección de inicio y el tamaño de la criatura eliminada. Nunca se eliminará la criatura en ejecución. En el caso de que sea la primera, se eliminará la siguiente.

Si la lista sólo tiene una criatura no la borraremos por ser la que se está ejecutando actualmente e indicaremos que ha habido error. Si sólo hay una criatura y se llama a esta función es porque esta criatura está intentando reproducirse, está pidiendo espacio para su hija y no hay suficiente en la memoria. Que no haya espacio suficiente en la memoria habiendo sólo una criatura significa que ésta ocupa la mayor parte de la misma. Por esto indicamos un error que luego será recogido en el procedimiento *life* de TSistemaOperativo y se acabará con la vida del sistema, ya que esta criatura nunca podrá reproducirse a menos que una mutación muy acertada (y muy improbable) la modificase de tal manera que cambiara la solicitud de espacio de memoria para su hija y que ésta consiguiera reproducirse y dar lugar a nuevas criaturas que pueblen la sopa.

Si hay más de una criatura eliminamos la primera del reaper si no es la que se está ejecutando actualmente, en cuyo caso eliminaríamos la siguiente en el orden del reaper.

En caso en que la criatura que borremos esté en proceso de reproducción devolveremos también los datos de la criatura hija: su dirección de comienzo, su tamaño y su genotipo junto con una variable booleana que nos indicará este hecho.

ORDENAREAPER:

Este procedimiento se encarga de reordenar el reaper, considerando que la única criatura que ha cambiado su número de errores es la criatura en curso. Por tanto, el nodo de esa criatura será el único que deba actualizar su posición en el reaper respecto a los otros según haya aumentado o disminuido su número de errores.

Para ello miramos cuál es la criatura anterior en el orden del reaper y se ordenan los dos nodos según el número de errores, intercambiándolos usando la función *conmutaNodos* si es necesario y continuando este intercambio hasta que el nodo se encuentre detrás de otro con número de errores mayor y por delante de una criatura con número de errores menor.

Puede ocurrir que la criatura haya reducido sus errores, y en este caso se realizará la operación de desplazamiento del nodo de la criatura pero en sentido contrario. Se comprueba cuál es la criatura posterior en el orden del reaper y se ordenan los dos nodos según el número de errores, intercambiándolos usando la función *conmutaNodos* si es necesario.

CONMUTANODOS:

Este procedimiento intercambia en la lista los nodos pasados por parámetro en el orden del reaper. Se intercambiará su orden en el reaper, pero se mantendrá el orden en el slicer de las criaturas contenidas en cada nodo.

4.8. TNODOESPLIBRE

Éstos son los nodos que forman la lista de espacio libre TListaEspLibre. Cada nodo representa un espacio libre de la memoria, indicado por sus direcciones de inicio y de fin. Como la lista de espacio libre es una lista enlazada, cada nodo guarda referencias a sus nodos anterior y posterior.

Todos los atributos de esta clase tienen métodos para acceder a ellos o modificarlos.

4.9. TLISTAESPACIOLIBRE

La lista de espacio libre es una lista enlazada de nodos de tipo TNodoEspLibre. Para poder buscar el nodo que corresponda a la hora de solicitar espacio o liberarlo, se utilizan punteros al primer nodo y al último usado, *pri* y *cur*. Además, la lista guarda la cantidad de nodos libres que contiene.

Esta clase es la que soporta la lógica de gestión del espacio libre de memoria. La política es de partición dinámica y asignación de espacio y algoritmo de ubicación de primer ajuste. El espacio libre en la memoria se recoge como el conjunto de huecos libres ordenados por dirección de menor a mayor. Cuando se libera espacio de memoria se mira si se puede añadir ese hueco a alguno de los existentes ya en la lista de espacio libre si fuera contiguo al

liberado. A la hora de asignar espacio en memoria se busca el primer hueco libre cuyo capacidad satisfaga la petición, creando un nodo de espacio libre con la cantidad sobrante en caso de que el hueco encontrado fuera mayor que la cantidad requerida.

Todos los atributos de esta clase tienen métodos para acceder a ellos o modificarlos.

SOLAPANODOS:

Este procedimiento se encarga de comprobar si el bloque de espacio libre en curso se puede solapar con su anterior y/o su posterior en la lista formando un solo bloque de tamaño mayor.

Solapamos manteniendo el nodo anterior. Así si éste es el primero, no habrá que hacer comprobaciones para actualizar este puntero. Nunca podrá ocurrir entonces que el primero se destruya, ya que nunca se podrá solapar con el último (habiendo nodos en medio). Los nodos que se destruyen no podrán ser nunca el primero.

SOLICITAESPACIO:

Esta función devuelve la dirección de un bloque libre de tamaño igual o mayor que el solicitado por parámetro.

Si el bloque que devuelve es del mismo tamaño que el requerido se borra el bloque de la lista. Si es mayor, se borra y se inserta un nuevo bloque del tamaño del espacio sobrante.

DEVUELVEESPACIO:

Este procedimiento inserta un nuevo bloque de espacio libre en la lista. Se comprueba si se puede solapar con el bloque anterior y/o con el posterior de la lista. El bloque a liberar viene dado por las direcciones de comienzo y de fin que se pasan como parámetros.

La búsqueda de la posición donde insertar el nuevo bloque se realizará secuencialmente en orden creciente de las direcciones de comienzo de los bloques a partir del bloque en curso.

4.10. TBANCOGENES

El banco de genes será el encargado de recopilar la información presente en el sistema respecto a genotipos diferentes y número de criaturas vivas de cada genotipo, así como registrar información en una serie de ficheros de texto[64]. TBancoGenes es la clase principal del conjunto de ellas mediante las cuales mantenemos esta información.

La implementación del banco de genes es completamente independiente del código del sistema ALiS siendo la clase TBancoGenes la que ofrece al sistema operativo tres métodos

[64] Los ficheros que generará serán el registro de nacimientos y muertes y los ficheros .gen que recogerán los genotipos y respectivos genomas para cada tamaño de criaturas. [Ver apartado 6.1. Entrada del Sistema, donde se describe el formato de estos ficheros que serán entradas de la aplicación de análisis]

que deberán ser invocados cada vez que acontece en el sistema un hecho relevante para la criatura:

- Nace una criatura: se llama al método del banco de genes *nuevaCriatura*
- Muere una criatura: se llama al método del banco de genes *eliminaCriatura*
- Muta una criatura: se llama al método del banco de genes *mutaCri*atura

De esta forma se registrará la evolución de las criaturas en el sistema.

El atributo principal del banco de genes es una lista del tipo TListaTamanios, en la cual se mantendrá la lista de los genotipos presentes en el sistema ordenados en primer lugar por tamaños y, en un segundo nivel, para cada tamaño por la lista de genotipos de ese tamaño.

También dispone como atributo del fichero en el que irá registrando los nacimientos y muertes (RegBirthDeath.txt) que irá actualizando cada vez que haya un cambio al respecto en el sistema y que mantendrá abierto durante toda la ejecución.

REGISTRANACIMIENTO:

Función encargada de volcar la información del nacimiento de una criatura en el fichero RegBirthDeath.txt en el formato determinado para la visualización y análisis[65]

REGISTRAMUERTE:

Función encargada de volcar la información de la muerte de una criatura en el fichero RegBirthDeath.txt en el formato determinado para la visualización y análisis. [66]

NUEVACRIATURA:

Función que se invocará para registrar en el banco de genes la aparición de una nueva criatura en el sistema. Se deberán pasar por parámetro todos los datos necesarios para recoger información lo más completa posible de la criatura en cuestión: el número de instrucción ejecutada del sistema en que nace la criatura, genoma de la criatura, tamaño, dirección de inicio, genoma de la madre, genotipo de la madre, dirección de inicio de la madre, dirección de la instrucción *divide* cuya ejecución es la responsable del nacimiento de la criatura, fecha de nacimiento de la madre.

A su vez, esta función devolverá el genotipo asignado a esa criatura en el banco de genes.

Los pasos para el registro de la criatura son los siguientes:

- Si la criatura y la madre tienen el mismo genoma, tendrán el mismo genotipo. Hay una criatura más en el sistema con el genotipo de la madre.
- Si no tiene el mismo genoma que la madre, puede que tenga igual genoma que otra criatura ya presente en el sistema. Se procede a buscar en la estructura

[65] [Ver apartado 6.1. Entrada del Sistema, del Analizador Semántico de Alto Nivel]
[66] [Ver apartado 6.1. Entrada del Sistema, del Analizador Semántico de Alto Nivel]

del bando de genes dicha criatura. Si se encontrara, ya se tendría genotipo para la nueva criatura.

- Si no existiera en ese momento en el sistema una criatura con el mismo genoma, puede que en el pasado u otra ejecución apareciera dicha criatura. Se procede a buscar en los ficheros .gen el genoma de la criatura que acaba de nacer.
- Si no se encontrara en ninguno de los tres casos anteriores un genoma igual, se procederá a buscar un nuevo nombre para la criatura e insertarla en el banco de genes.

BUSCADISCO:

Será la función auxiliar utilizada por la anterior para realizar la búsqueda en los archivos .gen presentes en el directorio de la aplicación.

ELIMINACRIATURA:

Función a invocar cuando en el sistema muera una criatura. Se indicarán los datos identificativos de la criatura que muere: número de instrucción en la ejecución del sistema en que muere la criatura, tamaño, genotipo de la criatura y fecha en que nació la criatura (número de instrucción). Se procederá a decrementar el número de criaturas presentes en el sistema con ese genotipo y, en caso de ser la última, se eliminará del banco de genes. Se registrará su muerte en el fichero registro de nacimientos y muertes.

MUTACRIATURA:

Cuando mute en el sistema el contenido de una dirección de memoria perteneciente a una criatura habrá que indicar este hecho en el banco de genes. Se indicará el tamaño y genotipo de la criatura antes de la mutación y el genoma nuevo de la criatura. Se procederá a operar como la eliminación de una criatura con genotipo antiguo y la inserción de una criatura con el genoma nuevo. La asignación de genotipo será igual que en el caso de la llamada a *nuevaCriatura*. En el fichero de registro de nacimientos y muertes se registrará en el mismo instante de tiempo una muerte y un nacimiento de acuerdo a los datos de la criatura.

DATOSGENOTIPOCRIAS:

Esta función permite obtener la información contenida en el banco de genes respecto a genotipos presentes y número de criaturas de cada genotipo resumida en dos arrays: la lista de genotipos y para cada genotipo el número de criaturas. Es de utilidad para presentación gráfica y para guardar los datos en un fichero de texto para el posterior análisis.

4.11. TLISTATAMANIOS

Esta clase recoge la lista de todos los tamaños diferentes de criaturas presentes en el sistema. Para cada tamaño distinto se recogerá información sobre los genotipos de ese tamaño como describiremos en los siguientes apartados. Además recoge información del número total de criaturas registradas y número de genotipos distintos en el sistema.

INCREMENTA:

Dado un tamaño y genotipo se invoca a esta función para que aumente en uno el número de criaturas presentes en el sistema. Esta función será invocada cuando se identifique que la nueva criatura es igual a su madre y sólo haga falta modificar el número de las criaturas presentes del genotipo correspondiente.

BUSCAINSERTAR:

Realiza la búsqueda del genoma dado entre las criaturas presentes en el banco de genes. En caso de encontrar una criatura con igual genoma devolverá su genotipo y la función retornará *true*; en caso de no encontrarlo la función retornará *false.*

ANIADE:

Cuando se haya localizado el genoma de la criatura en el fichero .gen, se procederá a insertarla en el banco de genes, con el genotipo conseguido del fichero, mediante esta función.

ANIADENUEVO:

Esta función se invocará cuando, al nacer una nueva criatura en el sistema, no se ha podido insertar en el banco de genes por ninguno de los métodos anteriores: no es igual a la madre, no estaba presente en el banco una criatura de igual genoma y tampoco existía en fichero un nombre para ella. Esta función será la que generará un nuevo nombre, genotipo, que asignar a la criatura.

BUSCAELIMINAR:

Función que, dado un tamaño y genotipo, registrará la desaparición de una criatura con esos datos. Si fuera la última criatura de un tamaño determinado, desaparecerá el nodo con información de ese tamaño. Esta función se invocará cuando en el sistema muera alguna criatura, por tanto, el genotipo de esa criatura existirá en el banco de genes ya que fue el dado en su registro y, por ello, en esta función no se pueden dar errores por no encontrar el genotipo indicado.

DATOSGENOTIPOCRIAS:

Devuelve los datos resumidos del banco de genes recorriendo cada nodo de la lista de tamaños.

4.12. TNODOTAMANIO

Esta clase corresponde a los nodos de la lista de tamaños: TListaTamanios. Cada nodo recoge la información de todas las criaturas presentes en el sistema con un tamaño determinado. Así, en este tipo de nodos, se mantiene una lista con los genotipos de las criaturas del tamaño que corresponde al nodo: TListaGenotipos[67].

Como podemos observar, los métodos de los que disponemos en esta clase se denominan igual que los de la clase TListaTamanios. Es así ya que la funcionalidad es la misma, pero restringida al conjunto de criaturas pertenecientes a cada nodo, es decir, al conjunto de criaturas con el mismo tamaño.

INCREMENTA:

Función que aumenta en uno el número de criaturas presentes en el sistema con un determinado genotipo.

BUSCAINSERTAR:

Realiza una búsqueda en la lista de genomas del tamaño del nodo para localizar si existe genotipo con el genoma pasado por parámetro. Se indicará si se ha encontrado o no el genoma y, en caso positivo, se devolverá el genotipo correspondiente a dicho genoma.

ANIADENUEVO:

Añade un nuevo genoma a la lista de genomas del tamaño del nodo. Se deberá asignar un nuevo nombre de genotipo para la criatura con dicho genoma.

ANIADE:

Se añade un nuevo genotipo y genoma directamente en la lista de genomas del nodo.

BUSCAELIMINAR:

Se localiza el genotipo indicado por parámetro para proceder a la reducción en el registro de número de criaturas con ese genotipo y, en caso de ser la última criatura con ese nombre, eliminar la información del genotipo del banco de genes.

[67] [Ver apartado 4.13. TListaGenotipos]

DATOSGENOTIPOCRIAS:

Resume la información del banco de genes para los genotipos del tamaño del nodo.

4.13. TLISTAGENOTIPOS

Este tipo será una lista enlazada de nodos TNodoGenotipo. Así recogerá la información de toda una serie de genotipos.

Las primeras cuatro funciones son las que se invocarán desde los métodos de la clase TNodoGenotipo en cada uno de los casos descritos.

INCREMENTA:

Incrementa en uno el número de criaturas de un genotipo dado.

BUSCAINSERTAR:

Busca en la lista de genotipos un genoma igual al pasado por parámetro. En caso de encontrarlo devolverá el genotipo correspondiente.

ANIADENUEVO:

Añade un nuevo genoma a la lista.

ANIADE:

Añade un nuevo genotipo y su genoma asociado a la lista.

BUSCAELIMINAR:

Realiza una búsqueda del genoma indicado por parámetro procediendo a su eliminación de la lista.

MIRARGUARDARGENOTIPO:

Esta función será la encargada de, cuando se supere el número de criaturas presentes en el banco de genes indicado por LIM_SAVE_DISK[68] proceder a guardar el genotipo y genoma de esas criaturas en los ficheros .gen [69].

[68] [Ver apartado 3.8. Parámetros del Sistema]
[69] [Ver apartado 6.1. Entrada del Sistema, del Analizador Semántico de Alto Nivel]

LEEFICHERO:

Función encargada de leer el contenido de un fichero .gen cargando su contenido en una lista de genotipos. Dicha lista servirá para poder realizar búsquedas y así poder tratar la información contenida en el fichero .gen indicado obteniendo los datos que se deseen sin tener que leer de fichero más que la primera vez.

DATOSGENOTIPOSCRIAS:

Esta función será invocada por los respectivos métodos con igual nombre explicados en apartados anteriores. En concreto, en esta clase se procede a recuperar los datos de la lista de genotipos desde la que se invoca la función.

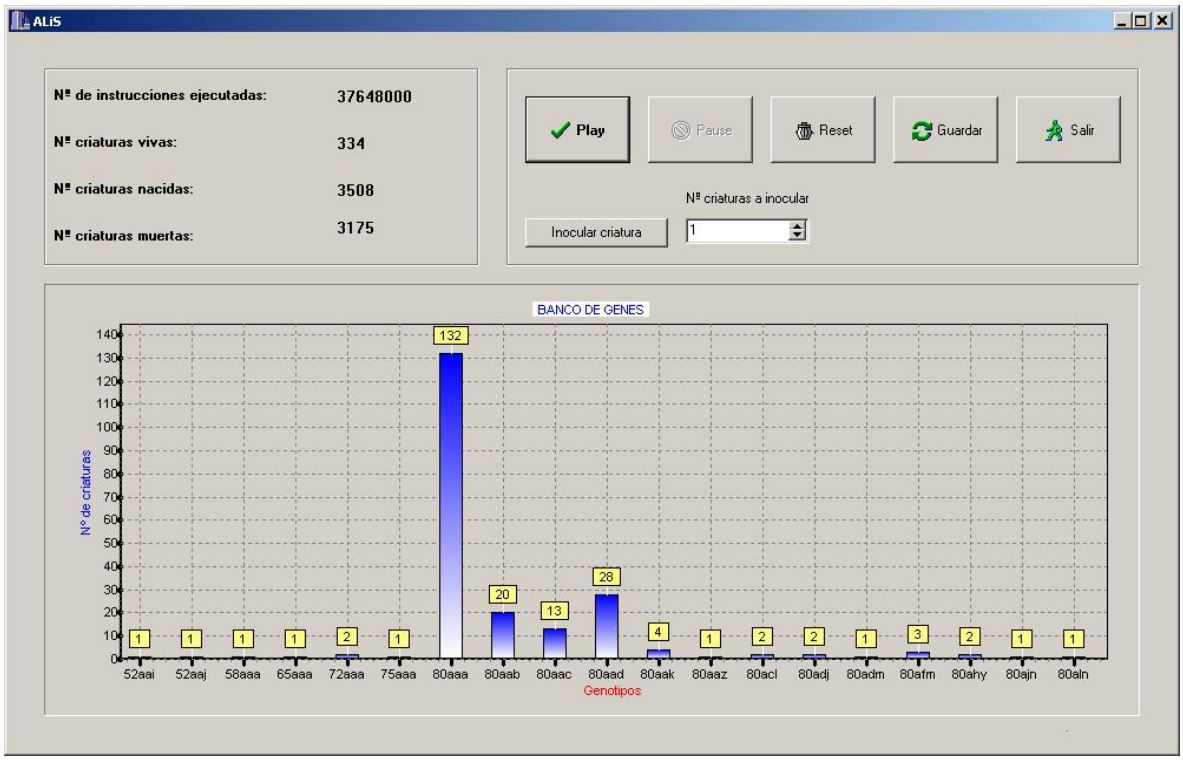
4.14. TNODOGENOTIPO

Por último, esta clase recoge toda la información de un genotipo: el nombre del genotipo, tamaño del genoma, genoma asociado al genotipo, número de criaturas con dicho genotipo y si se ha recuperado o guardado en el fichero .gen correspondiente

Dispondremos de métodos para la consulta y actualización de todos los datos anteriores.

5.INTERFAZ GRÁFICO

El interfaz gráfico de ALiS es un módulo independiente del sistema ALiS que nos ayuda a ver en tiempo real el contenido del banco de genes así como poder manejar el sistema ALiS: arrancar, inocular criaturas, guardar la información en fichero, etc.



En él mostramos el número de instrucciones que lleva ejecutadas el sistema, el número de criaturas nacidas, el número de criaturas vivas y el número de criaturas muertas desde el comienzo de la ejecución. Además se muestra un gráfico de barras en el que sobre el eje X aparecen los distintos genotipos del banco de genes y sobre el eje Y el número concreto de criaturas de cada genotipo. El gráfico de barras se puede mover a la izquierda y a la derecha manteniendo pulsado el botón derecho del ratón y arrastrando en el sentido deseado. Para volver a la situación inicial, es decir, para que se muestren las barras desde la primera, mantendremos pulsado el botón izquierdo del ratón y dibujaremos un pequeño cuadrado.

Para comenzar una ejecución del sistema primero hemos de inocular la sopa con al menos una criatura. Para ello seleccionamos el número de criaturas que queremos inocular (de 1 a 100) y pulsamos el botón “Inocular criatura”. Se nos mostrará un cuadro de dialogo desde el que elegiremos el archivo correspondiente a la criatura que queramos introducir. Los archivos que contienen este tipo de información tienen la extensión .inoc y contienen como primera línea el tamaño de la criatura y a continuación el genoma de la misma.

Una vez inoculada la sopa ya podremos comenzar la ejecución pulsando el botón “Play”. En cualquier momento de la ejecución podremos inocular nuevas criaturas.

También se nos ofrece un botón de pausa “Pause” para parar la ejecución, la cual se podrá continuar pulsando de nuevo el botón “Play”.

Otro botón es el de “Reset” que nos permite parar la ejecución borrando el gráfico de barras y los archivos .gen guardados por el sistema, para comenzar otra ejecución.

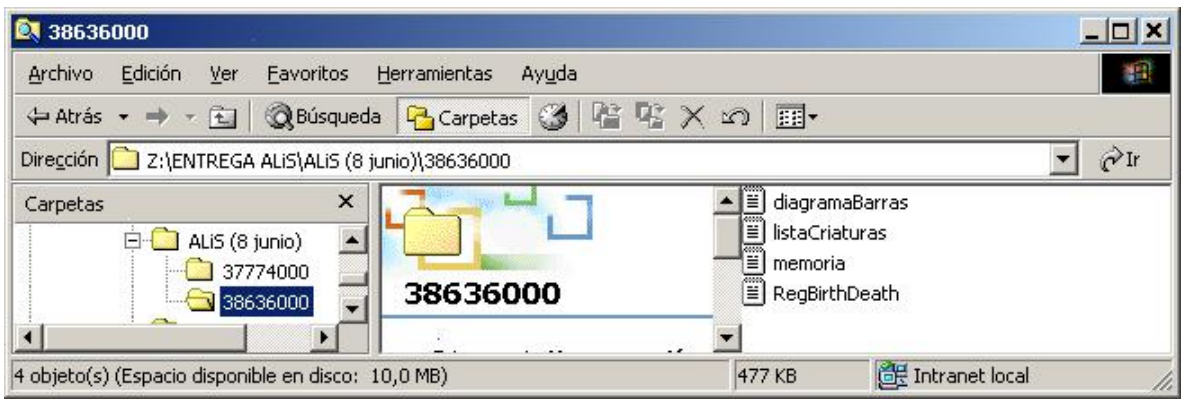
También tenemos el botón de “Guardar” que salva la lista de criaturas, el contenido de la memoria y los datos del banco de genes en archivos. Cada vez que se guarden los datos se creará un directorio que llevará por nombre el número de instrucción por la que va ejecutando el sistema, donde se guardarán estos archivos y así poder identificar y organizar los resultados obtenidos por la ejecución en distintos momentos de tiempo.

Por último, para terminar la ejecución y salir de la aplicación usaremos el botón “Salir”.

6.ANALIZADOR SEMÁNTICO DE ALTO NIVEL

6.1. ENTRADA DEL SISTEMA

Como se ha comentado en el apartado anterior, el interfaz gráfico de ALiS permite guardar la información del sistema, cuando lo deseamos, pulsando el botón “Guardar”. Al hacerlo, se generará un directorio con nombre el número de instrucción en la que se encuentre el sistema. Dentro de ese directorio se guardarán cuatro ficheros de texto donde se recogerá el estado del sistema en ese instante de ejecución.



Los ficheros de texto y la información que recogen es la siguiente:

- memoria.txt : recoge el contenido de la memoria en ese instante. El formato es el siguiente: la primera línea es el número de direcciones de la memoria (su tamaño), las sucesivas líneas recogerán un carácter por línea que será la representación de las instrucciones de ALiS en el sistema (habrá tantas líneas como sea el tamaño de la memoria).

- listaDeCriaturas.txt: recoge por línea la información de cada criatura presente en el sistema en el momento de grabar la información en el orden del slicer. Habrá tantas líneas como criaturas vivas en el sistema. Los campos de cada línea son:

    - Dirección de inicio en memoria de la criatura
    - Dirección última en memoria de la criatura
    - Tamaño de la criatura
    - Genotipo asignado a la criatura
    - "T" si la criatura había ejecutado una instrucción *mal* y todavía no había ejecutado una instrucción *divide*. "F" en otro caso.
    - En caso de que el campo anterior fuera "T", este campo indica la dirección inicial del fragmento de memoria asignado a la criatura al ejecutar la instrucción *mal*. Si el valor del campo anterior era "F" el contenido de este campo puede ser cualquier número.
    - Bajo las mismas circunstancias del campo anterior, éste refleja la dirección final del fragmento de memoria que posee la criatura tras ejecutar la instrucción *mal*.
    - Número de errores que lleva acumulados la criatura desde su nacimiento. Este número de errores será el campo que se toma como referencia para el orden del reaper.
    - Número de hijas que ha tenido la criatura desde su nacimiento. Recoge el número de hijas independientes, es decir, hijas para las que se ha ejecutado la instrucción *divide* y ya son seres que están en el sistema ejecutándose, no se incluye la posible futura hija que esté en ese momento creando la criatura (si ha ejecutado la instrucción *mal* y todavía no ha ejecutado el *divide*)
    - Contenido del registro ax de la criatura. Este campo se recoge ya que la dirección a la que apunte este registro es determinante para el bucle de copia (para conocer las instrucciones que se están copiando ya que

pueden pertenecer a la propia criatura o a cualquier otro espacio de la memoria).

- Contenido del registro bx de la criatura. Como en el caso anterior hay que tener en cuenta su valor para el análisis del código que esté ejecutando la criatura.

- Contenido del puntero de instrucción, IP, de la criatura. Nos permitirá saber si la criatura está ejecutando su propio código o instrucciones de cualquier otra dirección de memoria que no le pertenece.

- Número de instrucción en la que la criatura nació (cuando la madre ejecutó el correspondiente *divide*). Sería un símil a la fecha de nacimiento y nos permite identificar unívocamente a cada criatura (no puede haber dos criaturas que nazcan a la vez ya que en cada instante de ejecución es el turno de una criatura concreta)

- Dirección de comienzo de la madre en memoria. Éste será el primero de una serie de datos que recogemos sobre la madre de la criatura. Estos datos nos permitirán saber cómo se ha creado la criatura (posible mutación)

- Tamaño de la madre de la criatura

- Genotipo de la madre

- Dirección de la instrucción *divide* que ejecutó la madre en el momento de crear a la criatura. Con esta información podremos saber si la madre estaba ejecutando código propio o ajeno. Este dato, visto globalmente para todas las criaturas, nos puede aportar información de quién está realmente procreando en el sistema.

- Número de instrucción en la que nació la madre. Este dato nos permite identificar unívocamente a la madre de la criatura y poder analizar posteriormente si sigue presente en el sistema o ha muerto, y todos los datos asociados a esta criatura de los que dispongamos en cada caso.

- RegBirthDeath.txt : Este archivo se va actualizando por el sistema ALiS cada vez que hay un nacimiento o muerte de criatura, indicando en cada caso en una línea de texto la información apropiada. Así, este fichero refleja en orden

cronológico los nacimientos y muertes de crías. En el momento de pulsar el botón “Guardar” se copiará este fichero en el directorio generado para ese instante de ejecución, reflejando este fichero copiado el registro de nacimientos y muertes hasta el número de instrucción en que se salva la información. El formato por línea de este fichero es:

  - Para un nacimiento :
    - Número de instrucción de nacimiento
    - “b”
    - Tamaño de la criatura nacida
    - Genotipo asignado a la criatura nacida
    - La dirección inicial de la criatura
    - Tamaño de la madre
    - Nombre de la madre
    - Dirección inicial de la madre en memoria
    - Número de instrucción en que nació la madre
    - Valor del puntero de instrucción de la madre (IP) en el momento de nacimiento de la hija (instrucción divide)
  - Para una muerte:
    - Número de instrucción en la que muere la criatura
    - “d”
    - Tamaño de la criatura que fallece
    - Genotipo de la criatura que fallece
    - Número de instrucción en la que nació la criatura que fallece.

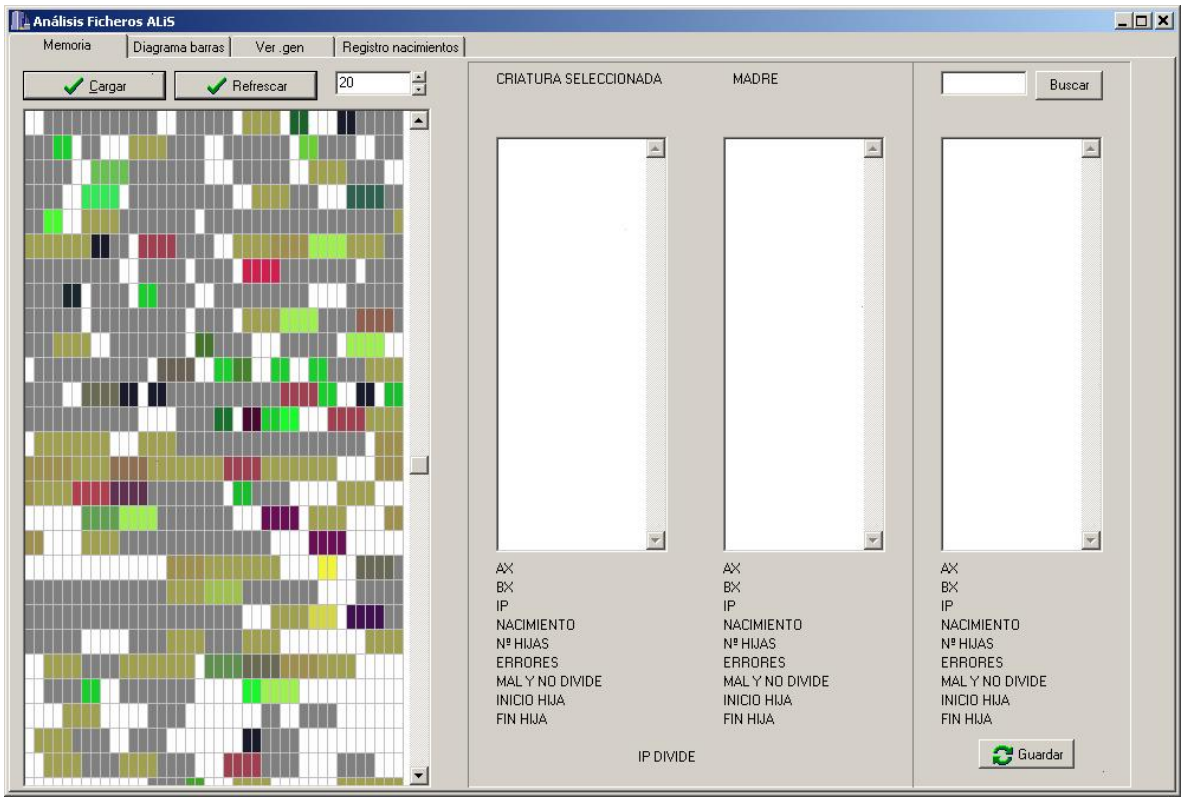
- diagramaBarras.txt: recoge la información mostrada en el interfaz gráfico en el diagrama de barras genotipos-número de criaturas. Así podremos conocer y examinar en ese instante exactamente el número de genotipos que existían y el número de criaturas de cada genotipo presentes en la sopa.

Con estos ficheros de texto tenemos información completa del sistema en el instante de ejecución en que se grabaron. Escogimos guardar la información en ficheros de texto ya que, en un primer momento, analizábamos éstos directamente abriendo los mismos con cualquier editor u hoja de cálculo. Así podíamos examinar la información tanto a nivel de cada criatura como a nivel global. Una vez que desarrollamos una serie de pautas que siempre seguíamos a la hora de analizar dichos ficheros, generamos una herramienta que nos permite analizar los

mismos de una forma mucho más cómoda para el usuario. Aún así, sigue siendo de ayuda examinar directamente los ficheros textuales ya que aportan a nivel general bastante información, como puede ser, ver a lo largo de los nacimientos la dirección de la instrucción *divide* que en concreto que se está ejecutando, observar a nivel general el relleno de la memoria, etc.

6.2. UTILIZACIÓN DEL SISTEMA

La aplicación para el análisis semántico de los resultados presenta el siguiente aspecto:



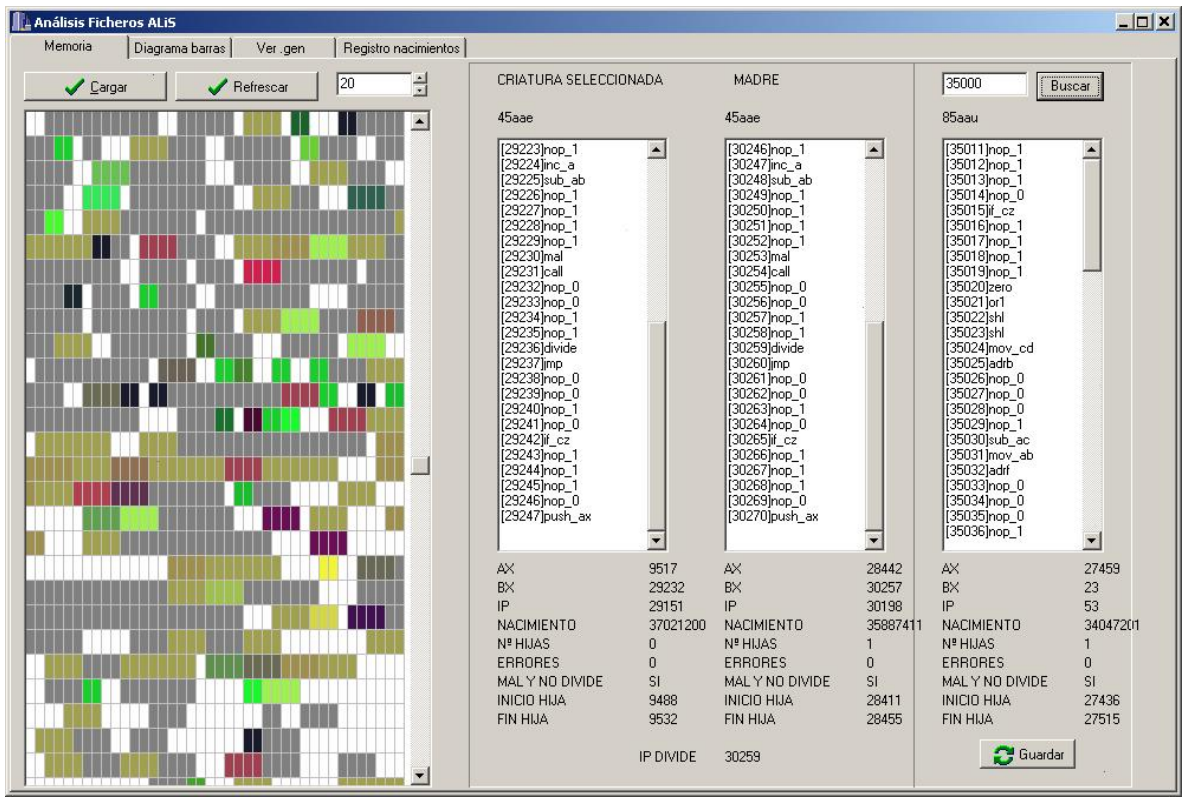
Disponemos de cuatro páginas en cada una de las cuales podremos analizar diferente información. La idea de esta aplicación es poder analizar entre las cuatro páginas toda la información que se recoge en los ficheros de texto que se guardan desde ALiS mediante el botón “Guardar”. En cada página se cargará un tipo de información, así que, para poder realizar el análisis, lo coherente es cargar los ficheros que se guardaron en el mismo instante de ejecución; es decir, los ficheros que están dentro del directorio generado por ALiS cuando se pulsó el botón "Guardar" en un instante de ejecución (número de instrucción en la ejecución del sistema).

A continuación describimos en detalle cada página.

6.2.1. PÁGINA *MEMORIA*

En la primera página de la aplicación podremos observar el contenido de la memoria y obtener la información de las criaturas presentes en la sopa.

El primer paso será cargar la información de los ficheros. Presionando el botón "Cargar" aparecerá una ventana de diálogo para la selección de archivos. En primer lugar seleccionaremos el fichero que recoge el contenido de la memoria: memoria.txt; una vez cargado, volverá aparecer el mismo diálogo, en esta ocasión se deberá seleccionar el fichero de lista de criaturas: listaDeCriaturas.txt. Cargados los dos ficheros podremos observar en la parte izquierda de la página algo parecido a esto:



El *grid* refleja el contenido de la memoria. Cada cuadrícula del *grid* corresponde a un fragmento de memoria de tamaño *zoom* número de instrucciones, donde *zoom* es el número que está recogido en el *edit* situado encima del *grid*. Este *zoom* se puede modificar en cualquier momento pulsando las flechas correspondientes para subir o bajar su valor, así como refrescar el contenido en pantalla con el *zoom* correspondiente mediante el botón "Refrescar". Con *zoom* igual a 1, obtendremos datos reales, cada recuadro corresponde a 1 dirección de memoria. Con *zoom* mayor que 1 se reduce la definición de las criaturas. Las cuadrículas se encuentran coloreadas según el genotipo de la criatura que ocupa las posiciones de memoria de esa cuadrícula. Cada color se corresponde con un genotipo distinto. De este modo, podemos ver la disposición de los tipos de criaturas en memoria así como observar los genotipos más numerosos, los menos,.... Las cuadrículas que aparecen en gris corresponden al espacio de memoria que ha sido reservado por criaturas mediante la

instrucción *mal* pero que todavía no pertenece a una criatura independiente. Las cuadrículas blancas son espacio de memoria que no pertenece a ninguna criatura.

Pinchando con el ratón en una cuadrícula podremos observar la información asociada a dicha cuadrícula.

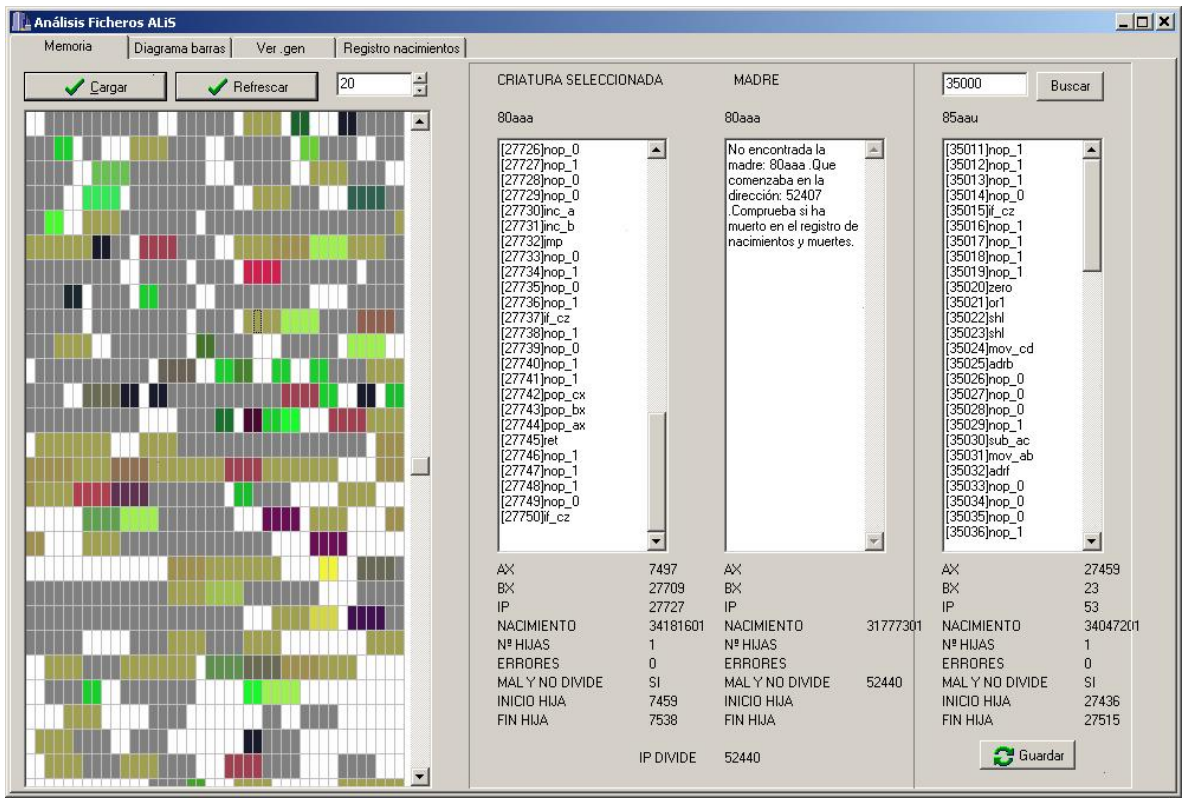
Si la cuadrícula corresponde a direcciones de memoria ocupadas por una criatura, en el espacio a la derecha del *grid* se mostrará toda la información de la criatura asociada y la información que se pueda extraer de la madre de dicha criatura.

Respecto a la información de la criatura podremos ver los campos que hemos descrito del fichero listaDeCriaturas.txt. y las instrucciones presentes en su espacio en memoria.

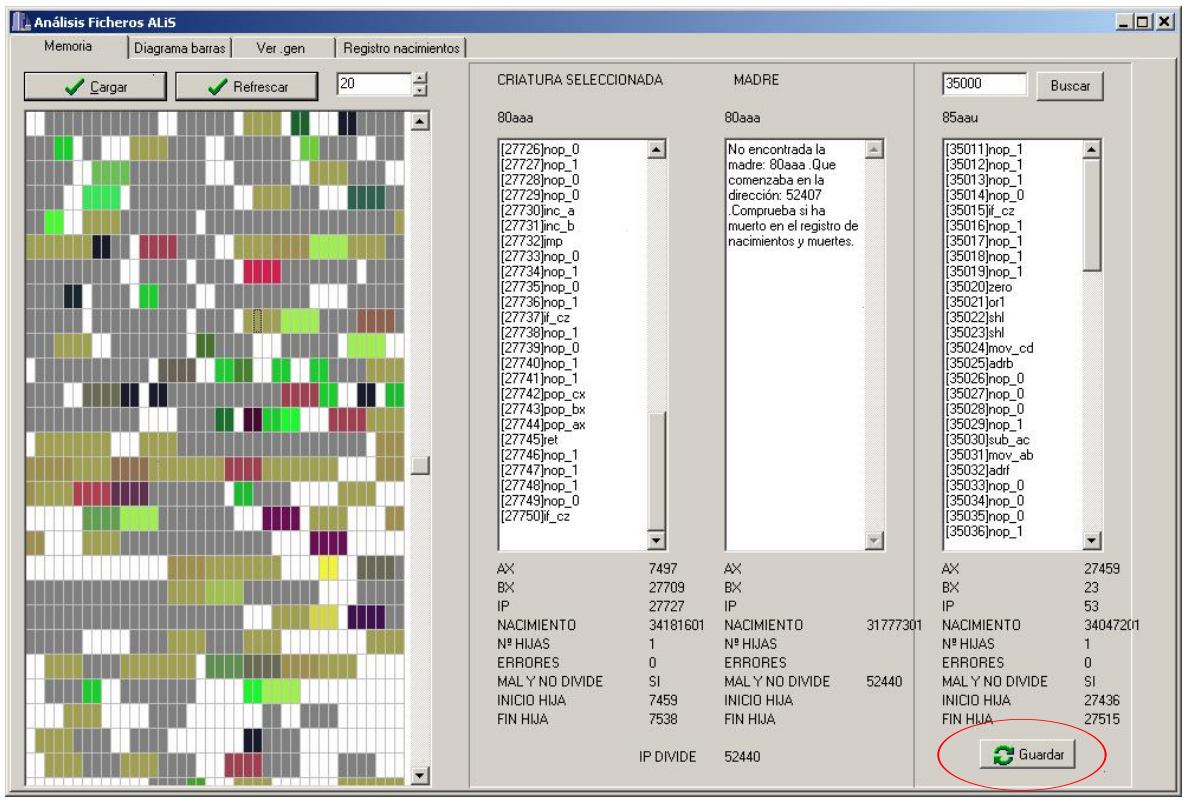
Respecto a la información de la madre, los datos de los que dispongamos dependerán de si ésta se encontraba viva en la sopa cuando se guardó la información o no. Si se encontraba en la sopa, podremos obtener para ella la misma información que para cualquier criatura. Si no se encontraba en la sopa, la madre había fallecido anteriormente y por ello, podremos ver algunos datos básicos: tamaño, genotipo, dirección de inicio, fecha de nacimiento y dirección de la instrucción *divide* que ejecutó para procrear a la criatura pinchada. Con esta información y otras páginas de la aplicación de análisis podremos obtener información más completa de la madre.

Pinchando en una cuadrícula gris, espacio reservado por una criatura mediante la instrucción *mal*, podremos ver el contenido de ese espacio de memoria y la criatura a la que pertenece, la que realizó *mal*.

Pinchando en una cuadrícula blanca, podremos ver el contenido de ese espacio de memoria, pero ningún dato más ya que no pertenece a ninguna criatura presente en la sopa.



En la parte derecha de la página podemos observar un espacio donde presentar información completa de una criatura. Dicha información se mostrará cuando se introduzca en el *edit* situado en la parte superior una dirección de memoria y se pulse el botón “Buscar”. Al hacerlo se mostrará, en caso de existir, la información de la criatura que posee la dirección de memoria introducida. Ésta es una forma de acceso directo en la búsqueda de criaturas en memoria. En este espacio de la página también disponemos de un botón “Guardar”. Pulsando este botón podremos guardar en un fichero de texto el genoma de la criatura que estemos visualizando en ese espacio en el formato de los ficheros .inoc para poder realizar inoculaciones en el sistema ALiS[70].



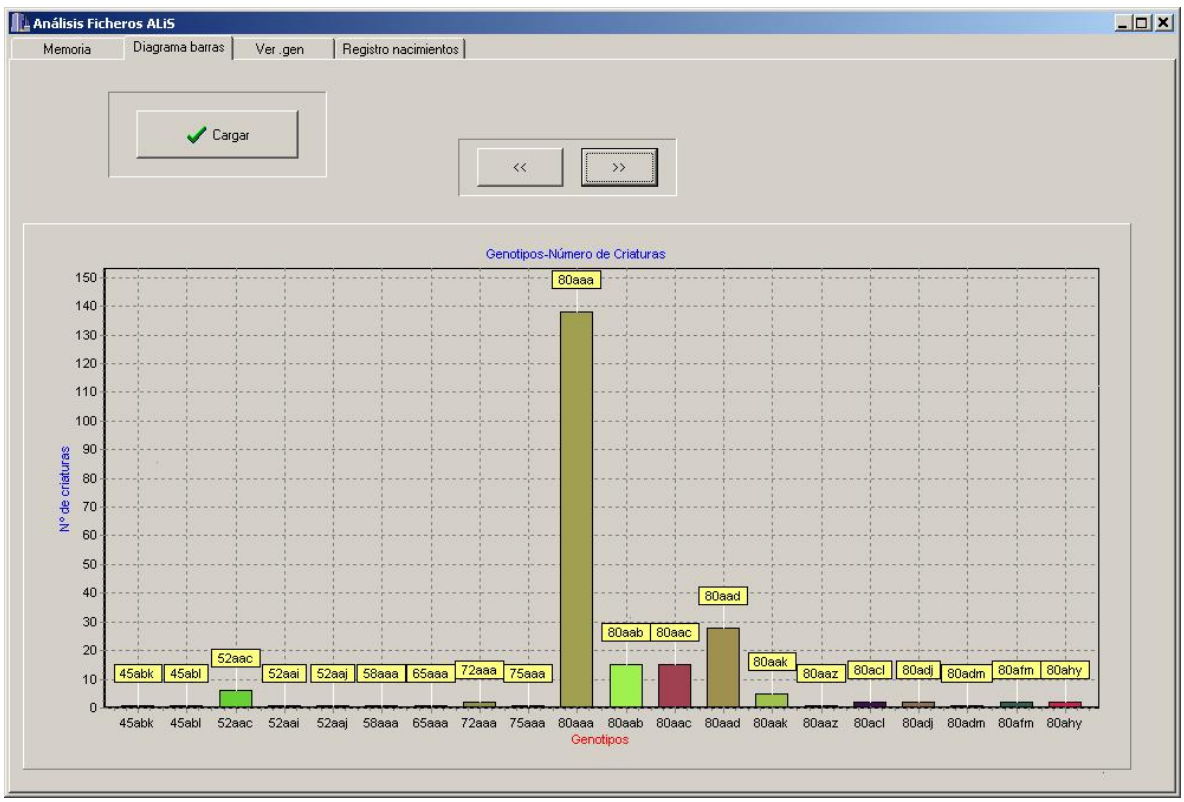
[70] [Ver apartado 4.2. TMemoria, sección de inoculaciones]

6.2.2. PÁGINA DIAGRAMA BARRAS

En esta página podemos cargar la información que se observa en el sistema ALiS dinámicamente a lo largo de su ejecución, pero para el instante determinado en que se guardó la información en el fichero diagramaBarras.txt que carguemos dentro de esta página.

Pulsando el botón “Cargar” podremos seleccionar el fichero de diagrama de barras deseado, pasando a cargar su contenido en el interfaz. La disposición del diagrama de barras es la misma que en el sistema ALiS: las abscisas recogen los diferentes genotipos y las ordenadas el número de criaturas de cada genotipo. La particularidad de este diagrama de barras es que cada barra se coloreará del color correspondiente al genotipo al que pertenece. Así, al ser este coloreado el mismo que para la página *Memoria* nos será de gran ayuda al realizar el estudio de la evolución de ALiS.

El diagrama de barras se ha dividido en páginas para poder ver un número máximo de barras a la vez en pantalla. Podremos cambiar de página mediante los botones situados por encima del diagrama de barras que contienen la flecha respectiva según el sentido de avance de la página.

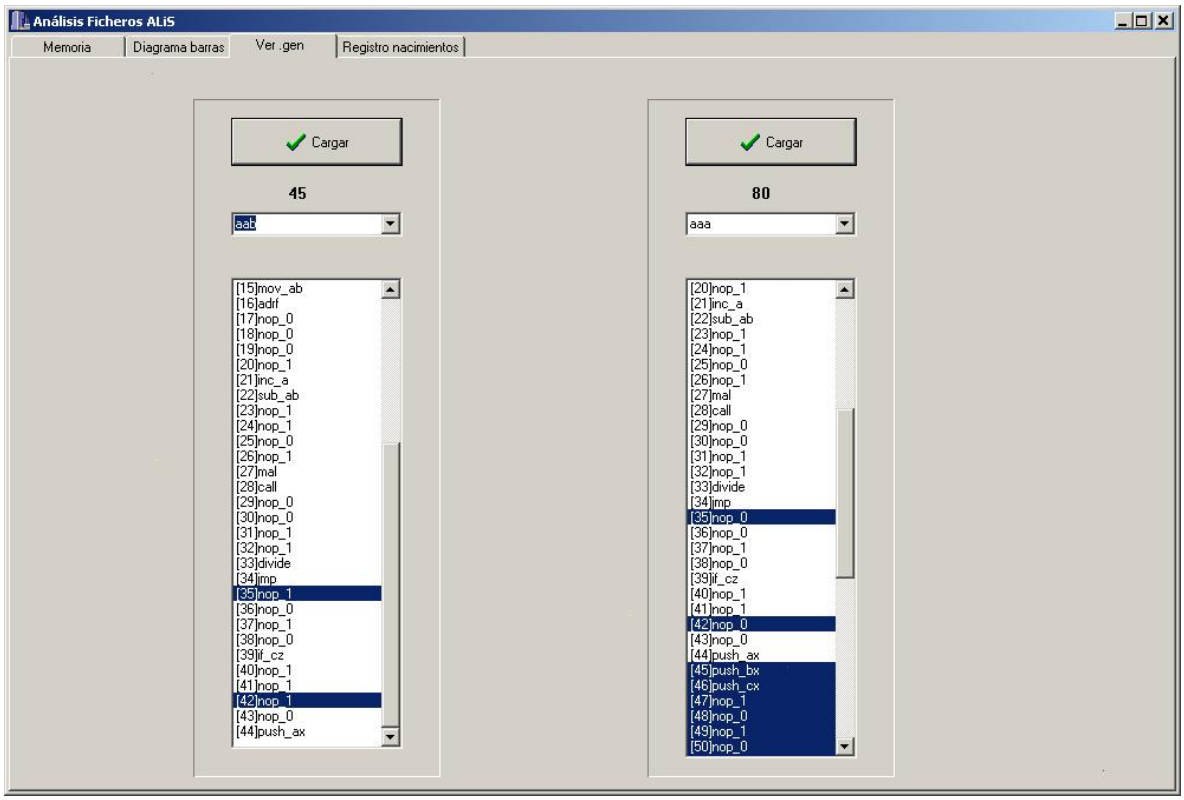


6.2.3. PÁGINA *VER .GEN*

Disponemos de esta página de la aplicación para poder ver el tipo de ficheros denominados .gen, generados por el banco de genes[71].

Disponemos de dos espacios duplicados: con botón de “Cargar”, desplegable y espacio de visualización. En cada espacio podremos cargar un fichero .gen distinto o el mismo fichero. Presionando el botón “Cargar”, una vez leído el fichero, podremos ver en la pestaña desplegable los diferentes genotipos que estuvieran contenidos en el fichero. Seleccionando uno de ellos, en la parte de visualización observaremos el conjunto de instrucciones (genoma) correspondiente al genotipo seleccionado.

Hemos decidido disponer de dos espacios en paralelo para poder realizar comparaciones de código (genoma). Así, al seleccionar en la pestaña un genotipo y cargar el genoma correspondiente, se mostrarán en un color distinto aquellas instrucciones en que difiera del genoma presente en el espacio de visualización opuesto.



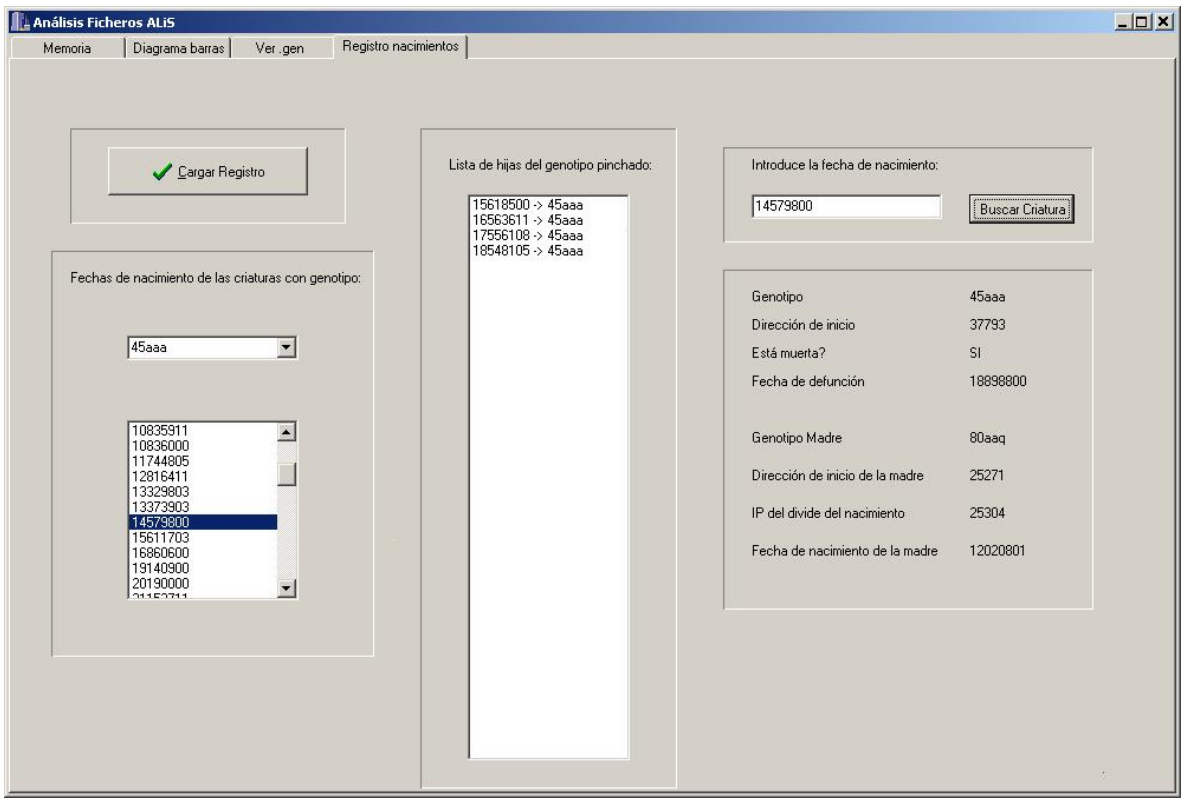
[71] [Ver apartado 4.10. TbancoGenes]

6.2.4. PÁGINA REGISTRO NACIMIENTOS

Mediante esta página podremos estudiar la información contenida en el fichero RegBirthDeath.txt. Como con otras páginas, presionando el botón “Cargar” y seleccionando el fichero de nacimientos y muertes que queramos analizar, dispondremos de la información en la aplicación de análisis.

Una vez cargado el archivo, seleccionaremos, del desplegable situado debajo, el genotipo que deseemos estudiar. Al pinchar, aparecerá en el espacio de visualización una lista de números que corresponden al instante de ejecución en que nació cada criatura con el genotipo seleccionado. Así, podemos ver la lista de criaturas de un genotipo que han aparecido en el sistema a lo largo de la ejecución. A su vez, pinchando sobre un número determinado podremos ver información en la parte central de la página. En este espacio se muestra la lista de las criaturas hijas que ha tenido la criatura concreta seleccionada. Podremos ver el instante de nacimiento de la hija y el genotipo de la misma en formato: *instante nacimiento* -> *genotipo*.

La parte derecha de la página está disponible para realizar búsquedas por fecha de nacimiento. Introduciendo una fecha (número de instrucción de ejecución del sistema) y pulsando el botón “Buscar Criatura” podremos ver una serie de datos sobre la criatura que nació en el instante indicado. Podremos conocer su genotipo y dirección de inicio, si sigue presente en el sistema o murió en algún instante y algunos datos de su madre que nos permite identificarla unívocamente: genotipo, dirección de inicio, fecha de nacimiento y dirección de la instrucción *divide* causante del nacimiento de la criatura que se buscó.



7.RESULTADOS

Una vez que el sistema fue completamente implementado, se comenzaron a realizar pruebas para comprobar su capacidad evolutiva. Para ello se probó inicialmente la reproducción de la criatura ancestral en un medio sin cambios, y posteriormente se fueron introduciendo en el sistema distintos tipos de mutaciones y otras variantes de comportamiento. Vamos a ver las distintas pruebas realizadas.

7.1. SISTEMA SIN EVOLUCIÓN

Inicialmente las pruebas del sistema se realizaron inoculando la sopa de 60.000 instrucciones con un único individuo de 80 instrucciones denominado "criatura ancestral"[72]. El paso del tiempo en el sistema se mide en términos de cuántas instrucciones han sido ejecutadas en la simulación. El ancestro ejecuta 839 instrucciones en su primera reproducción, y 813 en cada reproducción adicional. Dado que en estas primeras pruebas no se permite evolución, la célula ancestral se reproduce produciendo copias idénticas a ella misma.

De esta forma, la sopa se llena hasta que las células aparecidas y el espacio que tienen reservado para reproducirse ocupa el umbral establecido del 80% del total de la memoria. Al sobrepasar este nivel el reaper comienza a actuar, eliminando a las criaturas con más errores o más antiguas. En este caso en que no se han introducido elementos evolutivos, la ejecución de las criaturas es determinista, y por tanto ninguna de ellas acumulará errores. Así las primeras criaturas en desaparecer serán las primeras que nacieron, dejando libres las primeras posiciones de la memoria.

En este caso sin evolución, el sistema ejecuta aproximadamente unas 410.000 instrucciones antes de que la memoria se llene hasta el umbral mencionado anteriormente con unas 335 criaturas de genotipo 80aaa, sin contar con sus hijas en proceso de gestación. Una vez que el reaper comienza a eliminar criaturas, la memoria se mantiene ocupada alrededor de un 80%, y con un número de criaturas en la sopa alrededor de 335.

7.2. PRIMEROS TIPOS DE MUTACIONES

Las primeras mutaciones introducidas en el sistema fueron la mutación de fondo y la mutación de copia[73]. Se han probado las dos mutaciones juntas y por separado, y también con distintas inicializaciones de la memoria (sin inicializar, inicializada con instrucciones al azar, e inicializada con el código de la criatura ancestral). Los resultados se exponen a continuación.

[72] [Ver apartado 3.5. La Criatura Ancestral]
[73] [Ver apartado 3.6. Mutaciones]

7.2.1. MUTACIÓN DE FONDO AISLADA

La mutación de fondo cambia el contenido de una posición de memoria escogida aleatoriamente por una instrucción elegida al azar del conjunto de instrucciones válidas[74]. La posición de memoria a mutar se escoge usando el parámetro del sistema MUT_FONDO. Así, cada MUT_FONDO instrucciones ejecutadas se escoge aleatoriamente una de ellas. Cuando se llegue a la ejecución de ese número de instrucción, se elige al azar una posición de la memoria y se muta su contenido por una instrucción válida del conjunto de instrucciones.

7.2.1.1. PRUEBAS SIN INICIALIZAR LA MEMORIA

Inicialmente las pruebas se realizan sin inicializar la memoria de ninguna manera, dejando que cada posición contenga caracteres "basura" no válidos. Simplemente se introduce la criatura ancestral al inicio de la memoria antes de comenzar la ejecución.

El resultado de introducir esta mutación en el sistema es poco significativo. En las distintas ejecuciones realizadas, se producen mutaciones que dan lugar a la aparición de nuevos genotipos de tamaño 80 y otros tamaños de criaturas.

La aparición de nuevos genotipos de tamaño 80 se debe a dos causas principales. Por un lado, a la mutación de instrucciones en criaturas de genotipo 80aaa. En las pruebas realizadas no se ha obtenido ninguna nueva criatura de este tamaño con código válido para poder realizar satisfactoriamente la función de reproducción. Esto se debe a que es poco probable que una instrucción cambie por otra que produzca un comportamiento válido de la criatura, debido a que las instrucciones del conjunto de instrucciones se comportan, en general, de manera muy diferente. Por otro lado, hay ocasiones en que las criaturas de tamaño 80 no están rellenas de instrucciones válidas, debido a que el bucle de copia de sus madres no se ha ejecutado correctamente y no se han copiado instrucciones en su espacio de memoria.

En cuanto a la aparición de criaturas de distinto tamaño, la mayor parte de las veces se trata de criaturas de tamaño muy pequeño, como por ejemplo 1 o 4. Ninguna criatura de estos tamaños puede ser válida, y en la mayor parte de los casos se deben a mutaciones que estropean el funcionamiento de las criaturas de mayor tamaño de las que son hijas. El comportamiento de estas criaturas es el siguiente: como no tienen en su código suficientes instrucciones para un comportamiento relevante, su puntero de instrucción sigue leyendo y ejecutando instrucciones más allá de su espacio reservado, llegando generalmente al código de otra criatura válida, el cual comienzan a ejecutar.

En cierta medida podríamos considerar que este comportamiento es un tipo de comportamiento altruista. La criatura no tiene inconveniente en ejecutar código de otras pasando, en este caso, a reproducir otras criaturas. Pero en este caso esta forma de ejecución es demasiado común, consecuencia de la falta de ajuste del sistema.

[74] [Ver apartado 3.4.1. Conjunto de Instrucciones]

Los nuevos genotipos tardan en aparecer, ya que hasta que la sopa no está razonablemente llena es muy poca la probabilidad de que una mutación de fondo caiga en el genoma de una criatura.

7.2.1.2. PRUEBAS INICIALIZANDO LA MEMORIA CON INSTRUCCIONES ALEATORIAS

En este caso, antes de comenzar la ejecución, la memoria se inicializa rellenándola con instrucciones aleatorias para cada posición.

Los resultados obtenidos de esta forma son muy semejantes a los del apartado anterior, con la diferencia de que ahora ya no aparecen criaturas rellenas de instrucciones no válidas. Sin embargo, el comportamiento de estas criaturas no será correcto, ya que su secuencia de instrucciones será aleatoria y no tendrá sentido. No se puede esperar un comportamiento válido de ellas.

7.2.1.3. PRUEBAS INICIALIZANDO LA MEMORIA CON EL CÓDIGO DE LA CRIATURA ANCESTRAL

Dado que ni inicializando la memoria aleatoriamente ni sin inicializarla los resultados eran satisfactorios, se pensó que tal vez el sistema podría comportarse más correctamente si se rellenaba la memoria inicialmente con el código válido de la criatura ancestral.

De esta forma se obtienen resultados muy semejantes a los casos anteriores, con la diferencia de que no sólo las criaturas aparecen rellenas de instrucciones, sino que además este relleno suele contener trozos de código válido de la criatura ancestral. Esto provoca que las criaturas se reproduzcan en mayor número de forma correcta, y que aparezcan con menor frecuencia criaturas de tamaño muy pequeño y por tanto incorrecto.

Sin embargo, el sistema sigue sin producir evolución válida, ya que con sólo esta mutación de fondo es muy difícil que las criaturas muten de manera que tengan hijas con distintos tamaños y comportamientos válidos.

7.2.2. MUTACIÓN DE COPIA AISLADA

La mutación de copia hace que en ocasiones las instrucciones copiadas a la célula hija no sean las correspondientes a la célula madre[75]. La instrucción copiada que debe mutar se escoge usando el parámetro del sistema MUT_COPIA. Así, cada MUT_COPIA instrucciones ejecutadas se escoge aleatoriamente una de ellas. Cuando se vaya a copiar ese número de instrucción, se elige al azar otra instrucción válida del conjunto de instrucciones y se copia esa en lugar de la que corresponde.

[75] [Ver apartado 3.6. Mutaciones]

7.2.2.1. PRUEBAS SIN INICIALIZAR LA MEMORIA

Inicialmente las pruebas se realizan sin inicializar la memoria de ninguna manera, dejando que cada posición contenga caracteres "basura" no válidos. Simplemente se introduce la criatura ancestral al inicio de la memoria antes de comenzar la ejecución.

El resultado de esta mutación tampoco es significativo, aunque el rango de genotipos nuevos aparecidos es mayor que en el caso de sólo utilizar la mutación de fondo. En las distintas ejecuciones realizadas, se producen mutaciones que dan lugar a la aparición de nuevos genotipos de tamaño 80 y otros tamaños de criaturas.

La aparición de nuevos genotipos de tamaño 80 se debe a que cuando una criatura válida de este tamaño se reproduce y una de sus instrucciones copiadas muta, la criatura hija sigue teniendo el tamaño de la madre, pero distinto genotipo. Sin embargo, la mayor parte de estas mutaciones provoca que la nueva criatura no pueda realizar la reproducción de forma correcta, y por tanto vaya acumulando errores y finalmente sea eliminada por el reaper. Nuevamente, como en el caso de la mutación de fondo, es poco probable que una instrucción cambie por otra que produzca un comportamiento válido de la criatura, debido a que las instrucciones del conjunto de instrucciones se comportan, en general, de manera muy diferente. Por otro lado, hay ocasiones en que las criaturas de tamaño 80 no están rellenas de instrucciones válidas, debido a que el bucle de copia de sus madres no se ha ejecutado correctamente y no se han copiado instrucciones en su espacio de memoria.

En cuanto a la aparición de criaturas de distinto tamaño, la mayor parte de las veces se trata de criaturas de tamaño muy pequeño, por ejemplo 1, al igual que ocurría en el caso de la mutación de fondo. Ninguna criatura de este tamaño puede ser válida, y en la mayor parte de los casos se deben a mutaciones que estropean el funcionamiento de las criaturas de mayor tamaño de las que son hijas. El comportamiento de estas criaturas es el siguiente: como no tienen en su código suficientes instrucciones para un comportamiento relevante, su puntero de instrucción sigue leyendo y ejecutando instrucciones más allá de su espacio reservado, llegando generalmente al código de otra criatura válida, el cual comienzan a ejecutar.

De nuevo aparece un comportamiento altruista, pero, demasiado común entre las criaturas obtenidas, signo de que en el sistema se favorece en demasía esta evolución.

En estas pruebas se han obtenido cantidades significativas de criaturas de ciertos genotipos distintos del de la criatura ancestral, pero al analizar estas criaturas se ha descubierto que la mayor parte están vacías, es decir, no contienen instrucciones válidas. Por tanto, todas ellas serán hijas de otra criatura del mismo tamaño que debido a las mutaciones ha estropeado su código, y por tanto, su comportamiento.

7.2.2.2. PRUEBAS INICIALIZANDO LA MEMORIA CON INSTRUCCIONES ALEATORIAS

En este caso, antes de comenzar la ejecución, la memoria se inicializa rellenándola con instrucciones aleatorias para cada posición.

En general, los resultados obtenidos de esta forma son muy semejantes a los del apartado anterior, con la diferencia de que ahora ya no aparecen criaturas rellenas de instrucciones no válidas. A pesar de ello, el comportamiento de estas criaturas no será correcto, ya que su secuencia de instrucciones será aleatoria y no tendrá sentido. No se puede esperar un comportamiento válido de ellas.

Sin embargo, en una de las ejecuciones se obtuvo una criatura 76aaa que parecía reproducirse de forma correcta. Probando a comenzar las ejecuciones con esta criatura, y no con la ancestral, se comprobó que no sólo esta criatura se reproduce por sí misma, sino que también da lugar a evolución, aunque de forma un tanto diferente al resto de criaturas. Esta evolución acaba conduciendo a la aparición de criaturas de tamaño gigantesco (1292, por ejemplo), que no pueden reproducirse, y que además causan la desaparición del resto de criaturas.

Esta criatura 76aaa tiene el mismo genoma que la criatura ancestral 80aaa, con la diferencia de que carece de plantilla de inicio y que en la instrucción número 14 de la criatura ancestral tiene un *po*p_bx en lugar de *sub_ac*. Al comenzar una ejecución con esta criatura, una de las primeras operaciones que realiza es examinarse a sí misma para conocer su tamaño. Cuando busca su plantilla de inicio, no la encuentra y se produce un error, pero en el registro ax, que es donde se debería guardar la dirección correspondiente, se sigue almacenando el 0 inicial de la cpu. Además, en lugar de realizarse la operación *sub_ac* que resta del contenido de cx a ax, se realiza un *pop_bx* que desapila de la pila, y que en este momento no produce ningún error. De esta forma, la criatura sigue almacenando el valor 0 en el registro ax, que coincide con su inicio (al ser la primera criatura inoculada).

A pesar de esto, cuando la ejecución del sistema continúa, esta criatura no puede reproducirse tan correctamente como por ejemplo el ancestro, con lo que la evolución se produce de forma muy lenta y extraña. En general aparecen criaturas con tamaños múltiplos de 76, ya que como ninguna de las criaturas hereda de su madre una plantilla de inicio, según la posición de la memoria en la que se encuentren calcularán su tamaño desde el inicio de la memoria hasta su posición de final.

7.2.2.3. PRUEBAS INICIALIZANDO LA MEMORIA CON EL CÓDIGO DE LA CRIATURA ANCESTRAL

Como en el caso de la mutación aislada de fondo, se pensó que tal vez el sistema podría comportarse más correctamente si se rellenaba la memoria inicialmente con el código válido de la criatura ancestral.

De esta forma se obtienen resultados muy semejantes a los casos anteriores, con la diferencia de que no sólo las criaturas aparecen rellenas de instrucciones, sino que además

este relleno suele contener trozos de código válido de la criatura ancestral. Esto provoca que las criaturas se reproduzcan en mayor número de forma correcta, y que aparezcan con menor frecuencia criaturas de tamaño muy pequeño y por tanto incorrecto.

Sin embargo, el sistema sigue sin producir evolución válida, ya que con sólo esta mutación de copia es muy difícil que las criaturas muten de manera que tengan hijas con distintos tamaños y comportamientos válidos.

Tampoco iniciando la ejecución con la criatura 76aaa obtenida anteriormente se produce una evolución válida. Como en el apartado anterior, las criaturas evolucionadas que aparecen acaban desapareciendo de la sopa, dando a lugar a una única criatura de tamaño muy grande que no se puede reproducir.

7.2.3. Mutaciones de Fondo y de Copia

Una vez probadas las dos mutaciones por separado, se realizaron pruebas con ambas al mismo tiempo. Los resultados no difieren significativamente de los obtenidos por separado en los tres casos de inicialización de la memoria (sin inicialización, inicialización con instrucciones aleatorias, e inicialización con el código de la criatura ancestral).

En general, la evolución sigue sin ser correcta, apareciendo criaturas sin funcionalidad válida, que en el caso en que la memoria no está inicializada además no suelen contener instrucciones correctas.

En el caso de las ejecuciones iniciadas con la criatura 76aaa, los resultados también son los mismos obtenidos anteriormente.

7.2.4. Conclusiones

Tanto la mutación de fondo como la mutación de copia dan lugar en general a criaturas de tamaño 80 cuya secuencia de instrucciones se diferencia de la de su madre en una única instrucción. Esto se debe a que las mutaciones dentro de una criatura, ya sea al mutar aleatoriamente una posición de la memoria, o al mutar una instrucción que se está copiando, no pueden dar lugar a que esa criatura cambie de tamaño, sino que sólo pueden modificar el contenido de su genoma. Sin embargo, al alterarse su secuencia de instrucciones, las mutaciones pueden afectar al proceso por el que una criatura se examina a sí misma para calcular su tamaño, provocando potencialmente que tenga descendencia con tamaño diferente al suyo.

Probando las dos mutaciones de forma aislada, se comprueba que aunque ninguna de las dos produce una evolución válida, ya que casi todas las criaturas tienen un funcionamiento incorrecto, la mutación de copia produce mayor variedad de genotipos. Esto se debe a que esta mutación siempre se realiza sobre el código de criaturas, mientras que la de fondo se realiza sobre posiciones aleatorias de la memoria. De esta forma, es mucho menos probable

que la mutación de fondo se produzca sobre una criatura, lo que da lugar a la aparición de menos genotipos distintos.

Por otro lado, en ninguno de los dos casos supone una mejora significativa la inicialización de la memoria antes de comenzar la simulación, ya sea con instrucciones aleatorias o con el código de la criatura ancestral. En ambos casos, cualquiera de las dos inicializaciones provoca que las criaturas contengan instrucciones válidas, pero no por ello su comportamiento es mejor.

Finalmente, las pruebas con ambas mutaciones al mismo tiempo muestran unos resultados semejantes a las pruebas realizadas por separado. Esto se debe a que ambas mutaciones funcionan en ámbitos diferentes. La mutación de fondo tiene mucha menos frecuencia que la de copia, y además es más improbable que repercuta directamente en las criaturas. Sin embargo, la mutación de copia sí afecta directamente a la reproducción de las criaturas, y además se da en una proporción mayor. Por ello, el comportamiento de ambas mutaciones juntas es muy semejante al comportamiento del sistema sólo con la mutación de copia, además de que las pruebas realizadas con las mutaciones por separado muestran resultados semejantes en ambos casos.

7.3. MUTACIÓN DE COMPORTAMIENTO

Otro tipo de mutación que se introdujo más tarde en el sistema fue la mutación de comportamiento[76]. Se ha probado esta mutación por separado, y posteriormente junto a las mutaciones anteriores de fondo y copia, y también con distintas inicializaciones de la memoria (sin inicializar, inicializada con instrucciones al azar, e inicializada con el código de la criatura ancestral).

Además, inicialmente la mutación de comportamiento se consideró de forma que el comportamiento de las instrucciones cambiaba mucho en caso de mutación. A la vista de los resultados, se modificó de forma que el cambio en el comportamiento de las instrucciones no fuera tan grande.

Los resultados se exponen a continuación.

7.3.1. MUTACIÓN DE COMPORTAMIENTO AISLADA

La mutación de comportamiento provoca que las instrucciones del lenguaje LALiS no sean deterministas, es decir, que no se comporten siempre de la misma manera. Según el valor del parámetro del sistema MUT_COMP se comportarán correctamente o no. Este parámetro indica el tanto por ciento de probabilidad de que la instrucción no se ejecute

[76] [Ver apartado 3.6. Mutaciones]

correctamente. En caso de que se determine en la ejecución de una instrucción que ésta no se comportará como corresponde, el comportamiento de la misma variará de una forma u otra.

En general, son las instrucciones de más bajo nivel, las que manipulan la cpu de las criaturas, las que pueden presentar este tipo de mutación. Por ejemplo, si la instrucción *inc_a* sufre una mutación de comportamiento, en lugar de incrementar el valor del registro ax en 1, lo dejará tal y como está.

Las pruebas con sólo esta mutación y los tres tipos de inicialización de la memoria (sin inicializar, inicializando con instrucciones aleatorias, inicializando con el código de la criatura ancestral) han producido resultados semejantes. En los tres casos no se da una evolución válida, ya que aunque aparecen distintos genotipos de tamaño 80 y genotipos de otros tamaños, ninguna de las nuevas criaturas tiene capacidad de reproducción. Además, en el caso de no tener la memoria inicializada, la mayoría de las criaturas están vacías o presentan grandes huecos. Esto se debe a que la mutación repercute directamente en el comportamiento de la reproducción de las criaturas, haciendo que el proceso se lleve a cabo de forma incorrecta.

7.3.2. Mutación de Comportamiento con Mutación de fondo y copia

El comportamiento del sistema con los tres tipos de mutación activadas es una mezcla del comportamiento de cada una de ellas por separado. No se produce una evolución válida, ya que aunque aparecen distintos genotipos de distintos tamaños, las criaturas no pueden reproducirse, y en el caso en que la memoria no se inicializa, la mayoría de ellas no ha sido copiada correctamente y presenta grandes huecos sin instrucciones.

7.3.3. Conclusiones

Inicialmente, la mutación de comportamiento provocaba que la ejecución de las instrucciones variara demasiado cuando se producía la mutación. Por ejemplo, si la instrucción *inc_a* sufría una mutación de comportamiento, en lugar de incrementar el valor del registro ax en 1, lo incrementaba en un valor aleatorio entre 0 y 10. Esto provocaba que la reproducción de las criaturas fuera extraña, sobre todo cuando los valores de los registros eran los afectados. La consecuencia más común era encontrar grandes huecos vacíos en las criaturas hijas, debido a que los registros habían actualizado de forma incorrecta sus valores durante la copia del código de la madre al código de la hija. Por ello se modificaron las mutaciones de comportamiento de las instrucciones, haciendo que sus efectos no fueran tan radicalmente diferentes de la ejecución normal de la instrucción[77].

Finalmente, las pruebas de la mutación de comportamiento por separado, y junto a las mutaciones de fondo y copia, nos llevan a la conclusión de que es necesario seguir ajustando el sistema para conseguir evoluciones correctas, en las que las criaturas con nuevos genotipos

[77] [Ver apartado 3.6. Mutaciones]

que vayan apareciendo sean funcionalmente correctas y capaces de reproducirse, además de presentar un genoma suficientemente completo.

7.4. Restricciones de Tamaño de Criatura

En las pruebas realizadas hasta este momento, había ocasiones en que aparecían criaturas de tamaño muy grande (a partir de 1000 instrucciones), que si no se podían reproducir satisfactoriamente, o si pretendían tener hijas todavía más grandes que ellas, provocaban la eliminación de la mayor parte de las criaturas de la sopa, y por tanto causaban una evolución fallida.

En otras ocasiones, el problema era la aparición de criaturas de tamaño demasiado pequeño (1, 2, 4, ...) que no podían presentar funcionalidad válida en su código. Además estas criaturas no presentaban gran número de fallos por la poca probabilidad de que se produjera un error en sus contadas instrucciones. De esta forma, estas criaturas permanecían en la sopa indefinidamente, mientras que criaturas más "útiles" morían.

Por todo esto, se pensó en establecer un límite de tamaño para las criaturas, tanto mínimo como máximo. Así se introdujeron en el sistema los parámetros TAM_MIN_CRIA y TAM_MAX_CRIA[78]. Cuando una criatura solicita espacio para su hija, se realiza una comprobación usando estos dos parámetros, y en caso de que el tamaño solicitado sea estrictamente menor que TAM_MIN_CRIA, o estrictamente mayor que TAM_MAX_CRIA, se produce un error en la instrucción y no se concede espacio a la criatura.

A continuación, vamos a volver a estudiar el efecto de las mutaciones anteriormente mencionadas por separado y en conjunto.

7.4.1. Mutación de Fondo

Las pruebas realizadas utilizando sólo la mutación de fondo dan los mismos resultados que antes de añadir la restricción de tamaño sobre las criaturas. Las únicas diferencias son que la evolución se produce de forma más lenta al restringir el tamaño de las criaturas válidas, y que en la mayor parte de los casos las criaturas ya no aparecen vacías, sino que tienen su genoma relleno de instrucciones válidas. Tampoco se ha detectado ninguna diferencia significativa en las pruebas al realizar las tres inicializaciones posibles de la memoria (sin inicializar, inicialización con instrucciones aleatorias e inicialización con el contenido de la criatura inicial).

7.4.2. Mutación de Copia

[78] [Ver apartado 3.8. Parámetros del Sistema]

Las pruebas realizadas utilizando sólo la mutación de copia presentan unos resultados muy semejantes a los obtenidos antes de añadir los límites en el tamaño de las criaturas válidas, aunque en el caso en que no se inicializa la memoria las criaturas aparecen rellenas de instrucciones válidas, no como antes de añadir las restricciones de tamaño de las criaturas. Tampoco se ha detectado ninguna diferencia significativa en las pruebas al realizar las tres inicializaciones posibles de la memoria (sin inicializar, inicialización con instrucciones aleatorias e inicialización con el contenido de la criatura inicial).

7.4.3. Mutación de Comportamiento

Las pruebas con sólo esta mutación y los tres tipos de inicialización de la memoria (sin inicializar, inicializando con instrucciones aleatorias, inicializando con el código de la criatura ancestral) han producido resultados semejantes a los obtenidos antes de introducir el límite de tamaño en las criaturas. En los tres casos no se da una evolución válida, ya que aunque aparecen distintos genotipos de tamaño 80 y genotipos de otros tamaños, ninguna de las nuevas criaturas tiene capacidad de reproducción. Además, en el caso de no tener la memoria inicializada, la mayoría de las criaturas están vacías o presentan grandes huecos.

7.4.4. Mutaciones de Fondo, Copia y Comportamiento

El comportamiento del sistema con los tres tipos de mutación activadas es una mezcla del comportamiento de cada una de ellas por separado. No se produce una evolución válida, ya que aunque aparecen distintos genotipos de distintos tamaños, las criaturas no pueden reproducirse, y en el caso en que la memoria no se inicializa, la mayoría de ellas no ha sido copiada correctamente y presenta grandes huecos sin instrucciones.

En la mayoría de los casos la mayor parte de los nuevos genotipos aparecidos son del mismo tamaño que la criatura inicial, y en casi todas las ejecuciones no suele aparecer nada más que una criatura de cada genotipo. Esto se debe a que las mutaciones provocan cambios en el genotipo de las criaturas, pero no criaturas válidas capaces de reproducirse por sí mismas.

7.4.5. Conclusiones

La introducción de unos límites máximo y mínimo para el tamaño de las criaturas mejora la evolución del sistema en el sentido de que se impide la aparición de criaturas demasiado pequeñas para que tengan funcionalidad correcta, o demasiado grandes que llenen la sopa y hagan que el resto de las criaturas más prometedoras desaparezcan.

Sin embargo, el comportamiento general del sistema es el mismo que antes de introducir estos límites. En general las evoluciones producidas no son válidas, en el sentido de

que no aparecen demasiados genotipos y tamaños distintos, y que la mayor parte de las criaturas no presentan funcionalidad válida ni capacidad de reproducción. Por tanto, será necesario seguir ajustando el sistema para conseguir que se produzcan evoluciones correctas, en las que las criaturas con nuevos genotipos que vayan apareciendo sean funcionalmente correctas y capaces de reproducirse, además de presentar un genoma suficientemente completo.

7.5. MUTACIÓN DE PLANTILLA

En vista de los resultados anteriores, se puede sacar como conclusión que es extremadamente difícil que las mutaciones introducidas hasta ahora en el sistema den lugar a genotipos de criaturas con un código funcionalmente válido y con capacidad de reproducción.

El estudio de las criaturas que surgen a lo largo de las distintas ejecuciones del sistema se basa en gran medida en la comparación de su genoma y comportamiento con los de la criatura ancestral. El ancestro es una criatura artificial escrita pensando en un comportamiento determinado, y que presenta dos funciones básicas: la determinación del tamaño de la criatura, y la reproducción usando un bucle de copia de instrucciones de la madre a la hija. En cierto modo, la funcionalidad de la criatura ancestral se encuentra en trozos de código delimitados por plantillas, que son las que permiten el direccionamiento dentro de la sopa.

Partiendo de estas conclusiones, se decidió introducir un nuevo tipo de mutación que sólo actúa en las plantillas de una criatura que se está copiando. De esta forma, cuando una criatura va a ejecutar un *divide*, realiza una mutación sobre una de las instrucciones *no-*operaci*ón* de alguna de las plantillas de la célula hija cambiando el *nop* correspondiente por su complementario.

Este tipo de mutación es prometedora porque actúa directamente sobre las características que presentas más potencial evolutivo, tales como la determinación del propio tamaño o la copia de instrucciones a la célula hija, entre otros.

Esta mutación se realiza en una proporción determinada por el parámetro del sistema PROP_MUT_PLANT[79], que indica cada cuantas ejecuciones de instrucciones *divide* se realiza la mutación.

[79] [Ver apartado 3.8. Parámetros del Sistema]

7.5.1. Mutación de Plantilla Aislada

Las pruebas realizadas usando sólo la mutación de plantilla producen evoluciones bastante prometedoras en principio, ya que aparecen genotipos de números más variados que los encontrados en las pruebas anteriores.

Sin embargo, a pesar de que las nuevas criaturas deberían presentar trozos de funcionalidad de la criatura ancestral, esto en general no ocurre. En el caso en que no se inicializa la memoria antes de comenzar la ejecución, las mutaciones en las plantillas provocan que las criaturas resultantes no sean capaces de reproducirse adecuadamente y tengan hijas vacías. Cuando la memoria se inicializa antes de la ejecución, ya sea aleatoriamente o con el código de la criatura inicial, las criaturas aparecidas contienen instrucciones válidas en su genoma, pero su comportamiento no es correcto porque este código no es el que les ha transmitido la madre, sino el que se encontraba inicialmente en la memoria.

7.5.2. Mutaciones de Plantilla y Mutación de Fondo

Las pruebas realizadas usando las mutación de plantilla junto con la mutación de fondo producen resultados muy semejantes a los obtenidos sólo con la mutación de fondo, aunque también presentan características propias del uso de la mutación de fondo por separado.

Aparecen en las ejecuciones genotipos y tamaños variados, pero no se produce una evolución correcta. Cuando la memoria no se inicializa al empezar la simulación, la mayor parte de las criaturas aparecidas no contienen código válido, y cuando se inicializa, ya sea con instrucciones aleatorias o con el código de la criatura inicial, el comportamiento no suele ser el deseado.

7.5.3. Mutaciones de Plantilla, Fondo y Copia

Las pruebas realizadas con las mutaciones de plantilla, fondo y copia no presentan resultados significativamente diferentes a los anteriormente mencionados. En general, aparecen bastantes tamaños y genotipos nuevos, que en la mayor parte de los casos no contienen código, si no se ha inicializado la memoria, ni comportamiento válidos, si la memoria ha sido inicializada bien aleatoriamente o bien con el código de la criatura inicial.

7.5.4. Mutaciones de Plantilla, Fondo, Copia y Comportamiento

Los resultados utilizando al mismo tiempo las cuatro mutaciones posibles siguen presentando los mismos resultados que usándolas por partes o de forma aislada. No se producen evoluciones válidas, presentando las criaturas aparecidas un comportamiento no válido y un código vacío en el caso de que no se inicialice la memoria antes de comenzar la ejecución.

7.5.5. Conclusiones

A pesar de que la diversidad de genotipos y tamaños aumenta al utilizar la nueva mutación de plantilla, se sigue dando el problema de que las criaturas no se copian correctamente si el comportamiento de su madre está muy dañado. Esto provoca la proliferación de criaturas vacías, o en el caso de que la memoria haya sido inicializada, bien con instrucciones escogidas aleatoriamente o bien con el código de la criatura inicial, criaturas rellenas con el código que había a priori en el espacio de memoria que se les ha asignado.

7.6. Proporción de Relleno para una Criatura

En las pruebas realizadas hasta aquí, se puede observar que el mayor problema que presentan las criaturas obtenidas, y que provoca evoluciones no válidas, es que cuando una célula con comportamiento incorrecto se reproduce, en la mayor parte de los casos su bucle de copia no se realiza de forma correcta y la célula resultante aparece vacía o inicializada con las instrucciones presentes inicialmente en la memoria.

Este problema llevó a pensar que se podría mejorar el comportamiento del sistema si se obligara a que las criaturas estuvieran rellenas con el código de la madre en una cierta proporción. Para ello se añadió un parámetro PROP_MOV[80], que indica el tanto por ciento de la criatura hija que debe estar rellena por la madre antes de ejecutar el *divide* correspondiente.

Para realizar el cálculo del porcentaje de criatura que ha sido copiada, se lleva la cuenta del número de instrucciones *mov_iab* ejecutadas por la madre. Cada vez que se ejecuta esta instrucción, la madre copia el contenido de una posición de memoria al espacio de la hija. La instrucción copiada no tiene por qué pertenecer obligatoriamente a la célula madre, ya que las mutaciones pueden provocar que una criatura transmita un genoma distinto del suyo a sus hijas. Si en el momento en que la célula madre desea ejecutar un *divide*, la proporción de código copiado a la hija no es correcta, se produce un error y no se realiza el *divide*.

Veamos ahora los comportamientos más relevantes del sistema combinando de distintas formas los tipos de mutaciones disponibles.

[80] [Ver apartado 3.8. Parámetros del sistema]

7.6.1. Mutación de Fondo

Las pruebas realizadas con esta nueva característica y usando únicamente la mutación de fondo, muestran que las evoluciones se producen más lentamente, ya que el sistema no permite la aparición de criaturas con su genoma vacío.

Y por fin, se producen las primeras evoluciones válidas. En los primeros miles de instrucciones la diversidad aparecida es menor que en pruebas anteriores, pero se ha comprobado que acaban apareciendo criaturas funcionalmente correctas que se reproducen apropiadamente, teniendo hijas siempre del mismo tamaño y genotipo, que no tienen por qué ser los suyos propios si se ha producido alguna mutación en su genoma.

7.6.2. Mutación de Plantilla

Las pruebas realizadas con esta nueva característica y usando únicamente la mutación de plantilla también muestran evoluciones válidas. Aparecen criaturas de distintos tamaños y genotipos, que siempre tienen su genoma lleno de instrucciones copiadas por la madre, y que en algunos casos presentan funcionalidad válida y capacidad de reproducción.

Aunque las criaturas de tamaño 80 siguen dominando la sopa, los nuevos tamaños más comunes son 28, 45, 52, 85 y 93. En las pruebas realizadas se han encontrado criaturas de tamaño 45 y 85 capaces de reproducirse y de transmitir su genoma a sus descendientes.

Una de las criaturas de tamaño 85 capaces de reproducirse es la criatura 85aaa. Esta criatura contiene el código de la criatura ancestral con dos diferencias principales. En primer lugar, a su inicio tiene 5 instrucciones más, que corresponden a las 5 últimas instrucciones de la criatura ancestral. En segundo lugar, la plantilla que sigue al *adrb* que se usa para determinar la dirección de inicio de la criatura ha mutado, con tal suerte que su complementaria ya no es la plantilla de inicio de la ancestral, sino casualmente la plantilla de fin que se ha copiado al inicio de su genoma. De esta forma, esta criatura calcula correctamente su dirección de inicio y su tamaño, y puede transmitir correctamente su genoma a sus descendientes. Además, en algunas de estas pruebas se produjo por primera vez la aparición de una de las relaciones más comunes en la naturaleza: el parasitismo. Se encontró una criatura 45aab que tenía hijas de su mismo genotipo, pero que no contenía el bucle de copia, ya que estaba formada sólo por las 45 primeras instrucciones de la criatura ancestral. Por tanto, esta criatura no puede reproducirse por ella misma. Gracias a la membrana semi-permeable de las criaturas, que permite la lectura y ejecución de cualquier código de la sopa, las criaturas pueden encontrar las plantilla complementarias buscadas en el genoma de otras, e incluso pueden ejecutar este código ajeno. De esta forma, si una criatura 45aab aparece mezclada con criaturas de genotipo 80aaa, cuando intenta llamar al procedimiento de copia no encontrará la plantilla correspondiente en su propio genoma, pero sí en el de otras criaturas 80aaa si se encuentran dentro del límite de búsqueda. Así, la criatura 45aab llevará su puntero de instrucción al procedimiento de copia de la 80aaa y lo ejecutará copiando su genoma en el de la hija. Para la reserva de memoria vuelve a su propio genoma, pero para el bucle de

reproducción siempre emplea el código de la criatura 80aaa. Por tanto, la criatura 45aab es un parásito de la 80aaa, ya que se reproduce gracias a ella.

7.6.3. Mutación de Fondo, Copia, Comportamiento y Plantilla

Las pruebas realizadas con esta nueva característica y usando las cuatro mutaciones posibles, han dado lugar a evoluciones válidas en las que aparecen gran variedad de genotipos y de tamaños de criatura: 25, 28, 43, 44, y muchos más. Al igual que en apartado anterior, algunas de las criaturas aparecidas presentas funcionalidades válidas y son capaces de reproducirse, transmitiendo su genoma a sus descendientes.

7.6.4. Conclusiones

Al introducir este nuevo límite del porcentaje que se debe rellenar de una criatura para que ésta sea válida, se ha conseguido ajustar el sistema lo suficiente para que se produzcan por fin evoluciones correctas. Llegados a este punto, es necesario realizar muchas pruebas variando los parámetros secundarios, que tienen menos repercusión en los resultados, y estudiar los resultados y el comportamiento de las criaturas representativas aparecidas.

7.7. Factor de Ejecución Proporcional al Tamaño de las Criaturas

Tras las primeras evoluciones satisfactorias, se consideró que podía ser relevante considerar que el entorno pueda proporcionar los turnos de tiempo a las criaturas según su tamaño, dando prioridad a las más pequeñas, a las más grandes o a ambas por igual. Para conseguir esto se introdujo en el sistema el parámetro FACTOR_SLICE[81].

[81] [Ver apartado 3.8. Parámetros del Sistema]

Ahora, en lugar de que los tiempos de cpu concedidos por el sistema operativo sean constantes, dependen del tamaño de la criatura según la siguiente regla:

$$time\_slice = TIEMPO\_VIDA * tamaño^{FACTOR\_SLICE}$$

donde los distintos parámetros tienen el siguiente significado:

- `time_slice`. Número de instrucciones que se asignarán a la criatura en el fragmento de tiempo.
- `TIEMPO_VIDA`. Parámetro del sistema que indica el número de instrucciones que se asignan a una criatura en el caso neutral.
- `tamaño`. Tamaño de la criatura que se va a ejecutar.
- `FACTOR_SLICE`. Parámetro del sistema introducido en este apartado.

De esta forma, si FACTOR_SLICE es igual a 0, se asignarán los fragmentos de cpu independientemente del tamaño de las criaturas. Si FACTOR_SLICE es mayor que 0, las criaturas más grandes serán favorecidas y tendrán más tiempo para ejecutarse, siempre proporcionalmente a su tamaño. Si FACTOR_SLICE es menor que 0, las más favorecidas serán las criaturas pequeñas.

Veamos ahora los comportamientos más relevantes del sistema con este factor.

7.7.1. Cualquier Tipo De Mutación

El FACTOR_SLICE influye en el comportamiento del sistema de una manera similar sean cuales sean las mutaciones introducidas en él. Aunque, como se ha explicado, según el valor de este factor se beneficia a las criaturas grandes o pequeñas dándoles más o menos tiempo de ejecución, la influencia no es apreciable directamente.

Con un FACTOR_SLICE negativo se beneficia a las criaturas pequeñas, pero dadas las pruebas realizadas a partir de la criatura ancestral de tamaño 80, esta influencia no es inmediata. El hecho de que la criatura introducida en primer lugar en la sopa sea la 80aaa provoca que desde los primeros miles de instrucciones este tipo de criaturas se procreen siendo, inmediatamente, las criaturas más abundantes en la sopa. Así, aunque aparezcan criaturas más pequeñas, lograr para éstas una presencia notable lleva varios millones de instrucciones. Lo que ocurre es que con un FACTOR_SLICE negativo se logra que se alcance un número significativo antes que con un FACTOR_SLICE igual a 0, por ejemplo.

Si el FACTOR_SLICE es positivo, el hecho de aparecer y lograr presencia para las criaturas pequeñas es algo muy improbable en los primeros millones de intrucciones. A pesar de ello, llegarán a aparecer.

7.8. OTRAS EJECUCIONES

7.8.1. EVOLUCIÓN DEL SISTEMA EN EJECUCIONES PROLONGADAS

Tras todas las pruebas realizadas combinando diferentes parámetros del sistema y proporciones de las diferentes mutaciones, hemos realizado una serie de pruebas más largas para estudiar la evolución del sistema en un periodo largo de tiempo, llegando a los 200 millones de instrucciones ejecutadas. Para dichas pruebas hemos configurado el sistema con los parámetros que han resultado mejores para la obtención de evolución:

- VIDA_CELULA 100 y FACTOR_SLICE –0.5 : favorecemos la evolución de criaturas más pequeñas. De esta forma el crecimiento del número de parásitos en la sopa es mayor.
- No habrá mutación de fondo ni de comportamiento.
- Para la mutación de copia, MUT_COPIA será 10000
- Para la mutación de plantilla, MUT_PLANT será 2. Así incentivamos que la mutación de plantilla aparezca en proporción mayor que la mutación de cualquier otra instrucción en la copia.
- El resto de los valores de los parámetros son los que se han comentado que toman por defecto.[82]

Bajo estas condiciones el comportamiento general que hemos obtenido es el siguiente:

En los primeros 20 millones de instrucciones no aparecen muchas criaturas pequeñas. Las criaturas 80aaa se reproducen de manera continua, siendo las criaturas dominantes en la sopa conservando más de 100 ejemplares presentes en memoria en todo momento. Como consecuencia de esta proliferación de criaturas ancestrales de tamaño 80 y de las mutaciones introducidas aparecen una gran cantidad de criaturas de tamaño 80 diferenciadas de la ancestral en una o dos instrucciones, pero cuyo genoma no tiene capacidad para autorreplicarse, y por ello, sólo aparece 1 criatura de cada ejemplar.

En torno a los 30 millones de instrucciones comienzan a aparecer parásitos. Hemos detectado 3 parásitos distintos de tamaño 45:

[82] [Ver apartado 3.8. Parámetros del Sistema]

45aab, el parásito prototípico nombrado por Thomas S. Ray.

```
nop_1           nop_0           nop_1           nop_0
nop_1           nop_0           nop_0           nop_1
nop_1           sub_ac          nop_1           nop_0
nop_1           mov_ab          mal             if_cz
zero            adrf            call            nop_1
or1             nop_0           nop_0           nop_1
shl             nop_0           nop_0           nop_1
shl             nop_0           nop_1           nop_0
mov_cd          nop_1           nop_1           push_ax
adrb            inc_a           divide
nop_0           sub_ab          jmp
nop_0           nop_1           nop_0
```

45aaf,

```
nop_1           nop_0           nop_1           nop_0
nop_1           nop_0           nop_1           nop_1
nop_1           sub_ac          nop_1           nop_0
nop_1           mov_ab          mal             if_cz
zero            adrf            call            nop_1
or1             nop_0           nop_0           nop_1
shl             nop_0           nop_0           nop_1
shl             nop_0           nop_1           nop_0
mov_cd          nop_1           nop_1           push_ax
adrb            inc_a           divide
nop_0           sub_ab          jmp
nop_0           nop_1           nop_0
```

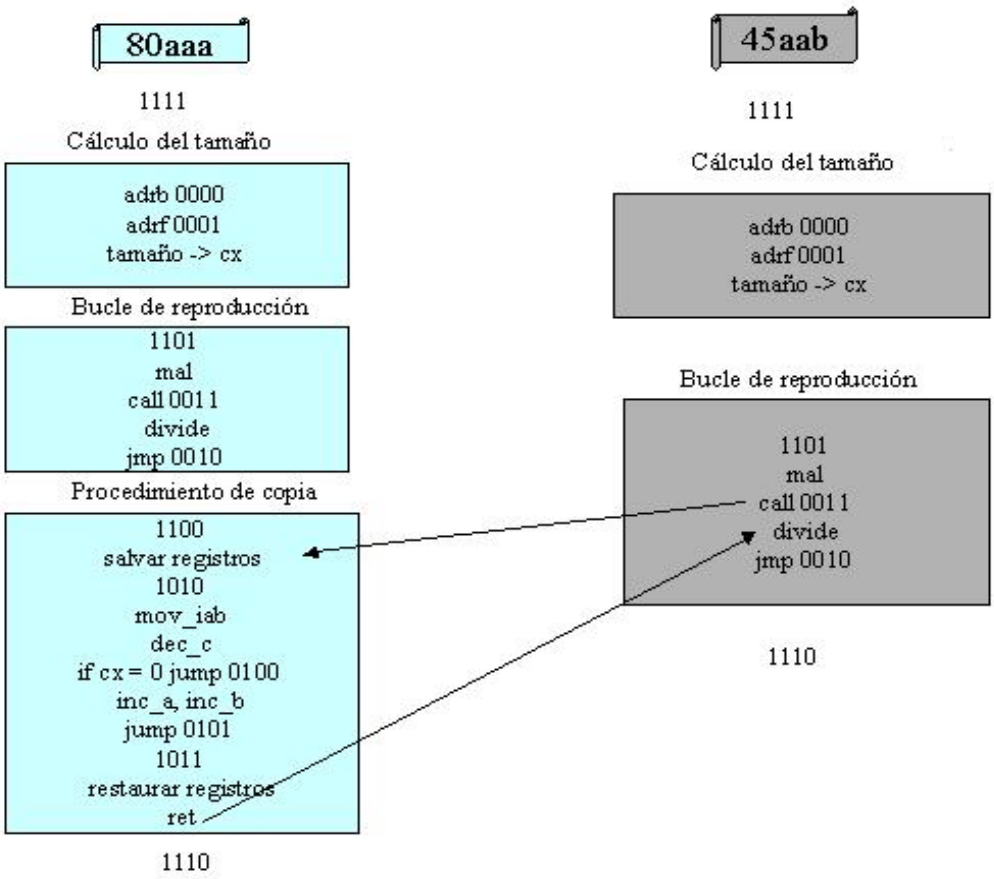
45abn,

```
nop_1           nop_0           nop_1           nop_0
nop_1           nop_0           nop_1           nop_1
nop_1           sub_ac          nop_1           nop_0
nop_1           mov_ab          mal             if_cz
zero            adrf            call            nop_1
or1             nop_0           nop_0           nop_1
shl             nop_0           nop_0           nop_1
shl             nop_0           nop_1           nop_0
mov_cd          nop_1           nop_1           push_ax
adrb            inc_a           divide
nop_0           sub_ab          jmp
nop_0           nop_0           nop_0
```

Estas criaturas se diferencian entre sí en la plantilla que sigue a la instrucción *sub_ab*. Todas ellas, en su primera ejecución, encuentran que la llamada a *call* no dispone de plantilla destino complementaria a la 0011 dentro de la propia criatura, por lo que el puntero de instrucción (IP) saltará a otra parte de la memoria no perteneciente a la criatura de tamaño 45. Esta llamada a *call* encaja perfectamente con el fragmento de código de las criaturas de tamaño 80, las 80aaa, exactamente con el bucle de copia de éstas. Así, tras ejecutar ese bucle de copia (fragmento de código ajeno) la ejecución de la instrucción *ret* provoca que el puntero de instrucción vuelva a apuntar a la criatura de tamaño 45 original pasando a ejecutar dentro de su espacio de memoria la instrucción *divide*. En este momento, cada parásito de los descritos avanza en su ejecución por un lado. El parásito prototípico, 45aab, salta de nuevo a su propia instrucción *mal* volviendo a ejecutar la secuencia de pasos descrita. Pero los otros

parásitos no pueden proceder así ya que la plantilla complementaria a la 0010 (posterior al *jmp*) no se encuentra dentro del código de la propia criatura. De nuevo será la criatura de tamaño 80 la que disponga de una plantilla complementaria y código con el que el parásito puede seguir ejecutando el bucle de copia.

La presencia de estos parásitos comienza a aumentar en la sopa, pero siempre el predominante es el parásito prototípico 45aab. La explicación que encontramos a este fenómeno es que es el parásito que emplea menor cantidad de código ajeno (número de instrucciones) y por ello, código menos susceptible a mutaciones que puedan alterar el funcionamiento perfecto del bucle de copia.

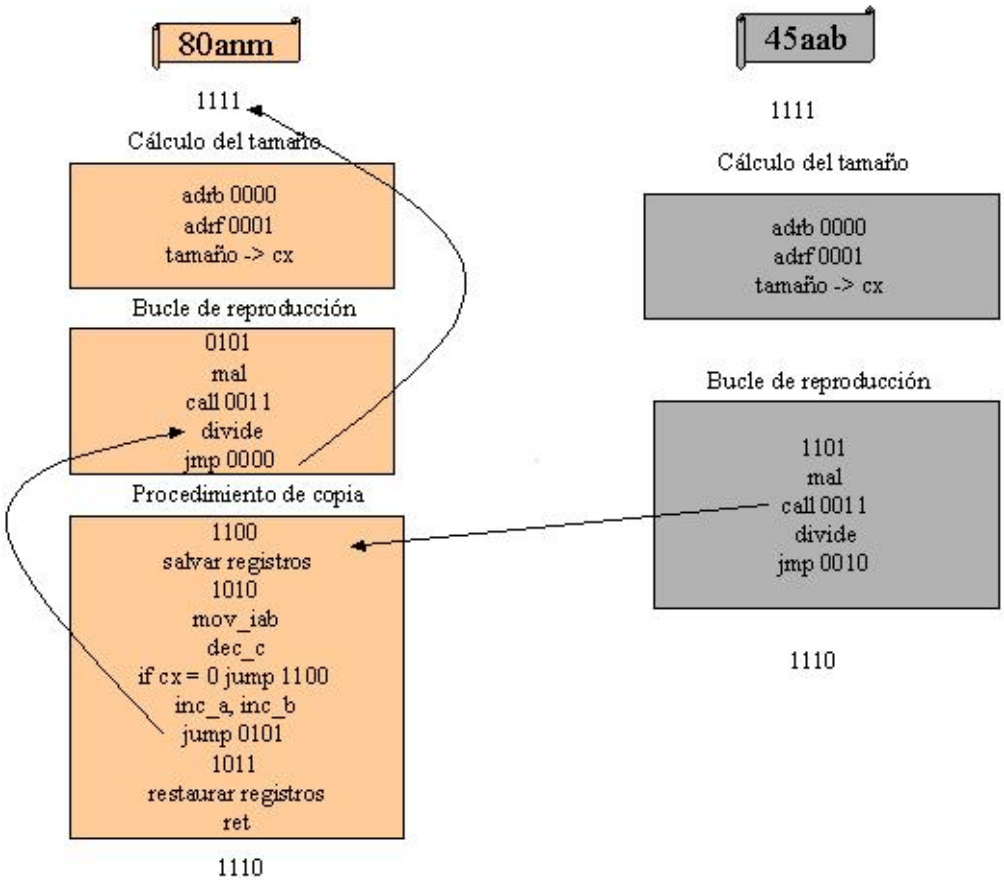


La relación de parasitismo 45aab - 80aaa se desarrolla durante los millones de instrucciones siguientes observando cómo estos dos tipos de criaturas son las más abundantes en la sopa, apareciendo fluctuaciones alternativas del número de ambas en memoria.

En los 45 millones de instrucciones, los parásitos dominan a las criaturas parasitadas. Cantidades aproximadas de presencia de ambos en memoria bajo este estado de "parasitación" son: 130 criaturas 45aab y 90 criaturas 80aaa. Llegando al extremo, en 50 millones de instrucciones, de 133 criaturas 45aab y 64 de las 80aaa. Es en este punto cuando hacen aparición con fuerza las criaturas 80anm. Son criaturas totalmente autorreplicantes, pero que difieren de la criatura ancestral introducida en que éstas recalculan siempre su tamaño

antes de reproducirse, para lo cual, tienen mutada una plantilla respecto a la 80aaa, la plantilla siguiente a *divide* y *jmp*. Y lo que es más importante, no ejecuta la instrucción *ret*, que permitía a los parásitos que el IP volviera a su propio código. Así, las 80anm, tras la ejecución del código de copia realizan un simple *jmp* al respectivo *divide*. Ésta es la inmunidad desarrollada.

```
80anm,
nop_1   inc_a   nop_0   nop_1
nop_1   sub_ab  nop_0   nop_0
nop_1   nop_0   push_ax nop_1
nop_1   nop_1   push_bx if_cz
zero    nop_0   push_cx nop_1
or1     nop_1   nop_1   nop_0
shl     mal     nop_0   nop_1
shl     call    nop_1   nop_1
mov_cd  nop_0   nop_0   pop_cx
adrb    nop_0   mov_iab pop_bx
nop_0   nop_1   dec_c   pop_ax
nop_0   nop_1   if_cz   ret
nop_0   divide  jmp     nop_1
nop_0   jmp     nop_1   nop_1
sub_ac  nop_0   nop_1   nop_1
mov_ab  nop_0   nop_0   nop_0
adrf    nop_0   nop_0   nop_1
nop_0   nop_0   inc_a   nop_0
nop_0   if_cz   inc_b   if_cz
nop_0   nop_1   jmp
nop_1   nop_1   nop_0
```



En los 60 millones de instrucciones ya aparecen 20 criaturas de genotipo 80anm. En los siguientes 20 millones de instrucciones la criatura 45aab sufrirá el efecto de su proliferación. Las criaturas 80aaa prácticamente habrán desaparecido, habrá unas 30 criaturas en toda la sopa, número insuficiente para proporcionar todos los bucles de copia requeridos por los parásitos. Éstos comienzan a morir, resultando la criatura más prolífica la 80anm. En torno a las 80 millones de instrucciones ya sólo habrá 30 criaturas 45aab. En 90 millones la situación será la siguiente: 6 criaturas 45aab, 38 criaturas 80aaa y 186 criaturas 80 anm. Así, en los 100 millones de instrucciones desaparecen completamente los parásitos. A partir de este instante las “reinas” de la sopa son las criaturas 80anm.

Continuamos la ejecución para observar si podían aparecer de nuevo parásitos, esta vez de la criatura 80anm. En este sentido, las observaciones han sido infructuosas. Quizá hemos interrumpido la ejecución antes de producirse el fenómeno, pero el margen de tiempo dejado parecía lo suficientemente amplio como para que en la sopa se diera la presencia de nuevas formas de vida capaces de replicarse.

7.8.2. ESTUDIO DEL PARÁSITO PROTOTÍPICO

Una de las pruebas fue enfocada a encontrar una criatura inmune al parásito 45. Para ello se inocularon en la sopa solamente criaturas del genotipo 45aab (prototipo de parásito) y varios genotipos de tamaño 79 obtenidos en otras ejecuciones. El objetivo era encontrar, si es que había alguno, uno o más genotipos del genoma 79 que fueran inmunes al parásito y ver si efectivamente, como describía Thomas S. Ray, las criaturas de tamaño 45 iban desapareciendo y las de 79 aumentando. A lo largo del tiempo de ejecución se veía como las criaturas entraban en un ciclo estable en las que ambos tamaños convivían sin problemas. Al estudiar el comportamiento de las mismas se comprobó que ninguna de las 79 inoculadas era inmune. De hecho tenían el mismo comportamiento que las de tamaño 80, teniendo una prole de tamaño 80. Por lo tanto, las criaturas de tamaño 45 parasitaban igualmente el bucle de copia de estas criaturas y evolucionaban de la misma manera que Thomas S. Ray comentó, pero para el caso de criaturas de tamaño 80.

A continuación podemos ver el genoma del parásito 45aab y el genoma de una de las criaturas de tamaño 79 con dicho comportamiento.

45aab,

```
nop_1   nop_0   nop_1   nop_0
nop_1   nop_0   nop_0   nop_1
nop_1   sub_ac  nop_1   nop_0
nop_1   mov_ab  mal     if_cz
zero    adrf    call    nop_1
or1     nop_0   nop_0   nop_1
shl     nop_0   nop_0   nop_1
shl     nop_0   nop_1   nop_0
mov_cd  nop_1   nop_1   push_ax
adrb    inc_a   divide
nop_0   sub_ab  jmp
nop_0   nop_1   nop_0
```

El parásito 45aab utiliza para su copia en memoria el bucle de copia de otras criaturas. Podemos observar que su instrucción *call* viene seguida de la plantilla 0011, no presente dentro de la criatura de tamaño 45. Dicha plantilla es exactamente la presente en la criatura ancestral como inicial para el bucle de copia. Dado que la instrucción *call* guarda en la pila la dirección de retorno, cuando al estar ejecutando el código de copia de otra criatura se ejecuta el *ret* se volverá a la dirección del propio parásito pasando a ejecutar el *divide*. Dentro del parásito existe un bucle por el cual se ejecuta el *call* y en el retorno el *divide* de forma infinita.

```
79afa,

nop_1           sub_ab          nop_1           jmp
nop_1           nop_1           nop_0           nop_0
nop_1           nop_1           nop_0           nop_1
zero            nop_0           push_ax         nop_0
or1             nop_1           push_bx         nop_1
shl             nop_1           push_cx         if_cz
shl             mal             nop_1           nop_1
mov_cd          call            nop_0           nop_0
adrb            nop_0           nop_1           nop_1
nop_0           nop_0           nop_0           nop_1
nop_0           nop_1           mov_iab         pop_cx
nop_0           nop_1           dec_c           pop_bx
sub_ac          divide          if_cz           pop_ax
mov_ab          jmp             jmp             ret
adrf            nop_0           nop_0           nop_1
nop_0           nop_0           nop_1           nop_1
nop_0           nop_1           nop_0           nop_1
nop_0           nop_0           nop_0           nop_0
nop_1           if_cz           inc_a           nop_1
inc_a           nop_1           inc_b
```

7.8.3. ALGUNAS CRIATURAS INTERESANTES OBTENIDAS A LO LARGO DE LAS PRUEBAS

A lo largo de las pruebas realizadas, han aparecido una serie de criaturas con códigos interesantes.

```
80aav,

nop_1           inc_a           nop_0           nop_1
nop_1           sub_ab          nop_0           nop_0
nop_1           nop_0           push_ax         nop_1
nop_1           nop_1           push_bx         if_cz
zero            nop_0           push_cx         nop_1
or1             nop_1           nop_1           nop_0
shl             mal             nop_0           nop_1
shl             call            nop_1           nop_1
mov_cd          nop_0           nop_0           pop_cx
adrb            nop_0           mov_iab         pop_bx
nop_0           nop_1           dec_c           pop_ax
nop_0           nop_1           if_cz           ret
nop_0           divide          jmp             nop_1
nop_0           jmp             nop_1           nop_1
sub_ac          nop_0           nop_1           nop_1
mov_ab          nop_0           nop_0           nop_0
adrf            nop_0           nop_0           if_cz
nop_0           nop_0           inc_a
nop_0           if_cz           inc_b
nop_0           nop_1           jmp
nop_1           nop_1           nop_0
```

Esta criatura es como la ancestral 80aaa sólo que, en cada vuelta del bucle de copia, salta a la primera instrucción, recalculando siempre su tamaño.

72aaa,

```
nop_1
nop_1
nop_1
nop_1
zero
or1
shl
shl
mov_cd
adrb
nop_0
nop_0
nop_0
nop_0
sub_ac
mov_ab
adrf
nop_0
nop_0
```

```
nop_0
nop_0
inc_a
sub_ab
nop_1
nop_1
nop_0
nop_1
mal
call
nop_0
nop_0
nop_1
nop_1
divide
jmp
nop_0
nop_0
nop_1
```

```
nop_0
if_cz
nop_1
nop_1
nop_0
nop_0
push_ax
push_bx
push_cx
nop_1
nop_0
nop_1
nop_0
mov_iab
dec_c
if_cz
jmp
nop_0
nop_1
```

```
nop_0
nop_0
inc_a
inc_b
jmp
nop_0
nop_1
nop_0
nop_1
if_cz
nop_1
nop_1
nop_1
nop_1
if_cz
```

Posee 8 instrucciones menos que la criatura ancestral 80aaa, pero el resto del código es exactamente el mismo. Esta criatura se comportaría como un parásito de la 80aaa para poder ejecutar la recuperación de los valores de la pila (*pop*) y el retorno (*ret*).

93aaa,

```
nop_1
nop_1
nop_1
nop_1
zero
or1
shl
shl
mov_cd
adrb
nop_0
nop_0
nop_0
nop_0
sub_ac
mov_ab
adrf
nop_0
nop_0
nop_0
nop_1
inc_a
sub_ab
```

```
nop_1
nop_1
nop_0
nop_1
mal
call
nop_0
nop_0
nop_1
nop_1
divide
jmp
nop_0
nop_0
nop_1
nop_0
if_cz
nop_1
nop_1
nop_0
nop_0
push_ax
push_bx
```

```
push_cx
nop_1
nop_0
nop_1
nop_0
mov_iab
dec_c
if_cz
jmp
nop_0
nop_1
nop_0
nop_0
inc_a
inc_b
jmp
nop_0
nop_1
nop_0
nop_1
if_cz
nop_1
nop_0
```

```
nop_1
nop_1
pop_cx
pop_bx
pop_ax
ret
nop_1
nop_1
nop_0
nop_0
if_cz
pop_bx
pop_ax
ret
nop_1
nop_1
nop_1
nop_0
if_cz
```

Esta criatura ejecuta el mismo código que la 80aaa, sólo que posee 13 instrucciones de más dentro de su espacio, pero que no sirven para nada. Es completamente funcional por ella misma, pero la velocidad de reproducción, para copia de todas sus instrucciones, es menor que la de la 80aaa, sucediendo que, en ninguna de las ejecuciones realizadas, la criatura 93aaa lograba una presencia significativa. Bajo condiciones del medio que pudieran beneficiarla, por ejemplo, otorgar prioridad a criaturas más grandes, esta criatura podría alcanzar importancia dentro del medio.

Las criaturas anteriores, si bien no han destacado por su presencia total en la sopa, hemos hallado en ellas características de autorreplicación, que, bajo circunstancias adecuadas, podrían facilitar que estas criaturas se desarrollaran en el medio digital.

A continuación presentamos dos parásitos. Para el primero, el comportamiento es como el del parásito típico, sólo que posee instrucciones dentro de su genoma que sobran, nunca las va a ejecutar. El segundo es un tipo de parásito como el visto 72aaa, pero también con instrucciones que nunca va a ejecutar pero que pertenecen a la criatura. Vemos aquí el mismo fenómeno que para la criatura 93aaa, la presencia de instrucciones inservibles dentro de la criatura que hacen que sea mayor dentro de la sopa.

```
52aaa,
nop_1           nop_0           nop_1           if_cz
nop_1           sub_ac          mal             nop_1
nop_1           mov_ab          call            nop_1
nop_1           adrf            nop_0           nop_0
zero            nop_0           nop_0           nop_1
or1             nop_1           nop_1           push_ax
shl             nop_0           nop_1           push_bx
shl             nop_1           divide          push_cx
mov_cd          inc_a           jmp             nop_1
adrb            sub_ab          nop_0           nop_0
nop_0           nop_1           nop_0           nop_1
nop_0           if_cz           nop_0           nop_0
nop_0           nop_0           nop_0           mov_iab

77aab,

nop_1           nop_0           nop_1           inc_b
nop_1           inc_a           nop_1           jmp
nop_1           sub_ab          nop_0           nop_0
nop_1           nop_1           nop_0           nop_1
zero            nop_1           push_ax         nop_0
or1             nop_0           push_bx         nop_1
shl             nop_1           push_cx         if_cz
shl             mal             nop_1           nop_0
mov_cd          call            nop_0           nop_1
adrb            nop_0           nop_1           nop_1
nop_0           nop_0           nop_0           nop_1
nop_0           nop_1           mov_iab         pop_cx
nop_0           nop_1           dec_c           nop_1
nop_0           divide          if_cz           nop_1
sub_ac          jmp             jmp             nop_1
mov_ab          nop_0           nop_0           nop_1
adrf            nop_0           nop_1           zero
nop_0           nop_1           nop_0
nop_0           nop_0           nop_0
nop_0           if_cz           inc_a
```

Otro parásito interesante es el 50aaa, que en algunas ejecuciones ha logrado muy buenos resultados:

50aaa,

```
nop_1           mov_cd          inc_a           jmp
nop_1           adrb            sub_ab          nop_0
nop_1           nop_0           nop_1           nop_0
nop_0           nop_0           nop_1           nop_1
if_cz           nop_0           nop_1           nop_0
nop_1           nop_1           nop_1           if_cz
nop_1           sub_ac          mal             nop_1
nop_1           mov_ab          call            nop_1
nop_1           adrf            nop_0           nop_1
zero            nop_0           nop_0           nop_0
or1             nop_0           nop_1           push_ax
shl             nop_0           nop_1
shl             nop_1           divide
```

Como otros parásitos comentados, a partir de la segunda ejecución no parasita únicamente el bucle de copia en sí, sino todo el código de la criatura ancestral 80aaa incluida la instrucción *mal*.

La criatura 85aaa ha sido mencionada en anteriores ocasiones. Suele aparecer en casi todas las ejecuciones, además, en los primeros miles de instrucciones. A pesar de su capacidad de autorreproducción, nunca llega a ser una de las más abundantes en la sopa.

85aaa,

```
nop_1           nop_0           if_cz           jmp
nop_1           nop_0           nop_1           nop_0
nop_1           nop_0           nop_1           nop_1
nop_0           nop_1           nop_0           nop_0
if_cz           inc_a           nop_0           nop_1
nop_1           sub_ab          push_ax         if_cz
nop_1           nop_1           push_bx         nop_1
nop_1           nop_1           push_cx         nop_0
nop_1           nop_1           nop_1           nop_1
zero            nop_1           nop_0           nop_1
or1             mal             nop_1           pop_cx
shl             call            nop_0           pop_bx
shl             nop_0           mov_iab         pop_ax
mov_cd          nop_0           dec_c           ret
adrb            nop_1           if_cz           nop_1
nop_0           nop_1           jmp             nop_1
nop_0           divide          nop_0           nop_1
nop_0           jmp             nop_1           nop_0
nop_1           nop_0           nop_0           if_cz
sub_ac          nop_0           nop_0
mov_ab          nop_1           inc_a
adrf            nop_0           inc_b
```

7.8.4. CRIATURA ANCESTRAL DE TAMAÑO 22

Las pruebas comentadas se han realizado tomando como criatura ancestral la 80aaa, pero como semilla primitiva se puede inocular cualquier otra criatura, siempre que tenga sentido ser la criatura ancestral, es decir, que sea autorreplicante.

Thomas S. Ray ha codificado una criatura de este tipo y de tamaño 22:

```
22aaa,

nop_0                                   push_bx
adrb                                    nop_0
nop_1                                   mov_iab
divide                                  dec_c
sub_ac                                  if_cz
mov_ab                                  ret
adrf                                    inc_a
nop_0                                   inc_b
inc_a                                   jumpb
sub_ab                                  nop_1
mal                                     mov_iab
```

En las ejecuciones realizadas con esta criatura no hemos observado una evolución con presencia de distintos tipos de criaturas. Algunas observaciones son:

- La velocidad de reproducción es altísima y en los primeros miles de instrucciones la sopa está llena, consecuencia de que el programa que se ejecuta tiene la cuarta parte de instrucciones que el anterior utilizado: 80aaa
- Dos criaturas dominan la sopa, la 22aaa y la 22aaf. Esta última es como la criatura 22aaa sólo que tiene mutadas la primera y tercera instrucción (plantillas) por lo que el funcionamiento del código es exactamente igual.
- Aparecen bastantes criaturas de tamaño 14. Todas ellas tienen como antepasado a la criatura 22aac cuya plantilla tras el *push_bx* está mutada. Este cambio provoca que en el cálculo del tamaño de la criatura se tome éste como 14 (posición de la plantilla mutada) y así aparecen estas criaturas de tamaño menor.

8.CONCLUSIONES

El proyecto ALiS surgió a partir del conocimiento del sistema Tierra de Thomas S. Ray. Partimos de los requisitos mínimos para este tipo de sistemas y comenzamos a crear nuestro propio mundo virtual. El objetivo consistía en conseguir reproducir los mismos resultados que Tierra, o por lo menos lo más parecidos, aparte de investigar variaciones de nuestra propia cosecha.

Una vez decidida la arquitectura del sistema y terminada la implementación del mismo nos dispusimos a estudiar los resultados que las mutaciones ejercían sobre la criatura ancestral y su prole. Hay que decir que pensamos que nada más incluirlas conseguiríamos obtener los resultados de Tierra, ya que una vez conseguido que la criatura ancestral se reprodujera y viviera sin problemas en el sistema, sólo era cuestión de introducir estas variaciones para que surgiera todo tipo de vida. No alcanzamos a prever los cambios tan radicales que se producen dependiendo del tipo de mutación y de la frecuencia con la que se produzca. Además no tuvimos en cuenta ciertas restricciones que podían hacer la vida de nuestras criaturas en nuestro mundo un poco más llevadera, como fue restringir el tamaño mínimo y máximo de las criaturas, para así eliminar criaturas no válidas que quitaban oportunidades a las válidas.

Un arduo estudio e investigación de los resultados que íbamos obteniendo nos fue llevando a este tipo de conclusiones haciendo que nuestro sistema se separase de Tierra y evolucionase hacia sus propias necesidades; unas veces acertadas como fue obligar a las criaturas a que rellenasen un tanto por ciento de los genomas de sus hijas, y otras no tanto, como por ejemplo, el rellenar la sopa con instrucciones aleatorias o con el genoma de la criatura ancestral, o el gratificar a una criatura cada vez que se reprodujese disminuyendo mucho el número su número de errores.

Una vez determinados todos los ajustes que se han comentado a lo largo de la descripción de las pruebas comenzaron a surgir criaturas con plena funcionalidad y comportamientos destacables como fueron los parásitos y las criaturas inmunes a los mismos, además de criaturas de distintos genotipos capaces de autorreproducirse.

Respecto al sistema Tierra original, en este sentido, hemos logrado obtener resultados evolutivos comparables: la obtención de parásitos e hiperparásitos es común en ambos sistemas. Aunque los genomas descritos por Thomas Ray y los nuestros coinciden en ocasiones, en otras, en ALiS han aparecido genomas no referenciados por Ray con comportamientos perfectamente clasificables. A pesar de estas similitudes, no hemos logrado obtener hitos como la reproducción sexual.

Otro problema que nos surgió fue el estudio de los resultados ya que el titánico número de criaturas distintas que se obtenían hacía muy difícil el seguimiento de las mismas a base de consultar el contenido de la memoria o de la lista de criaturas en los archivos generados. Aparte de construir un banco de genes que organizara a las criaturas según su tamaño y su

genotipo, construimos un analizador de criaturas que nos hiciera más fácil la tarea de la búsqueda de criaturas válidas y el estudio de los diversos comportamientos.

En nuestro caso el sistema acaba aquí, pero la investigación en el mismo podría continuarse hacia muy diversos campos como pueden ser el estudio de las cadenas alimenticias, la competición por la materia, la multicelularidad, la reproducción sexual tanto en organismos haploides como diploides, el desarrollo de distintos conjuntos de instrucciones que puedan formar parte de los genomas de las criaturas, el desarrollo de poblaciones en red que se relacionen entre sí intercambiando información genética propia de cada población, etc.

9.REFERENCIAS

[1] Ackley, D. H. y Littman, M.S. “Learning from natural selection in an artificial environment.” In: *Proceedings of the International Joint Conference on Neural Networks, Volume I, Theory* Track, Neural and Cognitive Sciences Track, IJCNN Winter 1990, Washington, DC. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1990.

[2] Bagley, R.J., Farmer, J.D., Kauffman, S.A., Packard, N.H., Perelson, A.S. y Stadnyk, I.M. "Modelling adaptative biological systems." Unpublished paper, 1989.

[3] Cariani, P. "Emergence and artificial life" In: *Artificial Life II*, edited by C.Langton, D.Farmer y S.Rasmussen. Redwood City, CA: Addison-Wesley, 1991, 000-000

[4] Cohen, F. *Computer viruses: theory and experiments*. Ph.D. dissertation, U. of Southern California, 1984.

[5] Dawkins, R. *The blind wat*chmaker. New York: W.W. Norton & Co.,1987.

[6] Dawkins, R. “The evolution of evolvability.” In: *Artificial life: proceedings of an* interdisciplinary workshop on the synthesis and simulation of living systems, edited by C. Langton. Redwood City, CA: Addison-Wesley, 1989, 201-220.

[7] Denning, P.J. “Computer viruses.” *Amer. Sci*. 76 (1988): 236-238.

[8] Dewdney, A.K. “Computer recreations: In the game called Core War hostile programs engage in a battle of bits.” *Sci. Amer*. 250 (1985a): 14-22.

[9] Dewdney, A.K. “Computer recreations: A core war bestiary of viruses, worms and other threats to computer memories” *Sci. Amer*. 252 (1985a): 14-23.

[10] Dewdney, A.K. “Computer recreations: Exploring the field of genetic algorithms in a primordial computer sea full of flibs.” *Sci. Amer*. 253 (1985b): 21-32.

[11] Dewdney, A.K. “Computer recreations: A program called MICE nibbles its way to victory at the first core war tournament.” *Sci. Amer*. 256 (1987): 14-20.

[12] Dewdney, A.K. “Of worms, viruses and core war.” *Sci. Amer*. 260 (1989): 110-113.

[13] Farmer, J.D., Kauffman, S.A. y Packard, N.H. "Autocatalytic replication of polymers." *Physica D* 22 (1986): 50-67.

[14] Holland, J.H. *Adaptation in natural and artificial systems: an introductory analysis with* applications to biology, control, and artificial intelligence. Ann Arbor: Univ. of Michigan Press, 1975.

[15] Holland, J.H. "Studies of the spontaneous emergence of self-replicating systems using cellular automata and formal grammars." In: *Automata, Languages, Development*, edited by Lindenmayer, A. y Rozenberg, G. New York: North-Holland, 1976, 385-404.

[16] Langton, C.G. "Studying artificial life with cellular automata." *Physica D* 22 (1986): 120-149.

[17] Packard, N.H. “Intrinsic adaptation in a simple model for evolution.” In: *Artificial Life:* proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems, edited by C.Langton. Redwood City, CA: Addison-Wesley, 1989, 141-155.

[18] Pattee, H.H. "Simulations, realizations, and theories of life." In: *Artificial Life: proceedings of* an interdisciplinary workshop on the synthesis and simulation of living systems, edited by C.Langton. Redwood City, CA: Addison-Wesley, 1989, 63-77

[19] Rasmussen, S., Knudsen, C., Feldberg, R. y Hindsholm, M. "The coreworld: emergence and evolution of cooperative structures in a computational chemistry" *Physica D* 42 (1990): 111-134.

[20] Ray, T. S. "Evolution, ecology and optimisation of digital organisms." Santa Fe Institute working paper 92-08-042, 1992.

[21] Ray, T. S. 1994. "Jugué a ser Dios y creé la vida en mi computadora." In: Claudio Gutiérrez [ed], *Epistemología e Informática*, 257-267. San José, Costa Rica: UNED, 1993.

[22] Ray, T. S. "Overview of Tierra at ATR." In: *Technical Information, No.15, Technologies for* Software Evolutionary Systems. ATR-HIP. Kyoto, Japan. 2001.

[23] Rheingold, H. (1988). Computer viruses. *Whole Earth Review* Fall (1988): 106.

[24] Spafford, E.H., Heaphy, K.A. y Ferbrache, D.J. *Computer viruses, dealing with electronic* vandalism and programmed threats. ADAPSO, 1300 N. 17th Street, Suite 300, Arlington, VA 22209, 1989.

Silvia García Díaz, Carmen Gutiérrez Ramírez y Raquel Hervás Ballesteros autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo. Silvia García Díaz

Fdo. Carmen Gutiérrez Ramírez

Fdo. Raquel Hervás Ballesteros