Universidad de las Ciencias Informáticas

Facultad 5

Incorporación de algoritmos genéticos a la biblioteca de inteligencia artificial.

Trabajo de Diploma para optar por el Título de

Ingeniero en Ciencias Informáticas

Autores: Irina Marrero Borges.
Yoan Parra Nápoles.

Tutor: Ing. Yidier Romero Zaldívar.

Ciudad de la Habana.
Junio del 2007.

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo al <nombre área> de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Irina Marrero Borges

_____
Firma del Autor

Yoan Parra Nápoles

_____
Firma del Autor

Ing. Yidier Romero Zaldívar.

_____
Firma del Tutor

Datos de Contacto:

Yidier Romero Zaldívar.

Graduado de Ingeniero en Ciencias Informáticas en el año 2006

Profesor de la Universidad de las Ciencias Informáticas desde ese mismo año.

Jefe de la asignatura Inteligencia Artificial en la facultad 5.

yromero@uci.cu.

Dedicatoria.

A mi abuelo Nenin que donde quiera que esté estará orgulloso de mí.
A mi abuela, por su dulzura y cariño.
A Yurell, por ser padre y amigo a la vez, por guiarme siempre, por ser fuente de mi inspiración.
A Carmen, por ser lo que más quiero en esta vida, por sus consejos, por ser mi razón de ser.
A mi tía y a Lori por ser especiales en mi vida, mil gracias por sus consejos.
A mi hermano Ismar gracias... por siempre estar junto a mí.
A mi familia, siempre conmigo, en cada paso que doy en la vida.

Irina Marrero Borges...

A mis padres Ana y Pedro, gracias por a su cariño, guía y apoyo he llegado a realizar uno de los anhelos más grandes de mi vida.
A mí querida hermana Yaima y en especial a mi bella niña Lilian por ser mi principal inspiración.

Yoan Parra Nápoles…

Agradecimientos.

A mi tutor Yidier por la preocupación y responsabilidad mostrada.
A Yoan Parra por compartir conmigo largas horas de trabajo.
Al profesor Frank Puig por guiarme en el desarrollo de esta tesis.
Al profesor Karel Pérez por la ayuda brindada.

A mis compañeros de aula, por compartir 5 años juntos, por regalarme una sonrisa cuando realmente me hacia falta. A todos los quiero mucho.

A Yurima, por ser de esas personas que te dan la mano cuando crees que el mundo se te viene encima.
A mis compañeras de apto por ser la familia mas cercana cuando se esta tan lejos.
A Alexito por dejarme descubrir un gran amigo.

A Eder por darme tantas alegrías juntas, por ser especial en mi vida.
A Maura, tata, y Genaro por confiar en mí.

A Niurka y Tony, por hacerme más fácil estos cinco años.
A Dania, Yanis, tantas atenciones y ayuda infinita…mil gracias.
A José por la preocupación infinita.
A Tere por compartir con mi familia tantos años...por ser parte de ella.
A todos mil Gracias…

Irina Marrero Borges.

Agradecimientos.

A todos los que han colaborado con la realización de este trabajo o que han influido en nuestras vidas durante estos 5 años de carrera. No quisiera dejar pasar por alto a personas que merecen un reconocimiento especial.

A mi tutor Yidier por su preocupación y sus consejos.
Al profesor Frank Puig por su gran ayuda.
A mi compañera Irina por acompañarme en la realización de este sueño.
A Yaself por su colaboración.
A Alexander, Jorge, Javier, Daniel, Luis Daniel por permitirme tenerlos como amigos.
A todos mis compañeros de aula por darme tantas alegrías.
A los socios del apartamento del frente, Leo, el Kenya, a Markitos, el Fácil y demás del piquete.

A todos los que no estén igual muchas gracias.

Yoan Parra Nápoles.

Resumen.

La inteligencia artificial (IA) constituye todo un potencial por descubrir que podría revolucionar los juegos de cualquier género. Dentro de la IA encontramos los algoritmos genéticos. Teniendo como objetivo esta investigación desarrollar un módulo de algoritmo genético para la biblioteca de inteligencia artificial para entornos virtuales.

En la investigación realizada como parte de este trabajo se abordan los conceptos necesarios y el funcionamiento de los algoritmos genéticos. En este documento se muestra las facilidades que brindan estos algoritmos así como la funcionalidad de la biblioteca con este módulo.

Se expone, a continuación de la fase de investigación, el diseño e implementación de un conjunto de clases que permiten conformar el módulo. Como resultado de este proceso el módulo que se obtuvo cuenta con las características necesarias para su acople a la biblioteca. Con el uso del módulo desarrollado se facilita la programación de la parte inteligente de los juegos, reduciendo además el tiempo de desarrollo de estas aplicaciones.

Palabras claves: inteligencia artificial, algoritmos genéticos, biblioteca de inteligencia artificial, entornos virtuales.

Índice.

Capítulo 1: Fundamentación Teórica. 13
1.1 Algoritmos genéticos. 14
1.2 Funcionamiento de los Algoritmos genéticos. 14
1.3 Elementos de un algoritmo genético. 16
1.4 Operadores genéticos. 17
1.4.1 Selección. 18
1.4.2 Cruzamiento. 18
1.4.3 Mutación. 19
1.5 ¿Por qué usar el algoritmo genético? 20
1.6 Diferentes aplicaciones. 21
1.7 Algoritmos genéticos en el desarrollo de juegos. 26
1.8 Biblioteca de Inteligencia Artificial. 28
Capítulo 2: Características del módulo de algoritmo genético. 30
2.1 Objeto de estudio. 31
2.1.1 Problema y situación problémica. 31
2.1.2 Objeto de automatización. 31
2.1.3 Información que se maneja. 31
2.1.4 Propuesta de sistema. 31
2.2 Modelo de dominio. 34
2.3 Diagrama de actividades. 35
2.4 Glosario de términos. 38
2.5 Dependencias y relaciones con otro software. 40
Capítulo 3: Diseño e implementación del módulo de algoritmo genético. 41
3.1 Diagrama de clases de diseño por paquetes. 42
3.2 Paquete módulo de algoritmo genético. 43
3.3 Relación entre las clases de los paquetes. 44
3.4 Descripción de las clases de diseño. 45
3.5 Diagrama de componentes. 47
Capítulo 4: Pruebas al módulo de algoritmo genético. 49
4.1 Funcionamiento del ejemplo. 50
4.2 Rendimiento. 51
4.3 Complejidad del algoritmo genético. 53
Conclusiones. 54
Recomendaciones. 55
Bibliografía consultada. 58
Glosario de abreviaturas. 60
Anexos 61
- **Anexo (juego de Roles).** 61

Índice de Figuras.

- Figura 1 Algoritmo genético. 13
- Figura 2 Sistema de monitorización y supervisión inteligente de fermentaciones y procesos. 23
- Figura 3 Robot utilizando AG. 24
- Figura 4 Localización de objetos. 25
- Figura 5 Evolución de los algoritmos genéticos. 30
- Figura 6 Modelo de dominio para módulo de algoritmo genético. 35
- Figura 7 Diagrama de Actividades. 37
- Figura 8 Diagrama de clases de diseño por paquetes. 42
- Figura 9 Diagrama de clases del paquete módulo de algoritmo genético. 43
- Figura 10 Relación entre clases. 44
- Figura 11 Diagrama de componentes. 48
- Figura 12 Población y Tiempos. 52

Índice de Tablas.

- Tabla 1 Módulo de algoritmo genético. 42
- Tabla 2 Biblioteca inteligencia artificial. 42
- Tabla 3 Descripción Clase GA. 46
- Tabla 4 Descripción clase GA_Node. 47
- Tabla 5 Población y tiempos. 51
- Tabla 6 Número de iteraciones y Tiempo. 52
- Tabla 7 Iteraciones y Tiempo. 53

Introducción.

La inteligencia artificial es una rama de la Informática que pretende desarrollar programas en los que el ordenador manifieste conductas típicas de los seres inteligentes. En la actualidad la IA vive un período de auge en el que se desarrollan y comercializan satisfactoriamente aplicaciones de esta ciencia en muchos campos.

Un área importante de la inteligencia artificial es la de los algoritmos genéticos (AG), en la cual las soluciones a un problema se codifican en genotipos, que son entonces manipulados. Se comprueba la idoneidad de cada genotipo y los mejores se reproducen entre ellos para producir la siguiente generación. Esta secuencia se repite hasta que se descubre la solución o la mejor aproximación posible.

Los AG son algoritmos de optimización, búsqueda y aprendizaje inspirados en los procesos de evolución natural y evolución genética. (1)

En la actualidad las aplicaciones de estos algoritmos están enfocadas a resolver problemas de optimización. (2) Mientras el poder de la evolución gana reconocimiento cada vez más generalizado, los algoritmos genéticos se utilizan para abordar una amplia variedad de problemas en un conjunto de campos sumamente diversos como la física, la química, la ciencia de los materiales, la biología, la biotecnología y la farmacología, los videos juegos y entrenadores virtuales demostrando claramente su capacidad y su potencial. (3)

La Universidad de las Ciencias Informáticas (UCI), un centro docente-productivo, desarrolla en sus laboratorios un número importante de proyectos. Entre los productos desarrollados por estos proyectos tienen lugar los relacionados con la Realidad Virtual (RV) y donde se encuentran los juegos. ¿Qué sería de un juego donde los enemigos se comportan de forma poco "inteligente"?, simplemente no sería divertido jugar, y rápidamente aprenderíamos los movimientos y su comportamiento. Aquí es donde podemos aplicar conocimientos de IA en nuestros juegos, dándole un toque más real y más entretenido.

La Facultad 5 -encargada de desarrollar estas aplicaciones- comenzó la construcción de una biblioteca de inteligencia artificial. La misma carece de las funcionalidades que brindan los AG lo cual provoca que los proyectos productivos cuando tienen que aplicar esta técnica de IA demoren mucho tiempo en su desarrollo o desistan de esa solución que por lo general es muy ventajosa.

El **problema** a resolver es: ¿cómo desarrollar un módulo con algoritmos genéticos para entornos virtuales integrado a la biblioteca de inteligencia artificial?

El **objeto de estudio** de la investigación planteada lo constituye: "Los algoritmos genéticos para entornos virtuales". Definiéndose como **campo de acción** "un módulo de algoritmo genético para entornos virtuales integrado a la biblioteca de inteligencia artificial".

El **objetivo** de este trabajo es desarrollar un módulo con algoritmos genéticos integrado a la biblioteca de inteligencia artificial para entornos virtuales.

Para el total cumplimiento del mismo se trazan las siguientes tareas de investigación:

- Estudiar bibliografías que aborden el tema de los algoritmos genéticos con el objetivo de adquirir los conocimientos necesarios para llegar a implementar un módulo.
- Analizar el funcionamiento de la biblioteca de inteligencia artificial para facilitar y lograr un trabajo eficiente.
- Desarrollar soluciones técnicas para alcanzar los objetivos propuestos.
- Realizar el diseño de un módulo con algoritmos genéticos para incorporarlo a la biblioteca de inteligencia artificial que permita la interacción de los elementos virtuales.
- Implementar el módulo antes mencionado.

La investigación esta estructurada en capítulos:

En un capítulo inicial, **Fundamentación Teórica**, se realizará un análisis bibliográfico donde se investiguen las características de los AG. En el segundo capítulo donde se describen las **Características del sistema**, se expondrán las características que presentará el sistema como solución a los problemas planteados. En el tercero, **diseño e implementación del sistema**, se localizarán los artefactos necesarios correspondientes a este flujo de RUP. Un cuarto capítulo de **Pruebas** al módulo. Finalmente, se ofrece un glosario de abreviaturas y otro de términos para ayudar a la comprensión del lenguaje técnico utilizado en el trabajo.

Capítulo 1: Fundamentación Teórica.

Introducción.

Los algoritmos genéticos se basan en el principio de la evolución, es decir, la supervivencia de los más aptos. Por lo tanto las técnicas de la programación evolutiva, basadas en algoritmos genéticos, son aplicables a muchos problemas de optimización, como la optimización de funciones lineales y no lineales con restricciones. La importancia de estas técnicas sigue creciendo, ya que la evolución de los programas es paralela en la naturaleza, y el paralelismo es una de las direcciones más prometedoras en las ciencias de la computación.

En el presente capítulo se exponen los principales conceptos de los algoritmos genéticos y características de su funcionamiento así como las aplicaciones de los mismos.

- Figura 1 Algoritmo genético.

1.1 Algoritmos genéticos.

La naturaleza es sabia, cualquier animal o vegetal en sucesivas generaciones ha sido capaz de adaptarse a cambios en su forma de vida, e incluso en su estructura, para combatir cualquier agente que hiciera peligrar su supervivencia (evolución de Darwin). El hombre con el estudio de las ciencias de la computación ha imitado estos algoritmos, definiéndolo en esta esfera como:

AG son una técnica de optimización y búsqueda basada en los principios de la selección natural. Un AG permite evolucionar una población compuesta de varios individuos bajo reglas de selección especificadas a un estado que maximiza el "fitness" (la aptitud). (4)

El método fue desarrollado por John Holland durante la década de los 60 y popularizado por uno de sus estudiantes, David Goldberg, quien fue capaz de resolver un problema complejo acerca de transmisión en tuberías de gas en 1989. Holland fue el primero en intentar desarrollar una base teórica para los algoritmos genéticos a través de su teorema de esquemas. (5).

1.2 Funcionamiento de los Algoritmos genéticos.

Los algoritmos genéticos establecen una analogía entre el conjunto de soluciones de un problema, llamado fenotipo, y el conjunto de individuos de una población natural, codificando la información de cada solución en una cadena, generalmente binaria, llamada cromosoma. Los símbolos que forman la cadena son llamados genes. Cuando la representación de los cromosomas se hace con cadenas de dígitos binarios se le conoce como genotipo. Los cromosomas evolucionan a través de iteraciones, llamadas generaciones. En cada generación, los cromosomas son evaluados usando alguna medida de aptitud. Las siguientes generaciones (nuevos cromosomas), llamada descendencia, se forman utilizando dos operadores, de cruzamiento y de mutación. (6)

Se genera aleatoriamente la población inicial, que está constituida por un conjunto de cromosomas, que representan las posibles soluciones del problema. En caso de no hacerlo aleatoriamente, es importante

garantizar que dentro de la población inicial, se tenga la diversidad estructural de estas soluciones para tener una representación de la mayor parte de la población posible o al menos evitar la convergencia prematura. A cada uno de los cromosomas de esta población se aplicará la función de aptitud para saber qué tan "buena" es la solución que se está codificando.

Después de saber la aptitud de cada cromosoma se procede a elegir los cromosomas que serán cruzados en la siguiente generación.

Los cromosomas con mejor aptitud tienen mayor probabilidad de ser seleccionados.

El cruzamiento es el principal operador genético, representa la reproducción sexual, opera sobre dos cromosomas a la vez para generar dos descendientes donde se combinan las características de ambos cromosomas padres.

El AG se deberá detener cuando se alcance la solución óptima, pero ésta generalmente se desconoce, por lo que se deben utilizar otros criterios de detención. Normalmente se usan dos criterios: correr el AG un número máximo de iteraciones (generaciones) o detenerlo cuando no haya cambios en la población.

El problema de selección de variables se puede ver como un problema de optimización, ya que si se quiere encontrar, bajo alguna heurística, el subconjunto de variables que potencialicen la diferenciación y las semejanzas de objetos de clases diferentes y de la misma clase respectivamente.

Un algoritmo genético tiene también una serie de parámetros que se tienen que fijar para cada ejecución, como los siguientes:

- **Tamaño de la población**: debe de ser suficiente para garantizar la diversidad de las soluciones, y, además, tiene que crecer más o menos con el número de bits del cromosoma, aunque nadie ha aclarado cómo tiene que hacerlo. Por supuesto, depende también del ordenador en el que se esté ejecutando.

- **Condición de terminación:** lo más habitual es que la condición de terminación sea la convergencia del algoritmo genético o un número prefijado de generaciones.

Seudocódigo de un Algoritmo genético. (7) (8)

```
BEGIN /*Algoritmo Genético*/
    Generar una población inicial.
     Computar la función de evaluación de cada individuo.
      WHILE NOT Terminado DO
      BEGIN /* Producir nueva generación*/
          FOR Tamaño población /2 DO
          BEGIN /* Ciclo reproductivo*/
              Seleccionar dos individuos de la anterior generación,
              Para el cruce (probabilidad de selección proporcional a la función
              de evaluación del individuo).
              Cruzar  con cierta probabilidad los dos individuos obteniendo dos descendientes.
              Mutar los dos descendientes con cierta probabilidad.
              Computar la función de evaluación de los descendientes mutados.
              Insertar los dos descendientes mutados en la nueva generación.
          END
          IF   la población ha convergido THEN
               Terminado:=TRUE
       END
END
```

1.3 Elementos de un algoritmo genético.

Como los Algoritmos genéticos se encuentran basados en los procesos de evolución de los seres vivos, casi todos sus conceptos se basan en conceptos de biología y genética que son fáciles de comprender. (9)

Individuo: Un individuo es un ser que caracteriza su propia especie. El individuo es un cromosoma y es el código de información sobre el cual opera el algoritmo. Cada solución parcial del problema a optimizar está codificada en forma de cadena o String en un alfabeto determinado, que puede ser binario. Una cadena representa a un cromosoma, por lo tanto también a un individuo y cada posición de la cadena representa a un gen. Esto significa que el algoritmo trabaja con una codificación de los parámetros y no con los parámetros en sí mismos. El genotipo, es el conjunto de genes ordenados y representa las

características del individuo. Cada individuo tiene una medida de su adecuación como solución al problema. (10)

Población: A un conjunto de individuos (Cromosomas) se le denomina población. El método de algoritmo genético consiste en ir obteniendo de forma sucesiva distintas poblaciones. Por otra parte un Algoritmo genético trabaja con un conjunto de puntos representativos de diferentes zonas del espacio de búsqueda y no con uno solo.

Codificación: Se supone que los individuos (posibles soluciones del problema), pueden representarse como un conjunto de parámetros (que denominaremos genes), los cuales agrupados forman una ristra de valores (a menudo referida como cromosoma). Si bien el alfabeto utilizado para representar los individuos no debe necesariamente estar constituido por el {0,1} buena parte de la teoría en la que se fundamentan los Algoritmos genéticos utiliza dicho alfabeto. Se trabaja principalmente con una cadena de bits pero se pueden utilizar arreglos, árboles y listas.

Función evaluación: La única restricción para usar un algoritmo genético es que exista una función llamada evaluación, que le informe de cuan bueno es un individuo dado en la solución de un problema. Esta función de evaluación es el principal enlace entre el algoritmo genético a un problema real, es la efectividad y eficiencia de la función evaluación que se tome, por lo tanto debe procurarse que la función de evaluación sea similar, sino igual a la función objetivo que se quiere optimizar. Esta medida se utiliza como parámetro de los operadores y guía la obtención de nuevas poblaciones.

1.4 Operadores genéticos.

Para el paso de una generación a la siguiente se aplican una serie de operadores genéticos. Los más empleados son los operadores de selección, cruce y mutación.

1.4.1 Selección.

Los algoritmos de selección serán los encargados de escoger qué individuos van a disponer de oportunidades de reproducirse y cuáles no.

Puesto que se trata de imitar lo que ocurre en la naturaleza, se ha de otorgar un mayor número de oportunidades de reproducción a los individuos más aptos. Por lo tanto la selección de un individuo estará relacionada con su valor de ajuste. No se debe sin embargo eliminar por completo las opciones de reproducción de los individuos menos aptos, pues en pocas generaciones la población se volvería homogénea.

Un Algoritmo genético puede utilizar muchas técnicas diferentes para seleccionar a los individuos que deben copiarse hacia la siguiente generación, a continuación aparecen algunas de las más comunes. Algunas de estas técnicas son mutuamente excluyentes, pero otras pueden utilizarse en combinación, algo que se hace a menudo. (11)

- Selección elitista
- Selección proporcional a la aptitud
- Selección por rueda de ruleta
- Selección escalada
- Selección por torneo
- Selección por rango
- Selección generacional
- Selección por estado estacionario
- Selección jerárquica

1.4.2 Cruzamiento.

Una vez seleccionados los individuos, éstos son recombinados para producir la descendencia que se insertará en la siguiente generación.

Algunos aspectos importantes a tener en cuenta son:

- Los hijos deberían heredar algunas características de cada padre. Si éste no es el caso, entonces estamos ante un operador de mutación.
- Se debe diseñar de acuerdo a la representación.
- La recombinación debe producir cromosomas válidos.
- Se utiliza con una probabilidad alta de actuación sobre cada pareja de padres a cruzar (PC entre 0.6 y 0.9), si no actúa los padres son los descendientes del proceso de recombinación de la pareja.

El cruzamiento se puede realizar de diferentes formas. Básicamente en cualquiera de ellas lo que se hace es elegir a dos individuos para que intercambien segmentos de su código y así crear una descendencia artificial donde los individuos sean una combinación de sus padres, simulándose el proceso convencional de la recombinación que realizan los cromosomas durante la reproducción sexual.

Existen multitud de algoritmos de cruce. Sin embargo los más empleados son los que se detallarán a continuación:

- Cruce de 1 punto.
- Cruce de 2 puntos.
- Cruce uniforme.

1.4.3 Mutación.

La mutación de un individuo provoca que alguno de sus genes, generalmente uno sólo, varíe su valor de forma aleatoria.

Aunque se pueden seleccionar los individuos directamente de la población actual y mutarlos antes de introducirlos en la nueva población, la mutación se suele utilizar de manera conjunta con el operador de cruce. Primeramente se seleccionan dos individuos de la población para realizar el cruce. Si el cruce tiene éxito entonces uno de los descendientes, o ambos, se muta con cierta probabilidad. Se imita de esta manera el comportamiento que se da en la naturaleza, pues cuando se genera la descendencia siempre se produce algún tipo de error, por lo general sin mayor trascendencia, en el paso de la carga genética de padres a hijos.

La probabilidad de mutación es muy baja, generalmente menor al 1%. Esto se debe sobre todo a que los individuos suelen tener un ajuste menor después de mutados. Sin embargo se realizan mutaciones para garantizar que ningún punto del espacio de búsqueda tenga una probabilidad nula de ser examinado.

La mutación más usual es el reemplazo aleatorio. Este consiste en variar aleatoriamente un gen de un cromosoma. Si se trabaja con codificaciones binarias consistirá simplemente en negar un bit. También es posible realizar la mutación intercambiando los valores de dos alelos del cromosoma. Con otro tipo de codificaciones no binarias existen otras opciones:

- Incrementar o decrementar a un gen una pequeña cantidad generada aleatoriamente.
- Multiplicar un gen por un valor aleatorio próximo a 1.

Aunque no es lo más común, existen implementaciones de algoritmos genéticos en las que no todos los individuos tienen los cromosomas de la misma longitud. Esto implica que no todos ellos codifican el mismo conjunto de variables. En este caso existen mutaciones adicionales como puede ser añadir un nuevo gen o eliminar uno ya existente.

1.5 ¿Por qué usar el algoritmo genético?

Los algoritmos genéticos fueron diseñados para resolver los problemas de optimización no lineales con gran cantidad de variables cuya relación es muy difícil de predecir (por ello, en la inteligencia artificial de un juego se aplica perfectamente). (12)

La solución genética es innecesaria si nos encontramos con alguna de las condiciones siguientes:

El problema puede resolverse analíticamente. Desarrollar una solución para el problema de la búsqueda de un valor x que maximice un polinomio es un derroche de tiempo, porque una solución exacta puede encontrarse calculando la derivada del polinomio.

El problema es lineal en su totalidad. Si el problema puede ser representado por un sistema de ecuaciones lineales, cualquiera de las técnicas clásicas de optimización (por ejemplo, el método simplex) puede resolverlo.

Se conoce que **existe un único valor óptimo y la distribución es homogénea** (por ejemplo, no es un máximo local)**.** En este caso, el algoritmo escalador de colinas simple puede resolverlo.

Las soluciones candidatas pueden ser enumeradas en un espacio razonable de tiempo. En este caso, es mejor que trate con todas ellas.

¿Cuándo no funcionan?

Ejecutar un algoritmo genético requiere normalmente de una significativa inversión de tiempo. Incluso nuestra simple simulación de un troll o nomo consumió alrededor de 22 minutos de procesamiento en un Pentium III a 800MHz. Por tanto, es poco probable que alguien pueda lograr que evolucione (me refiero a buscar una solución con AG) un programa de jugar ajedrez, incluso con una computadora como la Multivac de Asimov (Multivac cuyo nombre es derivado de la Univac un meinframe que si existió y se utilizaba para grandes procesamientos de datos y Asimov es un escritor de ficción que describe en uno de sus cuentos una computadora que resolvía todo tipo de problemas) a su disposición.

Por otra parte, dado que el número de cromosomas posibles aumenta exponencialmente con el número de genes en el modelo, el diseño de una arquitectura conformista para sus individuos puede ser de gran importancia.

1.6 Diferentes aplicaciones.

Los algoritmos genéticos se han utilizado para la resolución de problemas y para la elaboración de modelos. (13)

Optimización: en una gran variedad de tareas, incluyendo la optimización numérica, y los problemas de optimización combinatoria, como el problema del viajante y la calidad de sonido.

Programación automática: para desarrollar los programas de ordenador para tareas específicas, y otras de diseño computacional de las estructuras, por ejemplo, los autómatas celulares y las redes de clasificación.

Máquina robot y aprendizaje: para muchas aplicaciones de la máquina de aprendizaje, incluyendo la predicción de estructura de proteínas. También se han utilizado para diseñar redes neuronales, para evolucionar las normas de aprendizaje de los sistemas de clasificación o de los sistemas de producción simbólica, y para el diseño y control de robots.

Modelos económicos: para modelar los procesos de innovación, el desarrollo de estrategias de oferta, y el surgimiento de los mercados económicos.

Modelos de sistemas inmunológicos: para modelar diversos aspectos de la física del sistema inmunológico, incluida la mutación de un individuo durante toda la vida y el descubrimiento de varias familias de genes en la evolución en el tiempo.

Modelos ecológicos: para modelar los fenómenos ecológicos, tales como las carreras de armamentos biológicos, la simbiosis y el flujo de recursos en ecología.

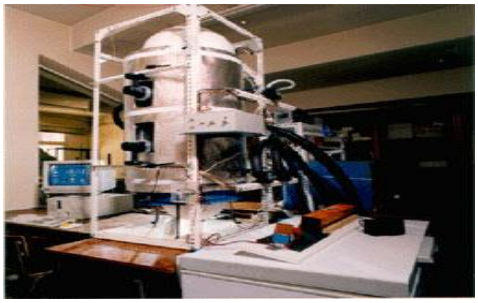
Modelos de genética de poblaciones: para el estudio de cuestiones en la genética de poblaciones.

Las interacciones entre evolución y aprendizaje: para estudiar como el aprendizaje individual y la evolución de las especies afectan una a la otra.

Modelos de sistemas sociales: para el estudio de los aspectos evolutivos de los sistemas sociales, como el comportamiento de las hormigas.

Comportamiento de una "anguila metálica": Científicos de la Universidad de Edimburgo han desarrollado un programa informático que utiliza algoritmos genéticos para controlar el comportamiento de una "anguila metálica" de 140 metros de largo que produce energía a partir de las olas del mar.

Supervisión inteligente y optimización de fermentaciones y procesos: Se ha creado un sistema de monitorización y supervisión inteligente de fermentaciones y procesos, mediante ordenador y sensores. Además, se ha puesto a punto un método de optimización de producción, tiempo y calidad.



- Figura 2 Sistema de monitorización y supervisión inteligente de fermentaciones y procesos.

La supervisión se efectúa mediante un sistema de sensorización de desarrollo propio, y utilizando modelos del proceso como referencia. La optimización se basa en **algoritmos genéticos**, que encuentran rápidamente una solución, de modo que cabe reaccionar a tiempo ante cambios en el proceso, buscando el mejor resultado en cuanto a producción, tiempo y calidad.

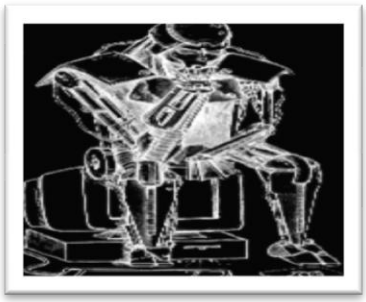
El sistema de supervisión puede detectar a tiempo tendencias no deseadas en un proceso. Acoplado con la optimización, puede reaccionar ante cambios del proceso, reconduciéndolo al mejor comportamiento.

El sistema tiene aplicación para el control de calidad del agua, industria alimentaria e industria farmacéutica.

Aprendiendo comportamientos de un Robot utilizando algoritmos genéticos.

El Robot juega cada vez más un papel importante en diferentes industrias. Tiene que actuar con gran precisión y eficiencia. Esto puede no sonar muy difícil si el entorno en el que opera el robot no cambia, ya que el comportamiento de un robot podría ser pre-programado. Sin embargo, si el medio ambiente es siempre cambiante, se vuelve extremadamente difícil, para los programadores de averiguar todo lo posible de los comportamientos del robot. La aplicación de robots en un entorno cambiante es inevitable en la tecnología moderna, pero también es cada vez más frecuente. Esto, evidentemente, ha dado lugar al desarrollo de un robot de aprendizaje.

La motivación es que cometer errores en el sistema real puede ser costoso y peligroso. Además, las limitaciones de tiempo pueden limitar el grado de aprendizaje en el mundo real. Desde el aprendizaje se requiere experimentando con conductas que pueden producir ocasionalmente resultados indeseables si se aplica a la vida real.



- Figura 3 Robot utilizando AG.

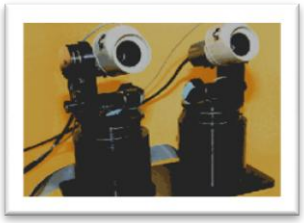
Rol de los algoritmos genéticos.

Los algoritmos genéticos son técnicas de búsqueda de adaptación que pueden aprender las estructuras de los conocimientos de alto rendimiento. Proceden de una fuerza implícita en la búsqueda paralela de espacios de soluciones que realiza a través de una población de soluciones candidatas y esta población es manipulada en la simulación. La solución candidata representa todos los posibles comportamientos de los robots y basado en el rendimiento global de los candidatos, cada uno de ellos podría ser asignado un valor de evaluación. Los operadores genéticos se podrían aplicar para mejorar el rendimiento de los comportamientos de la población. Un ciclo de ensayo de todos los comportamientos se define como una generación, y se repite hasta que evolucione un buen comportamiento. El buen comportamiento es aplicado al mundo real. También debido a la naturaleza de los algoritmos genéticos, el conocimiento inicial no tiene que ser muy bueno.

El sistema descrito ha sido utilizado para aprender comportamientos para simular el control de vehículos submarinos autónomos, la evasión de misiles, y otras tareas simuladas.

Algoritmos Genéticos para la localización de objetos en una escena Compleja.

Con el fin de ofrecer máquinas con la capacidad de interactuar en una escena compleja, los entornos del mundo real, datos sensoriales deben ser presentados a la máquina. Un módulo de una de ellas que se ocupa de la entrada sensorial es el módulo de procesamiento de datos visual, también conocido como el módulo de visión por ordenador. Una tarea central de este módulo es la visión de reconocer imágenes de objetos del medio ambiente. (15)



- Figura 4 Localización de objetos.

La segmentación es el proceso de búsqueda de objetos de interés mientras que el trabajo de reconocer para ver si encuentra el objeto que coincida con los atributos predefinidos. Dado que las imágenes no pueden ser reconocidas hasta que se encuentren y separados de los antecedentes, es de suma importancia que este módulo de visión sea capaz de localizar los diferentes objetos de interés para los distintos sistemas con gran eficiencia.

Parámetros de algoritmos genéticos.

La tarea de localizar un determinado objeto de interés en un complejo escenario es bastante simple cuando se lanza en el marco de los algoritmos genéticos El uso de la metodología genética, puede elevar la configuración de fuerza bruta a una elegante solución a este problema complejo. Un experimento fue llevado a cabo sobre la base de la siguiente técnica, algoritmos genéticos optimizado para portabilidad y paralelismo desarrollado en la Universidad Estatal de Michigan.

Se ha demostrado que el algoritmo genético tiene un mejor desempeño en la búsqueda de áreas de interés, incluso en un complejo escenario del mundo real.

Vida artificial.

Los algoritmos genéticos son en la actualidad los más destacados y ampliamente utilizados en la evolución de los sistemas de vida artificial. Esta descentralización de los modelos proporciona una base para la comprensión de muchos otros sistemas y fenómenos en el mundo. Investigaciones sobre algoritmos genéticos ofrecen ejemplos en los que el algoritmo genético se utiliza para estudiar cómo el aprendizaje y la evolución interactúan, y el modelo de los ecosistemas, del sistema inmunológico, los sistemas cognitivos, y los sistemas sociales.

Papel en la toma de decisiones.

Los algoritmos genéticos parecen ser muy apropiado para apoyar el diseño y las fases de elección de la toma de decisiones.

La solución más conveniente en la última generación es la que maximiza o minimiza la función de evaluación, esta solución puede ser pensada con un algoritmo genético. Cuando la solución de los problemas multi-objetivo, el algoritmo genético lleva a cabo muchas soluciones satisfactorias en términos de los objetivos, y, a continuación, permite que el fabricante de decisión seleccione la mejor alternativa. Pueden ser de gran ayuda para el examen de las alternativas, ya que están diseñados para evaluar las posibles soluciones existentes y generar nuevas (y mejores) soluciones para la evaluación. Así los algoritmos genéticos pueden mejorar la calidad de la toma de decisiones.

1.7 Algoritmos genéticos en el desarrollo de juegos.

Los algoritmos genéticos son muy usados en escenarios donde el NPC adversario es capaz de responder y cambiar a causa del comportamiento del jugador. Si se toma como ejemplo hipotético un juego de roles y donde puedan jugar varios jugadores a la vez. En este juego los jugadores deben ser capaces de escoger entre varios personajes de diferentes características y habilidades. Esto significa que los NPC tendrán que ser áviles para derrotar a otros NPC con otras características distintas. Un jugador puede

seleccionar un personaje de tipo guerrero, que su estilo es pelear por la fuerza bruta. Por supuesto que este tipo de jugador tiene la posibilidad de escoger una serie de armas como son: una espada, un hacha, entre otras como estas para atacar, además, puede cubrirse con una armadura. Por otro lado existen otros tipos de personajes diferentes que el jugador puede escoger como: el mago, el cual tiene un comportamiento diferente al anterior. Para un diseñado de personajes le es engorroso crear un NPC con todas sus características para que interactúe con los demás NPCs, por lo tanto, esta tarea se complica un poco más cuando se trata de juegos multijugadores. En este tipo de juegos la computadora tiene el reto de hacer funcionar la combinación de un grupo diverso de NPCs. El número posible de combinaciones puede ser mucho mayor que el que pudiera desarrollar un diseñador de juegos.

Codificación

Se desea un tipo de NPC que sea capaz de enfrentarse a un jugador o a un grupo de jugadores. Esta no es una búsqueda que se puede calcular con antelación pues cada jugador o grupo de jugadores se comporta de manera diferente. Lo que se necesita es definir cuales son las situaciones y las respuestas respectivas que aumentan o disminuyen el nivel de rendimiento de la población. Por ejemplo, un situación posible puede ser cuando el jugador ataca a un NPC con un arma mágica (hechizos, etc). Para esta situación se puede crear varios tipos de respuestas posibles a esta acción del jugador: o que el NPC ataque al jugador, o que trate de escapar, o que trate de esconderse. Posteriormente se podrían codificar esta situación y las respuestas como un cromosoma y si este cromosoma se crea con la respuesta de atacar cuando para esta situación, entonces el jugador será atacado cuando use un arma mágica contra el NPC. Hasta el momento se describió un solo escenario pero se pueden definir varios más, en el anexo se detallan los escenarios restantes.

La generación inicial

Hasta el momento se tienen definidos los posibles escenarios que se usarán, los posibles comportamientos asociados a cada escenario y se creó un estructura llamada cromosoma que almacena esta información. El próximo paso es generar la población inicial, la cual se genera aleatoriamente (anexo).

Función de evaluación

Para determinar cuales son los mejores individuos que se comportan frente a otros jugadores se necesitamos un criterio de evaluación. Este criterio de evaluación se puede definir de varias maneras, generalmente en los juegos de rol se le asignan a cada jugador cierta puntuación que aumenta si el jugador acierta en un disparo y disminuye cuando recibe un disparo. En este ejemplo hipotético se define como función de evaluación la división de los hit points (golpes acertados) entre los damage points (daño recibido) y de esta manera los que mayor tengan este valor mejor se comportaran frente a otro jugador.

Evolución

A partir de que se tenga definida la generación inicial y el criterio de evaluación se aplican los operadores genéticos a dicha generación. En la selección se toma la generación inicial y se ordenan los individuos por el criterio de evaluación mencionado anteriormente. Una vez ordenados los individuos se seleccionan los N mejores (N, es la cantidad de individuos que se define para la población inicial). En el cruzamiento se agrupan los individuos en parejas y se les aplica el cruce de la manera siguiente: a partir de dos cromosomas se genera un tercero, se toman los genes pares de un progenitor y los genes impares del otro y con la unión de ambas selecciones queda constituido el nuevo cromosoma. En la mutación se define una probabilidad de mutación, este valor garantiza que el algoritmo encuentre soluciones que no es capaz de generar el proceso de cruzamiento, para este ejemplo se tomó el valor de 0.05. El proceso de mutación también se realiza aleatoriamente. Estas operaciones están detalladas en el anexo.

1.8 Biblioteca de Inteligencia Artificial.

En la UCI, en su polo de Realidad Virtual perteneciente a la Facultad 5, se construye una biblioteca de IA para el desarrollo de aplicaciones como juegos y simuladores. Dicha biblioteca surgió con el objetivo principal de separar la programación encargada de la visualización, la física y otras funcionalidades presentes en los sistemas de RV del código para darle inteligencia a los objetos de la escena.

Sin hacer uso de una biblioteca de IA, los programadores encargados de la inteligencia en estas aplicaciones se les hace difícil comenzar a programar sin haber concluido casi totalmente el desarrollo del resto del software. Utilizando una biblioteca que independice la IA del resto del producto los beneficios son muchos y entre ellos se encuentra la reducción del tiempo de desarrollo.

Otro objetivo de la biblioteca es permitir la reutilización del código en diferentes juegos u otras aplicaciones, evitando así tener que reprogramar íntegramente una y otra vez funciones muy similares, lo cual sucede muy frecuentemente.

Dicha biblioteca contará con varios algoritmos de inteligencia organizados por módulos entre los que se destacan las redes neuronales y los AG. A la misma, el empleo de la programación basada en componentes y el resto de su diseño le permitirán usarse en distintas aplicaciones y aprovechar sus similitudes con el propósito de aumentar la reutilización.

Capítulo 2: Características del módulo de algoritmo genético.

Introducción:

En el presente capítulo se comienza un acercamiento a la solución del problema planteado al comienzo del trabajo. Con el propósito de satisfacer los objetivos de la investigación. Para ello se ha realizado un amplio análisis de los operadores que conforman un algoritmo genético vistos en el primer capítulo, seleccionándose la combinación más óptima para resolver el problema.

- Figura 5 Evolución de los algoritmos genéticos.

2.1 Objeto de estudio.

2.1.1 Problema y situación problémica.

La Universidad de las Ciencias Informáticas (UCI), un centro docente-productivo, desarrolla en sus laboratorios un número importante de proyectos. Entre los productos desarrollados por estos proyectos tienen lugar los relacionados con la realidad virtual (RV) y donde se encuentran los juegos. ¿Qué sería de un juego donde los enemigos se comportan de forma poco "inteligente"?, simplemente no sería divertido jugar, y rápidamente aprenderíamos los movimientos y su comportamiento. Aquí es donde podemos aplicar conocimientos de IA en nuestros juegos, dándole un toque más real y más entretenido.

La Facultad 5 -encargada de desarrollar estas aplicaciones- comenzó la construcción de una biblioteca de inteligencia artificial. La misma carece de las funcionalidades que brindan los algoritmos genéticos lo cual provoca que los proyectos productivos cuando tienen que aplicar esta técnica de la inteligencia artificial demoren mucho tiempo en su desarrollo o desistan de esa solución que por lo general es muy ventajosa.

El problema científico a resolver es: ¿cómo desarrollar un módulo con algoritmos genéticos para entornos virtuales integrado a la biblioteca de inteligencia artificial?

2.1.2 Objeto de automatización.

El objeto de automatización de la presente investigación es el proceso de evolución de individuos que intervienen en un entorno sintético representado en las escenas virtuales.

2.1.3 Información que se maneja.

La información que se maneja es la referente a las características de los individuos que componen una población, representadas en los cromosomas generalmente formados por cadenas de bit (genes).

2.1.4 Propuesta de sistema.

A partir de los elementos y operadores presentes en los algoritmos genéticos estudiados en la fundamentación teórica (capítulo anterior), se propone como solución al problema planteado el desarrollo de un módulo que sea capaz de solucionar problemas variables en el tiempo. La sencillez de sus métodos

debe hacerlo más atractivo para los programadores y le proporcione a los juegos un mayor realismo en cuanto a el comportamiento de los individuos.

La propuesta de solución al problema planteado se describe a continuación:

Haciendo uso de las funcionalidades del *sistema de percepción* con el que cuenta la biblioteca que consiste en un conjunto de funcionalidades que permiten acceder a la información de las características del entorno virtual donde se encuentran los individuos así como a los datos de los cromosomas se conforma una **población Inicial.** La misma está definida como un vector donde cada posición está constituida por un individuo o cromosoma. La población inicial es la primera solución del problema la cuál mediante iteraciones evoluciona constituyendo soluciones mejoradas. Estará constituida por un solo individuo que puede tener diferentes codificaciones y sus características no determinan grandes cambios en la solución final.

Se propone realizar este módulo mediante la modelación e implementación de las siguientes funciones:

La función de **evaluación** determina si las características del individuo se aproximan a las deseadas. Esta función es redefinida por el programador según el resultado que desee. Posibilita que la evaluación sea específica del problema a que se enfrente. Al aplicarle la función a un determinado individuo se conoce del mismo si es más o menos apto para la solución a la que se pretende llegar.

La función de **mutación** descrita en el capítulo anterior tendrá uso en la solución que se propone. En el algoritmo mutarán los individuos escogidos teniendo en cuenta varios criterios. Si se tienen menos de 10 individuos a todos se le aplicará la mutación, sino se muta con un criterio probabilístico definido por el programador.

La especificación de la forma en que mutarán los individuos será redefinida por el programador que haga uso de la biblioteca según sus necesidades. Esta función a partir de un individuo devuelve el padre y sus descendientes.

Es necesario tener una función **cruzar** para obtener otros individuos con características similares a la de los ya existentes mediante la recombinación de estos. Se proponen tres formas para cruzar los individuos enumeradas a continuación:

- En el orden en que se encuentran en el vector. Desde la posición inicial hasta el último individuo, el primero con el segundo, el segundo con el tercero y así sucesivamente.
- Todos con todos. El primero con todos los que le siguen, el segundo con el primero y todos los que le siguen y así sucesivamente.
- Los n mejores. Se escoge una cantidad a cruzar según defina el programador teniendo en cuenta un criterio de evaluación y a estos se le aplica una de las formas de cruzamiento anteriores.

Esta función es reimplementable por el programador que haga uso de la biblioteca, con el propósito de que se ajuste las características de los cromosomas. La función recibe los individuos a cruzar y como resultado de ella se obtienen los nuevos individuos.

Con el propósito de elegir los mejores individuos es necesario contar con la función de **selección.** Esta función hará uso de la evaluación determinando cuales de los cromosomas son los más aptos. La selección será proporcional a la aptitud, o sea puede ser tanto para maximizar ciertas características como para minimizarlas. De esta manera se seleccionarán los individuos que pasarán a la siguiente generación limitados en cantidad por una constante definida por el programador. Esta cantidad determina la longitud del vector a analizar en la siguiente iteración.

El proceso del algoritmo genético es iterativo por lo cual debe presentar una **condición de parada**. Se propone como condiciones de terminación las siguientes.

- Si en la población no ocurren cambios entre una generación y la que le sigue.
- Si se alcanzó el número máximo de iteraciones.
- Si el margen de error es mayor al error calculado, con el objetivo de que si tengo una solución dentro de ese margen de error el algoritmo termine.

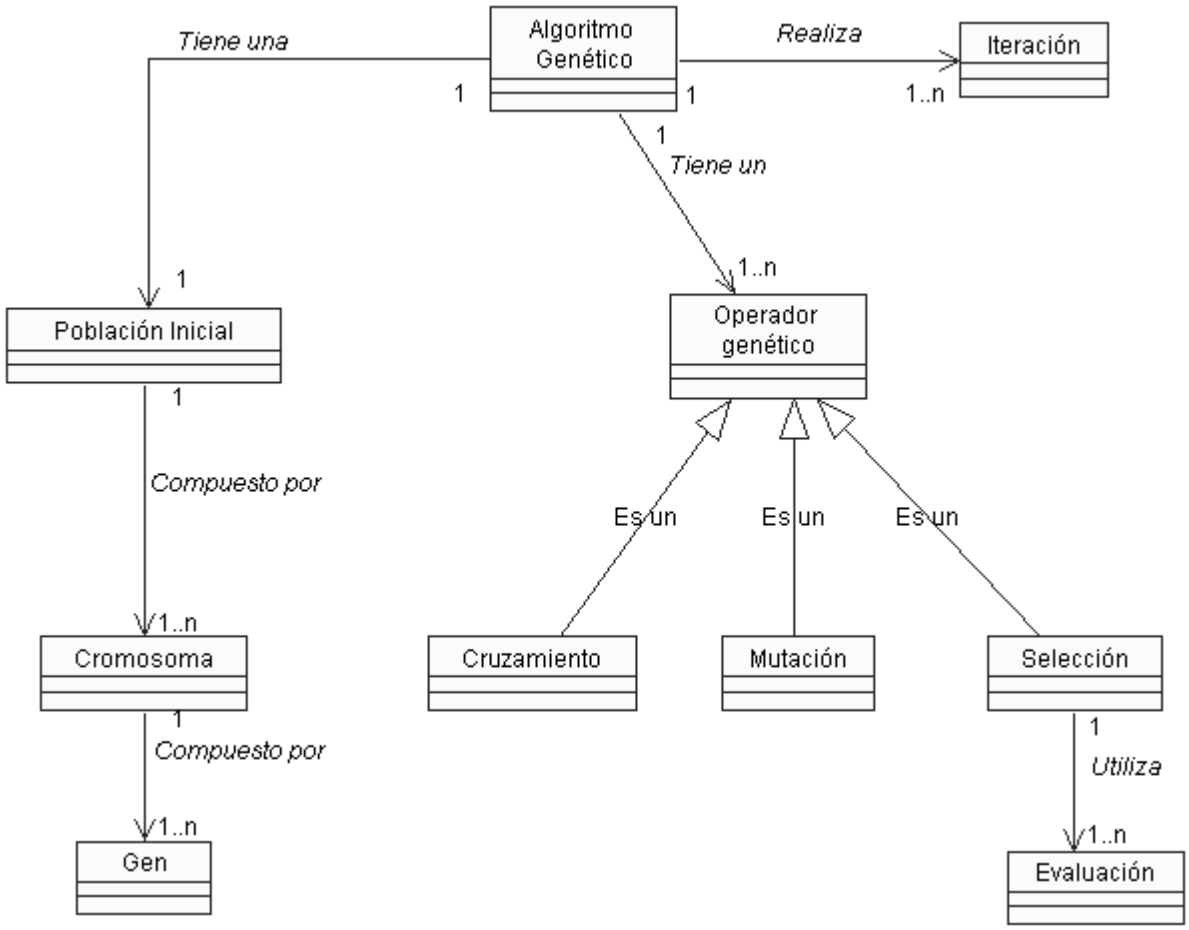
El AG generalmente sigue el orden de operaciones *Selección – Cruce – Mutación*, pero se propone como parte de esta solución realizar las operaciones en el orden *Mutación – Cruce – Selección*. Este orden

garantiza poder partir de una población inicial de un solo individuo y obtener una población más numerosa.

2.2 Modelo de dominio.

El modelo de dominio (o modelo conceptual) es una representación visual de los conceptos u objetos del mundo real significativos para un problema o área de interés. Representa clases conceptuales del dominio del problema. Representa conceptos del mundo real, no de los componentes de software. (17)

El modelo de dominio representado a continuación es un acercamiento a la solución propuesta, donde se modelan los principales conceptos así como sus relaciones con los que se trabajarán en el módulo a obtener.



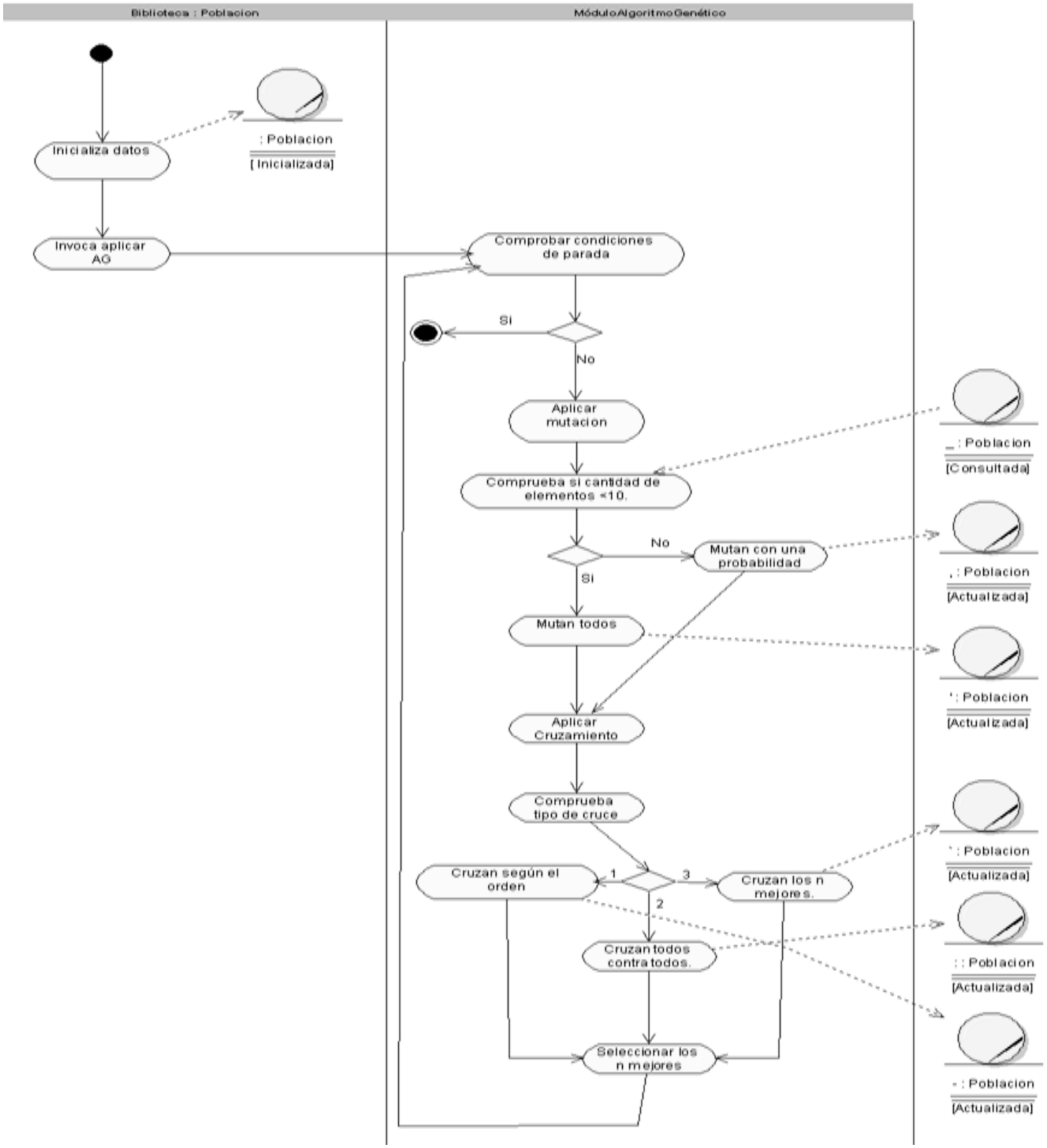
- Figura 6 Modelo de dominio para módulo de algoritmo genético.

2.3 Diagrama de actividades.

Un diagrama de actividades puede considerarse como un caso especial de un diagrama de estados en el cual casi todos los estados son estados de acción (identifican una acción que se ejecuta al estar en él) y casi todas las transiciones evolucionan al término de dicha acción (ejecutada en el estado anterior). Un

diagrama de actividades puede dar detalle a un caso de uso, un objeto o un mensaje en un objeto. (17) Permiten representar transiciones internas al margen de las transiciones o eventos externos. En la figura se presenta un ejemplo de diagrama de actividades para un mensaje de un objeto.

Se decidió hacer un diagrama de actividades por ser la mejor forma de representar o explicar como van ocurriendo los procesos en el módulo.



- Figura 7 Diagrama de Actividades.

Ahí se muestra un seudo código del módulo de algoritmo genético. Esto permite codificar un programa con mayor agilidad que en cualquier lenguaje de programación.

BEGIN /*Algoritmo Genético*/

WHILE NOT Terminado **DO**
BEGIN /* Producir nueva generación*/
Mutar los individuos.
Insertar los descendientes mutados en la nueva generación.
Cruzar los individuos.
Insertar los descendientes cruzados en la nueva generación.
Seleccionar los individuos de la anterior generación según la función de evaluación.

END
END

2.4 Glosario de términos.

En el modelo de dominio anterior se ilustran conceptos de los cuales es importante conocer su significado. El glosario y el modelo de dominio ayudan a los usuarios, clientes desarrolladores y otros interesados a utilizar un lenguaje común. Los ingenieros de hoy en día deben fundir el lenguaje de todos los participantes en uno solo consistente.

Algoritmo genético: Es un procedimiento inspirado en la evolución (Charles Darwin), programado en computadoras (ordenadores) y orientado a producir soluciones a problemas donde los tratamientos clásicos encuentran dificultades.

Cromosomas: Se divide en genes, cada uno de los cuales codifica un individuo. El contenido del gen representa una característica del individuo, por lo general se utiliza una representación binaria, sin embargo es posible utilizar valores reales.

Cruzamiento: los cruzamientos en los algoritmos genéticos tienen analogía con el proceso en que tiene lugar en la vida real. El cruzamiento combina las características en el proceso de reproducción de dos individuos diferentes, del cual resulta en una prole que se queda con parte del material genético de cada progenitor. De no existir el cruzamiento, los hijos serian una copia de uno de los padres.

Evaluación: Evalúa la aptitud de una función de cada cromosoma en la población de acuerdo a un criterio de aptitud.

Genes: es la unidad básica de herencia, un gen es la unidad mínima que se puede heredar, es decir, es la unidad mínima que puede ser tomada de uno de los progenitores para formar el nuevo individuo. Si las características heredables (como el aspecto de la cara) las descomponemos en otras subcaracterísticas (color de los ojos, color del pelo, tez), cuando ya no podamos dividir más esas características de forma que sigan siendo heredables, diremos que esa característica es debida a un gen. También es un gen la unidad mínima que se puede mutar.

Iteraciones: Repetición de una secuencia de instrucciones o eventos. Por ejemplo, en un lazo de programa, una iteración se produce una vez a través de las instrucciones del lazo.

Individuo: Generalmente representado por un solo cromosoma. El número de individuos dependerá del problema, en su conjunto forman poblaciones.

Mutación: Modificación de características de los cromosomas, generalmente está dada por la inversión de bits, se seleccionan y estos son invertidos.

Operadores genéticos: Para el paso de una generación a la siguiente se aplican una serie de operadores genéticos. Así se le llama al conjunto de funciones principales de un algoritmo genético selección, mutación y cruzamiento.

Población inicial: puede entenderse a partir de los siguientes significados.

Grupo de individuos de la misma especie que viven en un área específica, generalmente aislados hasta cierto punto de otros grupos similares.

Es el conjunto de individuos en un determinado momento o generación.

Generalmente se representan mediante cadenas binarias de longitud L que codifican el problema.

Selección: Selecciona cromosomas padres de la población de acuerdo a su aptitud, cuanto más apto es el individuo, posee mayor posibilidad de ser seleccionado.

2.5 Dependencias y relaciones con otro software.

El resultado de la investigación de este trabajo está concebido para que sea un módulo de la biblioteca de inteligencia artificial que se desarrolla en el proyecto “Interacción de elementos Inteligentes”. Las clases que formarán parte del módulo de AG deben tener una fuerte integración con la biblioteca debido a que utilizará algunas de sus funcionalidades para crear sus propias responsabilidades.

Capítulo 3: Diseño e implementación del módulo de algoritmo genético.

Introducción:

El diseño del módulo de algoritmo genético en esta investigación se define como el proceso de aplicar ciertas técnicas y principios con el propósito de definir un módulo, con suficientes detalles como para permitir su interpretación y realización física.

En este capítulo se describen todos los aspectos del módulo a construir mediante algunos artefactos generados en los flujos de trabajos de diseño e implementación que permitirán comprender mejor el módulo.

3.1 Diagrama de clases de diseño por paquetes.

- Figura 8 Diagrama de clases de diseño por paquetes.

Subsistemas	Clases de interés contenidas
Módulo de algoritmo genético	*GA* *GA_Node*

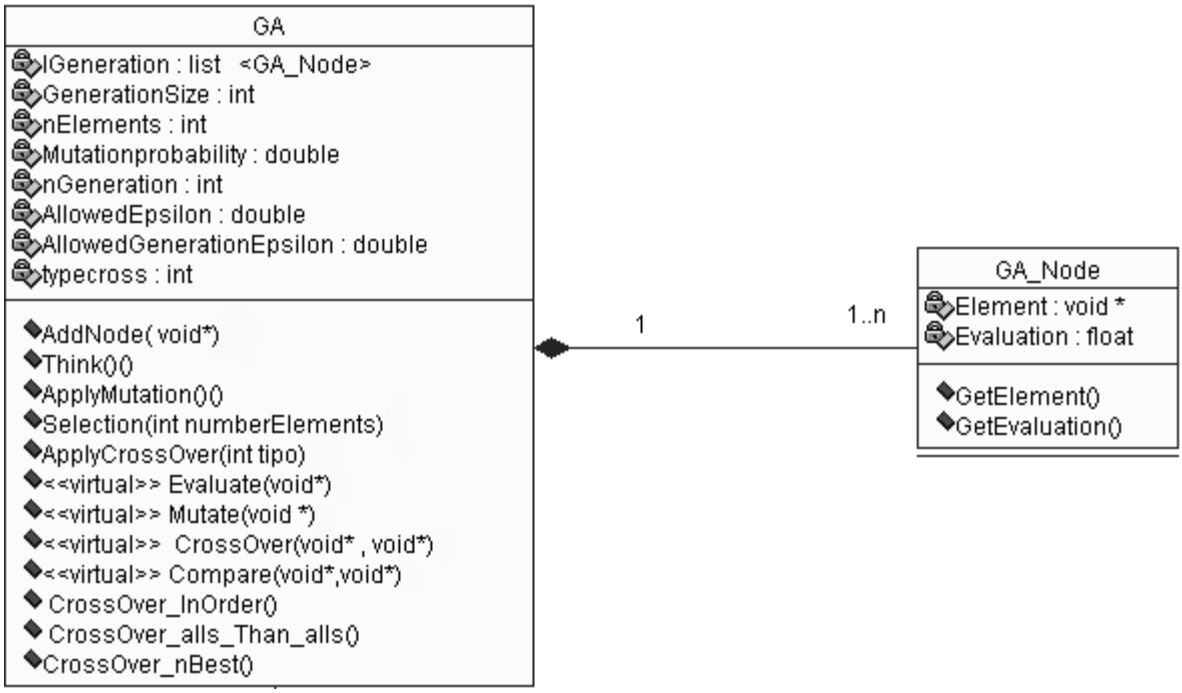
- Tabla 1 Módulo de algoritmo genético.

Subsistemas	Clases de interés contenidas
Biblioteca de inteligencia artificial	*World* *Servicios* *Think* *Path Finder* *Perception* *Actores* *Componentes* *Location* *Rigid* *Static* *Brain* *Sensor*

- Tabla 2 Biblioteca inteligencia artificial.

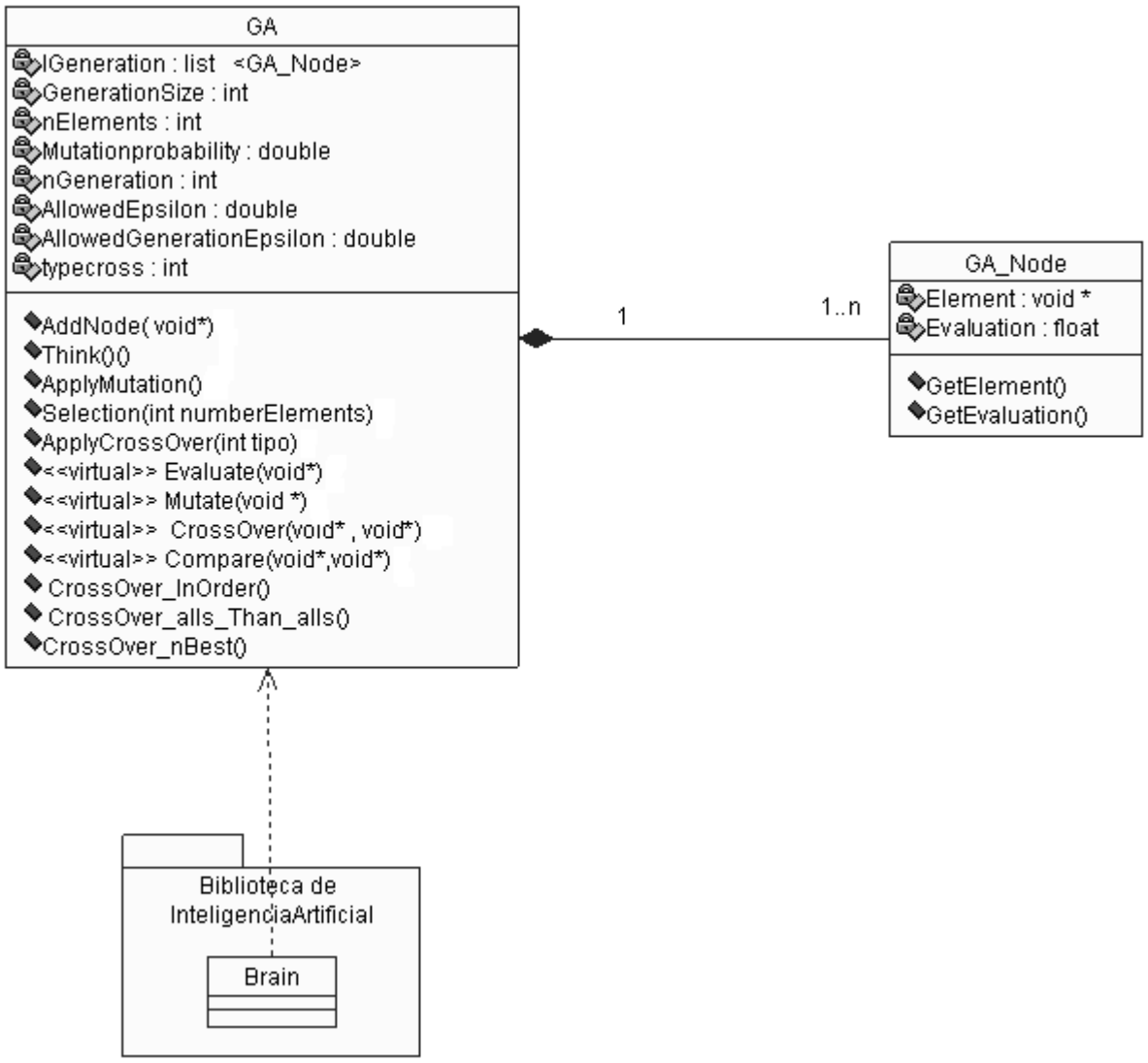
3.2 Paquete módulo de algoritmo genético.

Un **diagrama de clases** es un tipo de diagrama estático que describe la estructura de un sistema mostrando sus clases, atributos y las relaciones entre ellos. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro.



- Figura 9 Diagrama de clases del paquete módulo de algoritmo genético.

3.3 Relación entre las clases de los paquetes.



- Figura 10 Relación entre clases.

3.4 Descripción de las clases de diseño.

Nombre: *GA*	
Tipo de clase: Controladora	
Atributo	**Tipo**
lGeneration	*list <GA_Node>*
GenerationSize	*int*
nElements	*int*
Mutationprobability	*double*
nGeneration	*int*
AllowedEpsilon	*double*
AllowedGenerationEpsilon	*double*
typecross;	*int*
Para cada responsabilidad:	
Nombre:	*AddNode(void* Element)*
Descripción:	Adiciona un nuevo elemento a la lista.
Nombre:	*Think()*
Descripción:	Combina todos los operadores genéticos.
Nombre:	*ApplyMutation()*
Descripción:	En él se realiza la llamada al operador genético de mutación para cada elemento con una probabilidad de mutación igual a *Mutationprobability*.
Nombre:	*Selection(int numberElements);*
Descripción:	Selecciona los *elementos con mayor función de evaluación.*
Nombre:	*ApplyCrossOver(int tipo);*
Descripción:	Se llama al tipo de cruzamiento que se le especifica en la variable tipo.
Nombre:	*Evaluate(void*)*
Descripción:	Se implementa en las clases hijas y devuelve la aptitud de un individuo.
Nombre:	*Mutate(void *)*
Descripción:	Se implementa en las clases hijas y devuelve una lista con los elementos nuevos

	después de la mutación.
Nombre:	*CrossOver(void* , void*)*
Descripción:	Se implementa en las clases hijas y devuelve una lista con los hijos que resultaron de cruzar ambos elementos.
Nombre:	*Compare(void*,void*)*
Descripción:	Se implementa en las clases hijas y permite saber si dos elementos son exactamente iguales.
Nombre:	CrossOver_InOrder()
Descripción:	Se cruzan en el orden en que se encuentran en el vector. Desde la posición inicial hasta el último individuo, el primero con el segundo, el segundo con el tercero y así sucesivamente.
Nombre:	CrossOver_alls_Than_alls()
Descripción:	Todos con todos. El primero con todos los que le siguen, el segundo con el primero y todos los que le siguen y así sucesivamente.
Nombre:	CrossOver_nBest()
Descripción:	Los n mejores. Se escoge una cantidad a cruzar según defina el programador teniendo en cuenta un criterio de evaluación y a estos se le aplica una de las formas de cruzamiento anteriores.

- Tabla 3 Descripción Clase GA.

Nombre: *GA_Node*	
Tipo de clase: Entidad	
Atributo	**Tipo**
element	*void **
Evaluation	*float*
Para cada responsabilidad:	
Nombre:	*GetElement()*

Descripción:	Devuelve el atributo element
Nombre:	*GetEvaluation()*
Descripción:	Devuelve el atributo Evaluation que corresponde a la aptiitud del elemento.

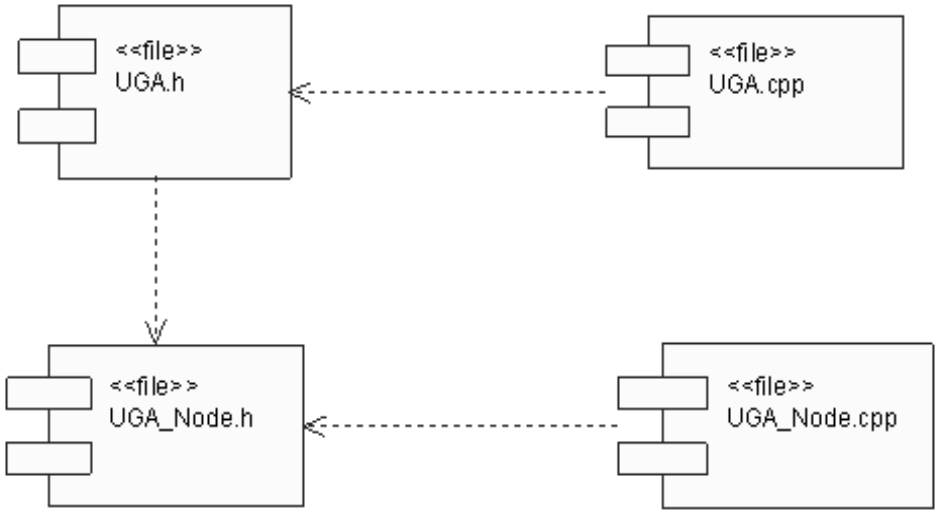
- Tabla 4 Descripción clase GA_Node.

3.5 Diagrama de componentes.

Un **diagrama de componentes** es un diagrama típico del Lenguaje Unificado de Modelado(UML).

Un diagrama de componentes representa la separación de un sistema de *software* en componentes físicos (por ejemplo archivos, cabeceras, módulos, paquetes, etc.) y muestra las dependencias entre estos componentes.

Debido a que estos son más parecidos a los diagramas de casos de usos estos son utilizados para modelar la vista estática de un sistema. Muestra la organización y las dependencias entre un conjunto de componentes. No es necesario que un diagrama incluya todos los componentes del sistema, normalmente se realizan por partes. Cada diagrama describe un apartado del sistema.



- Figura 11 Diagrama de componentes.

Capítulo 4: Pruebas al módulo de algoritmo genético.

Introducción:

En este capítulo se le hacen pruebas al módulo de algoritmo genético. Haciendo uso de un ejemplo o demo para garantizar la funcionalidad del mismo. Se prueba el rendimiento así como también la complejidad del algoritmo.

4.1 Funcionamiento del ejemplo.

Demo: Para probar el funcionamiento del módulo de algoritmo genético se realizó un ejemplo en el cual los individuos son círculos cuyo color está representado por la paleta de colores RGB (RVA en español) que consta, básicamente, de tres colores **primarios aditivos**: **R**ojo-**V**erde-**A**zul. Lo que nos permitirá el algoritmo genético es obtener el círculo de mayor color azul.

La tonalidad de cada color es un número entre 0----255.

Población Inicial: aleatorio o entrada manualmente.

Función de Evaluación: cantidad azul – (cantidad verde + cantidad rojo).

Mutación: los colores están representados de la siguiente manera:

R	G	B

Se escoge aleatorio un número entre 0—2. Que será la posición del gen que va a mutar.

Se escoge un número entre 0---255 que es el valor por el que va a mutar el gen escogido.

Cruzamiento: tengo dos individuos y escojo un número aleatorio entre 1—3 que será la cantidad de genes a cruzar y luego según ese número será la cantidad de números aleatorios entre 0—2 que se generarán para saber él o los genes a cruzar.

R	G	B
R1	G1	B1

Selección: Se seleccionan los n elementos de mayor función de evaluación.

4.2 Rendimiento.

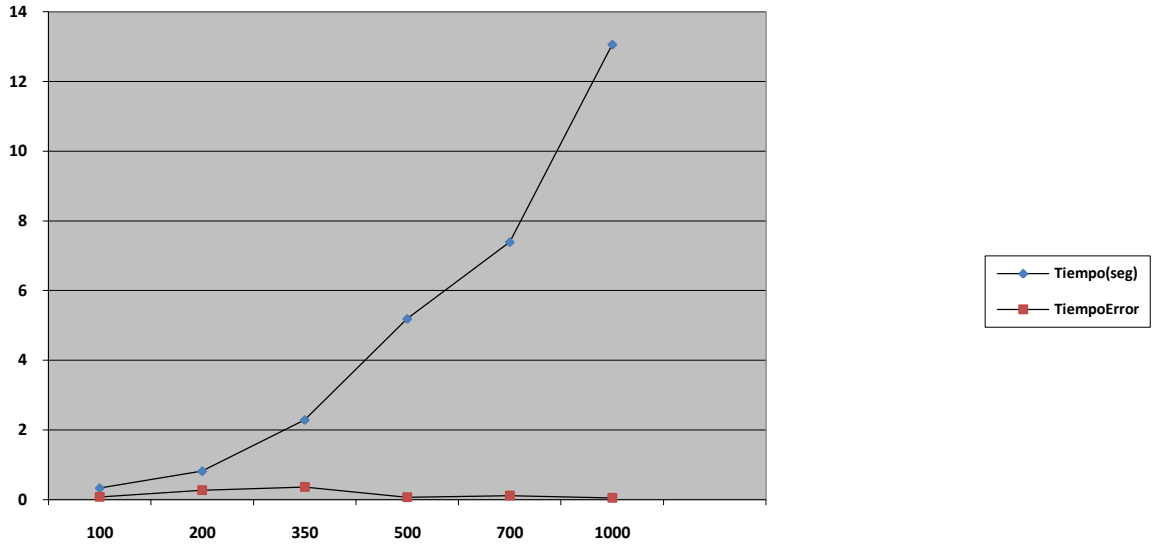
Constantes: número de iteraciones=100.

Error=150.

Población	Tiempo(seg)	Tiempo(seg)con Error	Error=150
100	0,328	0,078	
200	0,812	0,266	
350	2,282	0,360	
500	5,187	0,063	
700	7,390	0,109	
1000	13,062	0,047	

- Tabla 5 Población y tiempos.

Para probar el rendimiento del módulo se han hecho pruebas donde se han podido confeccionar algunas tablas. La primera prueba se muestra en la Tabla5 se puede ver como se comportan los tiempos al mantener constantes el número de iteraciones e ir variando el número de elementos en la población. Se han tomado dos tiempos uno sin el error es decir el algoritmo solo para por las iteraciones y otro tiempo con el error. Lo que ha demostrado que con el primer tiempo a medida que aumenta la población aumenta el tiempo. Con el segundo tiempo, este tomado con un error de 150 lo que se demuestra es que demora muy poco ya que llega muy rápido al error, es decir a mejorar la población sin llegar a completar las 100 iteraciones. Se ha tomado un error de 150 ya que se acerca mucho a una buena solución.

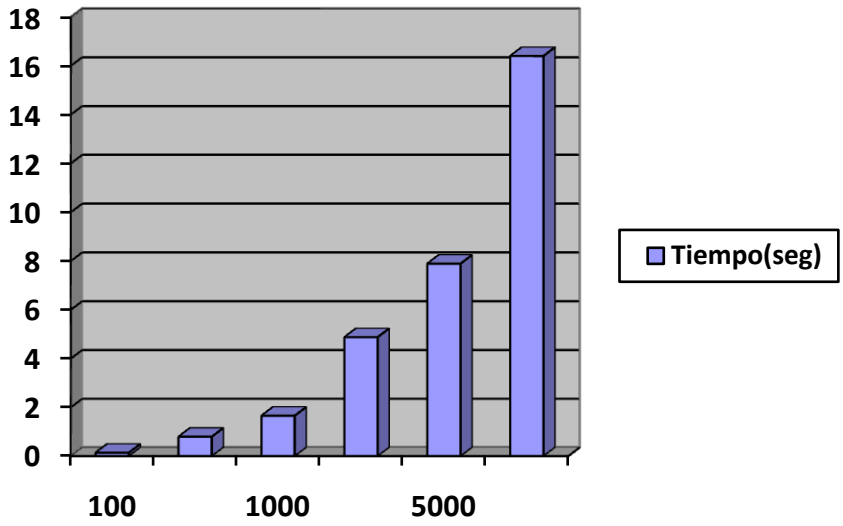


- Figura 12 Población y Tiempos.

En una segunda prueba se tiene el número de iteraciones y tiempo como muestra la figura y la tabla de iteraciones. Se puede ver como a medida que aumentan las iteraciones aumenta el tiempo de demora del algoritmo.

#iteraciones	Tiempo(seg)
100	0,141
500	0,797
1000	1,657
3000	4,876
5000	7,891
10000	16,404

- Tabla 6 Número de iteraciones y Tiempo.



- Tabla 7 Iteraciones y Tiempo.

4.3 Complejidad del algoritmo genético.

La complejidad del algoritmo genético en general va a ser igual a la suma de las complejidades de los métodos:
aplicar mutación, cruzamiento y selección:

Complejidad de la mutación: es un O(n) donde n es el numero de elementos que mutan.

Complejidad del cruzamiento: es la mayor complejidad de los 3 tipos de cruzamiento por tanto es la del cruzamiento todos contra todos que es un O(n^2) donde n es el numero de elementos.

Complejidad de la selección: O(n*m) donde n es el número de individuos que tengo en total y m el número de individuos a seleccionar.

La complejidad general es la suma de aplicar mutación, cruzamiento y selección es O(n^2) por ser la mayor entre O(n*m), O(n), O(n^2).

Conclusiones.

Una vez concluida la investigación previa al estudio de los algoritmos genéticos, se comprobó en la práctica que la utilización de los algoritmos genéticos en los proyectos relacionados con la realidad virtual en el desarrollo de juegos es escasa. Por lo que en muchas ocasiones se esta en presencia de situaciones donde la solución más óptima se puede dar aplicando los algoritmos genéticos. Con el propósito de ofrecer una solución a esta problemática se desarrolló un módulo capaz integrarse a la biblioteca de inteligencia artificial y darnos una solución mejorada.

El módulo resultante presenta características que lo hacen el complemento idóneo para que la biblioteca de inteligencia artificial sea funcional. Algunas de estas características son:

- Separar la programación de la parte inteligente de la visualización y diseño.
- Reutilización de código.
- Dedicarle mayor tiempo a la parte inteligente del juego
- Reducción del tiempo de desarrollo.
- Facilidades para los programadores a la hora de utilizar la técnica del algoritmo genético.

Se cumplió con el objetivo de este trabajo que es desarrollar un módulo con algoritmos genéticos integrado a la biblioteca de inteligencia artificial para entornos virtuales.

Recomendaciones.

Se recomienda:

- Implementar algunas funciones predefinidas para mayor facilidad al programador a la hora de hacer uso del módulo de algoritmo genético. Funciones como la de evaluación.

Referencias Bibliográficas.

1. **Herrera, Francisco.** *Técnicas de computación Flexibles.* Santiago de Compostela : s.n. Vol I.

2. *Algoritmos genéticos y Problemas de Visibilidad.* **Parejo, Carmen Cortez.** Sevilla : s.n., 1996.

3. *Algoritmos genéticos.* **Verdejo, Adrian Juan.** I, 2007 : s.n.

4. **Michalewicz.** *Genetic Algorithms+ Data Structure.*

5. **Haupt, Randy L and.** *Practical genetic algorithms.* 2004.

6. **GoldBerg, David E.** *Genetic Algorithms.* junio 1989.

7. **Reinhold, Van Nostrand.** *Handbook of Genetic Algorithms.* 1991.

8. *Travelling Salesman problemusing genetic algorithms.* http://www.lalena.com/ai/TSP

9. **lopez, Jesus Alfonso.** *Home page sobre Algoritmos geneticos.* 2000.

http://members.tripod.com/jesus_alfonso_lopez/AgIntro.html]

10. **Mitchell, Melanie.** *An introductions to genetic algorithms.* 2004.

11. **Vose, michael D.** *The simple genetic algorithms foundations and theory.* 1999.

12. **Austin, Paul Tozour_ Lon Storm.** *AI Game Programming Wisdom.*

13. **Coley, David.** *An introductions to genetic Algorithms for Scientist an engineers.* 1999.

14. **Tomasz.** *genetic algorithms.* 2004.

15. **David M. Bourge, Glenn Seeman.** *AI for Game Developers.* s.l. : O´Reilly, julio 2004.

16. *Definición del modelo del negocio y modelo del dominio.* **Dapena, MSc Martha D. Delgado.**

17. *State Diagram in UML.*

http://www.inf.udec.cl/revista/ediciones/edicion8/Rbc.pdf

http://www.developer.com/design/article.php/2247041

Bibliografía consultada.

1. Genetic Algorithms in Plain English. http://www.ai-junkie.com/ga/intro/gat1.html

2. **James, Greg.** Using Genetic Algorithms for Game AI. [En línea] 11 October 2005 .

http://www.gignews.com/gregjames1.htm

3. **David Goldberg, Addison Wesley.** Genetic Algorithms in Search, Optimization & Machine Learning . [En línea] 1989.

4. **Marks, Robert E.** Playing Games with Genetic Algorithms. [En línea] http://www.agsm.edu.au/~bobm/papers/shu.html

5. **Riechmann, Thomas.** Genetic algorithm learning and evolutionary games. [En línea]

6. **Davis, Lawrence "David".** Genetic Algorithms and their applications. [En línea]

http://www.informatics.indiana.edu/fil/CAS/PPT/Davis/

7. **Bouchard, Mark F. Bramlette and Eugene E.** Index of Most Important Applications of the Genetic Algorithms. [En línea]

http://www.lcc.uma.es/~ccottap/semEC/cap03/cap_3.html

Herrera, Francisco. *Técnicas de computación Flexibles.* Santiago de Compostela : s.n. Vol I.

2. *Algoritmos genéticos y Problemas de Visibilidad.* **Parejo, Carmen Cortez.** Sevilla : s.n., 1996.

3. *Algoritmos genéticos.* **Verdejo, Adrian Juan.** I, 2007 : s.n.

4. **Michalewicz.** *Genetic Algorithms+ Data Structure.*

5. **Haupt, Randy L and.** *Practical genetic algorithms.* 2004.

6. **GoldBerg, David E.** *Genetic Algorithms.* junio 1989.

7. **Reinhold, Van Nostrand.** *Handbook of Genetic Algorithms.* 1991.

8. *Travelling Salesman problemusing genetic algorithms.* http://www.lalena.com/ai/TSP

9. **lopez, Jesus Alfonso.** *Home page sobre Algoritmos geneticos.* 2000.

http://members.tripod.com/jesus_alfonso_lopez/AgIntro.html]

10. **Mitchell, Melanie.** *An introductions to genetic algorithms.* 2004.

11. **Vose, michael D.** *The simple genetic algorithms foundations and theory.* 1999.

12. **Austin, Paul Tozour_ Lon Storm.** *AI Game Programming Wisdom.*

13. **Coley, David.** *An introductions to genetic Algorithms for Scientist an engineers.* 1999.

14. **Tomasz.** *genetic algorithms.* 2004.

15. **David M. Bourge, Glenn Seeman.** *AI for Game Developers.* s.l. : O´Reilly, julio 2004.

16. *Definición del modelo del negocio y modelo del dominio.* **Dapena, MSc Martha D. Delgado.**

17. *State Diagram in UML.*

http://www.inf.udec.cl/revista/ediciones/edicion8/Rbc.pdf

http://www.developer.com/design/article.php/2247041

Glosario de abreviaturas.

IA: Inteligencia Artificial.

AG: Algoritmos Genéticos.

UCI: Universidad de las Ciencias Informáticas.

RV: Realidad Virtual.

UML: Lenguaje Unificado de Modelado.

RVA: consta, básicamente, de tres colores **primarios aditivos**: **R**ojo-**V**erde-**A**zul.

Anexos

- **Anexo (juego de Roles).**

```
#define kAttackedbyFighter                    0
#define kAttackedbyWizard                     1
#define kAttackedbyGroup                      2
#define kHealerPresent                        3
#define kAttackedByBlade                      4
#define kAttackedByBlunt                      5
#define kAttackedByProjectile                 6
#define kAttackedByMagic                      7
#define kAttackerWearingMetalArmor            8
#define kAttackerWearingLeatherArmor          9
#define kAttackerWearingMagicArmor            10
#define kImInGroup                            11
```

```
#define kRetreat            0
#define kHide               1
#define kWearMetalArmor     2
#define kWearMagicArmor     3
#define kWearLeatherArmor   4
#define kAttackWithBlade    5
```

```
#define kAttackWithBlunt            6
#define kAttackWithMagic          7



#define kChromosomes   12
Class ai_Creature
{
      public:
      int chromosomes[kChromosomes];
      ai_ Creature ();
      ~ai_ Creature ();
};


#define kChromosomes       12
#define kPopulationSize   100
Class ai_Creature
{
public:
      int chromosomes[kChromosomes];
      ai_ Creature ();
      ~ai_ Creature ();
```

```
};
ai_Creature   population[kPopulationSize];


void ai_ Creature::createIndividual(int i)
{
   switch (Rnd(1,5)) {
       case 1:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=kRetreat;
        break;
       case 2:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=kHide;
       break;
       case 3:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                          kAttackWithBlade;
       break;
       case 4:
          ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                          kAttackWithBlunt;
       break;
       case 5:
         ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                           kAttackWithMagic;
       break;
       }
```

```
    switch (Rnd(1,5)) {
       case 1:
          ai_Creature[i].chromosomes[kHealerPresent]=kRetreat;
       break;
       case 2:
          ai_Creature[i].chromosomes[kHealerPresent]=kHide;
       break;
       case 3:
          ai_Creature[i].chromosomes[kHealerPresent]=
                                                    kAttackWithBlade;
       break;
       case 4:
         ai_Creature[i].chromosomes[kHealerPresent]=
                                                    kAttackWithBlunt;
      break;
      case 5:
         ai_Creature[i].chromosomes[kHealerPresent]=
                                                    kAttackWithMagic;
      break;
      }



switch (Rnd(1,5)) {
        case 1:
           ai_Creature[i].chromosomes[kAttackedByBlade]=kRetreat;
```

```
   break;
   case 2:
      ai_Creature[i].chromosomes[kAttackedByBlade]=kHide;
   break;
   case 3:
      ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                        kWearMetalArmor;
   break;
   case 4:
      ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                        kWearMagicArmor;
   break;
   case 5:
      ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                    kWearLeatherArmor;
   break;
}
switch (Rnd(1,5)) {
   case 1:
      ai_Creature[i].chromosomes[kAttackedByBlunt]=kRetreat;
   break;
   case 2:
      ai_Creature[i].chromosomes[kAttackedByBlunt]=kHide;
   break;
   case 3:
```

```
      ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                    kWearMetalArmor;
   break;
   case 4:
      ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                    kWearMagicArmor;
   break;
   case 5:
      ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                    kWearLeatherArmor;
   break;
}
switch (Rnd(1,5)) {
   case 1:
      ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                  kRetreat;
   break;
   case 2:
      ai_Creature[i].chromosomes[kAttackedByProjectile]=kHide;
   break;
   case 3:
      ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                    kWearMetalArmor;
   break;
   case 4:
```

```
         ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                    kWearMagicArmor;
      break;
      case 5:
         ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                    kWearLeatherArmor;
      break;
   }
switch (Rnd(1,5)) {
      case 1:
         ai_Creature[i].chromosomes[kAttackedByMagic]=kRetreat;
      break;
      case 2:
         ai_Creature[i].chromosomes[kAttackedByMagic]=kHide;
      break;
      case 3:
         ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                     kWearMetalArmor;
      break;
      case 4:
         ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                    kWearMagicArmor;
      break;
      case 5:
         ai_Creature[i].chromosomes[kAttackedByMagic]=
```

```
                                                        kWearLeatherArmor;
        break;
   }



switch (Rnd(1,5)) {
        case 1:
           ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                         kRetreat;
        break;
        case 2:
           ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                        kHide;
        break;
        case 3:
           ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                        kAttackWithBlade;
        break;
        case 4:
           ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                        kAttackWithBlunt;
       break;
       case 5:
          ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                        kAttackWithMagic;
```

```
    break;
}
switch (Rnd(1,5)) {
    case 1:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                          kRetreat;
    break;
    case 2:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                           kHide;
    break;
   case 3:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                           kAttackWithBlade;
    break;
    case 4:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                          kAttackWithBlunt;
    break;
    case 5:
       ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                          kAttackWithMagic;
    break;
}
switch (Rnd(1,5)) {
```

```
      case 1:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                          kRetreat;
      break;
      case 2:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                          kHide;
      break;
      case 3:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                          kAttackWithBlade;
      break;
      case 4:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                        kAttackWithBlunt;
      break;
      case 5:
         ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                        kAttackWithMagic;
      break;
   }
}


#define kChromosomes       12
#define kPopulationSize   100
```

```
Class ai_Creature
{
public:
      int chromosomes[kChromosomes];
      float totalDamageDone;
      float totalDamageReceived;
      void createIndividual (int i);
      ai_ Creature ();
      ~ai_ Creature ();
};
ai_Creature   population[kPopulationSize];
```

```
#define kChromosomes       12
#define kPopulationSize   100
Class ai_Creature
{
public:
      int chromosomes[kChromosomes];
      float totalDamageDone;
      float totalDamageReceived;
      float fitness;
      void createIndividual (int i);
      void sortFitness (void);
      ai_ Creature ();
```

```
      ~ai_ Creature ();
};
ai_Creature   population[kPopulationSize];



void ai_ Creature:: sortFitness (void)
{
int   i int   j;
int   k;
float temp;
for (i=0;i<kPopulationSize;i++)
        ai_Creature[i].fitness = ai_Creature[i].totalDamageDone /
                                        ai_Creature[i].totalDamageReceived;
for (i = (kPopulationSize -- 1); i >= 0; i--)
    {
       for (j = 1; j <= i; j++)
           {
             if (ai_Creature[j-1].fitness < ai_Creature[j].fitness)
                 {
                    temp = ai_Creature[j-1].fitness;
                    ai_Creature[j-1].fitness=ai_Creature[j].fitness;
                    ai_Creature[j].fitness = temp;
                    temp = ai_Creature[j-1].totalDamageDone;
                    ai_Creature[j-1].totalDamageDone =
                                        ai_Creature[j].totalDamageDone;
```

```
                    ai_Creature[j].totalDamageDone = temp;
                    temp = ai_Creature[j-1].totalDamageReceived;
                    ai_Creature[j-1].totalDamageReceived =
                                ai_Creature[j].totalDamageReceived;
                    ai_Creature[j].totalDamageReceived = temp;
                    for (k=0;k<kChromosomes;k++)
                       {
                          temp = ai_Creature[j-1].chromosomes[k];
                          ai_Creature[j-1].chromosomes[k] =
                                        ai_Creature[j].chromosomes[k];
                          ai_Creature[j].chromosomes[k] = temp;
                       }
                }
          }
   }
```

```
#define kChromosomes       12
#define kPopulationSize   100
Class ai_Creature
{
public:
      int chromosomes[kChromosomes];
      float totalDamageDone;
```

```
      float totalDamageReceived;
      float fitness;
      void createIndividual (int i);
      void sortFitness (void);
      void crossover(int i, int j, int k);
      ai_ Creature ();
      ~ai_ Creature ();
};
ai_Creature   population[kPopulationSize];



void ai_ Creature:: crossover (int i, int j, int k)
   {
        ai_Creature[i].chromosomes[0]=ai_Creature[j].chromosomes[0];
        ai_Creature[i].chromosomes[1]=ai_Creature[k].chromosomes[1];
        ai_Creature[i].chromosomes[2]=ai_Creature[j].chromosomes[2];
        ai_Creature[i].chromosomes[3]=ai_Creature[k].chromosomes[3];
        ai_Creature[i].chromosomes[4]=ai_Creature[j].chromosomes[4];
        ai_Creature[i].chromosomes[5]=ai_Creature[k].chromosomes[5];
        ai_Creature[i].chromosomes[6]=ai_Creature[j].chromosomes[6];
        ai_Creature[i].chromosomes[7]=ai_Creature[k].chromosomes[7];
        ai_Creature[i].chromosomes[8]=ai_Creature[j].chromosomes[8];
        ai_Creature[i].chromosomes[9]=ai_Creature[k].chromosomes[9];
```

```
        ai_Creature[i].chromosomes[10]=ai_Creature[j].chromosomes[10];
        ai_Creature[i].chromosomes[11]=ai_Creature[k].chromosomes[11];
        ai_Creature[i].totalDamageDone=0;
        ai_Creature[i].totalDamageReceived=0;
        ai_Creature[i].fitness=0;
   }
```

```
#define kChromosomes       12
#define kPopulationSize   100
Class ai_Creature
{
public:
      int chromosomes[kChromosomes];
      float totalDamageDone;
      float totalDamageReceived;
      float fitness;
      void createIndividual (int i);
      void sortFitness (void);
      void crossover(int i, int j, int k);
      void randomMutation(int i);
      ai_ Creature ();
      ~ai_ Creature ();
};
```

```
ai_Creature   population[kPopulationSize];




void ai_ Creature::randomMutation(int i)
{
   if (Rnd(1,20)==1)
      switch (Rnd(1,5)) {
           case 1:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=kRetreat;
           break;
           case 2:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=kHide;
           break;
           case 3:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                          kAttackWithBlade;
           break;
           case 4:
              ai_Creature[i].chromosomes[kAttackedbyGroup]=
                                                        kAttackWithBlunt;
           break;
           case 5:
             ai_Creature[i].chromosomes[kAttackedbyGroup]=
```

```
                                                        kAttackWithMagic;
         break;
         }
if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kHealerPresent]=kRetreat;
         break;
         case 2:
            ai_Creature[i].chromosomes[kHealerPresent]=kHide;
         break;
         case 3:
            ai_Creature[i].chromosomes[kHealerPresent]=
                                                         kAttackWithBlade;
         break;
         case 4:
            ai_Creature[i].chromosomes[kHealerPresent]=
                                                        kAttackWithBlunt;
         break;
         case 5:
            ai_Creature[i].chromosomes[kHealerPresent]=
                                                        kAttackWithMagic;
         break;
       }
if (Rnd(1,20)==1)
```

```
    switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackedByBlade]=kRetreat;
         break;
         case 2:
            ai_Creature[i].chromosomes[kAttackedByBlade]=kHide;
         break;
         case 3:
            ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                        kWearMetalArmor;
         break;
         case 4:
            ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                        kWearMagicArmor;
         break;
         case 5:
            ai_Creature[i].chromosomes[kAttackedByBlade]=
                                                      kWearLeatherArmor;
         break;
    }
if (Rnd(1,20)==1)
    switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackedByBlunt]=kRetreat;
         break;
```

```
        case 2:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=kHide;
        break;
        case 3:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                        kWearMetalArmor;
        break;
        case 4:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                        kWearMagicArmor;
        break;
        case 5:
           ai_Creature[i].chromosomes[kAttackedByBlunt]=
                                                        kWearLeatherArmor;
        break;
   }
if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
        case 1:
           ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                      kRetreat;
        break;
        case 2:
           ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                       kHide;
```

```
         break;
         case 3:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                        kWearMetalArmor;
         break;
         case 4:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                        kWearMagicArmor;
         break;
         case 5:
            ai_Creature[i].chromosomes[kAttackedByProjectile]=
                                                        kWearLeatherArmor;
         break;
   }
if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                         kRetreat;
         break;
         case 2:
            ai_Creature[i].chromosomes[kAttackedByMagic]=kHide;
         break;
         case 3:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
```

```
                                                        kWearMetalArmor;
         break;
         case 4:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                        kWearMagicArmor;
         break;
         case 5:
            ai_Creature[i].chromosomes[kAttackedByMagic]=
                                                        kWearLeatherArmor;
         break;
   }
if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                        kRetreat;
         break;
         case 2:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                           kHide;
         break;
         case 3:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                             kAttackWithBlade;
         break;
```

```
         case 4:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                                kAttackWithBlunt;
         break;
         case 5:
            ai_Creature[i].chromosomes[kAttackerWearingMetalArmor]=
                                                              kAttackWithMagic;
         break;
   }
if (Rnd(1,20)==1)
   switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                                kRetreat;
         break;
         case 2:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                                kHide;
         break;
         case 3:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                              kAttackWithBlade;
         break;
         case 4:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
```

```
                                                          kAttackWithBlunt;
         break;
         case 5:
            ai_Creature[i].chromosomes[kAttackerWearingLeatherArmor]=
                                                          kAttackWithMagic;
         break;
      }
   if (Rnd(1,20)==1)
      switch (Rnd(1,5)) {
         case 1:
            ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                      kRetreat;
         break;
         case 2:
            ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                      kHide;
         break;
         case 3:
            ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                      kAttackWithBlade;
         break;
         case 4:
            ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                    kAttackWithBlunt;
         break;
```

```
            case 5:
               ai_Creature[i].chromosomes[kAttackerWearingMagicArmor]=
                                                                  kAttackWithMagic;
            break;
      }
}
```