· UNIVERSIDAD CARLOS III DE MADRID ·

PROYECTO FIN DE CARRERA

2005 / 2006

HERRAMIENTAS DE LA INTELIGENCIA ARTIFICIAL BIOINSPIRADAS

FERNANDO ALONSO MARTÍN

"La naturaleza inspira la inteligencia"

A la familia, por su apoyo

A los amigos, por escucharme

Especialmente a Pedro Isasi, por hacer ver que la informática también puede ser entretenida y un camino de reflexión sobre el mundo.

ÍNDICE

1. Introducción....7
1.1 Estructura del proyecto 10

2. Simulador de Algoritmos Genéticos....12
2.1 Introducción.... 12
2.2 Pseudo-código del algoritmo principal 15
2.3 Arquitectura de la aplicación 16
2.4 Particularidades de la implementación 19
2.4.1 Parámetros Estáticos.... 19
2.4.2 Parámetros dinámicos.... 22
2.5 Como codificar problemas.... 24
2.5.1 ¿Cómo crear una función de evaluación?.... 25
3.5.1.1. Métodos de implementación necesaria.... 25
3.5.1.2 Métodos de implementación opcional.... 26
2.5.2 ¿Cómo crear una población específica? 28
2.5.3 Extendiendo la funcionalidad del simulador: crear nuevos individuos.. 29
2.6 Modo funcionamiento 30
3.6.1 La interfaz gráfica de Usuario 30
3.6.1.1 Descripción de la interfaz de usuario 30
3.6.1.2 Ejecutar un problema desde la interfaz de usuario.... 36
3.6.1.3 Crear un nuevo problema desde la interfaz de usuario.... 46
3.6.2 Ejecutando el simulador sin la interfaz gráfica 52

3. Simulador de Enjambres de Partículas 55
3.1 Introducción.... 55
3.2 Pseudo-código del algoritmo principal.... 58
3.3 Arquitectura de la aplicación 59
3.4 Particularidades de la implementación 60
3.5 Como codificar problemas.... 61
3.5.1 ¿Cómo crear una función de evaluación?.... 62
4.5.1.1. Métodos de implementación necesaria.... 62
3.6 Modo funcionamiento 64
4.6.1 La interfaz gráfica de Usuario 64
4.6.1.1 Descripción de la interfaz de usuario 64
4.6.1.2 ejecutar un problema desde la interfaz de usuario.... 70
4.6.1.3 Crear un nuevo problema desde la interfaz de usuario.... 78
4.6.2 Ejecutando el simulador sin la interfaz gráfica 83

4. Simulador de Colonias de Hormigas....85
4.1 Introducción.... 85
4.2 Pseudo-código del algoritmo principal.... 87
4.3 Arquitectura de la aplicación 89
4.4 Particularidades de la implementación 90
4.4.1 Parámetros Estáticos.... 90
4.4.2 Parámetros Dinámicos.... 91
4.5 Como codificar problemas.... 92
3.5.1.1. Métodos de implementación necesaria.... 92

3.5.1.2 Métodos de implementación opcional 92
4.6 Modo funcionamiento 93
6.6.1 La interfaz gráfica de Usuario 93
6.6.1.1 Descripción de la interfaz de usuario 93
6.6.1.2 ejecutar un problema desde la interfaz de usuario........ 98
6.6.1.3 Crear un nuevo problema desde la interfaz de usuario........ 103
6.6.2 Ejecutando el simulador sin la interfaz gráfica 108

5. Simulador de Temple Simulado 110
5.1 Introducción 110
5.2 Pseudo-código del algoritmo principal 111
5.3 Arquitectura de la aplicación 112
5.4 Particularidades de la implementación 113
5.5 Como codificar problemas 115
5.5.1 ¿Cómo codificar las posibles soluciones al problema? 115
6.5.1.1. Métodos de implementación necesaria........ 116
5.5.2 ¿Cómo evaluar las posibles soluciones? 117
6.5.2.1. Métodos de implementación necesaria........ 117
6.5.2.2 Métodos de implementación opcional 118
5.6 Modo funcionamiento 119
6.6.1 La interfaz gráfica de Usuario 119
6.6.1.1 Descripción de la interfaz de usuario 119
6.6.1.2 ejecutar un problema desde la interfaz de usuario........ 124
6.6.1.3 Crear un nuevo problema desde la interfaz de usuario........ 129
6.6.2 Ejecutando el simulador sin la interfaz gráfica 134

6. Simulador de Redes de Neuronas Artificiales 136
6.1 Introducción 136
6.2 Pseudo-código del algoritmo principal 140
6.3 Arquitectura de la aplicación 141
6.4 Particularidades de la implementación 142
6.4.1 Autonormalización de las entradas y salidas 142
6.4.2 Distintas funciones de activación 143
6.4.3 Distintos modos de entrenamiento 143
6.4.4 Distintos algoritmos de aprendizaje 143
6.4.4.1 Backpropagation 144
6.4.4.2 Aprendizaje basado en algoritmos genéticos........ 146
6.4.4.3 Aprendizaje basado en Enjambres de partículas 146
6.5 Modo funcionamiento 147
6.5.1 La interfaz gráfica de Usuario 147
6.5.1.1 Descripción de la interfaz de usuario 147
6.6.1.2 modo de entrenamiento/validación/obtencion resultados........ 152
6.6.2 Ejecutando la red sin la interfaz gráfica 160

7. Conclusiones 163

8. Bibliografía 164
Papers 164
Literatura escrita 164

Página intenciadamente
dejada en blanco.

Introducción

El objetivo de este proyecto es construir una serie de herramientas o librerías que puedan ser usadas para la resolución de cualquier tipo de problema mediante técnicas de Inteligencia Artificial inspiradas es comportamientos o fenómenos biológicos.

La observación de la naturaleza ha sido una de las principales fuentes de inspiración para la propuesta de nuevos paradigmas computacionales. La gran variedad de problemas de optimización que ésta presenta, tales como: la supervivencia, la evolución de las especies, la búsqueda del camino crítico, entre otros, sugirieron diversos algoritmos computacionales que se inspiran en procesos naturales.

Con este proyecto, lo que hemos intentando es suministrar al usuario una serie de herramientas genéricas capaces de solucionar problemas del mundo real, basándose en algoritmos de búsqueda de soluciones evolutivos e inspirados en la observación de fenómenos de la naturaleza.

Las herramientas que se proporcionan permiten directamente crear problemas que las usen. Dejando únicamente al usuario la tarea de codificar el problema de la manera adecuada para que la herramienta que suministramos pueda resolverlo.

La aplicación en si esta compuesta por una serie de herramientas que pueden ser usadas de manera conjunta o individual para resolver un problema. Las herramientas que se han implementado en esta primera versión son:

- Simulador de Algoritmos Genéticos (AG)
- Simulador de Enjambres de Partículas (PSO)
- Simulador de Colonias de Hormigas
- Temple Simulado (Simulated Annealign)
- Red de Neuronas Artificiales (RNA)

***Figura 1.** Herramientas del proyecto*

Las 5 herramientas conforman una librería de desarrollo (HerramientasIA.jar) que permite usar cada uno de ellos por separado o conjuntamente, ya que mantienen un aspecto y arquitectura similar y coherente.

Cada herramienta se ha desarrollado siguiendo una arquitectura de 3 capas; por un lado se encuentra la aplicación (el algoritmo bio-inspirado), por otro lado se encuentran los problemas que se pueden resolver (la codificación de cada problema particular) y por último se encuentra la interfaz gráfica de la aplicación.

Cada herramienta se puede usar de dos maneras fundamentales. La primera es directamente desde la interfaz gráfica de la aplicación crear el problema y visualizar su evolución directamente en la interfaz gráfica. La segunda manera es sin uso de la interfaz gráfica, creando un proyecto externo e importando la librería de nuestra herramienta. Esta segunda manera de trabajar es muy útil, puesto que en la mayoría de los casos que se plantean en la vida real se resuelven directamente invocándose desde el código y sin necesidad de visualizar la interfaz gráfica suministrada por nuestra aplicación.

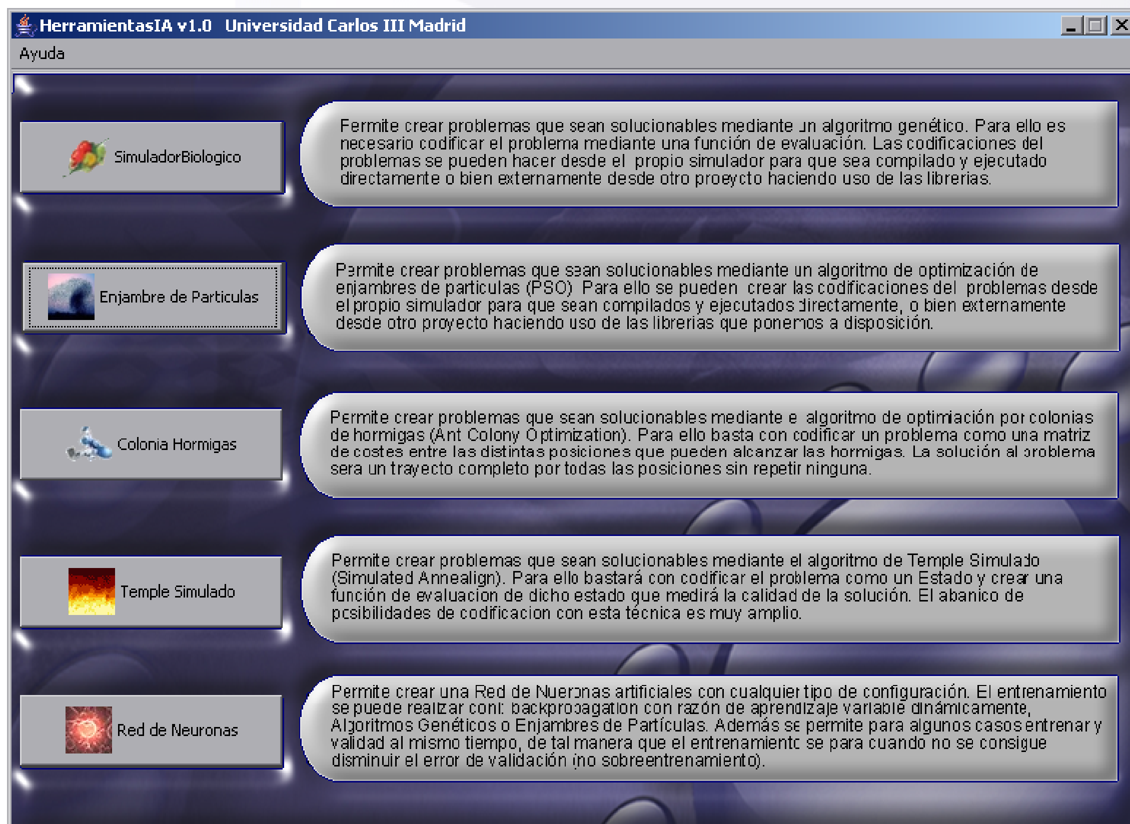


Figura 2. *Pantalla principal: selector de la herramienta*

1.1 Estructura del proyecto

La estructura del proyecto se puede ver en la siguiente pantalla:

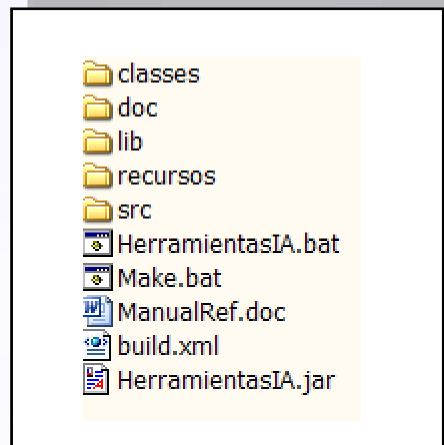


***Figura 3.** Estructura del proyecto*

El fichero principal que contiene todos los archivos necesarios para la aplicación es el fichero "**HerramientasIA.jar**", en el que se agrupan todas las herramientas desarrolladas en el proyecto y todo lo necesario para su funcionamiento, por ello constituye la librería que será necesario usar por cualquier desarrollador para usar las herramientas desarrolladas durante este proyecto.

Para recompilar la aplicación únicamente se deberá ejecutar el archivo **Make.bat**, que haciendo uso del fichero build.xml y de la herramienta Apache Ant (que no es necesario ser instalada, ya que las librerías necesarias se incluyen en el propio proyecto), compilará todas las clases, generará la documentación y creará la librería .jar.

El código fuente de la aplicación se agrupa en la carpeta "**src**". En ella, se agrupan a su vez, en distintas carpetas las distintas herramientas desarrolladas.

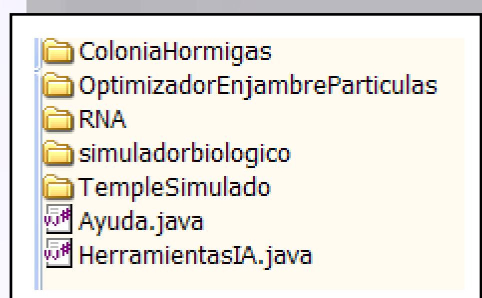


Figura 4. *Estructura de los ficheros fuente*

Cada herramienta a su vez se agrupa, en tres capas, la aplicación por un lado, la interfaz gráfica por otro lado y por último los problemas en otra carpeta (también se pueden crear problemas en cualquier otra carpeta de manera externa a la aplicación).

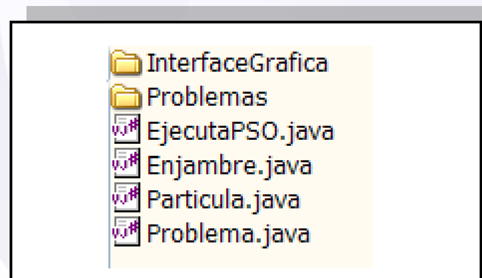


Figura 5. *Estructura en capas de cada herramienta*

En la carpeta “**classes**” se encuentran las clases compiladas del proyecto, y presenta la misma estructura que la carpeta “src”. Cada herramienta se agrupa en una carpeta, y a su vez cada herramienta se agrupa en tres vistas o capas.

La carpeta “**doc**” contiene todo el javadoc (documentación en formato html del código del proyecto), que proporciona información de todas las clases, métodos y atributos del proyecto.

En la carpeta “**recursos**” se encuentran archivos necesarios para la aplicación que son usados directamente desde el código, tales como imágenes, o archivos necesarios para los distintos problemas.

En la carpeta “**lib**” se encuentran librerías externas al proyecto pero que son necesarias. En ella se encuentran también las librerías de Ant que se usan para realizar el “make” de la aplicación.

2. Simulador de Algoritmos Genéticos

2.1 Introducción

Los algoritmos genéticos están inspirados en la teoría de la evolución de Darwin, la llamada selección natural, por la cual los individuos mejor adaptados al medio tienen mas probabilidades de reproducirse y obtener descendencia. El uso de los algoritmos genéticos para la resolución de un problema implica codificar el problema en un individuo; es decir, la posible solución del problema debe quedar determinado en el cromosoma del individuo. Los algoritmos genéticos se usan típicamente para resolver problemas muy complejos de resolver computacionalmente (por fuerza bruta) y en los que nos interesa dar una solución aproximada.

Para resolver problemas del mundo real, el usuario únicamente tendrá que preocuparse de hace una buena codificación del problema, esto es lo único tarea a realizar, pues de la simulación ya se encargara el simulador proporcionado en la herramienta que incluye este proyecto; por tanto nuestra tarea consiste en conseguir la codificación que mejor se adapte al problema.

El simulador manejará una población compuesta por individuos, sobre dichos individuos, irá aplicando en cada iteración (generación), básicamente 3 operadores: **reproducción**, **mutación** y **selección natural**. De tal manera que los mejores adaptados al problema (al contexto) son los que tienen mas probabilidades de reproducirse y los malos tienen menos posibilidades de transmitir su información genética a siguientes generaciones. Es importante que la selección natural se haga después de la mutación, para que pasen solo las mutaciones favorables.

Una buena codificación y evaluación de los individuos conseguirá que, en cada generación, sobrevivan los mejores, al aplicar el operador de selección natural, para poder cruzarlos. Mientras, los peores individuos morirán (no pasarán a la siguiente generación), no pasando sus genes a la siguiente generación. De esta manera, cada vez se irán creando individuos mejores al coger lo mejor de los 2 padres (si coge partes malas, lógicamente morirá debido a la presión selectiva). El tercer operador típico es la llamada mutación, consistente en la modificación fortuita de un gen, que aplicada a un individuo puede hacerle mejorar o empeorar; si mejora sobrevivirá, si empeora morirá

seguramente. El problema de la mutación es que debe ser baja, sino acabaríamos haciendo una búsqueda puramente aleatoria (la selección natural no valdría para nada).

Sobre estas 3 operaciones básicas caben diversas implementaciones, todas ellas válidas. Por ejemplo, para el caso de la reproducción se puede usar el algoritmo de los “Torneos” o el algoritmos de la “Ruleta”, entre otros. El método de los Torneos intenta asemejarse a la realidad, de tal manera que a la hora de reproducirse, cada individuo compite contra sólo un determinado número de individuos del entorno, y el mejor de ellos logra reproducirse; se organizan tantos torneos como individuos haya. Cuanto mayor sea el número de individuos compitiendo en cada torneo, la llamada “presión selectiva” o “presión natural”, más difícil tienen los individuos “malos” reproducirse, con el inconveniente de que a una presión natural muy elevada se puede perder la variabilidad genética y aparecer el problema de la “sobrepoblación”, es decir casi todos los individuos son iguales, debido a que descienden de los mismos padres. Suele ser conveniente, que primeramente la presión natural sea reducida, para que haya poca competencia y logren individuos no muy buenos “aparearse” para que se conserve la diversidad genética, y posteriormente ir aumentando el tamaño de dichos torneos para optimizar la solución dada por el mejor individuo.

En cuanto a la forma de aparearse, el sobrecruzamiento de la información genética, hay también varias modalidades de llevarlo a cabo. Se pueden fusionar los cadenas de cromosomas realizando un corte a la mitad de las mismas y uniéndolas, o bien se puede hacer el corte en una posición aleatoria del mismo, también se puede hacer una operación lógica sobre ambos genes, etc.

Otro factor a considerar son los individuos que pasan a la siguiente generación directamente por ser los mejores (“elitistas”). De esta manera sobrevivirán siempre al menos los mejores individuos.

En cuanto a la mutación, hemos dicho que valores altos, superiores a 0.5, pueden hacer que el problema derive en una búsqueda aleatoria. Cuando se desea aumentar la variabilidad genética para salir de un mínimo local conviene aumentar la mutación, mientras que cuando se quiere avanzar rápidamente hacia un mínimo, local o global, conviene disminuirla (explotacion vs exploración).

En conclusión, si alguien duda de la eficacia de esta manera de resolver problemas y de las teoría de Darwin (poniéndonos un poco mas filósofos), le recomendamos intente resolver problemas usando el Simulador Genético (que usa las 3 operaciones elementales) y verá como obtiene soluciones buenas, si la codificación también es buena.

2.2 Pseudo-código del algoritmo principal

```
//inicializamos los individuos de la población (dando valores a sus cromosomas)
poblacion.inicializarPoblacion( umeroIndividuos,longitudCromosoma,tipoIndividuo);


//hacemos una primera evaluación de cada individuos que compone la población
poblacion.aplicarSeleccionNatural();
actualizarParametrosDinamicos();
mostrarSimulacion(tipoSalidaResultados); //mostramos la generación 0


 do {  //cada iteración es una nueva generación

    poblacion.reproducir(presionSelectiva, elitismo,tipoSobrecruzamiento,mejorInmutable);
    poblacion.mutarPoblacion(probabilidadMutarGen);
    poblacion.aplicarSeleccionNatural();

      actualizarParametrosDinamicos();  //relativas a mejor individuo y puntuaciones

      if ( (presionSelectiva < numeroIndividuos) &&
           (iteraciones % pasosParaAumentarLaPresion == 0)) {
         presionSelectiva ++;
      }

      iteraciones++;

      if ( (iteraciones % iteracionesRedibujado) == 0) {
         mostrarSimulacion(tipoSalidaResultados);
      }

}while ( (puntuacionObtenida < puntuacionAObtener) &&
             (iteraciones < iteracionesMaximas));


mostrarResultadosFinales(tipoSalidaResultados);
```

Figura 6. *Código maestro del algoritmo genético*

2.3 Arquitectura de la aplicación

Figura 7. *Diagrama UML del paquete SimuladorBiologico*

<<Abstract>>
Individuo
double _valoracion
double _varlorMaximoPosible
double _valorMinimoPosible
inicializar()
mutar (double probabilidad)
toString() : String
cruzar (Individuo individuo) : Individuo
instanciarPoblacion (String tipoIndividuo, int tamañoCromosoma) : Individuo
equals (Object otroIndividuo): boolean

<<Interface>>
Contexto
evaluarIndividuo (Individuo ind): double

Package Individuos

IndividuoBinarioHaploide
byte[] cromosomas
inicialziar()
mutar(double prioridad)
toString(): String
cruzar(Inidivduo otro): Individuo

Thread

<<Abstract>>
Poblacion
Individuo[] _individuos
aplicarSeleccionNatural()
reproducir(int presionSelectiva, int elitismo)
inicializarPoblacion (int tamañoCromosoma)
mostrarPoblacion()
valorPoblacion(): double
mejorIndividuo(): Individuo
mtarPoblacion(double probabilidadIndiduo)
instanciarPoblacion (String tipo, int numIndividuos): Poblacion

Simulador
Poblacion _poblacion
int numIndividuos
int longitudCromosoma
int presionSelectiva
double probabilidadMutarGen
int pasosParaAumentarLaPresion
int elitismo
double puntuacionObtener
int iteracionesMaxiimas
int iteraciones
double puntuacionObtenida
double puntuacionPoblacionMedia
run()
mostrarResultados(String tipoSalida)
inicializar()
parar()
sincroncizarConGrafica()
despertar()

Package Problema Electoral

PoblacionElectoral
aplicarSeleccionNatural()
reproducir (inint presionSelec, int elitismo)

ContextoElectoral
evaluarIndividuo(Individuo individuo)

Contexto

Reprocuccion
torneos (Individuo[] individuos,
int presionSelectiva,
int elitismo)
static sobrecruzamiento (cromosoma1,
cromosoma2,
tipoCruce)

Asociacion/Agregación
Uso
Generalización (extiende)
Implementación

Figura 8. *Diagrama UML simplificado del paquete SimuladorBiologico*

Las principales clases, clases abstractas e interfaces a tener en cuenta son las siguientes:

Individuo: clase que declara como debe ser un individuo, es abstracta y deja para la implementación de los individuos concretos los métodos de inicializar, mutar, mostrarse, y cruzar; Proporciona un método estático que permite instanciar individuos concretos a modo de factoría. Como atributo que poseen todos los individuos es la valoración (en función de la evaluación que se les hace).

Población: clase abstracta que declara como debe ser toda población concreta, deja para la implementación de las poblaciones específicas la manera de aplicar la selección natural y reproducirse, proporciona implementación genérica para inicializarse, mostrarse, y evaluar el valor conjunto de la población. Tiene como atributo principal los distintos individuos que la conforman.

Contexto o Problema: interfaz genérica que declara el método de evaluar un individuo en función de su cromosoma, así en cada contexto específico un individuo será evaluado de distintas formas (según la implementación de dicho método o la también llamada función de fitness).

Reproducción: Clase que contiene diversas técnicas de reproducción que pueden ser usadas por las distintas poblaciones concretas para reproducirse, y los métodos que tienen los individuos para realizar el sobrecruzamiento entre sus cromosomas.

Simulador: Esta clase lanza un hilo de ejecución que se encarga de aplicar la selección natural, mutar y reproducir la población consiguiendo evolucionarla buscando una solución al problema en función de la función de evaluación. Implementa el código maestro del algoritmo genético visto en la sección "Pseudo-código del algoritmo principal".

2.4 Particularidades de la implementación

Los parámetros ajustables desde el simulador biológico son de dos tipos: estáticos y dinámicos. Los parámetros estáticos se configuran antes de iniciar la simulación y son invariables mientras que no acabe la simulación, mientras que los parámetros dinámicos pueden ser variados, en tiempo real, durante la simulación.

2.4.1 Parámetros Estáticos

- **Tipo de problema**: Escogemos el tipo de problema a intentar solucionar de entre todos los que tenemos implementados.
- **Tipo de individuo**: Escogemos el individuo que usaremos para resolver el problema de entre los que se nos ofrecen (implementados de momento están BinarioHaploide, BinarioDiploide, EnteroHaploide, PositivoHaploide, DecimalHaploide, DecimalDiploide).
- **Tipo de solución (individuo/población)**: Si elegimos individuo, la puntuación del individuo es la solución y si elegimos población, la puntuación será la suma de todos los individuos de dicha población; si dicho valor es mayor o igual que la puntuación obtenida el algoritmo genético se parará.
- **Puntuación obtener**: Es la puntuación a obtener para la solución, y que el algoritmo genético se pare.
- **Pasos máximos**: Número de generaciones máxima a generar antes de que se pare el algoritmo genético.
- **Tamaño de la población**: especifica el número de individuos que tendrá la población que resuelva el problema elegido.
- **Tamaño del cromosoma**: Es el número de genes que tendrá el cromosoma de los individuos que forman parte de la población (para el caso de individuos diploides será el de la mitad de sus cromosomas, ya que posee dos "hilos").

- **Tipo de sobrecruzamiento**: Escogemos el tipo de sobrecruzamiento con el cual se cruzaran dos individuos para generar uno nuevo. Los implementados hasta ahora son:
  - *Partir por la mitad*: el nuevo individuo es fruto de una mitad del cromosoma de un padre y otra mitad del cromosoma del otro padre.
  - *Partir aleatorio*: igual que el anterior solo que no se tiene porqué partir a la mitad sino en trozos de distinto tamaño.
  - *Especializado*: el individuo hijo tendrá en cada gen el de un padre o el del otro. Se decide para cada gen de que padre lo cogemos.
  - *Ninguno*: no se realiza sobrecruzamiento, es decir el problema evoluciona pero sin cruces (solo selección y mutación).
- **Presión selectiva**: Es el número de individuos contra los que tendrá que competir cada individuo para tener la posibilidad de tener descendencia. Cuanto mayor sea la presión menos posibilidades tiene de tener descendencia, ya que la competencia será mayor. En generaciones iniciales se aconseja una presión selectiva baja, para que no aparezcan súper individuos que sean los únicos que se reproduzcan, con la consiguiente perdida de diversidad genética en la población, para posteriormente irla aumentando.
- **Aumento presión**: Se refiere al número de generaciones que tienen que pasar para que se aumente en uno la presión selectiva.
- **Elitismo:** Hace referencia al número de individuos que pasarán directamente a la siguiente generación simplemente por ser los mejores, también tendrán la posibilidad de reproducirse, pero ellos no morirán y pasarán directamente a la siguiente generación; con estos conseguimos no perder la mejor solución obtenida hasta el momento. Estos individuos que pasan a la siguiente generación directamente podrá decidirse si son inmutables (no podrán sufrir mutación durante la inmediata siguiente generación con el objetivo de que no se pierda la mejor solución), o si se permite dicha mutación con el objetivo de que con dicha mutación se obtenga una mejor solución.

- **Mutar gen**: Es la probabilidad que tiene cada gen de cada individuo de sufrir una mutación. Una mutación alta hace la búsqueda casi aleatoria, pues deja de funcionar la selección natural, es decir, el valor medio de la población desciende, y no hay garantías de que al cruzarse uno bueno con otro de la población la solución se mantenga, lo mas normal es que empeore. Por tanto niveles de mutación bajas hacen converger rápidamente hacia buenos valores, pero por el contrario llegados ha este punto de estancamiento conviene aumentarlos ligeramente para conseguir otro individuo aún mejor.

- **Aumento mutación progresiva**: Consigue que la mutación vaya aumentando progresivamente avanza la simulación.

- **Ayuda**: Información textual sobre el problema escogido, en el que se explica la codificación del problema así como los parámetros de la simulación como hay que ajustarlos para ese problema.

- **Tipo de salida resultados**: Escogemos sobre que soporte se mostrarán los resultados de la simulación, bien en modo Gráfico (en la interfaz gráfica), en el Terminal, o bien en Fichero. Claro está que la salida gráfica es más costosa que la salida por terminal o fichero ya que crea otro hilo que tiene que ejecutar la gráfica sincronizándose con el hilo que ejecuta la simulación. Por tanto para obtener velocidad se recomienda la salida por terminal.

- **Activar Gráfica**: Si esta escogida la salida Gráfica, se podrá además activar una gráfica que muestre como va evolucionando la puntuación (fitness) de la solución hasta el momento.

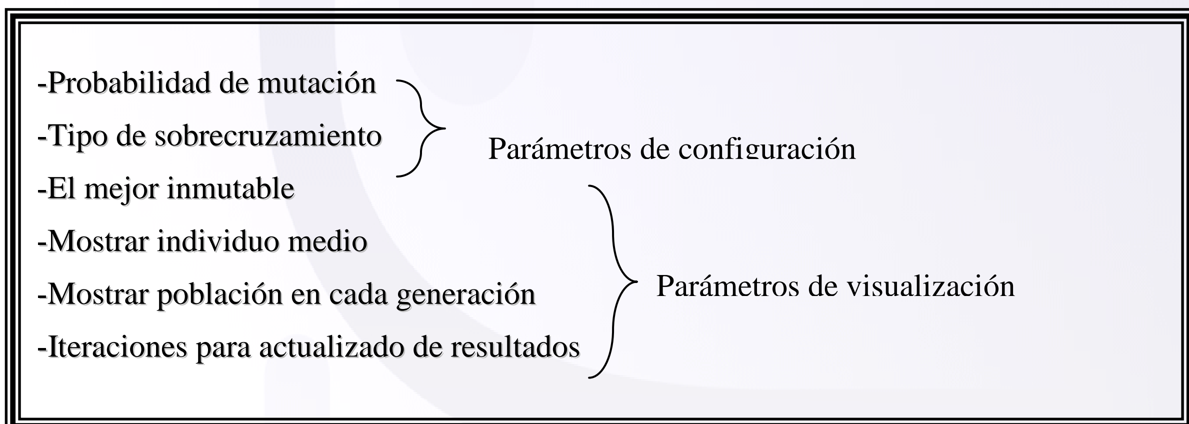
- **Mostrar individuo medio**: Se calcula el individuo medio (puntuación total de la población entre el nº de individuos) y se muestra, si además está activada la Gráfica se dibujará también la gráfica del individuo medio. Activarlo penaliza algo el rendimiento pues debe realizar más cálculos.

- **Mostrar población cada generación**: Si la salida es en modo terminal o en modo Fichero se podrá activar, de modo que se muestren todos los individuos que componen cada generación. Activarlo puede permitir ver todos los individuos, para

problemas con muchísimos individuos se hace inservible y confuso, a la vez que provoca una mayor salida de resultados ralentizando la simulación.

2.4.2 Parámetros dinámicos

Como la vida misma, al iniciarse la vida se tenía unos determinados valores de mutación, presión selectiva, formas de reproducirse… pero con el paso del tiempo todos estos factores pueden variar. Para simular esto, hemos conseguido que en tiempo de simulación dichos parámetros sean reconfigurables. Los parámetros que pueden reconfigurarse dinámicamente son los siguientes:



Mutación y “el mejor inmutable” tienen que ver con parámetros ajustables de la simulación, mientras que los demás tienen que ver en como visualizar los resultados de dicha simulación.

Por ejemplo, lanzamos la simulación y nos damos cuenta de que la mutación es muy pequeña y consigue rápidamente converger hacia un individuo bastante bueno, pero de ahí no mejora, una solución es aumentar manualmente la mutación, de esta manera es posible que los individuos encuentren alguna solución mejor.

Muy en consonancia con la mutación, también va el parámetro que determina si es o no susceptible de ser mutado el mejor (“el mejor inmutable”), con esto conseguimos que durante la generación que el individuo es el mejor no pueda ser mutado, para no perder sufrir una mutación desfavorable que empobrezca la puntuación; muchas veces en tiempo de ejecución si vemos que esta disminuye será mejor activarla de tal manera que no pueda mutar; si por el contrario no estando activada vemos que la solución está

estancada una solución puede ser desactivarla para que el mejor individuo pueda mutar y obtenerse una mejor solución (posiblemente aumentando también la prob. de mutación). Con mostrar individuo medio, podemos ver los valores medios de la población y como por selección natural se van acercando a la puntuación del mejor individuo, cuanto más cerca esta la población del mejor individuo (cuanta mas puntuación media hay) mas probabilidades tiene el mejor individuo de mejorar su solución, al cruzarse con individuos cada vez mejores. Esta información nos puede ser útil pero por el contrario penaliza en tiempo la simulación.

2.5 Como codificar problemas

En esta sección vamos a explicar como implementar la codificación de un problema usando algoritmos genéticos en nuestra aplicación, que básicamente consiste en la implementación de los métodos abstractos y la sobrescritura opcional de los métodos que no son abstractos de las dos clases abstractas: Problema y Población.

La codificación del problema se implementa mediante una función de evaluación (función de fitness) que soportará una serie de individuos. Podrán además darse los valores por defecto con los que se resuelve el problema mediante la simulación, como son el tamaño del cromosoma, número de iteraciones, puntuación a obtener, etc.

Finalmente habrá que implementar la población con la que se resolverá el problema, que puede ser simplemente decir a la aplicación cuales son los valores máximos y mínimos que puede tomar los genes de los cromosomas de los individuos; pudiendo llegar también a implementar (sobrescribiendo) la manera en que se realiza la selección natural y/o la reproducción.

2.5.1 ¿Cómo crear una función de evaluación?

La tarea de construir una función de evaluación, es la tarea de transformar un problema del mundo real en un problema de optimización de un array de datos, gracias al proceso de recombinación que proporciona el algoritmo genético.

3.5.1.1. MÉTODOS DE IMPLEMENTACIÓN NECESARIA

Para la codificación del problema es necesario al menos implementar los tres métodos declarados abstractos en la clase Problema, que son:

```
public double evaluarIndividuo (Individuo individuo)
public String[] individuosSoporta()
public boolean isMaximizationProblem()
```

Figura 9. *Métodos que deben ser implementados necesariamente en cada problema*

La función de fitness es “evaluarIndividuo”, en ella se deberá valorar el individuo en función de los valores de su cromosoma. Hay que tener en cuenta que se pueden tener distintos tipos de individuos para resolver el problema.

Así pues el código de la función de evaluación deberá ser del tipo:

```
public double evaluarIndividuo(Individuo individuo){
   if (individuo instanceOf tipoIndividuo){
      …
      return valoración;
   }else if (individuo instanceOf tipoIndividuo){
      …
      return valoración;
   }
}
```

Figura 10. *Estructura típica de la función de evaluación*

El método individuosSoporta deberá devolver en un array de String los individuos que soporta la función de evaluación implementada. Es decir para los tipoIndividuo que se han contemplado en la función de evaluación.

```
public String[] individuosSoporta(){
  String[] individuosSoportados =
     {"Binario Haploide","Binario Diploide"};
  return individuosSoportados;
}
```

Figura 11. *Ejemplo de implementación del método individuosSoporta()*

Por último, el metodo isMaximizationProblem, nos sirve para indicar si lo que queremos es un problema de maximización o de minimización. Para ello, simplemente habrá que devolver "*true*" si queremos maximizar o "*false*" si queremos minimizar.

```
public String[] isMaximizationProblem(){
  return true;
}
```

Figura 12. *Ejemplo de implementación del método isMaximizationProblem()*

3.5.1.2 MÉTODOS DE IMPLEMENTACIÓN OPCIONAL

Los métodos de implementación opcional tienen una implementación por defecto dada en la clase Problema, si bien, suele ser importante darle una implementación adaptada el problema en concreto. Son los siguientes métodos:

```
public String interpretacionDelIndividuo(Individuo individuo)
public double puntuacionAObtenerPorDefecto()
public int numeroIndividuosPorDefecto()
public int longitudCromosomaPorDefecto()
public double probabilidadMutarGenPorDefecto()
public int numIterParaAumentarPresionSelectivaPorDefecto()
public int elitismoPorDefecto()
public String tipoSobrecruzamientoPorDefecto()
public boolean mejorInmutablePorDefecto()
public int presionSelectivaPorDefecto()
public int iteracionesMaximasPorDefecto()
public boolean individuoPuedeSerSolucion()
public boolean poblacionPuedeSerSolucion()
public String obtenerAyuda()
```

Figura 13. *Métodos que pueden ser sobrescritos en cada nuevo problema*

Todos estos métodos sirven para obtener información sobre el problema, y tomarse en el caso de que no sean pasados otros valores en el momento de iniciar la simulación. En la interfaz gráfica, al seleccionar un problema, se mostrarán los valores por defecto de dicho problema, establecidos con estos métodos.

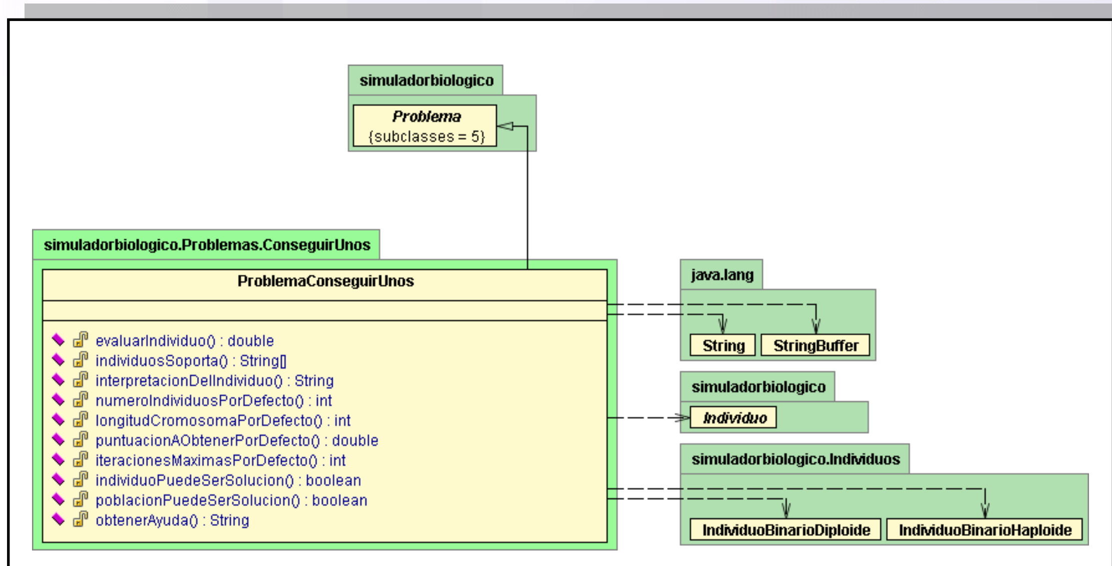


Figura 14. *Diagrama de implementación de un problema*

2.5.2 ¿Cómo crear una población específica?

Debemos indicar cual es el valor máximo y mínimo que podrá alcanzar cada gen del individuo. Para ello se deben implementar los dos métodos que se dejan como abstractos en la clase Población.

```
public double valorMinimoGenIndividuos()
public double valorMaximoGenIndividuos()
```

Figura 15. *Métodos que deben ser implementados necesariamente en cada población*

Además se podrá sobrescribir la manera en que se realiza la selección natural y la reproducción, para poderse realizar como se quiera. Por defecto, sino se sobrescriben estos métodos, la selección natural se hace valorando cada individuo con la función de evaluación antes creada para este problema y la reproducción se hace mediante el algoritmo de torneos.

```
public void aplicarSeleccionNatural()
public void reproducir(bolean isMaximizationProblem,
                       int presionSelectiva,
                       int elitismo,
                       String tipoSobrecruzamiento,
                       boolean mejorInmutable) throws Exception
```

Figura 16. *Métodos que pueden ser sobrescritos en cada nuevo problema*

2.5.3 Extendiendo la funcionalidad del simulador: crear nuevos individuos

En algún problema concreto que se quiera implementar puede que no nos sea suficiente con los individuos que se proporcionan implementados por defecto, por lo que puede convenir crearse un nuevo tipo de individuo.

Para crear un nuevo individuo se crea una clase que extienda Individuo, y debe proporcionar implementación para los métodos que quedan abstractos en Individuo que son: inicializarse, mostrarse, mutar y toString. Aparte se deberá en el método de la clase individuo que instancia individuos meter un par de líneas para que se pueda devolver dicho tipo de individuo.

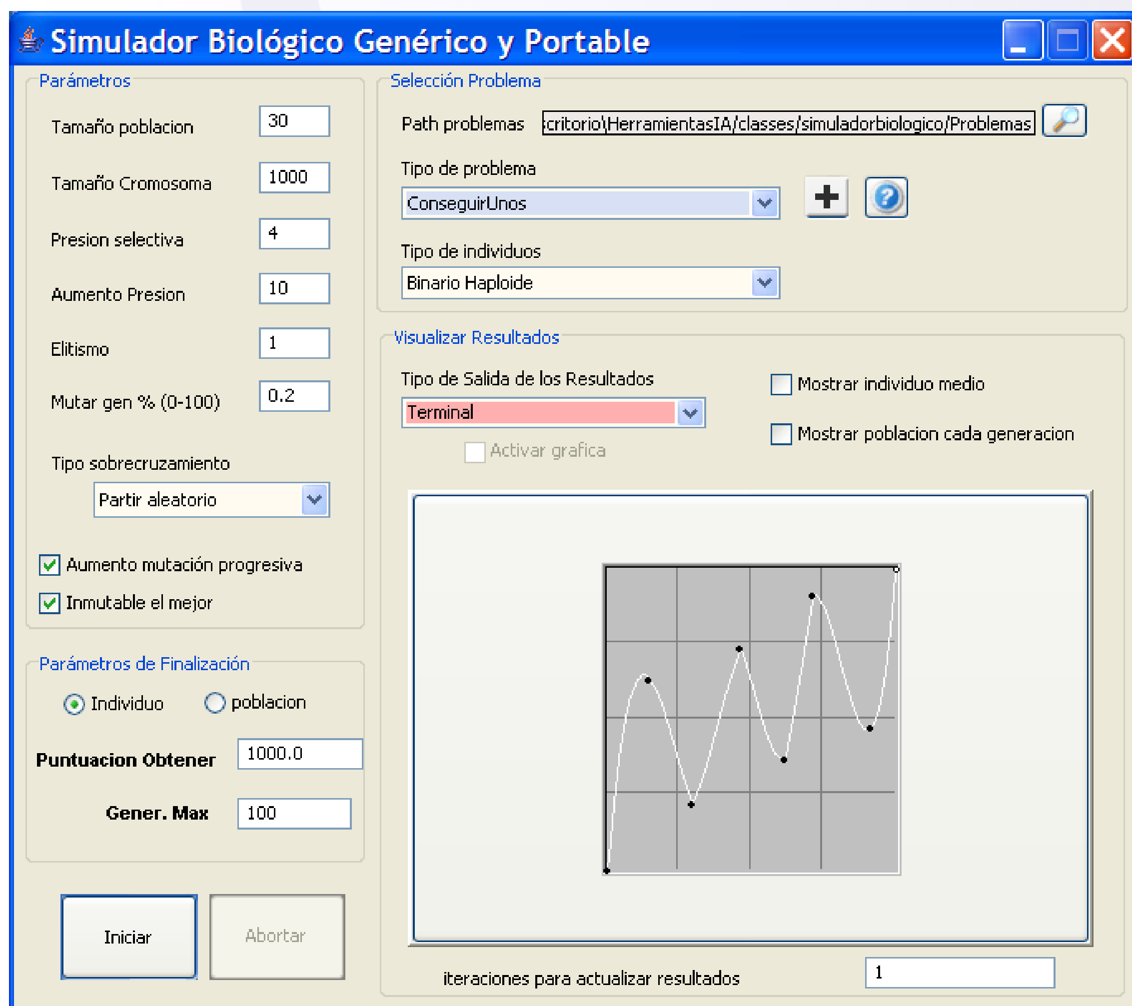
2.6 Modo funcionamiento

Existen principalmente dos maneras de usar la aplicación: mediante la interfaz gráfica, o usándose directamente desde otro programa externo. La ventaja de usar la interfaz de usuario, es poder ver directamente como evoluciona el resultado gráficamente, e inclusive modificar en tiempo de ejecución ciertos factores como la mutación o el tipo de sobrecruzamiento. A veces, no interesa hacer uso de la interfaz gráfica, y solamente nos interesa llamar al método que ejecuta la simulación sin mostrar interfaz gráfica. Vamos a presentar las dos maneras de usar la aplicación.

3.6.1 La interfaz gráfica de Usuario

3.6.1.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

Figura 17. *GUI inicial*



Esta es la vista que presenta la interfaz gráfica de usuario de la herramienta proporcionada para los Algoritmos Genéticos. Esta dividida en cuatro panes principales: parámetros de la simulación, parámetros de finalización de la simulación, selección del problema y panel de visualización de resultados.

En el panel llamado "**Parámetros**" , situado arriba a la izquierda se puede seleccionar el valor que tendrán los parámetros del algoritmo genético, tales como el número de individuos en la población (en cada generación), el tamaño del cromosoma de los individuos (número de genes o alelos), el número de individuos contra los que tiene que competir para reproducirse en cada generación cada individuo, cada cuantas iteraciones se aumenta en un uno la presión selectiva, cuantos individuos pasan directamente a la siguiente generación por ser los mejores, la probabilidad de mutar cada gen en cada generación, el tipo de sobrecruzamiento a realizar, determinar si se aumenta progresivamente la mutación y por último determinar si el mejor individuo esta expuesto también a mutación o es invariable (si se puede reproducir, pero no mutar).

En el panel "**Parámetros de Finalización**" se debe elegir cual es la puntuación máxima a obtener (referida al individuo o a la suma total de puntuaciones de los individuos de la población), y las generaciones máximas a evaluar, es decir, el número máximo de iteraciones del algoritmo.

El panel "**Selección del Problema**" se encarga de cargar los programas compilados previamente o crear directamente problemas nuevos para ser compilados y ejecutados desde la propia interfaz o de manera externa. La ruta donde de donde se cargan los problemas y en donde se crean puede ser introducida en el recuadro textual que tiene a su derecha una lupa. Si la ruta no se cambia, cargará los problemas que vienen implementados con el proyecto, si se indica otra ruta, el cargador de clases intentará cargar los problemas compilados que se encuentren en dicha ruta, pinchando en la lupa.

Para crear nuevos problemas, en la ruta indicada, se pedirá un nombre del problema, y se analizará dicha ruta; si la ruta era la original, se creará el problema dentro del propio proyecto, y se mostrará una ventana desde la que se puede trabajar con el código y compilarlo. Si el código queda compilado correctamente se podrá directamente usar desde la interfaz gráfica (sin tener que cerrar y volver a lanzar la

aplicación). Posteriormente el código puede ser otra vez editado y compilado usando el Make.bat del propio proyecto. Si por el contrario, se ha creado el problema en una ruta distinta de la de por defecto dentro del propio proyecto, se generarán los ficheros relativos al problema en la ruta indicada, así como un archivo *Make.bat* que permite compilar dichos archivos. Esta segunda opción, es importante para crear problemas que no serán incluidos posteriormente en el proyecto, y que simplemente quieren usar la librería HerramientasIA.jar para valerse de las herramientas que esta librería ofrece.

Para obtener ayuda sobre un problema creado y compilado, se podrá pinchar en el botón de ayuda, que mostrará la información relativa al problema que haya sido introducida por el programador.

Finalmente se puede seleccionar el tipo de individuo con el que se hará la simulación, de entre los soportados por el problema, también especificados por el propio programador.

En el panel "**Visualizar Resultados**" se puede elegir la manera en que los resultados del simulador serán presentados al usuario. Si se elige una presentación por "*terminal*", los resultados se irán mostrando por la consola del sistema operativo, si la opción elegida es por "fichero", los resultado se grabarán en un fichero con el nombre que se indique; y por último si los resultados se quieren mostrar en la interfaz gráfica se podrá elegir "Grafica"; dentro de esta última opción, se podrá además mostrar en una gráfica la evolución de tanto el mejor individuo como el individuo medio de cada generación.

Además en los tres modos de salida de resultados se podrá activar/desactivar que muestre en cada generación todos los individuos que la componen, pinchando en el recuadro "*Mostrar población cada generación*". También se podrá elegir cada cuantas generaciones se presentan los resultados en pantalla, el valor por defecto es de uno, es decir, en cada nueva generación se muestran los resultados alcanzados hasta el momento.

Por último decir que se puede iniciar la simulación pinchando en el botón "Iniciar" y pararla pinchando en "Abortar". Una vez que se inicia la simulación, los parámetros

que no se pueden modificar en tiempo de ejecución se ensombrecen para que no puedan ser modificados, y se vuelven a activar cuando la simulación acaba o se aborta.

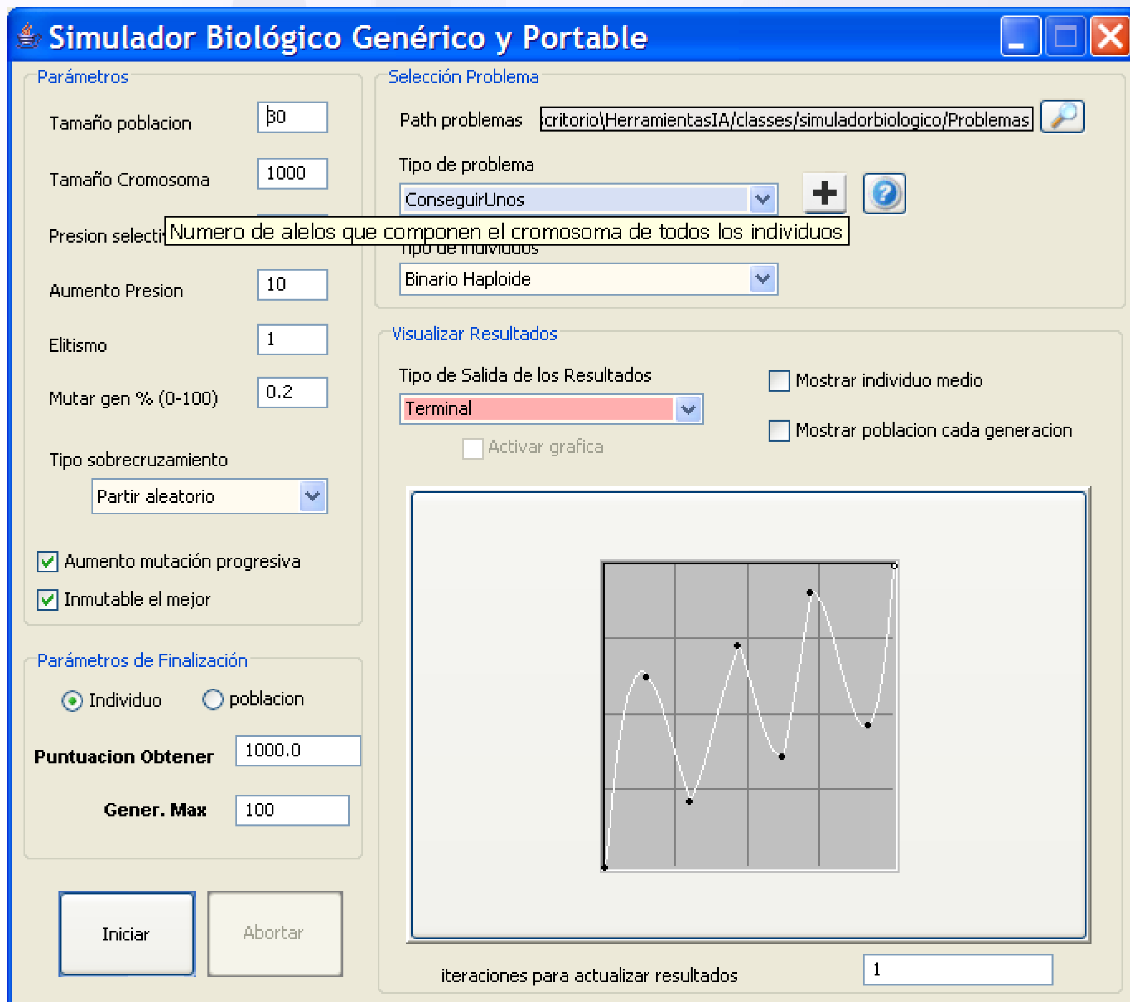


Figura 18. *GUI con información tipo "Tool Tip"*

El aspecto de la herramienta se ha intentado que quedará lo mas intuitivo y simple posible, pero aún así puede resultar en un principio algo confusa, para evitar confusiones, se ha añadido información textual a la mayoría de los elementos que la forman. Para obtener información dicha información basta con mantener el ratón situado unos segundos encima, momento en el cual aparecerá el llamado "tool tip". Incluso se puede obtener información sobre algún punto de la gráfica manteniendo el ratón encima del punto deseado y aparecerá las coordenadas relativas a ese punto de la gráfica.

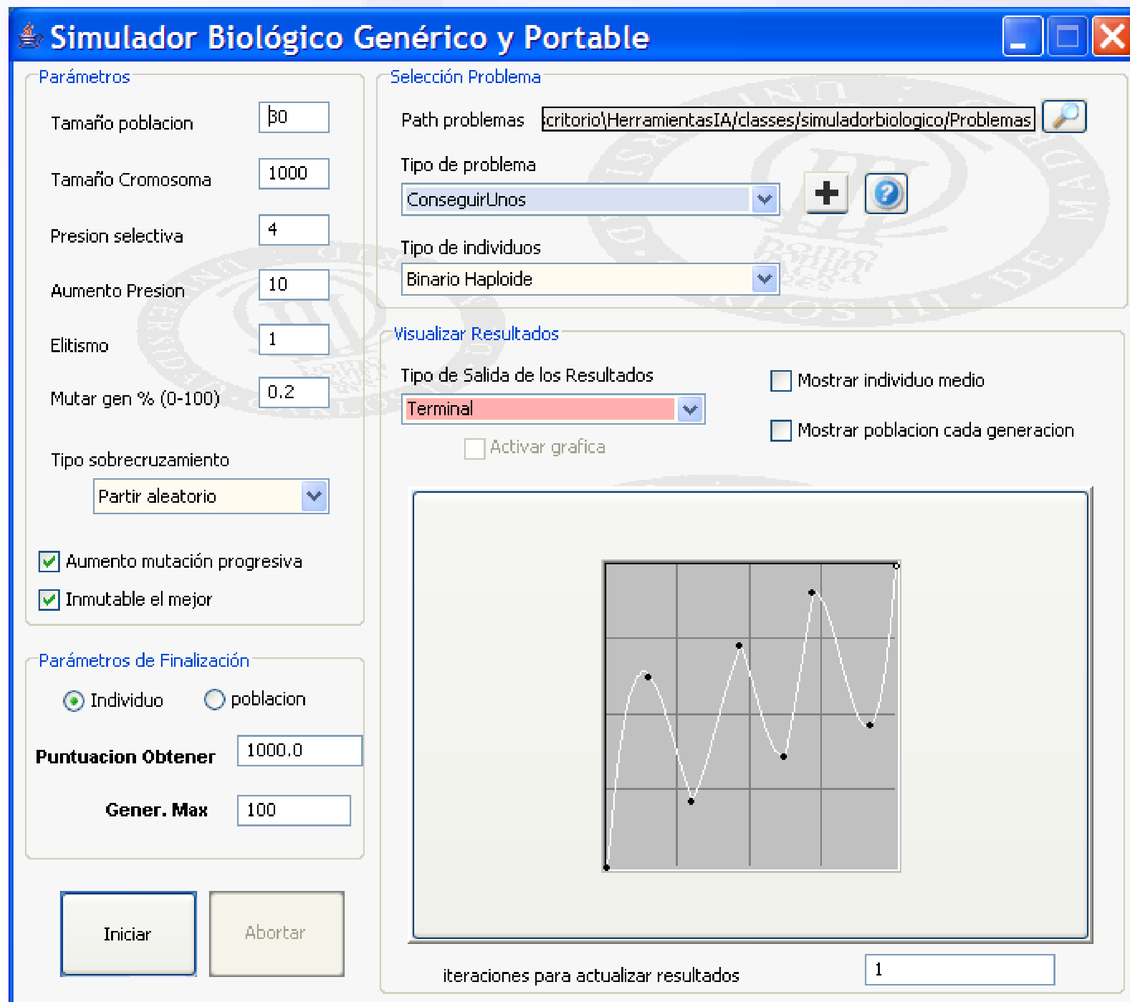


Figura 19. *GUI con nuevo skin (fondo)*

Para conseguir un aspecto más atractivo, se ha añadido además la característica común a todas las herramientas del proyecto de poder cambiar la foto de fondo (el skin o apariencia). Para ello basta con pinchar con el botón derecho sobre cualquier parte de la aplicación y pinchar en cambiar foto de fondo sobre el menú emergente que aparece, posteriormente se selecciona la foto deseada y quedará establecida como fondo. Por defecto en la carpeta "*skin*" del proyecto se adjuntan algunos fondos.

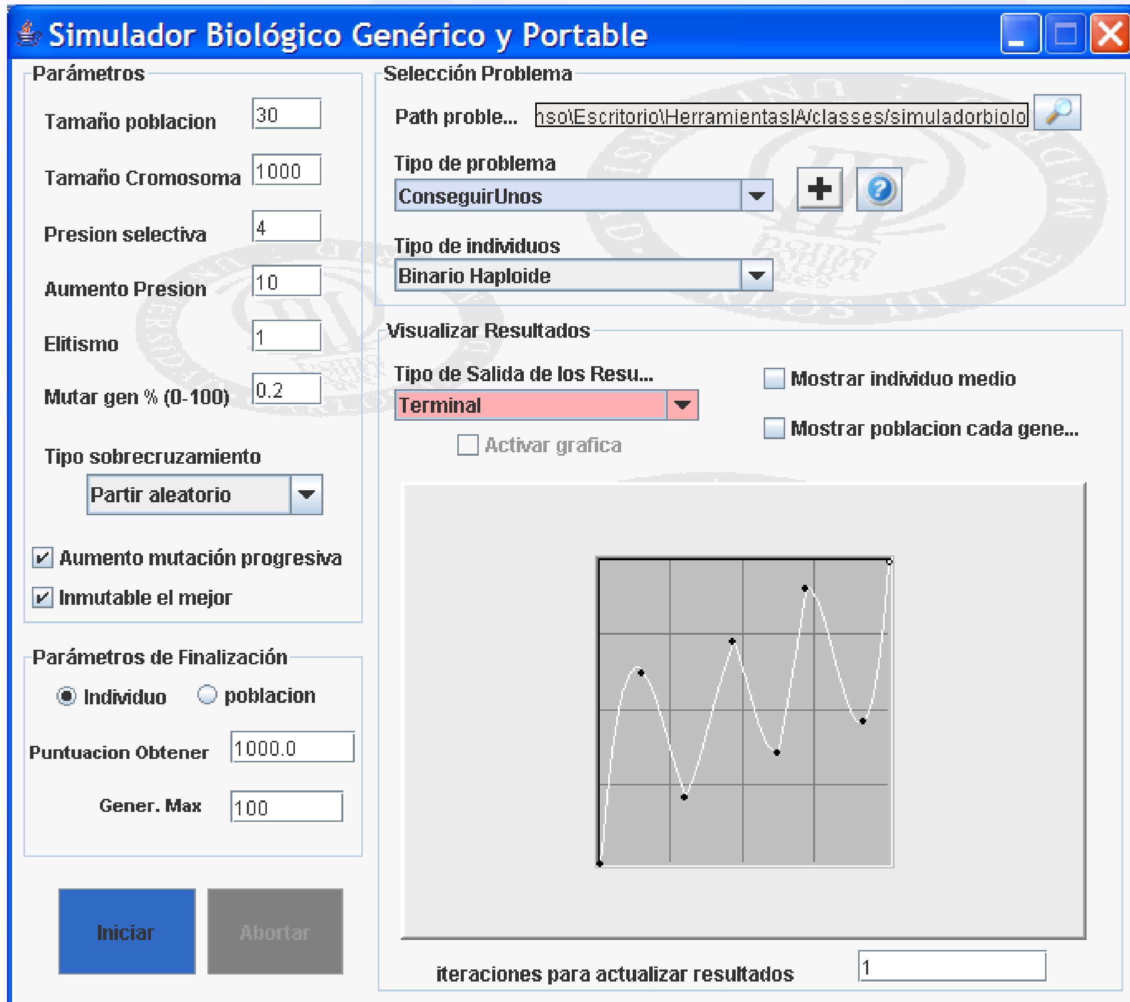


Figura 20. *GUI con nuevo Look & Feel*

También se añadido la posibilidad de cambiar el “look&feel” de la ventana, pudiendo intercambiarse entre varios “Look & Feel”, como por ejemplo: java, metal, windows, borland…. Nuevamente esta acción se puede realizar pinchando en el botón derecho del ratón y seleccionando en “cambiar loock&feel”, que irá alternando entre los disponibles.

3.6.1.2 EJECUTAR UN PROBLEMA DESDE LA INTERFAZ DE USUARIO

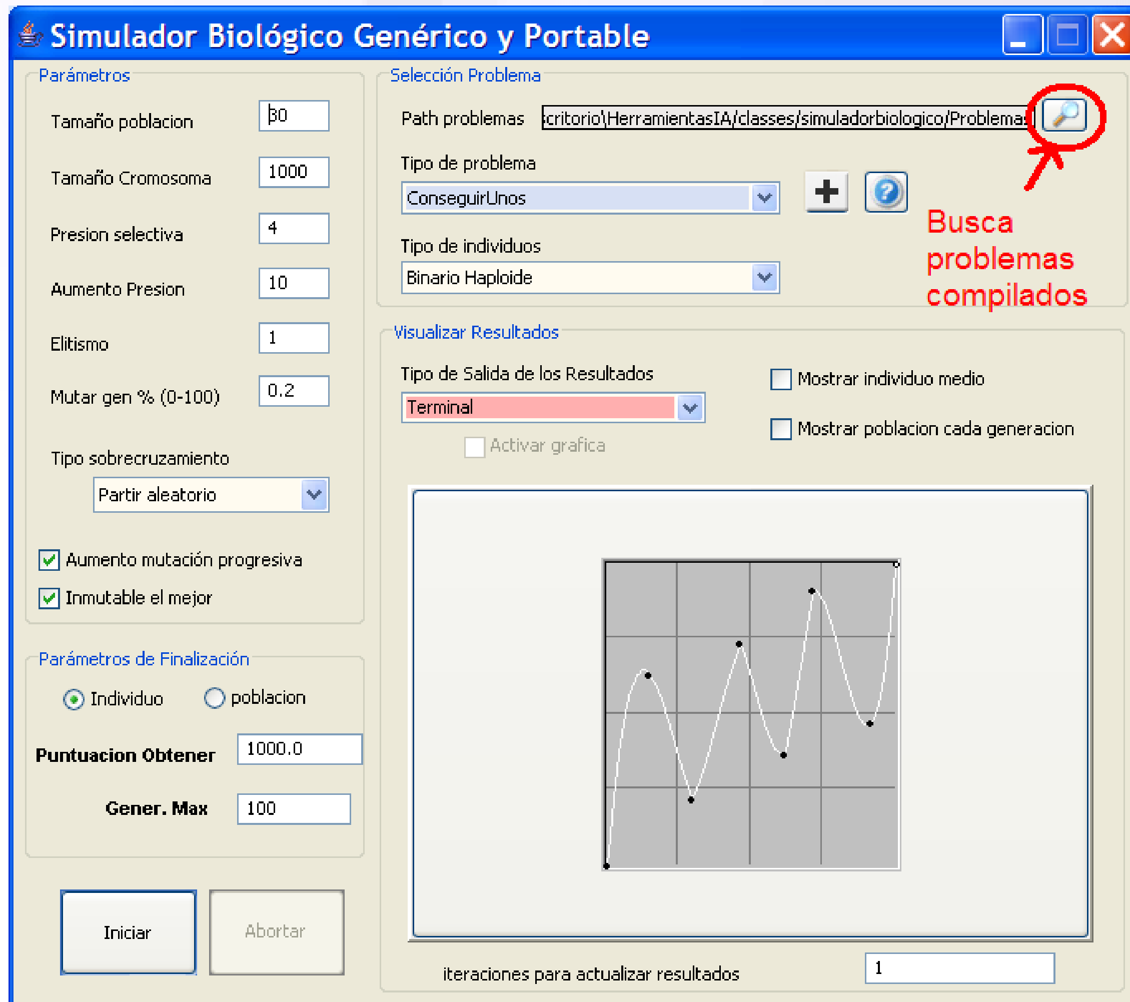


Figura 21. *Buscar problemas en la ruta indicada*

Para buscar los problemas implementados en el propio proyecto o en problemas externos al mismo, basta con indicar la ruta del problema y pinchar en la lupa. De esta manera se cargarán los problemas compatibles con el simulador. Si el problema es externo a la aplicación (se ha creado fuera de la ruta por defecto) se cargará mediante el cargador de clases de Java.

Notar que algunos problemas no podrán ser cargados y así se indicará por terminal, bien porque no están compilados, o no lo están adecuadamente, o simplemente porque no son válidos para ser ejecutados desde la propia interfaz gráfica, puesto que necesiten argumentos que no puedan ser pedidos desde la misma (como objetos), un ejemplo de un problema que no puede ser cargado desde la interfaz gráfica es el de ajustar los pesos de una red neuronal, ya que necesita recibir en el constructor del propio

problema la red de neuronas con la que va a trabajar y evaluar el individuo. Es decir, sólo podrán ser cargados en la interfaz gráfica aquellos problemas en cuyo constructor no se reciba ningún argumento (si pudiéndolos pedir posteriormente dentro del propio constructor). Los problemas que necesitan recibir argumentos en el constructor serán adecuados para ser lanzados sin la interfaz gráfica, como se puede ver en el punto “3.5.2 Ejecutando el simulador sin la interfaz gráfica”.

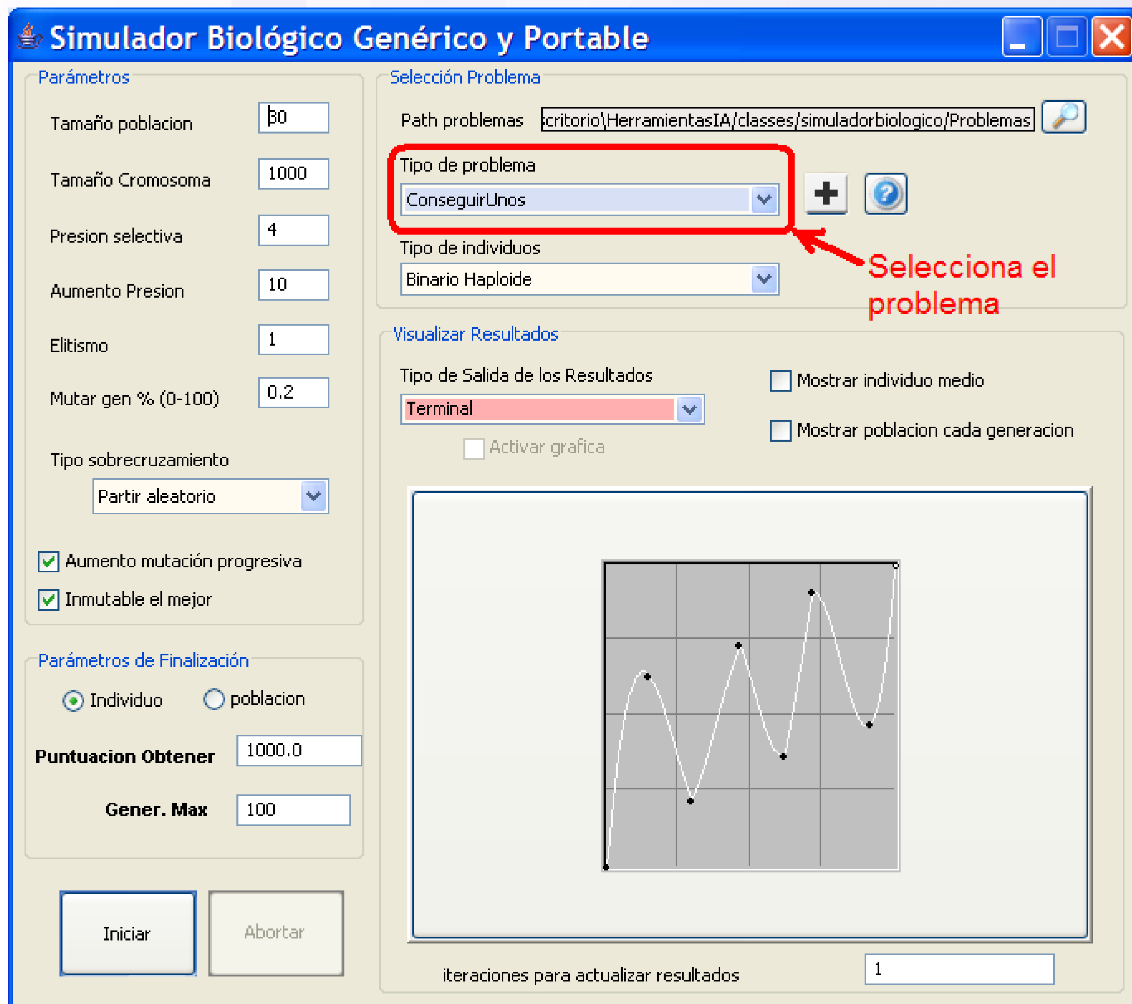


Figura 22. *Selecciona el problema que se quiere solucionar*

De entre los problemas cargados en la ruta indicada se podrá seleccionar aquel que queramos resolver. Una vez seleccionado se establecerán los valores por defecto del problema (si se ha indicado por el programador y sino los valores por defecto de cualquier problema).

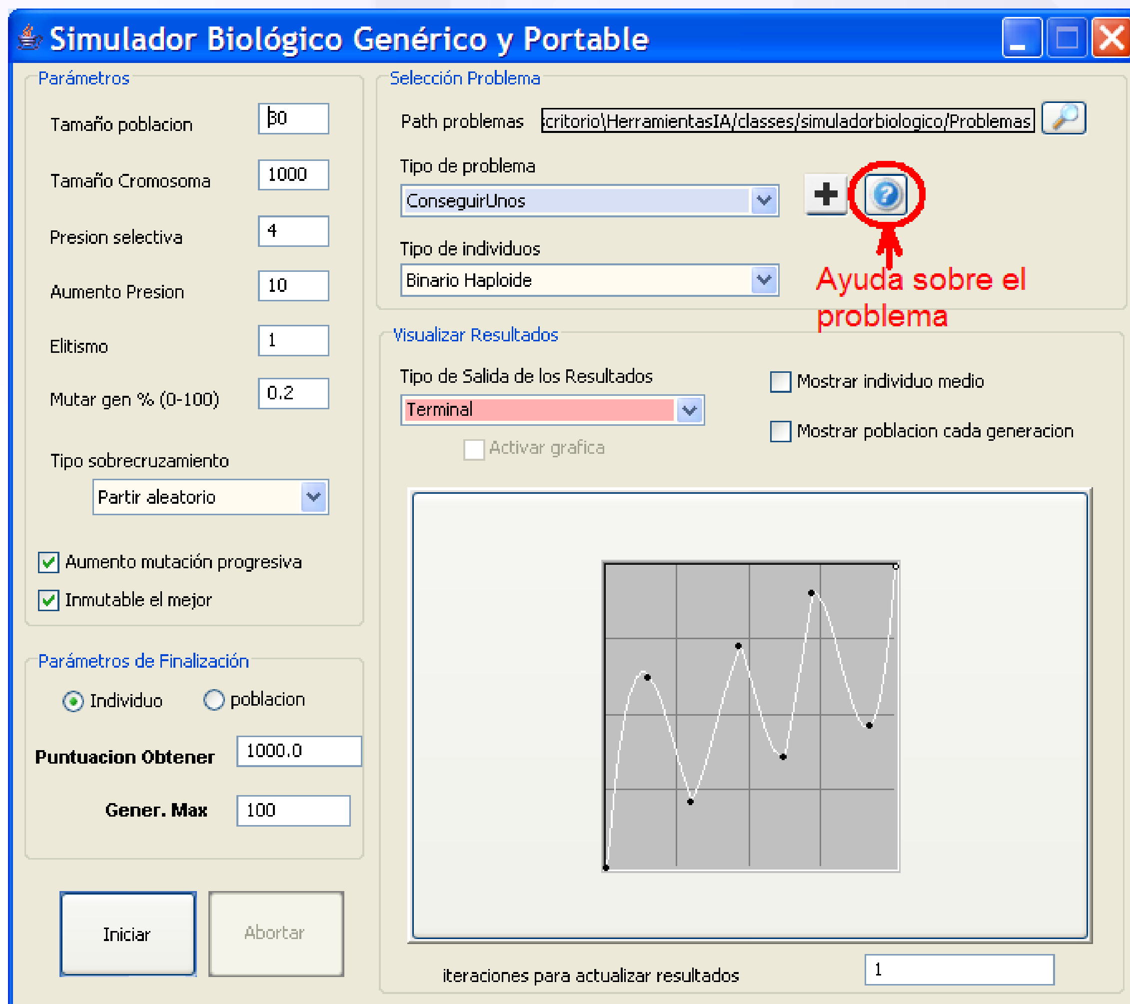


Figura 23. *Obtener ayuda sobre el problema seleccionado*

Si se pincha en el botón de ayuda, se podrá obtener ayuda sobre el propio problema y sobre los valores por defecto establecidos para ese problema. Esta ayuda la introduce el programador del problema, mostrándose la de por defecto sino se ha introducido ninguna nueva.

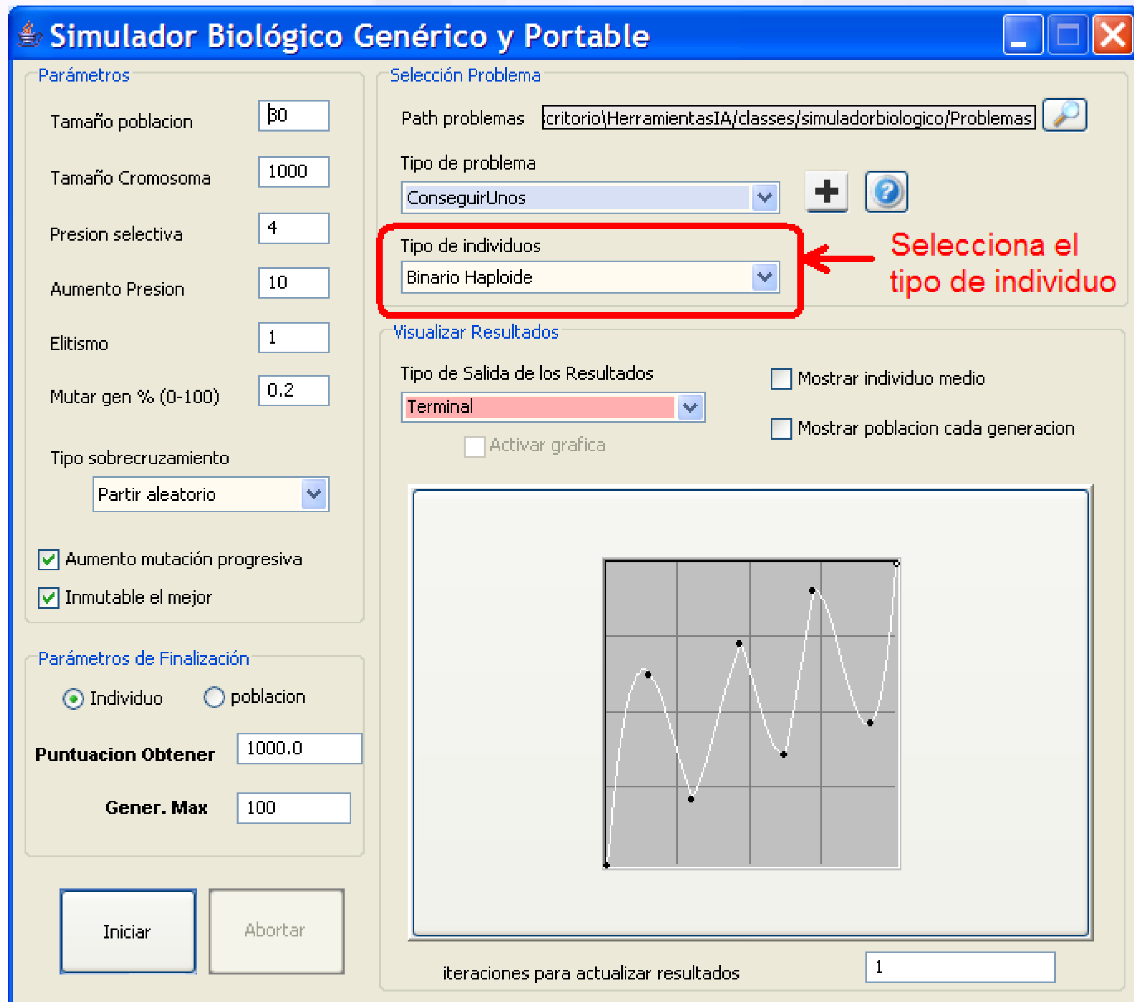


Figura 24. *Seleccionar el tipo de individuo*

Se podrá seleccionar el individuo con el que se va a solucionar el problema, de entre los soportados por dicho problema. De tal manera que se puede probar en varias simulaciones con distintos tipos de individuos y codificaciones: binarios, enteros, haploides, diploides …

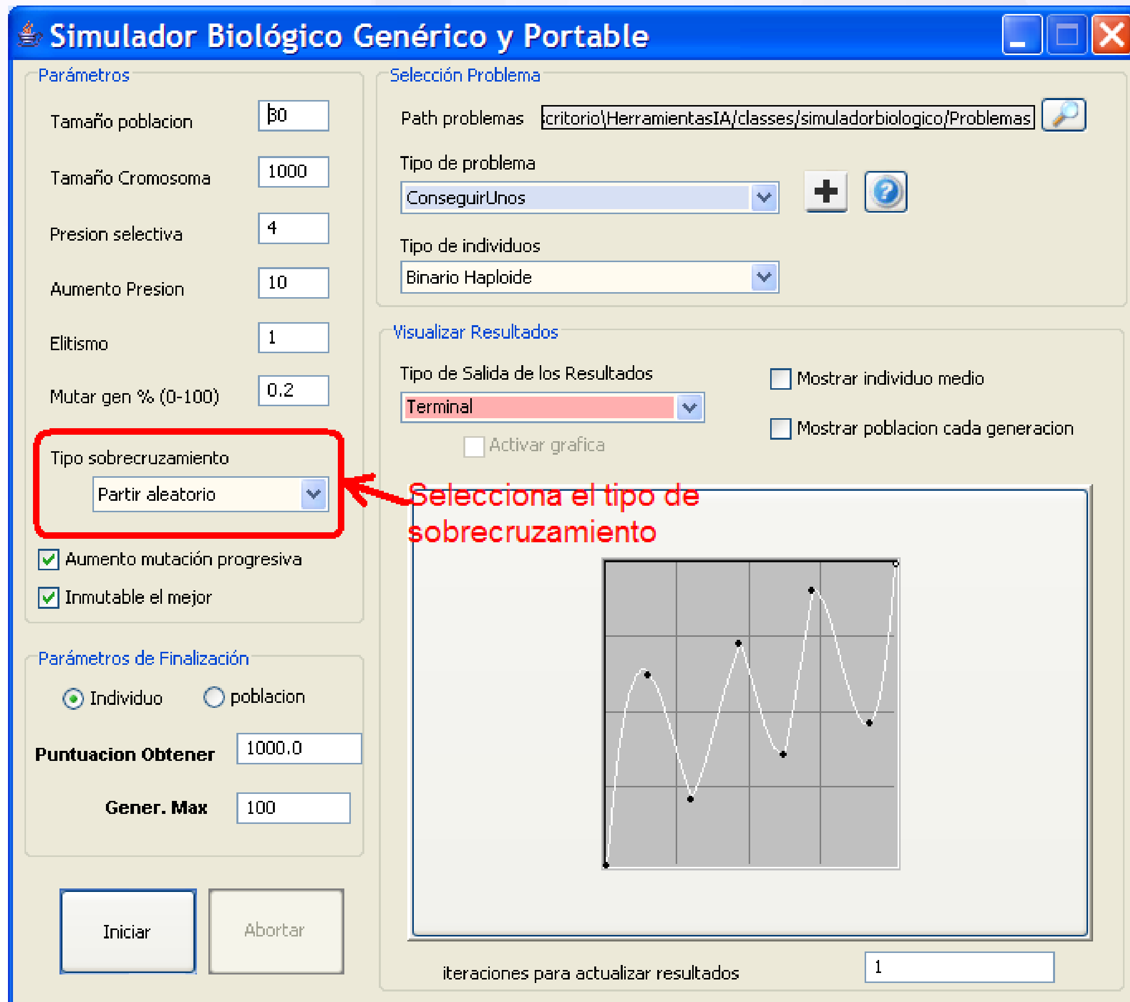


Figura 25. *Seleccionar el tipo de sobrecruzamiento*

Se puede seleccionar la manera en que se va a realizar el intercambio genético entre los dos progenitores para dar lugar a un nuevo individuo resultante de la recombinación genética de sus dos progenitores. El sobrecruzamiento puede ser cambiado incluso durante la propia simulación, siendo esto un factor muy interesante a la hora de conseguir que el algoritmo salga de un estado de estancamiento.

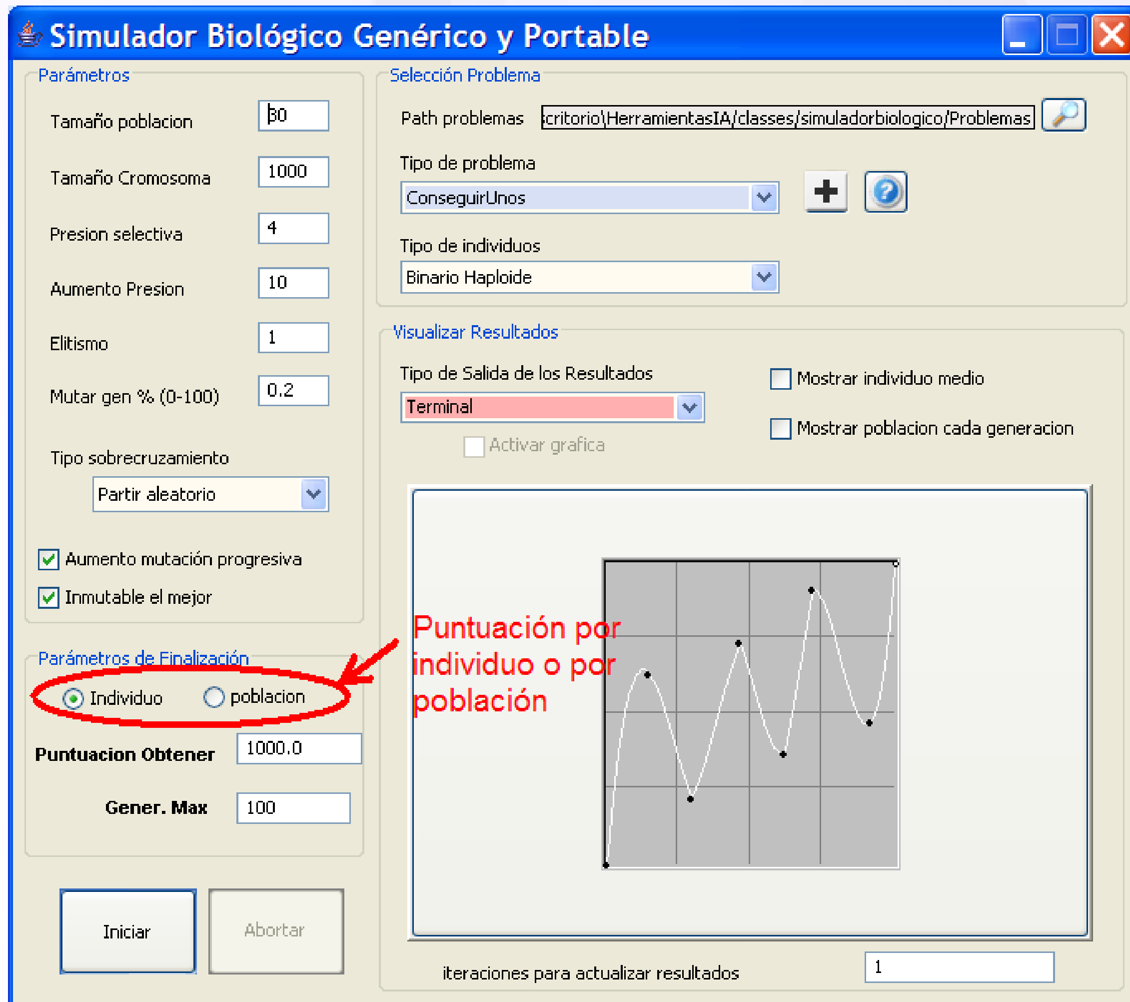


Figura 26. *Tipo de solución*

La solución que se quiere obtener puede ser un individuo en concreto o toda la población de una determinada generación, para ello, se puede indicar en que tipo de problema nos encontramos. Por defecto la puntuación a obtener será el de la población. Se deberá indicar la puntuación que se quiere obtener en caso de solución óptima, es decir, la máxima valoración que puede obtener un individuo; al igual que también se debe indicar el número máximo de generación a explorar.

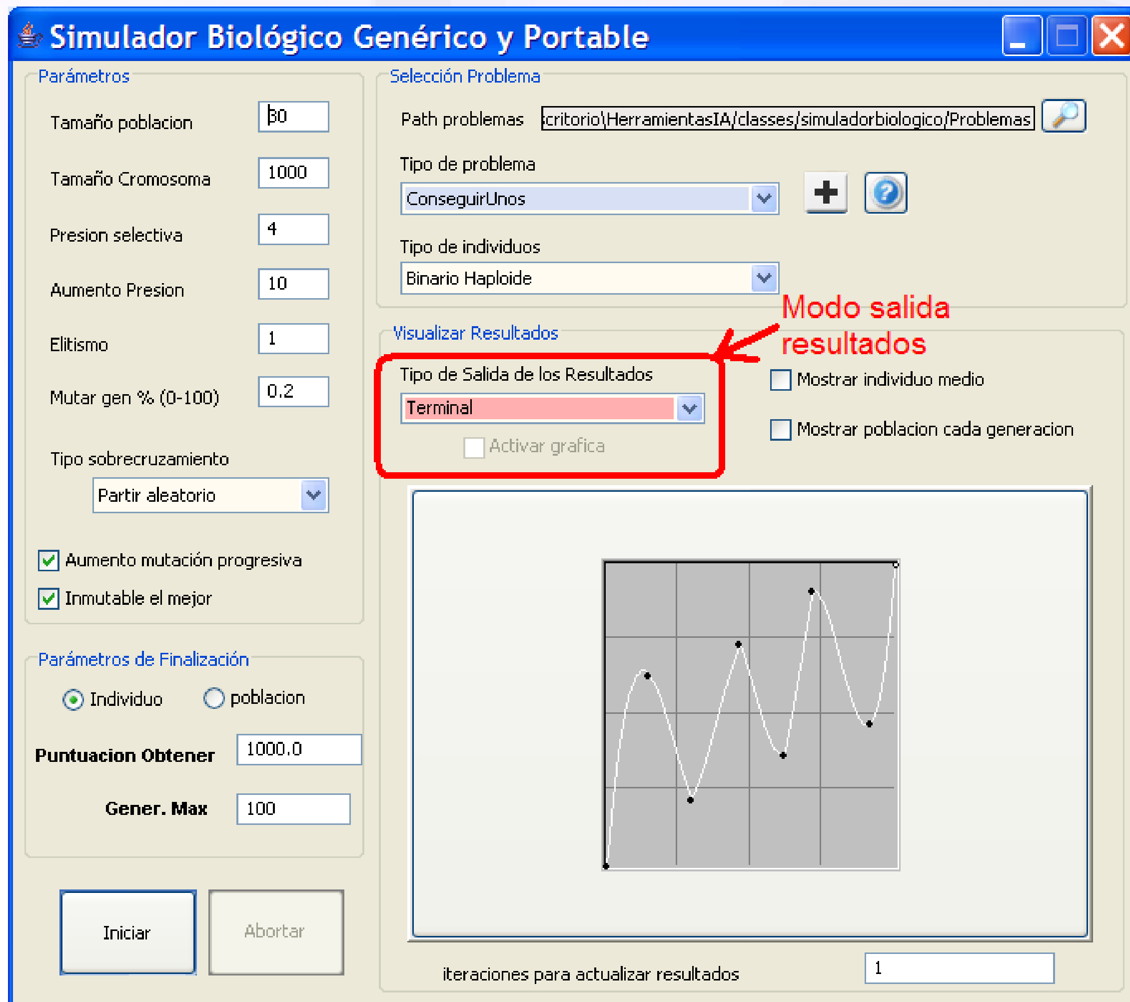


Figura 27. *Visualizar los resultados*

Los resultados y la evolución del algoritmo se pueden mostrar de distintas maneras. La manera usual de mostrar los resultados será de manera gráfica (es decir en la interfaz gráfica), pudiéndose escoger la opción de generar la gráfica que relacione la puntuación obtenida en cada generación. También podrán mostrarse la evolución y los resultados por consola o por fichero. En los tres casos se puede ir mostrando además del individuo mejor, el individuo medio, y todos los individuos que conforman la generación (para este último caso suele ser mas conveniente elegir la opción de visualizar los resultados en fichero). Estas opciones de visualización pueden cambiarse incluso durante la propia simulación. Por último mencionar que se puede decidir cada cuantas iteraciones se mostrará la evolución del algoritmo, por defecto es en todas.

Hacer notar, que cuanta mas salida de resultados se realicen mayor tiempo se invertirá en la simulación.

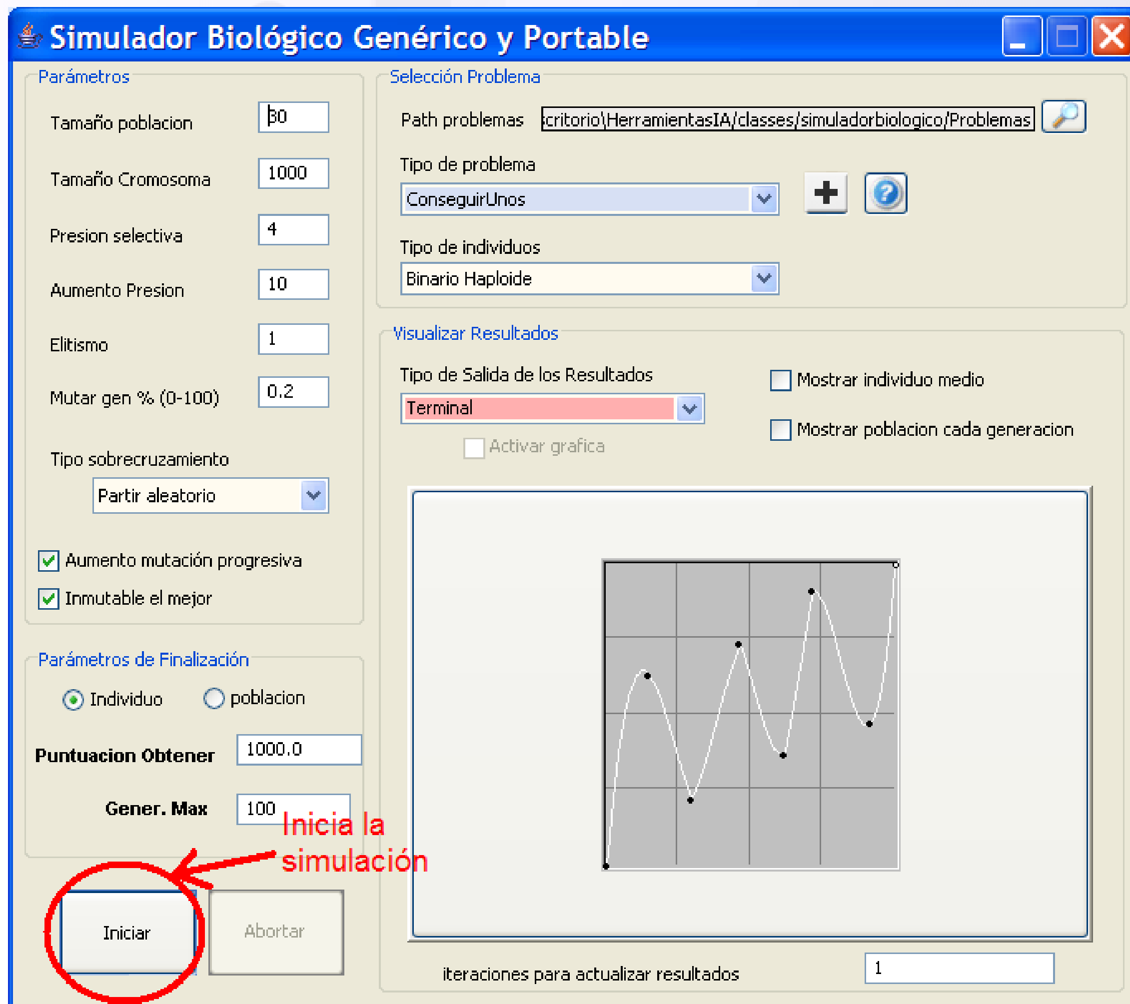


Figura 28. *Iniciar/parar la simulación*

Con los botones “Iniciar” y “Abortar” se podrá, como su nombre dicen, iniciar y abortar la simulación respectivamente. En el caso de abortar la simulación se mostrará el mejor individuo obtenido hasta el instante actual y se parará de ejecutar el algoritmo genético, pudiéndose volver a iniciar desde el principio con los mismos o con otros parámetros distintos a los de la anterior simulación.

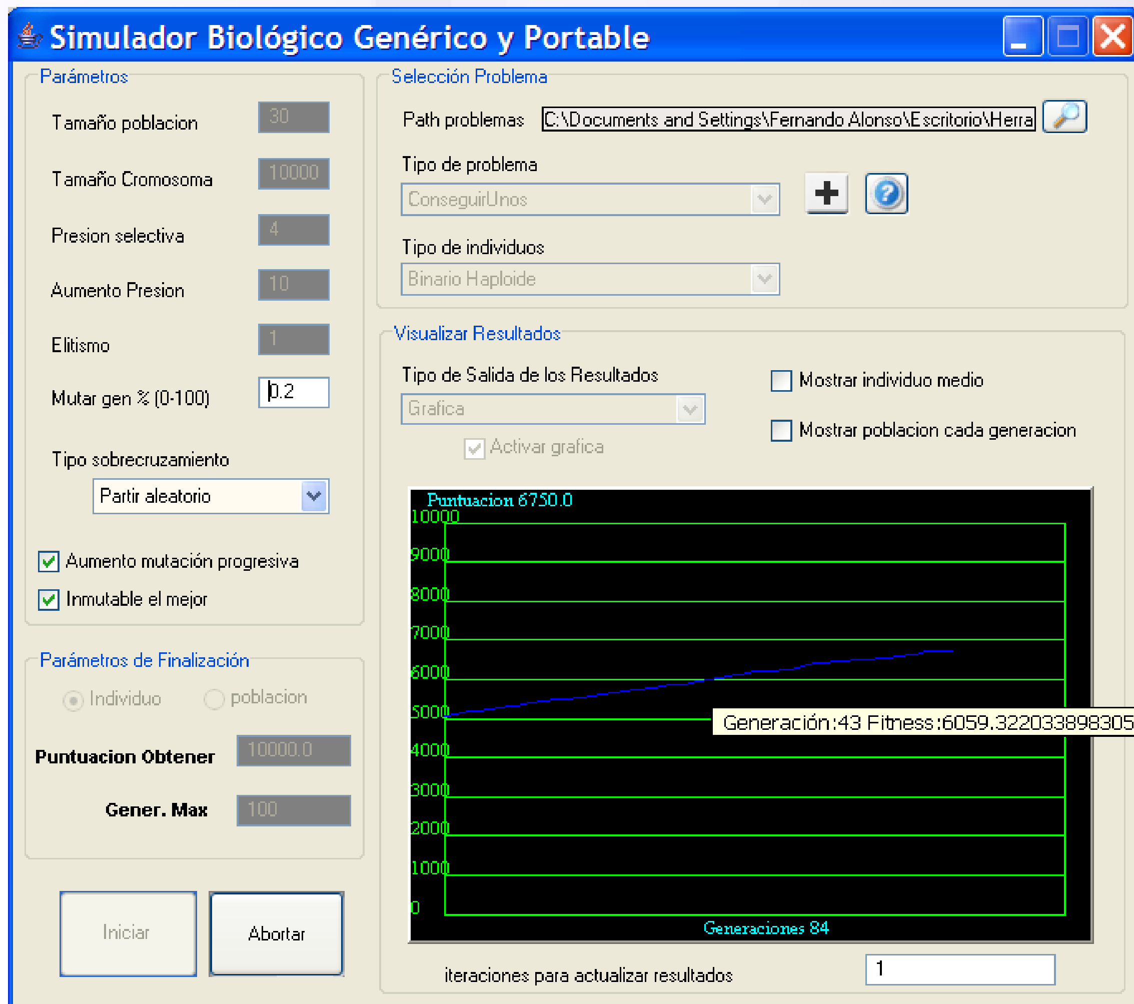


Figura 29. *Ejecutando simulación*

Una vez iniciada la simulación se desactivan (ensombrecen) los parámetros que no pueden ser modificados en tiempo de ejecución, los que no se desactivan sí pueden ser modificados durante la simulación. También se puede ver como se va creando dinámicamente la gráfica que relaciona las generaciones con la puntuación obtenida. Si se mantiene el ratón encima de cualquier punto de la gráfica se pueden ver sus coordenadas exactas.

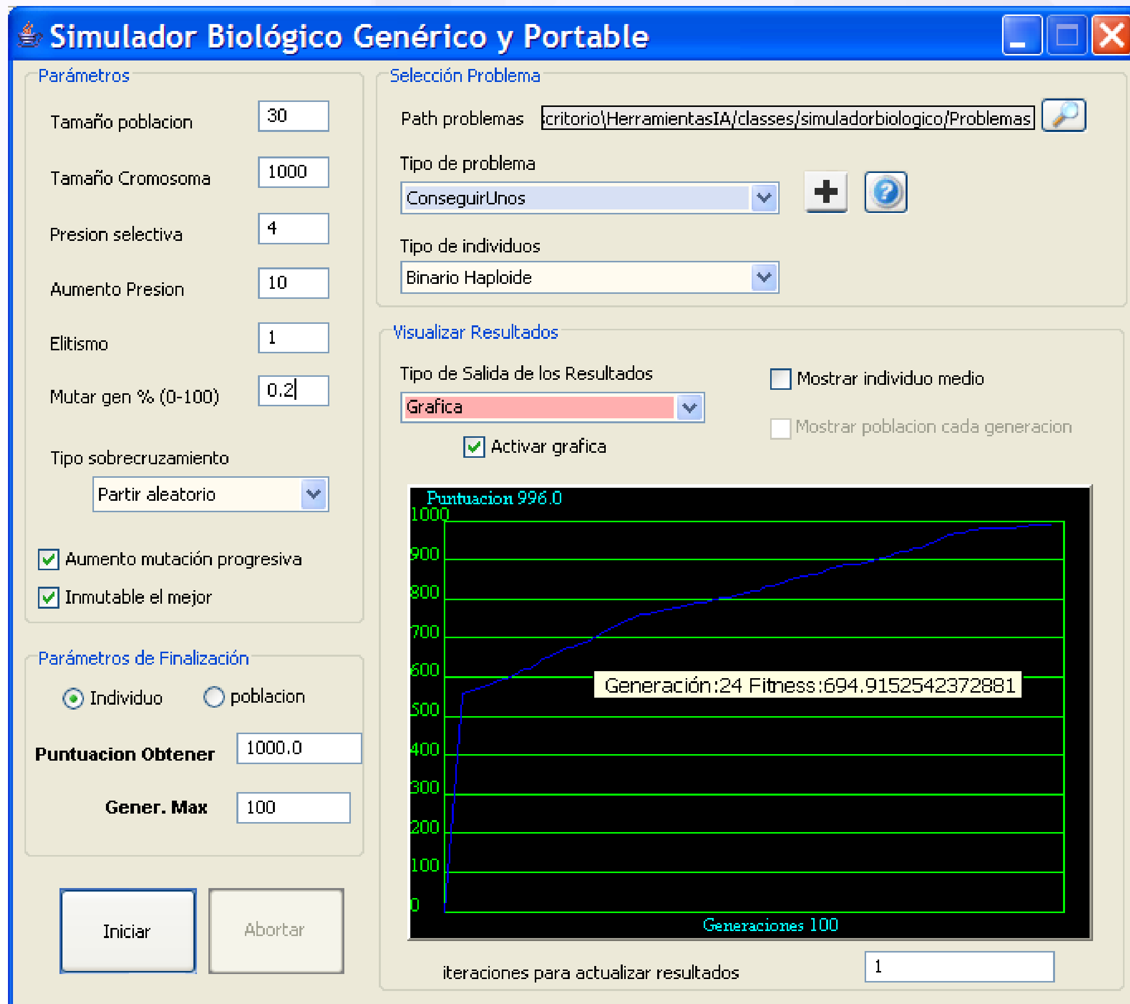


Figura 30. *Fin de la simulación*

Al acabar la simulación se puede seguir inspeccionando la gráfica, y se vuelven a activar todos los parámetros para que puedan ser modificados y se pueda volver a iniciar la misma simulación o cualquier otra.

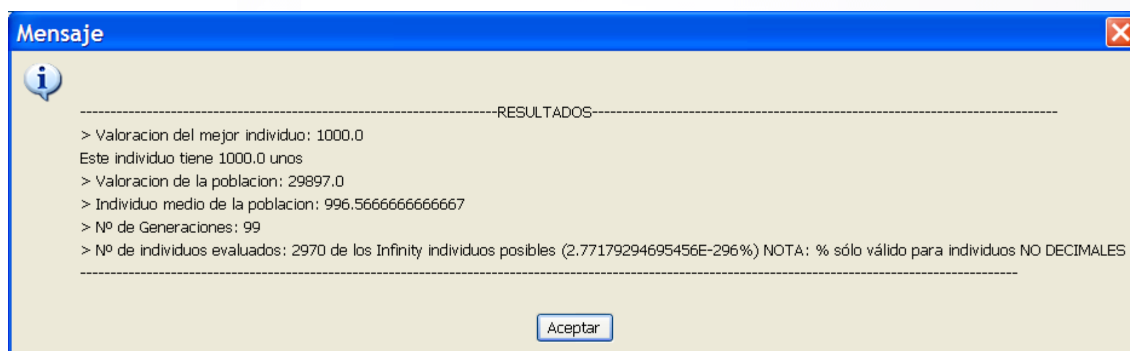


Figura 31. *Resultados de la simulación*

Al finalizar la simulación se muestran los resultados obtenidos.

3.6.1.3 CREAR UN NUEVO PROBLEMA DESDE LA INTERFAZ DE USUARIO

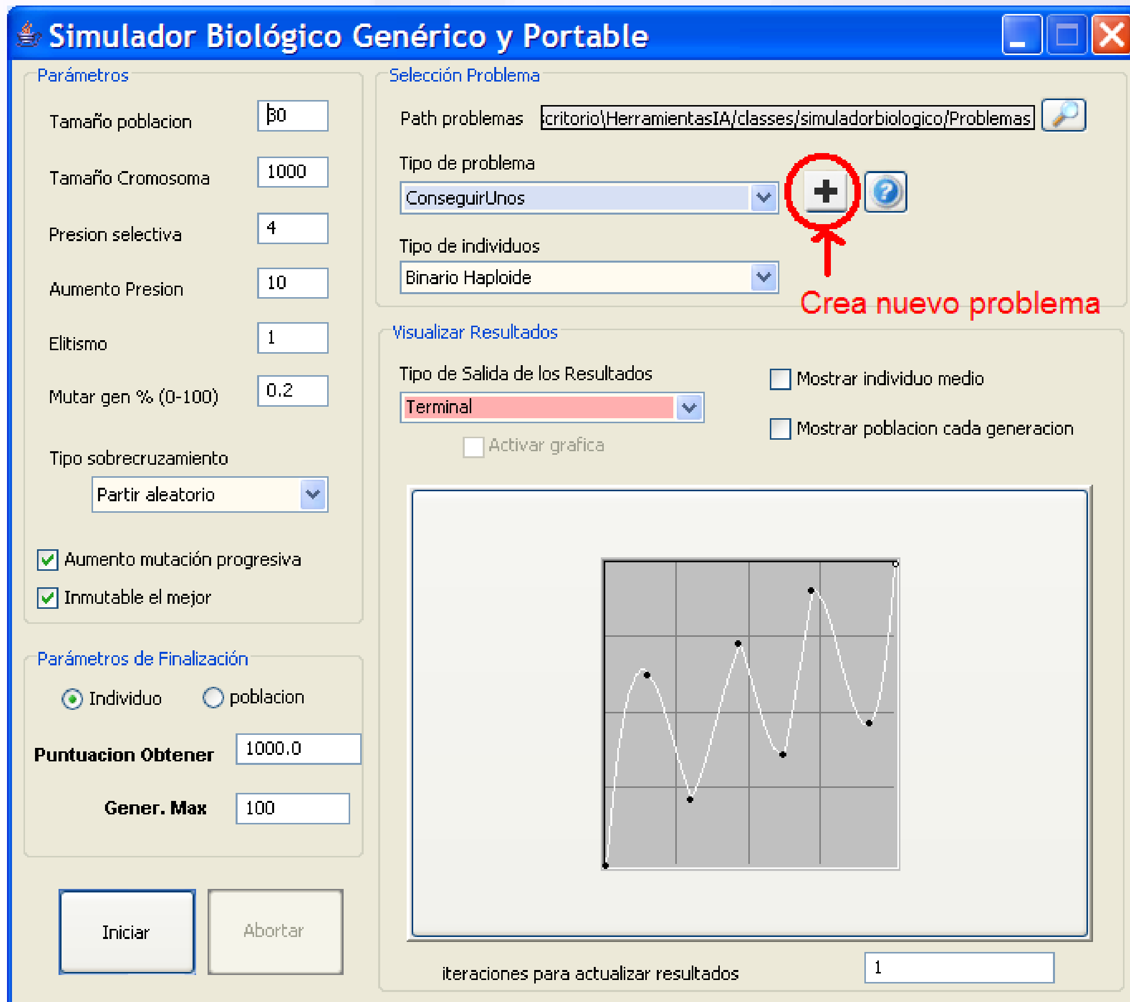


Figura 32. *Crear nuevo problema*

Si se desea crear un problema que estará dentro del propio proyecto (cada vez que se haga un make será también compilado) se debe dejar la ruta especificada en "*Path problemas*" sin modificar. Vamos a analizar ambos casos:

- CREANDO EL PROBLEMA DENTRO DEL PROPIO PROYECTO

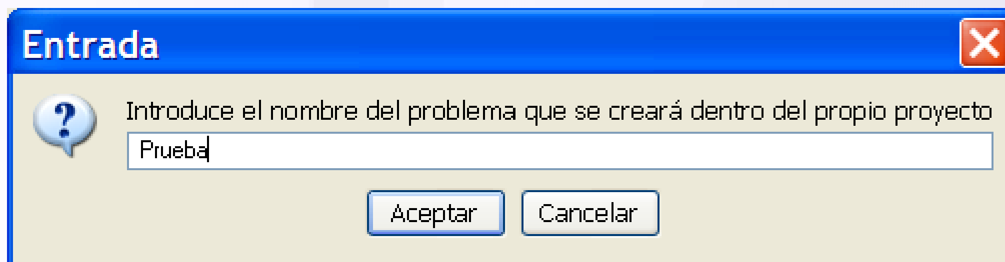


Figura 33. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a las clases que se crearán y al paquete donde se introducirán ambas clases.

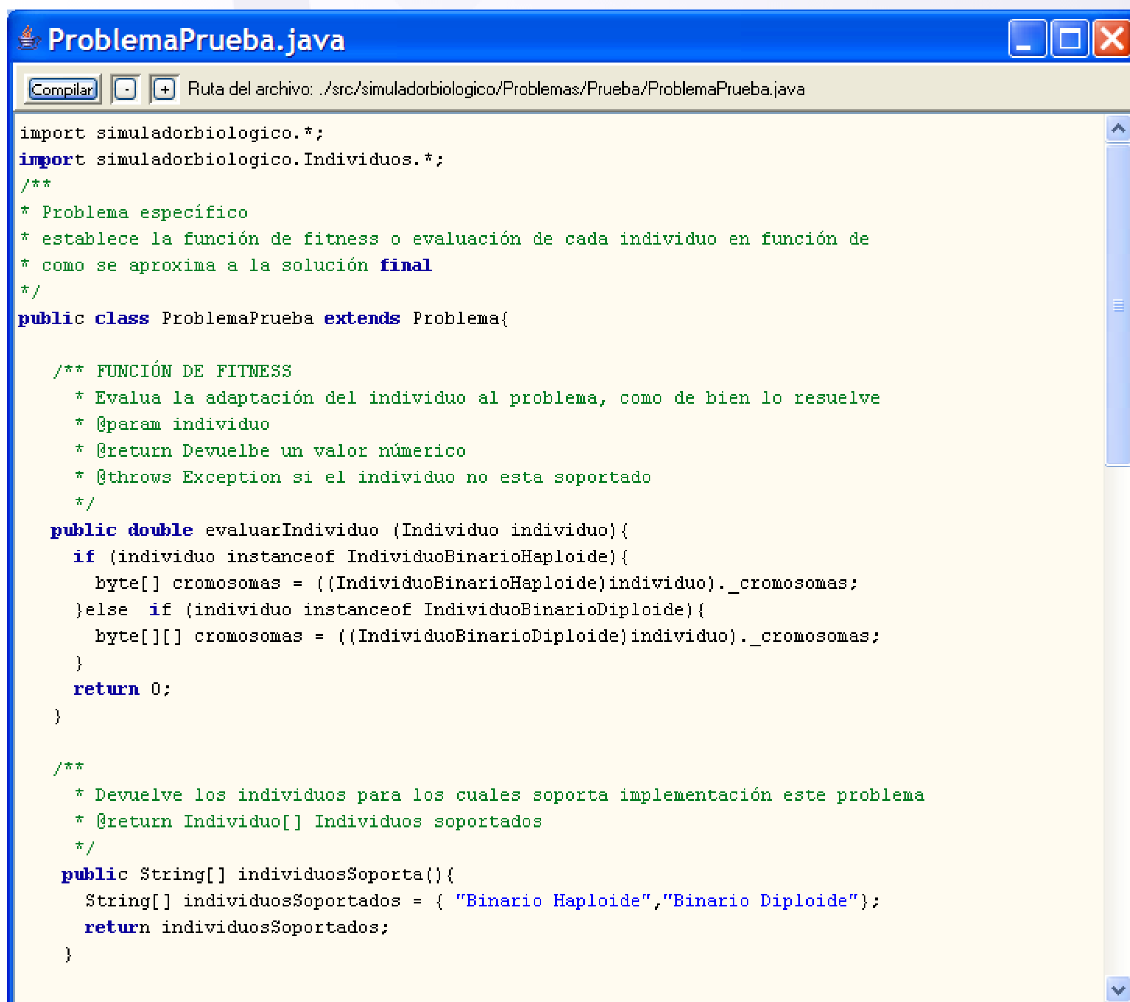


Figura 34. *Implementar el problema*

A continuación se muestra un editor desde el que se puede editar y compilar el problema. De primeras se muestra un esqueleto por defecto de cómo debe ser el problema, este se debe modificar para adaptarlo a la codificación del problema que queramos hacer y finalmente compilarlo.

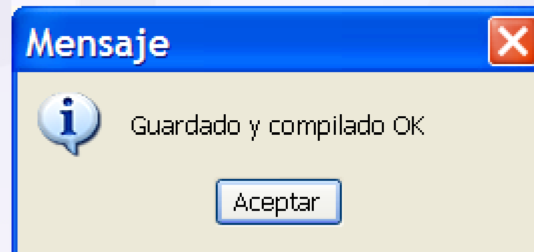


Figura 35. *Problema guardado y compilado*

Si el problema compila correctamente, nos pondrá guardado y compilado. El archivo .java se guarda dentro de la carpeta indicada en el path por defecto que es src/simuladorbiologico/problemas/nombreProblema y el .class se guarda en la misma ruta pero de la carpeta classes. Posteriormente puede ser editado nuevamente y compilado compilando todo el proyecto usando el *make.bat* que se suministra con el proyecto.

PoblacionPrueba.java

Compilar [-] [+] Ruta del archivo: ./src/simuladorbiologico/Problemas/Prueba/PoblacionPrueba.java

```
import simuladorbiologico.*;

/** Poblacion basada en reproduccion de torneos*/
public class PoblacionPrueba  extends Poblacion{

    /**Creamos individuos diploides con codificacion binaria */
    public PoblacionPrueba (Problema problema){
        super(problema);
    }

    /** @return Valor minimo que podra tomar cada gen de cada individuo de la poblacion */
    public double valorMinimoGenIndividuos(){
        return 0;
    }

    /** @return Valor maximo que podra tomar cada gen de cada individuo de la poblacion */
    public double valorMaximoGenIndividuos(){
        return 1;
    }


    // PUEDEN SER SOBRESCRITOS LOS SIGUIENTES MÉTODOS (sino borrense)

    /** Se le da una valoración a cada individuo para tener mas probabilidades de sobrevivir */
    public void aplicarSeleccionNatural(){
       for (int i = 0; i < _individuos.length; i++){
           _individuos[i]._valoracion = _contexto.evaluarIndividuo(_individuos[i]);
       }
    }

   /**
     * La población se reproduce en función a las valoraciones de cada individuo; notar que cada individuo de la nueva
     * poblacion no ha sido evaluado todavia ni ha sufrido todavia procesos de seleccion
     * @param presionSelectiva int  a numero mas grande mas presion se hace, cuanto mas pequeño menos presion
     * @param elitismo int Numero de individuos que pasan directamente a la siguiente generacion por ser los mejores
     * @param tipoSobrecruzamiento String la forma en la que se realizará el sobrecruzamiento entre los individuos
     * @param mejorInmutable boolean Determina si el mejor individuo está expuesto a mutación o no
```

Figura 36. *Implementar la población*

A continuación se muestra un editor desde el que se puede editar y compilar la población para el problema. De primeras se muestra un esqueleto por defecto de cómo debe ser la codificación, este puede ser modificado para indicar el valor máx. y min. que puede tomar cada gen, así como para realizar un tipo de selección natural y/o reproducción distintas de las usadas por defecto (selección darviniana y reproducción por torneos).

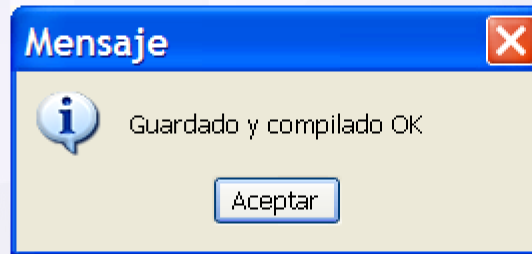


Figura 37. *Crear nuevo problema*

Si esta bien implementado y se pincha en el botón de compilar, mostrará el mensaje de la figura anterior. Nuevamente decir que puede ser vuelto a editar y recompilar desde cualquier editor y compilando usando el make del proyecto.

Finalmente se puede seleccionar el problema que se acaba de crear para poder ser ejecutado, tal y como se dijo en el apartado "ejecutar un problema desde la interfaz de usuario", sin tener que cerrar y volver a lanzar la aplicación.

- CREANDO EL PROBLEMA FUERA DEL PROYECTO

Cuando se cambia el path por defecto donde se crearán los problemas, se crean en una ruta distinta de la asignada en el proyecto, por lo que se generarán en la ruta indicada los ficheros esqueleto del problema para que puedan ser editados y compilados con posterioridad.

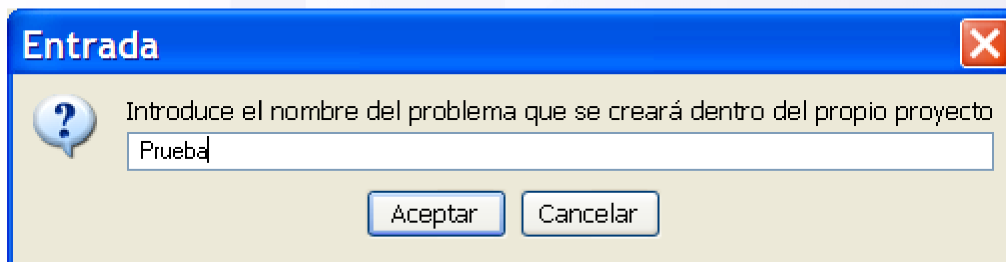


Figura 38. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a las clases que se crearán y al paquete donde se introducirán ambas clases.

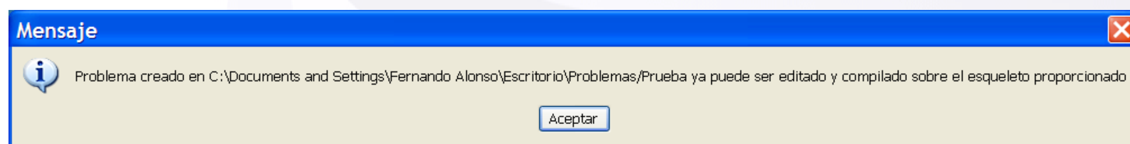


Figura 39. *Mensaje informativo de que se han guardado los ficheros*

Si la ruta indicada es correcta se creara el problema en la ruta indicada y se mostrará el mensaje de la figura anterior. Se habrán guardado los archivos java en la ruta indicada pero no habrán sido compilados. Para compilarlos se deberá ejecutar el archivo “*Make.bat*” que se genera conjuntamente con los archivos, y para los que únicamente usa para compilarlos es la librería “*HerramientasIA.jar*”.

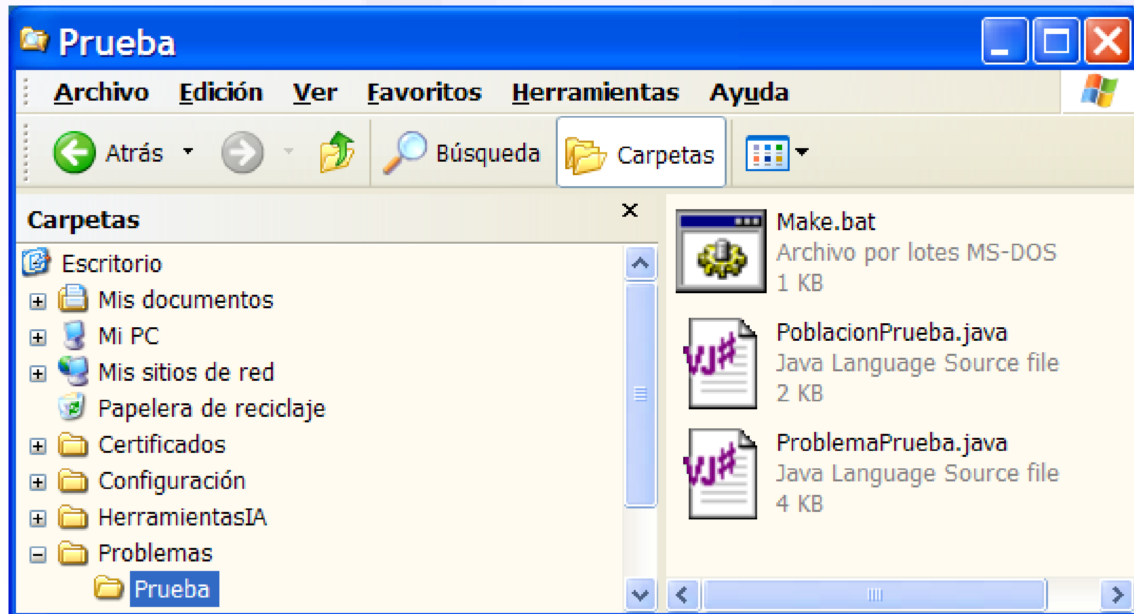


Figura 40. *Archivos fuentes con el esqueleto del problema creado*

Una vez compilados ejecutando Make.bat, puede ser el problema cargado y ejecutado como se comenta en la sección "ejecutar un problema desde la interfaz de usuario".

3.6.2 Ejecutando el simulador sin la interfaz gráfica

Un problema creado valiéndonos de la interfaz gráfica de usuario, como se dice en la sección "Crear un nuevo problema desde la interfaz de usuario", o sin valernos de ella, puede ser ejecutado sin necesidad de ejecutar el main de la aplicación y por tanto de la interfaz gráfica. Para ello se deberá desde el programa que se quiera usar escribir las siguientes líneas:

```
//importamos el simulador biológico y los individuos
import simuladorbiologico.*;
import simuladorbiologico.Individuos.*;

//instanciamos el problema y su población
Problema problemaPrueba = new ProblemaPrueba();
Población poblacionPrueba = new PoblacionPrueba(problemaPrueba);

//lanzamos el algoritmo genético
Individuo individuoSolucion =
                    EjecutaAG.lanzarAlgoritmoAG(poblacionPrueba,"Binario Haploide");

//obtenemos la solución
byte[] solución = individuoSolucion._cromosomas;
```

Figura 41. *Ejecuta problema con los parámetros por defecto*

Lanza el algoritmo genético de manera síncrona y con los valores por defecto dados en la implementación del problema (sino se han sobrescrito los métodos que dan los valores por defecto se quedan los que tienen todos los problemas). Notar que en este caso, se pueden ejecutar problemas cuyo constructor reciba argumentos (como por ejemplo objetos).

```
import simuladorbiologico.*;
import simuladorbiologico.Individuos.*;

//instanciamos la población y el problema
Problema problemaPrueba= new ProblmeaPrueba(parametrosProblema);
Poblacion poblacionPrueba = new PoblacionPrueba(problemaPrueba);

//parámetros de la simulación
int longitudCromosoma=10;
double puntuacion=100;
int elitismo=1;
int numeroIndividuos=60;
int presionSelectiva=3;
double probabilidadMutarGen=0.2;
int pasosParaAumentarLaPresion=(((int)iteraccionesMax*15)/numeroIndividuos);
long iteraccionesMax = 1000;
String tipoSobrecruzamiento="Partir aleatorio";
int iterMostrarResultados=1;
boolean mejorInmutalbe = false;
Object[] parametrosProblema = {argumentoNecesarios};


//lanzamos el algoritmo genético
Individuo individuoSolucion =
EjecutaAG.lanzarAlgoritmoAG(poblacionPrueba,
                            "Decimal Haploide”,
                             puntuacion,
                             numeroIndividuos,
                             longitudCromosoma,
                             presionSelectiva,
                             probabilidadMutarGen,
                             pasosParaAumentarLaPresion,
                            elitismo,
                            iteraccionesMax,
                            tipoSobrecruzamiento,
                            iterMostrarResultados,
                            false,
                            parametrosProblema);

//obtenemos la solucion
double[] solucion = ((IndividuoDecimalHaploide) individuoSolucion)._cromosomas;
```

Figura 42. *Ejecuta problema sin usar los valores por defecto*

Esta segunda manera de lanzar el algoritmo es lanzándolo con los valores concretos que queramos darle, sin tener en cuanta los valores por defecto codificados. Un ejemplo de esta llamada, se hace desde la herramienta Red de Neuronas Artificiales

para entrenar la red mediante algoritmos genéticos, ver la sección “Particularidades de la implementación”.

3. Simulador de Enjambres de Partículas

3.1 Introducción

Este tipo de técnica se engloba dentro de lo que ha venido en denominarse "Swarm intelligence" que son un conjunto de técnicas de inteligencia artificial basadas en el estudio del comportamiento colectivo en sistemas descentralizados y autoorganizativos.

Un enjambre está típicamente formado por una población de agentes simples (partículas) que interactúan localmente con otros y con su entorno. Aunque normalmente no hay una estructura de control centralizado dictando como los agentes individualmente deberían comportarse. Interactuar localmente entre tales agentes a menudo, provoca que surga un comportamiento emergente global o colectivo. Ejemplos de este tipo de sistemas pueden ser encontrados fácilmente en la naturaleza, como colonias de hormigas, bandadas de pájaros, etc. Inspirándose en este tipo de fenómenos, se han creado algoritmos dentro del campo de la inteligencia artificial como son el "Optimizador por Enjambres de Partículas" o el "Optimizador por Colonias de Hormigas".

Particle Swarm Optimization o PSO es un algoritmo globlal para problemas difíciles en el que la mejor solución puede ser representado por un punto en un espacio n-dimensional (codificación del problema como partícula). Las partículas son colocadas en el espacio con una velocidad inicial. Las partículas se mueven en el espacio de soluciones y son evaluadas de acuerdo a la función de fitness en la que se codifica el problema en cada iteración. En un cierto plazo, las partículas son aceleradas hacia esas otras partículas mejor valoradas, hasta que alguna partícula alcanza la valoración deseada por el criterio de parada.

Cada partícula tiene las siguientes características:

- Tiene una posición y una velocidad que servirá para calcular la siguiente posición.

- Conoce su posición, que es evaluada mediante la función de fitness.

- Conoce la valoración de sus vecinos (si no se consideran vecinos las valoraciones de todas las partículas del enjambre) y es capaz de saber cuál es la mejor.

- Recuerda su mejor posición alcanzada a lo largo de su vida.

La ventaja principal del acercamiento sobre otras estrategias globales de optimización tales como el "Temple Simulado" es que el número grande de miembros que componen el enjambre hace que la técnica sea impresionante resistente al problema de los mínimos locales, sin embargo la codificación de los problemas es mas estricta.

El método de optimización PSO guarda ciertas similitudes con los GA. Ambos métodos estocásticos se inicializan con una población de soluciones potenciales, partículas o agentes en el caso de PSO y cromosomas o individuos en el caso de GA, y realizan la búsqueda de una solución óptima en base a un proceso iterativo y generacional, utilizando una función de evaluación para medir la precisión de cada solución. Así mismo, ninguna de las dos técnicas garantiza el éxito pleno de la optimización, el cual está relacionado directamente con la configuración del algoritmo para el problema a optimizar. A grandes rasgos, se puede afirmar que PSO es a la simulación de la interacción social entre individuos, lo que los algoritmos genéticos son a la simulación de la evolución de las especies.

Los GA utilizan los operadores de selección, cruce y mutación, cada uno de los cuales admite múltiples implementaciones y tiene diferentes parámetros a seleccionar y afinar de acuerdo con la naturaleza del problema electromagnético a optimizar. Sin embargo, el PSO únicamente tiene un operador, la velocidad de partícula. Aunque el PSO no tiene explícitamente definidos operadores de evolución tales como cruce o mutación, en realidad, el ajuste de la posición de las partículas en la dirección de la mejor solución personal, y de la mejor solución de conjunto, tiene un significado conceptualmente similar al de cruce y mutación en los GA.

Otra diferencia existente entre GA y PSO hace referencia a la capacidad de control sobre la convergencia de la población. Las probabilidades de cruce y mutación en los GA sugieren una vía para controlar la convergencia del método y, por tanto, la

diversidad de los cromosomas, mientras que en PSO el peso inercial permite realizar dicho control de forma mucho más sutil y directa. El mecanismo que posee PSO para compartir información entre congéneres es significativamente distinto al de los GA. En los GA todos los cromosomas comparten información entre sí, de forma que la población se mueve como un grupo hacia una región óptima, mientras que en PSO es el mejor agente, el que transmite la información al resto y condiciona el movimiento del enjambre. En este caso, la evolución sólo busca la mejor solución, de tal forma que, por lo general, todas las partículas tienden a converger más rápidamente hacia la solución óptima.

En comparación con los GA, las diferencias y ventajas del método PSO se resumen en tres puntos fundamentales: la sencillez de implementación, la rapidez de convergencia y el reducido número de parámetros a ajustar para sintonizar el algoritmo.

	PSO	**GA**
Concepto de la simulación	Interacción social	Evolución de las especies
Garantizan solución óptima	No	No
Operadores	Velocidad	Selección, cruce, mutación
Factores que influyen en la diversidad	Inercia	Probabilidad de cruce y probabilidad de mutación.
Parámetros a ajustar	Pocos	Muchos

3.2 Pseudo-código del algoritmo principal

```
Particula[] particulas;
Problema contextoProblema;
Particula mejorParticula;
double valoracionTotalEnjambre;

do{  //para cada iteración

        valoracionTotalEnjambre = 0;
        for (int i = 0; i < numParticulas; i++){   //para cada particula

           //actualizamos su posicion
           particulas[i].actualizarPosicion();

           //la valoramos según la función de evaluación para el problema
           valoracionParticula = contextoProblema.funcionEvaluacion(particulas[i]);

           //la asignamos dicha valoración
           particulas[i].setValoracion(valoracionParticula);

           //vemos si se ha convertido en la mejor particula del enjambre
           if (valoracionParticula > mejorParticula.getValoracion())
                 mejorParticula = particulas[i];

           //le asignamos la mejor posición que puede haber observado
           particulas[i].setMejorPosicionObsrvada(mejorParticula.getPosicion());

           //le actualizamos la velocidad
           particulas[i].actualizarVelocidad(inercia,cteAtraccion1,cteAtraccion2);

          //sumamos su valoración a la del total del enjambre
          valoracionTotalEnjambre+=valoracionParticula;
        }

    iteraccionesRealizadas++;
    disminucionInercia();
     mostrarResultadosGeneracion();

 }while( (mejorParticula.getValoracion() < valoracionObtener) &&
          (iteraccionesRealizadas<iteraccionesMax) );

return mejorParticula;
```

Figura 43. *Código maestro del PSO*

3.3 Arquitectura de la aplicación

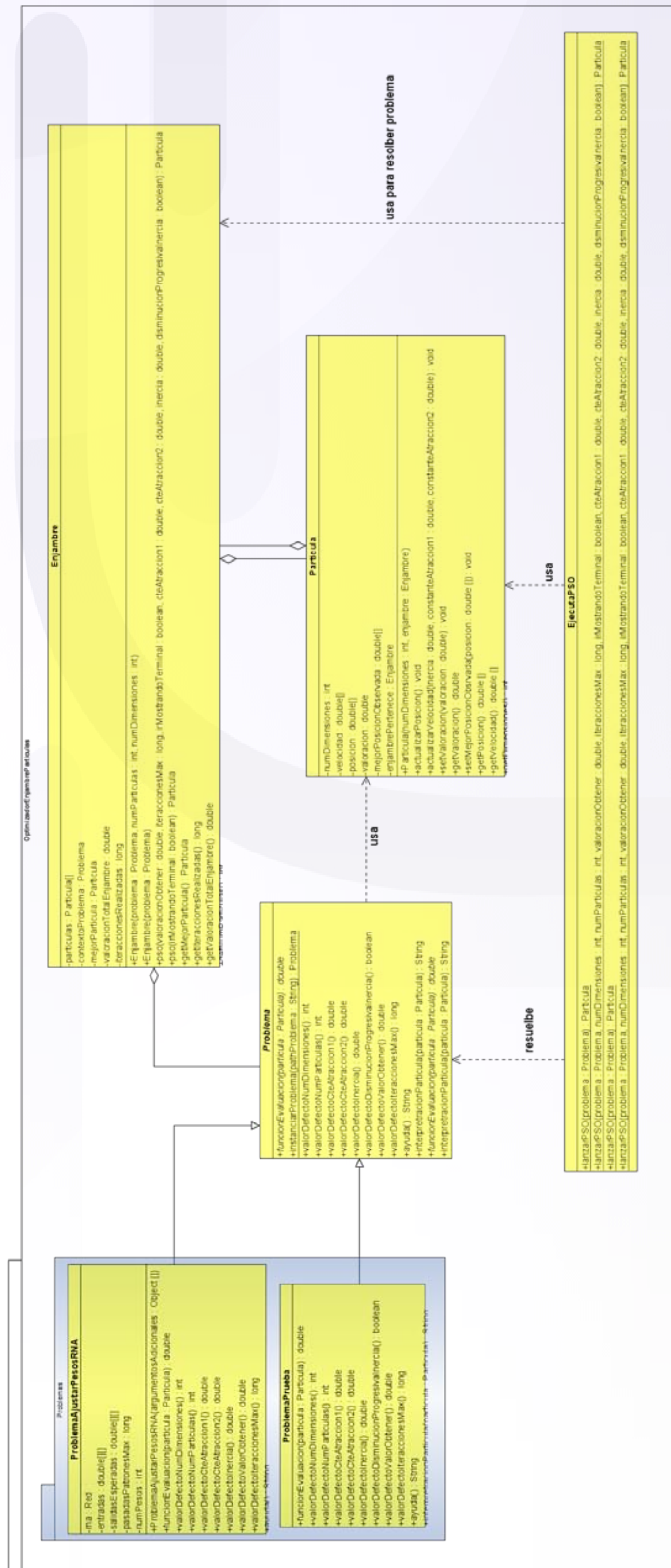


Figura 44. *Diagrama UML del Optimizador de Enjambres de Partículas*

3.4 Particularidades de la implementación

Se puede establecer una clasificación de los esquemas clásicos de PSO atendiendo principalmente a dos características del swarm: la forma en la que se sincroniza la transmisión de información entre los agentes de la población y la topología que se confiere a la población, es decir, el modo en el que se transmite la información entre cada partícula y sus convecinos. Atendiendo a esta clasificación surgen las versiones con actualizaciones síncrona y asíncrona, y las topologías global y local. Estas variantes combinadas entre sí resumen las cuatro versiones del algoritmo. Por último también se puede hablar de un esquema de PSO binario que tampoco vamos a analizar y de algunas variaciones del algoritmo para resolver problemas muy concretos, que tampoco vamos a analizar ya que nuestro objetivo es el de implementar uno genérico que resuelva todo tipo de problemas.

Nosotros hemos optado por la implementación mas sencilla y estándar del algoritmo que es una solución **SINCRONA** y **GLOBAL**. En la solución sincrona, las partículas se mueven en paralelo, es decir en cada iteración se evalúa el fitness de todas las partículas y se actualiza sus posiciones y velocidades, con lo cuál todas las partículas comparten la misma información acerca de la mejor solución desde el comienzo de la iteración. Este modelo además es susceptible de ser ejecutado en paralelo en múltiples procesadores. En cuanto a la cuestión de que usa un esquema de topología Global, quiere decir que todas las partículas conocen en cada iteración cual es la posición mejor alcanzada, en un esquema local, las partículas sólo conocen la mejor posición alcanzada por las partículas vecinas. La solución local mejora la robustez del algoritmo frente a los mínimos locales, pero no lo solventa por completo (no garantiza la mejor solución posible).

Una modificación que se ha introducido y es importante para evitar la llamada "explosión del PSO", es poder limitar la velocidad y la posición mediante una velocidad máxima y una posición máxima a alcanzar que actual de pared en el espacio de búsqueda. Esto es necesario, sino la partícula se puede salir del espacio de búsqueda o moverse con tanta velocidad que se "salte" soluciones. Si no se quiere limitar la velocidad ni la posición basta con devolver los valores máximos que admite el tipo básico usado.

3.5 Como codificar problemas

En esta sección vamos a explicar como implementar la codificación de un problema usando enjambres de partículas en nuestra aplicación, que básicamente consiste en la implementación de los métodos abstractos y la sobrescritura opcional de los métodos que no son abstractos de la clase Problema.

La codificación del problema se implementa mediante una función de evaluación (función de fitness). Podrán además darse los valores por defecto con los que se resuelve el problema mediante la simulación, como son el número de dimensiones, número de partículas, valor de las constantes, etc.

Así pues, para resolver un problema de la vida real hay que saber modelarlo como una partícula, o como un conjunto de partículas (siendo la solución el total de partículas del enjambre). Cada partícula queda determinada por su posición y velocidad. Por lo tanto la codificación del problema se debería basar en estos dos vectores que representan cada partícula.

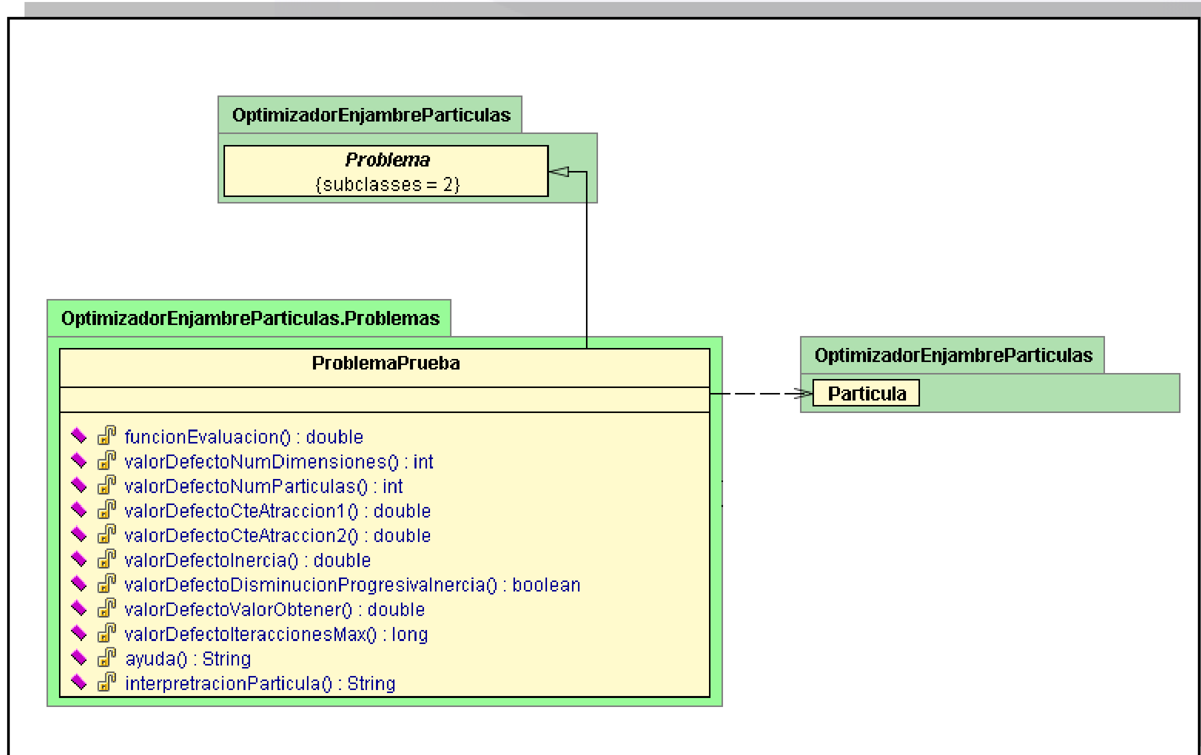


Figura 45. *Diagrama de implementación de un problema*

3.5.1 ¿Cómo crear una función de evaluación?

La tarea de construir una función de evaluación, es la tarea de transformar un problema del mundo real en un problema de optimización de posición de las partículas en un sistema de coordenadas multidimensional.

4.5.1.1. MÉTODOS DE IMPLEMENTACIÓN NECESARIA

Para la codificación del problema es necesario al menos implementar el método declarado como abstracto en la clase Problema, que es:

```
public abstract double funcionEvaluacion(Particula particula)
public abstract boolean isMaximiceFuntion()
```

Figura 46. *Métodos que deben ser implementados necesariamente en cada problema*

Este método es simplemente la función de fitness, en ella se deberá valorar la partícula en función de su posición y/o velocidad. Así pues el código de la función de evaluación deberá ser del tipo:

```
public double funcionEvaluacion(Particula particula){
  double[] posicion = particula.getPosicion();
  double[] velocidad = particula.getVelocidad();
  …
  return valoración;
}
```

Figura 47. *Estructura típica de la función de evaluación*

El segundo método sirve únicamente para indicar si se trata de un problema de maximización o un problema de minimización de la función de evaluación.

4.5.1.2 Métodos de implementación opcional

Los métodos de implementación opcional tienen una implementación por defecto dada en la clase Problema, si bien, suele ser importante darle una implementación adaptada el problema en concreto para ajustar bien los parámetros del algoritmo. Son los siguientes métodos:

```
public int valorDefectoNumDimensiones()
public int valorDefectoNumParticulas()
public double valorDefectoCteAtraccion1()
public double valorDefectoCteAtraccion2()
public double valorDefectoInercia()
public boolean valorDefectoDisminucionProgresivaInercia()
public double valorDefectoValorObtener()
public long valorDefectoIteraccionesMax()
public String ayuda()
public String interpretracionParticula(Particula particula)
```

Figura 48. *Métodos que pueden ser sobrescritos en cada nuevo problema*

Todos estos métodos establecen los valores por defecto con los que se realizará la simulación salvo que se especifiquen otros valores. Al seleccionar un problema en la interfaz gráfica, se mostrarán los valores por defecto de dicho problema, establecidos con estos métodos. El método ayuda sirve para proporcionar ayuda textual sobre la codificación del problema y los valores que deben tomar los parámetros en la simulación, la interpretación de la partícula sirve para que una vez que se ha solucionado el problema, informar al usuario sobre como la codificación de esa partícula resuelve el problema.

3.6 Modo funcionamiento

Existen principalmente dos maneras de usar la aplicación: mediante la interfaz gráfica, o usándose directamente desde otra aplicación externa. La ventaja de usar la interfaz de usuario, es poder ver directamente como evoluciona el resultado gráficamente. No siempre nos interesa hacer uso de la interfaz gráfica, y en esos casos solamente nos interesa llamar al método que ejecuta la simulación sin mostrar interfaz gráfica. Vamos a presentar las dos maneras de usar la aplicación.

4.6.1 La interfaz gráfica de Usuario

4.6.1.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

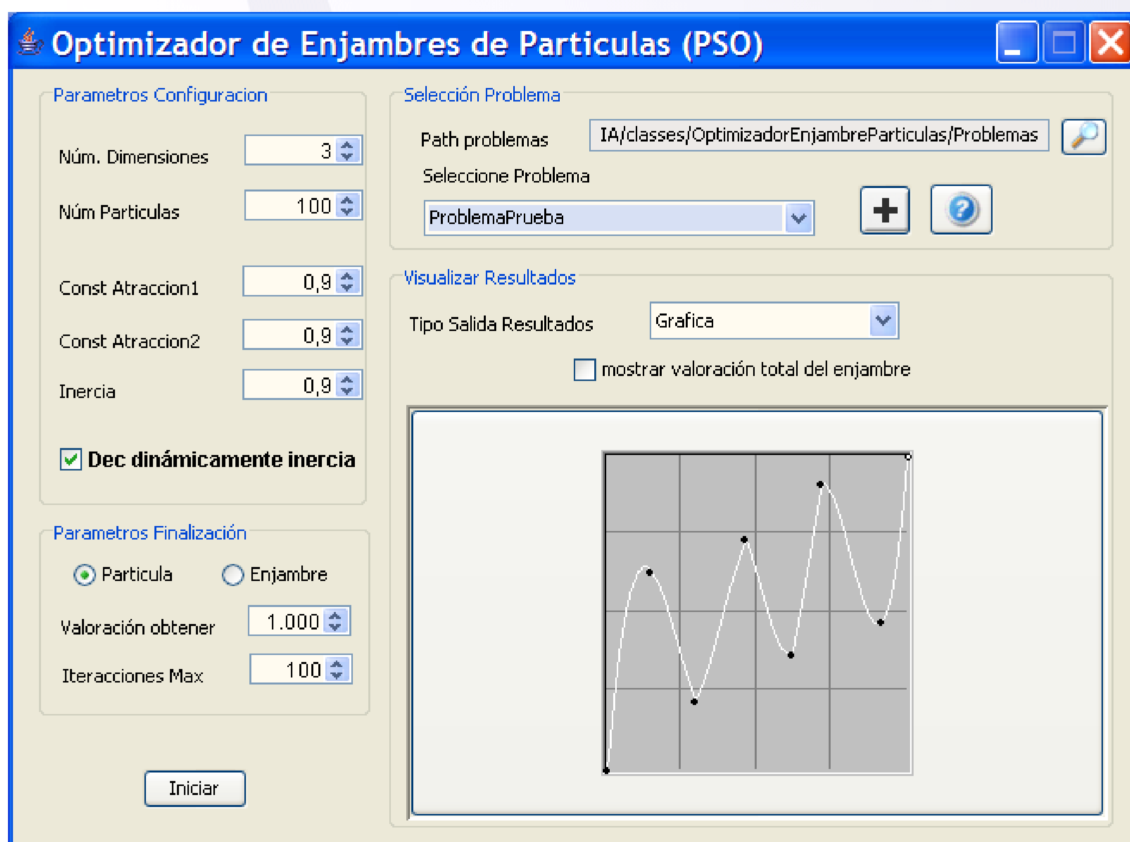


Figura 49. *GUI inicial*

Esta es la vista que presenta la interfaz gráfica de usuario de la herramienta proporcionada para la optimización mediante Enjambres de Partículas. Esta dividida en

cuatro panes principales: parámetros de la simulación, parámetros de finalización de la simulación, selección del problema y panel de visualización de resultados.

En el panel llamado "**Parámetros**" , situado arriba a la izquierda se puede seleccionar el valor que tendrán los parámetros del algoritmo PSO, tales como el número de dimensiones en el espacio euclideo que se mueven las partículas, el número de partículas que conforman el enjambre, la constante de atracción 1 que es un factor que escala cuanto atrae la mejor partícula observada localmente por cada partícula a esa propia partícula, la constante de atracción 2 que indica cuanto atrae la mejor partícula del enjambre al resto de partículas, la inercia que indica como influye la velocidad de la generación anterior a la siguiente, y por último la posibilidad de decrementar dinámicamente la inercia hasta un valor de 0.4, de esta manera conseguimos que primero se "exploren" el mayor número posible de soluciones y posteriormente se produzca una "explotación" de la solución.

En el panel "**Parámetros de Finalización**" se debe elegir cual es la puntuación máxima a obtener (referida a la partícula o a la suma total de puntuaciones de las partículas del enjambre), y las generaciones máximas a evaluar, es decir, el número máximo de iteraciones del algoritmo.

El panel "**Selección del Problema**" se encarga de cargar los programas compilados previamente o crear directamente problemas nuevos para ser compilados y ejecutados desde la propia interfaz o de manera externa. La ruta donde de donde se cargan los problemas y en donde se crean puede ser introducida en el recuadro textual que tiene a su derecha una lupa. Si la ruta no se cambia, cargará los problemas que vienen implementados con el proyecto, si se indica otra ruta, el cargador de clases intentará cargar los problemas compilados que se encuentren en dicha ruta, pinchando en la lupa.

Para crear nuevos problemas, en la ruta indicada, se pedirá un nombre del problema, y se analizará dicha ruta; si la ruta era la original, se creará el problema dentro del propio proyecto, y se mostrará una ventana desde la que se puede trabajar con el código y compilarlo. Si el código queda compilado correctamente se podrá directamente usar desde la interfaz gráfica (sin tener que cerrar y volver a lanzar la aplicación). Posteriormente el código puede ser otra vez editado y compilado usando el

Make.bat del propio proyecto. Si por el contrario, se ha creado el problema en una ruta distinta de la de por defecto dentro del propio proyecto, se generarán los ficheros relativos al problema en la ruta indicada, así como un archivo *Make.bat* que permite compilar dichos archivos. Esta segunda opción, es importante para crear problemas que no serán incluidos posteriormente en el proyecto, y que simplemente quieren usar la librería HerramientasIA.jar para valerse de las herramientas que esta librería ofrece.

Para obtener ayuda sobre un problema creado y compilado, se podrá pinchar en el botón de ayuda, que mostrará la información relativa al problema que haya sido introducida por el programador.

Finalmente se puede seleccionar el tipo de individuo con el que se hará la simulación, de entre los soportados por el problema, también especificados por el propio programador.

En el panel "**Visualizar Resultados**" se puede elegir la manera en que los resultados del simulador serán presentados al usuario. Si se elige una presentación por "*terminal*", los resultados se irán mostrando por la consola del sistema operativo, si la opción elegida es por "fichero", los resultado se grabarán en un fichero con el nombre que se indique; y por último si los resultados se quieren mostrar en la interfaz gráfica se podrá elegir "Grafica"; dentro de esta última opción, se podrá además mostrar en una gráfica la evolución de tanto el mejor individuo como el individuo medio de cada generación.

Además en los tres modos de salida de resultados se podrá activar/desactivar que se muestre la partícula media del enjambre activando la casilla "*mostrar valoración total del enjambre"*.

Por último decir que se puede iniciar la simulación pinchando en el botón "Iniciar". Una vez que se inicia la simulación, los parámetros que no se pueden modificar en tiempo de ejecución se ensombrecen para que no puedan ser modificados, y se vuelven a activar cuando la simulación acaba.

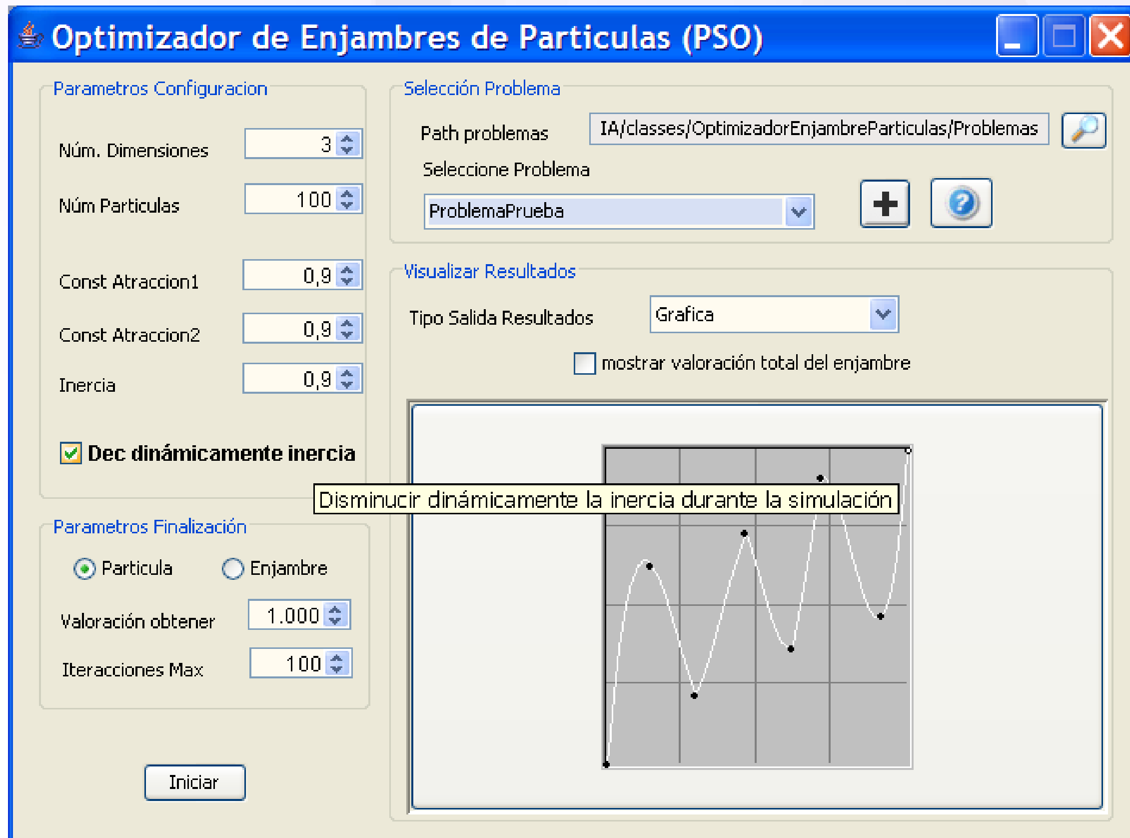


Figura 50. *GUI con información tipo "Tool Tip"*

El aspecto de la herramienta se ha cuidado sobremanera intentando que quedará lo mas intuitivo y simple posible, pero aún así puede resultar en un principio algo confusa, para evitar confusiones, se ha añadido información textual a la mayoría de los elementos que la forman. Para obtener información dicha información basta con mantener el ratón situado unos segundos encima, momento en el cual aparecerá el llamado "tool tip". Incluso se puede obtener información sobre algún punto de la gráfica manteniendo el ratón encima del punto deseado y aparecerá las coordenadas relativas a ese punto de la gráfica.

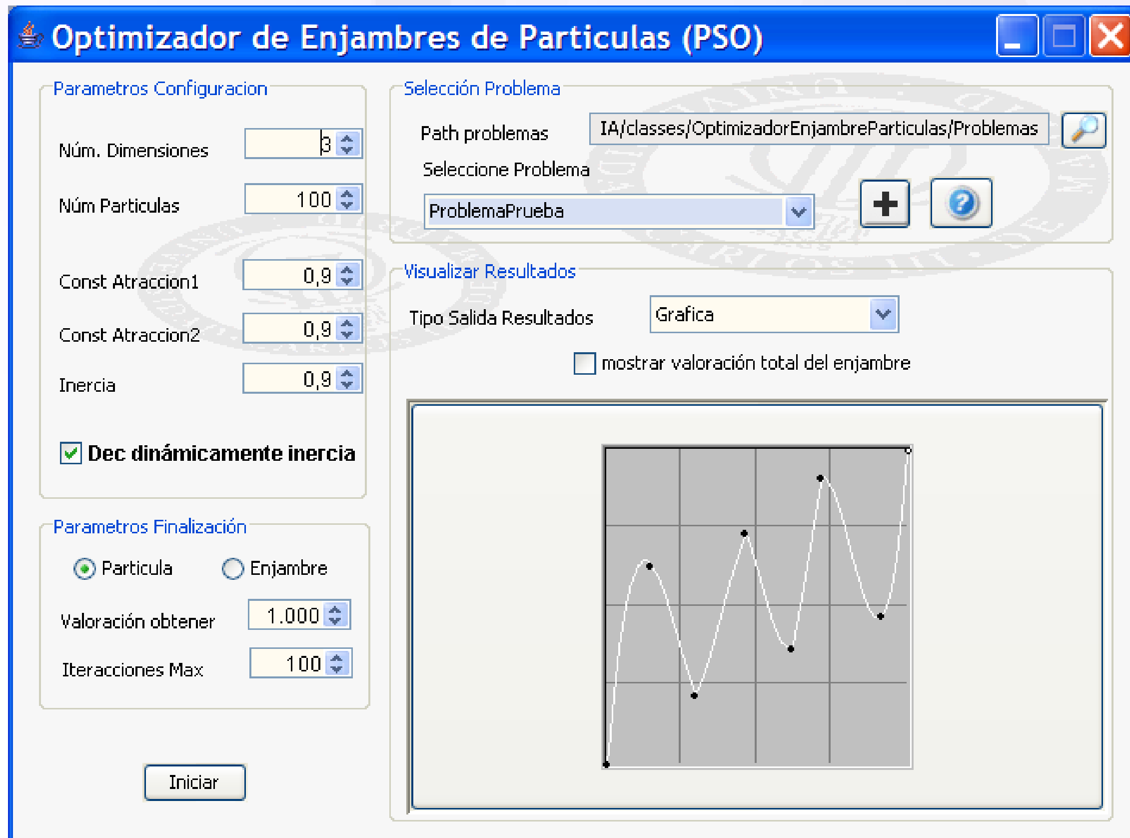


Figura 51. *GUI con nuevo skin (fondo)*

Para conseguir un aspecto más atractivo, se ha añadido además la característica común a todas las herramientas del proyecto de poder cambiar la foto de fondo (el skin o apariencia). Para ello basta con pulsar con el botón derecho sobre cualquier parte de la aplicación y pinchar en cambiar foto de fondo sobre el menú emergente que aparece, posteriormente se selecciona la foto deseada y quedará establecida como fondo. Por defecto en la carpeta "*skin*" del proyecto se adjuntan algunos fondos.

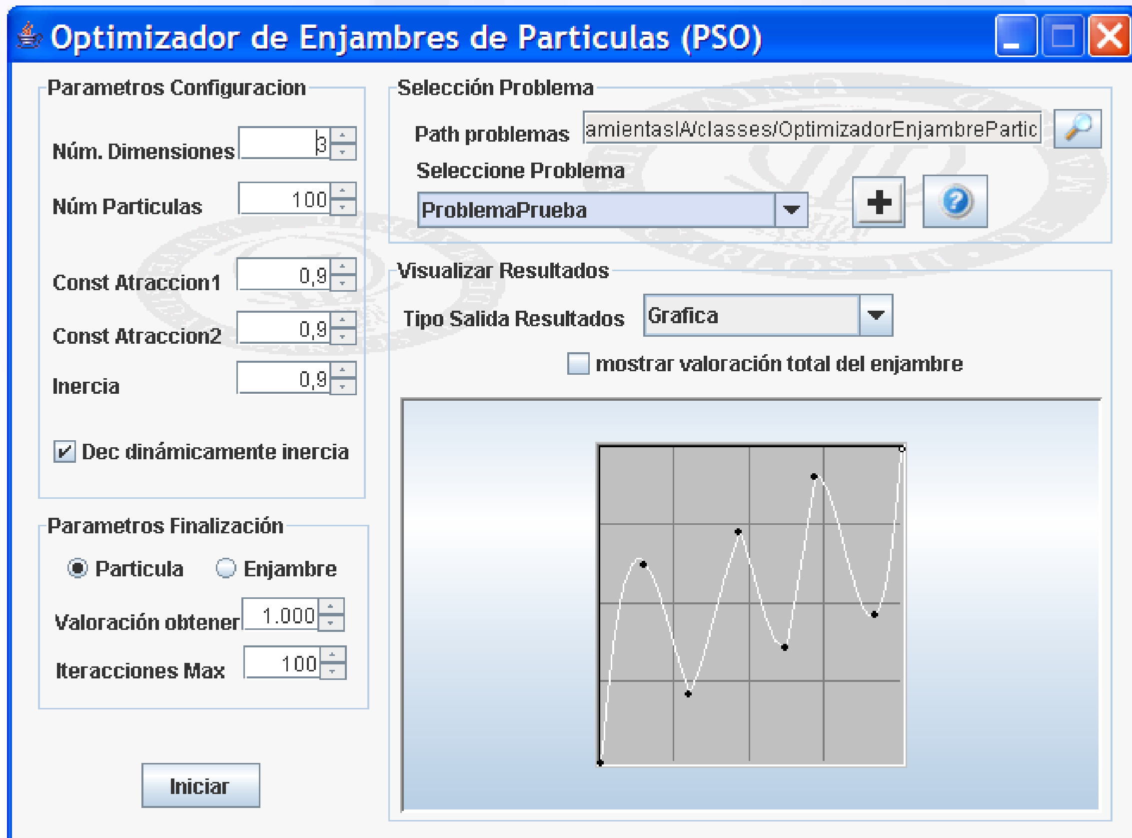


Figura 52. *GUI con nuevo Look & Feel*

También se añadido la posibilidad de cambiar el "look&feel" de la ventana, pudiendo intercambiarse entre varios "Look & Feel", como por ejemplo: java, metal, windows, borland…. Nuevamente esta acción se puede realizar pulsando en el botón derecho del ratón y seleccionando en "cambiar loock&feel", que irá alternando entre los disponibles.

4.6.1.2 EJECUTAR UN PROBLEMA DESDE LA INTERFAZ DE USUARIO

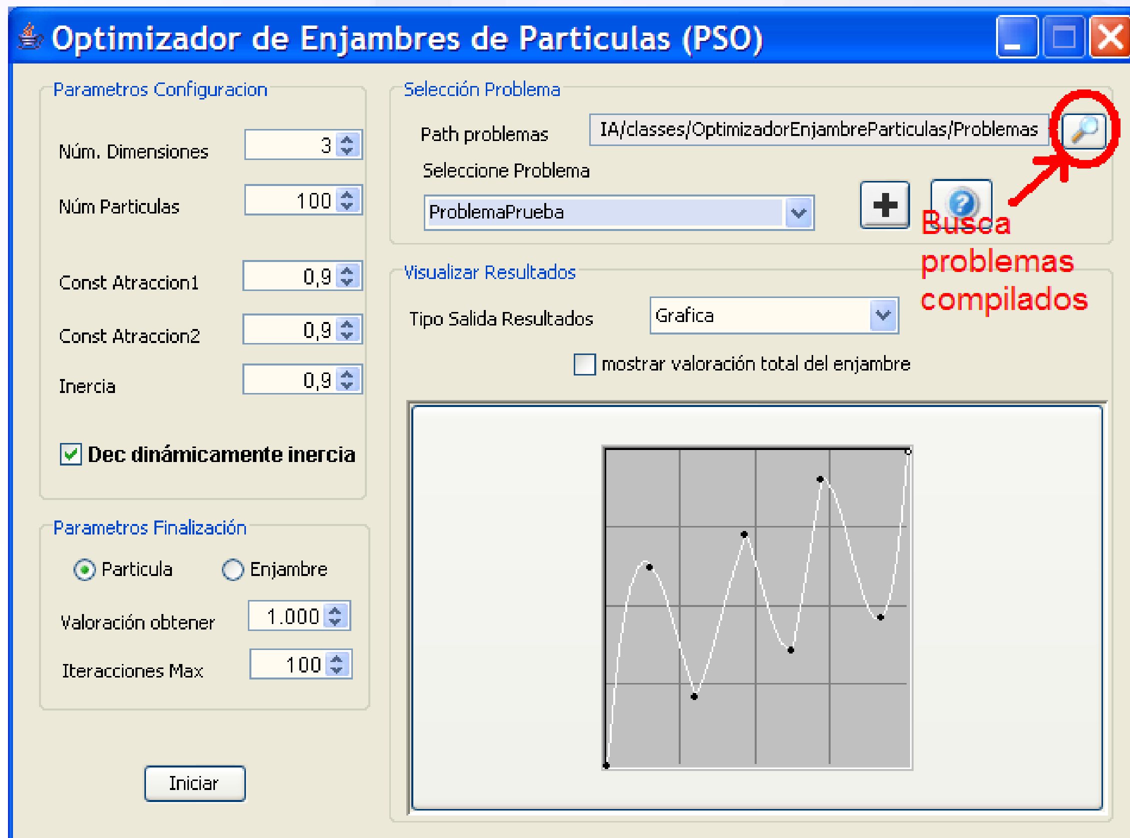


Figura 53. *Buscar problemas en la ruta indicada*

Para buscar los problemas implementados en el propio proyecto o en problemas externos al mismo, basta con indicar la ruta del problema y pinchar en la lupa. De esta manera se cargarán los problemas compatibles con el simulador. Si el problema es externo a la aplicación (se ha creado fuera de la ruta por defecto) se cargará mediante el cargador de clases de Java.

Notar que algunos problemas no podrán ser cargados y así se indicará por terminal, bien porque no están compilados, o no lo están adecuadamente, o simplemente porque no son válidos para ser ejecutados desde la propia interfaz gráfica, puesto que necesiten argumentos que no puedan ser pedidos desde la misma (como objetos), un ejemplo de un problema que no puede ser cargado desde la interfaz gráfica es el de ajustar los pesos de una red neuronal, ya que necesita recibir en el constructor del propio problema la red de neuronas con la que va a trabajar y evaluar el individuo. Es decir, sólo podrán ser cargados en la interfaz gráfica aquellos problemas en cuyo constructor no se reciba ningún argumento (si pudiéndolos pedir posteriormente dentro del propio constructor). Los problemas que necesitan recibir argumentos en el constructor serán

adecuados para ser lanzados sin la interfaz gráfica, como se puede ver en el punto "3.5.2 Ejecutando el simulador sin la interfaz gráfica".

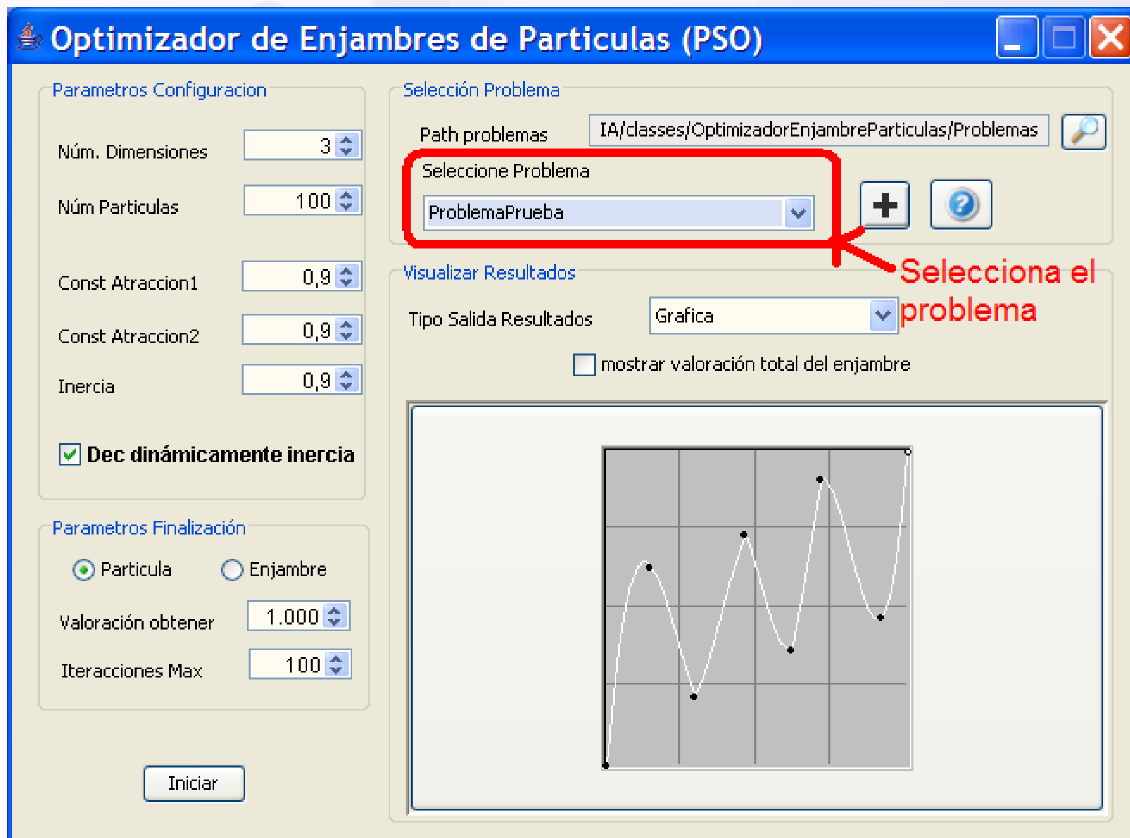


Figura 54. *Selecciona el problema que se quiere solucionar*

De entre los problemas cargados en la ruta indicada se podrá seleccionar aquel que queramos resolver. Una vez seleccionado se establecerán los valores por defecto del problema (si se ha indicado por el programador y sino los valores por defecto de cualquier problema).

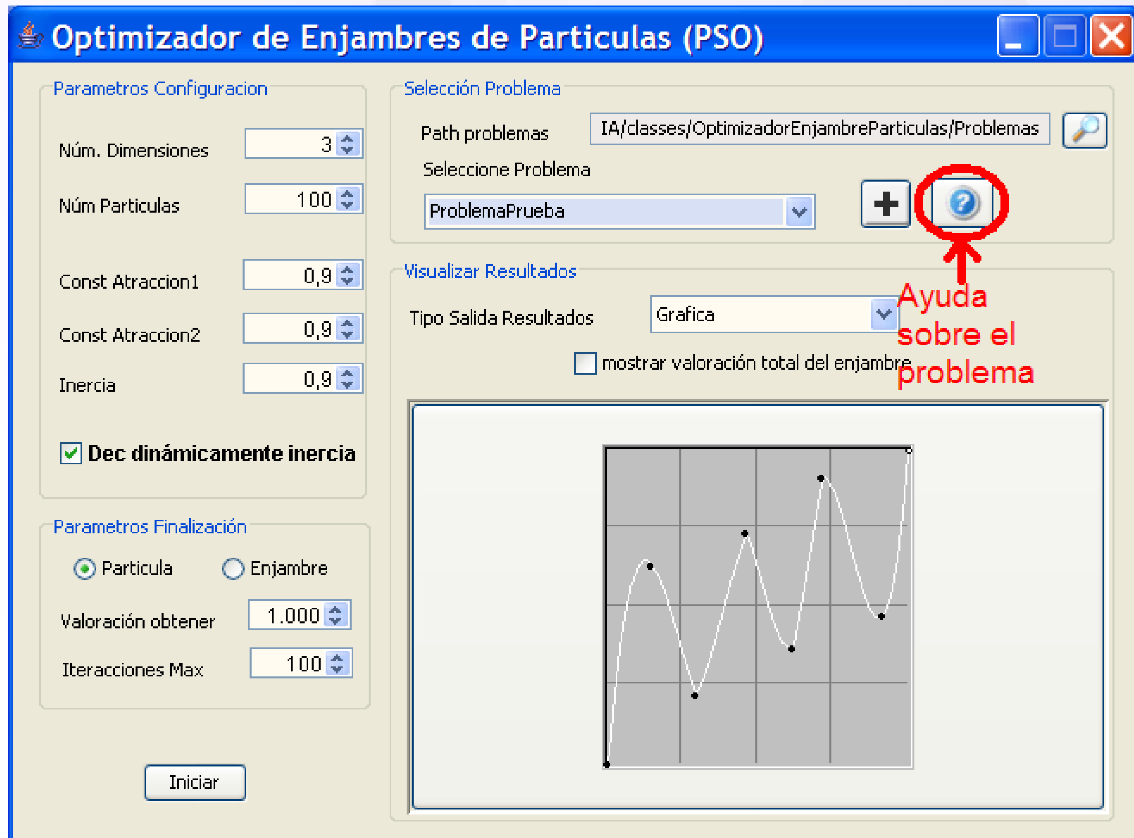


Figura 55. *Obtener ayuda sobre el problema seleccionado*

Si se pincha en el botón de ayuda, se podrá obtener ayuda sobre el propio problema y sobre los valores por defecto establecidos para ese problema. Esta ayuda la introduce el programador del problema, mostrándose la de por defecto sino se ha introducido ninguna nueva.

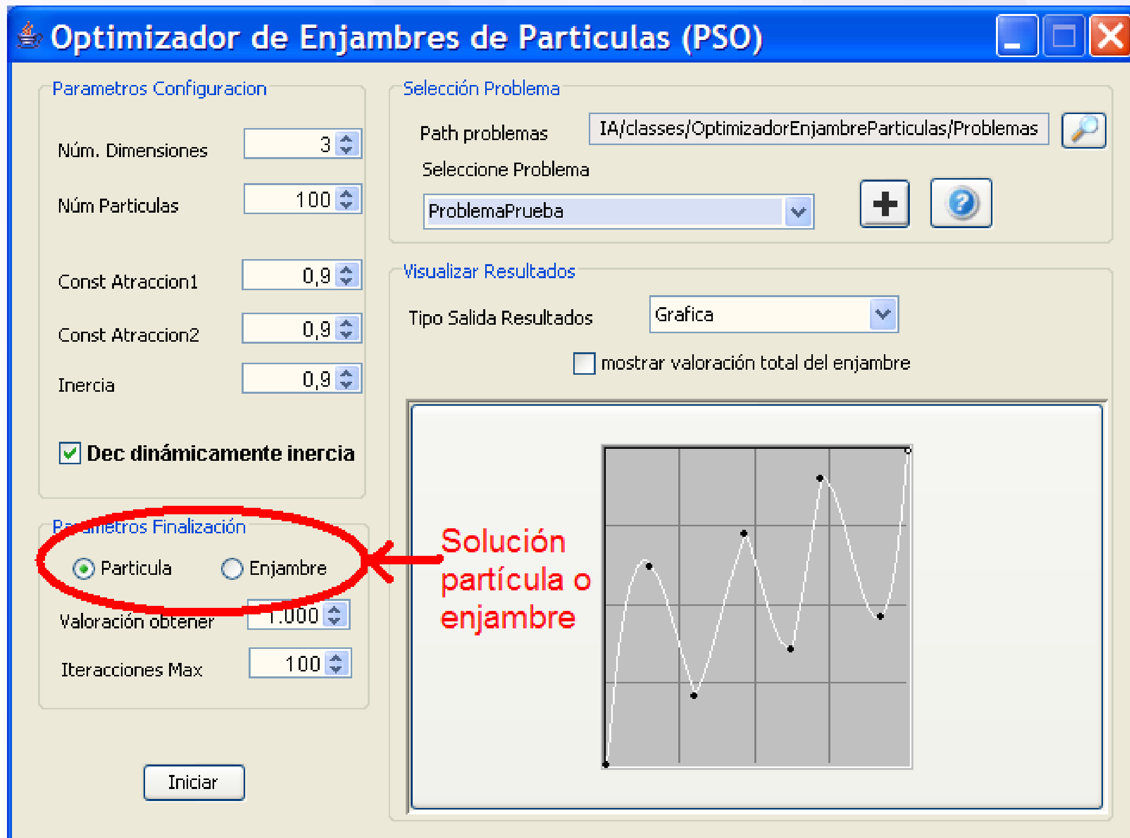


Figura 56. *Tipo de solución*

La solución que se quiere obtener puede ser un individuo en concreto o toda la población de una determinada generación, para ello, se puede indicar en que tipo de problema nos encontramos. Por defecto la puntuación a obtener será el de la población. Se deberá indicar la puntuación que se quiere obtener en caso de solución óptima, es decir, la máxima valoración que puede obtener un individuo; al igual que también se debe indicar el número máximo de generación a explorar.

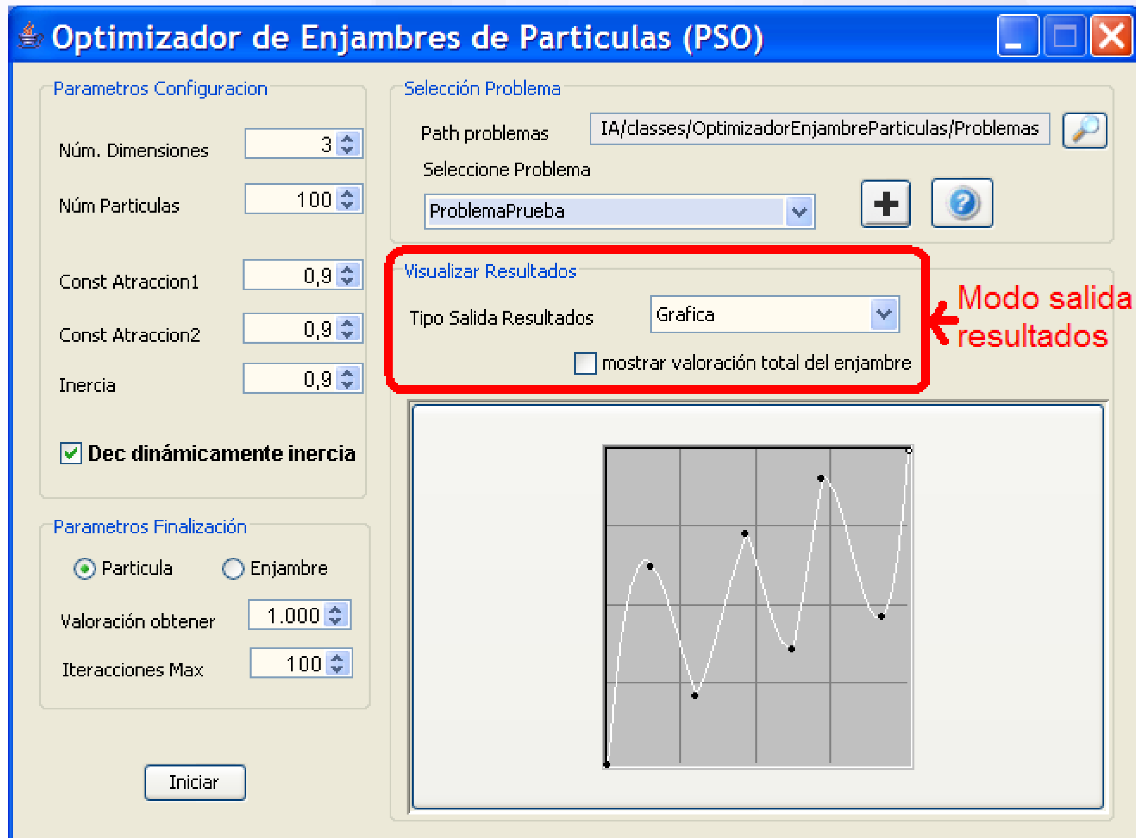


Figura 57. *Visualizar los resultados*

Los resultados y la evolución del algoritmo se pueden mostrar de distintas maneras. La manera usual de mostrar los resultados será de manera gráfica (es decir en la interfaz gráfica), pudiéndose escoger la opción de generar la gráfica que relacione la puntuación obtenida en cada generación. También podrán mostrarse la evolución y los resultados por consola o por fichero. En los tres casos se puede ir mostrando además de la mejor partícula, la partícula media. Estas opciones de visualización pueden cambiarse incluso durante la propia simulación. Por último mencionar que se puede decidir cada cuantas iteraciones se mostrará la evolución del algoritmo, por defecto es en todas.

Hacer notar, que cuanta mas salida de resultados se realicen mayor tiempo se invertirá en la simulación.

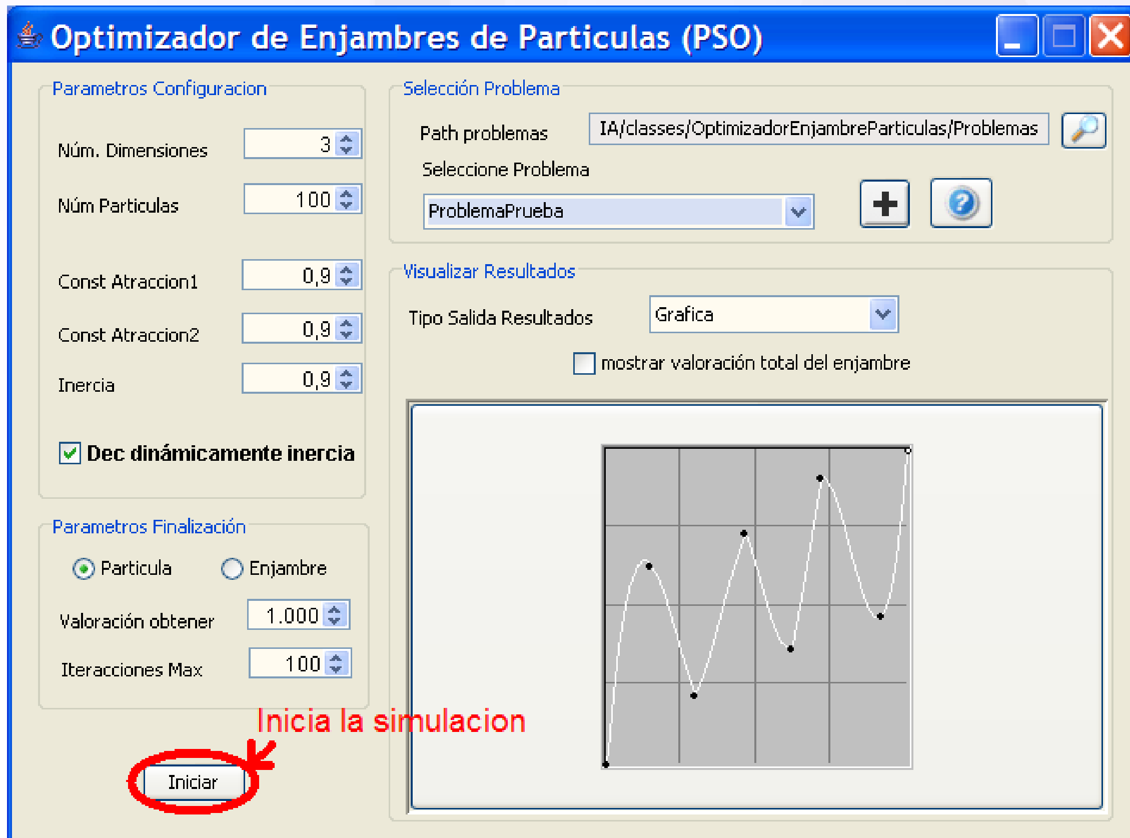


Figura 58. *Iniciar/parar la simulación*

Con los botones "Iniciar" y "Abortar" se podrá, como su nombre dicen, iniciar y abortar la simulación respectivamente. En el caso de abortar la simulación se mostrará la mejor partícula obtenido hasta el instante actual y se parará de ejecutar el algoritmo de optimización, pudiéndose volver a iniciar desde el principio con los mismos o con otros parámetros distintos a los de la anterior simulación.

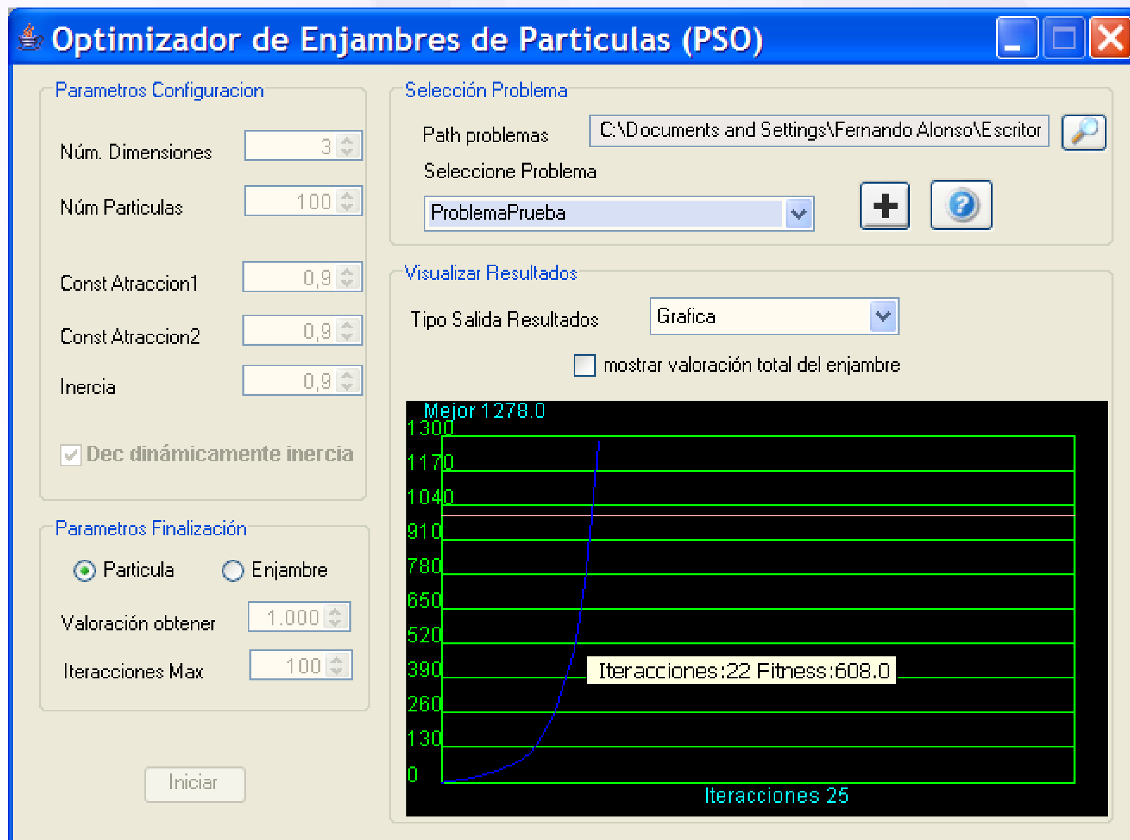


Figura 59. *Ejecutando simulación*

Una vez iniciada la simulación se desactivan (ensombrecen) los parámetros que no pueden ser modificados en tiempo de ejecución, los que no se desactivan sí pueden ser modificados durante la simulación. También se puede ver como se va creando dinámicamente la gráfica que relaciona las iteraciones con la puntuación obtenida. Si se mantiene el ratón encima de cualquier punto de la gráfica se pueden ver sus coordenadas exactas. En la gráfica se representa en el eje horizontal las iteraciones, del algoritmo, ya realizadas, mientras que en el eje vertical se representa la puntuación obtenida. La línea horizontal rosa que cruza la gráfica, es la puntuación que se desea obtener. Muchas veces suele ocurrir que en una iteración no se alcance la puntuación deseada, y sin embargo en la siguiente se rebase en mucho esa puntuación, por lo que se ha reescalado la gráfica para que no se salga de la cuadrícula.

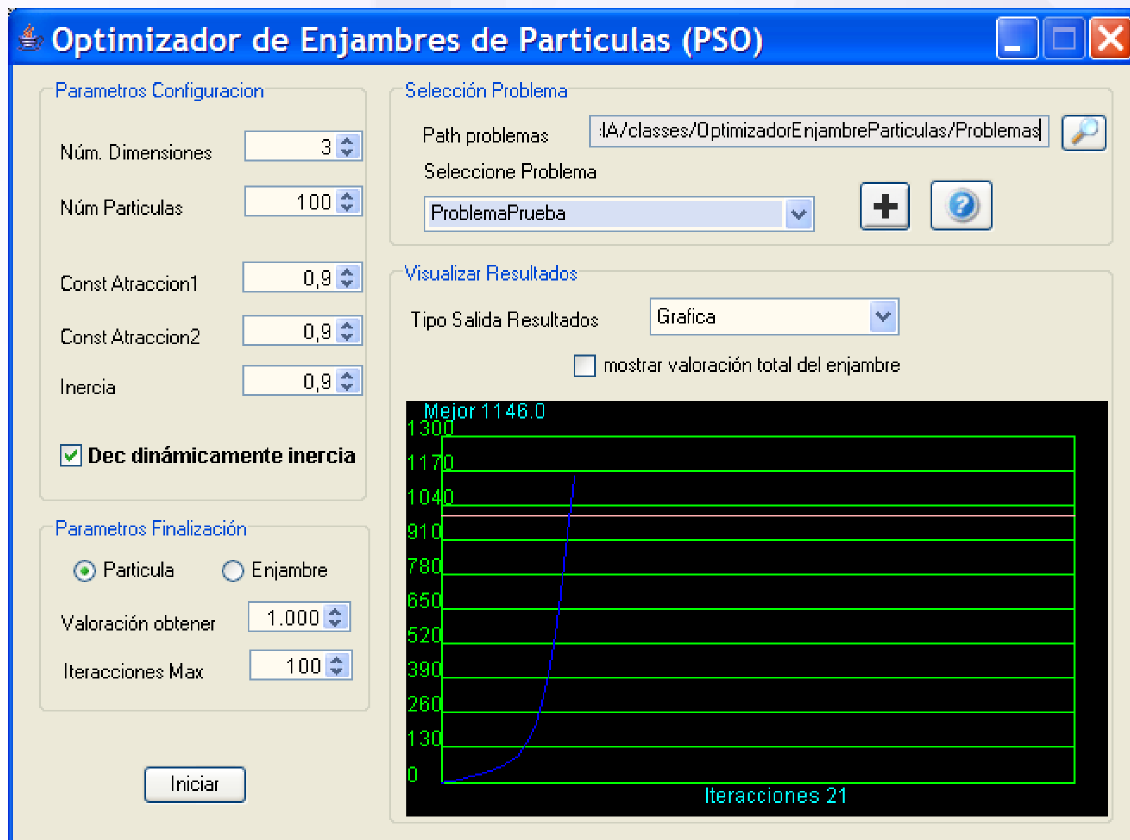


Figura 60. *Fin de la simulación*

Al acabar la simulación se puede seguir inspeccionando la gráfica, y se vuelven a activar todos los parámetros para que puedan ser modificados y se pueda volver a iniciar la misma simulación o cualquier otra.

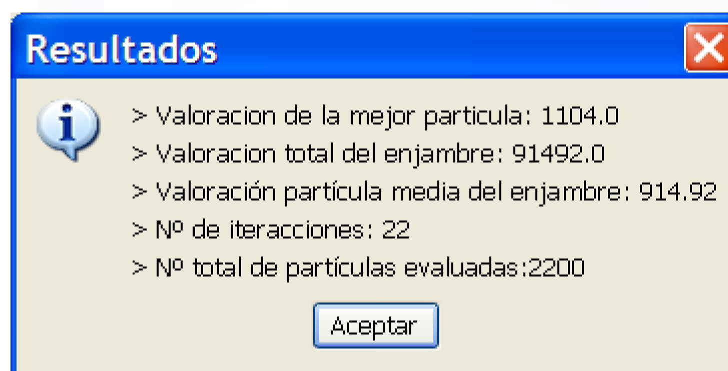


Figura 61. *Resultados de la simulación*

Al finalizar la simulación se muestran los resultados obtenidos.

4.6.1.3 CREAR UN NUEVO PROBLEMA DESDE LA INTERFAZ DE USUARIO

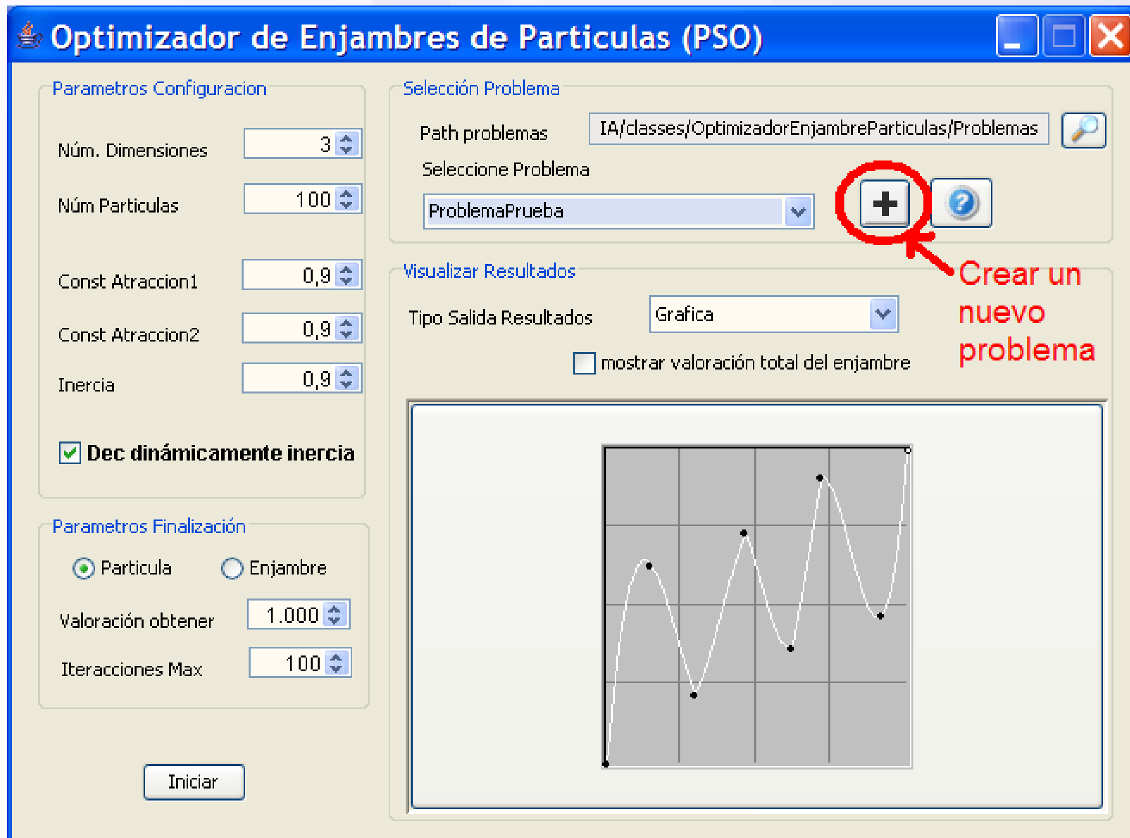


Figura 62. *Crear nuevo problema*

Si se desea crear un problema que estará dentro del propio proyecto (cada vez que se haga un make será también compilado) se debe dejar la ruta especificada en "*Path problemas*" sin modificar. Vamos a analizar ambos casos:

- CREANDO EL PROBLEMA DENTRO DEL PROPIO PROYECTO

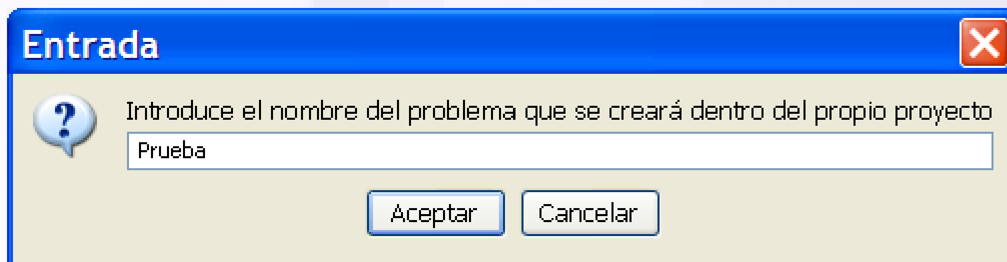


Figura 63. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a la clase que se creará.

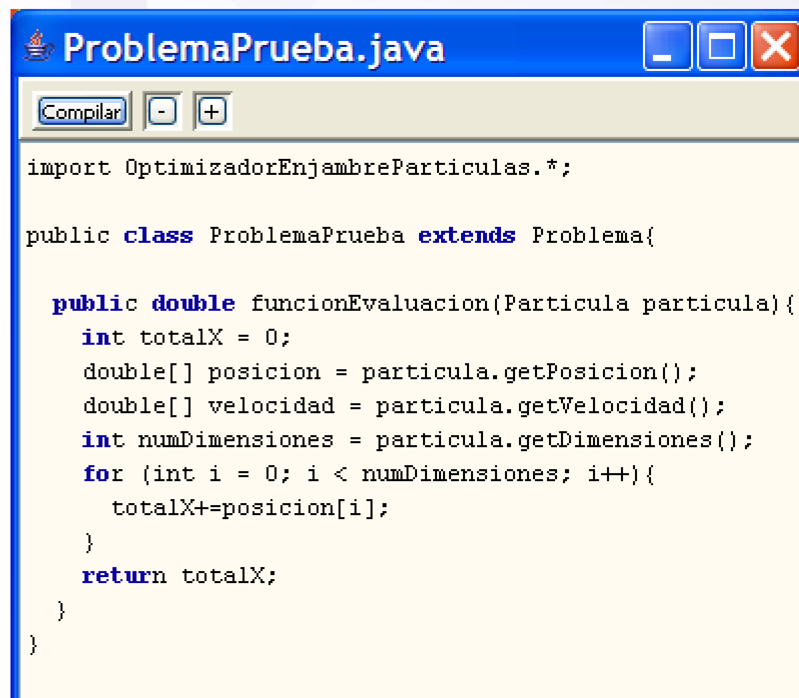


Figura 64. *Implementar el problema*

A continuación se muestra un editor desde el que se puede editar y compilar el problema. De primeras se muestra un esqueleto por defecto de cómo debe ser el problema, este se debe modificar para adaptarlo a la codificación del problema que queramos hacer y finalmente compilarlo.

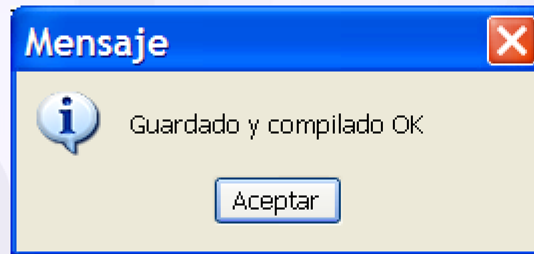


Figura 65. *Problema guardado y compilado*

Si el problema compila correctamente, nos pondrá guardado y compilado. El archivo .java se guarda dentro de la carpeta indicada en el path por defecto que es *src/enjambreParticulas/problemas* y el .class se guarda en la misma ruta pero de la carpeta classes. Posteriormente puede ser editado nuevamente y compilado compilando todo el proyecto usando el *make.bat* que se suministra con el proyecto.

Finalmente se puede seleccionar el problema que se acaba de crear para poder ser ejecutado, tal y como se dijo en el apartado "ejecutar un problema desde la interfaz de usuario", sin tener que cerrar y volver a lanzar la aplicación (salvo que se haya ejecutado la aplicación directamente desde el jar, que entonces debería recompilarse todo el proyecto con el *make* y volver a lanzarlo).

- CREANDO EL PROBLEMA FUERA DEL PROYECTO

Cuando se cambia el path por defecto donde se crearán los problemas, se crean en una ruta distinta de la asignada en el proyecto, por lo que se generarán en la ruta indicada los ficheros esqueleto del problema para que puedan ser editados y compilados con posterioridad.

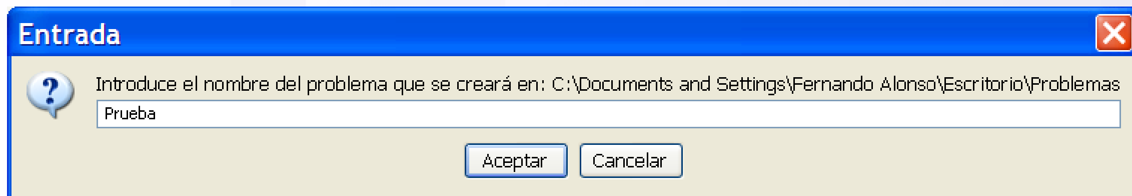


Figura 66. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a la clase que se creará.

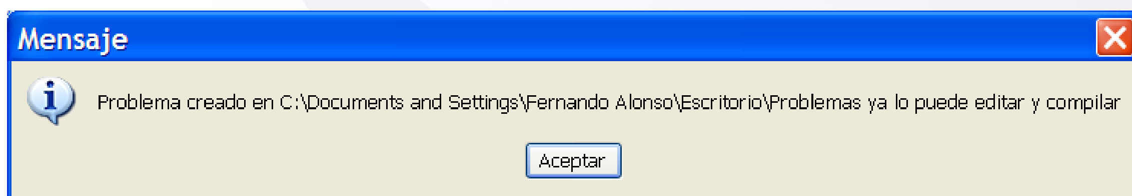


Figura 67. *Mensaje informativo de que se han guardado los ficheros*

Si la ruta indicada es correcta se creara el problema en la ruta indicada y se mostrará el mensaje de la figura anterior. Se habrán guardado los archivos java en la ruta indicada pero no habrán sido compilados. Para compilarlos se deberá ejecutar el archivo "*Make.bat*" que se genera conjuntamente con los archivos, y para los que únicamente usa para compilarlos es la librería "*HerramientasIA.jar*".

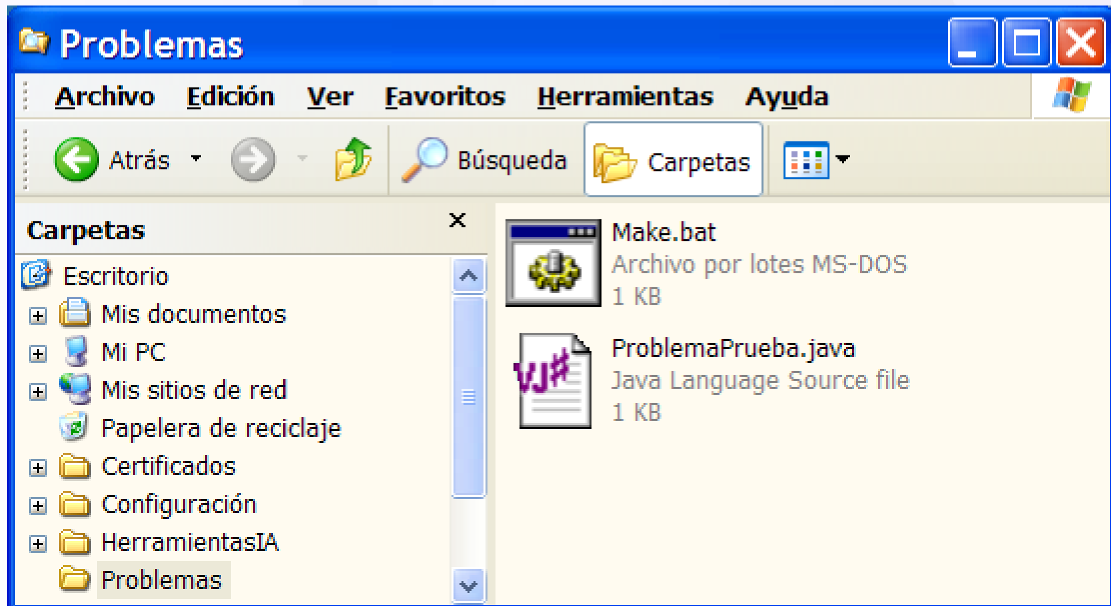


Figura 68. *Archivos fuentes con el esqueleto del problema creado*

Una vez compilados ejecutando "*Make.bat*", puede ser el problema cargado y ejecutado como se comenta en la sección "ejecutar un problema desde la interfaz de usuario".

4.6.2 Ejecutando el simulador sin la interfaz gráfica

Un problema creado valiéndonos de la interfaz gráfica de usuario, como se dice en la sección "Crear un nuevo problema desde la interfaz de usuario", o sin valernos de ella, puede ser ejecutado sin necesidad de ejecutar el main de la aplicación y por tanto de la interfaz gráfica. Para ello se deberá desde el programa que se quiera usar escribir las siguientes líneas:

```
//importamos el Optimizador de Enjambres de Partículas
import OptimizadorEnjambreParticulas.*;

//instanciamos el problema
Problema problemaPrueba = new ProblemaPrueba();

//lanzamos el algoritmo
Particula particulaSolucion =  EjecutaPSO.lanzarPSO (problemaPrueba);

//obtenemos la solución
 double[] solucion = particulaSolucion.getPosicion();
```

Figura 69. *Ejecuta problema con los parámetros por defecto*

Lanza el optimizador de manera síncrona y con los valores por defecto dados en la implementación del problema (sino se han sobrescrito los métodos que dan los valores por defecto se quedan los que tienen todos los problemas). Notar que en este caso, se pueden ejecutar problemas cuyo constructor reciba argumentos, como por ejemplo objetos, aunque en el ejemplo no se ha pasado ningún argumento; mientras que ejecutando el problema mediante interfaz gráfica el problema no podía recibir nada por argumentos, puesto que se instancia el problema desde la propia interfaz gráfica con el constructor por defecto sin argumentos.

```
//importamos el Optimizador de Enjambres de Partículas
import OptimizadorEnjambreParticulas.*;

//instanciamos el problema
Problema problemaPrueba = new ProblemaPrueba(parametrosProblema);

//parámetros de la simulación
int numDimensiones = 3;
int numParticulas = 700;
double valoracionObtener = 100;
long iteraccionesMax = 1000;
boolean irMostrandoTerminal = true;
double cteAtraccion1=0.9;
double cteAtraccion2=0.9;
double inercia=0.9;
boolean disminucionProgresivaInercia = true:

//lanzamos el algoritmo PSO
Particula particulaSolucion =
OptimizadorEnjambreParticulas.EjecutaPSO.lanzarPSO(problemaPrueba,
                                                   numDimensiones,
                                                   numParticulas,
                                                   valoracionObtener,
                                                   iteraccionesMax,
                                                   irMostrandoTerminal,
                                                   cteAtraccion1,
                                                   cteAtraccion2,
                                                   intercia,
                                                   disminucionProgresivaInercia);


//obtenemos la solución
 double[] solucion = particulaSolucion.getPosicion();
```

Figura 70. *Ejecuta problema sin usar los valores por defecto*

Esta segunda manera de lanzar el algoritmo es lanzándolo con los valores concretos que queramos darle, sin tener en cuanta los valores por defecto codificados. Un ejemplo de esta llamada, se hace desde la herramienta Red de Neuronas Artificiales para entrenar la red mediante algoritmos genéticos, ver la sección "Particularidades de la implementación".

4. Simulador de Colonias de Hormigas

4.1 Introducción

Este tipo de técnica se engloba dentro de lo que ha venido en denominarse "Swarm intelligence" que son un conjunto de técnicas de inteligencia artificial basadas en el estudio del comportamiento colectivo en sistemas descentralizados y autoorganizativos.

Es un algoritmo de optimización heurístico (de minimización concretamente) que puede ser usado para encontrar soluciones aproximadas a problemas difíciles de optimización combinacional. El algoritmo se ha venido a denominar ACO (Ant Colony Optimization) y en el las hormigas artificiales representan soluciones gracias al movimiento en el problema grafo.

Las hormigas artificiales tratan de imitar a las reales, por ello resulta realmente interesante analizar como las hormigas buscan su alimento y logran establecer el camino más corto para luego regresar a su nido. Para esto, al moverse una hormiga, deposita una sustancia química denominada feromona como una señal odorífera para que las demás puedan seguirla.

Las feromonas son un sistema indirecto de comunicación química entre animales de una misma especie, a modo de información, la cual es recibida en el sistema olfativo del animal receptor, quien interpreta esas señales, jugando un papel importante en la organización y la supervivencia de muchas especies.

Al iniciar la búsqueda de alimento, una hormiga aislada se mueve a ciegas, es decir, sin ninguna señal que pueda guiarla, pero las que le siguen deciden con buena probabilidad seguir el camino con mayor cantidad de feromonas. Pongamos el ejemplo de un grupo de hormigas intentando evitar un obstáculo entre el hormiguero y la comida:

Las hormigas llegan a un punto donde tienen que decidir por uno de los caminos que se les presenta, lo que resuelven de manera aleatoria. En consecuencia, la mitad de las hormigas de dirigirán hacia un extremo y la otra mitad hacia el otro extremo.

Como las hormigas se mueven aproximadamente a una velocidad constante, las que eligieron el camino más corto alcanzarán el otro extremo más rápido que las que tomaron el camino más largo, quedando depositado mayor cantidad de feromona por unidad de longitud. La mayor densidad de feromonas depositadas en el trayecto más corto hace que éste sea más deseable para las siguientes hormigas y por lo tanto la mayoría elige transitar por él. Considerando que la evaporación de la sustancia química hace que los caminos menos transitados sean cada vez menos deseables y la realimentación positiva en el camino con más feromonas, resulta claro que al cabo de un tiempo casi todas las hormigas transiten por el camino más corto.

4.2 Pseudo-código del algoritmo principal

```
//inicializamos la matriz de feromonas con el valor por defecto
double valorFeromonaDefecto = 1 / (numPosiciones * costeAproximadoTrayecto);
for (int i: matrizFeromona)
    for (int j: matrizFeromona)
        matrizFeromona[i][j] = valorFeromonaDefecto

//colocamos aleatoriamente cada hormiga
for (int i = 0; i < numHormigas; i++) {
    int posicionAleatoria = Math.random()*100)% numPosiciones;
    hormigas[i].setPosicionOrigen(posicionAleatoria);
}

iteraciones = 0;
//cada iteración es un posible camino completo para cada hormiga
do {
    //Calculamos trayectos completos: que pasen por todas las posiciones
    for (int numPosVisit = 0; numPosVisit < numPosiciones-1; numPosVisit++) {
        //Para cada hormiga un trayecto completo
        for (int i = 0; i < numHormigas; i++) {
            int posActual = hormigas[i].getPosicion();
            //elegimos la proxima ciudad a visitar
            int posSiguiente = hormigas[i].proximaPosicion(matrizCostes,
                                                            matrizFerormona,
                                                            factorExplotacion);
            //la hormiga se mueve a esa posición con un cierto coste
            hormigas[i].setPosicion(posSiguiente,matrizCostes[posActual][posSiguiente]);
            //incremntamos el valor de feromona en ese nuevo movimiento
            matrizFerormona[posActual][posSiguiente]=
                ((1 - factorEvaporacionLocal) * matrizFerormona[posActual][posSiguiente]) +
                (factorEvaporacionLocal * valorFeromonaDefecto);
        }
    }

    //vemos cual es el trayecto de menor coste en esta iteración de todas las hormigas
    int mejorHormiga = 0;
    for (int i = 0; i < numHormigas; i++) {
        if ( hormigas[i].getCosteTrayecto() <  hormigas[mejorHormiga].getCosteTrayecto()) {
          mejorHormiga = i;
        }
    }

    //vemos si es mejor que el trayecto mejor que se había conseguido
    trayectoMejorIteracion = hormigas[mejorHormiga].getPosicionesVisitadas();
    costeMejorTrayectoIteracion = hormigas[mejorHormiga].getCosteTrayecto();
    if (costeMejorTrayectoIteracion < costeTrayectoMejor){
        costeTrayectoMejor=costeMejorTrayectoIteracion;
        for (int i = 0; i < trayectoMejorIteracion.length; i++){
            trayectoMejor[i] = trayectoMejorIteracion[i];
        }
    }
```

```
    //reforzamos el mejor trayecto obtenido con ferormonas y disminuimos el resto
    for (int i = 0; i < numPosiciones;  i++) {
        for (int j=0; j < numPosiciones; j++){
                if (!estaEnTrayectoMejor(i,j,trayectoMejor)){
                    matrizFeromona[i][j]=(1-factorEvaporacionGlobal)*matrizFeromona[i][j];
                }else{
                    matrizFeromona[i][j]=(1-
factorEvaporacionGlobal)*matrizFeromona[i][j]+(factorEvaporacionGlobal*(costeTrayectoMej
or));
        }
    }

    //borramos los trayectos realizados por las hormigas y su coste
    for (int i = 0; i < numHormigas; i++) {
        hormigas[i].reiniciarTrayecto();
    }

    iteraciones++;

    //podemos ir aumentando la probabilidade realizar explotación (para elegir siguiente ciudad)
    if (incremenarFactorExplotacion)  actualizarFactorExplotacion(iteracionesMax);

} while ((iteraciones < iteracionesMax) && (costeTrayectoMejor > costeTrayectoDeseado));
```

Figura 71. *Algoritmo principal del ACO*

4.3 Arquitectura de la aplicación

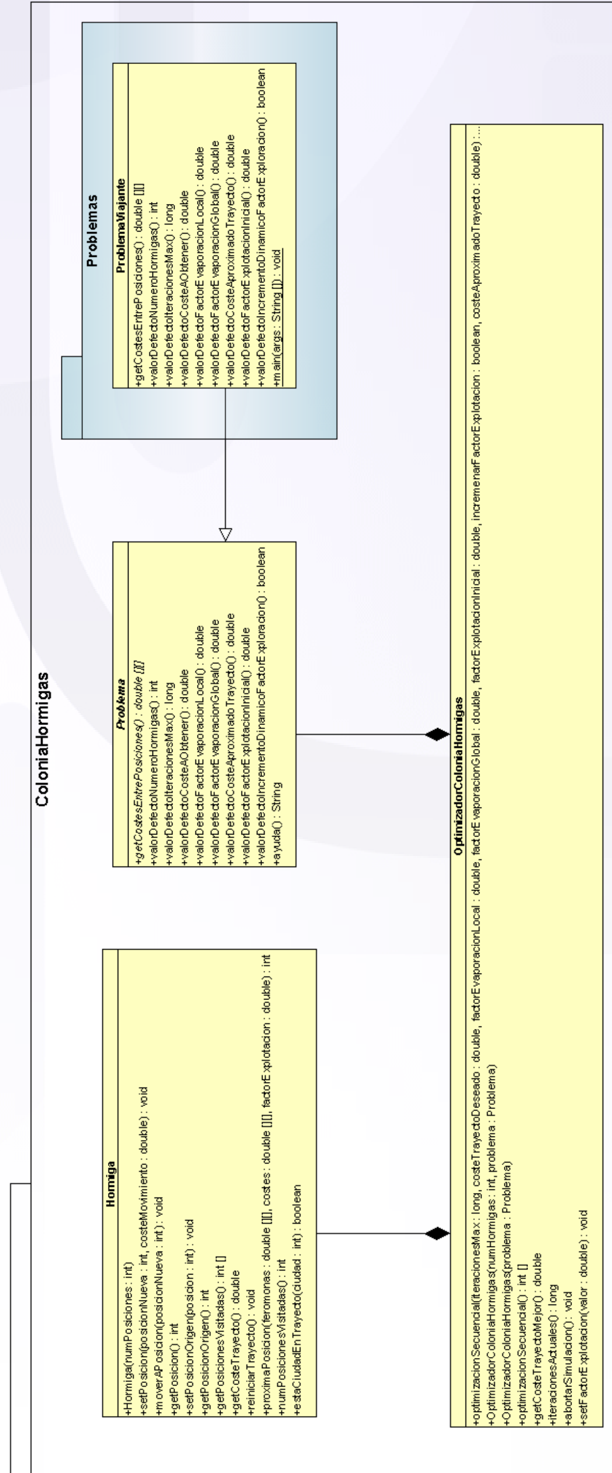


Figura 72. *Diagrama UML del Optimizador por Colonia de Hormigas*

4.4 Particularidades de la implementación

El algoritmo que se ha implementado es el llamado secuencial, es decir, para cada iteración se realiza un camino completo de cada hormiga artificial, es en ese momento cuando se ve cual de ellas ha realizado el mejor camino posible y se refuerza el valor de feromona en ese camino. Un algoritmo paralelo, ejecutaría asíncronamente cada hormiga y se actualizaría en todo momento el valor de feromona al moverse la hormiga (en el secuencial también se produce esta actualización) pero no se produciría un reforzamiento por feromona del mejor camino obtenido hasta el momento.

El algoritmo paralelo es el mas parecido a lo que ocurre en la naturaleza, y en futuras versiones seguramente se implemente.

4.4.1 Parámetros Estáticos

Los parámetros iniciales que necesita el algoritmo y que no podrán ser modificados durante su ejecución son los que siguen a continuación. Por defecto cada problema puede dar un valor por defecto a cada uno de estos parámetros, para no tener que ajustarlos nuevamente cada vez que se selecciona el problema.

- **Número de hormigas**: Número de hormigas artificiales que participan en el algoritmo de optimización.
- **Coste aproximado inicial**: Es una constante que usa el algoritmo que es una “burda” estimación del coste la solución inicial del algoritmo.
- **Factor Evaporación Local**: Indica la rapidez con la que se evapora la feromona que va dejando cada hormiga al moverse en cada nuevo movimiento.
- **Factor Evaporación Global**: Indica la rapidez con la que se evapora la feromona que se deja de reforzamiento al mejor camino encontrado hasta el momento.

- **Coste minimizado**: Coste mínimo del trayecto que se desea obtener como solución.

- **Iteraciones Máximas**: Iteraciones máximas del algoritmo antes de que se pare. Por cada iteración se produce un trayecto completo de cada hormiga.

4.4.2 Parámetros Dinámicos

Estos parámetros pueden modificarse en tiempo real durante la simulación, con el objeto de intentar salir de mínimos locales o acelerar la convergencia hacia una solución según la evolución del algoritmo.

- **Factor Explotación Inicial**: Es la probabilidad inicial de realizar explotación frente a exploración. Conviene que inicialmente este valor sea pequeño para luego irlo aumentando.

- **Factor explotación dinámico**: Sirve para indicar al algoritmo si se le permite modificar dinámicamente el factor de explotación para intentar aproximarse cada vez mas a la solución. Normalmente este factor debería ir aumentando, salvo que se detecte que se está en un mínimo local y se vuelve a rebajar para permitir mayor grado de exploración.

4.5 Como codificar problemas

En esta sección vamos a explicar como implementar la codificación de un problema usando colonias de hormigas en nuestra aplicación, que básicamente consiste en la implementación de los métodos abstractos y la sobrescritura opcional de los métodos que no son abstractos de las dos clase abstractas: Problema..

La codificación del problema se implementa mediante matriz de costes que determina el coste de moverse entre las posibles distintas posiciones. Una solución para un problema consiste en un trayecto que pase por todo las posibles posiciones o estados. De tal manera que para codificar un problema, hay que saberlo adaptar a una matriz de costes.

3.5.1.1. MÉTODOS DE IMPLEMENTACIÓN NECESARIA

Para la codificación del problema es necesario implementar el método declarados como abstracto en la clase Problema, que es:

```
public abstract double[][] getCostesEntrePosiciones();
```

Figura 73. *Métodos que deben ser implementados necesariamente en cada problema*

3.5.1.2 MÉTODOS DE IMPLEMENTACIÓN OPCIONAL

Los métodos de implementación opcional tienen una implementación por defecto dada en la clase Problema, que serán los parámetros por defecto con los que se ejecutaría el optimizador, si bien, suele ser importante darle una implementación adaptada el problema en concreto. Son los siguientes métodos:

```
public int valorDefectoNumeroHormigas()
public long valorDefectoIteracionesMax()
public double valorDefectoCosteAObtener()
public double valorDefectoFactorEvaporacionLocal()
public double valorDefectoFactorEvaporacionGlobal()
public double valorDefectoCosteAproximadoTrayecto()
public double valorDefectoFactorExplotacionInicial()
public boolean valorDefectoIncrementoDinamicoFactorExploracion()
public String ayuda()
```

Figura 74. *Métodos que pueden ser sobrescritos en cada nuevo problema*

4.6 Modo funcionamiento

Existen principalmente dos maneras de usar la aplicación: mediante la interfaz gráfica, o usándose directamente desde otro programa externo. La ventaja de usar la interfaz de usuario, es poder ver directamente como evoluciona el resultado gráficamente. A veces, no interesa hacer uso de la interfaz gráfica, y solamente nos interesa llamar al método que ejecuta la simulación sin mostrar interfaz gráfica. Vamos a presentar las dos maneras de usar la aplicación.

6.6.1 La interfaz gráfica de Usuario

6.6.1.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

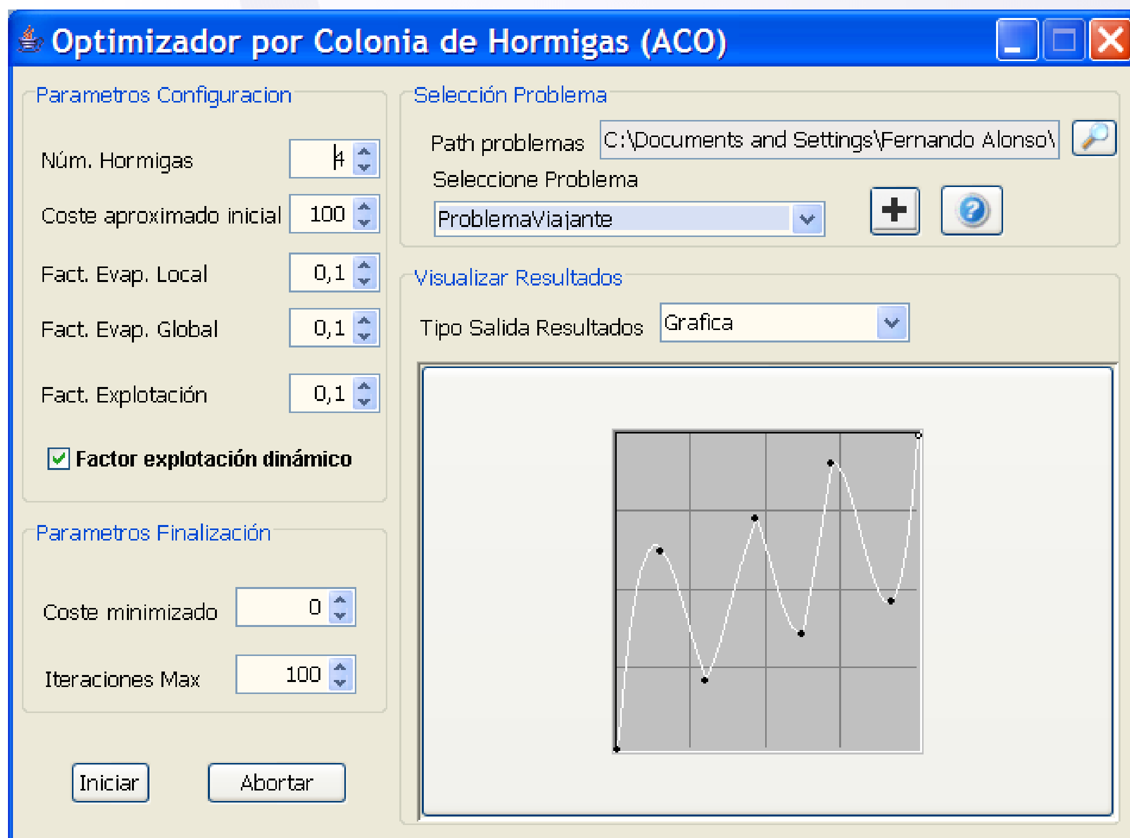


Figura 75. *GUI inicial*

Esta es la vista que presenta la interfaz gráfica de usuario de la herramienta proporcionada para la optimización mediante Temple Simulado. Esta dividida en cuatro panes principales: parámetros de la simulación, parámetros de finalización de la simulación, selección del problema y panel de visualización de resultados.

En el panel llamado "**Parámetros**" , situado arriba a la izquierda se puede seleccionar el valor que tendrán los parámetros del algoritmo, tales como:

- Número de hormigas que se encargarán de recorrer las posibles soluciones.
- Coste aproximado del trayecto inicial (es un parámetro que es necesario aplicar al algoritmo, y que se puede obtener mediante una heurística aplicada previamente, si bien no es muy determinante en la obtención de soluciones al problema, conviene fijar un coste inicial alto)
- Factor de evaporación local que indica como de rápido se evapora la feromona que deja cada hormiga en cada cambio de posición o nodo del trayecto,
- Factor de evaporación global indica como de rápido se evapora la feromona que refuerza el mejor trayecto obtenido hasta el momento
- Factor de explotación sirve para tener un balance entre explotación y exploración, si se elige un factor de explotación dinámico, este irá creciendo hasta alcanzar su valor máximo cuando se lleguen a las iteraciones máximas marcadas.

En el panel "**Parámetros de Finalización**" se debe elegir el coste a obtener, es decir el coste del mejor trayecto que se desea obtener, a partir del cual, una vez que obtengamos un camino de ese coste dejamos de seguir buscando caminos de menor coste, y el número de iteraciones máximas del algoritmo.

El panel "**Selección del Problema**" se encarga de cargar los programas compilados previamente o crear directamente problemas nuevos para ser compilados y ejecutados desde la propia interfaz o de manera externa. La ruta donde de donde se cargan los problemas y en donde se crean puede ser introducida en el recuadro textual que tiene a su derecha una lupa. Si la ruta no se cambia, cargará los problemas que vienen implementados con el proyecto, si se indica otra ruta, el cargador de clases intentará cargar los problemas compilados que se encuentren en dicha ruta, pinchando en la lupa.

Para crear nuevos problemas, en la ruta indicada, se pedirá un nombre del problema, y se analizará dicha ruta; si la ruta era la original, se creará el problema dentro del propio proyecto, y se mostrará una ventana desde la que se puede trabajar con el código y compilarlo. Si el código queda compilado correctamente se podrá directamente usar desde la interfaz gráfica (sin tener que cerrar y volver a lanzar la aplicación). Posteriormente el código puede ser otra vez editado y compilado usando el *Make.bat* del propio proyecto. Si por el contrario, se ha creado el problema en una ruta distinta de la de por defecto dentro del propio proyecto, se generarán los ficheros relativos al problema en la ruta indicada, así como un archivo *Make.bat* que permite compilar dichos archivos. Esta segunda opción, es importante para crear problemas que no serán incluidos posteriormente en el proyecto, y que simplemente quieren usar la librería HerramientasIA.jar para valerse de las herramientas que esta librería ofrece.

Para obtener ayuda sobre un problema creado y compilado, se podrá pinchar en el botón de ayuda, que mostrará la información relativa al problema que haya sido introducida por el programador.

En el panel "**Visualizar Resultados**" se puede elegir la manera en que los resultados del simulador serán presentados al usuario. Si se elige una presentación por "*terminal*", y si los resultados se quieren mostrar en la interfaz gráfica se podrá elegir "Grafica" que hará la gráfica en tiempo real de la evolución de la temperatura del sistema.

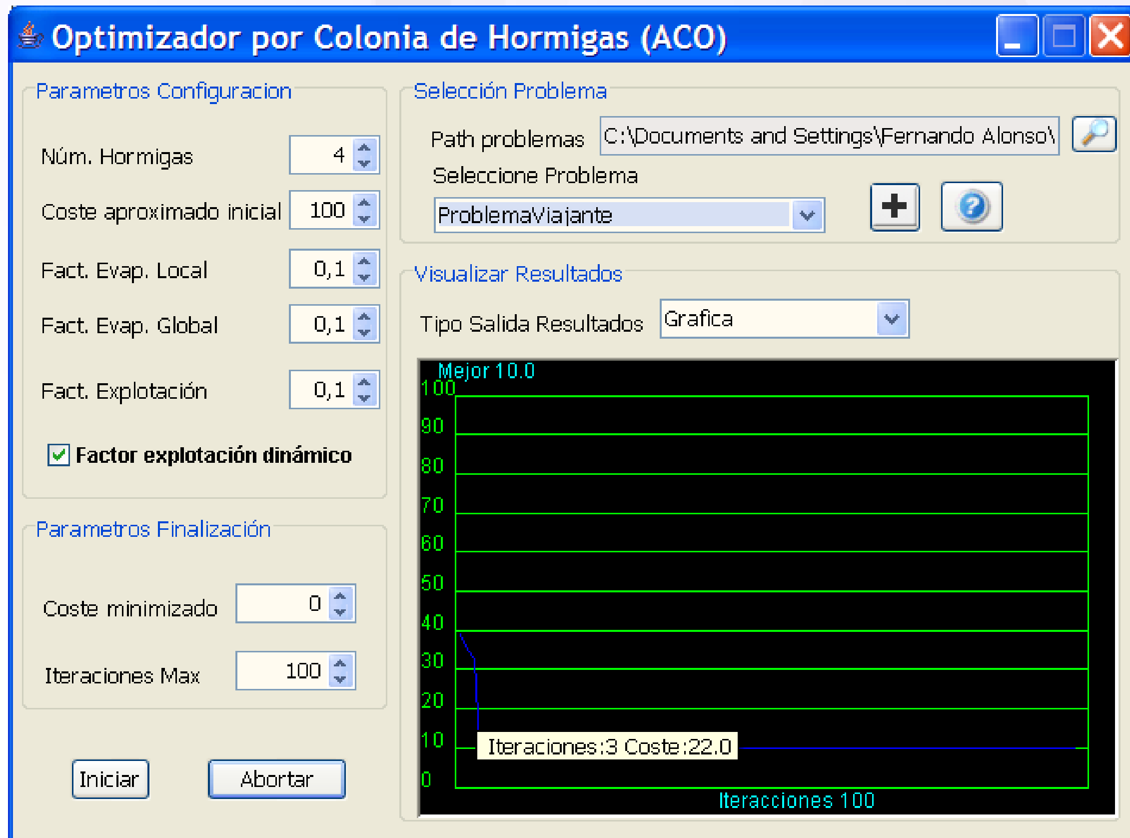


Figura 76. *GUI con información tipo "Tool Tip"*

Se ha añadido información textual a la mayoría de los elementos que la forman. Para obtener información dicha información basta con mantener el ratón situado unos segundos encima, momento en el cual aparecerá el llamado "tool tip". Incluso se puede obtener información sobre algún punto de la gráfica manteniendo el ratón encima del punto deseado y aparecerá las coordenadas relativas a ese punto de la gráfica.

Para conseguir un aspecto más atractivo, se ha añadido además la característica común a todas las herramientas del proyecto de poder cambiar la foto de fondo (el skin o apariencia). Para ello basta con pinchar con el botón derecho sobre cualquier parte de la aplicación y pinchar en cambiar foto de fondo sobre el menú emergente que aparece, posteriormente se selecciona la foto deseada y quedará establecida como fondo. Por defecto en la carpeta "*skin*" del proyecto se adjuntan algunos fondos.

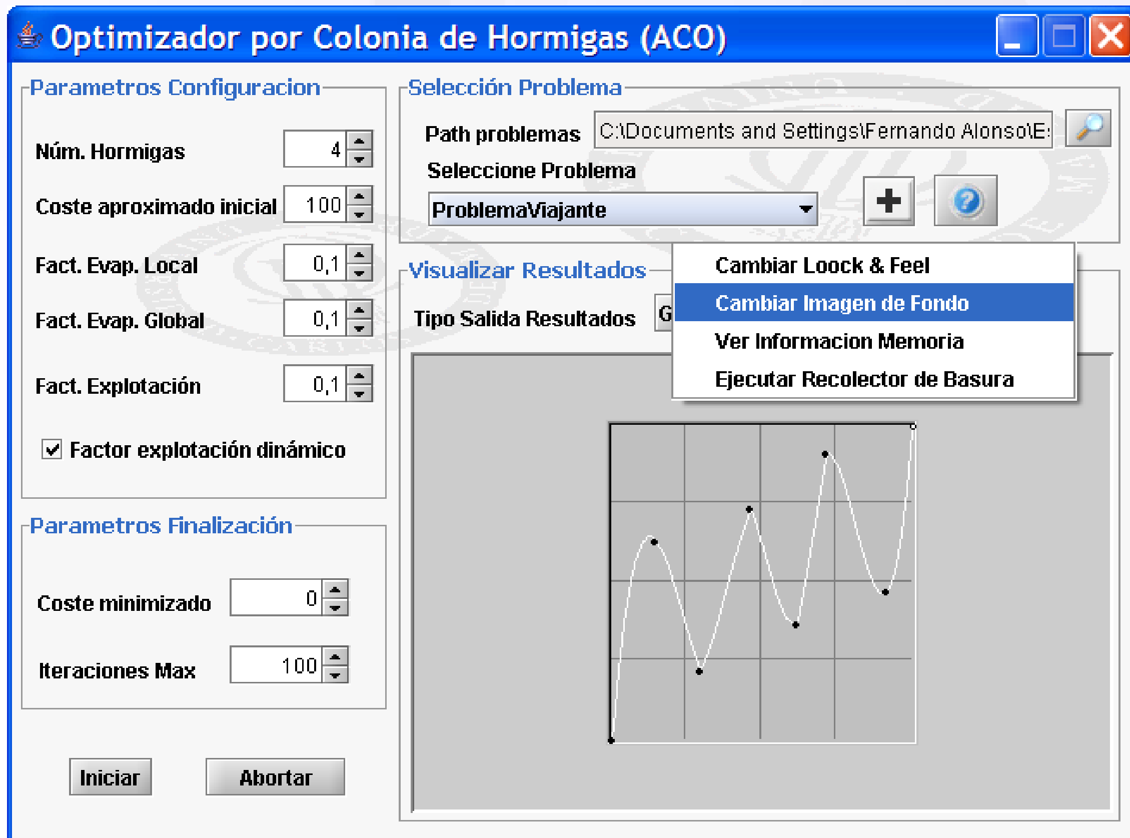


Figura 77. *GUI con nuevo Look & Feel*

También se añadido la posibilidad de cambiar el "look&feel" de la ventana, pudiendo intercambiarse entre varios "Look & Feel", como por ejemplo: java, metal, windows, borland…. Nuevamente esta acción se puede realizar pinchando en el botón derecho del ratón y seleccionando en "cambiar loock&feel", que irá alternando entre los disponibles.

6.6.1.2 EJECUTAR UN PROBLEMA DESDE LA INTERFAZ DE USUARIO

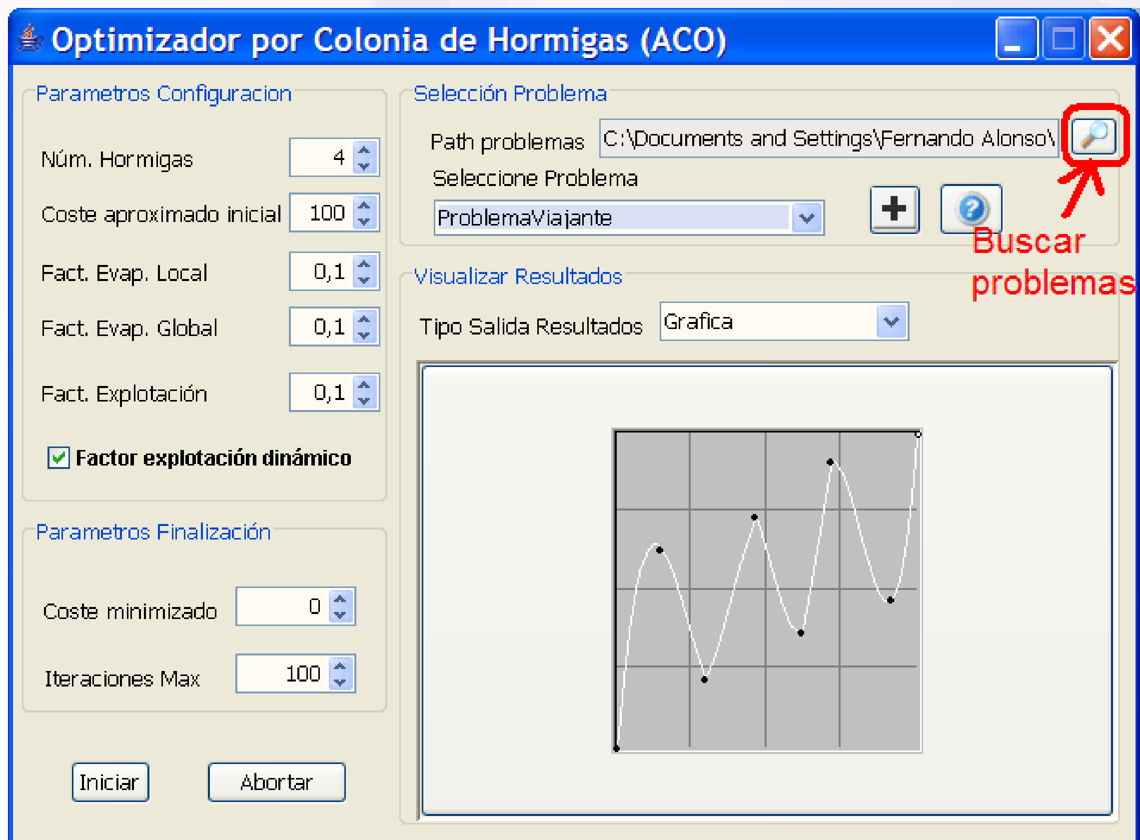


Figura 78. *Buscar problemas en la ruta indicada*

Para buscar los problemas implementados en el propio proyecto o en problemas externos al mismo, basta con indicar la ruta del problema y pinchar en la lupa. De esta manera se cargarán los problemas compatibles con el simulador. Si el problema es externo a la aplicación (se ha creado fuera de la ruta por defecto) se cargará mediante el cargador de clases de Java.

Notar que algunos problemas no podrán ser cargados y así se indicará por terminal, bien porque no están compilados, o no lo están adecuadamente, o simplemente porque no son válidos para ser ejecutados desde la propia interfaz gráfica, puesto que necesiten argumentos que no puedan ser pedidos desde la misma (como objetos). Es decir, sólo podrán ser cargados en la interfaz gráfica aquellos problemas en cuyo constructor no se reciba ningún argumento (si pudiéndolos pedir posteriormente dentro del propio constructor). Los problemas que necesitan recibir argumentos en el constructor serán adecuados para ser lanzados sin la interfaz gráfica, como se puede ver en el punto "6.5.2 Ejecutando el simulador sin la interfaz gráfica".

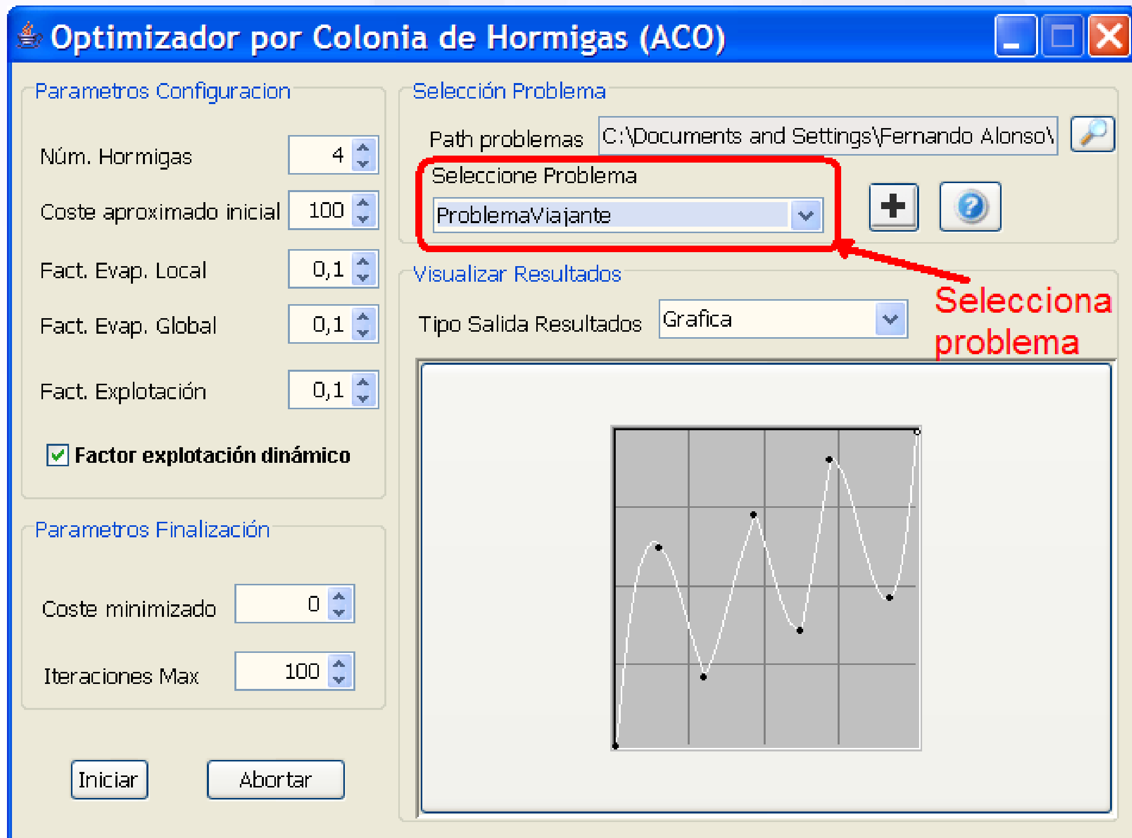


Figura 79. *Selecciona el problema que se quiere solucionar*

De entre los problemas cargados en la ruta indicada se podrá seleccionar aquel que queramos resolver. Una vez seleccionado se establecerán los valores por defecto del problema (si se ha indicado por el programador y sino los valores por defecto de cualquier problema).

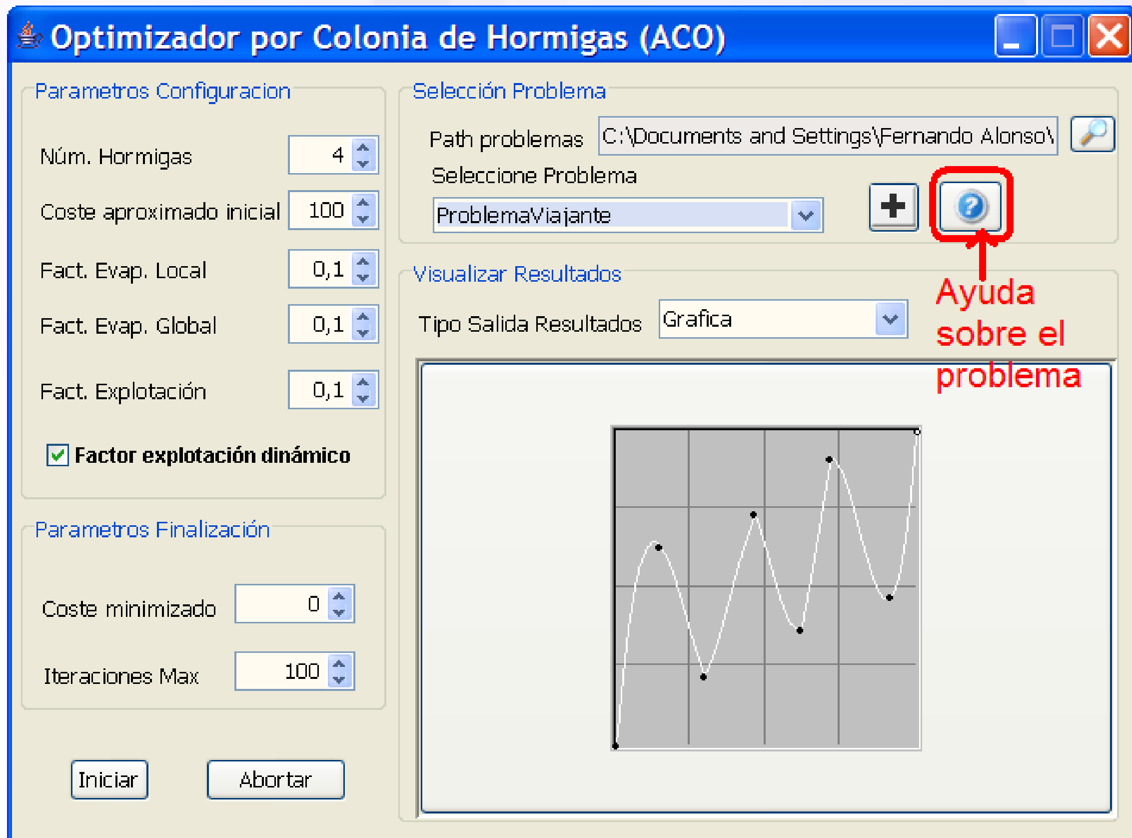


Figura 80. *Obtener ayuda sobre el problema seleccionado*

Si se pincha en el botón de ayuda, se podrá obtener ayuda sobre el propio problema y sobre los valores por defecto establecidos para ese problema. Esta ayuda la introduce el programador del problema, mostrándose la de por defecto sino se ha introducido ninguna nueva.

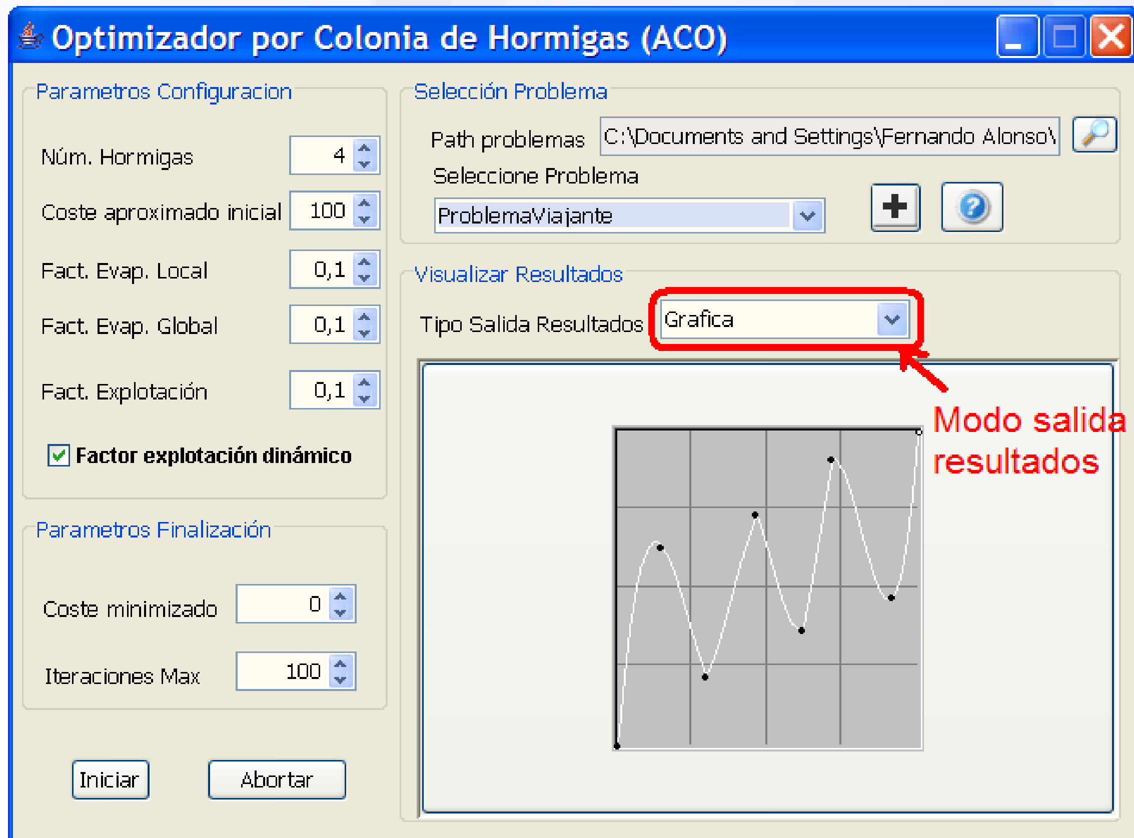


Figura 81. *Visualizar los resultados*

Los resultados y la evolución del algoritmo se pueden mostrar de distintas maneras. La manera usual de mostrar los resultados será de manera gráfica (es decir en la interfaz gráfica), pudiéndose escoger la opción de generar la gráfica que relacione la puntuación obtenida en cada generación. También podrán mostrarse la evolución y los resultados por consola. Estas opciones de visualización pueden cambiarse incluso durante la propia simulación.

Hacer notar, que cuanta mas salida de resultados se realicen mayor tiempo se invertirá en la simulación.

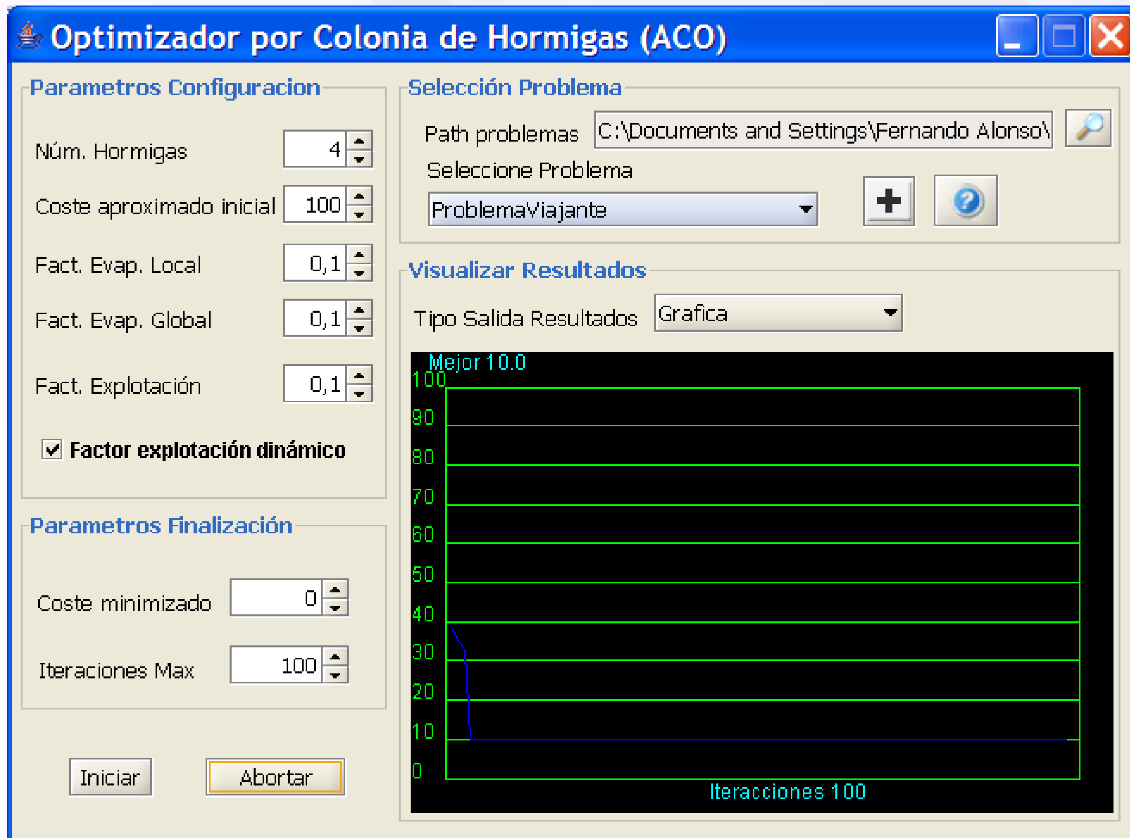


Figura 82. *Ejecutando simulación*

Al iniciar la simulación pinchando en el botón de iniciar se e puede ver como se va creando dinámicamente la gráfica que relaciona las generaciones con la puntuación obtenida. Si se mantiene el ratón encima de cualquier punto de la gráfica se pueden ver sus coordenadas exactas. En la gráfica se representa en el eje horizontal las iteraciones del algoritmo ya realizadas, mientras que en el eje vertical se representa la temperatura global del sistema.

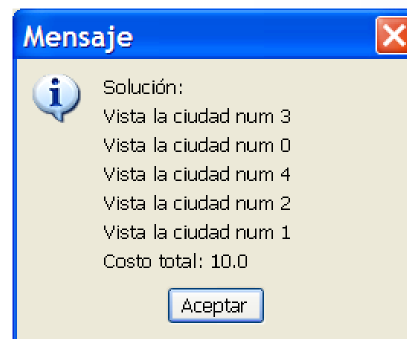


Figura 83. *Resultados de la simulación*

Al finalizar la simulación se muestran los resultados obtenidos directamente de la interpretación que se da del estado solución.

6.6.1.3 CREAR UN NUEVO PROBLEMA DESDE LA INTERFAZ DE USUARIO

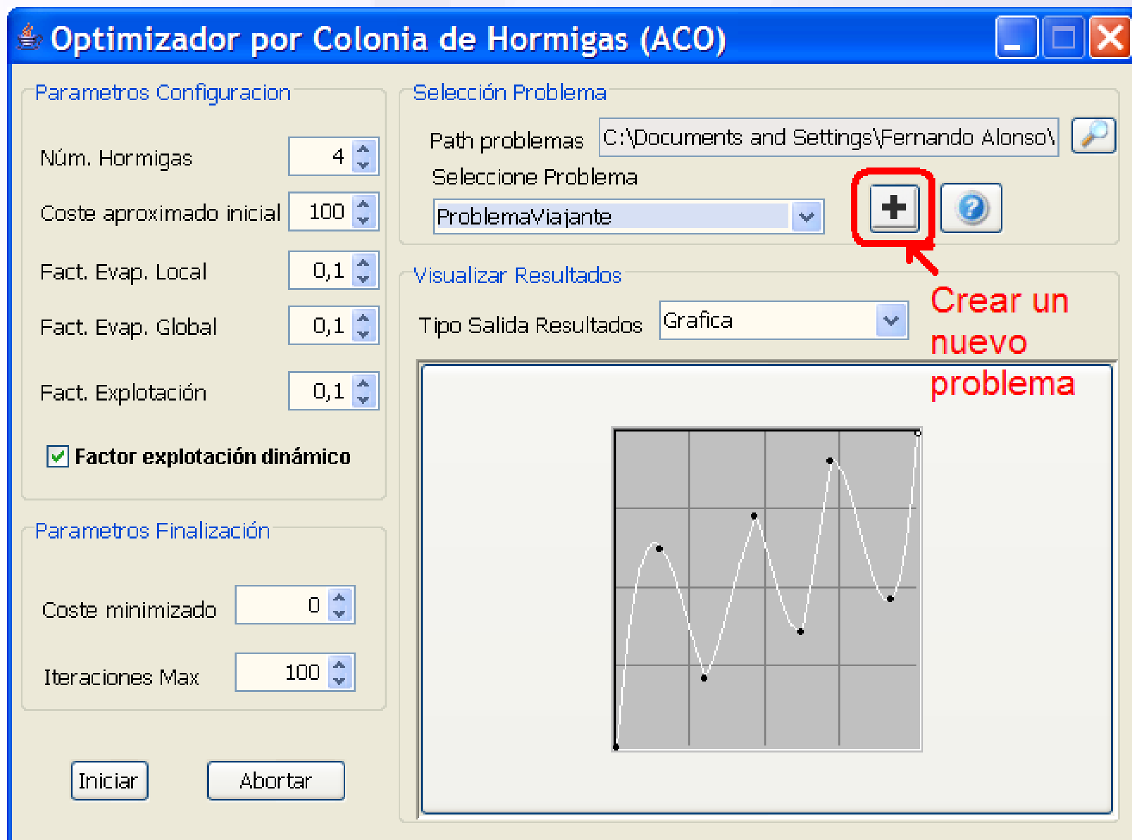


Figura 84. *Crear nuevo problema*

Si se desea crear un problema que estará dentro del propio proyecto (cada vez que se haga un make será también compilado) se debe dejar la ruta especificada en "*Path problemas*" sin modificar ya que apunta directamente a la carpeta con los problemas de Temple Simulado del proyecto. Vamos a analizar ambos casos:

- CREANDO EL PROBLEMA DENTRO DEL PROPIO PROYECTO

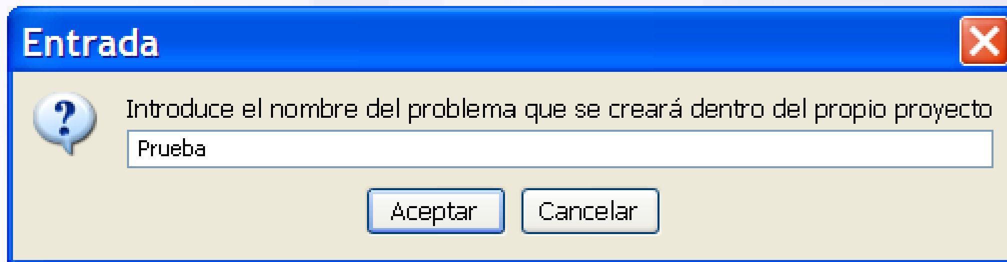


Figura 85. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a la clase que se creará.

ProblemaPrueba.java

Compilar - +

```
import ColoniaHormigas.*;
 /**
  * IMplementacion del problema
  */
 public class ProblemaPrueba extends Problema{

   /**
    * Debe definirse para cada problema en concreto la matriz de costes
    * @return int[] Matriz de costes definiada para el problema concreto
    */
   public double[][] getCostesEntrePosiciones(){
       return null;
   }


   //_____________Métodos que pueden ser sobrescritos_______________________
   public int valorDefectoNumeroHormigas(){ return 4; }
   public long valorDefectoIteracionesMax(){ return  50; }
   public double valorDefectoCosteAObtener(){ return 0; }
   public double valorDefectoFactorEvaporacionLocal(){ return 0.1; }
   public double valorDefectoFactorEvaporacionGlobal(){ return 0.1; }
   public double valorDefectoCosteAproximadoTrayecto(){ return 100; }
   public double valorDefectoFactorExplotacionInicial(){ return 0.1; }
   public boolean valorDefectoIncrementoDinamicoFactorExploracion(){ return true; }



  /** Probando el problema */
   public static void main(String args[]){
     ProblemaPrueba problemaPrueba = new ProblemaPrueba();
     OptimizadorColoniaHormigas aco = new OptimizadorColoniaHormigas(problemaPrueba);
     int[] trayectoSolucion = aco.optimizacionSecuencial();
     for (int i = 0; i < trayectoSolucion.length; i++){
       System.out.println("Posicion visitada: " + trayectoSolucion[i]);
     }
   }

 }
```

Figura 86. *Implementación del problema*

Una vez compilada también la clase problema que contiene la función de evaluación se podrá seleccionar el problema que se acaba de crear para poder ser ejecutado, tal y como se dice en el apartado "6.6.2 Ejecutar un problema desde la interfaz de usuario", sin tener que cerrar y volver a lanzar la aplicación (salvo que se haya ejecutado la aplicación directamente desde el jar, que entonces debería recompilarse todo el proyecto con el *make* y volver a lanzarlo).

CREANDO EL PROBLEMA FUERA DEL PROYECTO

Cuando se cambia el path por defecto donde se crearán los problemas, se crean en una ruta distinta de la asignada en el proyecto, por lo que se generarán en la ruta indicada los ficheros esqueleto del problema para que puedan ser editados y compilados con posterioridad.

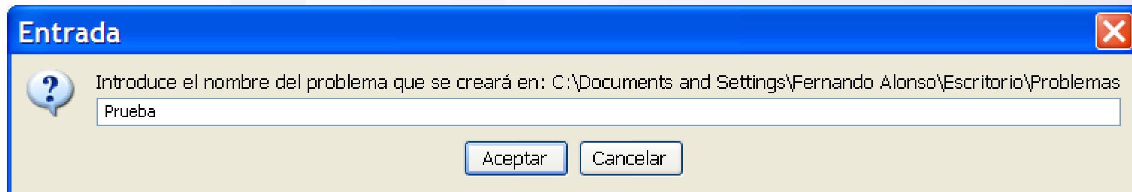


Figura 87. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a la clase que se creará.

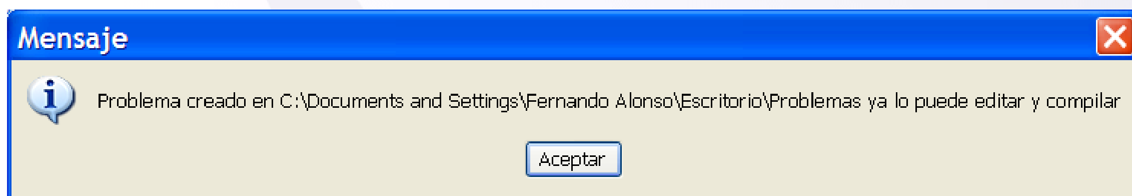


Figura 88. *Mensaje informativo de que se han guardado los ficheros*

Si la ruta indicada es correcta se creara el problema en la ruta indicada y se mostrará el mensaje de la figura anterior. Se habrán guardado los archivos java en la ruta indicada pero no habrán sido compilados. Para compilarlos se deberá ejecutar el archivo "*Make.bat*" que se genera conjuntamente con los archivos, y para los que únicamente usa para compilarlos es la librería "*HerramientasIA.jar*".

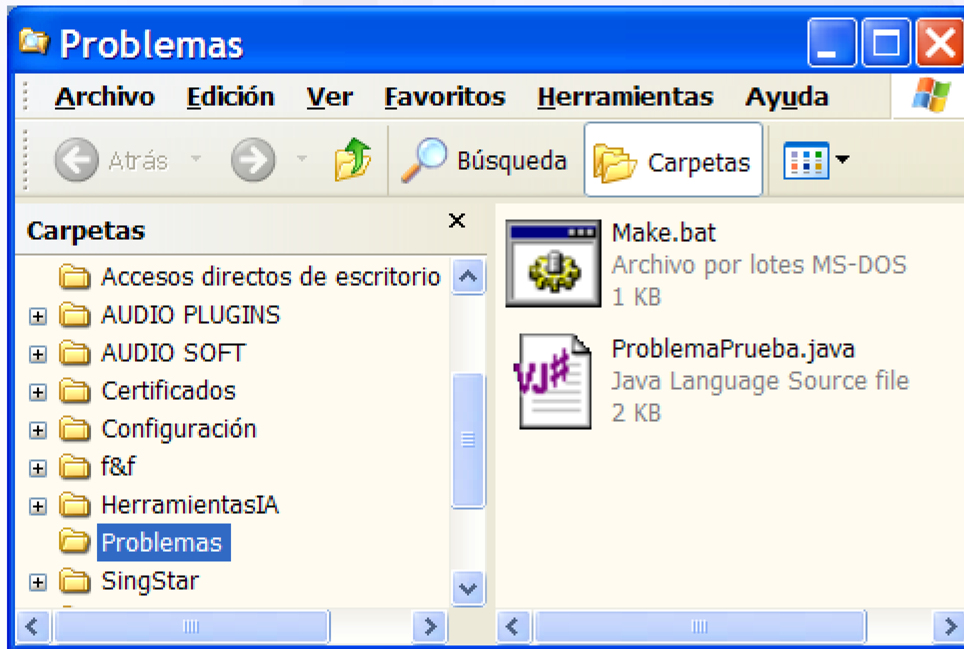


Figura 89. *Archivos fuentes con el esqueleto del problema creado*

Una vez compilados ejecutando “*Make.bat*”, puede ser el problema cargado y ejecutado usando la interfaz gráfica “6.6.1.2 Ejecutar un problema desde la interfaz de usuario” o bien “6.6.2 Ejecutando el simulador sin la interfaz gráfica”

6.6.2 Ejecutando el simulador sin la interfaz gráfica

Un problema creado valiéndonos de la interfaz gráfica de usuario, como se dice en la sección "6.6.1.3 Crear un nuevo problema desde la interfaz de usuario" o sin valernos de ella, puede ser ejecutado sin necesidad de ejecutar el main de la aplicación y por tanto de la interfaz gráfica. Para ello se deberá desde el programa que se quiera usar escribir las siguientes líneas:

```
//importamos el paquete
import ColoniaHormigas.*;

//instanciamos el problema
ProblemaViajante problemaViajante = new ProblemaViajante();

//instanciamos la clase Temple Simualdo
OptimizadorColoniaHormigas aco = new OptimizadorColoniaHormigas(problemaViajante);

//lanzamos el algoritmo con los valores por defecto
int[] trayectoSolucion = aco.optimizacionSecuencial();

//mostramos la solución encontrada por pantalla

for (int i = 0; i < trayectoSolucion.length; i++){
  System.out.println("Posicion visitada: " + trayectoSolucion[i]);
}
```

Figura 90. *Ejecuta problema con los parámetros por defecto*

Lanza el optimizador de manera síncrona y con los valores por defecto dados en la implementación del problema (sino se han sobrescrito los métodos que dan los valores por defecto se quedan los que tienen todos los problemas). Notar que en este caso, se pueden ejecutar problemas cuyo constructor reciba argumentos, como por ejemplo objetos, aunque en el ejemplo no se ha pasado ningún argumento; mientras que ejecutando el problema mediante interfaz gráfica el problema no podía recibir nada por argumentos, puesto que se instancia el problema desde la propia interfaz gráfica con el constructor por defecto sin argumentos.

```
//importamos el paquete
import ColoniaHormigas.*;

//instanciamos el problema
ProblemaViajante problemaViajante = new ProblemaViajante();

 //instanciamos la clase Temple Simualdo
OptimizadorColoniaHormigas aco = new OptimizadorColoniaHormigas(problemaViajante);

 //lanzamos el algoritmo
long iteracionesMax=100;
double costeTrayectoDeseado=15;
dobule factorEvaporacionLocal=0.1;
double factorEvaporacionGlobal=0.1;
double factorExplotacionInicial=0.1;
bolean incrementarFactorExplotacion=true;
double costeAproximadoInicialTrayecto=30;
boolean mostrarInformacion=true;

 int[] trayectoSolucion = aco.optimizacionSecuencial(iteracionesMax,
                                                     costeTrayectoDeseado,
                                                     factorEvaporacionLocal,
                                                     factorEvaporacionGlobal,
                                                     factorExplotacionInicial,
                                                     incrementarFactorExplotacion,
                                                     costeProximadoInicialTrayecto,
                                                     mostrarInformacion);

 //mostramos la solución encontrada por pantalla
 for (int i = 0; i < trayectoSolucion.length; i++){
   System.out.println("Posicion visitada: " + trayectoSolucion[i]);
```

Figura 91. *Ejecuta problema sin usar los valores por defecto*

Esta segunda manera de lanzar el algoritmo es lanzándolo con los valores concretos que queramos darle, sin tener en cuanta los valores por defecto codificados.

5. Simulador de Temple Simulado

5.1 Introducción

Es una meta-heurística para problemas de optimización globa (de minimización concretamente)l, es decir, encontrar una buena aproximación al óptimo global de una función en un espacio de búsqueda grande. En inglés se conoce como Simulated annealing (SA). Es un tipo de algoritmo de ascensión de colinas estocástica, es decir, de elección de un sucesor de entre todos los posibles según una distribución de probabilidad, pudiendo ser el sucesor peor que el actual.

El nombre e inspiración viene del proceso de templado (annealing) en metalurgia, una técnica que incluye calentar y luego enfriar controladamente un material para aumentar el tamaño de sus cristales y reducir sus defectos. El calor causa que los átomos se salgan de sus posiciones iniciales(un mínimo local de energía) y se muevan aleatoriamente; el enfriamiento lento les da mayores probabilidades de encontrar configuraciones con menor energía que la inicial. Por lo tanto el algoritmo trata de ir transitando de estados partiendo una temperatura inicial alta hasta llegar a una temperatura final baja.

En cada paso, el simulated annealing considera algunos vecinos del estado actual y probabilísticamente decide si cambiar el sistema a un estado vecino o quedarse en el estado actual. Las probabilidades se escogen para que el sistema tienda finalmente a estados de menor energía. Típicamente este paso se repite hasta que se alcanza un estado suficientemente bueno para la aplicación o hasta que se cumpla cierto tiempo número de iteraciones y etapas. Otra cualidad del simulated annealing es que la temperatura va disminuyendo gradualmente conforme avanza la simulación. Hay muchas maneras de disminuir la temperatura, siendo la más usual la exponencial, es decir el factor por el que se multiplica la temperatura suele estar entre 0 y 1.

Para cada problema hay que definir por un lado cada estado, y a cuales puede transitar (son vecinos). Al igual que la función que evalúa la energía de cada estado.

5.2 Pseudo-código del algoritmo principal

```
 temperaturaActual = temperaturaInicial;
estadoActual = estadoInicial;
temperaturaEstadoActual = problema.funcionEvaluacionEnergia(estadoActual);

/*mientras la temperatura actual sigua siendo mayor que la final a obtener tenemos
  que seguir minimizándola: generamos etapas*/
while ((temperaturaActual >= temperaturaFinal) && (numEtapa<etapasMax)){
   //iteramos en cada ETAPA cambiando de estados
   while ((aceptacionesRealizadas < aceptacionesMaxEtapa) &&
             (iteracionesRealizadas < iteracionesMaxEtapa)) {
     //obtenemos un estado vecino al actual
     estadoNuevo = estadoActual.obtenerUnSiguiente();
     //evaluamos la temperatura (energía) del nuevo estado
     temperaturaEstadoNuevo=problema.funcionEvaluacionEnergia(estadoNuevo);
     //si el estado nuevo es de menor temperatura pasamos a ese estado
     if (temperaturaEstadoNuevo < temperaturaEstadoActual) {
          estadoActual = estadoNuevo;
           temperaturaEstadoActual = temperaturaEstadoNuevo;
           aceptacionesRealizadas++;
    /*sino es de menor temperatura el nuevo estado, si se da una cierta probabilidad
      también se realiza el cambio (evitar mínimos locales)*/
     }else{
          distribucionProbabilidad=Math.exp((temperaturaEstadoActual-
                                       temperaturaEstadoNuevo) / temperaturaActual);
           if (Math.random() < distribucionProbabilidad) {
               estadoActual = estadoNuevo;
               temperaturaEstadoActual = temperaturaEstadoNuevo;
               aceptacionesRealizadas++;
            }
       }
       iteracionesRealizadas++;
    } //fin etapa
    //si se ha cambiado de etapa por llegar al max de cambios de estado enfriamos
    if (aceptacionesRealizadas == aceptacionesMaxEtapa)
         temperaturaActual *= factorEnfriamiento;
     //si se ha cambiado de etapa por llegar al max de iteraciones permitidas calentamos
    else if (iteracionesRealizadas == iteracionesMaxEtapa)
         temperaturaActual *= factorCalentamiento;
   /*incrementamos el número de iteraciones máximas permitidas por etapa ya que cada
   vez es mas difícil el movimiento a un estado mejor*/
    iteracionesMaxEtapa *= factorIncrementoIteraccionexMaxEtapa;      numEtapa++;
    aceptacionesRealizadas = 0;
    iteracionesRealizadas = 0;
  } //fin del algoritmo
  return estadoActual;
```

Figura 92. *Algoritmo principal del Temple Simulado*

5.3 Arquitectura de la aplicación

TempleSimulado

TempleSimulado
-problema : Problema
-temperaturaActual : double
-estadoActual : Estado
-numEtapa : int
+TempleSimulado(problema : Problema)
+templadoSimulado() : Estado
+templadoSimulado(temperaturaInicial : double, temperaturaFinal : double, etapasMax : long, iteracionesMaxEtapa : long, aceptacionesMaxEtapa : int, factorEnfriamiento : double, factorCalentamiento : double, factorIncrementoIteraccionexMaxEtapa : double, estadoInicial : Estado) : Estado
+temperaturaActual() : double
+numEtapa() : int

Realiza el temple simulado con los valores por defecto

<<interface>>
Estado
+obtenerUnSiguiente() : Estado
+descripcionTextualEstado() : String

Problema
+funcionEvaluacionTemperatura(estado : Estado) : double
+estadoInicial() : Estado
+instanciarProblema(pathProblema : String) : Problema
+valorDefectoTemperaturaInicial() : double
+valorDefectoTemperaturaFinal() : double
+valorDefectoEtapasMax() : long
+valorDefectoIteracionesMaxEtapa() : long
+valorDefectoAceptacionesMaxEtapa() : int
+valorDefectoFactorEnfriamiento() : double
+valorDefectoFactorCalentamiento() : double
+valorDefectoFactorIncrementoIteracionesMaxEtapa() : double
+ayuda() : String

Problemas

Viajante

EstadoViajante
-ordenCiudadesVisitar : int[]
+EstadoViajante(ordenCiudadesVisitar : int [])
+obtenerUnSiguiente() : Estado
+descripcionTextualEstado() : String
+obtenerOrdenCiudadesVisitar() : int []

ProblemaViajante
-distancias : int[][]
+ProblemaViajante(matrizDistancias : int [][])
+estadoInicial() : Estado
+funcionEvaluacionTemperatura(estado : Estado) : double
+main(args : String []) : void
+ayuda() : String

Figura 93. *Diagrama UML del Temple Simulado*

5.4 Particularidades de la implementación

El algoritmo del Temple Simulado básico consiste simplemente en transitar entre los estados del espacio de soluciones posibles intentando disminuir la energía en cada salto a un nuevo estado, y sino fuera posible esto, se cambiaría a estados de energía mayor si se da una cierta probabilidad que sigue una distribución par intentar evitar mínimos locales.

Se parte de una temperatura inicial alta que se intentará ir disminuyendo si se consigue realizar durante cada etapa un número determinado de saltos a estados de menor temperatura (aceptaciones). Si por el contrario, no se logran realizar ese determinado número de saltos a estados de menor energía en cada etapa, la temperatura global del sistema aumenta. Cuando la temperatura tiende al mínimo, la probabilidad de transitar a estados de mayor energía tiende a cero asintóticamente. Así, cada vez el algoritmo acepta menos movimientos que aumenten la energía. El objetivo final es encontrar un estado de la menor temperatura posible, y un indicativo claro de ello es llegar a una temperatura final del sistema lo mas baja posible.

Este algoritmo básico se ha optimizado con una serie de matices que vemos a continuación.

La probabilidad de hacer la transición al nuevo estado (incremento de energía) es una función de la diferencia de energía entre el estado vecino y el estado actual, pues bien, una cualidad importante del método es que la probabilidad de transición es siempre distinta de cero, aún cuando sea positiva, es decir, el sistema puede pasar a un estado de mayor energía (peor solución) que el estado actual. Esta cualidad impide que el sistema se quede atrapado en un óptimo local.

Si la probabilidad de transición es negativa, es decir, la transición disminuye la energía, el movimiento es aceptado con probabilidad P=1.

En el algoritmo se ha implementado un factor de enfriamiento entre 0 y 1, lo que implica una función de enfriamiento exponencial. El factor de calentamiento en cambio debe ser mayo que 1.

El estado inicial del que se parte debe ser especificado al hacer la llamada al algoritmo, típicamente suele ser un estado válido del espacio de soluciones al azar.

La cantidad de iteraciones que estamos dispuestos a realizar para conseguir realizar un número determinado de aceptaciones, es decir, el número de iteraciones que se puede permitir en cada etapa para transitar n veces a estados de menor energía, varía dinámicamente, de tal manera que a medida que va disminuyendo la temperatura global del sistema, el número de iteraciones máximas por etapa va aumentando, ya que es mas difícil cada vez realizar n aceptaciones para el mismo número de iteraciones. Lógicamente el número de iteraciones debe ser siempre mayor o igual que el número de aceptaciones.

5.5 Como codificar problemas

En esta sección vamos a explicar como implementar la codificación de un problema usando Temple Simulado en nuestra aplicación.

5.5.1 ¿Cómo codificar las posibles soluciones al problema?

La codificación del problema se hace mediante los llamados Estados, siendo un estado concreto una posible solución al problema. De tal manera que el conjunto de estados posibles es el conjunto de soluciones posibles para el problema.

El estado se puede codificar con prácticamente total libertad, en este sentido, la libertad de codificación es mucho mayor que con otras técnicas como Algoritmos Genéticos o Enjambres de Partículas. El estado concreto que se cree debe implementar la interfaz “*Estado.java*”, los demás métodos o atributos que se usen quedan a libertad del usuario y del propio problema que se este codificando.

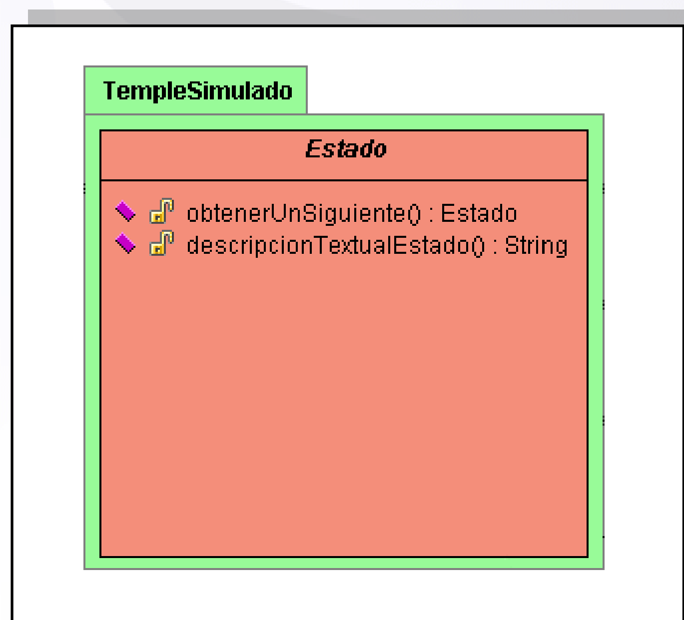


Figura 94. *Interface Estado*

6.5.1.1. MÉTODOS DE IMPLEMENTACIÓN NECESARIA

Para la codificación del problema es necesario al menos implementar el método declarado como abstracto en la clase Problema, que es:

```
public Estado obtenerUnSiguiente();
public String descripcionTextualEstado();
```

Figura 95. *Métodos a implementar en la codificación del estado*

Un ejemplo de implementación de estado, viene dada con el problema de ejemplo que hemos codificado, como es el del problema del viajante. El problema del viajante consiste en visitar todas las ciudades (un número determinado) sin repetir ninguna ciudad en el viaje, de tal manera que el coste total del trayecto sea el mínimo posible. En este ejemplo, cada estado codifica un posible trayecto válido que pueda seguir el viajante.

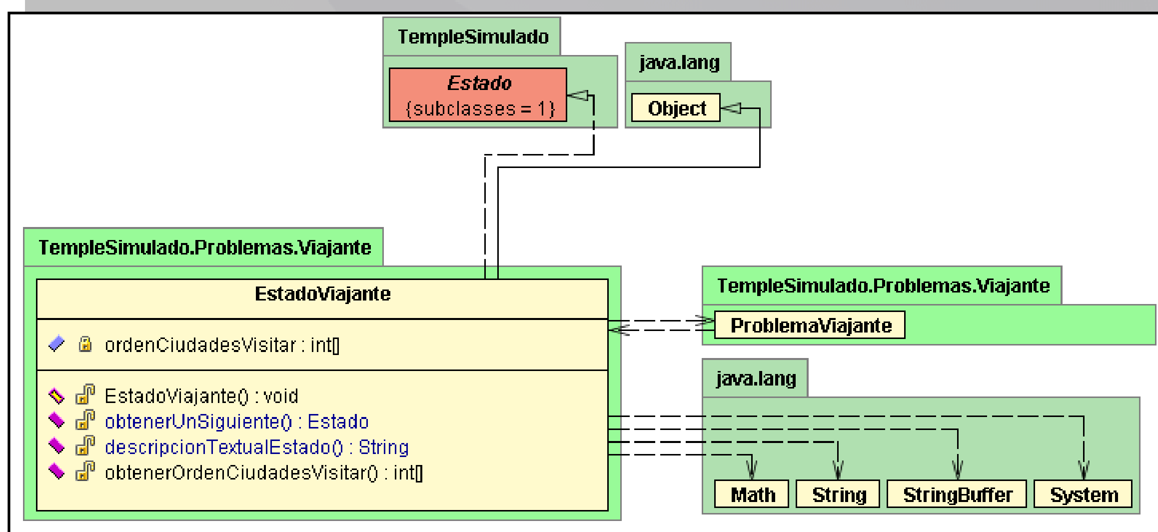


Figura 96. *Codificación problema viajante*

Como se puede ver, la codificación de soluciones posibles a problemas mediante este esquema da bastante libertad al programador.

5.5.2 ¿Cómo evaluar las posibles soluciones?

Por otra parte, hay que ser capaz de evaluar los posibles estados (las posibles soluciones al problema), para ello se debe implementar la función heurística que mide la energía de cada estado. Para ello se debe extender la clase abstracta "*Problema.java*" que deja sin implementación la función de evaluación, y da un valor por defecto a los parámetros que tomará el algoritmo de Temple Simulado que pueden ser variados opcionalmente si se sobrescribe la implementación de dichos métodos.

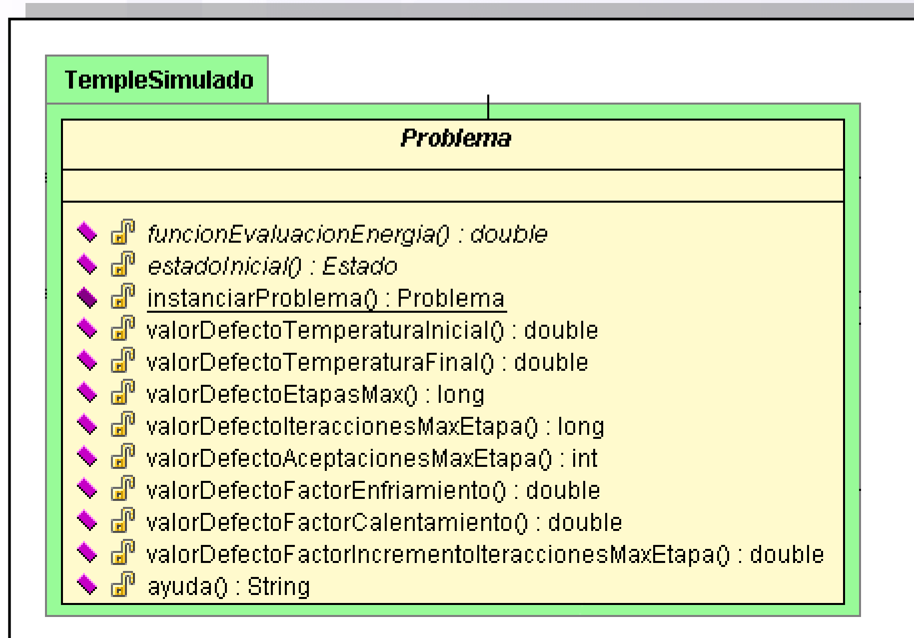


Figura 97. *Clase abstracta Problema*

En el caso del problema del viajante, la función que evalúa la energía de cada estado, y por lo tanto la calidad de la solución, es simplemente el coste total del trayecto que se representa.

6.5.2.1. MÉTODOS DE IMPLEMENTACIÓN NECESARIA

Para la codificación del problema es necesario al menos implementar el método declarado como abstracto en la clase Problema, que es:

```
public abstract double funcionEvaluacionEnergia(Estado estado)
public abstract Estado estadoInicial()
```

Figura 98. *Método que deben ser implementados necesariamente en cada problema*

6.5.2.2 MÉTODOS DE IMPLEMENTACIÓN OPCIONAL

Los métodos de implementación opcional tienen una implementación por defecto dada en la clase Problema, si bien, suele ser importante darle una implementación adaptada el problema en concreto para ajustar bien los parámetros del algoritmo. Son los siguientes métodos:

```
public double valorDefectoTemperaturaInicial()
public double valorDefectoTemperaturaFinal()
public long valorDefectoEtapasMax()
public long valorDefectoIteraccionesMaxEtapa()
public int valorDefectoAceptacionesMaxEtapa()
public double valorDefectoFactorEnfriamiento()
public double valorDefectoFactorCalentamiento()
public double valorDefectoFactorIncrementoIteraccionesMaxEtapa()
public String ayuda()
```

Figura 99. *Métodos que pueden ser sobrescritos en cada nuevo problema*

Todos estos métodos sirven para obtener información sobre el problema, y tomarse en el caso de que no sean pasados otros valores en el momento de iniciar la simulación. En la interfaz gráfica, al seleccionar un problema, se mostrarán los valores por defecto de dicho problema, establecidos con estos métodos. Básicamente el método ayuda sirve para proporcionar ayuda textual sobre la codificación del problema y los valores que deben tomar los parámetros en la simulación.

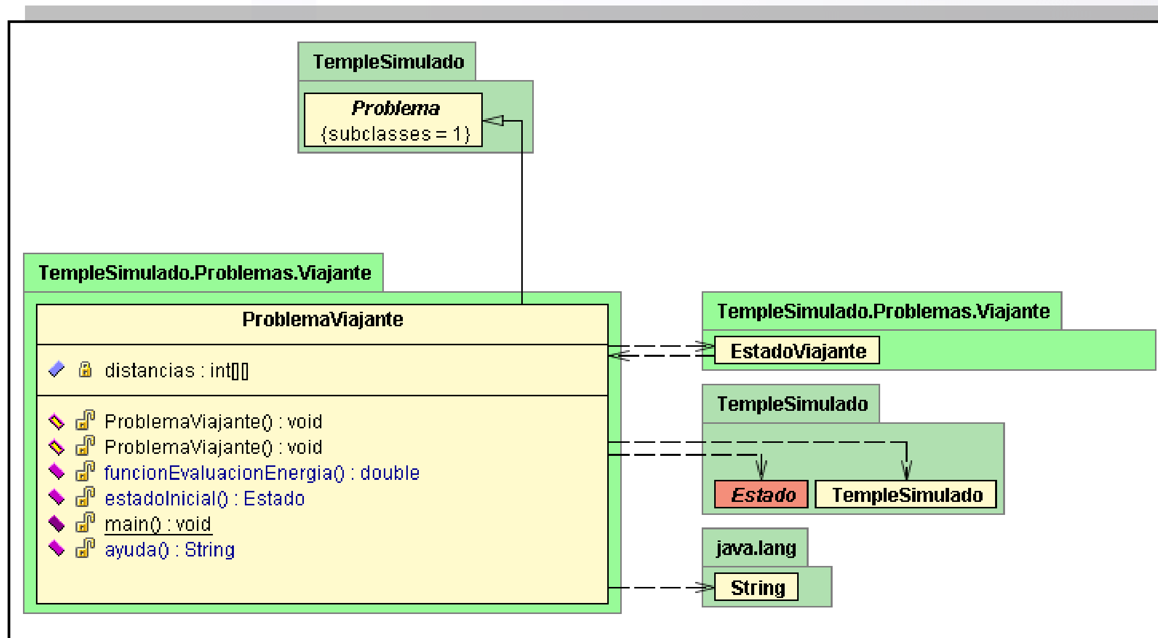


Figura 100. *Clase ProblemaViajante para evaluar los Estados del problema del viajante*

5.6 Modo funcionamiento

Existen principalmente dos maneras de usar la aplicación: mediante la interfaz gráfica, o usándose directamente desde otro programa externo. La ventaja de usar la interfaz de usuario, es poder ver directamente como evoluciona el resultado gráficamente. A veces, no interesa hacer uso de la interfaz gráfica, y solamente nos interesa llamar al método que ejecuta la simulación sin mostrar interfaz gráfica. Vamos a presentar las dos maneras de usar la aplicación.

6.6.1 La interfaz gráfica de Usuario

6.6.1.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

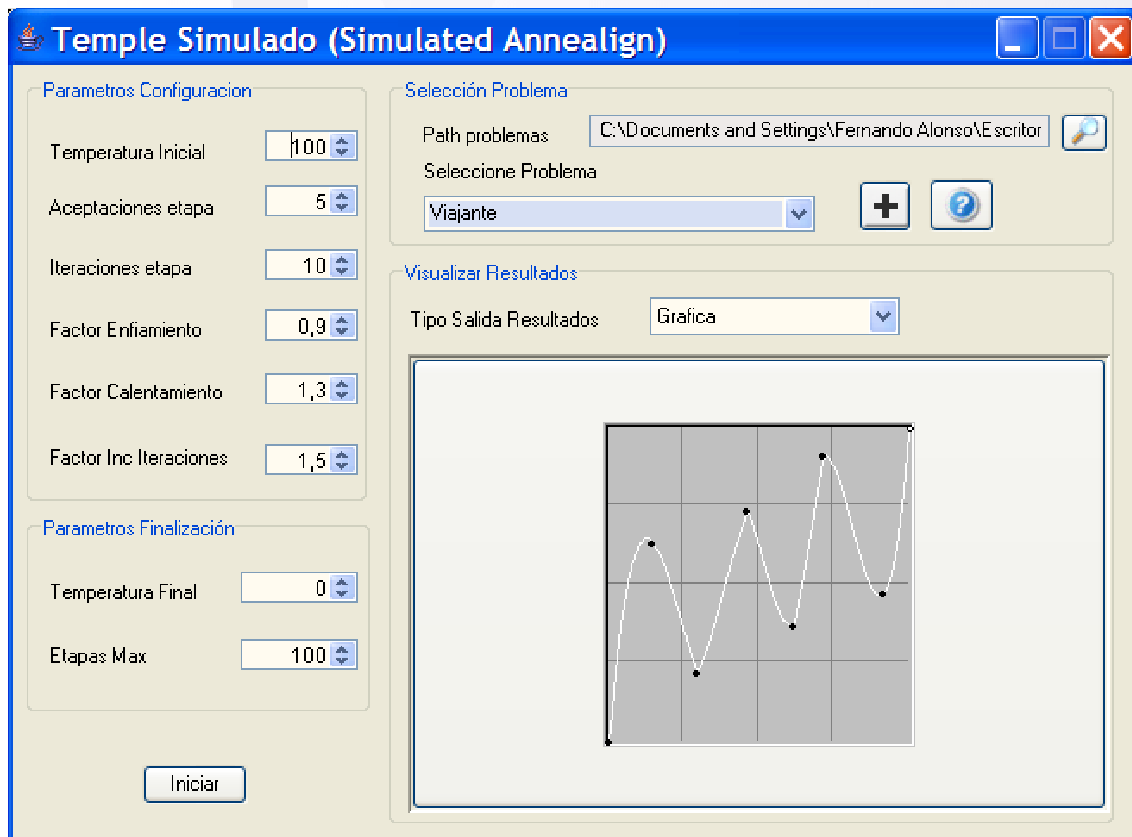


Figura 101. *GUI inicial*

Esta es la vista que presenta la interfaz gráfica de usuario de la herramienta proporcionada para la optimización mediante Temple Simulado. Esta dividida en cuatro panes principales: parámetros de la simulación, parámetros de finalización de la simulación, selección del problema y panel de visualización de resultados.

En el panel llamado "**Parámetros**" , situado arriba a la izquierda se puede seleccionar el valor que tendrán los parámetros del algoritmo, tales como la temperatura inicial del sistema de la que partimos y queremos minimizar, el número de aceptaciones a realizar en cada etapa, el número de iteraciones iniciales por cada etapa (que posteriormente se irán incrementando), el factor de enfriamiento de la temperatura que estará comprendido entre 0 y 1, el factor de calentamiento que deberá ser mayor que 1, y el factor de incremento de las iteraciones máxima a realizar en cada etapa, siendo también mayor que 1 obligatoriamente.

En el panel "**Parámetros de Finalización**" se debe elegir cual es la puntuación final a obtener (deberá ser lógicamente menor que la inicial), y las etapas máximas a generar.

El panel "**Selección del Problema**" se encarga de cargar los programas compilados previamente o crear directamente problemas nuevos para ser compilados y ejecutados desde la propia interfaz o de manera externa. La ruta donde de donde se cargan los problemas y en donde se crean puede ser introducida en el recuadro textual que tiene a su derecha una lupa. Si la ruta no se cambia, cargará los problemas que vienen implementados con el proyecto, si se indica otra ruta, el cargador de clases intentará cargar los problemas compilados que se encuentren en dicha ruta, pinchando en la lupa.

Para crear nuevos problemas, en la ruta indicada, se pedirá un nombre del problema, y se analizará dicha ruta; si la ruta era la original, se creará el problema dentro del propio proyecto, y se mostrará una ventana desde la que se puede trabajar con el código y compilarlo. Si el código queda compilado correctamente se podrá directamente usar desde la interfaz gráfica (sin tener que cerrar y volver a lanzar la aplicación). Posteriormente el código puede ser otra vez editado y compilado usando el *Make.bat* del propio proyecto. Si por el contrario, se ha creado el problema en una ruta distinta de la de por defecto dentro del propio proyecto, se generarán los ficheros relativos al problema en la ruta indicada, así como un archivo *Make.bat* que permite compilar dichos archivos. Esta segunda opción, es importante para crear problemas que no serán incluidos posteriormente en el proyecto, y que simplemente quieren usar la librería HerramientasIA.jar para valerse de las herramientas que esta librería ofrece.

Para obtener ayuda sobre un problema creado y compilado, se podrá pinchar en el botón de ayuda, que mostrará la información relativa al problema que haya sido introducida por el programador.

En el panel "**Visualizar Resultados**" se puede elegir la manera en que los resultados del simulador serán presentados al usuario. Si se elige una presentación por "*terminal*", y si los resultados se quieren mostrar en la interfaz gráfica se podrá elegir "Grafica" que hará la gráfica en tiempo real de la evolución de la temperatura del sistema.

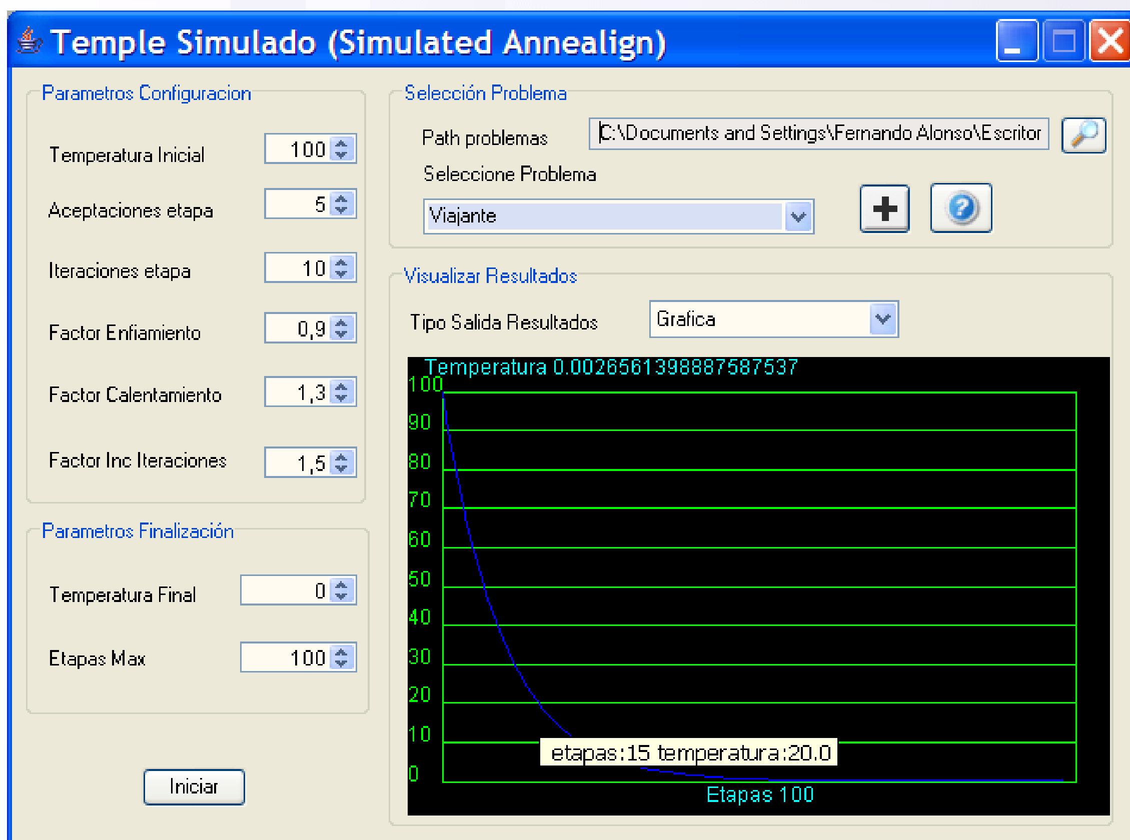


Figura 102. *GUI con información tipo "Tool Tip"*

Se ha añadido información textual a la mayoría de los elementos que la forman. Para obtener información dicha información basta con mantener el ratón situado unos segundos encima, momento en el cual aparecerá el llamado "tool tip". Incluso se puede obtener información sobre algún punto de la gráfica manteniendo el ratón encima del punto deseado y aparecerá las coordenadas relativas a ese punto de la gráfica.

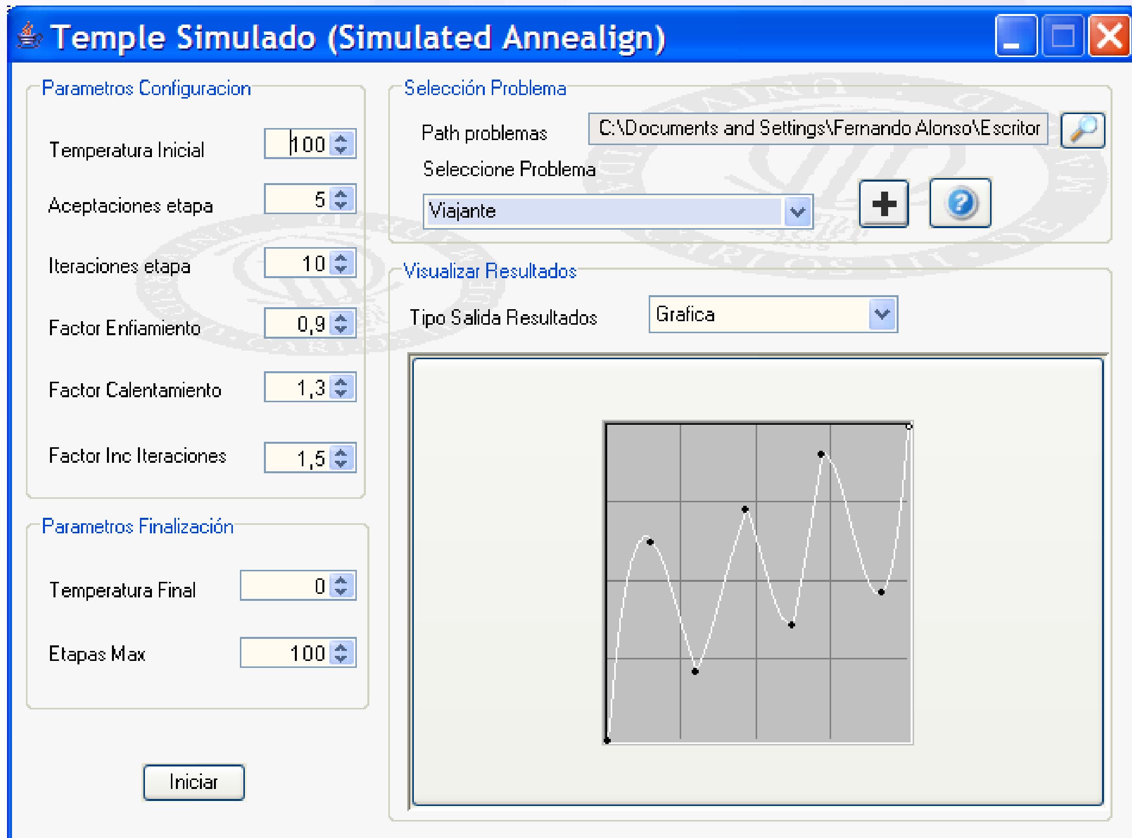
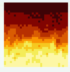


Figura 103. *GUI con nuevo skin (fondo)*

Para conseguir un aspecto más atractivo, se ha añadido además la característica común a todas las herramientas del proyecto de poder cambiar la foto de fondo (el skin o apariencia). Para ello basta con pinchar con el botón derecho sobre cualquier parte de la aplicación y pinchar en cambiar foto de fondo sobre el menú emergente que aparece, posteriormente se selecciona la foto deseada y quedará establecida como fondo. Por defecto en la carpeta “*skin*” del proyecto se adjuntan algunos fondos.



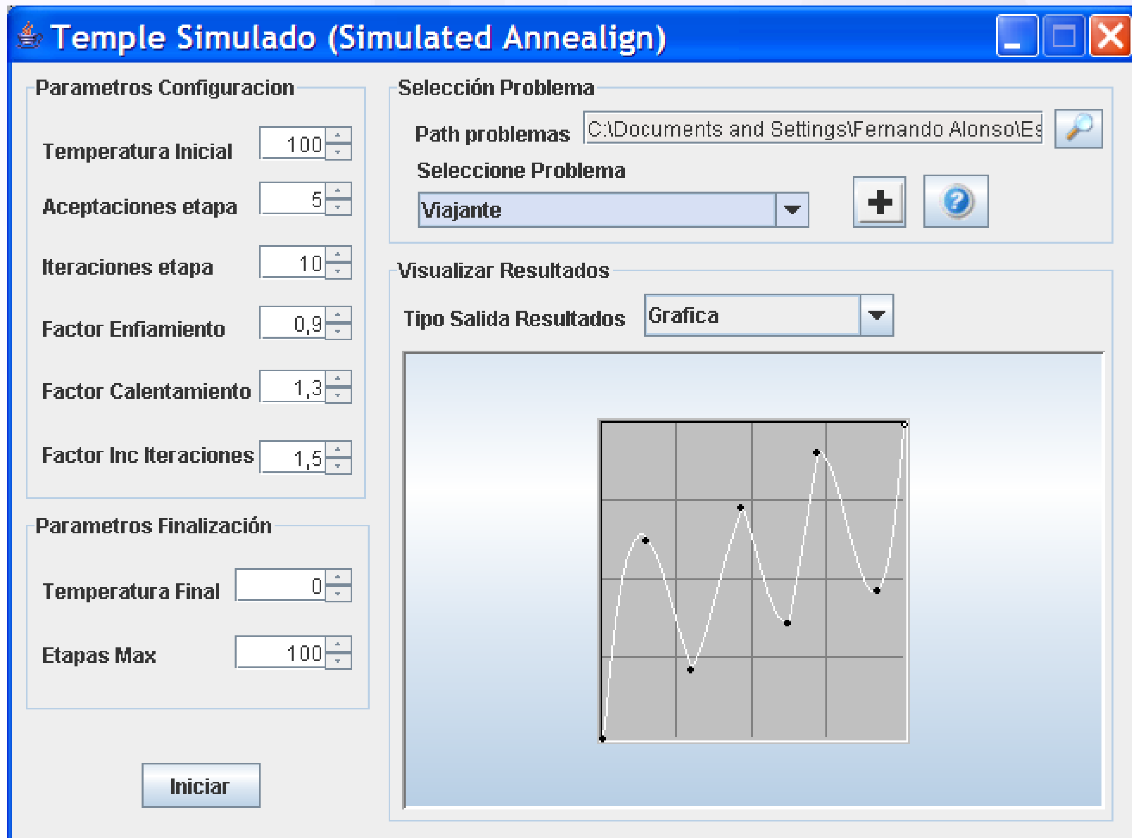


Figura 104. *GUI con nuevo Look & Feel*

También se añadido la posibilidad de cambiar el "look&feel" de la ventana, pudiendo intercambiarse entre varios "Look & Feel", como por ejemplo: java, metal, windows, borland…. Nuevamente esta acción se puede realizar pinchando en el botón derecho del ratón y seleccionando en "cambiar loock&feel", que irá alternando entre los disponibles.

6.6.1.2 EJECUTAR UN PROBLEMA DESDE LA INTERFAZ DE USUARIO

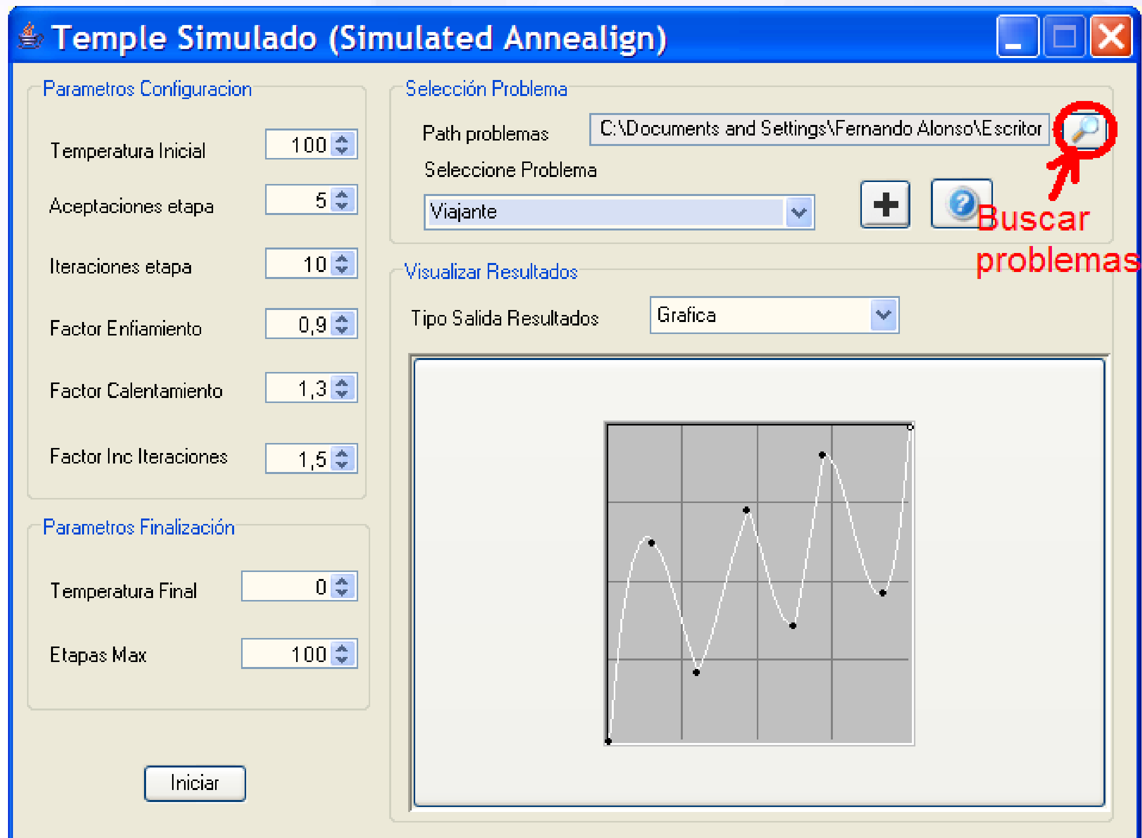


Figura 105. *Buscar problemas en la ruta indicada*

Para buscar los problemas implementados en el propio proyecto o en problemas externos al mismo, basta con indicar la ruta del problema y pinchar en la lupa. De esta manera se cargarán los problemas compatibles con el simulador. Si el problema es externo a la aplicación (se ha creado fuera de la ruta por defecto) se cargará mediante el cargador de clases de Java.

Notar que algunos problemas no podrán ser cargados y así se indicará por terminal, bien porque no están compilados, o no lo están adecuadamente, o simplemente porque no son válidos para ser ejecutados desde la propia interfaz gráfica, puesto que necesiten argumentos que no puedan ser pedidos desde la misma (como objetos). Es decir, sólo podrán ser cargados en la interfaz gráfica aquellos problemas en cuyo constructor no se reciba ningún argumento (si pudiéndolos pedir posteriormente dentro del propio constructor). Los problemas que necesitan recibir argumentos en el constructor serán adecuados para ser lanzados sin la interfaz gráfica, como se puede ver en el punto "6.5.2 Ejecutando el simulador sin la interfaz gráfica".

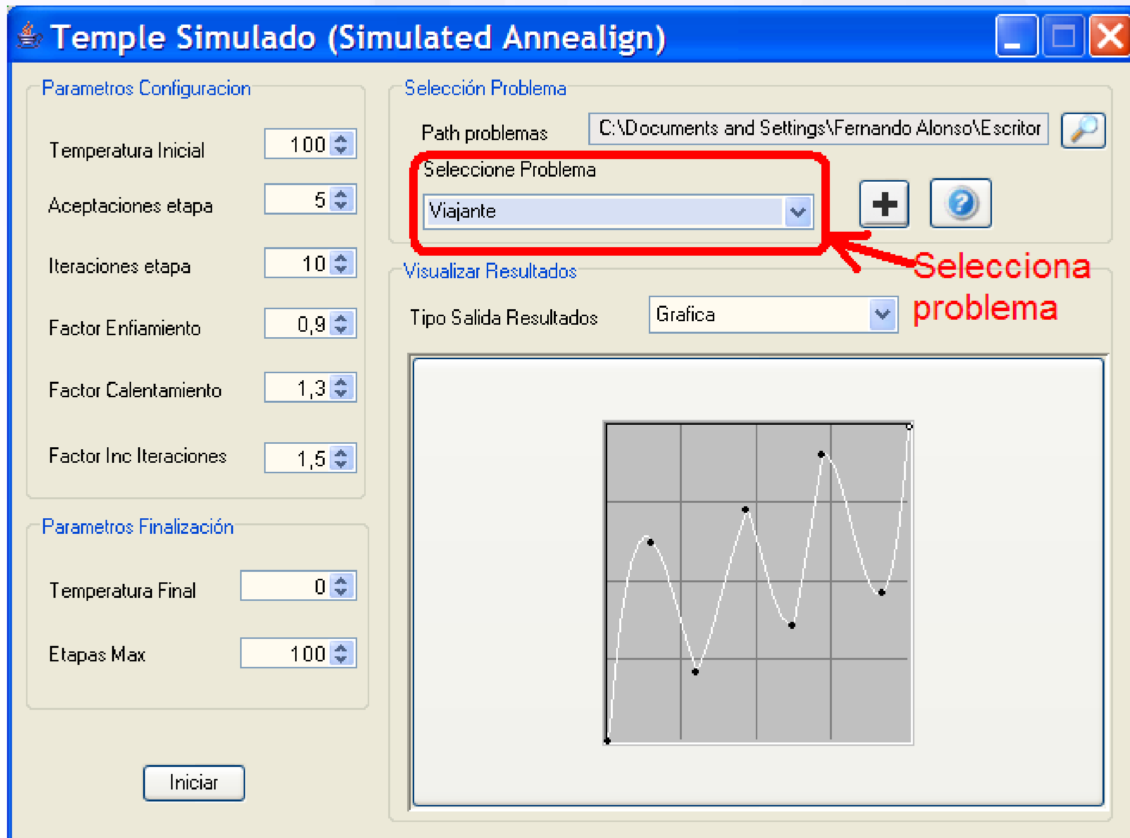


Figura 106. *Selecciona el problema que se quiere solucionar*

De entre los problemas cargados en la ruta indicada se podrá seleccionar aquel que queramos resolver. Una vez seleccionado se establecerán los valores por defecto del problema (si se ha indicado por el programador y sino los valores por defecto de cualquier problema).

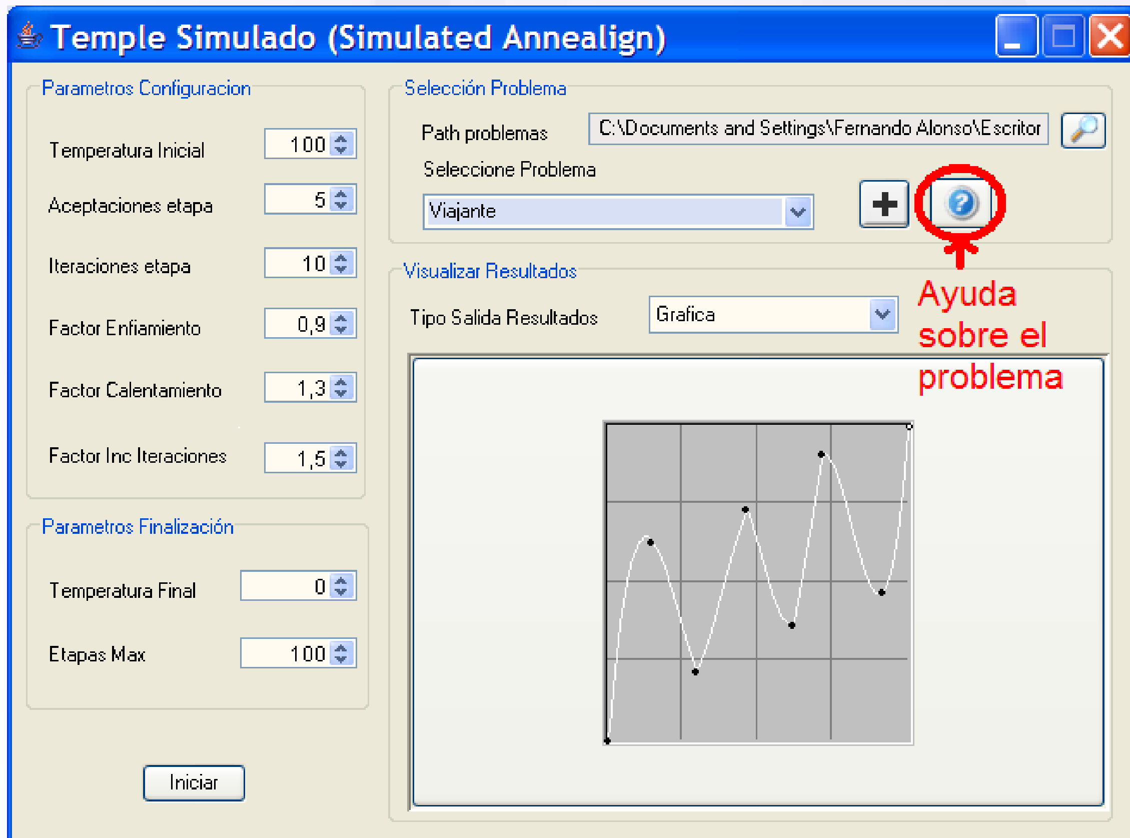


Figura 107. *Obtener ayuda sobre el problema seleccionado*

Si se pincha en el botón de ayuda, se podrá obtener ayuda sobre el propio problema y sobre los valores por defecto establecidos para ese problema. Esta ayuda la introduce el programador del problema, mostrándose la de por defecto sino se ha introducido ninguna nueva.

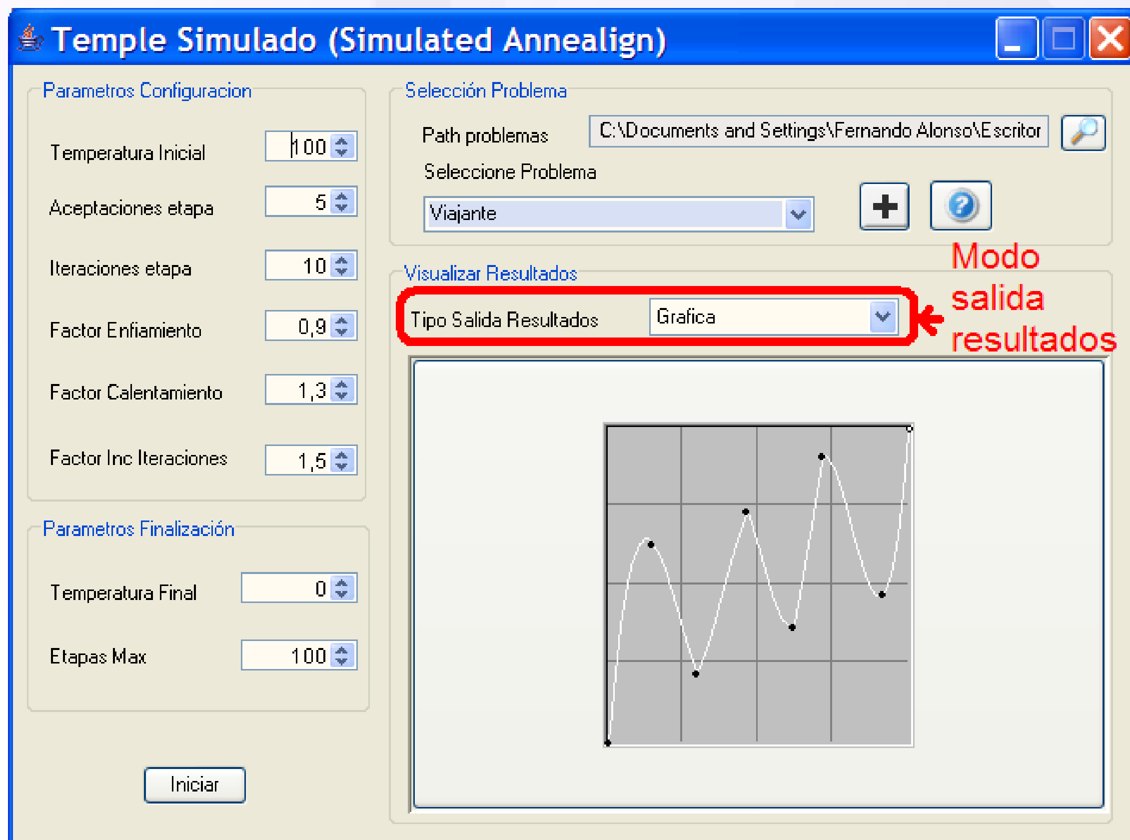


Figura 108. *Visualizar los resultados*

Los resultados y la evolución del algoritmo se pueden mostrar de distintas maneras. La manera usual de mostrar los resultados será de manera gráfica (es decir en la interfaz gráfica), pudiéndose escoger la opción de generar la gráfica que relacione la puntuación obtenida en cada generación. También podrán mostrarse la evolución y los resultados por consola o por fichero. En los tres casos se puede ir mostrando además del individuo mejor, el individuo medio, y todos los individuos que conforman la generación (para este último caso suele ser mas conveniente elegir la opción de visualizar los resultados en fichero). Estas opciones de visualización pueden cambiarse incluso durante la propia simulación. Por último mencionar que se puede decidir cada cuantas iteraciones se mostrará la evolución del algoritmo, por defecto es en todas.

Hacer notar, que cuanta mas salida de resultados se realicen mayor tiempo se invertirá en la simulación.

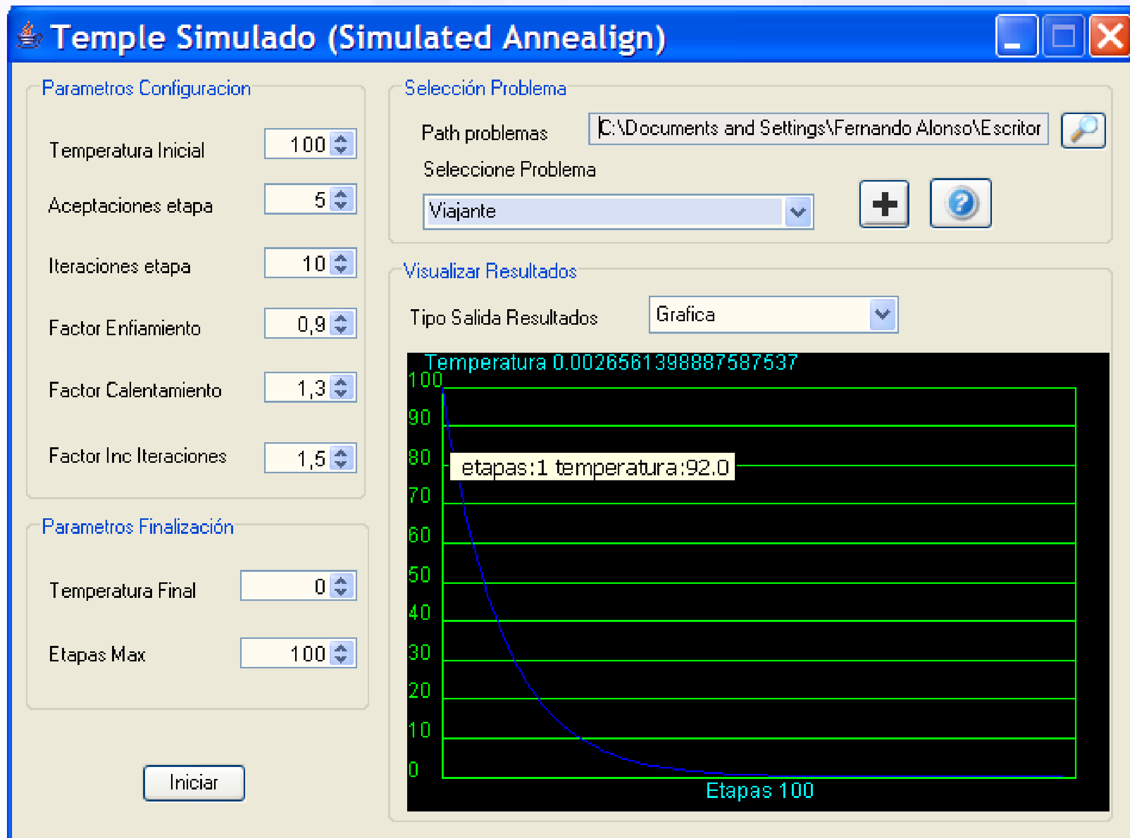


Figura 109. *Ejecutando simulación*

Al hincar la simulación pinchando en el botón de iniciar se e puede ver como se va creando dinámicamente la gráfica que relaciona las generaciones con la puntuación obtenida. Si se mantiene el ratón encima de cualquier punto de la gráfica se pueden ver sus coordenadas exactas. En la gráfica se representa en el eje horizontal las etapas, del algoritmo, ya realizadas, mientras que en el eje vertical se representa la temperatura global del sistema.

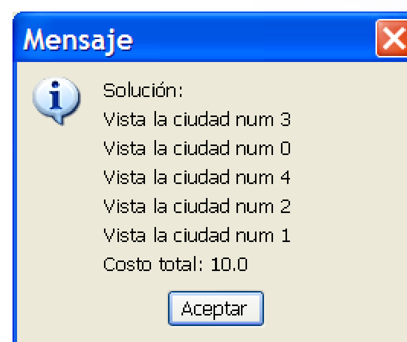


Figura 110. *Resultados de la simulación*

Al finalizar la simulación se muestran los resultados obtenidos directamente de la interpretación que se da del estado solución.

6.6.1.3 CREAR UN NUEVO PROBLEMA DESDE LA INTERFAZ DE USUARIO

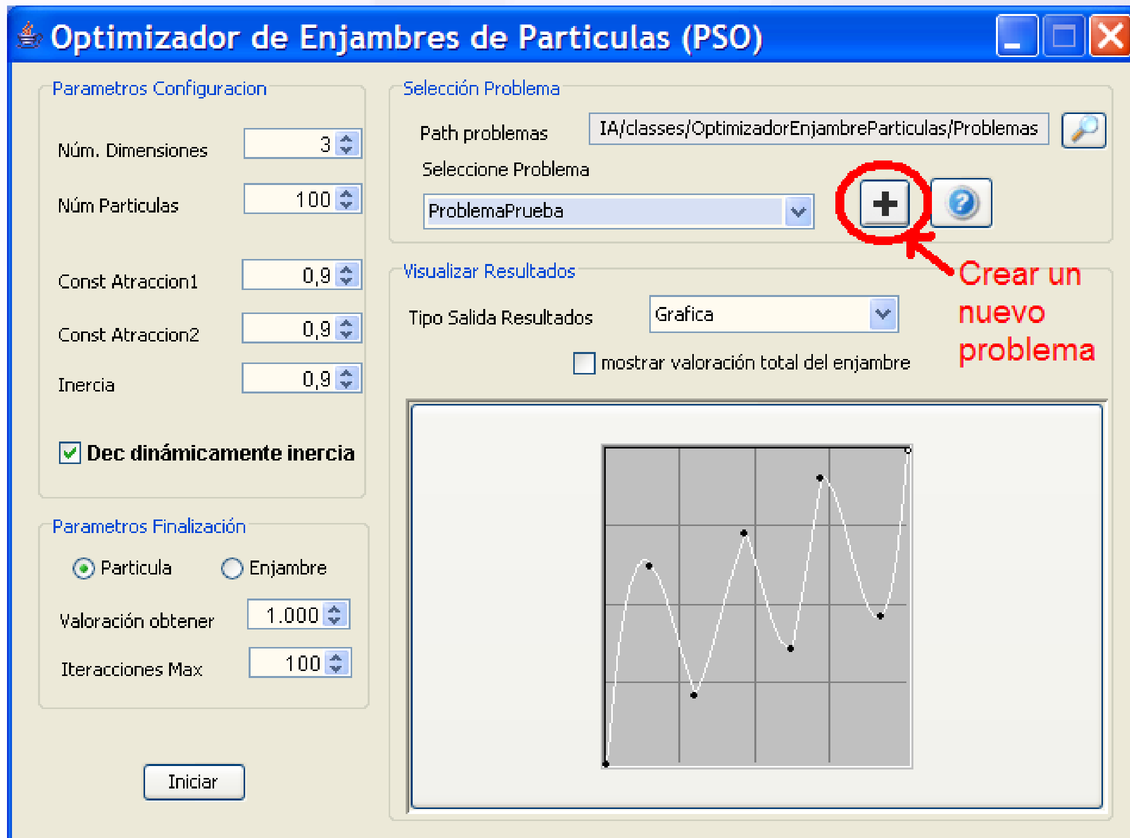


Figura 111. *Crear nuevo problema*

Si se desea crear un problema que estará dentro del propio proyecto (cada vez que se haga un make será también compilado) se debe dejar la ruta especificada en "*Path problemas*" sin modificar ya que apunta directamente a la carpeta con los problemas de Temple Simulado del proyecto. Vamos a analizar ambos casos:

- CREANDO EL PROBLEMA DENTRO DEL PROPIO PROYECTO

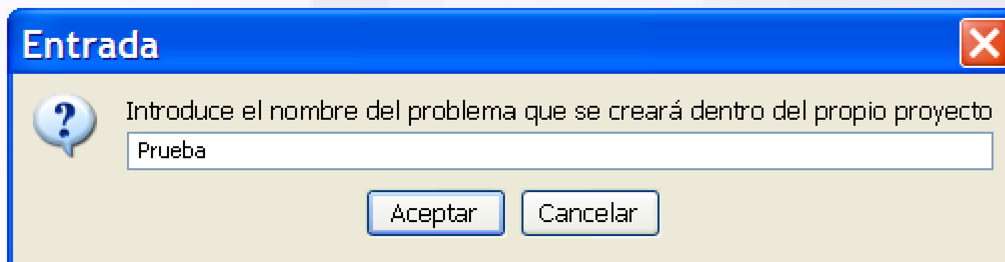


Figura 112. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a la clase que se creará.

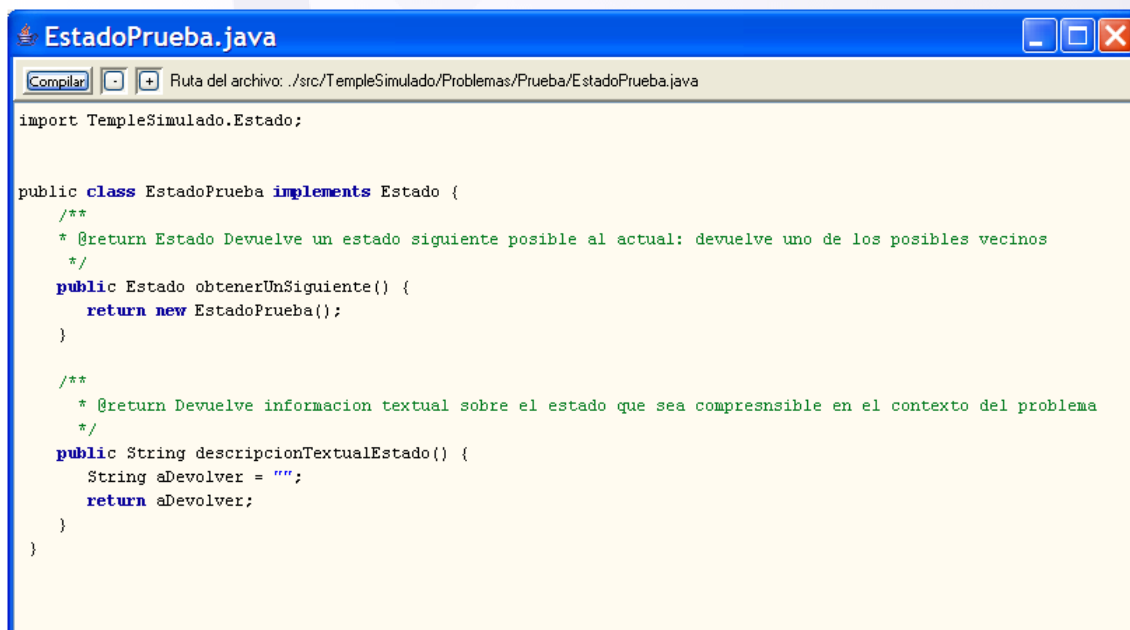


Figura 113. *Estado del problema*

A continuación se muestra un editor desde el que se puede editar y compilar la clase relativa a la codificación de la solución del problema mediante estados. Inicialmente se muestra un esqueleto de código por defecto de cómo debe ser el problema, este se debe modificar para adaptarlo a la codificación del problema que queramos hacer y finalmente compilarlo.

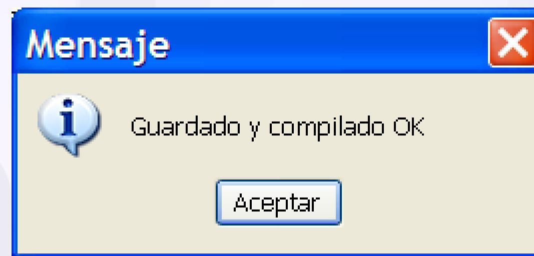
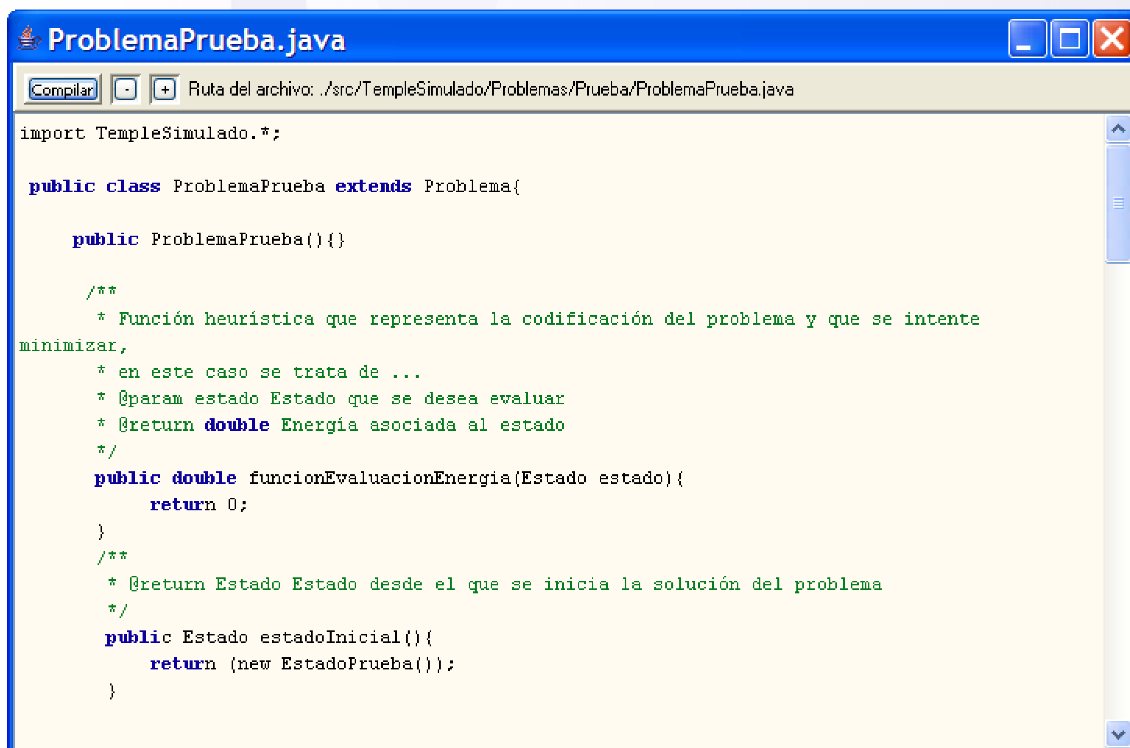


Figura 114. *Problema guardado y compilado*

Si el código compila correctamente, nos pondrá guardado y compilado. El archivo .java se guarda dentro de la carpeta indicada en el path por defecto que es *src/templeSimulado/problemas/nombreProblema* y el .class se guarda en la misma ruta pero de la carpeta classes.




```
import TempleSimulado.*;

 public class ProblemaPrueba extends Problema{

     public ProblemaPrueba(){}

       /**
        * Función heurística que representa la codificación del problema y que se intente
minimizar,
        * en este caso se trata de ...
        * @param estado Estado que se desea evaluar
        * @return double Energía asociada al estado
        */
        public double funcionEvaluacionEnergia(Estado estado){
             return 0;
        }
        /**
         * @return Estado Estado desde el que se inicia la solución del problema
         */
         public Estado estadoInicial(){
             return (new EstadoPrueba());
         }
```

Figura 115. *Implementación del problema*

Una vez compilada también la clase problema que contiene la función de evaluación se podrá seleccionar el problema que se acaba de crear para poder ser ejecutado, tal y como se dice en el apartado "6.6.2 Ejecutar un problema desde la interfaz de usuario", sin tener que cerrar y volver a lanzar la aplicación (salvo que se haya ejecutado la aplicación directamente desde el jar, que entonces debería recompilarse todo el proyecto con el *make* y volver a lanzarlo).

- CREANDO EL PROBLEMA FUERA DEL PROYECTO

Cuando se cambia el path por defecto donde se crearán los problemas, se crean en una ruta distinta de la asignada en el proyecto, por lo que se generarán en la ruta indicada los ficheros esqueleto del problema para que puedan ser editados y compilados con posterioridad.

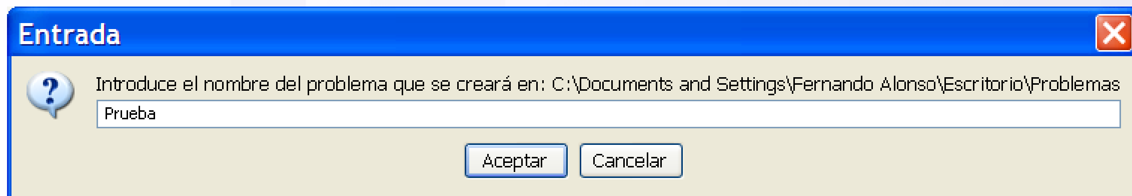


Figura 116. *Introducir nombre del problema*

Primeramente se deberá indicar el nombre que tendrá el problema, que dará nombre a la clase que se creará.

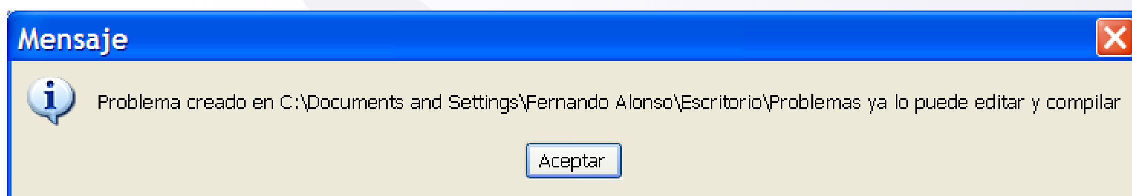


Figura 117. *Mensaje informativo de que se han guardado los ficheros*

Si la ruta indicada es correcta se creara el problema en la ruta indicada y se mostrará el mensaje de la figura anterior. Se habrán guardado los archivos java en la ruta indicada pero no habrán sido compilados. Para compilarlos se deberá ejecutar el archivo “*Make.bat*” que se genera conjuntamente con los archivos, y para los que únicamente usa para compilarlos es la librería “*HerramientasIA.jar*”.

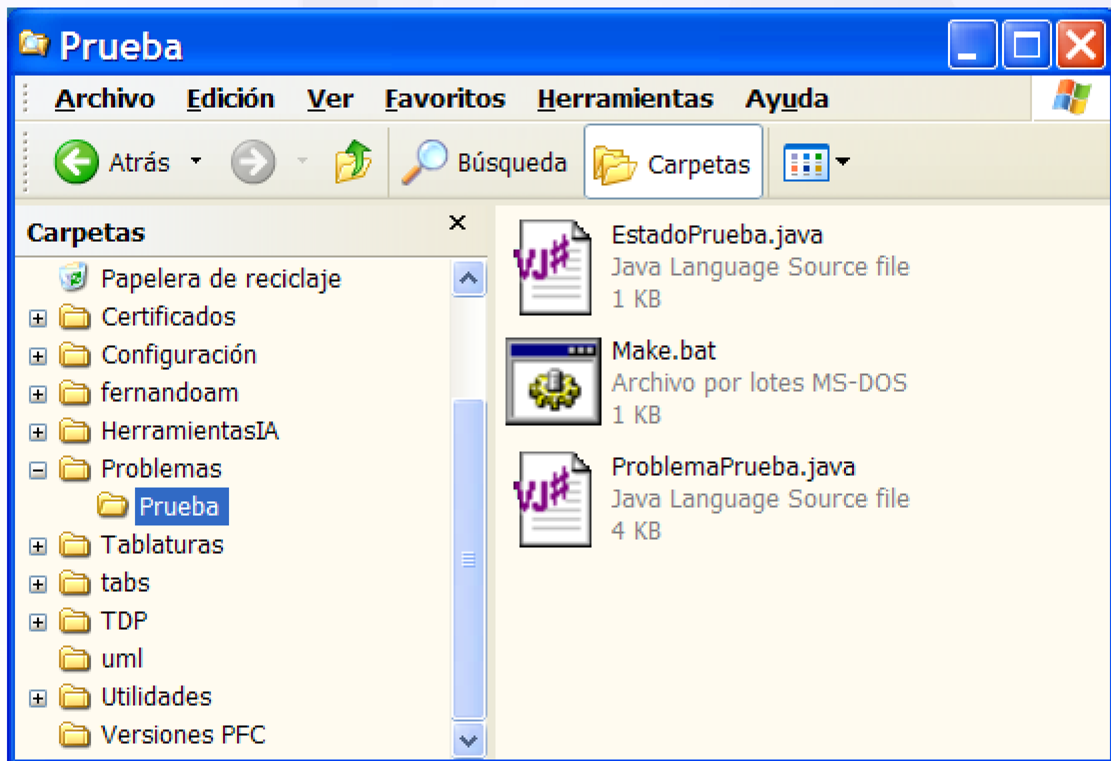


Figura 118. *Archivos fuentes con el esqueleto del problema creado*

Una vez compilados ejecutando "*Make.bat*", puede ser el problema cargado y ejecutado usando la interfaz gráfica "6.6.1.2 Ejecutar un problema desde la interfaz de usuario" o bien "6.6.2 Ejecutando el simulador sin la interfaz gráfica"

6.6.2 Ejecutando el simulador sin la interfaz gráfica

Un problema creado valiéndonos de la interfaz gráfica de usuario, como se dice en la sección "6.6.1.3 Crear un nuevo problema desde la interfaz de usuario" o sin valernos de ella, puede ser ejecutado sin necesidad de ejecutar el main de la aplicación y por tanto de la interfaz gráfica. Para ello se deberá desde el programa que se quiera usar escribir las siguientes líneas:

```
//importamos el TempleSimulado
import TempleSimulado.*;

  //instanciamos el problema
  Problema problemaPrueba = new ProblemaPrueba();

  //instanciamos la clase Temple Simuaaldo
  TempleSimulado temple = new TempleSimulado(problemaPrueba);

  //lanzamos el algoritmo con los valores por defecto
  Estado solucion = temple.templadoSimulado();
```

Figura 119. *Ejecuta problema con los parámetros por defecto*

Lanza el optimizador de manera síncrona y con los valores por defecto dados en la implementación del problema (sino se han sobrescrito los métodos que dan los valores por defecto se quedan los que tienen todos los problemas). Notar que en este caso, se pueden ejecutar problemas cuyo constructor reciba argumentos, como por ejemplo objetos, aunque en el ejemplo no se ha pasado ningún argumento; mientras que ejecutando el problema mediante interfaz gráfica el problema no podía recibir nada por argumentos, puesto que se instancia el problema desde la propia interfaz gráfica con el constructor por defecto sin argumentos.

```
//importamos el TempleSimulado
import TempleSimulado.*;

  //instanciamos el problema
  ProblemaViajante problemaPrueba = new ProblemaPrueba();

  //instanciamos la clase Temple Simualdo
  TempleSimulado temple = new TempleSimulado(problemaPrueba);

  //lanzamos el algoritmo
   double temperaturaInicial=1000;
   double temperaturaFinal=0;
   long etapasMax=100;
   long  iteracionesMaxEtapa=10;
   int aceptacionesMaxEtapa=5;
   double factorEnfriamiento=0.3;
   double factorCalentamient=1.5;
   double factorIncrementoIteraccionexMaxEtapa=1.3;
   Estado estadoInicial=new EstadoPrueba();

   Estado solucion = temple.templadoSimulado(temperaturaInicial,
                                             temperaturaFinal,
                                             etapasMax,
                                             iteracionesMaxEtapa,
                                             aceptacionesMaxEtapa,
                                             factorEnfriamiento,
                                             factorCalentamiento,
                                             factorIncrementoIteraccionexMaxEtapa,
                                             estadoInicial);
```

Figura 120. *Ejecuta problema sin usar los valores por defecto*

Esta segunda manera de lanzar el algoritmo es lanzándolo con los valores concretos que queramos darle, sin tener en cuanta los valores por defecto codificados.

6. Simulador de Redes de Neuronas Artificiales

6.1 Introducción

Una RNA es muy parecida a lo que en matemática discreta se estudia como "grafos". En el caso más simple, cada neurona tiene una serie de variables respecto a sí misma:

- **Estado de activación**: Lo más simple es considerarlo "1" y "0", como on/off

-**Conexiones**: Siguiendo el símil del grafo, estos serían los caminos (unidireccionales) que comunican la neurona con otras.

- **Pesos de las conexiones:** Se trata de una cantidad que mide la "fuerza" de esta conexión. Multiplicado por el estado de activación daría la cantidad numérica que se envía a otra neurona para su activación. Así, una neurona que está activada (1) conectada con un peso de (5) a otra, le enviará en términos matemáticos "5" de input.

- **Umbral de activación**: La cantidad de "input" necesario para activar la neurona. Para decidir si una neurona está activada o no, tenemos en cuenta su entrada. Pongamos que su umbral es "11" y que recibe de tres neuronas conectadas a ella "+4", "+9" y "-1". El resultado es "+12", lo cual supera el umbral y activa la neurona. Sin embargo, si el "input" fuera "+4" y "+5", el total sería "+9" lo cual no llega al umbral que hemos definido; la neurona destino estaría por tanto con un estado "0".

Con estos conceptos en mente - aunque simplificados, pues umbral y activación pueden ser funciones, etc -, que aclaran mucho incluso las ideas sobre el funcionamiento de las redes de neuronas naturales, toca explicar lo que más intrigante resulta; ¿ cómo aprenden estas redes ?

El concepto clave es que los pesos de conexión entre neuronas pueden - y han de ser - modificados.

Como ejemplo, valga un aprendizaje "supervisado": supongamos tres capas de neuronas, compuestas por 5, 5 y 2 de ellas respectivamente, y que están conectadas la primera capa (input) con la segunda y la segunda con la tercera (que será la de output).

Esto significa que al ser la de 2 la capa final y teniendo 4 combinaciones (00,01,10,11), podemos identificar cuatro "patrones" distintos. Esto es, que quizá si activamos de la primera capa las neuronas 1 y 3 obtengamos que se activa la neurona 1 de la tercera capa por el juego de activaciones/pesos en la segunda y tercera capas.

Pongámonos en esa situación, lo siguiente que pensaríamos es, ¿se trata de una respuesta correcta la del output respecto al input recibido? Si esto es así, **reforzamos** las conexiones que estaban activas, y **disminuímos** las otras. Asi, la próxima vez que se presente un patrón parecido a (1,3) habrá más probabilidades de que la respuesta sea correcta, y menos de que se de esta respuesta en caso de no ser así. Tras hacer mucho estas veces, la red aprenderá a discriminar los patrones que le enseñemos.

Por supuesto que también hay métodos de aprendizaje "no supervisado", en que las neuronas aprenden por sí mismas; el aprendizaje competitivo es un buen ejemplo. Ya está claro el método por el cual las RNA aprenden para adaptarse al input y realizar una respuesta correcta. De hecho, estos algoritmos han demostrado servir de maravilla para cosas como reconocimiento de caras, visión artificial... imaginemos que la capa de input de la red de neuronas fueran los pixels de una cara que se capta mediante una cámara. Entrenando a la red, se la podría hacer diferenciar rostros con una paralelización absoluta y una posibilidad ínfima de error... ya hay bancos que confían en programas de redes de neuronas para distinguir mediante biometría de la pupila a sus clientes, lo que muestra que esto es muy efectivo.

Las RNA son dos cosas; primero un intento de imitar nuestra forma de pensar, por otro lado un magnífico algoritmo basado en la paralelización masiva, al contrario de los sistemas informáticos habituales que se basan en procesar las cosas en serie. Esa, es también la forma que tiene el ser humano de pensar. En un sentido simbólico, la red de neuronas es una “caja negra” por la cual se reciben unas entradas y se generan unas salidas.

Muchas veces se dice que los ordenadores han superado al hombre; sin embargo no somos capaces de mantener una buena conversación con uno, y cosas que para nosotros son tan sencillas como identificar un rostro en una multitud, para el ordenador basado en los algoritmos en serie es casi imposible. Si a nosotros nos hablan de un "animal que tiene trompa" inmediatamente pensamos en un elefante. Sin embargo, un

ordenador habría de buscar recursivamenteen su base de datos sobre animales, uno por uno, hasta encontrar coincidencias. El famoso "Deep Blue" busca recursivamente puntuando las posibles jugadas que realizar en su turno al ajedrez, pero no elimina jugadas automáticamente como hace el jugador humano.

Es sencillo de explicar; imaginemos que tuviésemos una base de datos de 500 neuronas donde pudiesemos identificar X animales. Si se activase la neurona de entrada "tener trompa", automáticamente se produciría el output de que se trata de un elefante. Esto por ejemplo, es lo que hacen las redes artificiales de Hopfield.

¿Cuál es la base de nuestro pensamiento, pues?. La clasificación de patrones, y la reacción ante ellos. Durante todo el día estamos clasificando cosas; lo que vemos lo identificamos respecto a un concepto, utilidad, etc. Respondemos con patrones de conducta grabados a situaciones conocidas como puede ser algo tan sencillo como ir de compras. Distinguimos el estado de ánimo de aquel con quien hablamos, y constantemente analizamos sus gestos y palabras, dividiéndolas según los significados que implican o su entonación. Incluso, cuando juzgamos algo como "bueno" o "malo", cuando pensamos que algo es "justo" o "injusto", no estamos más que haciendo una clasificación...

Parándose a pensar, es fácil ver cómo la mayoría de nuestra actividad responde a este funcionamiento de clasificación de input en patrones. Eso sí, imagino que aún hay dos preguntas pendientes - aparte de que qué es la consciencia, la cual es casi imposible responder... de momento - que uno puede hacerse: ¿cómo aprendemos nosotros, y qué son los sentimientos?

Curiosamente, la respuesta a las dos preguntas está muy relacionada. La base del aprendizaje es el estímulo positivo/negativo; estímulos positivos refuerzan conductas mientras que los negativos disminuyen la probabilidad de que ocurran. Evidentemente, hay una pequeña base genética para determinar qué es positivo y qué negativo, que luego se complejiza enormemente hasta llegar a un punto en que estamos totalmente condicionados por nuestro ambiente. Somos capaces, incluso, de vencer "instintos" como el de supervivencia o el de reproducción, y de tener una visión muy personal de lo que significa un estímulo positivo y uno negativo.

Por ejemplo, un bebé dice "mamá" por primera vez. Al hacerlo simplemente es que lo ha oído unas cuantas veces, no sabe lo que significa pero está aprendiendo e imitando sonidos. De sus padres va a recibir un estímulo positivo en forma de caricias y demás, que van a reforzar esa conducta. Gracias a eso, poco a poco irá aprendiendo... si al decir una palabra los padres golpeasen al hijo, sería bastante difícil que este aprendiera a hablar.

6.2 Pseudo-código del algoritmo principal

6.3 Arquitectura de la aplicación

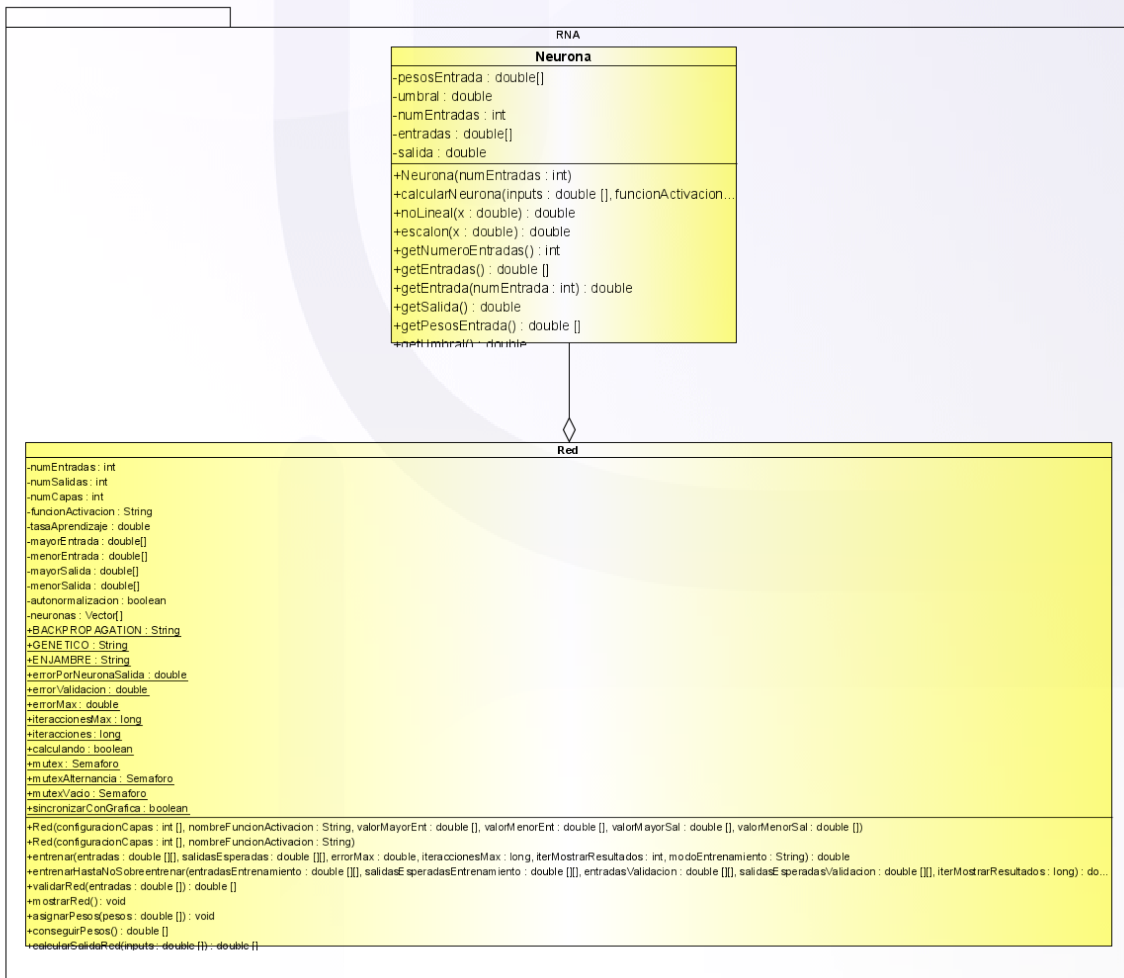


Figura 121. *Diagrama UML de la Red de Neuronas Artificiales*

6.4 Particularidades de la implementación

6.4.1 Autonormalización de las entradas y salidas

Las entradas y las salidas de la red de neuronas se suelen pasar sin normalizar, es decir, tal y como se manejan en el mundo real, pero la red de neuronas artificiales internamente necesita trabajar con los valores normalizados, es decir comprendidos entre 0 y 1.

La implementación que se ha realizado de la red de neuronas abstrae al usuario de la tarea de normalizar las entradas, puesto que la propia red de neuronas normaliza las entradas y desnormaliza las salidas resultantes.

La normalización se realiza por cada neurona de entrada, es decir, que teniendo en cuenta el valor más alto y mas bajo de entre los que recibe en el entrenamiento y en la validación escala el resto a esos valores, tomando el valor mas alto recibido un uno, y el valor mas bajo recibido un cero. Para la desnormalización de las salidas se realiza el proceso inverso, teniendo en cuenta el valor más alto y más bajo que pueden tomar cada neurona de salida.

Notar que si los valores de validación, son más altos o bajos que los de entrenamiento y no se entrena mediante el entrenamiento/validación simultaneo sería necesario en la llamada a la red, indicarle explícitamente cuales son los valores mas altos y bajos por cada neurona de entrada y de salida. Esto es así, porque si se entrena para unos determinados valores, la normalización queda escalada a los valores de los patrones de entrenamiento, mientras que si luego en la validación se presentan valores mas altos o bajos, el escalado anteriormente realizado no es bueno.

6.4.2 Distintas funciones de activación

Existen varias funciones para activar la salida de cada neurona, de momento se encuentran implementadas la función sigmoidal (o no lineal) y la función escalón.

Si se desean implementar nuevas funciones únicamente hay que implementarla dentro de la clase "Neurona.java" y a la hora de crear la red indicarle que se va a usar el nombre de esa función de activación (por reflexión el programa es capaz de invocar esa función de activación).

La elección de una correcta función de activación influye directamente en el proceso de aprendizaje, ya que una mala función de activación puede provocar que el algoritmo no aprenda. Normalmente se suele usar la función "noLineal" (sigmoidal).

6.4.3 Distintos modos de entrenamiento

Básicamente existen dos paradigmas o formas de entrenar una red de neuronas, diciéndole explícitamente el número de iteraciones máximas de entrenamiento, o bien diciéndole que entrene y valide al mismo tiempo y pare únicamente cuando empiece a sobreentrenarse. Estas dos maneras de entrenamiento han sido implementadas en la aplicación.

6.4.4 Distintos algoritmos de aprendizaje

Muy directamente relacionado con el modo de entrenamiento que se escoja se puede entrenar la red con algoritmos de aprendizaje distintos. En la implementación que hemos desarrollado, si se elige el modo de entrenamiento clásico (fijando las iteraciones máximas) se puede elegir entre el modo de entrenamiento basado en *backpropagation*, o en algoritmos genéticos o en enjambres de partículas.

6.4.4.1 BACKPROPAGATION

El *Backpropagation* es uno de los algoritmos de aprendizaje supervisado de redes de neuronas más populares. Los vectores de entrada y los correspondientes vectores deseados se usan para entrenar una red hasta cuando ella pueda aproximar una función, asociando los vectores de entrada con los vectores de salida específicos, ó clasificar los vectores de entrada de una manera apropiada tal como nosotros la definimos.

El *backpropagation* estándar es un algoritmo de gradiente descendente, en el cual los pesos de la red son movidos a lo largo del negativo del gradiente de la función de ejecución. El término *backpropagation* se refiere a la manera como el gradiente es calculado para redes multicapa no lineales. Existe un cierto número de variaciones en el algoritmo básico.

Además se ha incorporado una novedad bastante interesante y es el de ir cambiando dinámicamente la razón de aprendizaje. La razón de aprendizaje es la responsable de la velocidad de convergencia del algoritmo. A velocidades muy altas, el algoritmo puede estar "rebotando" sin ser capaz de disminuir el error, en cambio a velocidades muy bajas (razón de aprendizaje baja) el tiempo de convergencia y la probabilidades de quedarse en un mínimo local son muy elevadas. Por ello, se modifica dinámicamente la razón de aprendizaje, según en que momento se encuentre el algoritmo, para intentar evitar quedarse en mínimos locales.

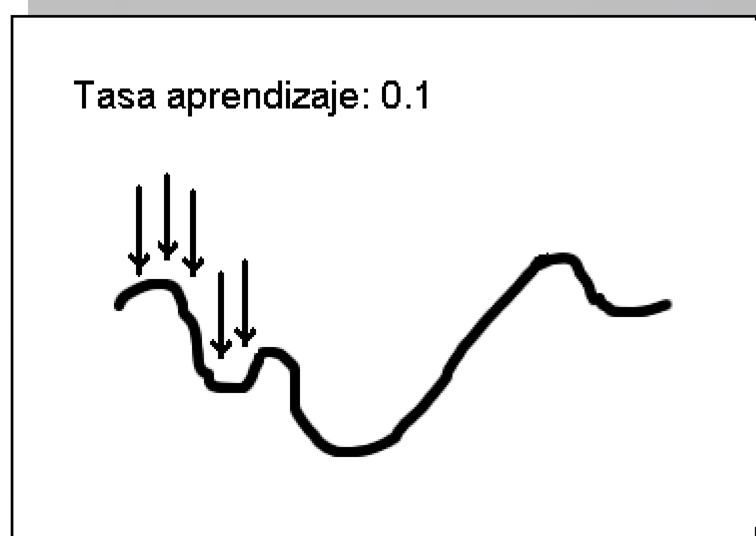


Figura 122. *Backpropagation con tasa de aprendizaje pequeña*

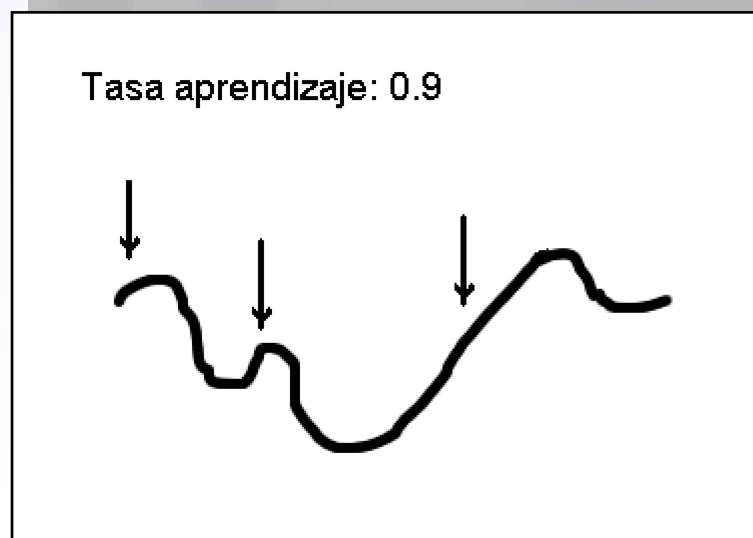


Figura 123. *Backpropagation con tasa de aprendizaje alta*

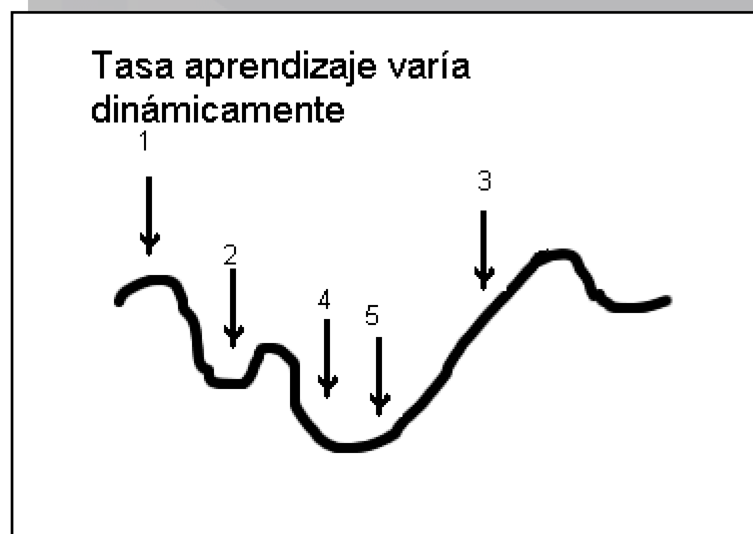


Figura 124. *Backpropagation con tasa de aprendizaje dinámica*

6.4.4.2 APRENDIZAJE BASADO EN ALGORITMOS GENÉTICOS

El aprendizaje basado en algoritmos genéticos se ha realizado creando un problema precisamente para resolver este problema y se basa en codificar en cada individuo todos los pesos de la red de neuronas, y su evaluación es inversamente proporcional al error que se comente al pasar los patrones de entrenamiento a la red cuyos pesos son los codificados en el individuo.

6.4.4.3 APRENDIZAJE BASADO EN ENJAMBRES DE PARTÍCULAS

El aprendizaje basado en enjambres de partículas se ha realizado creando un problema precisamente para resolver este problema, y se basa en codificar en cada partícula todos los pesos de la red de neuronas, de la siguiente manera. Cada partícula tendrá tantas dimensiones como pesos tenga la red de neuronas (número de conexiones entre neuronas). De tal manera que la posición de la partícula determina los pesos de la red. Una partícula de n-dimensiones, estará situada en un espacio n-dimensional, y por tanto su posición tendrá n-componentes, de tal manera que cada componente de la posición es un peso de la red de neuronas. La manera de evaluar la partícula es inversamente proporcional al error que se comete al pasar los patrones de entrenamiento a la red cuyos pesos son codificados en la partícula.

6.5 Modo funcionamiento

Existen principalmente dos maneras de usar la aplicación: mediante la interfaz gráfica, o usándose directamente desde código Java. La ventaja de usar la interfaz de usuario, es poder ver directamente como evoluciona el resultado gráficamente, pero a veces, no interesa hacer uso de la interfaz gráfica. Vamos a presentar las dos maneras de usar la aplicación.

6.5.1 La interfaz gráfica de Usuario

6.5.1.1 DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

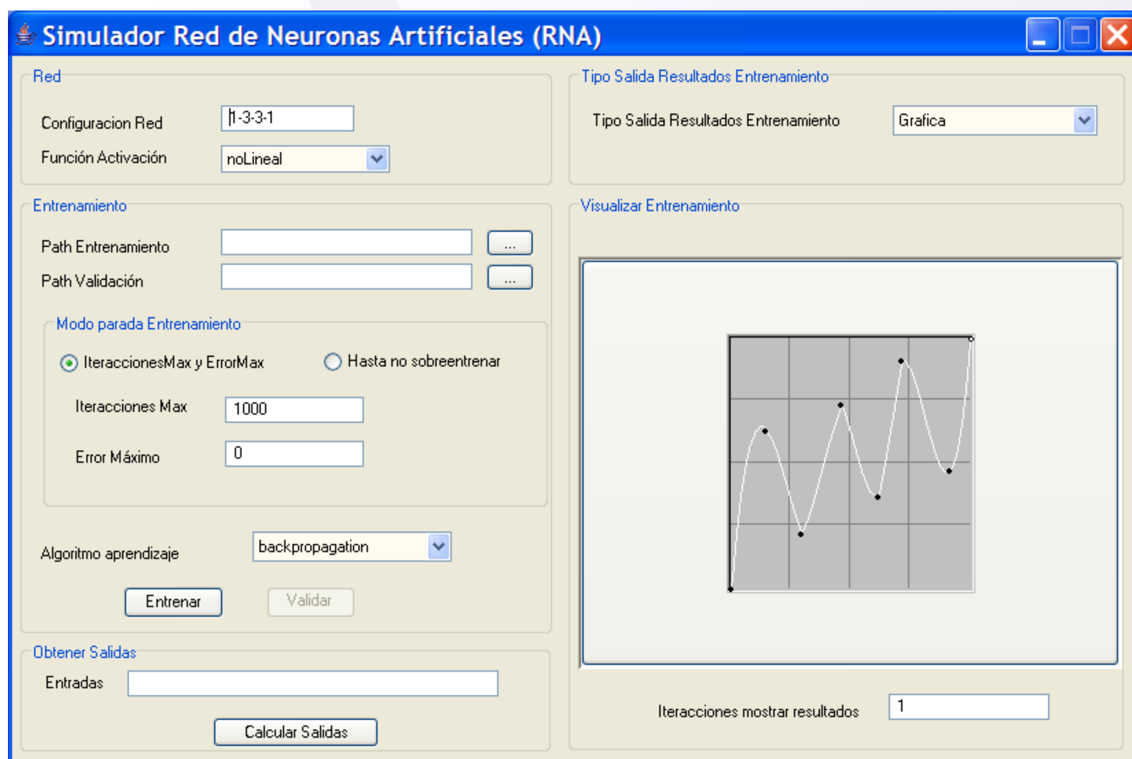


Figura 125. *GUI inicial*

Esta es la vista que presenta la interfaz gráfica de usuario de la herramienta proporcionada para las Redes de Neuronas Artificiales. Esta dividida en varios paneles principales: configuración de la red, entrenamiento, obtención de salidas, tipo de salida de resultados, y visualización de los resultados.

En el panel llamado "**Red**" se puede configurar la topología de la red, es decir en número de neurona que se localizan en cada capa. Podrá haber tantas capas como se quiera, al igual que tantas neuronas por capa como se desee. En cuanto a la función de activación determina como se activa la salida de cada neurona, por defecto están implementadas dos funciones la noLineal (sigmoidal) y la escalón. Pero se pueden implementar nuevas funciones en la clase "*Neurona.java*", que por reflexión podrán ser usadas directamente.

En el panel "**Entrenamiento**" se indican la ruta donde se localiza el fichero de entrenamiento y el fichero de validación. Con ellos se puede realizar dos tipos de entrenamiento. El primero tipo de entrenamiento consiste en entrenar durante una serie de iteraciones y hasta disminuir un determinado error, esta es la manera tradicional de realizar el entrenamiento. El segundo modo de entrenar, consiste en entrenar y al mismo tiempo validar, de tal manera que el entrenamiento sólo se parará si el error de entrenamiento esta disminuyendo y en cambio el error de validación está aumentando durante una serie de iteraciones, esta manera de trabajar nos permite entrenar hasta el momento justo en que empieza a producirse el sobreentrenamiento.

Por último se puede seleccionar el tipo de algoritmo de aprendizaje de la red. El algoritmo típico de aprendizaje es el algoritmo de "*backpropagation*" o también llamado de "*retropropagación*", que está implementado de tal manera que la razón de aprendizaje varia dinámicamente para evitar que el aprendizaje se quede estancado en un mínimo local. Aparte de esta función de entrenamiento existen dos tipos mas de entrenamiento basados precisamente en las demás herramientas incluidas en este proyecto como son los algoritmos genéticos y los enjambres de partículas. Que se explicaron en el punto "Distintos algoritmos de aprendizaje".

El panel "**Obtener Salidas**" se puede introducir manualmente las entradas que se le quieren pasar a la red, para que nos diga las salidas que se producen. Es una manera de realizar la validación directamente sobre la interfaz gráfica.

En el panel "**Tipo Salidas Resultados Entrenamiento**" se puede seleccionar si la salida se hará directamente en modo gráfico o por terminal. Si el algoritmo de aprendizaje es *backpropagatión* se puede elegir el tipo de salida, sin embargo para enjambres de partículas y algoritmos genéticos sólo se puede en modo texto, ya que el

entrenamiento se hace gracias a las herramientas respectivas de una manera externa a la propia red de neuronas.

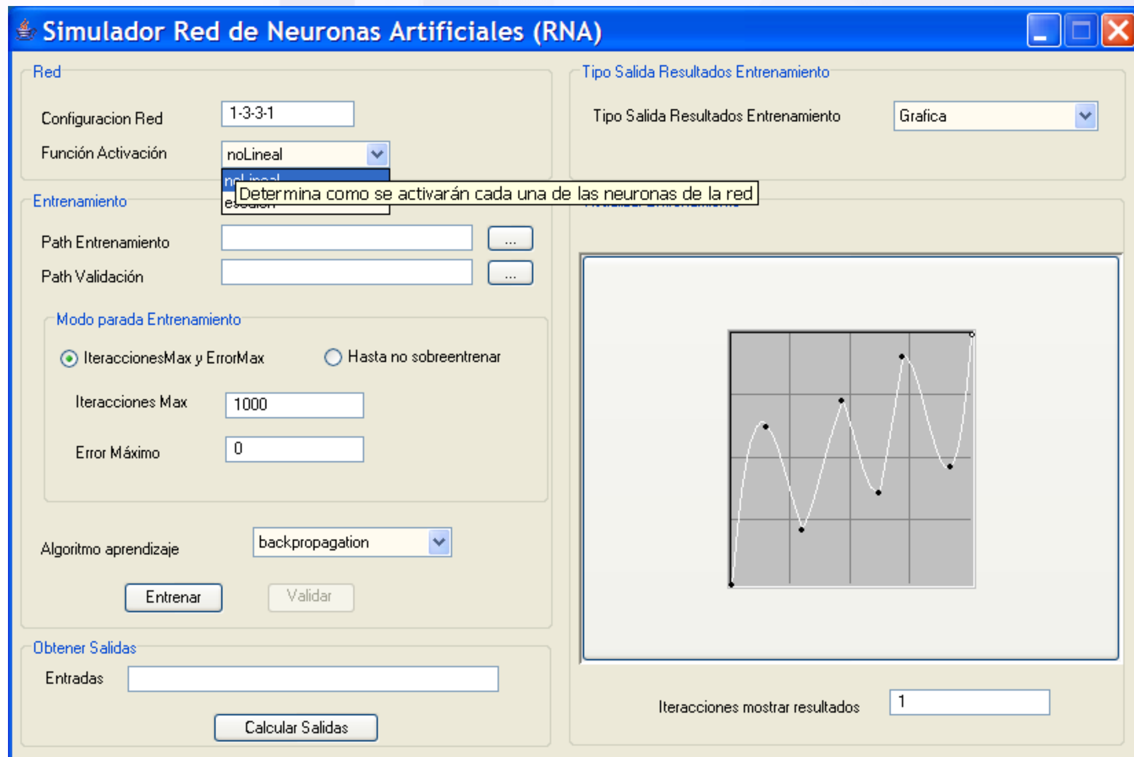


Figura 126. *GUI con información tipo "Tool Tip"*

El aspecto de la herramienta se ha intentado que quedará lo mas intuitivo y simple posible, pero aún así puede resultar en un principio algo confusa, para evitar confusiones, se ha añadido información textual a la mayoría de los elementos que la forman. Para obtener información dicha información basta con mantener el ratón situado unos segundos encima, momento en el cual aparecerá el llamado "tool tip". Incluso se puede obtener información sobre algún punto de la gráfica manteniendo el ratón encima del punto deseado y aparecerá las coordenadas relativas a ese punto de la gráfica.

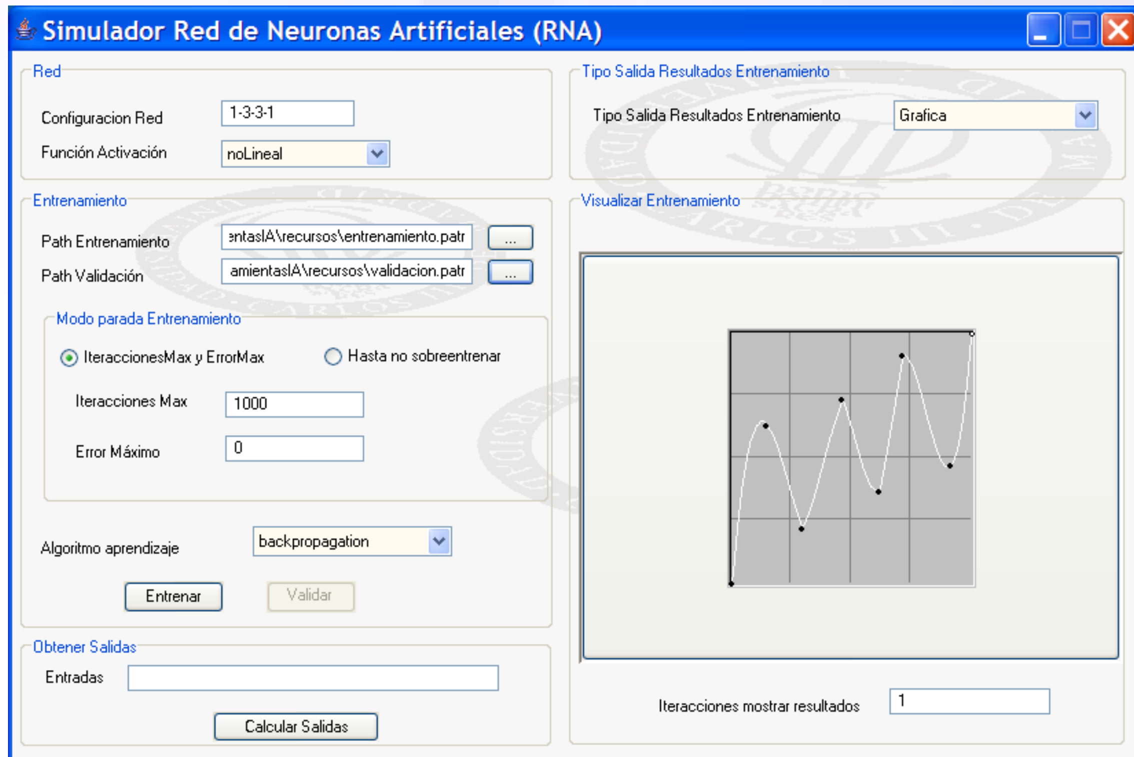


Figura 127. *GUI con nuevo skin (fondo)*

Para conseguir un aspecto más atractivo, se ha añadido además la característica común a todas las herramientas del proyecto de poder cambiar la foto de fondo (el skin o apariencia). Para ello basta con pinchar con el botón derecho sobre cualquier parte de la aplicación y pinchar en cambiar foto de fondo sobre el menú emergente que aparece, posteriormente se selecciona la foto deseada y quedará establecida como fondo. Por defecto en la carpeta "*skin*" del proyecto se adjuntan algunos fondos.

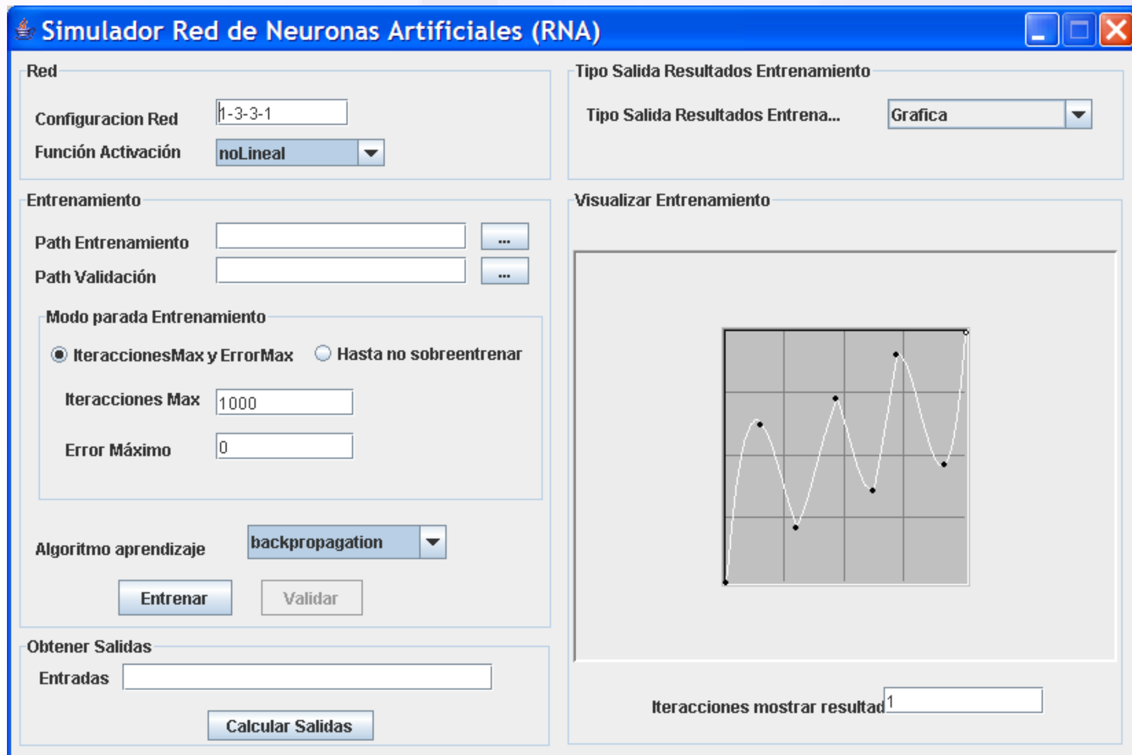


Figura 128. *GUI con nuevo Look & Feel*

También se añadido la posibilidad de cambiar el "look&feel" de la ventana, pudiendo intercambiarse entre varios "Look & Feel", como por ejemplo: java, metal, windows, borland…. Nuevamente esta acción se puede realizar pinchando en el botón derecho del ratón y seleccionando en "cambiar loock&feel", que irá alternando entre los disponibles.

6.6.1.2 MODO DE ENTRENAMIENTO/VALIDACIÓN/OBTENCION RESULTADOS

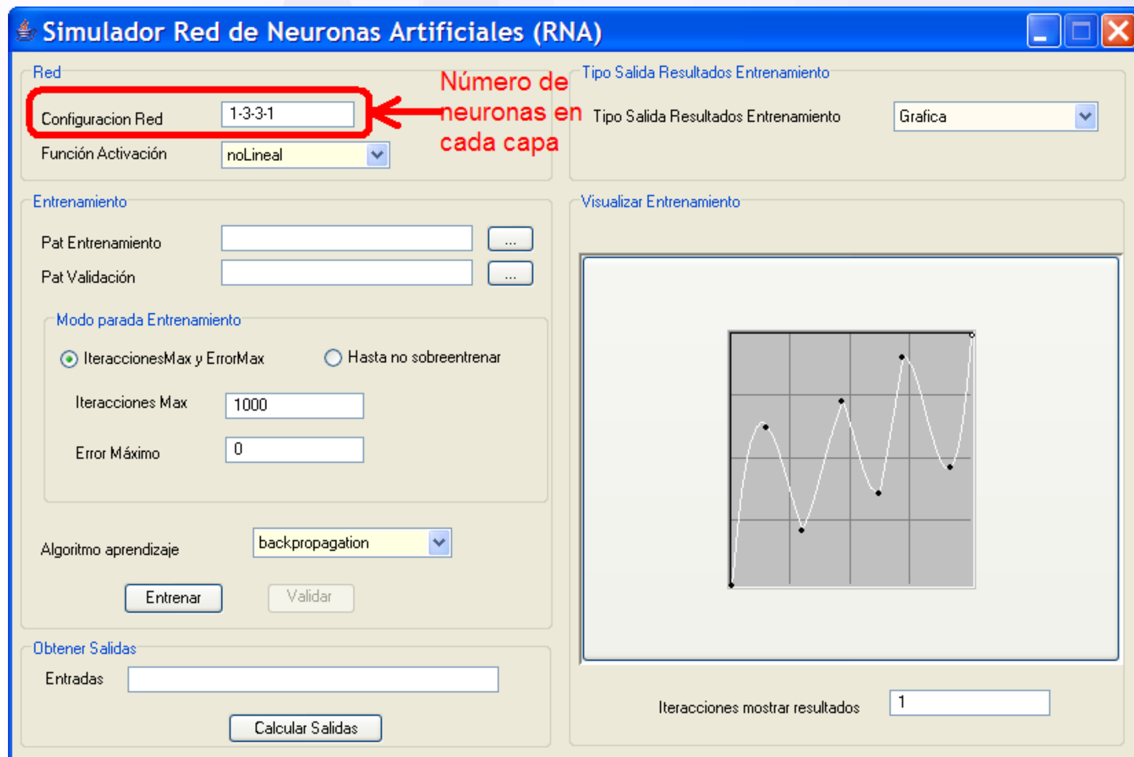


Figura 129. *Topología de la red*

Primeramente se debe elegir la topología que tendrá la red con la que queremos trabajar, asegurándonos de que el número de entradas y de salidas concuerda con las que hemos establecido en los ficheros de entrenamiento y validación. A continuación se debe indicar la función de activación para cada neurona, normalmente es preferible usar la función sigmoidal "*noLineal*", mientras que la función "escalón" se suele usar para realizar clasificación.

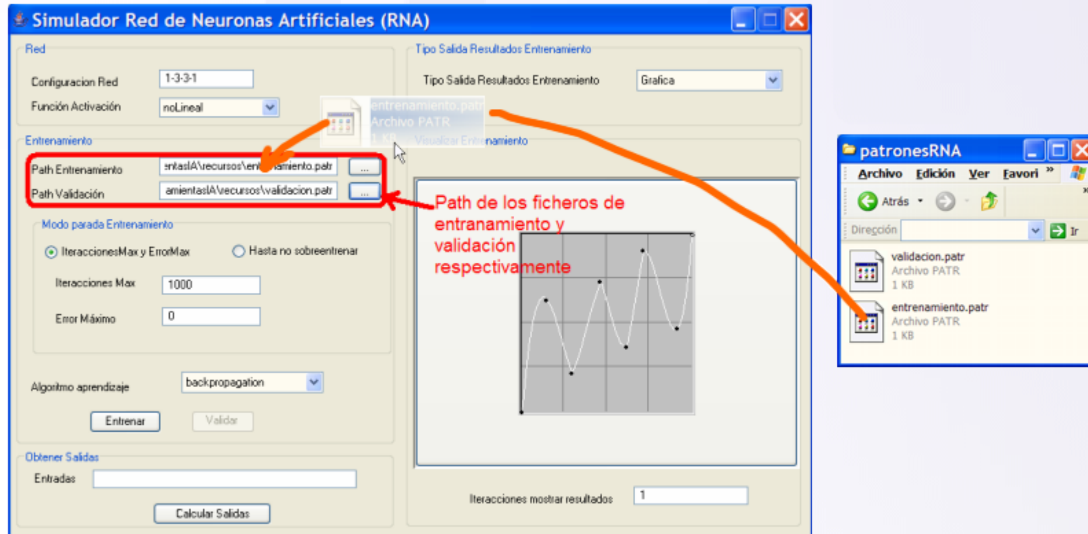


Figura 130. *Ficheros de entrenamiento y validación*

Lo siguiente a realizar es indicar donde se encuentran los ficheros de entrenamiento y los de validación (esté sólo es necesario si se quiere realizar también validación). Para indicar la ruta de los ficheros, hay varias maneras de introducirla:

- Escribiendo directamente la ruta a mano del directorio
- Pinchando en el archivo y arrastrándolo hasta el recuadro (drag&drop).
- Pinchando en el botón "…" y seleccionando el archivo en la nueva ventana que aparece.

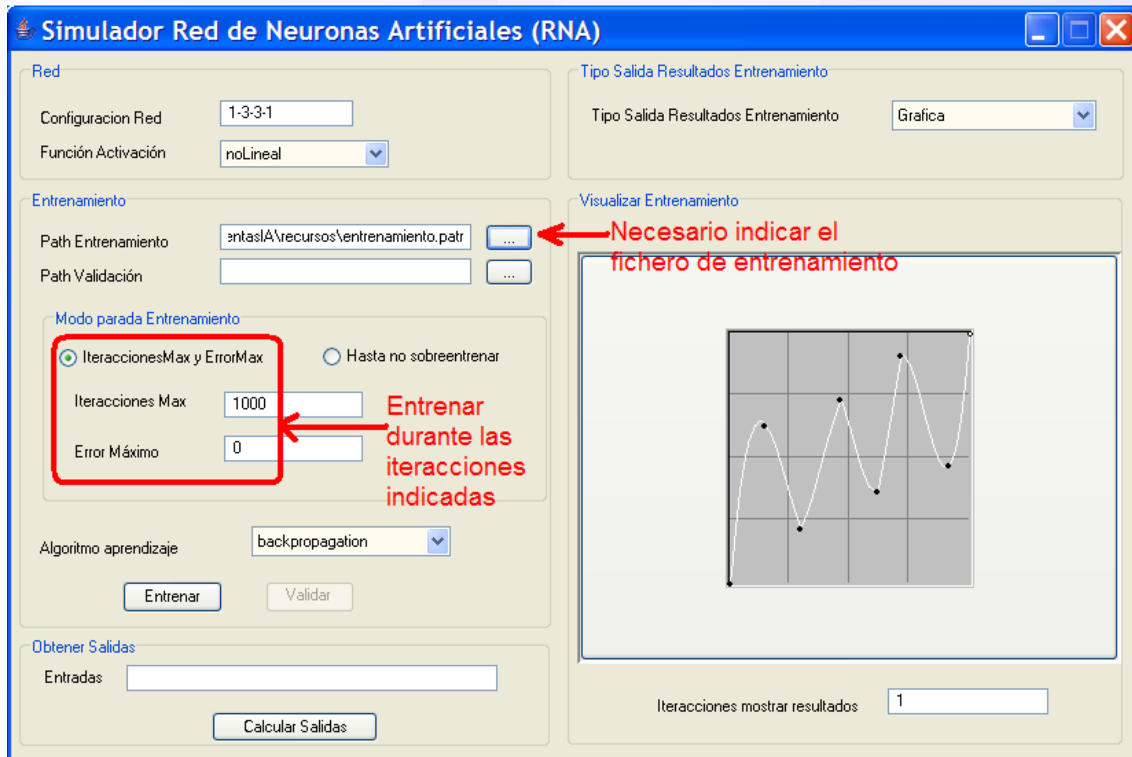


Figura 131. *Modo entrenamiento clásico*

Se deberá elegir que tipo de entrenamiento queremos realizar, si elegimos el modo "*IteraccionesMax y ErrorMax*" deberemos indicar durante cuantas iteraciones se quiere entrenar y cual es el error máximo permitido. Si por el contrario, elegimos el modo de entrenamiento "*hasta no sobrentrenar*" deberemos cerciorarnos de que hemos introducido el path del fichero de validación, pero no deberemos seleccionar el algoritmo de aprendizaje, puesto que únicamente funciona con backpropagation.

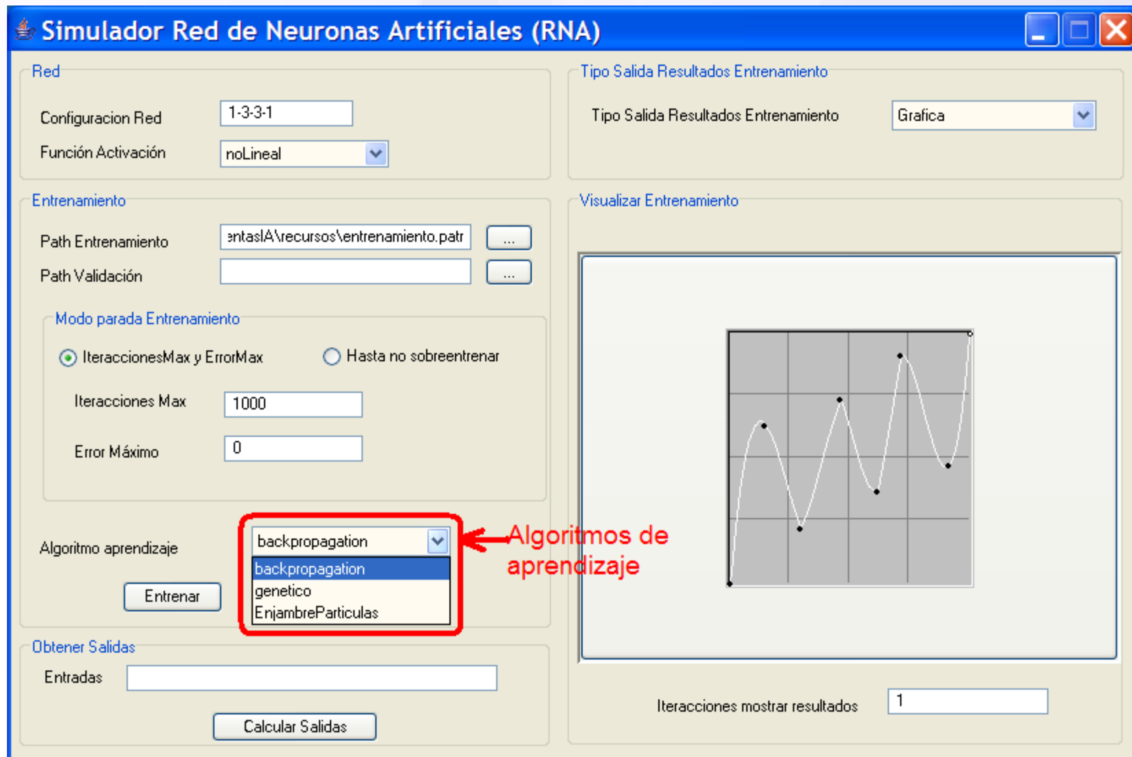


Figura 132. *Algoritmos de aprendizaje*

Si se ha seleccionado el modo de entrenamiento clásico se podrá escoger entre realizar el aprendizaje con backpropagation, alg. genéticos o enjambres de partículas.

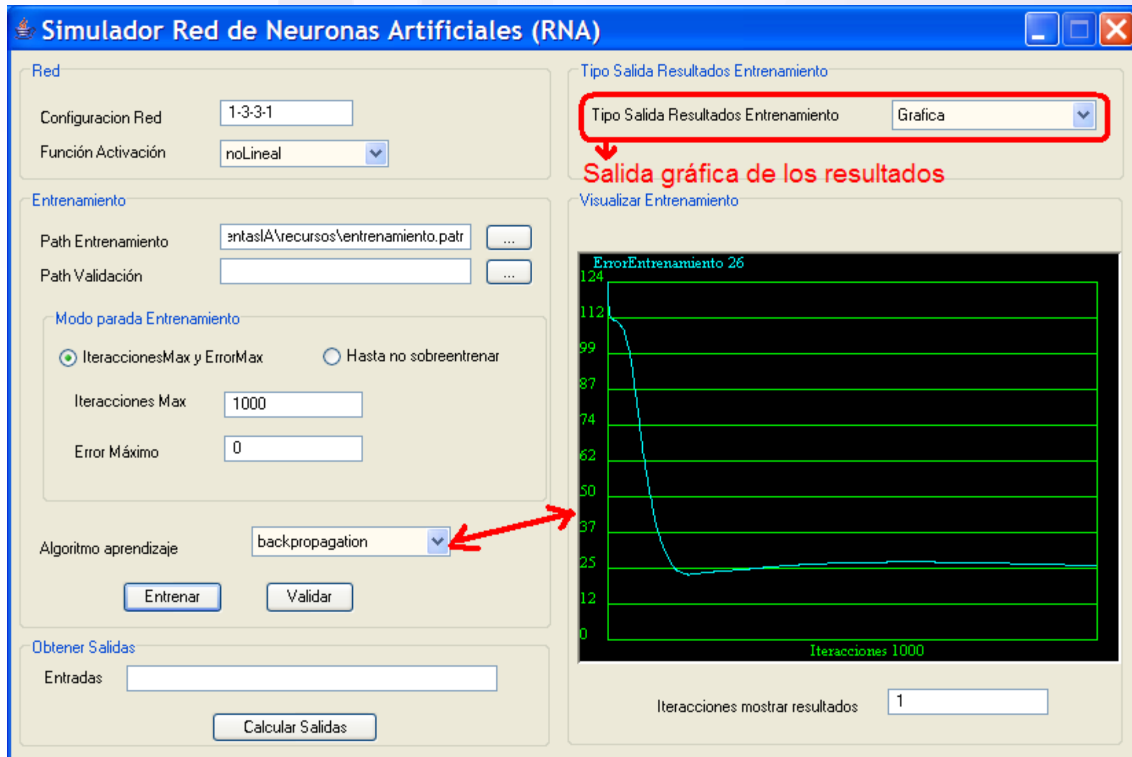


***Figura 133.** Gráfica del entrenamiento por backpropagation*

Si se ha el algoritmo de aprendizaje de backpropagation, se puede escoger que la salida sea en modo gráfico, para ver gráficamente como evoluciona el entrenamiento y/o la validación simultánea.

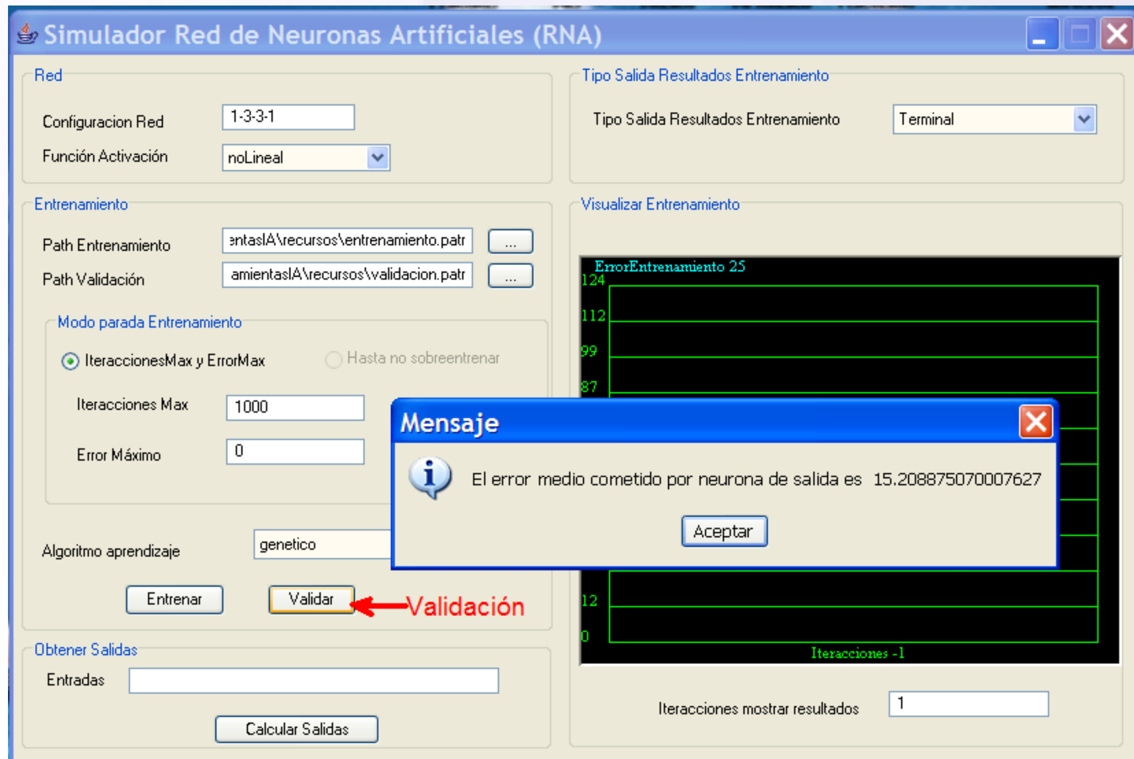


Figura 134. *Validando la red*

Una vez se ha entrenado la red, se puede validar, para ello se debería haber indicado donde se encuentra el fichero de validación y pinchar en validar. En un dialogo se mostrará el error medio que se comete por cada neurona de salida (sin normalizar, es decir el error real). Para el caso de la figura, como sólo se tiene una neurona de salida, es el error en media que se comete al aproximar la función cuadrado, que es para la que se ha realizado el entrenamiento.

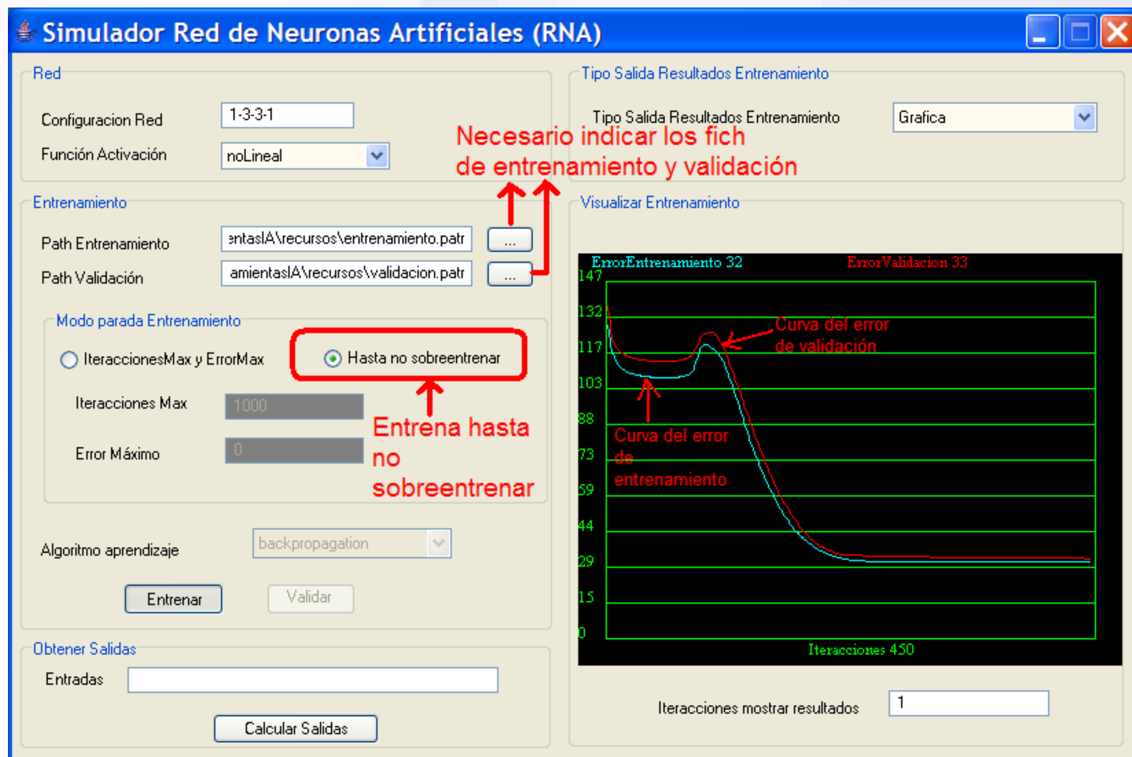


Figura 135. *Modo de entrenamiento y validación simultáneos*

El modo de entrenamiento "*Hasta no sobrentrenar*", como su nombre dice, trata de entrenar la red hasta el punto óptimo en el que no se sobrentrene, es decir, entrenar la red hasta que el error de validación deje de disminuir y empiece a crecer. Para ello lo que se hace es generar simultáneamente las curvas de entrenamiento y validación (entrenamiento y validación simultáneos) de tal manera que sólo se para el entrenamiento si el error de entrenamiento está disminuyendo y el error de validación aumenta durante un conjunto de iteraciones. Notar que si el error de entrenamiento aumenta y el error de validación también aumenta no se deberá de parar el entrenamiento, puesto que únicamente se está intentando salir de un mínimo local.

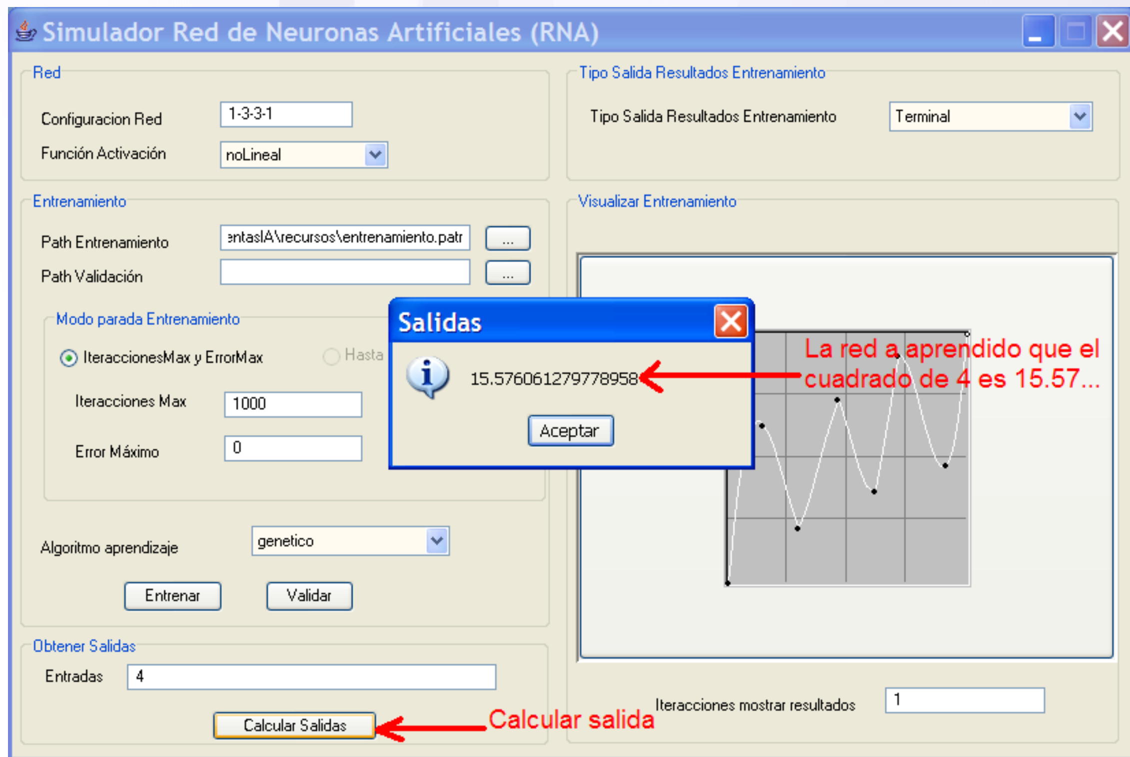


Figura 136. *Calcular salidas*

Una vez entrenada la red, podemos usarla para su verdadero fin, que es pasarle cualquier tipo de entrada y obtener sus respectivas salidas. Esto se puede hacer directamente desde la gráfica introduciendo las entradas sin normalizar y pinchando en calcular salidas que devolverá las salidas también sin normalizar. En la figura se puede ver, que la red dice que el cuadrado de 4 es 15.57… que se aproxima bastante a la solución real.

6.6.2 Ejecutando la red sin la interfaz gráfica

```
int[] configuracion = { 2,3,1 };
String funcionActivacion = "noLineal";

//creando la red sin indicar valores máximos y mínimos para la normalización
Red rna = new Red(configuracion,funcionActivacion);

//creando la red indicándole valores máximos y mínimos para la normalización
double[] valorMayorEnt = {1};
 double[] valorMenorEnt = {0};
double[] valorMayorSal = {1};
double[] valorMenorSal = {0};
Red rna = new Red(configuracion,funcionActivacion,
                    valorMayorEnt,valorMenorEnt,
                    valorMayorSal,valorMenorSal);
```

Figura 137. *Distintos modos de crear la red*

```
//aquí realizamos el entrenamiento con Enjambres de Partículas
long iteraciones = 200;
double errorMax = 0.05;
double[][]  entradas =         { {1,1},{1,0},{0,1},{0,0}};
double[][]  salidasEsperadas =  { {0},{1},{1},{0}};
rna.entrenar(entradas,salidasEsperadas,errorMax, iteraciones,1,Red. ENJAMBRE);

//aquí realizamos el entrenamiento con Algoritmos Genéticos
long iteraciones = 200;
double errorMax = 0.05;
double[][]  entradas =         { {1,1},{1,0},{0,1},{0,0}};
double[][]  salidasEsperadas =  { {0},{1},{1},{0}};
rna.entrenar(entradas,salidasEsperadas,errorMax, iteraciones,1,Red.GENETICO);

//aquí realizamos el entrenamiento con backpropagation
long iteraciones = 200;
double errorMax = 0.05;
double[][]  entradas =         { {1,1},{1,0},{0,1},{0,0}};
double[][]  salidasEsperadas =  { {0},{1},{1},{0}};
rna.entrenar(entradas,salidasEsperadas,errorMax,
iteraciones,1,Red.BACKPROPAGATION);

//aquí realizamos el entrenamiento con backpropagation y hasta no sobrentrenar
double[][]  entradasEntrenamiento =        { {1,1},{1,0},{0,1},{0,0}};
double[][]  salidasEntrenamiento =  { {0},{1},{1},{0}};
double[][] entradasValidacion = {{1,1},{0,1}};
double[][] salidasValidacion = {{0},{1}};
rna.entrenarHastaNoSobreentrenar(entradasEntrenamiento,
                                 salidasEntrenamiento,
                                 entradasValidacion,
                                 salidasValidacion,
                                 1);
```

Figura 138. *Distintos modos de entrenar la red*

```
import RNA.*;

 //estructura de la red
  int[] configuracion = { 2,3,1 };
String funcionActivacion = "noLineal";
Red rna = new Red(configuracion,funcionActivacion);

 //patrones de entrenamiento
 double[][]  entradas =          { {1,1},{1,0},{0,1},{0,0}};
 double[][]  salidasEsperadas =  { {0},{1},{1},{0}};

 //entrenamiento
rna.entrenar(entradas,salidasEsperadas,errorMax,
iteraciones,1,Red.BACKPROPAGATION);

//validacion
double[] entradaValidacion1 = {1,1};
double salida[] = rna.validarRed(entradaValidacion1);
System.out.println("La red dice que el xor de 1,1 es: " +  salida[0]);
double[] entradaValidacion2 = {0,1};
salida = rna.validarRed(entradaValidacion2);
System.out.println("La red dice que el xor de 0,1 es: " +  salida[0]);
double[] entradaValidacion3 = {1,0};
salida = rna.validarRed(entradaValidacion3);
System.out.println("La red dice que el xor de 1,0 es: " +  salida[0]);
double[] entradaValidacion4 = {0,0};
salida = rna.validarRed(entradaValidacion4);
System.out.println("La red dice que el xor de 0,0 es: " +  salida[0]);
```

Figura 139. *Usando la RNA directamente desde el código para aprender la función XOR*

7. Conclusiones

8. Bibliografía

Papers

http://es.geocities.com/matlabudes/ponenciarna.doc

http://personal5.iddeo.es/wintrmute/ia/neuronal.htm

http://es.wikipedia.org/wiki/Simulated_annealing

http://www.tesisenxarxa.net/TESIS_UC/AVAILABLE/TDR-0305107-180847//5de8.JRPL_cap5.pdf

http://neo.lcc.uma.es/staff/jmgn/doc/Memoria_PFC_JMGN.pdf

http://www.cnc.una.py/cms/invest/download.php?id=778622,136,4

http://www.cnc.una.py/cms/invest/download.php?id=265444,61,1

http://www.cnc.una.py/cms/invest/download.php?id=250344,48,1

Literatura escrita