**TÍTULO:**

# OPTIMIZACIÓN DE MEMORIA DINÁMICA EN SISTEMAS EMPOTRADOS MULTIMEDIA MEDIANTE COMPUTACIÓN EVOLUTIVA.

**AUTORES:**

Sergio Belmar Argudo
César M. González Iñiguez
Pablo Virseda Benito

**PROFESORES:**

José Ignacio Hidalgo Pérez
David Atienza Alonso

**CURSO:**

2005 / 2006

Proyecto de Sistemas Informáticos, Facultad de Informática, Universidad Complutense de Madrid

Agradecimientos:

*Queremos agradecer, por un lado, tanto a José Ignacio Hidalgo (Iñaki) como a David Atienza por guiarnos a lo largo de este proyecto y ser pacientes con nosotros, así como al IMEC en la persona de Christophe Poucet por su ayuda con las aplicaciones multimedia utilizadas como casos de estudio en esta memoria.*

*También queremos agradecer al encargado de asignar los proyectos de Sistemas Informáticos en el curso 2005/2006 por asignarnos un proyecto que no estaba en la lista de los 30 escogidos por nosotros.*

*Y por supuesto, a todos nuestros familiares y amigos, por soportar nuestras ausencias prolongadas por tiempos indefinidos, y sobre todo a Marta, Joana y Ana.*

**Índice:**

Capítulo 0: Resumen del Proyecto. ....- 4 -
0.1- Resumen en Castellano ....- 4 -
0.2- English summary....- 5 -
Capítulo 1: Introducción.....- 6 -
Capítulo 2: Optimización de memoria dinámica en sistemas empotrados multimedia......- 9 -
2.1- Situación actual de los sistemas empotrados multimedia. ....- 9 -
2.1.1- Tipos de sistemas empotrados.....- 9 -
2.1.2- Características de los sistemas empotrados multimedia. ....- 10 -
2.2- Arquitecturas Hardware. ....- 12 -
2.3- Arquitecturas Software.....- 14 -
Capítulo 3: Computación Evolutiva. ....- 16 -
3.1- Algoritmos Genéticos Simples.....- 16 -
3.1.1- Población inicial.....- 17 -
3.1.2- Función de coste.....- 18 -
3.1.3- Codificación de las soluciones. ....- 18 -
3.1.4- Operador de selección. ....- 19 -
3.1.5- Operador de cruce. ....- 20 -
3.1.6- Operador de mutación. ....- 21 -
3.1.7- Tamaño de la población. ....- 21 -
3.2- Algoritmos Genéticos Multiobjetivo.....- 22 -
3.2.1- VEGA....- 23 -
Capítulo 4: Datos de Entrada....- 26 -
4.1- Metodología de Optimización.....- 26 -
4.1.1- Partición algorítmica. ....- 27 -
4.1.2- Exploración de las estructuras dinámicas de datos. ....- 27 -
4.1.3- Exploración automática de las estructuras dinámicas de datos.....- 28 -
4.2- Biblioteca de estructuras de datos dinámicas.....- 30 -
4.3- Resultados de los Perfiles.....- 38 -
Capítulo 5: Nuestra Aplicación. ....- 45 -
5.1- Desarrollo de la aplicación.....- 45 -
5.2- Funcionamiento de la aplicación....- 51 -
Capítulo 6: Datos de Salida. ....- 54 -
6.1- Significado de los valores. ....- 54 -
Capítulo 7: Un Ejemplo.....- 56 -
7.1- Optimización de “VDrift”. ....- 56 -
7.2- Resultados en otras aplicaciones. ....- 66 -
Capítulo 8: Conclusiones.....- 67 -
Objetivos conseguidos.....- 67 -
Mejoras futuras.....- 70 -
Anexo I: Estructura de clases de la aplicación. ....- 71 -
Parte gráfica:....- 72 -
Motor genético: ....- 73 -
Anexo II: Listado de palabras para la búsqueda bibliográfica. ....- 75 -
Anexo III: Autorización de difusión.....- 76 -
Bibliografía:....- 77 -

# Capítulo 0: Resumen del Proyecto.

## *0.1- Resumen en Castellano*

*Nos encontramos a las puertas de la cuarta generación de telefonía móvil. Esta revolución tecnológica integrará en un único dispositivo sumamente compacto, una gran cantidad de aplicaciones, que van desde las propias de un teléfono como tal, hasta otras que podrían encontrarse instaladas en un PC de sobremesa, como por ejemplo, sistemas de videoconferencia, reproductores de vídeo y sonido, juegos en tres dimensiones, programas de retoque fotográfico, sintonizadoras de televisión y un largo y variado etcétera.*

*Estos dispositivos, a pesar de que el avance tecnológico en cuanto a integración de circuitos integrados ha sido enorme y ha conseguido procesadores idóneos para el tipo de aplicaciones requeridas, siguen adoleciendo de ciertas limitaciones. Los mayores inconvenientes surgen del hecho de que las aplicaciones usadas en los sistemas empotrados suelen provenir de sistemas de sobremesa, con diferentes restricciones, entre las que destaca el uso de memoria, en particular la memoria dinámica (actualmente un ordenador de sobremesa posee entre 512 y 1024 MB de memoria RAM, frente a los 32 o 64 con los que puede contar una PDA de última generación).*

*Es por ello que una de las principales tareas en el proceso de adaptación de aplicaciones informáticas para sistemas empotrados multimedia es la optimización de este sistema de memoria dinámica, eligiendo adecuadamente las Estructuras de Datos Dinámicas (en adelante, EDDs) que mejor se adapten a las necesidades de la aplicación y del sistema.*

*Para optimizar el uso de memoria dinámica, el programador habitualmente dispondrá de una biblioteca de EDDs (arrays dinámicos, listas enlazadas, etc.) entre las que se debe escoger dependiendo de nuestras necesidades o restricciones en cuanto a rendimiento, uso de memoria y consumo de energía.*

*Normalmente, ésta tarea se realiza mediante la ejecución de una búsqueda exhaustiva para la obtención de un frente de Pareto (conjunto de puntos óptimos de alguna de la métricas que se desean optimizar, como el rendimiento, el consumo energético o el uso de memoria) que abarque todas las posibilidades sobre las que realizar la elección final. Está búsqueda es efectiva pero presenta el inconveniente del alto coste computacional.*

*El propósito de nuestro proyecto ha sido la implementación de un algoritmo genético multiobjetivo (del tipo VEGA), en el que dada una aplicación para un sistema empotrado a optimizar, devuelva las mejores estructuras de datos dinámicas para las correspondientes variables dinámicas de dicha aplicación. Para ello, nos hemos basado en los resultados obtenidos en los perfiles de ejecución de aplicaciones reales y de resultados teóricos sobre las implementaciones de dichas estructuras de datos. Los datos teóricos se han obtenido aplicando la metodología desarrollada por el profesor David Atienza en su tesis doctoral "Metodología multinivel de refinamiento del subsistema de memoria dinámica para los sistemas empotrados multimedia de altas prestaciones". Como podremos comprobar a lo largo del desarrollo de esta memoria, los resultados obtenidos demuestran la validez de nuestra aproximación.*

## *0.2- English summary*

*Presently, we are looking forward to achieve, among others technologic revolutions, the fourth generation of mobile telephony. In this forthcoming technology, a great amount of applications will be integrated in a extremely compact device. These applications do not exclusively belong to the field of mobile devices, but at the same time, they could be installed in a desktop PC. Representative examples of such applications are videoconference systems, video and audio players, 3D games, photographic editors, television tuners, etcetera.*

*These devices, although the technological progress has been impressive, as far as circuit integration is concerned, and has enabled the production of suitable embedded processors for the type of applications to be desired in such devices, overall designs keep suffering from certain limitations. Moreover, high-performance embedded systems struggled to execute these complex applications because they usually come from desktop systems, with very different restrictions regarding memory use features, and more concretely not concerned with an efficient use of the dynamic memory. In fact, a desktop computer typically includes between 512 and 1024 MB of RAM memory, as opposed to the 32 or 64 MB present in last-generation PDAs). Therefore, one of the main tasks to be developed during the porting process of multimedia applications for embedded multimedia systems is the optimization of the dynamic memory subsystem. To this end, it is required to suitably choose the Dynamic Data Types (DDTs, from now on) according to the specific application and final system requirements.*

*In order to optimize the use of dynamic memory, we possess a library of DDTs (dynamic arrays, linked lists, etc.) from where we can select the best one in each case according to the desired ultimate necessities or restrictions of the metrics: performance, memory use and energy consumption.*

*Normally, this task is carried out by performing an exhaustive evaluation of the design space of DDT implementations (i.e., multiple executions) for the application to attain the Pareto's front, which is the set of optimal implementation points for the aforementioned different metrics that to be optimized (e.g., performance, power consumption or memory use). This Pareto's front must include all the optimal possibilities among which the final selection can be made.*

*The objective of our project has been the implementation of a multi-objective genetic algorithm (i.e., VEGA type), where given an application to be optimized for a certain embedded system, it returns the best DDT for each of included dynamic variable in the target application. The developed framework makes use of application profiling and performs the exploration using as basis to prune the design space the theoretical modelling on the implementations of the DDTs presented in the methodology developed by Prof. David Atienza in his thesis, entitled "Multi-Level Refinement Methodology of the Dynamic Memory Subsystem for High-Performance Multimedia Embedded Systems".*

# Capítulo 1: Introducción.

Nos encontramos a las puertas de la cuarta generación de telefonía móvil. Esta revolución tecnológica integrará en un único dispositivo sumamente compacto, una gran cantidad de aplicaciones, que van desde las propias de un teléfono como tal, hasta otras que podrían encontrarse instaladas en un PC de sobremesa, como por ejemplo, sistemas de videoconferencia, reproductores de vídeo y sonido, juegos en tres dimensiones, programas de retoque fotográfico, sintonizadoras de televisión y un largo y variado etcétera. No sólo en los nuevos teléfonos móviles tenemos esta combinación de multimedia y alta integración, sino que las PDAs (Personal Digital Assistant) o las videoconsolas portátiles son otro claro exponente de sistemas empotrados multimedia.

Estos dispositivos, a pesar de que el avance tecnológico en cuanto a integración de circuitos integrados ha sido enorme y ha conseguido procesadores idóneos para el tipo de aplicaciones requeridas, siguen adoleciendo de ciertas limitaciones. Los mayores inconvenientes surgen del hecho de que las aplicaciones usadas en los sistemas empotrados suelen provenir de sistemas de sobremesa, con diferentes restricciones, entre las que destaca el uso de memoria, en particular la memoria dinámica (actualmente un ordenador de sobremesa posee entre 512 y 1024 MB de memoria RAM, frente a los 32 o 64 con los que puede contar una PDA de última generación).

Es por ello que una de las principales tareas en el proceso de adaptación de aplicaciones informáticas para sistemas empotrados multimedia es la optimización de este sistema de memoria dinámica, eligiendo adecuadamente las Estructuras de Datos Dinámicas (en adelante, EDDs) que mejor se adapten a las necesidades de la aplicación y del sistema.

Para optimizar el uso de memoria dinámica, el programador habitualmente dispondrá de una biblioteca de EDDs (arrays dinámicos, listas enlazadas, etc.) entre las que se debe escoger dependiendo de nuestras necesidades o restricciones en cuanto a rendimiento, uso de memoria y consumo de energía.

Normalmente, ésta tarea se realiza mediante la ejecución de una búsqueda exhaustiva para la obtención de un frente de Pareto (conjunto de puntos óptimos de alguna de la métricas que se desean optimizar, como el rendimiento, el consumo

energético o el uso de memoria) que abarque todas las posibilidades sobre las que realizar la elección final. Está búsqueda es efectiva pero presenta el inconveniente del alto coste computacional.

Los algoritmos evolutivos han demostrado ser una herramienta eficiente para resolver problemas complejos de optimización en tiempos aceptables por las aplicaciones. El problema que tratamos este trabajo es claramente un problema multi-objetivo. Durante los últimos cinco años se han desarrollado una gran cantidad de algoritmos evolutivos multi-objetivo [1] que encuentras frentes de Pareto para problemas de gran dificultad. Dentro de éstos, uno de los primeros que demostró ser eficiente fue la aproximación propuesta por Schaffer [2] para resolver problemas multi-objetivo mediante un Algoritmo Genético Evaluado por Vectores (VEGA).

El propósito de nuestro proyecto ha sido la implementación de un algoritmo genético multiobjetivo (del tipo VEGA), en el que dada una aplicación para un sistema empotrado a optimizar, devuelva las mejores estructuras de datos dinámicas para las correspondientes variables dinámicas de dicha aplicación. Para ello, nos hemos basado en los resultados obtenidos en los perfiles de ejecución de aplicaciones reales y de resultados teóricos sobre las implementaciones de dichas estructuras de datos. Los datos teóricos se han obtenido aplicando la metodología desarrollada por el profesor David Atienza en su tesis doctoral *"Metodología multinivel de refinamiento del subsistema de memoria dinámica para los sistemas empotrados multimedia de altas prestaciones"* [3]. Como podremos comprobar a lo largo del desarrollo de esta memoria, los resultados obtenidos demuestran la validez de nuestra aproximación.

El resto de la memoria está organizada como sigue.

En el Capítulo 2 hacemos una introducción acerca de los sistemas empotrados multimedia y describimos su arquitectura tanto hardware como software.

En el Capítulo 3 explicamos las bases de la computación evolutiva comenzando desde los algoritmos genéticos simples, repasando todos sus operadores, hasta llegar los algoritmos multi-objetivo explicando en más detalle la implementación VEGA, que ha sido usada en este proyecto.

En el Capítulo 4 explicamos los datos de entrada de nuestra aplicación, los cuales tienen su origen en datos teóricos obtenidos de la tesis de David Atienza y en datos experimentales sacados a partir de el programa a optimizar.

En el Capítulo 5 comentamos el diseño y funcionamiento de nuestra aplicación, así como las fases que hemos seguido en su desarrollo.

En el Capítulo 6 vemos el significado práctico y teórico de los elementos de salida de nuestra aplicación.

En el Capítulo 7 hacemos un ejemplo paso a paso y así vemos más en detalle el funcionamiento del programa. También comentamos otros ejemplos de optimización con los que hemos trabajado a lo largo del proyecto.

El Capítulo 8 está dedicado a las conclusiones obtenidas tras la realización del proyecto, así como futuras líneas abiertas para la investigación y mejora en este campo.

Completamos esta memoria con anexos sobre la estructura de clases de la aplicación, palabras para la búsqueda bibliográfica y autorización de difusión, terminando con la bibliografía usada para la realización de este proyecto.

# Capítulo 2: Optimización de memoria dinámica en sistemas empotrados multimedia.

## *2.1- Situación actual de los sistemas empotrados multimedia.*

Está claro que los sistemas informáticos de sobremesa han experimentado una profunda evolución a lo largo de los últimos casi 35 años de existencia de los microprocesadores [4], de tal manera que en la actualidad, unas computadoras con unos diseños cada vez más compactos y atractivos, son capaces, con total tranquilidad, de realizar complejas tareas (correo electrónico, servicios multimedia, edición de vídeo, edición de audio, grabación de CD's y DVD's, aplicaciones de diseño infográficas, juegos en 3D, etc. ) que hace pocos años sólo estaban en manos de grandes y caras estaciones de trabajo.

Además de esta evidente mejoría de estos productos de sobremesa, las nuevas tecnologías han ampliado las prestaciones y capacidades de los sistemas dedicados (sistemas empotrados) [5], que han pasado de ser dispositivos de control diseñados para realizar funciones básicas y especificas en entornos fijos, al ser sistemas enormemente complicados, con funcionalidades casi comparables a sistemas de sobremesa en entornos heterogéneos, con unos fuertes requisitos (ligaduras de diseño) [3] en cuanto a consumo energético u ocupación de memoria se refiere.

### 2.1.1- Tipos de sistemas empotrados.

Actualmente, podemos encontrarnos con 2 principales tipos de sistemas empotrados:

**a)** Los sistemas más tradicionales, integrados en nuestra sociedad desde hace bastante tiempo en prácticamente cualquier dispositivo electrónico que manejemos en nuestra casa o que veamos en nuestro entorno, realizando tareas determinadas y prefijadas (con diferentes grados de complejidad), desde una televisión hasta un automóvil [3].

**b)** Los sistemas empotrados de nueva generación, con más funcionalidad que los anteriores y orientados directamente a servicios de red y multimedia, como pueden ser los teléfonos móviles de 4ª generación, PDA's, consolas portátiles, etc [3].

Estos sistemas, además de realizar tareas más complejas (por norma general) que los sistemas tradicionales, tienen la dificultad de tener que ejecutar una gran variedad de aplicaciones (por ejemplo, un teléfono móvil de ultima generación puede realizar funciones como videoconferencia, reproducción de audio mp3 y de video mp4, videojuegos en tres dimensiones, etc.), haciendo de estos sistemas equivalentes en prestaciones a entornos de sobremesa, pero con unas ligaduras de diseño e implementación mucho más restrictivas.

### 2.1.2- Características de los sistemas empotrados multimedia.

A continuación, pasamos a enumerar las principales características de los sistemas empotrados multimedia, sobre los que se centra el trabajo de este proyecto:

**a)** Restricciones severas en varias ligaduras de diseño (energía, rendimiento y tamaño). Los sistemas empotrados poseen unas ligaduras de diseño claramente mayores que los sistemas de sobremesa, en los que prima el rendimiento sobre otros factores como pueden ser el consumo de energía o el tamaño de la memoria del dispositivo, factores extremadamente determinantes en el diseño de dispositivos empotrados multimedia, primando en determinados casos sobre, por ejemplo, el rendimiento obtenido. [3]

**b)** Altamente reactivos y con restricciones en tiempo real. A diferencia de un ordenador de sobremesa, un sistema empotrado debe reaccionar a cambios en el entorno que le rodea (temperatura, presión, acciones del usuario) y al mismo tiempo realizar sus cálculos en tiempo real. [3]

**c)** Rango de funcionalidades extenso, pero acotado. Los nuevos sistemas empotrados multimedia de altas prestaciones deben ser capaces de ofrecer un amplio rango de servicios multimedia a los usuarios (reproducción de audio/video, juegos, etc.),

aunque este rango de aplicaciones es acotado y generalmente conocido en las etapas de diseño [3].

Además de estas características, no debemos olvidarnos de **la importancia del mercado** en la sociedad actual. Los sistemas empotrados de nueva generación están sometidos al mercado, lo que implica desarrollos con tiempos cada vez menores y cambiantes. Así, el tiempo de desarrollo de estos productos para lograr su introducción en el mercado de manera óptima se calcula actualmente en unos 8 meses [5], teniendo en cuenta que un retraso, por muy pequeño que sea, puede tener unas consecuencias económicas desastrosas para los fabricantes. También es imprescindible la ausencia de defectos en el momento del lanzamiento, ya que solventar esos fallos una vez que el dispositivo ya está en manos del usuario tiene unos costes económicos y de imagen de marca enormes.

*Así pues, parece necesario que las empresas desarrolladoras posean metodologías y herramientas de diseño e implementación que permitan el desarrollo de estos sistemas empotrados, tanto a nivel hardware como software, en el menor tiempo posible y con la calidad adecuada.*

## *2.2- Arquitecturas Hardware.*

Características principales de la arquitectura HW de los sistemas empotrados de altas prestaciones:

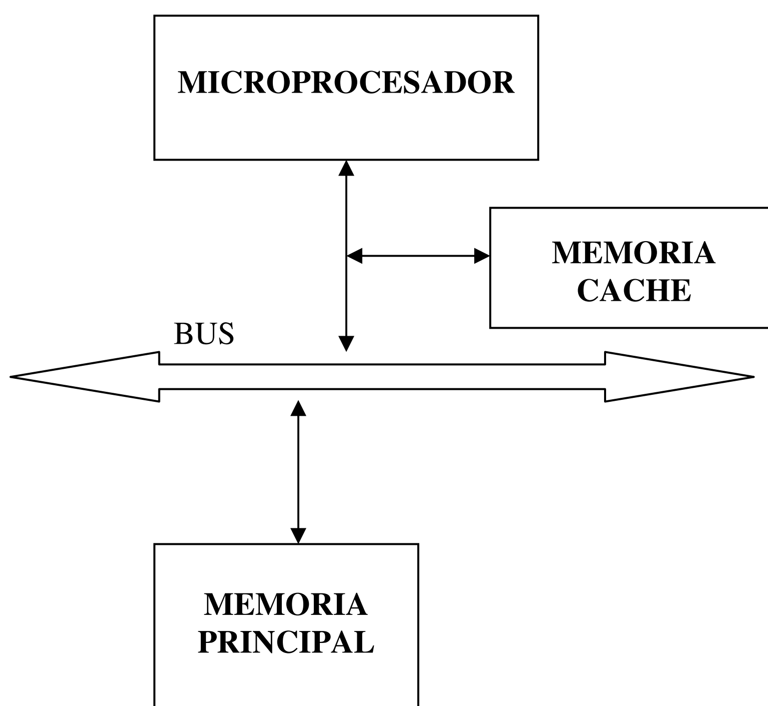
**a)** Diseños basados en Sistemas-En-Chip (SoC en inglés). Dada la dificultad de diseñar desde cero sistemas con las necesidades tan heterogéneas requeridas en la actualidad, la alternativa más viable a la hora de afrontar un nuevo diseño actualmente es el diseño global mediante la composición y reutilización de módulos o componentes diseñados de forma independiente [3].

**b)** Sistemas optimizados para bajo consumo. Los nuevos sistemas empotrados incluyen microprocesadores especialmente diseñados para lograr un consumo de energía muy reducido, soportando a la vez frecuencias de trabajo cada vez mayores, como resultado de un enorme esfuerzo en lo que a optimización se refiere en todos los niveles del procesador, como pueden ser el repertorio de instrucciones multimedia, el ajuste dinámico del voltaje y de la frecuencia de trabajo, la disminución de la energía necesaria para la activación y trabajo con los bancos de registros, mejoras en los predictores de ejecuciones y direcciones para las instrucciones de salto, optimización en los procesos de precompilación, etc [6].

**c)** Jerarquía de memoria multinivel optimizada. Para reducir el consumo y a la vez aumentar el rendimiento del sistema, el subsistema de memoria de los sistemas empotrados multimedia incluye una estructura multinivel con al menos un nivel de caché y una memoria principal compartida. En la actualidad, además de aumentar el número de niveles de caché (L2 o memoria caché de segundo nivel), se usan pequeñas memorias adicionales cercanas al procesador y con control software (Scratchpad) [7] que mejoran enormemente el rendimiento.

*Teniendo en cuenta la importancia del subsistema de memoria, el problema de su optimización ha sido considerado desde hace más de una década como uno de los problemas más importantes dentro del entorno de sistemas empotrados de alto rendimiento* [8] [9].

Para la implementación de nuestra aplicación, hemos considerado la siguiente arquitectura de memoria para el sistema objetivo, para el cual estamos realizando la optimización. Se puede ver que sólo hay un nivel de memoria caché (por defecto, 32KBytes, pero es un valor configurable en nuestro entorno) y que en caso de fallo es necesario acceder a la memoria principal a través del bus para acceder a los datos deseados.



## *2.3- Arquitecturas Software.*

Características principales de la arquitectura SW de los sistemas empotrados de altas prestaciones:

**a)** Doble vertiente en la componente software. La componente software ha de dar soporte a dos servicios diferentes en cuanto a requisitos y características [3]:

- Debe utilizar las optimizaciones a nivel hardware para cumplir con las restricciones de consumo y rendimiento.

- Debe suministrar una interfaz clara que permita reducir el tiempo de implementación para dicha plataforma. Para ello debe ofrecer la posibilidad de usar lenguajes de alto nivel (como C++ o Java) que permitan a los diseñadores de software abstraerse de la arquitectura hardware subyacente del sistema.

**b)** Ejecución de un número variable de aplicaciones multimedia concurrentes. Aunque las aplicaciones y servicios suministrados por los sistemas empotrados están definidos en el momento del diseño, el uso de dichos servicios se determina de forma dinámica por el usuario. Esto provoca la ejecución concurrente de varias aplicaciones, por lo que es obligatorio analizar y ajustar de manera dinámica los recursos disponibles (memoria y procesador), manteniendo al mismo tiempo los requerimientos de rendimiento y tiempo real de las aplicaciones [3].

**c)** Aplicaciones de usuario con requerimientos dinámicos de uso de memoria. Las aplicaciones suministradas a usuarios finales tienen una complejidad muy alta, con información que no es completamente conocida en tiempo de compilación. Esto se debe a que dependen de ciertos valores en el momento de la ejecución, variando en el tiempo. Dicha variación crea la necesidad de dar soporte avanzado a la gestión de datos dinámicos (aquellos reservados, usados y liberados en tiempo de ejecución) y a optimizar su uso.

*Estas variaciones pueden producir cambios en los requerimientos del sistema de memoria de más de 2 órdenes de magnitud, lo que demuestra la importancia del uso óptimo de la memoria dinámica* [10].

# Capítulo 3: Computación Evolutiva.

## *3.1- Algoritmos Genéticos Simples.*

Los algoritmos genéticos (AGs) [11] son procedimientos de búsqueda y optimización que tienen sus orígenes e inspiración en el mundo biológico. Se caracterizan por emular los comportamientos evolutivos de la naturaleza y se basan en la supervivencia del mejor individuo, siendo un individuo una solución potencial del problema que se implementa como una estructura de datos (en un AG simple, una cadena de bits o genes). Trabajan sobre poblaciones de soluciones que evolucionan de generación en generación mediante operaciones entre ellas (operadores genéticos) adaptadas al problema. Los parámetros que los controlan son variables en función de como se represente a los genes, del tipo de estructura de datos que implementa una solución, del tipo de operadores y de si éstos parámetros son variables o constantes. Según estos factores nos encontramos con un tipo de estrategia u otra.

Los algoritmos genéticos simples se basan en un principio básico de la evolución: los mejores individuos tienen una mayor probabilidad de reproducirse y sobrevivir que otros individuos menos adaptados al entorno [12]. Para implementar este principio, los algoritmos genéticos mantienen una población que evoluciona a través del tiempo y que al final convergen a una única solución. Los individuos de la población se representan mediante un cromosoma que codifica las variables del problema que se quiere resolver.

Como ya hemos dicho, normalmente se utiliza un cromosoma simple compuesto por una cadena de bits de longitud fija, pero existen algoritmos genéticos que codifican el problema utilizando varios cromosomas para representar una solución y otros que utilizan cromosomas de longitud variable. Cada individuo tiene asignado un valor de una función de coste, que mide la calidad de la solución y que es la herramienta que permite simular el concepto de individuos mejor adaptados. Aquellos individuos cuyo valor de la función de coste sea mejor tendrían más posibilidades de ser seleccionados para construir la siguiente población y por lo tanto, para pasar sus propiedades a los individuos de la siguiente generación.

Para implementar un algoritmo genético es necesario definir:

- Una población inicial.
- Una función de coste que evalúe a los individuos.
- Una codificación que permita representar las soluciones.
- El operador de selección.
- El operador de cruce.
- El operador de mutación.
- El tamaño de la población.

Explicamos todo esto con un ejemplo sencillo:

### 3.1.1- Población inicial.

Para comenzar se genera una población inicial a partir de la que se trabaja. Esta población se suele inicializar de una manera aleatoria, aunque existen implementaciones en las que se obtiene mediante otros métodos como una búsqueda local o cualquier otro tipo de algoritmo si se pretende que la búsqueda se inicie en una determinada dirección del espacio de soluciones. En nuestro proyecto hemos generado la población inicial de forma totalmente aleatoria. Cuando se han obtenido estos individuos iniciales, se repite el proceso interno del algoritmo tantas veces como indique la condición de parada.

Dicho proceso comienza evaluando la población y seleccionando los individuos que intervendrán en la formación de la siguiente generación. Seguidamente se aplican los operadores de cruce y mutación y se obtiene la nueva población a partir de la que se repiten los mismos pasos para ir avanzando en el proceso de búsqueda.

Para nuestro ejemplo consideramos una población inicial de 5 individuos siendo cada individuo una cadena de diez Bits (1's ó 0's) generados de forma aleatoria.

| 1001110111 |
|---|
| 0001000010 |
| 1100110101 |
| 0110010101 |
| 1011111100 |

### 3.1.2- Función de coste.

La función de coste debe evaluar a los individuos para indicar cuál es la calidad de la solución que representan y poder realizar el proceso de selección.

Normalmente los algoritmos genéticos tratan de maximizar una función, pero si lo que se quiere es minimizar, no hay mías que utilizar la inversa de la función objetivo, o bien multiplicarla por -1.

En ocasiones, aparte de una función de coste, tenemos otra función objetivo. Es decir, que aunque la evaluación se realiza de acuerdo a una función se trata de alcanzar un objetivo que se evalúa mediante otra función distinta.

Esto suele ocurrir cuando se tratan problemas con restricciones o problemas en los que se pretende optimizar distintos parámetros conocidos como problemas multiobjetivo.

La función de selección para el ejemplo es el número de 1's, es decir el mejor individuo será el que contenga mayor numero de 1's.

### 3.1.3- Codificación de las soluciones.

Las características que debe cumplir una buena codificación son las siguientes:

**a)** Debe representar todo el espacio de soluciones. Ésta primera característica es bastante obvia, pues si no se representa todo el espacio de soluciones, es evidente que se pueden estar ignorando soluciones, entre las que puede estar precisamente la que estamos buscando.

**b)** Debe asegurar que al aplicar los operadores de cruce y mutación no se generan individuos incorrectos o irreales es decir, que todas las soluciones generadas representan una solución verdadera del problema.

**c)** Deben cubrir todo el espacio de búsqueda de una manera continua, ya que si existen discontinuidades el proceso de búsqueda puede ser aleatorio y la efectividad del algoritmo será bastante difícil de conseguir.

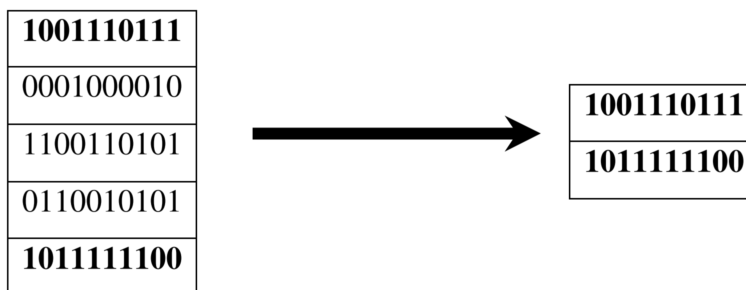
### 3.1.4- Operador de selección.

Los algoritmos genéticos utilizan un operador de selección que identifica a los mejores individuos de la población actual para utilizarlos como padres de la siguiente generación. Hay muchas formas de realizar la selección, pero siempre se debe asegurar que los mejores individuos tengan una mayor probabilidad de ser seleccionados. Una de las características más importantes de los algoritmos genéticos es que dejan un camino abierto a aquellas soluciones que no pertenecen a las mejores, para que puedan aportar parte de su información a la nueva generación.

La selección utilizada es muy importante para la correcta convergencia del algoritmo. La presión de selección debe ser tal que consiga un equilibrio entre exploración y explotación. Si la presión de selección es muy alta se corre el peligro que individuos de la población inicial con "fitness" superior a la media, que representan óptimos locales pero no globales, se reproduzcan en exceso provocando una perdida de diversidad y una convergencia prematura. En este caso se prima la explotación frente a la exploración. El caso contrario, una presión de selección baja puede provocar una búsqueda aleatoria o en el mejor de los casos una enorme ralentización del algoritmo. Probablemente, lo óptimo fuera un operador de selección que fuera evolucionando con el algoritmo de manera que en las primeras fases se primara la exploración y cuya

presión fuera creciendo hasta que se alcanzar un punto en el que se primara la explotación.

Para nuestro ejemplo si consideramos nuestra población inicial y tenemos en cuenta que sólo pasan a la siguiente generación los dos mejores, obtendríamos los dos siguientes individuos:




### 3.1.5- Operador de cruce.

Para explorar el espacio de soluciones los algoritmos genéticos utilizan dos operadores que tienen su base en la genética natural: los operadores de cruce y mutación. El operador de cruce consiste en elegir aleatoriamente un par de individuos de los seleccionados e intercambiar una parte de sus cromosomas entre sí. Este operador se aplica con una determinada probabilidad. El intercambio del código se realiza a partir de unos puntos que se seleccionan aleatoriamente.

Uno de los resultados de cruzar el primer individuo con el segundo de nuestra población ejemplo, sería:

Teniendo en cuenta que del primer progenitor escogemos sus cinco primeros Bits y del cinco últimos del segundo progenitor. Este es sólo una de las múltiples combinaciones que se pueden realizar a la hora de hacer un cruce.

### 3.1.6- Operador de mutación.

El operador de mutación es el encargado de realizar pequeños cambios en el código con una probabilidad muy pequeña. Se utiliza para reestablecer la diversidad que se haya podido perder con la aplicación sucesiva de los operadores de selección y cruce.

El método más habitual de realizar la mutación es seleccionar un gen del individuo y cambiar su alelo por otro de los del alfabeto. La probabilidad con la que estos cambios se producen suele ser baja, para evitar que el algoritmo genético realice una búsqueda aleatoria. Sin embargo en ocasiones puede ser necesario aumentar esta probabilidad para recuperar la diversidad de la población.

Para el ejemplo, el resultado de una posible mutación sería el siguiente:

Se puede observar que han mutado el segundo y cuarto Bit.

### 3.1.7- Tamaño de la población.

Un factor muy importante para la convergencia de los algoritmos genéticos es el tamaño de la población. El tiempo necesario para que un algoritmo genético converja a una solución única depende del tamaño de la población.

Aunque los algoritmos genéticos son eficientes, sin embargo no garantizan la obtención de una solución óptima. Su efectividad viene claramente determinada por el tamaño de la población. Es evidente que cuanto mayor sea el número de individuos se explorarán más zonas del espacio de soluciones. Pero también es bastante obvio que esto acarreará un costo computacional mayor. Por eso se debe buscar un compromiso entre el número de individuos utilizados y la calidad que se desea alcanzar.

## *3.2- Algoritmos Genéticos Multiobjetivo.*

La optimización con objetivos múltiples es, sin duda, un área de investigación muy importante tanto para los científicos como para los ingenieros, no sólo debido a que la mayoría de los problemas del mundo real tienen objetivos múltiples, sino también porque todavía restan por resolver muchas interrogantes en esta disciplina. De hecho, no hay ni siquiera una definición de "óptimo" que sea aceptada universalmente, como en el caso de la optimización con un solo objetivo. Esto hace difícil incluso poder comparar los resultados de una técnica con los de otra, porque normalmente la decisión acerca de cuál es la mejor respuesta corresponde realmente a un humano que aplicará sus propios criterios (los cuales suelen ser subjetivos).

La optimización con objetivos múltiples (base de los algoritmos genéticos multiobjetivo) puede definirse como el problema de encontrar un vector de variables de decisión que satisfaga las restricciones y optimice una función vectorial cuyos elementos representen las funciones objetivo. Estas funciones forman una descripción matemática de los criterios de desempeño que usualmente están en conflicto entre sí. Por lo tanto, el término "optimizar" significa encontrar una solución tal que proporcione valores para todos los objetivos que resulten aceptables para el diseñador [13].

Si sabemos que un algoritmo genético requiere de información escalar sobre el valor de aptitud de un individuo para operar, probablemente la idea más simple que podríamos proponer para lidiar con varios objetivos sería combinarlos en uno solo usando una suma, una multiplicación o cualquier otra combinación de operaciones aritméticas que se nos pueda ocurrir. Hay, sin embargo, problemas obvios con esta técnica. El primero de ellos es que debemos proporcionar información escalar precisa sobre el rango de los objetivos, a fin de evitar que uno de ellos domine a los demás. Esto implica que debemos saber, en la medida de lo posible, el comportamiento de cada una de las funciones objetivo, lo cual es normalmente (al menos en la mayoría de las aplicaciones del mundo real) un proceso muy costoso (en términos de tiempo de CPU) que suele estar fuera de nuestro alcance. Obviamente, si esta combinación de objetivos es posible (y lo es, en algunos casos), esta técnica no sólo es la más simple de implementar, sino que además es la más eficiente, porque no se requiere posterior interacción con el usuario y si el algoritmo genético tiene éxito en el proceso de

optimización, entonces los resultados serán al menos sub-óptimos en la mayoría de los casos.

Al proceso de combinar objetivos en una sola función se le denomina normalmente "función de agregación".

El método de "funciones de agregación" fue el primero en desarrollarse para la generación de soluciones no inferiores en problemas de optimización con objetivos múltiples. Esta técnica es muy eficiente desde el punto de vista de recursos de cómputo, y puede usarse para generar una solución fuertemente dominada que pueda usarse como un punto inicial para otras técnicas. El principal problema de esta técnica es cómo determinar los pesos apropiados cuando no tenemos suficiente información acerca del problema. En este caso, cualquier punto óptimo obtenido será una función de los coeficientes usados para combinar los objetivos. Esta técnica es muy simple y fácil de implementar, pero tiene como desventaja principal el perder soluciones potencialmente válidas, lo cual es un inconveniente serio si se pretenden resolver problemas del mundo real.

Tras estudiar la gran cantidad de posibles formas de implementar el algoritmo multiobjetivo para nuestro proyecto (MOGA, ASGA, NPGA, etc.) [1], y tras una primera y simple aproximación a la computación genética con objetivos múltiples con las funciones de agregación, optamos por la implementación de un algoritmo tipo VEGA, cuyo fundamento pasamos a explicar más en profundidad.

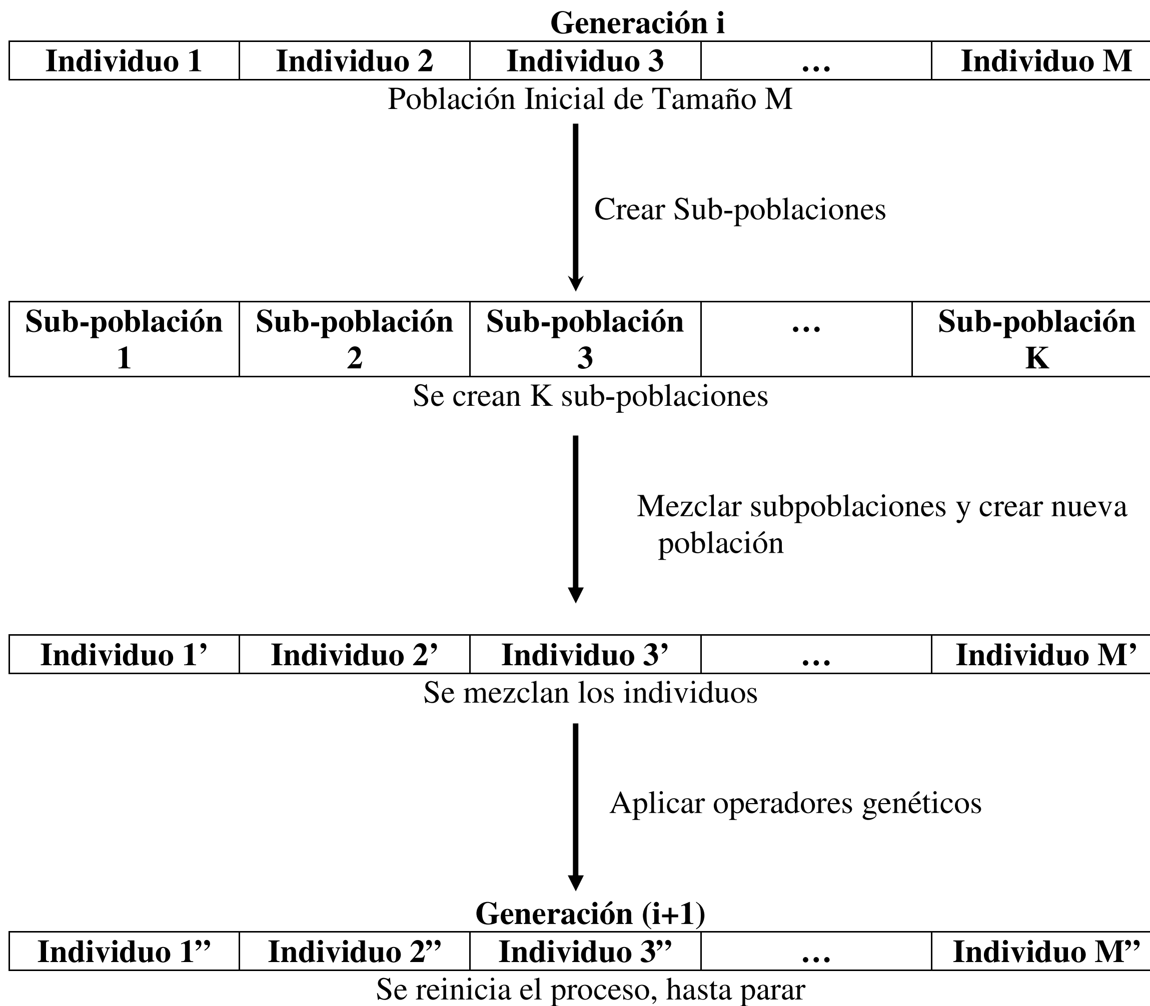
### 3.2.1- VEGA

Creado por David Schaffer en 1984 en su tesis "Multiple objective optimization with Vector Evaluated Genetic Algorithm" [2], la idea fue usar una extensión del algoritmo genético simple (SGA por sus siglas en inglés) a la que él llamó el "Vector Evaluated Genetic Algorithm" (VEGA a partir de ahora), y que difería del primero únicamente en la forma en que se efectuaba la selección. Este operador se modificó de tal manera que a cada generación se generaba un cierto número de sub-poblaciones efectuando una selección proporcional de acuerdo a la función objetivo en turno.

De tal forma, para un problema con k objetivos, se generaban k sub-poblaciones de tamaño M/k cada una (suponiendo un tamaño total de población de M). Estas sub-poblaciones se mezclarían luego entre sí a fin de obtener una nueva población de tamaño M, en la que el algoritmo genético podría aplicar los operadores de cruce y mutación de la forma usual.

Un problema de esta técnica es la "especiación", el cual consiste en que ciertas "especies" con características específicas evolucionan en direcciones distintas. Este problema se origina porque esta técnica selecciona individuos que son excelentes en una cierta dimensión de desempeño, pero no necesariamente en las otras. El peligro potencial con esto es que podríamos tener individuos que podrían ser muy útiles para soluciones compromiso, pero que no sobrevivirían bajo este esquema de selección, puesto que no es el mejor en ninguno de los objetivos del problema, sino sólo moderadamente bueno en todos.

Aquí vemos en forma de esquema la idea del algoritmo [1]:



# Capítulo 4: Datos de Entrada

## *4.1- Metodología de Optimización.*

El objetivo de este proyecto ha sido la optimización del uso de memoria dinámica en las aplicaciones diseñadas para sistemas empotrados multimedia de altas prestaciones minimizando los tiempos obtenidos por la siguiente metodología desarrollada por David Atienza en su Tesis "Metodología multinivel de refinamiento del subsistema de memoria dinámica para los sistemas empotrados multimedia de altas prestaciones" [3].

Esta metodología consta de una fase de división y dos fases de optimización, cada una de ellas encargada de refinar de manera independiente del hardware empleado un cierto componente del subsistema software de gestión de memoria dinámica del sistema dedicado (estructuras dinámicas de datos y gestor de memoria dinámica). Es importante mencionar que no se considera el refinamiento de las estructuras de variables estáticas (analizables en tiempo de compilación) ya que para ello existen otras técnicas complementarias.

Debido a la compleja arquitectura hardware/software que poseen los sistemas dedicados de altas prestaciones, la optimización de todos los elementos que integran el subsistema de memoria dinámica o que actúan con él obliga a aplicar refinamientos a diferentes niveles de abstracción. De lo contrario, los diseños finales serían subóptimos a nivel global [14]. Los resultados obtenidos en varios sistemas comerciales tales como una aplicación de renderizado escalable tridimensional o una aplicación de reconstrucción tridimensional, demuestran que la aplicación completa de esta metodología es capaz de lograr mejoras en todas las métricas relevantes (rendimiento, uso de memoria, consumo de energía) de más de tres órdenes de magnitud con respecto al diseño comercial inicial del sistema [15]. A continuación detallamos las fases básicas que se distinguen en la metodología propuesta.

### 4.1.1- Partición algorítmica.

La principal tarea a realizar al principio del flujo de diseño es la disgregación del código original de la aplicación en módulos de menor tamaño que se puedan analizar y optimizar individualmente de una manera eficiente según su carga final sobre el subsistema de memoria del sistema dedicado [16]. Como ejemplo de la necesidad de esta fase, baste indicar que las aplicaciones de última generación para reconstrucción de imágenes en 3D [17] o para el renderizado escalable de objetos complejos basado en triángulos pueden llegar a ocupar más de medio millón de líneas de código en un lenguaje de alto nivel de la complejidad de C++ [18].

Esta fase de división contempla como entrada el código de la aplicación original a ejecutar por el sistema dedicado, usando un lenguaje de alto nivel de abstracción tipo C++ [19]. Este código, aunque todavía muy alejado de la implementación optimizada final, permite la validación funcional de su comportamiento algorítmico.

### 4.1.2- Exploración de las estructuras dinámicas de datos.

Tras la división del sistema completo en sub-algoritmos o tareas básicas, en esta fase se realiza el refinamiento del modo en que la memoria dinámica es reservada y accedida por parte de las aplicaciones o servicios multimedia de usuario existentes en el sistema final. Esta memoria dinámica es utilizada por estructuras creadas en tiempo de ejecución que, debido a su complejidad, deben ser definidas específicamente para cada aplicación si se desea lograr un resultado óptimo.

Se debe caracterizar el uso de memoria y patrón de accesos concretos a memoria dinámica ocasionado por la aplicación particular que se está optimizando. Para ello no basta con realizar un análisis estático del código C++ de la aplicación que se desea optimizar, sino que también es necesario estudiar el comportamiento en tiempo de ejecución de cada una de las variables dinámicas existentes en el sistema. La razón es que el primer estudio estático es suficiente para identificar cuáles son los tamaños de los objetos dinámicos que se van a almacenar en las estructuras de datos pero no su cantidad real según cada caso de entrada. Por tanto, un estudio de comportamiento debe

ser realizado para cada entrada representativa del sistema que haya sido definida y proporcionada por el diseñador de la aplicación original para el sistema dedicado objetivo.

Con el fin de obtener el perfil de ejecución de una variable concreta de una manera coherente con los requisitos anteriormente mencionados , se ha diseñado una biblioteca orientada a objetos, implementada en código C++ y reutilizable para cualquier tipo de variable. Esta biblioteca es fácilmente configurable mediante la modificación de una serie de directivas de preprocesado y, dado que no sobrecarga en exceso la aplicación original (entre el 3% y el 15%), es posible estudiar el comportamiento final del sistema con relativa precisión.

Tras la inserción de la biblioteca y ejecución del nuevo código, se obtiene un informe detallado del número de accesos totales, de lectura, peticiones de asignación de memoria, liberación, etc. Además, se obtiene información detallada para cada ámbito definido en el sistema y un informe dividido de la información global relacionada con cada variable. [3]

### 4.1.3- Exploración automática de las estructuras dinámicas de datos.

Con la información obtenida mediante estos perfiles en ejecución, realizamos la ejecución de la aplicación de manera iterativa para el conjunto representativo de valores de entrada, definido por los diseñadores de cada aplicación concreta, utilizando cada una de las EDDs implementadas. Tras esto, el diseñador puede definir cuáles son las restricciones que desea respecto a las tres métricas usadas (energía, memoria y rendimiento), indicando unos valores concretos para el sistema dedicado que se está diseñando.

De entre todas las soluciones obtenidas, el diseñador no tiene nada más que escoger la que mejor se ajuste a los requerimientos de su diseño actual.

*El único problema de esta metodología es la gran cantidad de tiempo necesario para la exploración exhaustiva de las distintas EDDs, siendo del orden de días. Es por*

*ello que el objetivo de nuestro proyecto es obtener los mismos resultados mediante un algoritmo genético en un tiempo inferior (del orden de minutos).*

Para ello, nuestra aplicación posee dos fuentes principales:

- La biblioteca de estructuras de datos (EDD), previamente implementados y sobre la que trabajamos para realizar las elecciones adecuadas para las variables a optimizar.

- Los archivos de perfil en ejecución (profiling) generados por las aplicaciones a optimizar, en base a cuya información realizamos una u otra elección de EDD.

Estos datos nos fueron proporcionados por David Atienza [3]. La biblioteca de EDDs y su caracterización mediante ecuaciones los obtuvimos de su Tesis, mientras que los resultados de los perfiles en ejecución fueron obtenidos al aplicar su metodología a diferentes aplicaciones por miembros del Centro de Investigación Internacional Inter-University MicroElectronics Center, el IMEC (principalmente Christophe Poucet), centro con el cual hemos colaborado para obtener unos mejores resultados en la metodología previamente expuesta.

## 4.2- Biblioteca de estructuras de datos dinámicas.

La biblioteca de estructuras de datos dinámicas (en adelante EDD) es una implementación en C++ de las 16 posibles formas de almacenar los datos dinámicos en una aplicación, junto a unas ecuaciones mediante las cuales se obtienen los valores de interés para nuestra optimización [3].

Estos valores característicos están relacionados con el uso de la memoria y son los siguientes:

- Número de accesos medio para una escritura/lectura aleatoria (NAaleatorio).
- Número de accesos medio para un recorrido secuencial completo (NAsecuencial).
- Uso medio de memoria (UMmedio).
- Número de llamadas para la creación/destrucción de las EDDs (NAcreac/destruc).

Estos valores se miden en base al siguiente conjunto de variables:

- Cardinalidad del conjunto de claves (Ne).
- Tamaño de los elementos almacenados (Te)
- Tamaño de cada referencia (Tref)
- Tamaño de clave de acceso (Tcla).
- Número de elementos válidos almacenados (NVe)
- Grado de dispersión (GD = NVe / Ne)
- Factor de agrupamiento de elementos básicos (FA).
- δ es el porcentaje de agrupaciones de segundo nivel con algún elemento.

Tras la presentación de los datos que usaremos en nuestra implementación, pasamos a describir las distintas implementaciones de la biblioteca de EDD's, así como la forma en la que calculamos los valores necesarios de cada una de ellas [3]:

**-AR(T):**

Es un vector dinámico disperso donde cada posición contiene un elemento del tipo básico (o clase) *T*. Su tamaño se fija en el momento de su creación, en tiempo de ejecución, mediante una única llamada al método *new()* y se destruye después mediante una llamada al método *delete.* Sus características son las siguientes:

$NA_{aleatorio} = 1$
$NA_{secuencial} = N_e$
$UM_{medio} = N_e \times T_e$
$NA_{creac/destruc} = 1$

**-AR(T+ref):**

Esta implementación añade la transformación para optimizar el recorrido secuencial de los vectores dinámicos mediante la transformación de adición de una estructura enlazada. En esta implementación, las dos ecuaciones que varían respecto a las ecuaciones iniciales para el tipo AR(T) son las relativas al uso de memoria medio y el número de accesos para un recorrido secuencial:

$NA_{secuencial} = 2 \times NV_e = 2 \times GD \times N_e$
$UM_{medio} = N_e \times (T_e + T_{ref})$

**-PAR(T):**

Vector dinámico de punteros a elementos del tipo básico *T*. Es el resultado de añadir a un vector dinámico disperso un nivel de referencia. En este caso se realiza una primera llamada para crear todo el vector de punteros a elementos tipo T, y posteriormente se emplea una llamada independiente a la función *new()* para almacenar el espacio requerido para cada elemento individual. La eliminación de la estructura dinámica completa implica el mismo número de llamadas independientes a la función *delete()* ya que cada elemento debe eliminarse de manera individualizada. En esta implementación el coste en número de accesos por elemento es más alto debido al

coste adicional del puntero que hay que seguir. Las ecuaciones que resultan en este caso son las siguientes:

$NA_{aleatorio} = 2$
$NA_{secuencial} = N_e + NV_e = N_e \times (1 + NV_e)$
$UM_{medio} = N_e \times T_{ref} + NV_e \times T_e = N_e \times (T_{ref} + GD \times T_e)$
$NA_{creac/destruc} = 1 + NV_e = 1 + GD \times N_e$

**-PAR(T+ref):**

Añade la transformación para optimizar el recorrido secuencial de los vectores dinámicos a la EDD anterior. Las ecuaciones que varían en este caso son las siguientes:

$NA_{secuencial} = 3 \times N_e \times GD$
$UM_{medio} = N_e \times 2 \times T_{ref} + NV_e \times T_e = N_e \times ( 2 \times T_{ref} + GD \times T_e)$

**-SLL(T+cla)**:

Lista enlazada simple de elementos del tipo *T* con su clave explícita. Esta implementación reserva la memoria que se requiere para cada elemento del tipo T de manera individualizada, mediante una llamada al método *new()*. Para su eliminación se requiere una llamada al método *delete()* por cada objeto almacenado. Debido al puntero que almacena la dirección del último elemento que se ha accedido en el caso del recorrido secuencial, si se produce este tipo de patrón de accesos, la obtención del nuevo elemento requiere únicamente dos accesos y se puede acceder a la posición siguiente a la actualmente memorizada con un solo acceso adicional. Las ecuaciones que resultan en este caso son las siguientes:

$NA_{aleatorio} = (NV_e / 2) + 1 = (GD \times N_e / 2) + 1$

$NA_{secuencial} = 3 \times NV_e = 3 \times N_e \times GD$

$UM_{medio} = NV_e \times (T_{ref} + T_e + T_{cla}) + T_{ref} = N_e \times GD \times (T_{ref} + T_e + T_{cla}) + T_{ref}$

$NA_{creac/destruc} = 1 + NV_e$

**-DLL(T+cla):**

Lista doblemente enlazada de elementos del tipo *T* con su clave explícita. Esta implementación, de manera análoga a la anterior EDD, reserva el espacio ocupado por cada elemento en el momento que se desea almacenar cada uno de ellos, junto con dos

referencias, al elemento anterior y siguiente. La eliminación se realiza igualmente mediante una llamada al método *delete()* por cada objeto almacenado, que se encarga al mismo tiempo de actualizar las referencias correspondientes de los elementos circundantes. El número de accesos medio de esta implementación para recuperar un elemento es la mitad del caso anterior, ya que se puede comenzar por el extremo que esté más cercano a la posición del elemento que se desee acceder. En relación al acceso secuencial, se necesitan dos accesos para recuperar el elemento que se sitúa en la misma posición que el elemento anteriormente accedido, tres accesos para el elemento en la posición siguiente y tres accesos igualmente para el elemento en la posición anterior a la actualmente memorizada. Como consecuencia, esta implementación reduce el número de accesos medio con respecto a DLL(T+cla) al realizarse un recorrido secuencial (ya sea en orden ascendente o descendente de clave). Respecto al uso de memoria medio de esta implementación se observa que es mayor que en el caso anterior ya que hay dos punteros por elemento (al elemento anterior y al siguiente). Así pues, las ecuaciones que resultan en este caso son las siguientes:

$NA_{aleatorio} = (NV_e / 4) + 1 = (GD \times N_e / 4) + 1$
$NA_{secuencial} = 3 \times NV_e = 3 \times N_e \times GD$
$UM_{medio} = NV_e \times (2 \times T_{ref} + T_e + T_{cla}) + T_{ref}$
$NA_{creac/destruc} = 1 + NV_e = 1 + GD \times N_e$

**-PAR(AR(T))**:

Vector dinámico de punteros a vectores de elementos del tipo *T* que almacenan únicamente la primera clave del rango de claves explícitas de los elementos contenidos en cada posición ya que todos los vectores del segundo nivel son del mismo tamaño. Combina las dos implementaciones básicas de vectores dinámicos de la biblioteca para lograr optimizar el número de accesos y uso de memoria en aquellas EDDs donde se conoce el tamaño básico de los grupos de elementos que se van a almacenar, pero no qué grupos exactamente existirán en cada ejecución. Esta implementación utiliza una llamada el método *new()* para reservar el espacio en memoria de un conjunto de elementos del tipo T y dicha reserva de memoria global evita que se realicen múltiples llamadas para los elementos almacenados en el vector dinámico que ocupa dicha posición. De un modo análogo, la eliminación de una posición del vector de punteros, mediante una llamada a *delete()*, se encarga de eliminar el espacio ocupado por todos los elementos contenidos en el vector que ocupa dicha posición. De este modo, con esta

implementación, se logra optimizar el uso de memoria pero sin deteriorar significativamente el número de accesos a la EDD en caso de que la dispersión de elementos no sea uniforme, ya que el número de accesos medio a un elemento aleatorio es similar al caso AR(T). Por ello, el perfil dinámico de la aplicación que se desea optimizar se utiliza para ajustar el tamaño de ambos vectores. De un modo similar, en cuanto al uso de memoria, este tipo de implementación es la más eficiente de todas aquellas estructuras en las que el tamaño de los datos almacenados y procesados crece siempre incrementalmente. Las características finales de este tipo de implementación son las siguientes:

$NA_{aleatorio} = 2$

$NA_{secuencial} = N_e / FA + \delta \text{ x } N_e$

$UM_{medio} = N_e \text{ x } (T_{ref} / FA + \delta \text{ x } T_e)$

$NA_{creac/destruc} = 1 + \delta \text{ x } N_e$

**-PAR(AR(T+ref)):**

Implementación que optimiza adicionalmente el recorrido secuencial de los vectores dinámicos contenidos en cada posición del vector de punteros. Por tanto, esta EDD tiene prácticamente las mismas ecuaciones que la implementación anterior y sólo varían las siguientes:

$NA_{secuencial} = N_e / FA + 2 \text{ x } NV_e$

$UM_{medio} = N_e / FA \text{ x } T_{ref} + N_e \text{ x } \delta \text{ x } (T_e + T_{ref})$

**-SLL(AR(T)+cla)**:

Lista enlazada simple de vectores de elementos del tipo *T* más, en cada posición, de la primera y última claves explícita de los elementos almacenados. Esta implementación realiza una única llamada a *new()* para reservar el espacio ocupado por un vector de elementos del tipo T, que van siendo almacenados en cada posición a medida que se necesita. De manera similar, la eliminación de una posición de SLL(AR(T)+cla), necesita una única llamada a *delete()*, que se encarga de eliminar el espacio en memoria de los elementos contenidos en el vector dinámico que ocupa dicha posición. En el caso del recorrido secuencial se produce la ganancia al intentar acceder a

cualquiera de los dos niveles, ya que la dirección de la posición en la lista enlazada es guardada, así como la posición en el vector del nivel inferior. Las ecuaciones resultantes para este caso son las siguientes:

$NA_{aleatorio} = (\delta \times N_e / FA) / 2) + 2$

$NA_{secuencial} = 3 \times NV_e + \delta \times N_e$

$UM_{medio} = 2 \times (\delta \times N_e / FA) \times T_{ref} + 2 \times T_{ref} + \delta \times N_e \times T_e$

$NA_{creac/destruc} = (\delta \times N_e / FA) + \delta \times N_e$

**-SLL(AR(T+ref)+cla):**

Incluye la optimización del recorrido secuencial de los vectores dinámicos almacenados en cada posición. En este caso varían las ecuaciones de recorrido secuencial y uso de memoria con respecto a la implementación anterior:

$NA_{secuencial} = 3 \times NV_e + NV_e$

$UM_{medio} = 2 \times (\delta \times N_e / FA) \times T_{ref} + 2 \times T_{ref} + \delta \times N_e \times (T_e + T_{ref})$

**-DLL(AR(T)+cla)**:

Lista doblemente enlazada de vectores de elementos del tipo *T* y del rango de claves explícitas almacenadas. Su implementación es análoga a SLL(AR(T)+cla), ya que utiliza solamente una llamada al método *new()* o *delete()* para reservar/eliminar, respectivamente, el espacio utilizado por el vector de elementos almacenados en cada una de sus posiciones internas. La ventaja principal en este caso es que las búsquedas de elementos se pueden realizar desde cualquiera de los dos extremos, como ya sucedía con la estructura DLL(T+cla). Por tanto, las características de esta implementación de EDD son las siguientes:

$NA_{aleatorio} = (\delta \times N_e / FA) / 4) + 2$

$NA_{secuencial} = 3 \times NV_e + \delta \times N_e$

$UM_{medio} = 3 \times (\delta \times N_e / FA) \times T_{ref} + 3 \times T_{ref} + \delta \times N_e \times T_e$

$NA_{creac/destruc} = (\delta \times N_e / FA) + \delta \times N_e$

**-DLL(AR(T+ref)+cla):**

Se incluye también en la biblioteca la implementación que añade la optimización adicional para el recorrido secuencial de los vectores dinámicos contenidos. Las ecuaciones que difieren de las anteriores para DLL(AR(T)+cla) son las siguientes:

$NA_{secuencial} = 3 \text{ x } NV_e + NV_e$

$UM_{medio} = 2 \text{ x } (\delta \text{ x } N_e / FA) \text{ x } T_{ref} + 3 \text{ x } T_{ref} + \delta \text{ x } N_e \text{ x } (T_e + T_{ref})$

**-SLL(PAR(T)+cla)**:

Lista enlazada simple de vectores de punteros a elementos del tipo *T* y donde se almacenan un rango de claves explícitas. La reserva de memoria en esta implementación se realiza de manera individual para cada vector de punteros almacenado y, a su vez, cada elemento individual de este vector se crea mediante una llamada independiente a *new()*. La eliminación de la estructura se realiza de manera análoga, mediante llamadas independientes. En esta implementación se sacrifica más el acceso óptimo a los elementos almacenados, mediante un nivel de referencia adicional impuesto por el vector de punteros, a favor del uso eficiente de memoria. Las ecuaciones que resultan son las siguientes:

$NA_{aleatorio} = (\delta \text{ x } N_e / FA) / 2) + 3$

$NA_{secuencial} = 3 \text{ x } NV_e + \delta \text{ x } N_e + NV_e$

$UM_{medio} = 2 \text{ x } (\delta \text{ x } N_e / FA) \text{ x } T_{ref} + 2 \text{ x } T_{ref} + \delta \text{ x } N_e \text{ x } T_e + NV_e \text{ x } T_e$

$NA_{creac/destruc} = (\delta \text{ x } N_e / FA) + \delta \text{ x } N_e + NV_e$

**-SLL(PAR(T+ref)+cla) :**

Aparte de la implementación básica, incluimos la versión que añade la optimización adicional para un recorrido secuencial eficiente de los vectores dinámicos de punteros almacenados. En este caso se modifican las ecuaciones de recorrido secuencial y uso de memoria:

$NA_{secuencial} = 3 \text{ x } NV_e + 2 \text{ x } NV_e$

$UM_{medio} = 2 \text{ x } (\delta \text{ x } N_e / FA) \text{ x } T_{ref} + 2 \text{ x } T_{ref} + 2 \text{ x } \delta \text{ x } N_e \text{ x } T_{ref} + NV_e \text{ x } T_e$

**-DLL(PAR(T)+cla)**:

Lista doblemente enlazada de vectores de punteros a elementos del tipo *T* más los índices del rango de claves almacenados en cada vector de elementos. La reserva/liberación de memoria se realiza de un modo similar a SLL(PAR(T)+cla, la principal diferencia es la creación de nuevos compromisos entre uso de memoria y acceso eficiente al sistema debido al uso de dos referencias en cada elemento. En este caso las ecuaciones resultantes son las siguientes:

$NA_{aleatorio} = (\delta \text{ x } N_e / FA) / 4) + 3$

$NA_{secuencial} = 3 \text{ x } NV_e + \delta \text{ x } N_e + NV_e$

$UM_{medio} = 3 \text{ x } (\delta \text{ x } N_e / FA) \text{ x } T_{ref} + 3 \text{ x } T_{ref} + \delta \text{ x } N_e \text{ x } T_e + NV_e \text{ x } T_e$

$NA_{creac/destruc} = (\delta \text{ x } N_e / FA) + \delta \text{ x } N_e + NV_e$

**-DLL(PAR(T+ref)+cla):**

Incluye la optimización adicional para la disminución del número de accesos en caso de un recorrido secuencial de los elementos contenidos en los vectores dinámicos de punteros. En este caso las ecuaciones modificadas respecto a DLL(PAR(T)+cla) son solamente las siguientes:

$NA_{secuencial} = 3 \text{ x } NV_e + 2 \text{ x } NV_e$

$UM_{medio} = 3 \text{ x } (\delta \text{ x } N_e / FA) \text{ x } T_{ref} + 3 \text{ x } T_{ref} + 2 \text{ x } \delta \text{ x } N_e \text{ x } T_{ref} + NV_e \text{ x } T_e$

*Con todos estos datos para cada tipo de estructura, somos capaces de saber que implementación va a ser la óptima para cada variable estudiada, dependiendo del uso que se haga de ella, información que sacaremos previamente del perfil en ejecución de la aplicación objetivo.*

## *4.3- Resultados de los Perfiles.*

El resultado de la segunda fase de la metodología de optimización (exploración de las estructuras dinámicas de datos), es la generación de un archivo en el que se almacena el perfil dinámico en ejecución [3], con la información que necesitemos para realizar la optimización. El profiling está formado por las siguientes instrucciones, que son usadas para el manejo de la estructura de datos en el programa a optimizar. Estas instrucciones conllevan un número determinado de lecturas y/o escrituras. Este es el dato que usamos en la optimización.

A continuación mostramos los distintos tipos de operaciones que manejamos, explicando previamente los posibles parámetros que pueden utilizar estas operaciones:

- type_id (string) : tipo de la variable seleccionada.
- instance_id (number) : identificador para referenciar la variable seleccionada.
- sizeof element (number): tamaño de la variable seleccionada.
- address (address): dirección de la variable referenciada.
- offset (number): desplazamiento.
- index (number): índice o posición a acceder en un array.

**-LOG_VECTOR_CONSTRUCT_BEGIN:**

| Operación | Creamos un vector. |
|---|---|
| Parámetros | type_id (string),<br>instance_id (number),<br>sizeof element (number). |
| Número de accesos de lectura (Nr) | *0* |
| Número de accesos de escritura (Nw) | *0* |

**-LOG_VECTOR_DUPLICATE_BEGIN:**

| Operación | Constructor de copia basado en otro vector del mismo tipo. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>instance_id of vector that is copied from |
| Número de accesos de lectura (Nr) | *n* |
| Número de accesos de escritura (Nw) | *n* |
| Comentarios | Se añade *n* (estado de ocupación actual del vector) escrituras a la estructura que se crea, y *n* lecturas a la estructura fuente, para poder copiar de un lado a otro todos sus valores. |

**-LOG_VECTOR_DESTRUCT_BEGIN:**

| Operación | Llamada al destructor del vector. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number) |
| Número de accesos de lectura (Nr) | *0* |
| Número de accesos de escritura (Nw) | *n* |
| Comentarios | Siendo *n*, estado de ocupación actual del vector |

**-LOG_VECTOR_SWAP_BEGIN:**

| Operación | Intercambio de los punteros de dos vectores (es decir, intercambiar sus elementos), usado solo de forma interna para reutilizar el constructor de copia en el operador de asignación. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>instance_id of vector that is being swapped with |
| Número de accesos de lectura (Nr) | *0* |
| Número de accesos de escritura (Nw) | *1* |
| Comentarios | Se añade un escritra a las dos estructuras implicadas en el swap, en la id1(type_id) y en la id2 (instance_id) |

**-LOG_VECTOR_RESIZE_BEGIN:**

| Operación | Comienzo del cambio de tamaño de un vector. Dentro de esta instrucción habrá una reserva y una liberación de memoria. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number) |
| Número de accesos de lectura (Nr) | *n* |
| Número de accesos de escritura (Nw) | *n* |
| Comentarios | Sólo es necesario para las estructuras tipo vector. Siendo *n*, estado de ocupación actual del vector |

**-LOG_ITERATOR_NEXT_BEGIN:**

| Operación | El iterador es incrementado usando el operador ++. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>address (address) |
| Número de accesos de lectura (Nr) | *1* |
| Número de accesos de escritura (Nw) | *1* |
| Comentarios | Una lectura para las estructuras de tipo lista enlaza y doblemente enlazada. Para el array no tiene sentido un iterador. |

**-LOG_ITERATOR_PREVIOUS_BEGIN:**

| Operación | El iterador es decrementado usando el operador --. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>address (address) |
| Número de accesos de lectura (Nr)<br>Lista dinámica simple | *n* |
| Número de accesos de escritura (Nw)<br>Lista dinámica simple | *0* |
| Número de accesos de lectura (Nr)<br>Lista dinámica doblemente enlazada | *1* |
| Número de accesos de escritura (Nw)<br>Lista dinámica doblemente enlazada | *0* |
| Número de accesos de lectura (Nr)<br>Array | *1* |
| Número de accesos de escritura (Nw)<br>Array | *0* |
| Comentarios | En esta caso se diferencian claramente el número de lecturas en los tres tipos de estructuras de datos. Para el caso de la lista enlazada simple tomamos como lecturas el estado de ocupación actual de la estructura(*n*). |

**-LOG_ITERATOR_ADD_BEGIN:**

| Operación | Actualizamos el iterador mediante la operación +=. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>address (address)<br>offset (number) |
| Número de accesos de lectura (Nr)<br>Lista dinámica simple | *offset* |
| Número de accesos de escritura (Nw)<br>Lista dinámica simple | *0* |
| Número de accesos de lectura (Nr)<br>Lista dinámica doblemente enlazada | *offset* |
| Número de accesos de escritura (Nw)<br>Lista dinámica doblemente enlazada | *0* |
| Número de accesos de lectura (Nr)<br>Array | *0* |
| Número de accesos de escritura (Nw)<br>Array | *0* |
| Comentarios | Aquí el parametro *offset* me indica el numero de lecturas en las listas enlazada simple y doblemente enlazada. |

**-LOG_ITERATOR_SUB_BEGIN:**

| Operación | Actualizamos el iterador mediante la operación -=. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>address (address)<br>offset (number) |
| Número de accesos de lectura (Nr)<br>Lista dinámica simple | *n - offset* |
| Número de accesos de escritura (Nw)<br>Lista dinámica simple | *0* |
| Número de accesos de lectura (Nr)<br>Lista dinámica doblemente enlazada | *offset* |
| Número de accesos de escritura (Nw)<br>Lista dinámica doblemente enlazada | *0* |
| Número de accesos de lectura (Nr)<br>Array | *0* |
| Número de accesos de escritura (Nw)<br>Array | *0* |
| Comentarios | Es semejante a la operación anterior, pero en este caso al número de lecturas de la estructura del tipo lista enlazada simple se le resta de la ocupación actual (*n*) el *offset*. |

**-LOG_ITERATOR_GET_BEGIN:**

| Operación | Accedemos al iterador, ya sea a su referencia mediante * o con ->. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>address (address) |
| Número de accesos de lectura (Nr) | *1* |
| Número de accesos de escritura (Nw) | *1* |
| Comentarios | Añadimos una lectura y una escritura en todos los casos, ya que no sabemos con certeza si la operación es de lectura o escritura. |

**-LOG_VECTOR_GET_BEGIN:**

| Operación | El vector es modificado o accedido en un sitio específico, mediante un get o un set. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>index (number) |
| Número de accesos de lectura (Nr)<br>Lista dinámica simple | *index + 1* |
| Número de accesos de escritura (Nw)<br>Lista dinámica simple | *1* |
| Número de accesos de lectura (Nr)<br>Lista dinámica doblemente enlazada | *index + 1* |
| Número de accesos de escritura (Nw)<br>Lista dinámica doblemente enlazada | *1* |
| Número de accesos de lectura (Nr)<br>Array | *1* |
| Número de accesos de escritura (Nw)<br>Array | *1* |
| Comentarios | En el caso del array el acceso es instantaneo con sólo una escritura y una lectura. Mientras que en las listas enlazadas simple y doblemente enlazadas hay que sumarle a las lecturas el parametro *index + 1* y una a las escrituras. |

**-LOG_VECTOR_ADD_BEGIN:**

| Operación | Añadimos un elemento a un indice especifico. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>index (number) |
| Número de accesos de lectura (Nr)<br>Lista dinámica simple | *index* |
| Número de accesos de escritura (Nw)<br>Lista dinámica simple | *1* |
| Número de accesos de lectura (Nr)<br>Lista dinámica doblemente enlazada | *index* |
| Número de accesos de escritura (Nw)<br>Lista dinámica doblemente enlazada | *1* |
| Número de accesos de lectura (Nr)<br>Array | *0* |
| Número de accesos de escritura (Nw)<br>Array | *1* |
| Comentarios | Sólo añdimos una escritura en el caso del array, e *index* lecturas y una escritura para las listas enlazadas. |

**-LOG_VECTOR_REMOVE_BEGIN:**

| Operación | Borramos un elemento de una posición del array. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number)<br>index (number) |
| Número de accesos de lectura (Nr)<br>Lista dinámica simple | *index* |
| Número de accesos de escritura (Nw)<br>Lista dinámica simple | *1* |
| Número de accesos de lectura (Nr)<br>Lista dinámica doblemente enlazada | *index* |
| Número de accesos de escritura (Nw)<br>Lista dinámica doblemente enlazada | *1* |
| Número de accesos de lectura (Nr)<br>Array | *0* |
| Número de accesos de escritura (Nw)<br>Array | *1* |
| Comentarios | Es el mismo caso que la operación anterior pero haciendo un remove. |

**-LOG_VECTOR_CLEAR_BEGIN:**

| Operación | Borramos todos los elementos de un vector. |
|---|---|
| Parámetros | type_id (string)<br>instance_id (number)<br>sizeof element (number) |
| Número de accesos de lectura (Nr) | *0* |
| Número de accesos de escritura (Nw) | *n* |
| Comentarios | Para realizar un borrado completo de todos los elemetos de las estructuras es necesario realizar *n* (siendo *n* la ocupacion de la estructura de datos) escrituras. |

*Tras analizar el perfil en ejecución de la aplicación a optimizar, podemos extraer la siguiente información (para cada variable dinámica) que usaremos en nuestro algoritmo:*

*Número de lecturas($N_r$), número de escrituras ($N_w$), cardinalidad del conjunto de claves ($N_e$), tamaño de los elementos almacenados ($T_e$), tamaño de cada referencia ($T_{ref}$), tamaño de clave de acceso ($T_{cla}$), número de elementos válidos almacenados ($NV_e$), factor de agrupamiento de elementos básicos (FA).*

# Capítulo 5: Nuestra Aplicación.

## *5.1- Desarrollo de la aplicación.*

El desarrollo de la aplicación lo hemos dividido en 3 módulos, y cada módulo tiene sus fases de desarrollo:

**Módulo 1 "Interpretación Profilings":** Este módulo es el encargado de obtener la información necesaria de los profilings. A partir de él obtenemos el número de escrituras y lecturas de las variables para los tres tipos de estructuras de datos (array, lista enlazada simple y lista doblemente enlazada). Además de esto llevamos el control del estado de ocupación de la estructura de datos para calcular su tamaño máximo al que puede llegar. Las clases correspondientes a este módulo son:

- **Resultado:** Es la clase encargada de albergar las escrituras y lecturas de cada estructura de datos, así como su ocupación en dicha estructura. Obtiene la información de un archivo de texto con todo el perfil dinámico de la ejecución de una aplicación determinada.

- **ListaResultados:** En esta clase almacenamos una lista de *Resultados*. Cada variable leída del profiling tiene asociado un *Resultado* y su acceso se realiza mediante la *id* de la variable. Cuando los profilings tienen demasiadas variables, existe la posibilidad de hacer un filtrado de un número determinado de variables, y de esta forma no considerar todas las variables en la *listaResultados.*

**<u>Fase 1:</u>** Al principio de este módulo sólo considerábamos un tipo de profiling. Además solo teníamos en cuenta una única estructura de datos, una lista doblemente enlazada.

**<u>Fase 2:</u>** En esta fase, el programa ya era capaz de leer varios tipos de profilings y consideraba las tres estructuras de datos mencionadas antes. Aquí también se añadió la opción de filtrar la lista de resultados a un número

determinado de variables, puesto que cabía la posibilidad de leer hasta 800 variables.

**Módulo 2 "Genético":** En este módulo nos encargamos de llevar a cabo el algoritmo genético multiobjetivo (VEGA). El módulo genético ha ido evolucionado a medida que íbamos añadiendo mas funcionalidad al algoritmo.

- **Individuo:** Esta clase es la encargada de codificar de manera binaria la información de las variables dinámicas usadas en la aplicación que queremos optimizar. El resultado de la optimización devolverá los mejores individuos.

- **Población:** La población esta formada por una lista de individuos, la cual va evolucionando en nuestro algoritmo genético hasta encontrar las soluciones óptimas.

- **Vega:** Esta clase consiste en la implementación del algoritmo genético usado sobre las poblaciones. En cada iteración del algoritmo, se realiza un filtrado de individuas dentro de su población mediante una función de selección multiobjetivo. Para ello lo primero que realiza es la división de la población en tantas subpoblaciones como objetivos queramos optimizar.

  Para poder seleccionar los mejores individuos de casa subpoblación es necesario evaluarlos con respecto a sus objetivos de optimización (rendimiento, memoria y energía) mediante las siguientes ecuaciones:

  - Rendimiento = (NAaleatorio* (3*(lecturas + escrituras -2)/4)) + (NAsecuencial*((lecturas + escrituras -2)/4)) + (NAcreaDes * 2)

    El resultado se mide en accesos a memoria. Donde *lecturas* y *escrituras* son datos sacados del profiling y representan el número de lecturas y escrituras de cada variable a optimizar. *NAaleatorio* es el número de instrucciones necesarias para realizar un acceso aleatorio. *NAsecuencial* es el número de instrucciones necesarias para realizar un acceso secuencial. *NAcreaDes* es el número de

instrucciones para crear o destruir la estructura. Estos tres valores han sido obtenidos de forma teóricas para cada tipo de estructuras de datos.

- Memoria = UMmedio

El resultado se mide en Bytes. Donde *UMmedio* es el uso medio de memoria para una variable dependiendo del tipo de estructura dinámica usada para implementarla. Este dato tambien ha sido obtenido de forma teórica.

- Energía = ((Npa * Epa) + (Nrw * Erw) + (UMmedio * Eestatica))/1000

El resutado se mide en nJ (nanoJulios). Donde *Npa* es el número de fallos de caché. *Epa* es el consumo de acceso a memoria principal. *Nrw* es el número de lecturas y escrituras realizadas sobre memoria caché. *Erw* es el consumo de acceso a memoria caché. *UMmedio* es el tamaño en Bytes de la variable. *Eestatica* es el consumo por Byte de memoria.

Ante la imposibilidad de calcular los fallos de caché de manera experimental, realizamos una simplificación basándonos en datos estadísticos mediante los cuales consideramos el número de fallos de caché como el 2% del total de accesos realizados.

Después fusionamos los mejores individuos de estas subpoblaciones y aplicamos los operadores de mutación y cruce para obtener la siguiente generación de la población. Esta población esta formada por:

- Los mejores de la generación anterior, el 30%.
- El resultado de mutar los mejores de anterior generación, el 20%.
- El resultado de cruzar los mejores de la generación anterior, el 40%.

- Y para "renovar" la población, un 10% de individuos nuevos creados de forma aleatoria.

De esta manera se obtenía una nueva población que se consideraría en la siguiente iteración.

***Fase 1:*** Al inicio partíamos de algoritmos genéticos simples para las pruebas iniciales, y para familiarizarnos con ellos. Estos algoritmos eran monobjetivo, lo cual no eran suficientes para el tipo de optimización que queríamos llevar a cabo.

Para la clase *Individuo* solo considerábamos una única variable a optimizar, por lo que el tamaño del individuo estaba prefijado.

***Fase 2:*** En esta fase introducimos los algoritmos multiobjetivo, lo cual nos permitía orientar nuestros individuos hacia un compromiso entre los tres objetivos de optimización.

En la clase *Individuo,* pasamos de considerar una única variable a guardar la información de todas las variables que deseemos. Para ello un individuo pasaba a estar formado por varios fragmentos que nos son más que subindividuos que representan a una única variable.

En esta fase aparece la clase *Vega* que se encarga de implementar el algoritmo multiobjetivo, el cual llega a un compromiso entre las variables a optimizar.

***Fase 3:*** Es la última iteración de este módulo y conseguimos que se puedan optimizar uno, dos o tres objetivos. Además se usan los valores obtenidos del profiling, lo cual nos permite mejorar la función de evaluación del algoritmo multiobjetivo, ya que antes sólo considerábamos para optimizar el tipo de la variable, mientras que ahora tenemos en cuenta el uso que se hace de dicha variable en su entorno.

**Módulo 3 "Unidad Visual":** En este módulo implementamos un interfaz grafica donde se recogerán todos los datos del usuario. Consta de las siguientes clases:

- **UnidadVisual:** Es la clase que implementa el interfaz principal de la aplicación. Se recogen las siguientes opciones:
    - Indicar los objetivos de optimización.
    - Indicar las restricciones de rendimiento, memoria o uso de energía.
    - Para la ejecución del algoritmo genético se pueden cambiar algunos parámetros como el tamaño de la población, el número de iteraciones del algoritmo, así como el número de variables a optimizar.
    - En cuanto a la memoria, es posible fijar el tamaño de la memoria caché y de la memoria principal de la arquitectura objetivo.
    - La ubicación del archivo de profiling.
    - Salvar los resultados de la ejecución del algoritmo en un archivo de texto.

- **UnidadResultados:** En este interfaz se muestran los resultados de la ejecución del algoritmo. Para cada objetivo de optimización se muestran todas las variables y su estructura de datos correspondiente obtenida del algoritmo. Además de ello se muestran los resultados de energía, memoria y consumo para cada objetivo de optimización.

- **UnidadAyuda:** Es la interfaz encargada de proporcionar ayuda sobre el interfaz *UnidadVisual*.

- **DatosVisual:** Es una clase estática necesaria para almacenar los datos proporcionados por el usuario, recogidos en la clase *UnidadVisual*. Dichos datos son necesarios en otras clases de la aplicación.

***Fase 1:*** Las primeras interfaces usadas para mostrar los resultados de la aplicación eran mediante consola.

***Fase 2:*** A medida que íbamos ampliando el proyecto y viendo las distintas opciones que se podían aplicar al algoritmo, empezamos a implementar un interfaz gráfico. Éste permitía elegir los distintos objetivos a optimizar, el número de vueltas del algoritmo y el tamaño de la población.

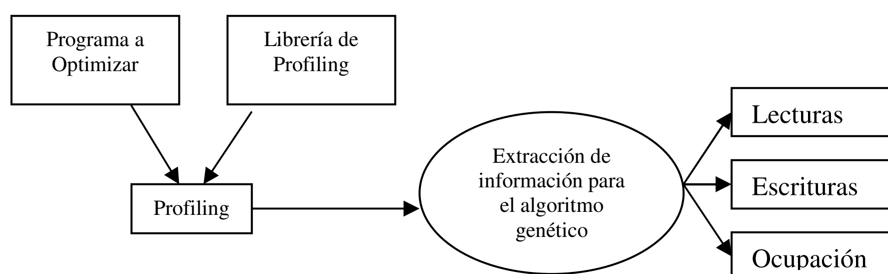
***Fase 3:*** En esta fase se añadieron el resto de funcionalidades que están en la actual interfaz. Se añadió la posibilidad de buscar la ruta del profiling, salvar los resultados, las restricciones y una mejora de la presentación de los resultados en la *UnidadResultados*.

## *5.2- Funcionamiento de la aplicación*

Partimos de una aplicación **A** con uso intensivo de memoria dinámica, ejecutable en un sistema empotrado, cuya gestión de memoria queremos optimizar, en base a estas tres métricas principales:

- **Rendimiento.**
- **Uso de memoria.**
- **Consumo energético.**

Para dicha aplicación **A**, obtenemos un perfil de ejecución, en el que tenemos, para cada variable, el número de lecturas y escrituras realizadas para la ejecución de dicho programa:




Una vez obtenido el fichero, lo recorremos para obtener la información necesaria para nuestra optimización, siendo ésta:

- El número de variables usadas a optimizar.
- El número de lecturas y escrituras para cada una de ellas.
- El nivel de ocupación de la variable.
- El tamaño máximo.
- El tamaño de los elementos básicos almacenados en cada variable.

Almacenamos dicha información para cada variable en una clase "Resultado", y guardamos todos ellos en una lista de resultados (clase ListaResultados).

Con esta información ya podemos empezar la ejecución del algoritmo genético en si. Para ello, creamos una población de X individuos (X deberá ser ajustado

mediante sucesivas pruebas, para obtener una población con un tamaño que nos permita obtener resultados en un tiempo, si no óptimo, sí relativamente bueno).

Cada individuo será creado de forma aleatoria y su tamaño es igual a multiplicar 13, que son los bits a optimizar para cada variable, por el número de variables con las que trabaje el programa de pruebas:

| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

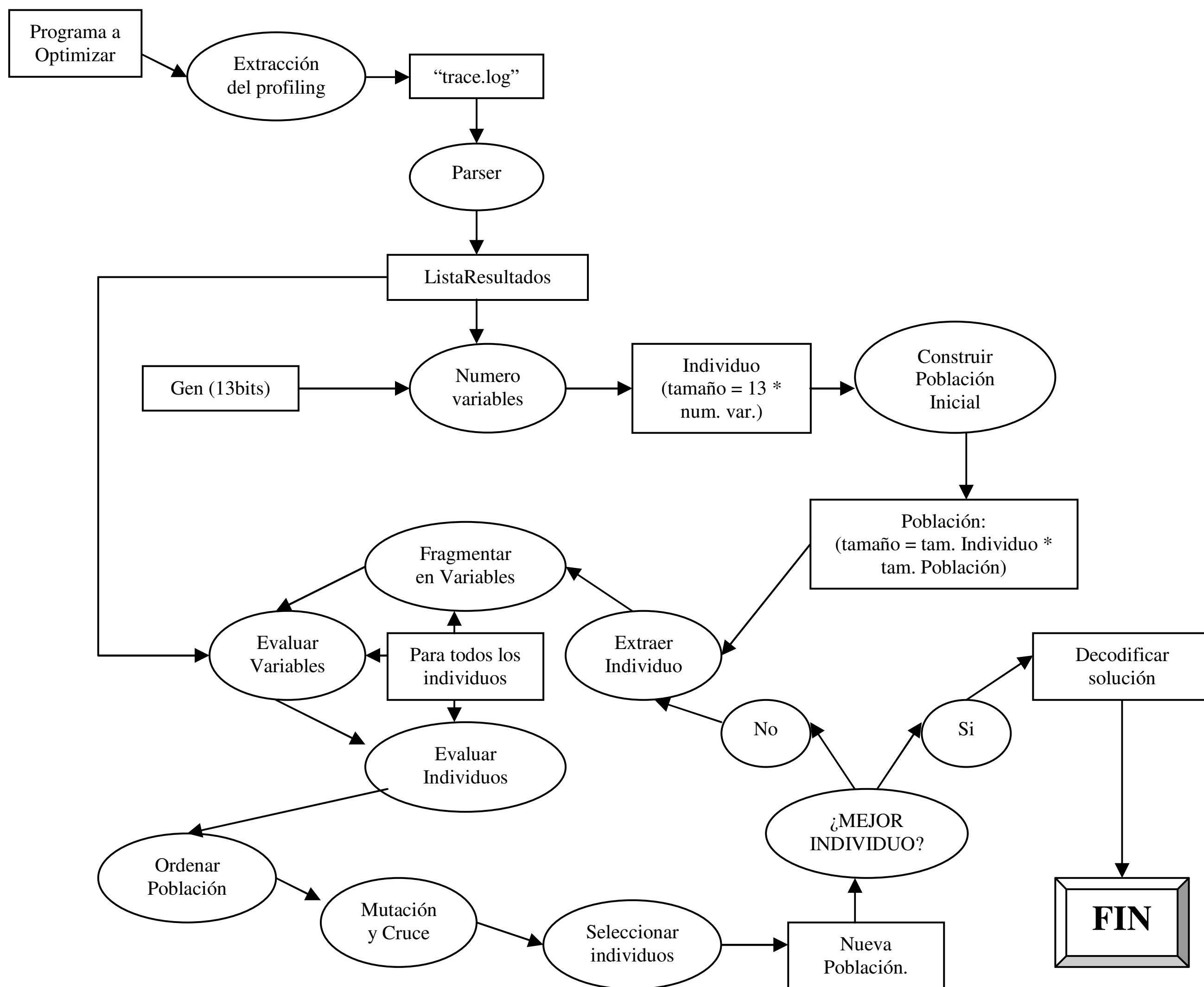
**X** Número de Variables.

Empieza la ejecución del algoritmo genético. Para cada individuo de la población, hay que conseguir su valor de evaluación. Para ello:

- Fragmentamos al individuo en las distintas variables.
- Para cada variable, calculamos su valor de Rendimiento, Memoria y Energía en la ejecución del programa (con la ayuda de la información de la lista de Resultados).
- Sumamos los valores de todas las variables para todas las métricas del individuo.
- Calculamos su valor de evaluación global, dependiendo del tipo de algoritmo que estemos usando.

Una vez que ya tenemos esa información para cada individuo, se realizan las funciones genéticas (selección, mutación y cruce) sobre los individuos de la población, con lo cual generamos la siguiente generación. Mediante sucesivas iteraciones del algoritmo (VEGA) va evolucionando las poblaciones hacia unos objetivos propuestos por el usuario. Una vez que ya tenemos la última población (óptima) decodificamos los individuos y mostramos el resultado obtenido.

A continuación mostramos un esquema que describe el flujo de ejecución de nuestra aplicación:



# Capítulo 6: Datos de Salida.

## 6.1- Significado de los valores.

El fin ultimo de nuestro proyecto es obtener las implementaciones óptimas de las EDD´s para las variables dinámicas extraídas del perfil en ejecución de a aplicación objetivo a optimizar.

Nuestro algoritmo devuelve una población óptima que guarda la información necesaria para tener el objetivo anteriormente descrito. Para conocer las implementaciones obtenidas, lo primero que hay que realizar es la fragmentación de individuo en las variables a optimizar e identificar cada fragmento de este individuo con dichas variables. Una vez realizada esta fragmentación el siguiente paso es decodificar la información de ese fragmento obteniendo de este modo los datos a optimizar deseados.

Para interpretar cada fragmento de cada individuo resultante de la última población nos basamos en la siguiente estructura:

| 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

Bits 0-1: Número de niveles de la estructura de datos a probar.
Bits 2-5: Campos básicos.
Bits 6-8: Número de elementos.
Bits 9-12: Estructura de datos a usar, la cual corresponde con la siguiente codificación:

- 0000 : AR(t)
- 0001 : AR(t + ref)
- 0010: PAR(t)
- 0011: PAR(t + ref)
- 0100: SLL(t + cla)
- 0101: DLL(t + cla)
- 0110: PAR(AR(t))
- 0111: PAR(AR(t + ref))
- 1000: SLL(AR(t)+ cla)
- 1001: SLL(AR(t + ref) + cla)
- 1010: DLL(AR(t) + cla)

- 1011: DLL(AR(t + ref) + cla)
- 1100: SLL(PAR(t) + cla)
- 1101: SLL(PAR(t + ref) + cla)
- 1110: DLL(PAR(t) + cla)
- 1111: DLL(PAR(t + ref) + cla)

A partir de estos datos presentamos el resultado de nuestra aplicación para cada variable a optimizar.

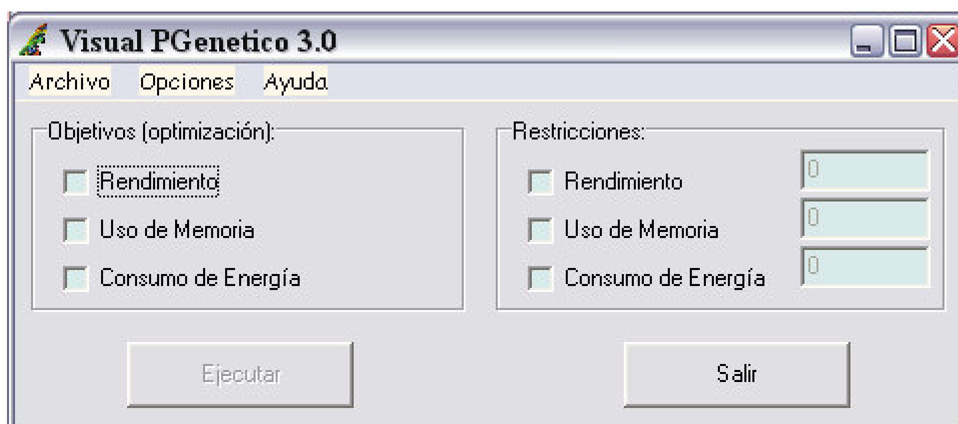
# Capítulo 7: Un Ejemplo.

En este capítulo se hará una demostración “paso a paso” de un ejemplo de ejecución completa del programa, procurando usar todas las funcionalidades de éste.

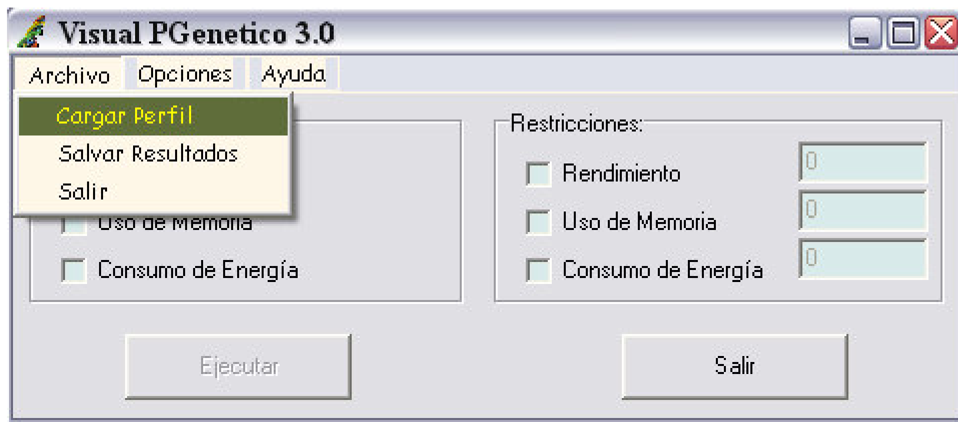
## *7.1- Optimización de “VDrift”.*

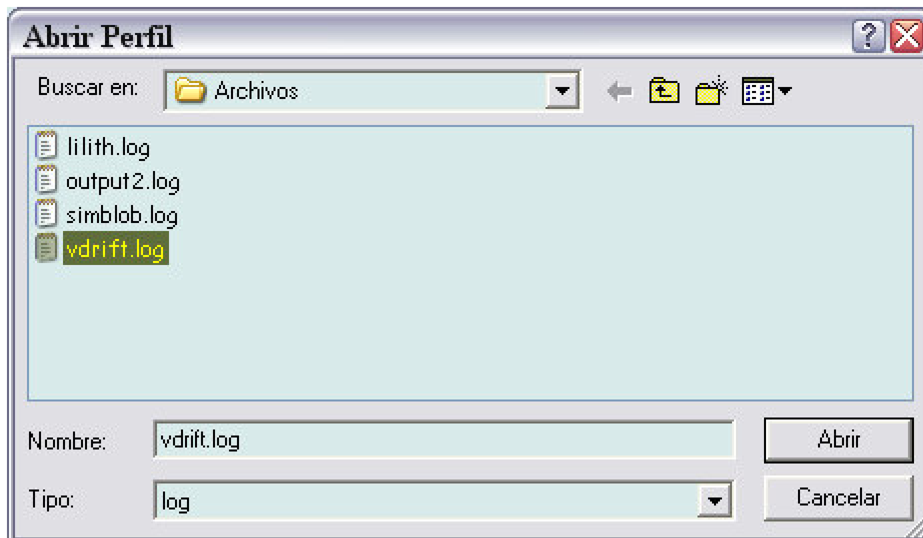
Vamos a optimizar las estructuras de datos de un juego multimedia, un simulador de conducción de coches (http://vdrift.net ).

Para comenzar abrimos el programa y nos encontramos con el formulario principal.

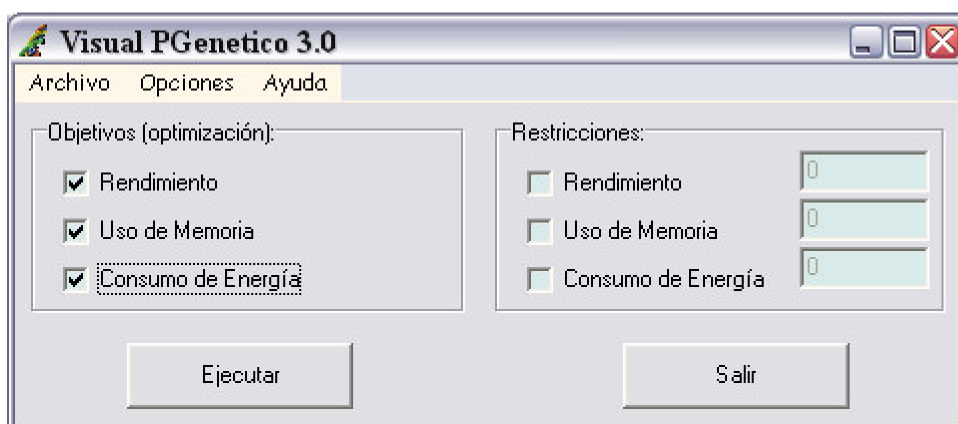



Debemos cargar el “profiling” de la aplicación a optimizar. Éste se consigue tras una ejecución de la aplicación y nos da valores muy importantes a la hora de optimizarla como el número de lecturas y escrituras de cada estructura de datos o el estado de ocupación en cada momento de dicha estructura. El archivo debe tener extensión *.log* y se puede cargar desde el menú Archivo.

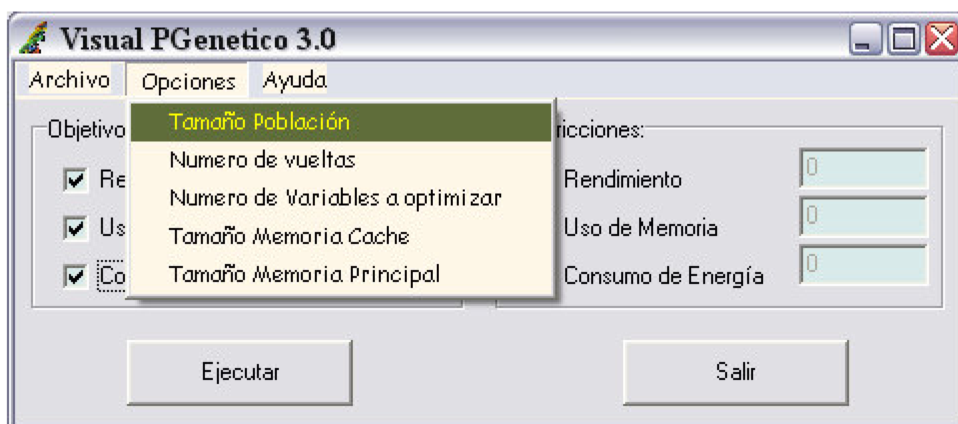

Visual PGenetico 3.0
Archivo
Opciones
Ayuda
Cargar Perfil
Salvar Resultados
Salir
Uso de Memoria
Consumo de Energía
Restricciones:
Rendimiento
Uso de Memoria
Consumo de Energía
0
0
0
Ejecutar
Salir

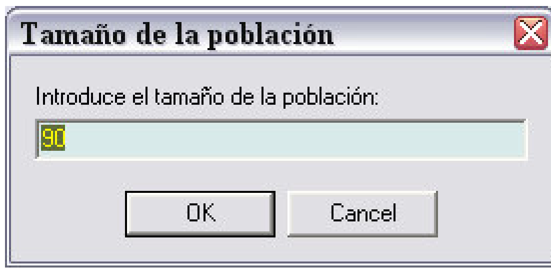

Abrir Perfil
Buscar en:
Archivos
lilith.log
output2.log
simblob.log
vdrift.log
Nombre:
vdrift.log
Abrir
Tipo:
log
Cancelar

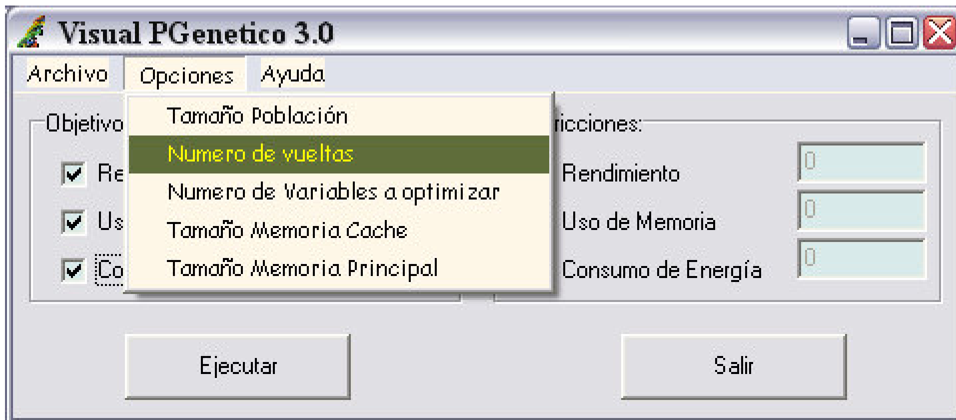
Una vez cargado el archivo de "profiling" deberemos marcar los campos a optimizar, rendimiento, uso de memoria o energía. Se puede mejorar un campo, dos de ellos o los tres, según las necesidades. Se seleccionan con un simple "tic" en el formulario principal.

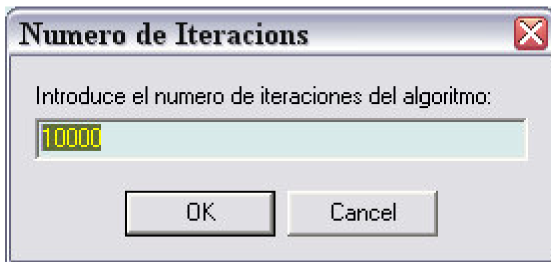


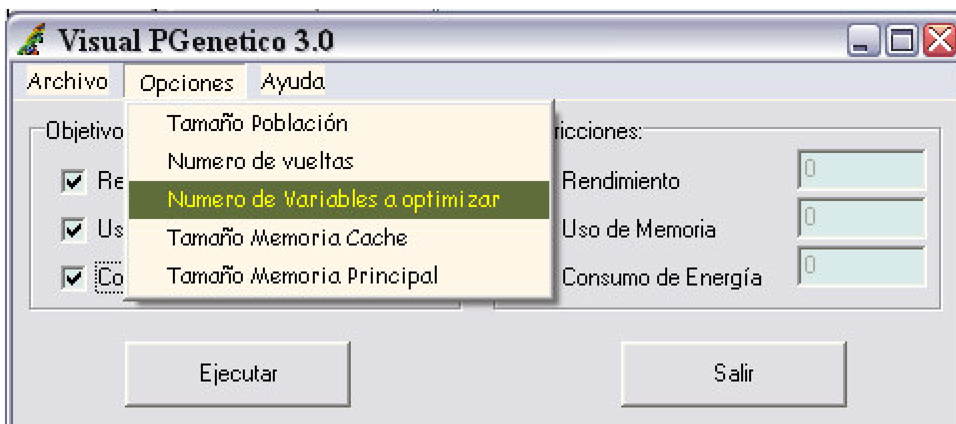

Ahora debemos insertar los parámetros para el algoritmo genético que trata de optimizar en los campos que hemos marcado anteriormente. Estos son: Tamaño de la población del algoritmo, número de vueltas (iteraciones) que dará el algoritmo y número de variables a optimizar. Este último parámetro selecciona cuantas variables se optimizarán independientemente del número de variables que haya en el "profiling". Las variables se escogerán por su importancia, es decir se seleccionarán las que más afecte su optimización al rendimiento final de la aplicación. Se desecharán variables con pocos accesos, ya que por mucho que se optimice la estructura de datos para ellas no se mejorarán excesivamente los resultados globales.

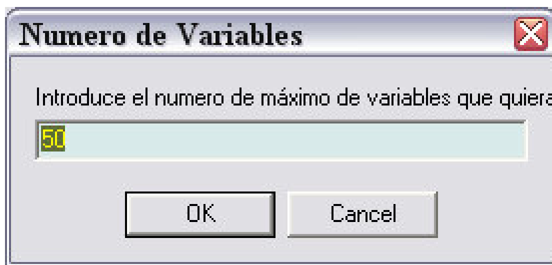



Tamaño de la población
Introduce el tamaño de la población:
90
OK
Cancel

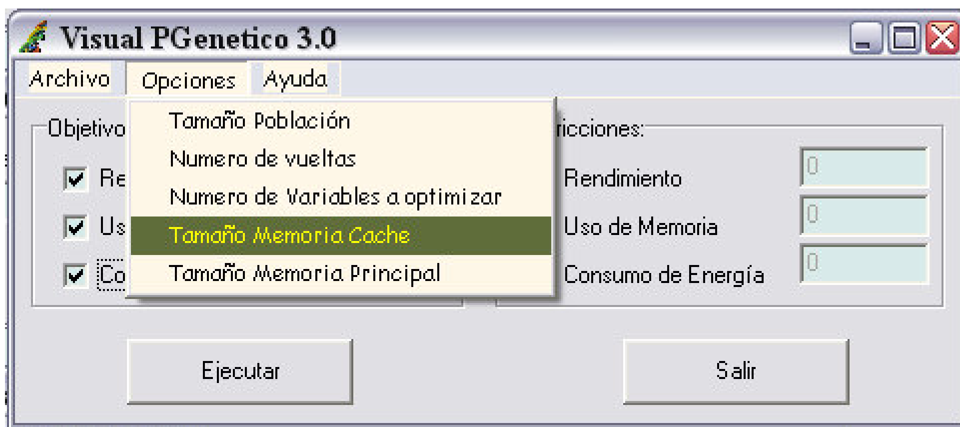

Visual PGenetico 3.0
Archivo
Opciones
Ayuda
Tamaño Población
Numero de vueltas
Numero de Variables a optimizar
Tamaño Memoria Cache
Tamaño Memoria Principal
Rendimiento
Uso de Memoria
Consumo de Energía
Ejecutar
Salir

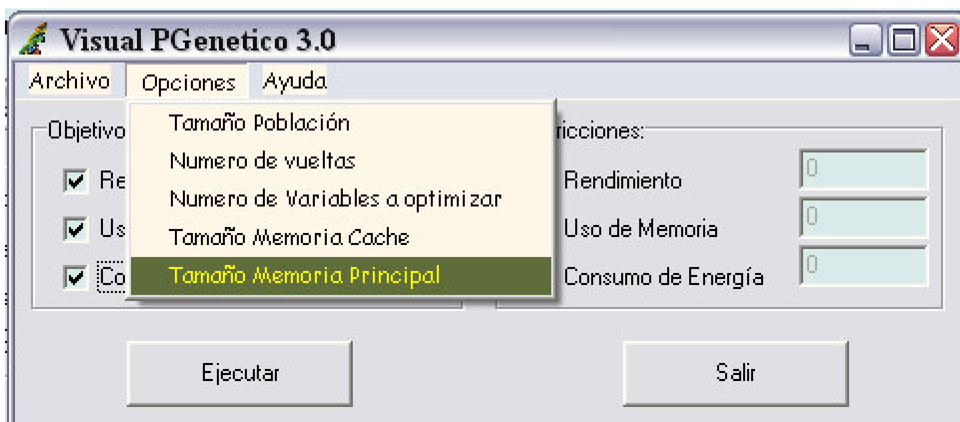

Numero de Iteracions
Introduce el numero de iteraciones del algoritmo:
10000
OK
Cancel

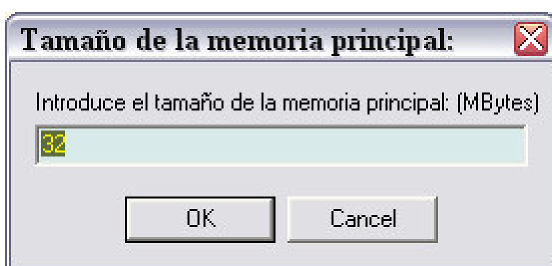

Visual PGenetico 3.0
Archivo
Opciones
Ayuda
Tamaño Población
Numero de vueltas
Numero de Variables a optimizar
Tamaño Memoria Cache
Tamaño Memoria Principal
Rendimiento
Uso de Memoria
Consumo de Energía
Ejecutar
Salir



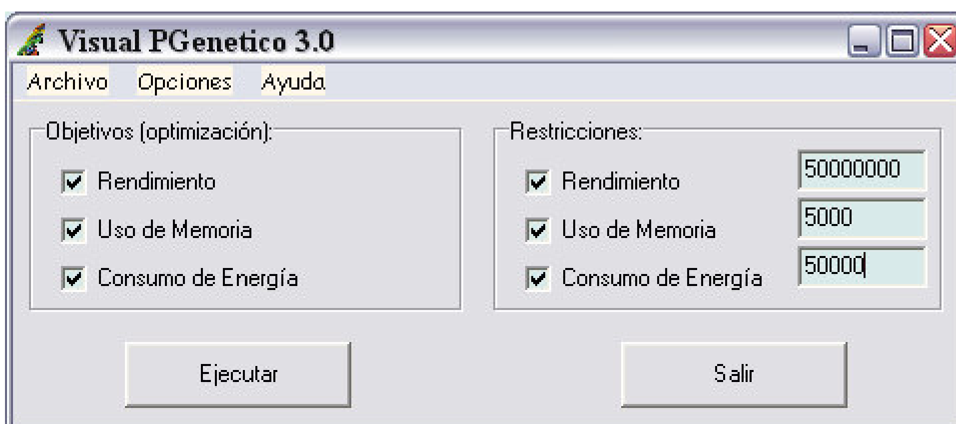

También se deben introducir los datos de los tamaños de memoria, tanto caché como principal, para los que se quiere optimizar el sistema.

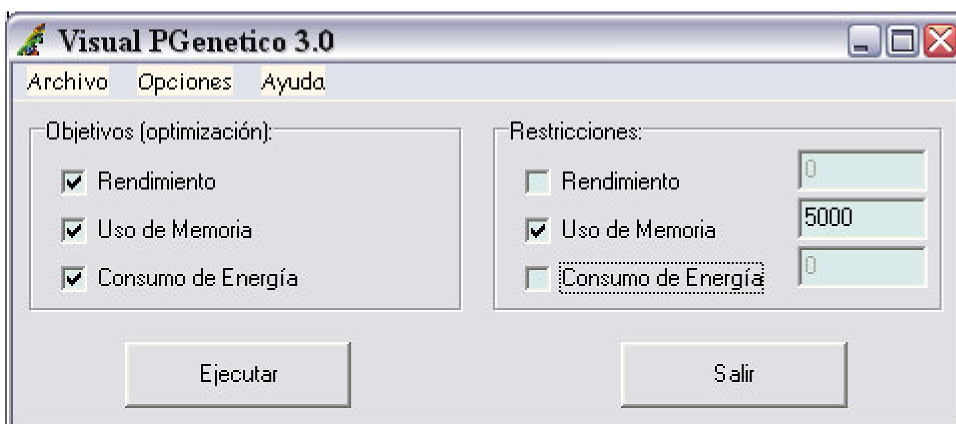




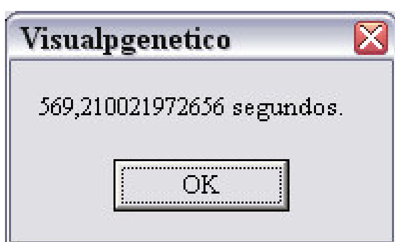


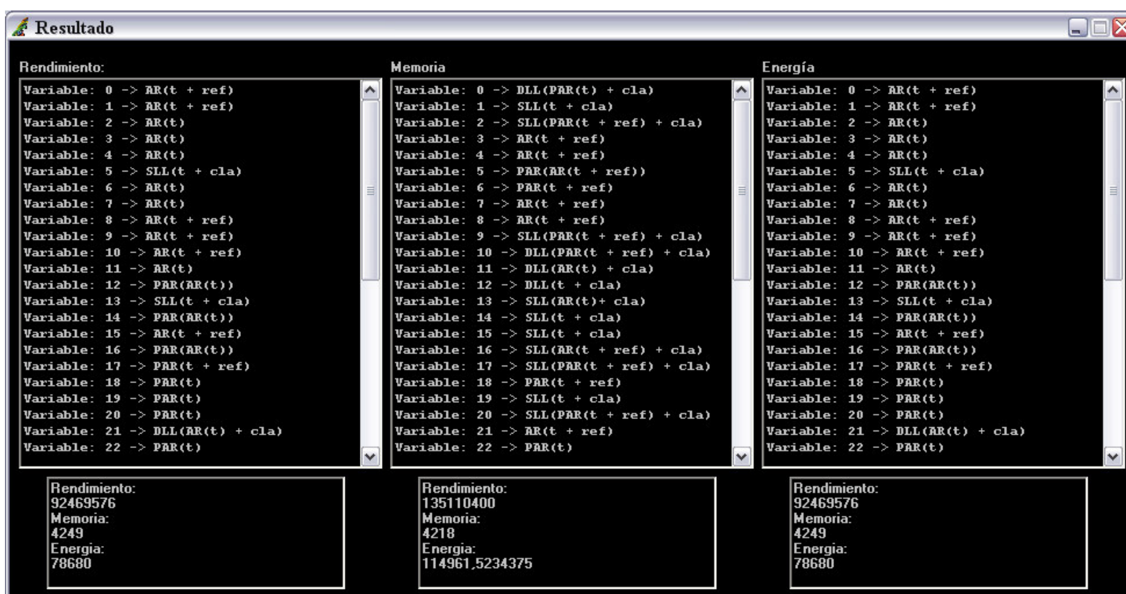

Por último se le pueden introducir las restricciones del sistema. Estos son valores máximos de rendimiento, uso de memoria y consumo de energía. Estos valores no deben ser sobrepasados en el resultado de la optimización porque se supone que son limitaciones físicas del sistema (o de especificación). También se puede elegir poner una, dos o las tres restricciones disponibles, a elección del usuario.



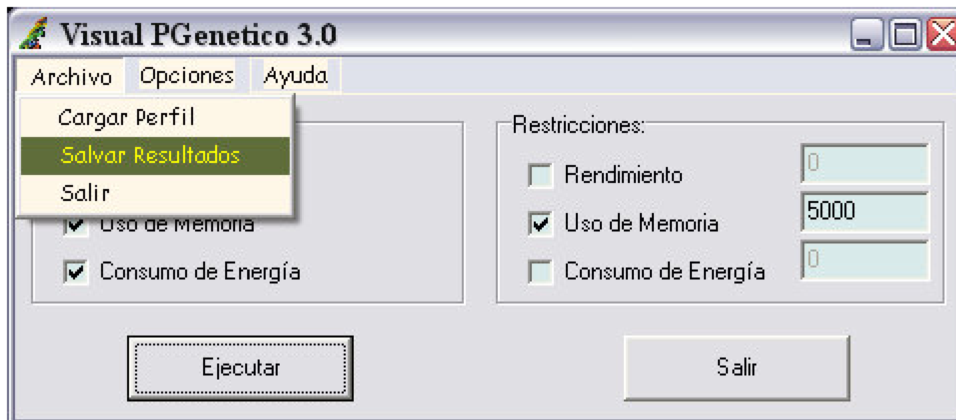

Por último se ejecuta el programa (mediante el botón "Ejecutar" del formulario principal). En este momento es cuando el programa se encarga de leer el archivo de "profiling" seleccionado, filtrando el número de variables que hemos insertado. Después se ejecuta el algoritmo genético y finalmente muestra una ventana con el tiempo que ha tardado. Al aceptar esta ventana saldrán los resultados de la optimización realizada.

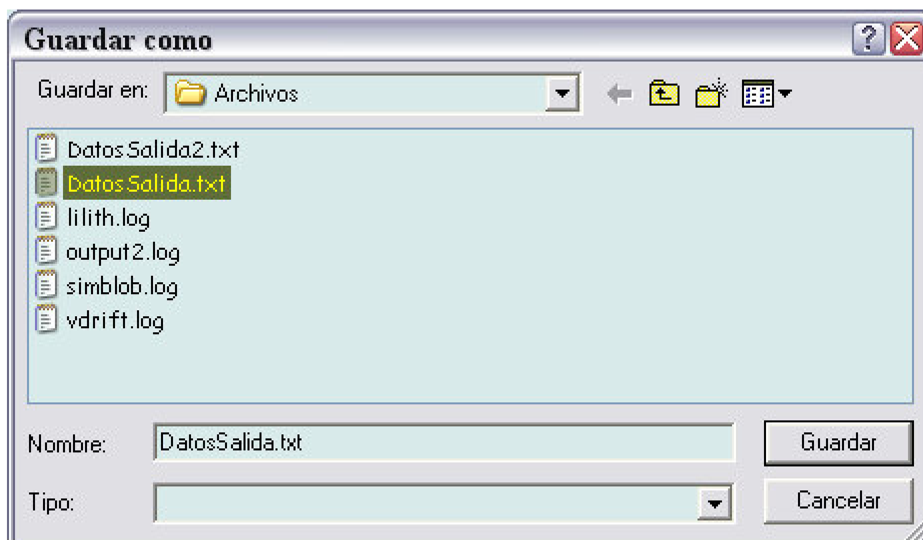


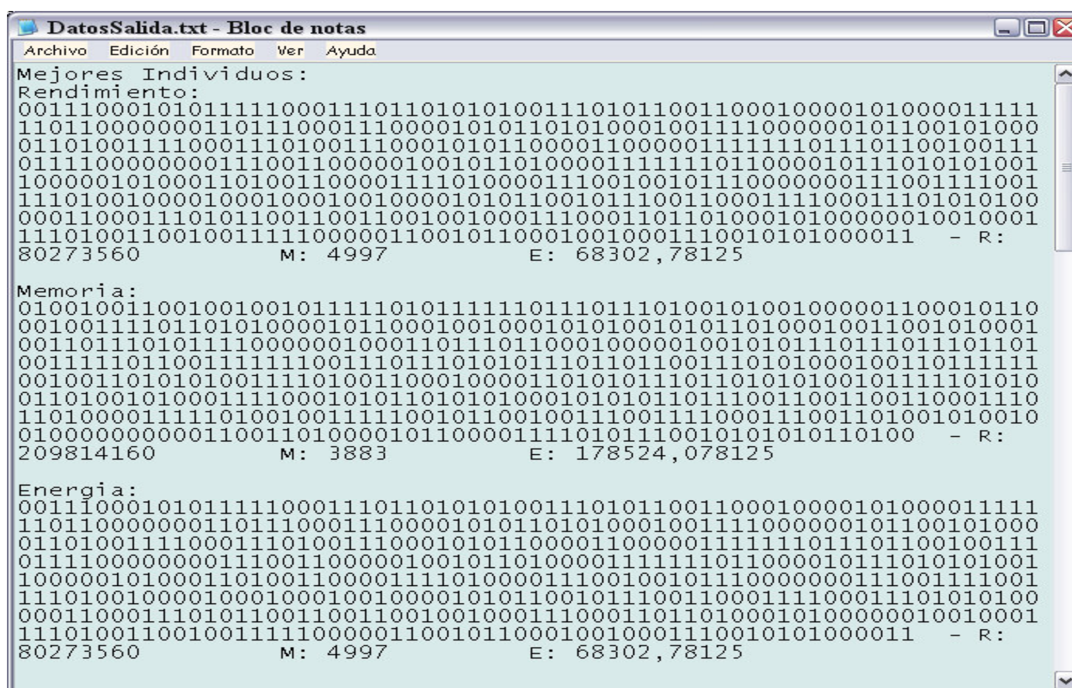

La ventana de resultados muestra las estructuras de datos óptimas para la optimización de cada uno de los campos optimizables. Se muestra el identificador de la variable seguido del tipo de estructura sugerida para ella. En el cuadro inferior se muestra el resultado de los tres campos (rendimiento, memoria y energía) para las estructuras sugeridas.

Una vez que se han obtenido los resultados de la optimización se pueden salvar los datos a un archivo de texto para almacenarlos una vez se haya cerrado el programa.



En este archivo se guardan los mejores individuos codificados en binario (con los 13 bits para cada variable, que representan su estructura, número de niveles, campos básicos y número de elementos, tal como se explica en la generación del Individuo) con los resultados obtenidos para los campos a optimizar.



DatosSalida.txt - Bloc de notas

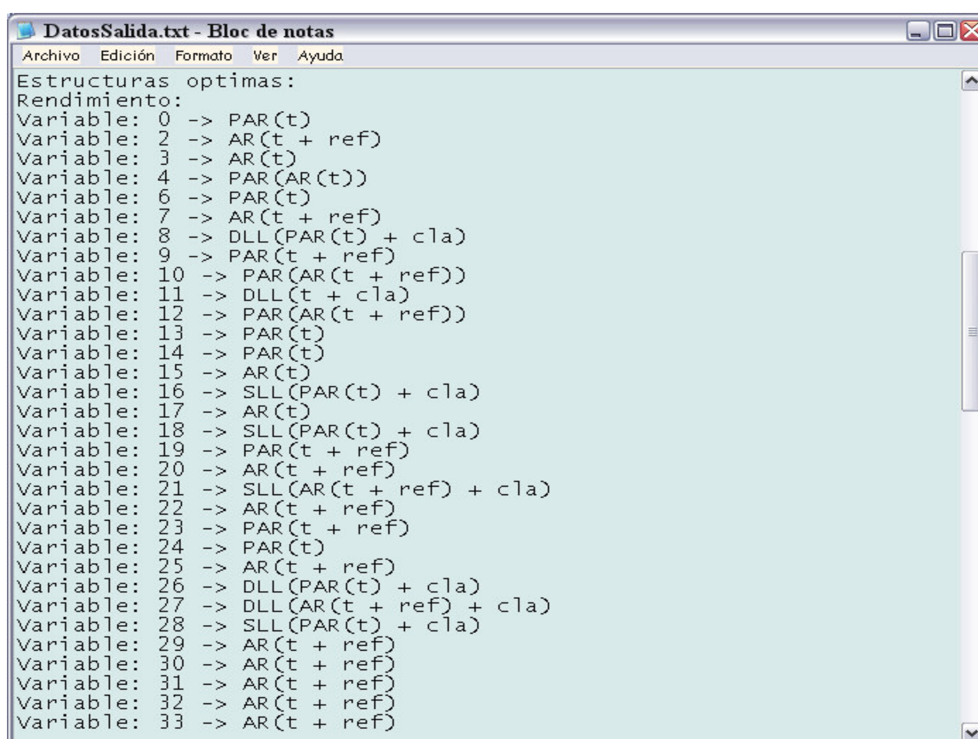
Archivo Edición Formato Ver Ayuda

```
Mejores Individuos:
Rendimiento:
0011100010101111110001110110101010011101011001100010000101000011111
1101100000001101110001110000101011010100010011110000000101100101000
0110100111100011101001110001010110000110000011111110111011001001111
0111100000000111001100000100101101000011111110110000101110101010011
1000001010001101001100001111010000111001001011100000000111001111001
1110100100001000100010010000101011001011100110001111000111010101000
0001100011101011001100110010010001110001101101000101000000010010001
111010011001001111110000011001011000100100011100101010100000111   - R:
80273560          M: 4997          E: 68302,78125

Memoria:
0100100110010010010111110101111110111011101001010010000011000101110
0010011110110101000010110001001000101010010101101000100110010100001
0011011101011110000000100011011101100010000010010101110111011101101
0011111011001111111100111011101010111011011001110101000100110111111
0010011010101001111010011000100001101010111011010101001011111010110
0110100101000111100010101101010100010101011011100110011001100011100
1101000011111010010011111001011001001110011110001110011010010100100
01000000000011001101000010110000111101011100101010101011010000 - R:
209814160         M: 3883          E: 178524,078125

Energia:
0011100010101111110001110110101010011101011001100010000101000011111
1101100000001101110001110000101011010100010011110000000101100101000
0110100111100011101001110001010110000110000011111110111011001001111
0111100000000111001100000100101101000011111110110000101110101010011
1000001010001101001100001111010000111001001011100000000111001111001
1110100100001000100010010000101011001011100110001111000111010101000
0001100011101011001100110010010001110001101101000101000000010010001
111010011001001111110000011001011000100100011100101010100000111   - R:
80273560          M: 4997          E: 68302,78125
```

Después se muestran las estructuras sugeridas en cada uno de los campos a optimizar (igual que en la ventana de resultados).

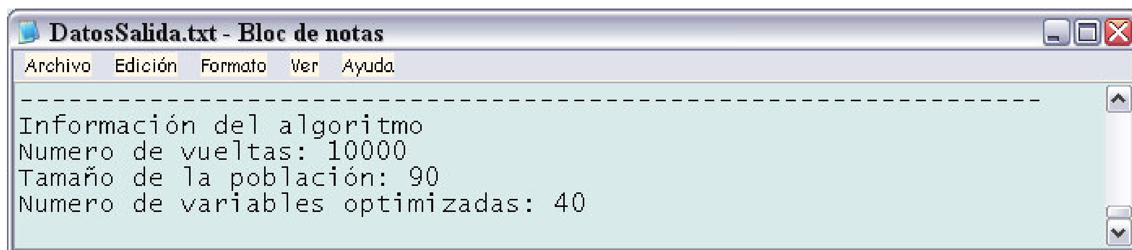


DatosSalida.txt - Bloc de notas

Archivo Edición Formato Ver Ayuda

```
Estructuras optimas:
Rendimiento:
Variable: 0 -> PAR(t)
Variable: 2 -> AR(t + ref)
Variable: 3 -> AR(t)
Variable: 4 -> PAR(AR(t))
Variable: 6 -> PAR(t)
Variable: 7 -> AR(t + ref)
Variable: 8 -> DLL(PAR(t) + cla)
Variable: 9 -> PAR(t + ref)
Variable: 10 -> PAR(AR(t + ref))
Variable: 11 -> DLL(t + cla)
Variable: 12 -> PAR(AR(t + ref))
Variable: 13 -> PAR(t)
Variable: 14 -> PAR(t)
Variable: 15 -> AR(t)
Variable: 16 -> SLL(PAR(t) + cla)
Variable: 17 -> AR(t)
Variable: 18 -> SLL(PAR(t) + cla)
Variable: 19 -> PAR(t + ref)
Variable: 20 -> AR(t + ref)
Variable: 21 -> SLL(AR(t + ref) + cla)
Variable: 22 -> AR(t + ref)
Variable: 23 -> PAR(t + ref)
Variable: 24 -> PAR(t)
Variable: 25 -> AR(t + ref)
Variable: 26 -> DLL(PAR(t) + cla)
Variable: 27 -> DLL(AR(t + ref) + cla)
Variable: 28 -> SLL(PAR(t) + cla)
Variable: 29 -> AR(t + ref)
Variable: 30 -> AR(t + ref)
Variable: 31 -> AR(t + ref)
Variable: 32 -> AR(t + ref)
Variable: 33 -> AR(t + ref)
```

Y por último, los parámetros que se le han dado al algoritmo genético en la prueba realizada.

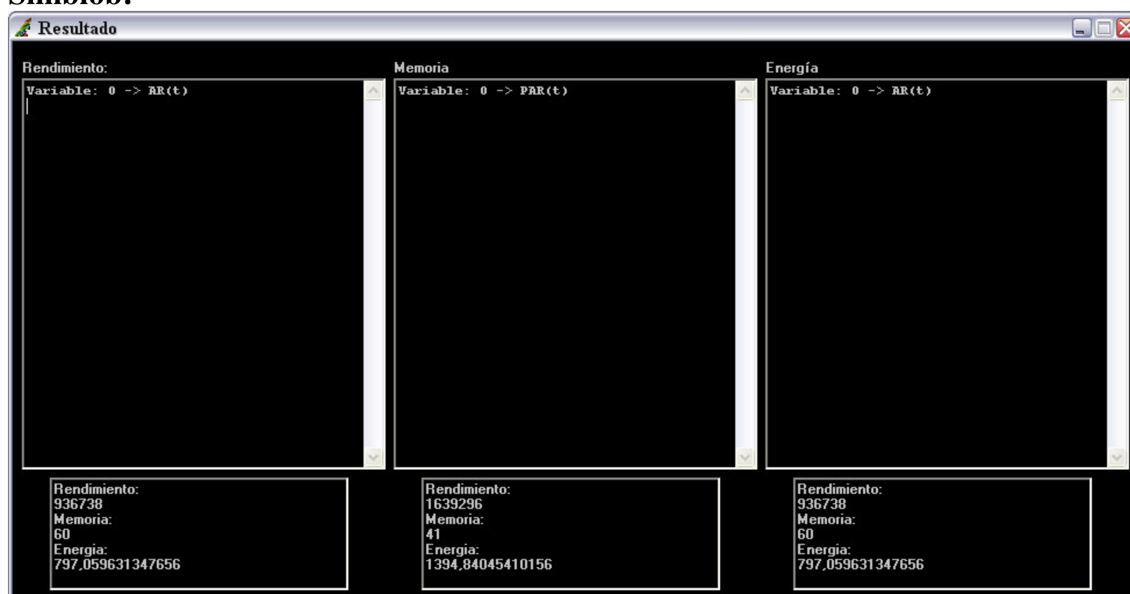


## *7.2- Resultados en otras aplicaciones.*

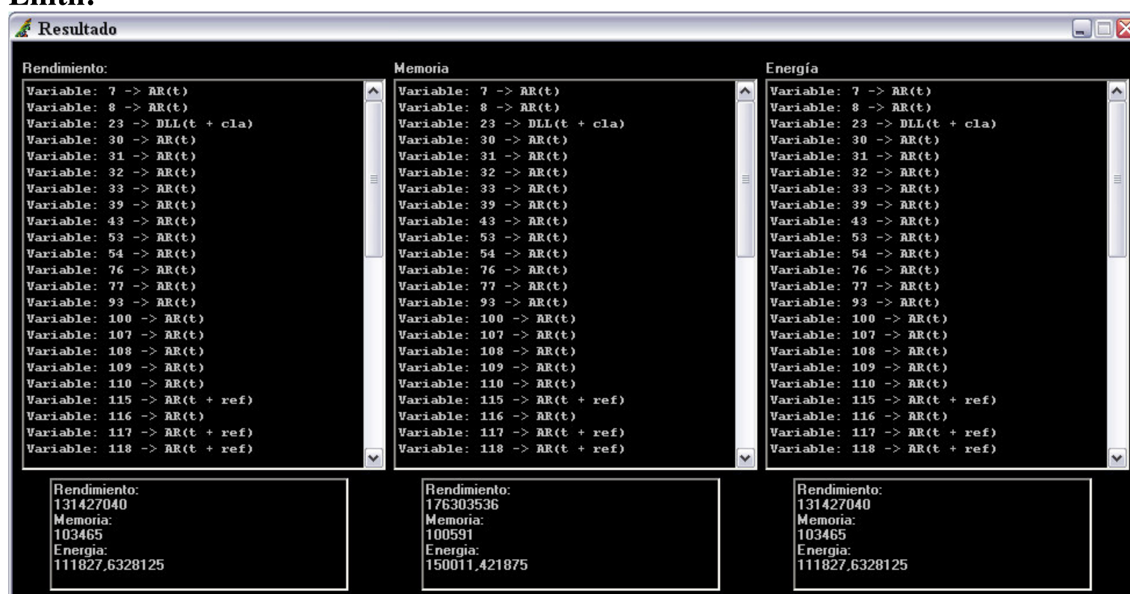
También hemos trabajado con otros dos programas multimedia: el "Simblob", y el "Lilith". El primero es un simulador físico para fluidos moviendose en superficies sólidas (http://simblob.sourceforge.net ) y el segundo un motor de creación de mundos 3D con efectos climáticos, búsqueda de rutas con A*, transiciones entre el día y la noche, etc. (http://www.grinninglizard.com/lilith ).

Los resultados obtenidos para estos dos programas son los siguientes:

**Simblob:**




**Lilith:**



# Capítulo 8: Conclusiones.

Tras realizar este proyecto y analizar sus resultados, podemos describir que objetivos hemos conseguido y que características principales hemos logrado implementar y, por otro lado, que características contemplamos introducir y que finalmente no forman parte del proyecto final, bien por que se salían de los objetivos iniciales, por su elevada dificultad o por la eterna falta de tiempo.

## *Objetivos conseguidos.*

El objetivo principal del proyecto era construir un algoritmo genético multiobjetivo para la optimización de la memoria dinámica en sistemas empotrados de última generación.

Hemos conseguido implementar un algoritmo genético, tanto simple, para un único objetivo, como multiobjetivo (VEGA), para dos o tres objetivos a optimizar.

Para ello, hemos profundizado en el tema de la programación evolutiva, el cual desconocíamos casi por completo, y a través de artículos y libros (citados en la bibliografía) hemos comprendido las bases y sido capaces de implementar un algoritmo VEGA, para dos o tres objetivos.

También hemos estudiado y analizado la arquitectura de los sistemas empotrados de última generación, para comprender qué estábamos realizando y el por qué de las limitaciones a las que se ven sometidos. Con esta información, y sabiendo que los tiempos necesarios para realizar optimizaciones en estos entornos de forma "manual" es del orden de días [3], y comprendiendo las reglas del mercado, justificamos la idoneidad de nuestro trabajo, con el cual podemos obtener los resultados en tiempos del orden de minutos.

Para realizar nuestro trabajo, necesitábamos los archivos de profiling, proporcionados por el IMEC, mediante una colaboración con un estudiante de doctorado de dicho centro (Christophe Poucet) a través de David Atienza, profesor de la facultad y sobre cuya metodología de optimización hemos trabajado. Dichos archivos

han tenido un formato más bien variable a lo largo del proyecto, lo cual nos ha permitido:

- Proveer información más exacta sobre los accesos de lectura/escritura efectivos al algoritmo de optimización.

- Permitir incluir resultados sobre iteradores de variables, que en las aplicaciones diseñadas con objetos más recientes son muy habituales para acceder a los datos.

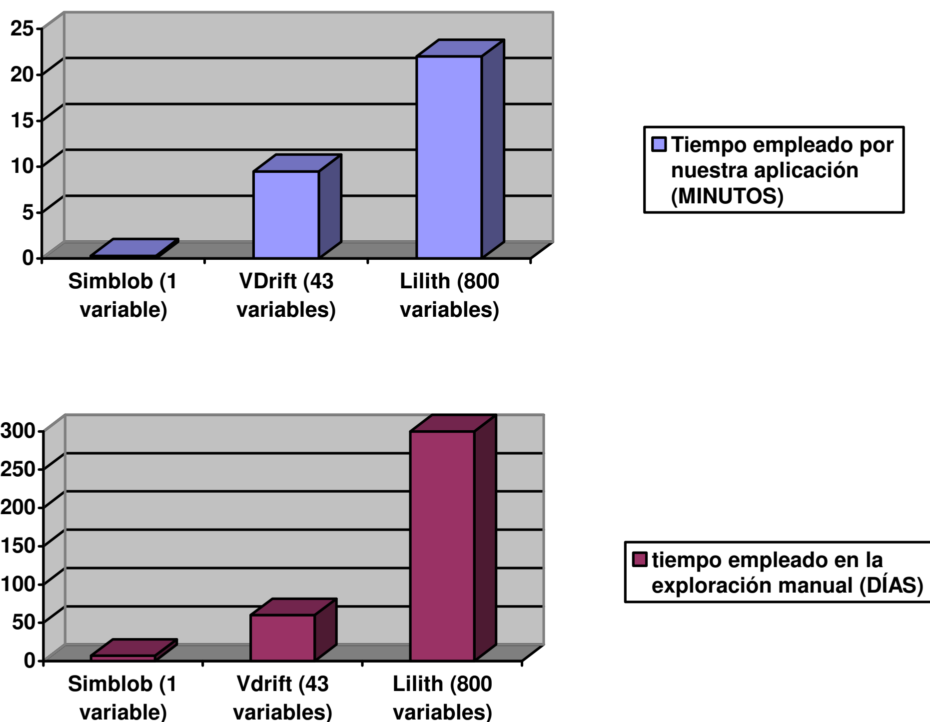
Así que hemos ido adaptándonos a ellos y depender de terceros, siendo nuestro proyecto capaz de reconocer varios formatos de trazas.

Los resultados obtenidos han sido contrastados con los obtenidos al hacer la optimización de forma manual por los miembros del IMEC, llegando a conclusiones similares en unos tiempos muy inferiores.

Tras ver los resultados y hablarlos con el IMEC hemos llegado a la conclusión de que el problema para compararlos con los valores de energía y rendimiento obtenidos allí es que la configuración de la jerarquía de memoria del IMEC es más compleja que la utilizada en este proyecto, ya que, allí, usan dos niveles de caché y una Scratchpad de 32KB. No obstante, hemos podido comprobar que gran parte de las variables (un 75-85% de media) poseen los mismos tipos de estructuras de datos que han sido elegido en el IMEC tras un proceso iterativo de optimización manual muy costoso en tiempo (ver más abajo). Por ejemplo, en VDrift, de las 43 variables 35 salen con la misma estructura base (un 80%) y en Simblob son correctas las 3 opciones óptimas de implementación para cada métrica estudiada (energía, rendimiento y uso de memoria) para la variable que dinámica que hay (100%).

En resumen, los resultados sacados por el algoritmo son correctos, sobre todo teniendo en cuenta que en una exploración exhaustiva, se requeriría al menos entre 1-2 meses para optimizar cada aplicación usada [3]. Más aún, en el IMEC, según nos han comunicado, han dedicado entre 2-3 meses para cada una aplicación de las aplicaciones utilizadas en esta memoria.

Mostramos una comparativa de los tiempos de ejecución de nuestro algoritmo sobre las aplicaciones de prueba dependiendo del número de variables a optimizar enfrentados con los tiempos utilizados en la actualidad para la búsqueda exhaustiva manual de las estructuras de datos a utilizar.




La implementación del entorno gráfico añade funcionalidad y simplicidad a nuestro proyecto, ya que podemos cargar el perfil de la aplicación que se desea optimizar y elegir que objetivos se optimizan, restricciones para cualquiera de ellos, elegir la precisión del algoritmo genético, así como la cantidad de memoria de la arquitectura objetivo a optimizar.

También podemos almacenar los resultados de la optimización en un archivo de texto, para un posterior estudio de los resultados sin tener que volver a correr el programa.

## *Mejoras futuras.*

En nuestro proyecto, finalmente, sólo implementamos un tipo de algoritmo genético multiobjetivo (VEGA) y sería interesante poder contrastar sus resultados implementando otros algoritmos.

Nuestra arquitectura objetivo tiene sólo un nivel de caché y otro de memoria principal. Para calcular los fallos de memoria caché, nos hemos basado en datos estadísticos fiables, aunque no del todo exactos. Las mejoras posibles en este caso serían poder elegir una jerarquía de memoria más avanzada, con varios niveles de caché, definiendo su asociatividad. Además, una vez definida más en detalle esta arquitectura de memoria, sería positivo usar herramientas que simulasen la ejecución de los programas a optimizar y generasen los fallos de memoria caché en sus distintos niveles de forma más exacta.

Otra inclusión adicional en la jerarquía de memoria simulada podría ser el uso de memorias on-chip controladas por software (memorias Scratchpad), que cada vez son más habituales en los sistemas empotrados debido a sus mejoras en cuanto al consumo de energía y rendimiento con respecto a las memorias caché [7].

Actualmente, nosotros devolvemos las implementaciones óptimas de las variables dinámicas de una aplicación. Sería muy interesante, partiendo del perfil en ejecución y del propio código fuente, tras realizar la optimización y saber los resultados óptimos, automatizar estos cambios en el código fuente, sin que fuera necesaria la actuación manual.

# Anexo I: Estructura de clases de la aplicación.

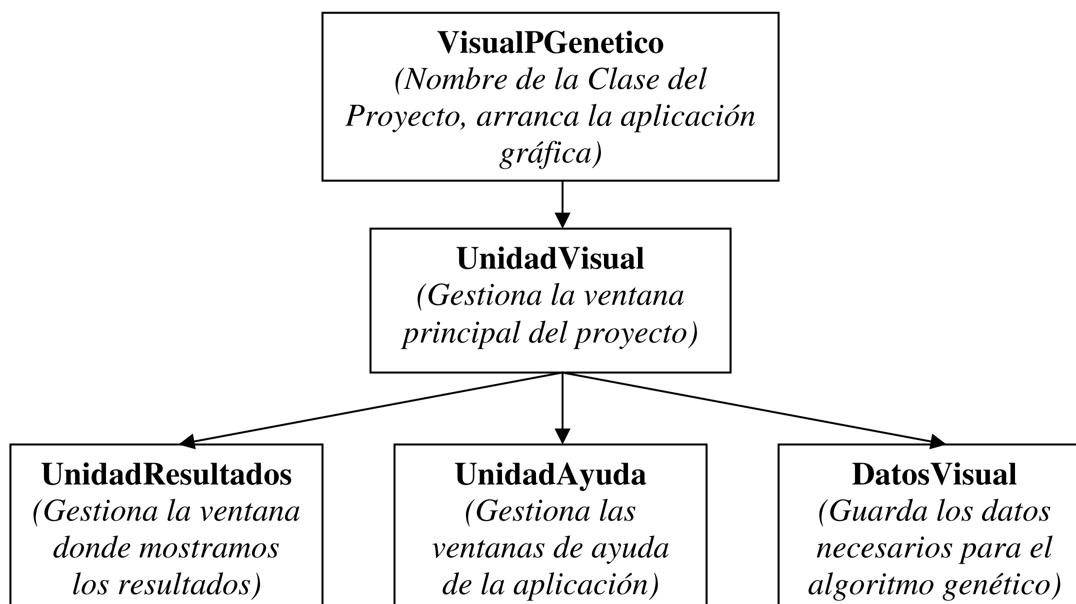
La implementación de nuestro proyecto la hemos realizado en el entorno de programación Borland C++ 5.0, bajo la plataforma Windows. Elegimos este entorno, a pesar de no ser la última versión, por el hecho de estar instalado en los ordenadores de la facultad y por la facilidad con la que se puede crear un entorno gráfico sencillo. Además, todos habíamos trabajado en este entorno con anterioridad.

En cuanto a la división en clases, que se puede consultar en la documentación digital suministrada, se puede diferenciar claramente entre la parte visual y el motor del algoritmo, que se puede usar tanto con dicha implementación grafica como en modo texto.

## *Parte gráfica:*

La parte gráfica esta formada por la clase principal, **UnidadVisual**, en la que se encuentra el formulario inicial, en la que podemos realizar las acciones necesarias para obtener una posible optimización de las estructuras dinámicas de una aplicación, modificando las opciones elegidas, tanto en el algoritmo genético como en la arquitectura objetivo, además de guardar en disco dichos resultados. Además de este formulario, tenemos otros dos, **UnidadAyuda**, encargada de manejar las ventanas de ayuda y mostrarlas en caso de que sea necesario, y la **UnidadResultados**, ventana en la que se muestran los resultados obtenidos tras realizar la optimización sobre un archivo de profiling.

También creamos la clase auxiliar **DatosVisual**, estática, para poder suministrar los datos modificados en el entorno visual (tamaño de la población del algoritmo genético, tamaño de la memoria caché, etc.) a la implementación del algoritmo genético.



## *Motor genético:*

Aparte esta parte gráfica, también tenemos lo que podemos considerar el motor genético, donde creamos las poblaciones de individuos que evolucionan hasta lograr nuestros objetivos.

La clase principal es Poblacion, que, básicamente, crea una lista de objetos de la clase Individuo, a los que se les aplica el algoritmo genético. Esta clase Poblacion, permite construir otra población a partir de sus progenitores, además de proporcionar métodos que nos devuelven los mejores Individuos con respecto a una de las tres opciones que optimizamos (rendimiento, memoria o energía).

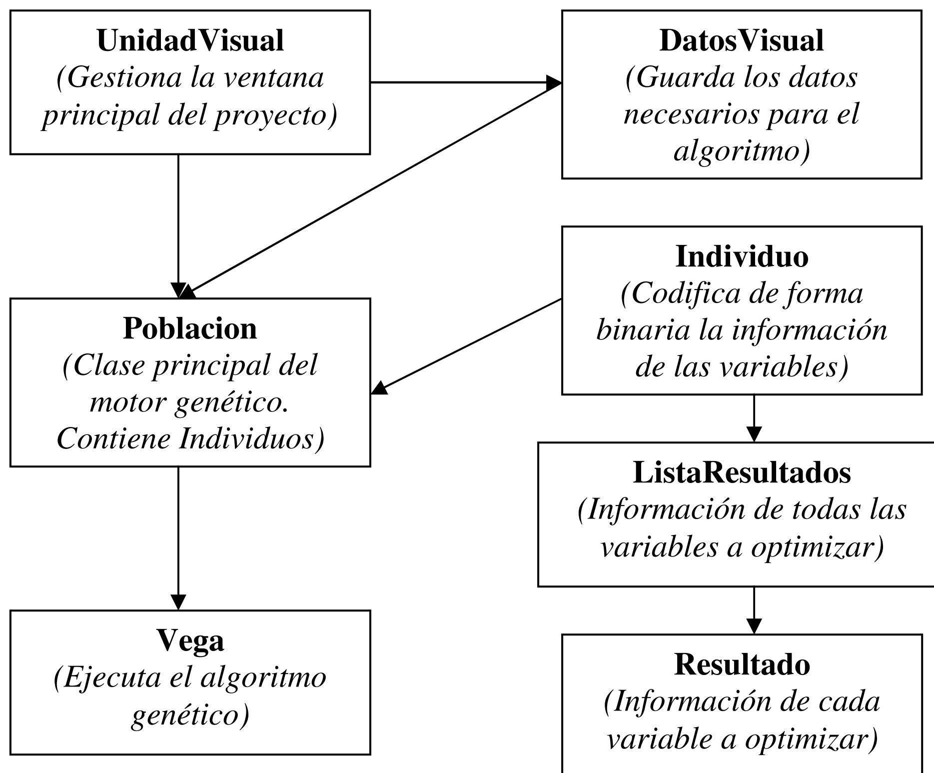
La clase Individuo, es la encargada de codificar la información, en forma binaria, de las variables dinámicas de la aplicación que queremos optimizar. Para ello, los individuos están divididos en fragmentos, uno para cada variable a optimizar.

Para realizar las optimizaciones para cada variable, necesitamos información acerca del su comportamiento en ejecución de dichas variables, que nos proporcionan en un fichero externo. Este fichero lo cargamos desde la ventana principal, y necesitamos de otra clase para extraer y almacenar esta información. Esta clase es Resultado, la cual, tras recorrer el perfil en ejecución cargado, obtiene y almacena los accesos realizados, el tamaño de dichas variables, etc.

Dado que en nuestra aplicación realizamos la exploración para todas las variables de una aplicación, necesitamos almacenar esta información para todos los individuos mediante la clase ListaResultados.

Por último, tenemos la clase Vega, que es la encargada de implementar los algoritmos genéticos, tanto simples, cuando queremos optimizar un único objetivo, como multiobjetivo, cuando queremos optimizar bajo dos o tres campos. Esta clase recibe una Poblacion, a la cual hace evolucionar dependiendo de los valores seleccionados.

Aquí se muestra un gráfico explicativo de la implementación de dicho motor genético:



# Anexo II: Listado de palabras para la búsqueda bibliográfica.

Memoria

Dinámica

Sistemas

Empotrados

Optimización

Genético

Multiobjetivo

Energía

Memoria

Rendimiento

# Anexo III: Autorización de difusión.

Nosotros, los autores del proyecto **"OPTIMIZACIÓN DE MEMORIA DINÁMICA EN SISTEMAS EMPOTRADOS MULTIMEDIA MEDIANTE COMPUTACIÓN EVOLUTIVA"**, autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Sergio Belmar Argudo: César M. González Iñiguez: Pablo Virseda Benito:

___________________ ___________________ ___________________

Madrid, a _____ de _____________ del 2006

# Bibliografía:

[1] Carlos A. Coello Coello, "Evolutionary Algorithms for solving Multiobjetive Problems.", 2002.

[2] Schaffer, J.D., "Multible Objetive Oprimization with vector evaluates genetic algorithms.", 1985.

[3] David Atienza, "Metodología multinivel de refinamiento del subsistema de memoria dinámica para los sistemas empotrados multimedia de altas prestaciones", 2005.

[4] IBM, "*The IBM Personal Computer Model 5150*", 1981.

[5] Frank Vahid, Tony Givargis, "*Embedded System Design: A Unified Hardware/Software Introduction*", 2002.

[6] David Seal, "*ARM Architecture Reference Manual, 2nd Edition*", 2004.

[7] Preeti Ranjan Panda, Nikil Dutt, Alexander Nicolau, "*Utilization of Scratch-Pad Memory in Embedded Processor Application*", 1997.

[8] Pret Ranjan Panda, F. Catthoor, Nikil Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, "*Data and Memory Optimizations for Embedded Systems*", 2001.

[9] Luca Benini, Giovanni De Micheli, "*System Level power Optimization techniques and tools*", 2000.

[10] E. G. Daylight, David Atienza, Arnout Vandecappelle, Francky Catthoor, Jose M. Mendias, "*Memory-Access-Aware Data Structure Transformations for Embedded Software with Dynamic Data Accesses*", 2004.

[11] J. Holland, "*Adaptation in Natural and Artificial Systems*", 1975.

[12] Goldberg, "*Genetic algorithms in search, optimization and machine Learning*", 1989.

[13] Osyczka, A., "*Multicriterion Optimization in Engineering with FORTRAN programs*", 1984.

[14] David Atienza, Marc Leeman, Jose M. Mendias, Francky Catthoor, Vincenzo De Florio, Geert Deconinck, "*Some Experiences on Dynamic Memory Management Refinement at System-Level for Multimedia Applications*", 2003.

[15] Marc Leeman, David Atienza, Jose M. Mendias, Francky Catthoor, "*Power Estimation Approach of Dynamic Data Storage on a Hardware Software Boundary Level*", 2003.

[16] E.G. Daylight, B. Demoen, F. Catthoor, "*Formally Specifying Dynamic Data Structures for Embedded Software Design: a More Thorough Initial Approach*", 2005.

[17] Reinhard Koch, Marc Pollefeys, Luc J. Van Gool, "*Realistic surface reconstruction of 3D scenes from uncalibrated image sequences*", 2000.

[18] John Cosmas, Itagaki Taki, Damian Green, Oleksiy Zalesny, Luc Van Gool, Marc Pollefeys, Roland Degeest, M. Waelkens, Karin Hraby, Martin Kampel, Robert Sablatnig, "*3D Murale*", 2002

[19] Dov Bulka, David Mayhew, "*Efficient C++*", Addison-Wesley, 2001