Sistemas Informáticos
Curso 2003-2004

Implementación de Juegos usando Algoritmos Evolutivos

Andrés Diéguez Alberte
Roberto Ovejero Málaga
Andrés Robledo Ibañez

Dirigido por:
Prof. Lourdes Araujo Serna
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

TABLA DE CONTENIDOS.

1. Introducción Programación Evolutiva.

-La Computación Evolutiva engloba un amplio conjunto de técnicas de resolución de problemas complejos, basadas en la emulación de procesos naturales de evolución. La principal aportación de la Computación Evolutiva a la metodología de resolución de problemas, consiste en el uso de mecanismos de selección de soluciones potenciales y de construcción de nuevos candidatos por recombinación de características de otros ya presentes, de modo parecido a como ocurre en la evolución de los organismos naturales. Entre esas técnicas destacan los Algoritmos Genéticos y la Programación Evolutiva.

-La Programación Evolutiva es una técnica de Computación Evolutiva útil para incorporar conocimiento específico a los Algoritmos Genéticos y de esta forma mejorar la eficiencia de éstos, ya que en los Algoritmos Genéticos se utilizan en gran medida datos pseudoaleatorios que pueden llevar a una eficiencia muy pobre (e incluso no encontrar soluciones al problema en cuestión). Pero estos datos pseudoaleatorios son los que dan genericidad a los Algoritmos Genéticos, de forma que se pueden utilizar en muchos problemas de optimización; a veces, según el problema en cuestión, puede que no se encuentre una solución óptima por los métodos tradicionales (quizás por no estar bien condicionados) mientras que los algoritmos genéticos son capaces de encontrarla. De esta forma, la Programación Evolutiva engloba la genericidad de los Algoritmos Genéticos para encontrar soluciones e incorpora conocimiento específico para generar eficiencia (encontrar más y mejores soluciones).

-Por otro parte, la programación de juegos está convirtiéndose hoy en día en un creciente campo donde se investigan y aplican tanto técnicas gráficas como complicados algoritmos que provienen de campos tales como la Inteligencia Artificial y la Música Interactiva. Con respecto a la algoritmia, en la programación de juegos se utilizan frecuentemente técnicas de optimización para resolver numerosos problemas que se plantean, así como muchas otras técnicas avanzadas y que no siempre se reconocen como tales: en realidad, todavía hay personas que piensan que la programación de juegos es, como lo era antaño, una programación “sucia” con muchos trucos para mejorar la velocidad, y sin uso apenas de algoritmos de organización. Hoy en día las cosas han cambiado (con la mejora y avance de las computadoras, más potentes, y de los lenguajes de programación y API´s, interfaces de programación de aplicaciones) y estamos ante uno de los campos más enormemente ampliados y comercializados: el mercado de los videojuegos.

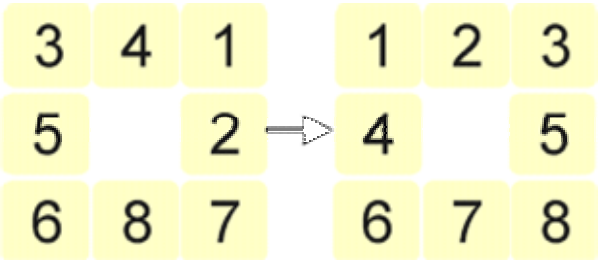
-En las siguientes páginas vamos a intentar acercar al lector un poco a la historia y las características de la programación evolutiva y los algoritmos genéticos.

1.1. Búsqueda, optimización y aprendizaje.

-Esta terna define los problemas a los que se pueden aplicar los algoritmos evolutivos, en cualquiera de sus formas. En realidad, los algoritmos de búsqueda abarcan prácticamente todo algoritmo para resolver problemas automáticamente. Habitualmente, en Informática se habla de búsqueda cuando hay que hallar información, siguiendo un determinado criterio, dentro de un conjunto de datos almacenados; sin embargo, aquí nos referiremos a otro tipo de algoritmos de búsqueda, a saber, aquellos que, dado el espacio de todas las posibles soluciones a un problema, y partiendo de una solución inicial, son capaces de encontrar la solución mejor o la única.

-El ejemplo clásico de este tipo de problemas se encuentra en los rompecabezas y juegos que se suelen abordar en **inteligencia artificial**. Un ejemplo es el *problema de las 8 reinas*, en el cual se deben de colocar 8 reinas en un tablero de ajedrez de forma que ninguna amenace a otra; o las *torres de Hanoi,* en el que, dada una serie de discos de radio decreciente apilados, hay que apilarlos en otro sitio teniendo en cuenta que no se puede colocar ningún disco encima de otro de radio inferior. Este tipo de problemas es fácil de abordar usando algoritmos "clásicos", como por ejemplo algoritmos recursivos o de tipo **voraz (*greedy*)**, sin embargo, hay otro tipo de problemas mucho más complicados, sobre todo los NP-completos (aquellos cuya complejidad crece con el tamaño del problema de forma exponencial) que no pueden ser abordados de esta forma. Algunos ejemplos de estos problemas serían los siguientes:

-8-puzzle, en el cual, como se muestra en la ilustración, a partir de una configuración inicial donde hay 8 cuadros desordenados, hay que llegar a otra configuración donde estén ordenados, usando para el intercambio cualquier posición que se halle vacía. El problema de búsqueda en este caso consiste en encontrar un camino que vaya desde la configuración inicial hasta la final.



-Problema del viajante, en el cual, dadas una serie de ciudades separadas por distancias diferentes, hay que calcular un camino tal que la distancia total recorrida sea mínima, y no que no repita ninguna ciudad. Este problema es NP-completo, y es paradigmático de este tipo de problemas.

-Mastermind: un jugador debe de averiguar una combinación de chinchetas de colores oculta por el otro jugador, y lo hace haciendo suposiciones sobre la combinación, y siendo contestado con una chincheta pequeña y blanca por cada acierto de color, y una negra por cada acierto de color o posición. Sólo una solución (entre 6^4) es la correcta, pero el jugador que ha puesto la combinación oculta va orientando la búsqueda mediante las chinchetas blancas o negras. El problema se hace exponencialmente más difícil cuando se aumenta el número de colores, y la longitud de la combinación.

-En muchos casos, la búsqueda está guiada por una función que indica lo buena que es esa solución, o el coste de la misma, o lo cerca que se está de la solución final, si es que se conoce; el problema se convierte entonces en un problema de optimización, es decir, encontrar la solución que maximiza la función *objetivo*, *de evaluación* o minimiza el coste. En términos formales, dada una función F de n variables $x_1,x_2,...,x_n$, optimizar la función consiste en encontrar la combinación de valores de x_i tales que $F(x_1, x_2,..., x_n)$ = Máximo. Se puede hablar de maximizar en vez de minimizar sin perder generalidad, ya que maximizar F equivale a minimizar $-F$.

-Generalmente, los problemas de optimización son tratados por la rama de las matemáticas denominada **Investigación Operacional**, aunque prácticamente todas las ramas de la ciencia y la ingeniería necesitan tratar con problemas de optimización en algún momento. Por ejemplo, en *teoría de juegos* se trata de maximizar la probabilidad de ganar, y en *reconocimiento de patrones* de minimizar el error de clasificación de un patrón desconocido (como una imagen de satélite digitalizada, o un canal procesado de una señal de un electroencefalograma).

-En algunos casos, la función de evaluación ni siquiera existe, o no es estática, sino que viene dada por el entorno de la solución. Por ejemplo, en un programa para jugar al **Othello**, **reversi o el 4 en raya**, la función de evaluación vendrá dada por su puntuación a la hora de jugar con los demás jugadores. Un autor, Pollack hizo enfrentarse a unas estrategias de juego con otras, de forma que según van evolucionando, la función de adaptación va variando.

-Hay muchas formar de abordar problemas de optimización. Algunas de ellas se verán en las siguientes secciones.

1.1.1. Método analítico

-Si existe la función F, es de una sola variable, y se puede derivar dos veces en todo su rango, se pueden hallar todos sus máximos, sean locales o globales. Sin embargo, la mayoría de las veces no se conoce la forma de la función F, y si se conoce, no tiene porqué ser diferenciable ni siquiera una vez. Incluso el tratamiento analítico para funciones de más de 1 variable es complicado.

1.1.2. Métodos exhaustivos, aleatorios y heurísticos

-Los métodos exhaustivos recorren todo el espacio de búsqueda, quedándose con la mejor solución, y los heurísticos utilizan reglas para eliminar zonas del espacio de búsqueda consideradas "poco interesantes". Algunos algoritmos de búsqueda, como el MiniMax, son de este tipo; se suelen utilizar en juegos para examinar y podar el árbol de posibilidades a partir de la jugada actual; *Deep Blue*, por ejemplo, juega de esta forma.

-En los métodos aleatorios, se va muestreando el espacio de búsqueda acotando las zonas que no han sido exploradas; se escoge la mejor solución, y, además, se da el intervalo de confianza de la solución encontrada.

1.1.3. Hillclimbing.

-En estos métodos, también denominados de *hillclimbing*, se va evaluando la función en uno o varios puntos, pasando de un punto a otro en el cual el valor de la evaluación es superior. La búsqueda termina cuando se ha encontrado el punto con un valor máximo. En general, un algoritmo escalador funciona de la forma siguiente

Algoritmo escalador

1. Escoger una solución inicial $(x_i,...,x_n)$
2. Mientras que siga subiendo el valor de F, hacer
3. Alterar la solución $(x'_i,...,x'_n) = (x_i,...,x_n) + (y_i,...,y_n)$, y evaluar F.
4. Si $F(x'_i,...,x'_n) > F(x_i,...,x_n)$, hacer $(x_i,...,x_n) = (x'_i,...,x'_n)$.
5. Volver a 2.

-Estos algoritmos toman muchas formas diferentes, según el número de dimensiones del problema solución, el valor del incremento y en la dirección en la cual se tiene que dar. En algunos casos se utiliza el llamado **Método Montecarlo** (por el casino), en el cual se escoge la nueva solución de forma aleatoria.

-El principal problema de este tipo de algoritmos es que se quedan en el pico más cercano a la solución inicial; además, no son válidos para problemas *multimodales*, en los cuales la función de coste tiene varios óptimos posibles.

1.1.4. Recocimiento simulado

-Conocido como *Simulated Annealing*, en inglés, el nombre viene de la forma como se consiguen ciertas aleaciones en forja; una vez fundido el metal, se va enfriando poco a poco, para conseguir finalmente la estructura cristalina correcta, que haga que la aleación sea dura y resistente.

-Este algoritmo se podría calificar como escalador estocástico, y su principal objetivo es evitar los mínimos locales en los que suelen caer los escaladores. Para ello, no siempre acepta la solución óptima, sino que a veces puede escoger una solución menos óptima, siempre que la diferencia entre ambos tenga un nivel determinado, que depende de un parámetro denominado *temperatura* (seguimos con la metáfora). El algoritmo de recocimiento simulado es el siguiente:

Algoritmo de recocimiento simulado

1. Inicializar la temperatura T, y la solución inicial $(x_i,...,x_n)$ y evaluar $F(x_i,...,x_n)$.
2. Repetir los pasos siguientes, hasta que la temperatura sea nula o el valor de F converja:
3. Disminuir la temperatura.
4. Seleccionar una nueva solución $(x_{i'},...,x'_n)$ en la vecindad de la anterior (*mutar la solución*), y evaluarla.
5. Si $F(x'_i,...,x'_n) > F(x_i,...,x_n)$, hacer $(x_i,...,x_n) = (x'_i,...,x'_n)$, si no, generar un número aleatorio R entre 0 y 1. Si , entonces $(x_i,...,x_n) = (x'_i,...,x'_n)$.

1.1.5. Técnicas basadas en población

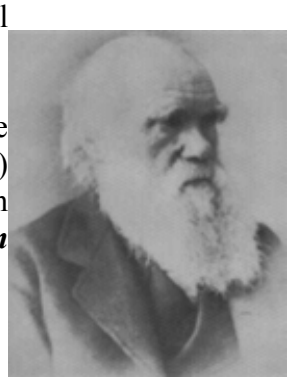
-Este tipo de técnicas pueden ser versiones de cualquiera de las anteriores, pero en vez de tener una sola solución, que se va alterando hasta obtener el óptimo, se persigue el óptimo cambiando y combinando varias soluciones; de esta forma es más fácil escapar de los mínimos locales tan temidos. Entre estas técnicas se hallan la mayoría de los algoritmos evolutivos.

1.1.6. Técnicas experimentales

-En algunos casos, solo el ojo humano es capaz de evaluar lo apropiada que es una solución a un tema determinado, por ejemplo, en problemas de diseño o de calidad. En este caso, se pueden utilizar cualquiera de las técnicas expuestas anteriormente, pero a la hora de evaluar una solución, un experto o experta tendrá que darle una puntuación. Por ejemplo, es lo que se usa cuando uno se prueba ropa para dar con la combinación correcta de colores y estilos.

1.2. La evolución

-Después de ver tipo de problemas a los que se pueden aplicar los algoritmos evolutivos, se va a estudiar qué es lo que inspira dichos algoritmos, la Naturaleza, ese fenómeno natural denominado evolución, y si optimiza o no.

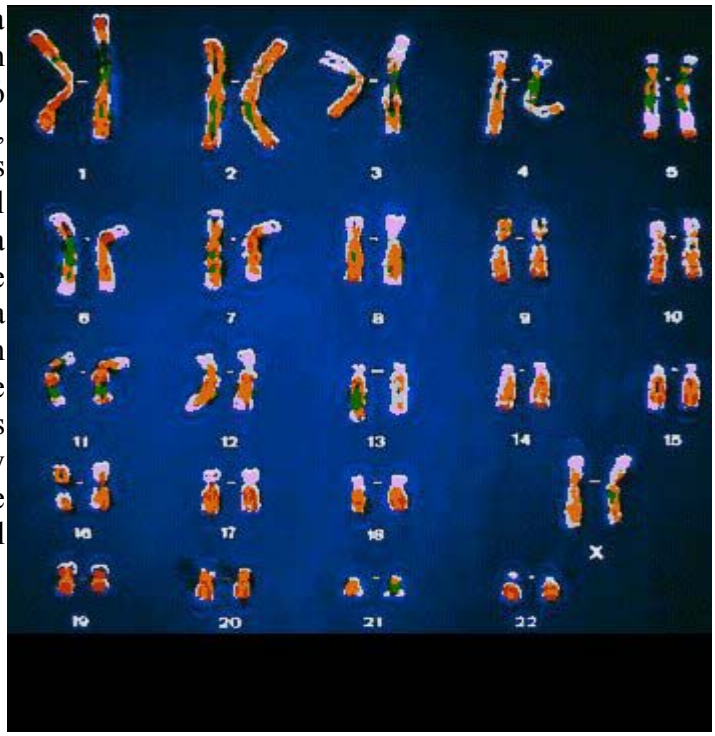


-La teoría de la evolución (que no es tal teoría, sino una serie de hechos probados), fue descrita por Charles Darwin (ver imagen) 20 años después de su viaje por las islas Galápagos en el *Beagle*, en el libro ***Sobre el Origen de las Especies por medio de la Selección***

Natural. Este libro fue bastante polémico en su tiempo, y en cualquier caso es una descripción incompleta de la evolución. La hipótesis de Darwin, presentada junto con Wallace, que llegó a las mismas conclusiones independientemente, es que pequeños cambios heredables en los seres vivos y la selección son los dos hechos que provocan el cambio en la Naturaleza y la generación de nuevas especies. Pero Darwin desconocía cuál es la base de la herencia, pensaba que los rasgos de un ser vivo eran como un fluido, y que los "fluidos" de los dos padres se mezclaban en la descendencia; esta hipótesis tenía el problema de que al cabo de cierto tiempo, una población tendría los mismos rasgos intermedios.

-Fue **Lendel** quien descubrió que los caracteres se heredaban de forma discreta, y que se tomaban del padre o de la madre, dependiendo de su carácter dominante o recesivo. A estos caracteres que podían tomar diferentes valores se les llamaron ***genes***, y a los valores que podían tomar, ***alelos***. En realidad, las teorías de Lendel, que trabajó en total aislamiento, se olvidaron y no se volvieron a redescubrir hasta principios del siglo XX. Además, hasta 1930 el geneticista inglés **Robert Aylmer** no relacionó ambas teorías, demostrando que los genes mendelianos eran los que proporcionaban el mecanismo necesario para la evolución.

-Más o menos por la misma época, el biólogo alemán **Walther Flemming** describió los cromosomas (ver ilustración), como ciertos filamentos en los que se agregaba la cromatina del núcleo celular durante la división; poco más adelante se descubrió que las células de cada especie viviente tenían un número fijo y característico de cromosomas.Y no fue hasta los años 50, cuando **Watson** y **Crick** descubrieron que la base molecular de los genes está en el ADN.



-Los cromosomas están compuestos de ADN, y por tanto los genes están en los cromosomas.

-La macromolécula de ADN está compuesta por bases *púricas y pirimidínicas,* la adenina, citosina, guanina y timina. La combinación y la secuencia de estas bases forma el **código genético**, único para cada ser vivo. Grupos de 3 bases forman un *codon*, y cada codon codifica un aminoácido (el que exprese ese aminoácido o no depende de otros factores); el código genético codifica todas las proteínas que forman parte de un ser vivo. Mientras que al código genético se le llama ***genotipo***, al cuerpo que

construyen esas proteínas, modificado por la presión ambiental, la historia vital, y otros mecanismos dentro del cromosoma, se llama ***fenotipo***.

-Todos estos hechos forman hoy en día la teoría del neo-darwinismo, que afirma que la historia de la mayoría de la vida está causada por una serie de procesos que actúan en y dentro de las poblaciones: reproducción, mutación, competición y selección. La evolución se puede definir entonces como cambios en el conjunto genético de una población.

-Un tema polémico, con opiniones variadas dependiendo de si se trata de informáticos evolutivos o de biólogos o geneticistas, es si la evolución optimiza o no. Según los informáticos evolutivos, la evolución optimiza, puesto que va creando seres cada vez más perfectos, cuyo culmen es el hombre; además, indicios de esta optimización se encuentran en el organismo de los animales, desde el tamaño y tasa de ramificación de las arterias, diseñada para maximizar flujo, hasta el metabolismo, que optimiza la cantidad de energía extraída de los alimentos. Sin embargo, los geneticistas y biólogos evolutivos afirman que la evolución no optimiza, sino que adapta y optimiza localmente en el espacio y el tiempo; evolución no significa progreso. Un organismo más evolucionado puede estar en desventaja competitiva con uno de sus antepasados, si se colocan en el ambiente del último.

1.2.1. Mecanismos de cambio en la evolución

-Estos mecanismos de cambio serán necesarios para entender los algoritmos evolutivos, pues se trata de imitarlos para resolver problemas de ingeniería; por eso merece la pena conocerlos en más profundidad. Los mecanismos de cambio alteran la proporción de alelos de un tipo determinado en una población, y se dividen en dos tipos: los que disminuyen la variabilidad, y los que la aumentan.

-Los principales mecanismos que disminuyen la variabilidad son los siguientes:

- **Selección natural**: los individuos que tengan algún rasgo que los haga menos válidos para realizar su tarea de seres vivos, no llegarán a reproducirse, y, por tanto, su patrimonio genético desaparecerá; algunos no llegarán ni siquiera a nacer. Esta selección sucede a muchos niveles: competición entre miembros de la especie (intraespecífica), competición entre diferentes especies, y competición predador-presa, por ejemplo. También es importante la selección sexual, en la cual las hembras eligen el mejor individuo de su especie disponible para reproducirse.

- **Deriva génica**: el simple hecho de que un alelo sea más común en la población que otro, causará que la proporción de alelos de esa población vaya aumentando en una población aislada, lo cual a veces da lugar a fenómenos de especiación.

-Otros mecanismos aumentan la diversidad, y suceden generalmente en el ámbito molecular. Los más importantes son:

- **Mutación**: la mutación es una alteración del código genético, que puede suceder por múltiples razones. En muchos otros casos, las mutaciones, que cambian un nucleótido por otro, son letales, y los individuos ni siquiera llegan a desarrollarse, pero a veces se da lugar a la producción de una proteína que aumenta la supervivencia del individuo, y que, por tanto, es pasada a la descendencia. Las mutaciones son totalmente aleatorias, y son el mecanismo básico de generación de variedad genética. A pesar de lo que se piensa habitualmente, la mayoría de las mutaciones ocurren de forma natural, aunque existen *sustancias mutagénicas* que aumentan su frecuencia.

- **Poliploidía**: mientras que las células normales poseen dos copias de cada cromosoma, y las células reproductivas una (haploides), puede suceder por accidente que alguna célula reproductiva tenga dos copias; si se logra combinar con otra célula diploide o haploide dará lugar a un ser vivo con varias copias de cada cromosoma. La mayoría de las veces, la poliploidía da lugar a individuos con algún defecto (por ejemplo, el tener 3 copias del cromosoma 21 da lugar al mongolismo), pero en algunos casos se crean individuos viables.

- **Recombinación**: cuando las dos células sexuales, o gametos, una masculina y otra femenina se combinan, los cromosomas de cada una también lo hacen, intercambiándose genes, que a partir de ese momento pertenecerán a un cromosoma diferente. A veces también se produce traslocación dentro de un cromosoma; una secuencia de código se elimina de un sitio y aparece en otro sitio del cromosoma, o en otro cromosoma.

- **Flujo genético**: o intercambio de material genético entre seres vivos de diferentes especies. Normalmente se produce a través de un vector, que suelen ser virus o bacterias; estas incorporan a su material genético genes procedentes de una especie a la que han infectado, y cuando infectan a un individuo de otra especie pueden transmitirle esos genes a los tejidos generativos de gametos.

-En resumen, la selección natural actúa sobre el fenotipo y suele disminuir la diversidad, haciendo que sobrevivan solo los individuos más aptos (aunque esta frase, bien mirada, es una redundancia: ¿Sobreviven los mejores o son los mejores porque sobreviven?); los mecanismos que generan diversidad y que combinan características actúan habitualmente sobre el genotipo.

1.3. Evolución de la informática evolutiva

-Ya que se han descrito cuales son los mecanismos de la evolución, y una amplia gama de problemas que pueden o no tener relación entre sí, llega la hora de contar, desde sus principios, como evolucionó la idea de simular o imitar la evolución con el objeto de resolver problemas humanos.

-Las primeras ideas, incluso antes del descubrimiento del ADN, vinieron de **Von Neumann,** uno de los mayores científicos de este siglo. Von Neumann afirmó que la vida debía de estar apoyada por un código que a la vez describiera como se puede construir un ser vivo, y tal que ese ser creado fuera capaz de autorreproducirse; por tanto, un autómata o máquina autorreproductiva tendría que ser capaz, aparte de contener las instrucciones para hacerlo, de ser capaz de copiar tales instrucciones a su descendencia.

-Sin embargo, no fue hasta mediados de los años cincuenta, cuando el rompecabezas de la evolución se había prácticamente completado, cuando **Box** comenzó a pensar en imitarla para, en su caso, mejorar procesos industriales. La técnica de Box, denominada EVOP (Evolutionary Operation), consistía en elegir una serie de variables que regían un proceso industrial. Sobre esas variables se creaban pequeñas variaciones que formaban un hipercubo, variando el valor de las variables una cantidad fija. Se probaba entonces con cada una de las esquinas del hipercubo durante un tiempo, y al final del periodo de pruebas, un comité humano decidía sobre la calidad del resultado. Es decir, se estaba aplicando mutación y selección a los valores de las variables, con el objeto de mejorar la calidad del proceso. Este procedimiento se aplicó con éxito a algunas industrias químicas.

-Un poco más adelante, en 1958, **Friedberg** y sus colaboradores pensaron en mejorar usando técnicas evolutivas la operación de un programa. Para ello diseñaron un código máquina de 14 bits (2 para el código de operación, y 6 para los datos y/o instrucciones); cada programa, tenía 64 instrucciones. Un programa llamado *Herman*, ejecutaba los programas creados, y otro programa, el *Teacher* o profesor, le mandaba a Herman ejecutar otros programas y ver si los programas ejecutados habían realizado su tarea o no. La tarea consistía en leer unas entradas, situadas en una posición de memoria, y debían depositar el resultado en otra posición de memoria, que era examinada al terminarse de ejecutar la última instrucción.

-Para hacer evolucionar los programas, Friedberg hizo que en cada posición de memoria hubiera dos alternativas; para cambiar un programa, alternaba las dos instrucciones (que eran una especie de alelos), o bien reemplazaba una de las dos instrucciones con una totalmente aleatoria.

-En realidad, lo que estaba haciendo es usar mutación para generar nuevos programas; al parecer, no tuvo más éxito que si hubiera buscado aleatoriamente un programa que hiciera la misma tarea. El problema es que la mutación sola, sin ayuda de la selección, hace que la búsqueda sea prácticamente una búsqueda aleatoria.

-Más o menos simultáneamente, **Bremmerman** trató de usar la evolución para "entender los procesos de pensamiento creativo y aprendizaje", y empezó a considerar la evolución como un proceso de aprendizaje. Para resolver un problema, codificaba las variables del problema en una cadena binaria de 0s y 1s, y sometía la cadena a mutación, cambiando un bit de cada vez; de esta forma, estableció que la tasa ideal de mutación debía de ser tal que se cambiara un bit cada vez. Bremmerman trató de resolver problemas de minimización de funciones, aunque no está muy claro qué tipo de selección usó, si es que usó alguna, y el tamaño y tipo de la población. En todo caso, se llegaba a un punto, la "trampa de Bremmerman", en el cual la solución no mejoraba; en

intentos sucesivos trató de añadir entrecruzamiento entre soluciones, pero tampoco obtuvo buenos resultados. Una vez más, el simple uso de operadores que creen diversidad no es suficiente para dirigir la búsqueda genética hacia la solución correcta; y corresponde a un concepto de la evolución darwiniano clásico, o incluso de libro de comic: por mutación, se puede mejorar a un individuo; en realidad, la evolución actúa a nivel de población.

-El primer uso de procedimientos evolutivos en inteligencia artificial se debe a **Reed, Toombs y Baricelli**, que trataron de hacer evolucionar un tahúr que jugaba a un juego de cartas simplificado. Las estrategias de juego consistían en una serie de 4 probabilidades de apuesta alta o baja con una mano alta o baja, con cuatro parámetros de mutación asociados. Se mantenía una población de 50 individuos, y aparte de la mutación, había intercambio de probabilidades entre dos padres. Es de suponer que los perdedores se eliminaban de la población (tirándolos por la borda). Aparte de, probablemente, crear buenas estrategias, llegaron a la conclusión de que el entrecruzamiento no aportaba mucho a la búsqueda. Los intentos posteriores, ya realizados en los años 60, ya corresponden a los algoritmos evolutivos modernos, y se han seguido investigando hasta nuestros días. Algunos de ellos son simultáneos a los algoritmos genéticos, pero se desarrollaron sin conocimiento unos de otros. Uno de ellos, la programación evolutiva de **Fogel**, se inició como un intento de usar la evolución para crear máquinas inteligentes, que pudieran prever su entorno y reaccionar adecuadamente a él. Para simular una máquina pensante, se utilizó un *autómata celular*. Un autómata celular es un conjunto de estados y reglas de transición entre ellos, de forma que, al recibir una entrada, cambia o no de estado y produce una salida. En la figura 7, se muestra un autómata con estados etiquetados por letras mayúsculas y representados por círculos, las reglas de transición, con líneas que los unen, los símbolos de entrada son 0 y 1, y los símbolos de salida con letras mayúsculas.

-**Fogel** trataba de hacer aprender a estos autómatas a encontrar regularidades en los símbolos que se le iban enviando. Como método de aprendizaje, usó un algoritmo evolutivo: una población de diferentes autómatas competía para hallar la mejor solución, es decir, predecir cual iba a ser el siguiente símbolo de la secuencia con un mínimo de errores; los peores 50% eran eliminados cada generación, y sustituidos por otros autómatas resultantes de una mutación de los existentes.

-De esta forma, se lograron hacer evolucionar autómatas que predecían algunos números primos (por ejemplo, uno, cuando se le daban los números más altos, respondía siempre que no era primo; la mayoría de los números mayores de 100 son no primos). En cualquier caso, estos primeros experimentos demostraron el potencial de la evolución como método de búsqueda de soluciones novedosas.

-Más o menos a mediados de los años 60, **Rechenberg** y **Schwefel** describieron las estrategias de evolución. Las estrategias de evolución son métodos de optimización paramétricos, que trabajan sobre poblaciones de cromosomas compuestos por números reales. Hay diversos tipos de estrategias de evolución, que se verán más adelante, pero en la más común, se crean nuevos individuos de la población añadiendo un vector mutación a los cromosomas existentes en la población; en cada generación, se elimina un porcentaje de la población (especificado por los parámetros y μ), y los restantes

generan la población total, mediante mutación y cruce. La magnitud del vector mutación se calcula adaptativamente.

Y, por supuesto, surgieron también los algoritmos genéticos. Pero esa es otra historia.

1.4. Algoritmo genético simple.

1.4.1. Introducción

-John Holland desde pequeño, se preguntaba cómo logra la naturaleza, crear seres cada vez más perfectos (aunque, como se ha visto, esto no es totalmente cierto, o en todo caso depende de qué entienda uno por *perfecto*). Lo curioso era que todo se lleva a cabo a base de interacciones locales entre individuos, y entre estos y lo que les rodea. No sabía la respuesta, pero tenía una cierta idea de como hallarla: tratando de hacer pequeños modelos de la naturaleza, que tuvieran alguna de sus características, y ver cómo funcionaban, para luego extrapolar sus conclusiones a la totalidad. De hecho, ya de pequeño hacía simulaciones de batallas célebres con todos sus elementos: copiaba mapas y los cubría luego de pequeños ejércitos que se enfrentaban entre sí.

-En los años 50 entró en contacto con los primeros ordenadores, donde pudo llevar a cabo alguna de sus ideas, aunque no se encontró con un ambiente intelectual fértil para propagarlas. Fue a principios de los 60, en la Universidad de Michigan en Ann Arbor, donde, dentro del grupo *Logic of Computers,* sus ideas comenzaron a desarrollarse y a dar frutos. Y fue, además, leyendo un libro escrito por un biólogo evolucionista, R. A. Fisher, titulado *La teoría genética de la selección natural,* como comenzó a descubrir los medios de llevar a cabo sus propósitos de comprensión de la naturaleza. De ese libro aprendió que la evolución era una forma de adaptación más potente que el simple aprendizaje, y tomó la decisión de aplicar estas ideas para desarrollar programas bien adaptados para un fin determinado.

-En esa universidad, Holland impartía un curso titulado *Teoría de sistemas adaptativos*. Dentro de este curso, y con una participación activa por parte de sus estudiantes, fue donde se crearon las ideas que más tarde se convertirían en los algoritmos genéticos.

-Por tanto, cuando Holland se enfrentó a los algoritmos genéticos, los objetivos de su investigación fueron dos:

- imitar los procesos adaptativos de los sistemas naturales, y
- diseñar sistemas artificiales (normalmente programas) que retengan los mecanismos importantes de los sistemas naturales.

-Unos 15 años más adelante, **David Goldberg,** actual delfín de los algoritmos genéticos, conoció a Holland, y se convirtió en su estudiante. Golberg era un ingeniero industrial trabajando en diseño de *pipelines*, y fue uno de los primeros que trató de aplicar los algoritmos genéticos a problemas industriales. Aunque Holland trató de disuadirle, porque pensaba que el problema era excesivamente complicado como para aplicarle algoritmos genéticos, Goldberg consiguió lo que quería, escribiendo un algoritmo genético en un ordenador personal Apple II. Estas y otras aplicaciones

creadas por estudiantes de Holland convirtieron a los algoritmos genéticos en un campo con base suficiente aceptado para celebrar la primera conferencia en 1985, ICGA´85. Tal conferencia se sigue celebrando bianualmente.

1.4.2. Anatomía de un algoritmo genético simple

-Los ***algoritmos genéticos*** son métodos sistemáticos para la resolución de problemas de búsqueda y optimización que aplican a estos los mismos métodos de la evolución biológica: selección basada en la población, reproducción sexual y mutación.

-Los algoritmos genéticos son métodos de optimización, que tratan de resolver el mismo conjunto de problemas que se ha contemplado anteriormente, es decir, hallar $(x_i,...,x_n)$ tales que $F(x_i,...,x_n)$ sea máximo. En un algoritmo genético, tras parametrizar el problema en una serie de variables, $(x_i,...,x_n)$ se codifican en un cromosoma. Todas los operadores utilizados por un algoritmo genético se aplicarán sobre estos cromosomas, o sobre poblaciones de ellos. En el algoritmo genético va implícito el método para resolver el problema; son solo parámetros de tal método los que están codificados, a diferencia de otros algoritmos evolutivos como la programación genética. Hay que tener en cuenta que un algoritmo genético es independiente del problema, lo cual lo hace un algoritmo *robusto*, por ser útil para cualquier problema, pero a la vez *débil*, pues no está especializado en ninguno.

-Las soluciones codificadas en un cromosoma *compiten* para ver cuál constituye la mejor solución (aunque no necesariamente la mejor de todas las soluciones posibles). El *ambiente,* constituido por las otras camaradas soluciones, ejercerá una presión selectiva sobre la población, de forma que sólo los mejor adaptados (aquellos que resuelvan mejor el problema) sobrevivan o leguen su material genético a las siguientes generaciones, igual que en la evolución de las especies. La diversidad genética se introduce mediante mutaciones y reproducción sexual.

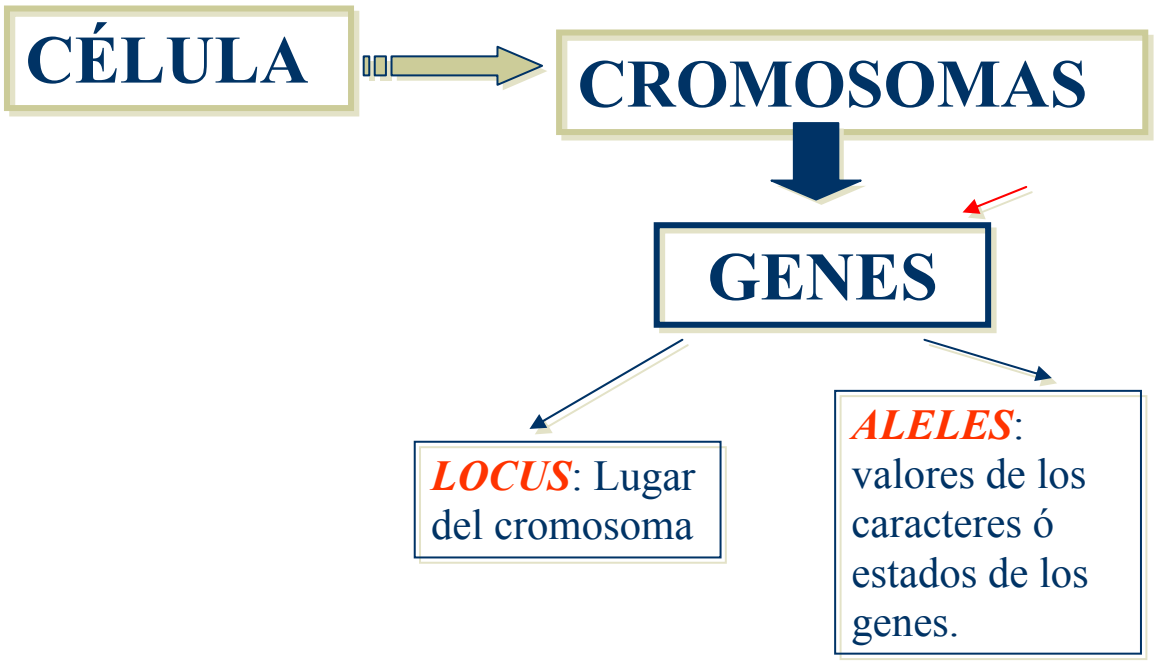
-En la Naturaleza lo único que hay que optimizar es la supervivencia, y eso significa a su vez maximizar diversos factores y minimizar otros. Un algoritmo genético, sin embargo, se usará para optimizar, habitualmente, sólo una función, no diversas funciones relacionadas entre sí simultáneamente. Este tipo de optimización, denominada optimización multimodal, también se suele abordar con un algoritmo genético especializado.

-Por lo tanto, un algoritmo genético consiste en lo siguiente: hallar de qué parámetros depende el problema, codificarlos en un cromosoma, y se aplican los métodos de la evolución: selección y reproducción sexual con intercambio de información y alteraciones que generan diversidad. En las siguientes secciones se verán cada uno de los aspectos de un algoritmo genético.

1.4.3. Vocabulario de los Algoritmos Genéticos

-Es importante, antes de aumentar la complejidad de la exposición, familiarizarse con algunos términos que se van a utilizar con asiduidad en este ámbito. Términos que nos van a resultar familiares porque ya hemos leído en los apartados anteriores en los que hemos hablado de las teorías de la evolución de los seres vivos y

que ahora vamos a trasladar a los algoritmos genéticos y que podremos ver la relación entre unos y otros.



-Una célula contiene numerosísimos cromosomas que la especializan. Un cromosoma es una cadena de genes.

Algoritmo Genético	Significado
Cromosomas(cadena, individuo)	Solución (*código*)
Genes (bits)	Parte de la solución.
Locus	Posición del Gen
Aleles	Valor del Gen
Fenotipo	Solución decodificada *(Apariencia Externa)*
Genotipo	Solución Codificada. *(Estructura Interna)*

1.4.4. Ventajas de los Algoritmos Genéticos

-Veamos que ventajas introducen la aplicación de esta técnica para la resolución de los problemas:

- No tienen muchos requerimientos matemáticos del problema de optimización.
- Debido a su naturaleza evolutiva => buscan soluciones sin considerar específico conocimiento del problema.
- Proporcionan una gran flexibilidad para hibridizarse con heurísticas dependiente del dominio => ***eficientes implementaciones.***
- Pueden manejar toda clase de función objetivo y restricciones definidas sobre un espacio de búsqueda discreto, continuo o mezclado.
- La estructura de los operadores los hace muy efectivos al realizar búsquedas global.

1.4.5. Diferencias con los métodos tradicionales de búsqueda.

Tradicionales	***A. G.***
3.- Trabaja con los propios parámetros.	3.-Emplea codificación de los parámetros.
4.- Usa información de las derivadas u otro conocimiento adicional.	4.- Usa información de la función objetivo.
5.- Emplean reglas de transición deterministas.	5.- Emplean reglas de transición probabilísticas.

1.4.6. Estructura de un Algoritmo Genético Simple

-Los algoritmos genéticos requieren que el conjunto se codifique en un ***cromosoma***. Cada cromosoma tiene varios ***genes***, que corresponden a sendos parámetros del problema. Para poder trabajar con estos genes en el ordenador, es necesario codificarlos en una *cadena*, es decir, una ristra de símbolos (números o letras) que generalmente va a estar compuesta de 0s y 1s. La representación de los genes de un individuo depende del problema. Cada problema requiere una representación distinta para conseguir los frutos deseados. Los cromosomas de un individuo habitualmente vienen representados por cadenas de 0s y 1s, pero también pueden ser árboles, vectores de colores, vectores de movimientos…en fin, en cada problema hay que estudiar como queremos que se represente la solución para determinar como queremos que se represente los cromosomas de los individuos.

-El número de bits usado para cada parámetro dependerá de la precisión que se quiera en el mismo o del número de opciones posibles (alelos) que tenga ese parámetro. Por ejemplo, si se codifica una combinación del Mastermind, cada gen tendrá tantas opciones como colores halla, el número de bits elegido será el $\log_2$ (número de colores). Para este mismo problema también se puede optar por representar cada individuo como una cadena de colores, donde la cadena representa una solución posible al problema.

-La mayoría de las veces, una codificación correcta es la clave de una buena resolución del problema. En todo caso, se puede ser bastante creativo con la codificación del problema. Esto puede llevar a usar cromosomas bidimensionales, o tridimensionales, o con relaciones entre genes que no sean puramente lineales de vecindad.

-Normalmente, la codificación es estática, pero en casos de optimización numérica, el número de bits dedicados a codificar un parámetro puede variar, o incluso lo que representen los bits dedicados a codificar cada parámetro. Algunos paquetes de algoritmos genéticos adaptan automáticamente la codificación según van convergiendo los bits menos significativos de una solución.

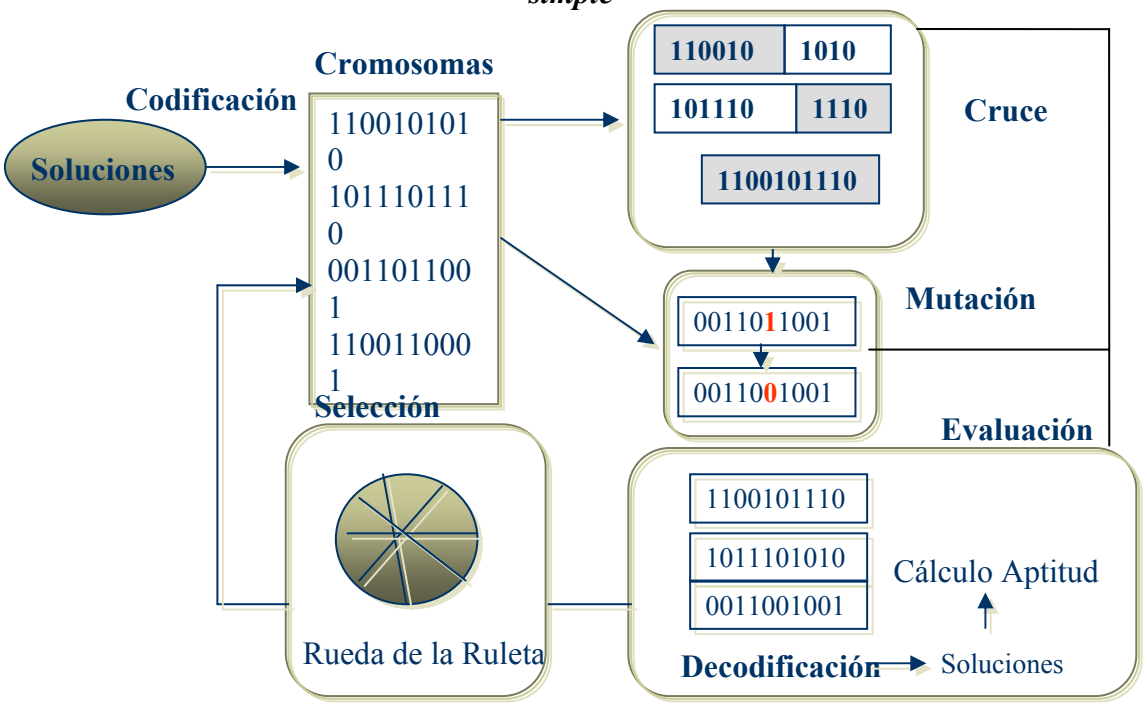
1.4.7. Algoritmo genético propiamente dicho

-Para comenzar la competición, se generan aleatoriamente una serie de cromosomas. El algoritmo genético procede de la forma siguiente:

Algoritmo genético

1. Evaluar la puntuación (*adaptación*) de cada uno de los genes.
2. Permitir a cada uno de los individuos reproducirse, de acuerdo con su puntuación.
3. Emparejar los individuos de la nueva población, haciendo que intercambien material genético, y que alguno de los bits de un gen se vea alterado debido a una *mutación* espontánea.

Esquema de un algoritmo genético simple



-*Más adelante veremos los operadores genéticos*, y hay tres principales: ***selección, cruce y mutación***.

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

- **Tamaño de la población**: debe de ser suficiente para garantizar la diversidad de las soluciones, y, además, tiene que crecer más o menos con el número de bits del cromosoma, aunque nadie ha aclarado cómo tiene que hacerlo. Por supuesto, depende también del ordenador en el que se esté ejecutando.
- **Condición de terminación:** lo más habitual es que la condición de terminación sea la convergencia del algoritmo genético o un número prefijado de generaciones.
- **Probabilidad de cruce:** puede que sea fijo o variable, intentando emular a la evolución natural, existe una probabilidad de que los individuos se crucen entre sí
- **Probabilidad de mutación:** parámetro, también puede ser fijo o variable, que determina la posibilidad de mutación que tienen los individuos de una población.

1.4.8. Evaluación y selección

-Durante la evaluación, se decodifica el gen, convirtiéndose en una serie de parámetros de un problema, se halla la solución del problema a partir de esos parámetros, y se le da una puntuación a esa solución en función de lo cerca que esté de

-Una vez evaluada la adaptación, se tiene que crear la nueva población teniendo en cuenta que los *buenos* rasgos de los mejores se transmitan a esta. Para ello, hay que seleccionar a una serie de individuos encargados de tan ardua tarea. Y esta selección, y la consiguiente reproducción, se puede hacer de dos formas principales:

- ***Basado en el rango***: en este esquema se mantiene un porcentaje de la población, generalmente la mayoría, para la siguiente generación. Se coloca toda la población por orden de adapatción, y los *M* menos dignos son eliminados y sustituidos por la descendencia de alguno de los *M* mejores con algún otro individuo de la población. Una variante de este es el muestreado estocástico universal, que trata de evitar que los individuos con más adaptación copen la población; en vez de dar la vuelta a una ruleta con una ranura, da la vuelta a la ruleta con N ranuras, tantas como la población; de esta forma, la distribución estadística de descendientes en la nueva población es más parecida a la real.

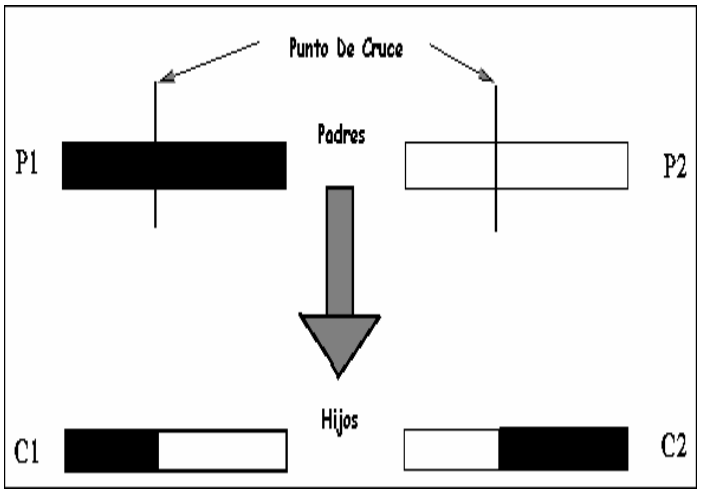
- ***Rueda de ruleta***: se crea un *pool* genético formado por cromosomas de la generación actual, en una cantidad proporcional a su adaptación. Si la proporción hace que un individuo domine la población, se le aplica alguna operación de escalado. Dentro de este *pool*, se cogen parejas aleatorias de cromosomas y se emparejan, sin importar incluso que sean del mismo progenitor (para eso están otros operadores, como la mutación). Hay otras variantes: por ejemplo, en la nueva generación se puede incluir el mejor representante de la generación actual. En este caso, se denomina método *elitista*.

- ***Selección de torneo***: se escogen aleatoriamente un número *T* de individuos de la población, y el que tiene puntuación mayor se reproduce, sustituyendo su descendencia al que tiene menor puntuación.

- ***Elitismo***: los mejores individuos de una generación de la población se reservan para la siguiente generación sin que entren en la selección y puedan caer por el azar fuera de los elegidos. De esta manera garantizamos que para la próxima generación se han quedado los mejores individuos. El número de individuos que se eligen debe ser establecido previamente.

1.4.9 Cruce

-Consiste en el intercambio de material genético entre dos cromosomas de dos individuos distintos. El *cruce* es el principal operador genético, hasta el punto que se puede decir que no es un algoritmo genético si no tiene *cruce*, y, sin embargo, puede serlo perfectamentesin mutación, según descubrió Holland.



-Para aplicar el cruce, entrecruzamiento o recombinación, se escogen aleatoriamente dos miembros de la población. No pasa nada si se emparejan dos descendientes de los mismos padres; ello garantiza la perpetuación de un individuo con buena puntuación. Sin embargo, si esto sucede demasiado a menudo, puede crear problemas: toda la población puede aparecer dominada por los descendientes de algún gen, que, además, puede tener caracteres no deseados. Esto se suele denominar en otros métodos de optimización ***atranque en un mínimo local***, y es uno de los principales problemas con los que se enfrentan los que aplican algoritmos genéticos.

-El cruce es el encargado de mezclar bloques buenos que se encuentren en los diversos progenitores, y que serán los que den a los mismos una buena puntuación. La presión selectiva se encarga de que sólo los buenos bloques se perpetúen, y poco a poco vayan formando una buena solución.

-El intercambio genético se puede llevar a cabo de muchas formas, pero hay dos grupos principales

- ***Crucer n-puntos***: los dos cromosomas se cortan por n puntos, y el material genético situado entre ellos se intercambia. Lo más habitual es un cruce de un punto o de dos puntos.

- ***Cruce uniforme***: se genera un patrón aleatorio de 1s y 0s, y se intercambian los bits de los dos cromosomas que coincidan donde hay un 1 en el patrón. O bien se genera un número aleatorio para cada bit, y si supera una determinada probabilidad se intercambia ese bit entre los dos cromosomas.

- ***Cruces especializados***: en algunos problemas, aplicar aleatoriamente el cruce da lugar a cromosomas que codifican soluciones inválidas; en este caso hay que aplicar el cruce de forma que genere siempre soluciones válidas. Un ejemplo de estos son los operadores de cruce usados en el problema del viajante.

Repercusión del número de cruces en el algoritmo:

Rata de cruce número esperado de cromosomas que se cruzaran

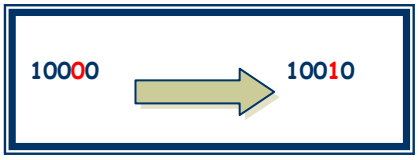
Alta: permite mayor exploración del espacio de búsqueda => reduce la posibilidad de quedar atrapado en un óptimo local.

Muy Alta: pérdida de tiempo computacional explorando regiones no prometedoras del espacio de búsqueda.

1.4.10. Mutación

-En la Evolución, una mutación es un suceso bastante poco común (sucede aproximadamente una de cada mil replicaciones), como ya se ha visto anteriormente. En la mayoría de los casos las mutaciones son letales, pero en promedio, contribuyen a la diversidad genética de la especie. En un algoritmo genético tendrán el mismo papel, y la misma frecuencia (es decir, muy baja).

-Una vez establecida la frecuencia de mutación, por ejemplo, uno por mil, se examina cada bit de cada cadena cuando se vaya a crear la nueva criatura a partir de sus padres (normalmente se hace de forma simultánea al cruce). Si un número generado aleatoriamente está por debajo de esa probabilidad, se cambiará el bit (es decir, de 0 a 1 o de 1 a 0). Si no, se dejará como está. Dependiendo del número de individuos que haya y del número de bits por individuo, puede resultar que las mutaciones sean extremadamente raras en una sola generación.



Rata de Mutación => ***pm : % del número total de genes en la población. => controlar la tasa a la cual se introducen nuevos genes en la población.***
Muy Baja: genes útiles no serán tratados.
Muy Alta: mucha perturbación aleatoria=> los hijos pierden parecido con sus padres=> algoritmo pierde habilidad de aprender del pasado.

-No hace falta decir que no conviene abusar de la mutación. Es cierto que es un mecanismo generador de diversidad, y, por tanto, la solución cuando un algoritmo genético está estancado, pero también es cierto que reduce el algoritmo genético a una búsqueda aleatoria. Siempre es más conveniente usar otros mecanismos de generación de diversidad, como aumentar el tamaño de la población, o garantizar la aleatoriedad de la población inicial.

-Este operador, junto con la anterior y el método de selección de ruleta, constituyen un ***algoritmo genético simple,*** **sga**, introducido por **Goldberg** en su libro.

1.4.11. Aplicando operadores genéticos

-En toda ejecución de un algoritmo genético hay que decidir con qué frecuencia se va a aplicar cada uno de los algoritmos genéticos; en algunos casos, como en la mutación o el cruce uniforme, se debe de añadir algún parámetro adicional, que indique con qué frecuencia se va a aplicar dentro de cada gen del cromosoma. La frecuencia de aplicación de cada operador estará en función del problema; teniendo en cuenta los efectos de cada operador, tendrá que aplicarse con cierta frecuencia o no. Generalmente, la mutación y otros operadores que generen diversidad se suele aplicar con poca frecuencia; la recombinación se suele aplicar con frecuencia alta.

-En general, la frecuencia de los operadores no varía durante la ejecución del algoritmo, pero hay que tener en cuenta que cada operador es más efectivo en un momento de la ejecución. Por ejemplo, al principio, en la fase denominada de *exploración*, los más eficaces son la mutación y la recombinación; posteriormente, cuando la población ha convergido en parte, la recombinación no es útil, pues se está trabajando con individuos bastante similares, y es poca la información que se intercambia. Sin embargo, si se produce un estancamiento, la mutación tampoco es útil, pues está reduciendo el algoritmo genético a una búsqueda aleatoria; y hay que aplicar otros operadores. En todo caso, se pueden usar operadores especializados.

2. Resúmenes

2.1. Resumen en inglés.

-The objective of this project, has been to implement conventional games applying the techniques of the evolutionary algorithms..

-The evolutionary algorithms are a group of algorithms search based on the evolution and natural selection of the species in the real world. These techniques include genetic algorithms, which work with binary structures to represent the individuals of a population.

-The games are a very interesting part within the field of the evolutionary programming, since in them many of the typical characteristics of the evolutionary algorithms can be applied.

-In this project we have implemented a great number of games like mastermind, puzzles, labyrinths... and other games where the opponent has been the computer.

-For each game we have chosen the representation of the individuals and the more appropriate function of aptitude to obtain games that gave back optimal solutions, purpose first of the evolutionary algorithms.

2.2. Resumen en español.

-El objetivo de este proyecto, ha sido implementar juegos convencionales aplicando las técnicas de los algoritmos evolutivos.

-Los algoritmos evolutivos son un grupo de algoritmos de búsqueda basados en la evolución y selección natural de las especies en el mundo real. Estas técnicas incluyen algoritmos genéticos, los cuales trabajan con estructuras binarias para representar a los individuos de una población.

-Los juegos son una parte muy interesante dentro del campo de la programación evolutiva, ya que en ellos se pueden aplicar muchas de las características típicas de los algoritmos evolutivos.

-En este proyecto hemos implementado un gran número de juegos como el mastermind, puzzles… y otros juegos donde el contrincante ha sido el ordenador.

-Para cada juego hemos elegido la representación de los individuos y la función de aptitud más apropiadas para conseguir juegos que devolvieran soluciones óptimas, finalidad primera de los algoritmos evolutivos.

2.3. Palabras clave.

Programación Evolutiva.
Algoritmos Genéticos.
Función de adaptación.
Población.
Operadores genéticos: mutación, selección, cruce.
Individuos.
Juegos Genéticos.
Elitismo.
Azar.
Selección natural.

3. JUEGOS IMPLEMENTADOS.

3.1. Juego de las Cuatro en Raya.

Índice:

1. Introducción

-Después de realizar todos los juegos, pensamos en realizar alguno con contrincante, es decir, algún juego donde un usuario jugará contra el ordenador o donde el ordenador jugara contra sí mismo, poniendo en práctica dos o más funciones de adaptación, y viendo cuál de ellas obtenía mejor resultado. Pronto nos dimos cuenta de las dificultades que entrañaría este tipo de juego, veamos por qué:

1. Pensar en juegos clásicos, como el ajedrez, las damas…requiere una cantidad de cálculos enorme, algo que ralentizaría mucho el juego…por no hablar de lo difícil y complicado que resultaría dar con una función de adaptación adecuada para el buen rendimiento del ordenador, en este caso. Pensamos que se conseguirían buenos resultados, pero con un trabajo muy tedioso, propio, quizás, de un proyecto único y exclusivo dedicado a ese menester.

2. Por otra parte, la mayoría de los juegos en los que se enfrentan dos o más jugadores, son juegos en los que uno de los jugadores realiza una jugada e inmediatamente se requiere la jugada de algún otro jugador, lo que provoca que en cada momento, lo que era bueno antes, puede no ser bueno en la jugada actual, y al no poder prever lo que va a hacer tu contrincante provoca que las decisiones se tomen en cada instante del juego. Eso, trasladado a un juego genético puede suponer, si quieres que los resultados sean buenos, una población nueva por cada jugada a realizar, y un recálculo constante de las funciones de adaptación y demás…

-Con todo y con eso, pensamos en un juego que no requiriera muchos cálculos, a priori, o no tantos como el ajedrez, y que sin embargo cubriera nuestro interés de hacer un juego con la filosofía pretendida. Nos decidimos por el cuatro en raya. El usuario compite con el ordenador. En un principio, en una primera versión, el primero que consiguiera las cuatro en raya era el ganador, pero en una segunda, el juego consistía en conseguir más fichas de cuatro en línea que el rival, ganando así en entretenimiento. La dificultad con la que nos encontramos, fue en dar con una implementación adecuada para que el ordenador se mostrara lo más competitivo posible, siempre teniendo en cuenta las limitaciones que el azar introduce en los algoritmos evolutivos. A continuación, presentamos el juego y sus características para que se entienda y se conozca mejor, en esta explicación se podrán ver con más detenimiento los problemas que nos han surgido, las distintas versiones que hemos hecho…Sólo destacar que el juego, para que ganara en complejidad, dista un poco de lo que es el típico cuatro en raya. En el cuatro en raya tradicional se juega sobre un tablero vertical donde las fichas siempre se ponen unas encima de otras, es decir no puedes poner una ficha en cualquier casilla, bien sea porque esa casilla está ocupada o porque la pila de fichas no haya llegado a la altura requerida. En nuestro caso, pensamos que era muy pocas las opciones a elegir por parte del ordenador, y que era bastante fácil por lo tanto decidir cuál es la casilla mejor, si el tablero era el clásico, así que pasamos a un tablero plano, tipo el del ajedrez, donde una ficha se puede poner en cualquier casilla, siempre, claro está , que esa casilla no esté ocupada.

2. Aspectos del Algoritmo Evolutivo

-Sin duda estamos ante el juego más atípico de todos los que hemos hecho. En un principio pensamos en generar individuos, que de alguna manera, representaran jugadas consecutivas, por ejemplo, tres, es decir que un individuo representaba tres jugadas, tres casillas en nuestro tablero para ser más precisos, pero pronto nos dimos cuenta que tal decisión sólo nos conducía a soluciones malas, es decir el ordenador, cuando quería reaccionar, ya había perdido, si el usuario jugaba de manera adecuada. Además, ¿cómo puedes esperar a poner tres casillas en el tablero, si no sabes a ciencia cierta lo que va a pasar después de tres rondas? Incluso puede que esas casillas, llegado el momento, estén ocupadas. Pronto nos dimos cuenta que nuestro planteamiento debía cambiar con respecto a lo visto anteriormente. La interacción con el usuario provoca que las jugadas se hagan dinámicas, y que lo que tú vayas a hacer dependa en todo momento de lo que el usuario (contrincante) haya hecho. Por ello decidimos generar una población de individuos en cada iteración, individuos que representarían una casilla a cubrir dentro del tablero.

a) Representación de la Población.

-La población es un conjunto de casillas, de números, que representan casillas, para ser más exactos. Todos esas casillas que genera el ordenador en cada ronda, deben caer, como es obvio, dentro de los límites del tablero, y además no pueden ocupar una casilla que ya está ocupada, o bien por el contrincante, o bien por nosotros mismos. La estructura que hemos utilizado para representar la población es la siguiente:

```
vector<TIndividuo*> _poblacion;/*Un vector de punteros a individuos*/
```

b) Generación de la Población.

-Como hemos dicho antes, la población es un vector de punteros a los individuos, la generación de los individuos es la común, los generamos y almacenamos junto a cada individuo su adaptación.

c) Representación de los Individuos y Generación de los Individuos.

-Un individuo es una casilla de nuestro tablero:

```
int x=casilla/12;  /*La fila*/
int y=casilla % 12  /*La columna*/
while(!(_juego->esNegro(x,y))){
casilla=random(143);
x=casilla/12;
y=casilla % 12;
}
_juego->ponerRojo(x,y); adaptacion();
```

-Como podemos ver, cada individuo, es una componente x y una componente y. Para que una casilla pueda ser ocupada tiene que ser negra, al principio, es decir, tiene que estar libre.

d) Función de Adaptación.

-Es aquí donde vamos a poner más énfasis, porque es aquí donde más empeño hemos puesto. Como decimos, la población es un conjunto de casillas que el ordenador baraja, es decir, el ordenador cuenta con una serie de posibilidades, y la decisión de cuál de todas ellas marcar es la decisión clave. Por lo tanto la función de adaptación es importantísima.

-Para que una jugada sea buena, se tiene que tener en cuenta, en un principio varias cosas:

a) Que la jugada consiga el mayor número posible de fichas en línea, siendo el cuatro en raya la mejor de las jugadas.

b) Que la jugada tapone las buenas jugadas de nuestro contrincante.

c) Y si nuestra jugada no consigue nada positivo en un primer instante, que la jugada por la que nos decidamos, si tenga expectativas de futuro, es decir que sea provechosa, que emprenda algo positivo.

-La cuestión está en determinar qué jugadas son más buenas que otras, y qué jugadas son claves en un momento dado, para darle más peso a unas que a otras. Como podemos ver es imprescindible conocer el estado del tablero en todo momento, y para ello contamos con una matriz en la que una casilla puede estar en los siguientes estados:

```
enum Estado{AZUL, ROJO, NEGRO, DADO, DADO2 };
```

-Vamos a explicar lo que significa cada estado:

AZUL, las casillas azules son las que ha puesto el usuario.
ROJO, las casillas rojas son las que ocupa el ordenador.
NEGRO, las casillas libres son negras.
DADO, las casillas que han pasado a formar parte de una jugada de cuatro en raya por parte del usuario pasan a este estado. Esto se hace con una única finalidad, apartarlas del resto de casillas para que no se confundan con las listas y para cambiarlas de color, para que se note que ya están marcadas y que forman parte de una jugada de cuatro en raya.
DADO2, lo mismo que dado pero en el caso de que la jugada haya sido realizada por parte del ordenador.

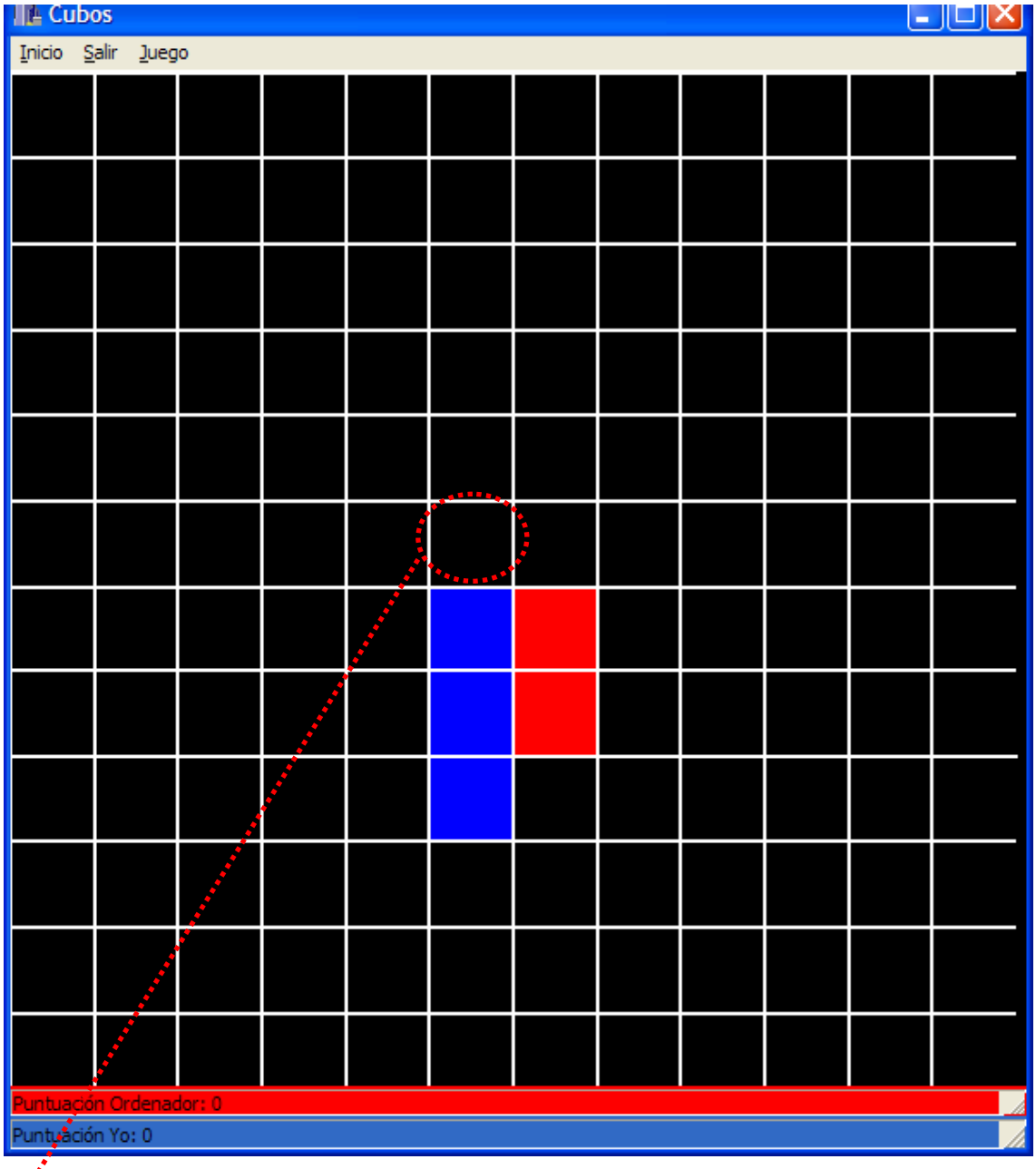
-Con esta decisión, la de contar con una matriz que represente el estado del tablero, conocemos la situación del juego en todo momento:

```
vector<vector<void*> > *matriz;
```

-El peso que hemos dado a cada jugada sigue la siguiente jerarquía:

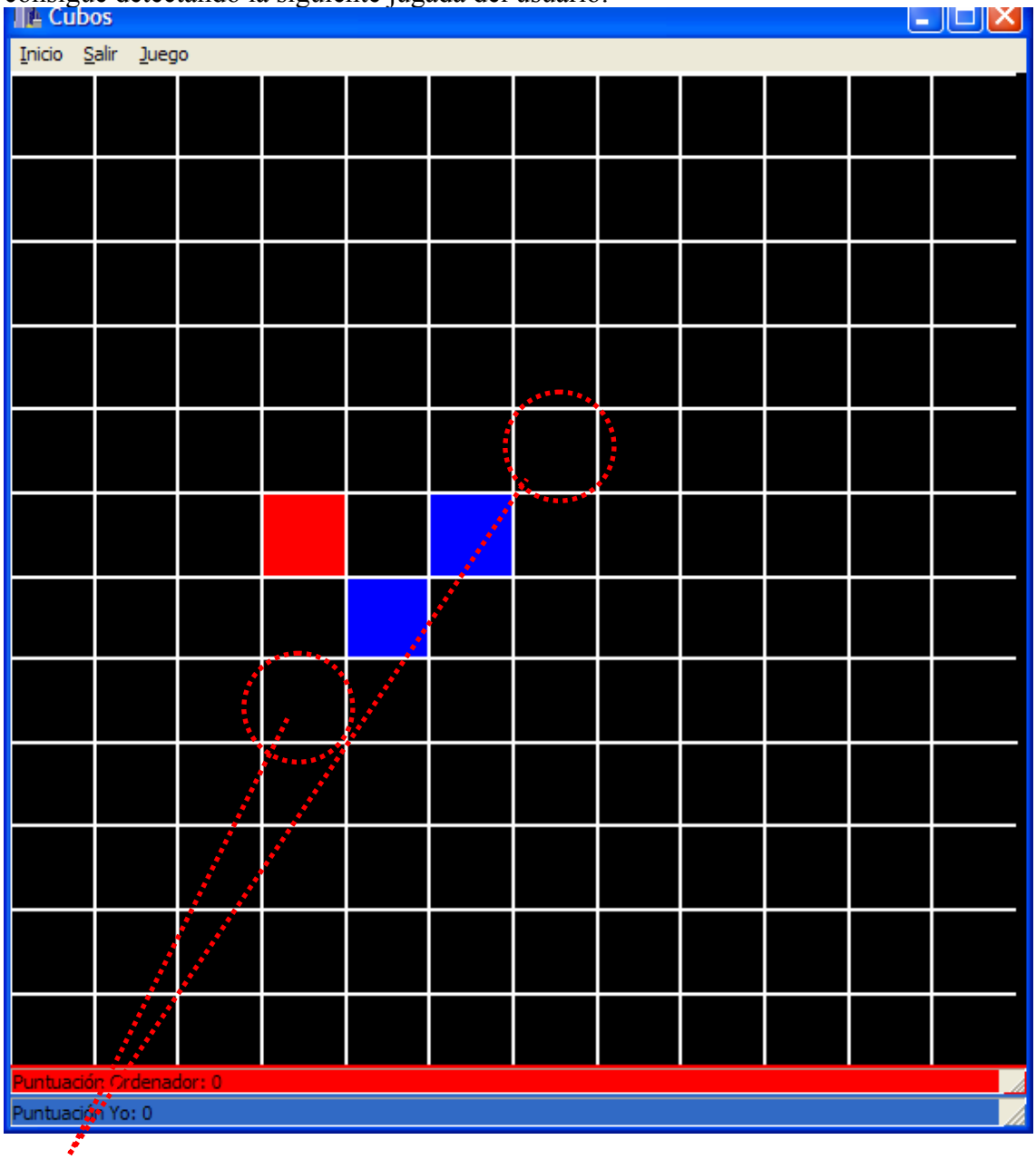
-**Aptitud = 3000**, si con la jugada conseguimos las cuatro en raya.

-**A la aptitud se le suma un valor de 1.500**, si bloquea alguna jugada de tres en raya del usuario y si el usuario no tiene ninguna posibilidad de conseguir por el otro extremo tres en raya. Es decir si el ordenador bloquea una jugada de tres en raya por un extremo, pero el otro extremo de la otra jugada está libre, no le damos ninguna puntuación ya que la jugada sería un tanto absurda. El usuario, en la siguiente iteración, lo que tiene que hacer es poner su ficha en el extremo opuesto que lo haya hecho el ordenador:



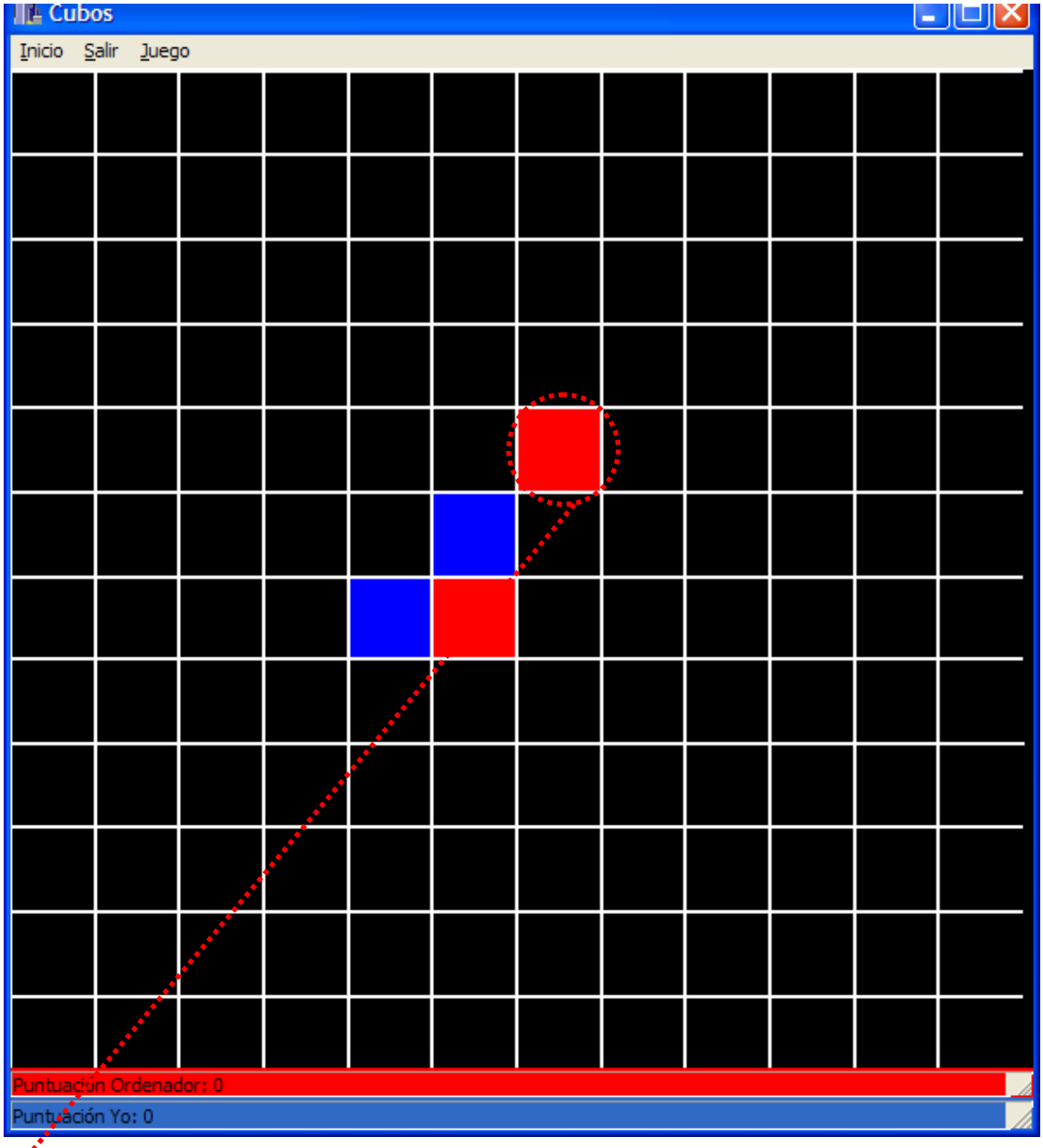
Poner aquí, sería una jugada perdida, porque el usuario tan sólo tiene que poner en el otro extremo para conseguir las cuatro en raya.

-A la aptitud se le suma 800 en caso de que consiga evitar que se de la situación anterior, es decir que el usuario consiga de manera irremediable las cuatro en raya, porque tenga tres en raya y dos posibilidades de hacer cuatro en raya. Esto se consigue detectando la siguiente jugada del usuario:



El ordenador debe ocupar alguna de las dos casillas.

El usuario juega de azul, y tiene dos en línea en diagonal, si el ordenador juega bien debería tapar alguno de los extremos del usuario, de esta manera garantiza que en la siguiente jugada el usuario puede conseguir tres en raya, pero con una única posibilidad de progresar hacia los cuatro en raya porque uno de los extremos está tapado. Así que una buena jugada por parte del ordenador sería cualquiera de las marcadas en la captura.



Aunque la jugada no es la misma, la situación es idéntica y vemos como el ordenador ha jugado de manera correcta.

-A la aptitud se le suma 850, en el caso de que la jugada consiga tres en raya. Recordemos que en esta segunda versión lo importante es hacer el mayor número de cuatros en raya posible, y si te puedes garantizar cuanto antes un cuatro en raya mejor. De esta manera si te garantizas un tres en raya que fácilmente se puede convertir en cuatro en raya, no hay prisa por evitar los tres en raya del contrincante, en el caso peor se empata.

-Por último, se multiplica por un peso de 50 por cada casilla en raya que se consiga con una jugada y 25 por cada casilla del contrincante que se bloquea.

-Hay que destacar la posibilidad de que en una misma jugada se consiga varias situaciones de las que consideramos buenas, por lo tanto si se da una no debemos de descartar las demás. Veamos el código que ordena todo esto:

```
if ((_juego->obtenerNumEnRayaVertical()==0)
   &&(_juego->obtenerNumEnRayaHorizontal()==0)
   &&(_juego->obtenerNumEnRayaDiagonal1()==0)
   &&(_juego->obtenerNumEnRayaDiagonal2()==0)
   &&(_juego->obtenerNumBloqueaEnVertical()==0)
   &&(_juego->obtenerNumEnRayaHorizontal()==0)
   &&(_juego->obtenerNumBloqueaEnDiagonal1()==0)
   &&(_juego->obtenerNumEnRayaDiagonal2()==0))
    /*Entonces es que solo hay una en raya y no bloquea ninguna jugada*/
  _aptitud=1;
 if ((_juego->obtenerNumEnRayaVertical()==4) ||
    (_juego->obtenerNumEnRayaHorizontal()==4)||
    (_juego->obtenerNumEnRayaDiagonal1()==4)||
    (_juego->obtenerNumEnRayaDiagonal2()==4))
  _aptitud=3000;
 if ((_juego->obtenerNumBloqueaEnVertical()==3)&&
    (!(_juego->obtenerPenVertical3())))
        _aptitud=_aptitud+1500;
 if ((_juego->obtenerNumBloqueaEnHorizontal()==3)&&
    (!(_juego->obtenerPenHorizontal3())))
        _aptitud=_aptitud+1500;
 if ((_juego->obtenerNumBloqueaEnDiagonal1()==3)&&
    (!(_juego->obtenerPenDiagonal13())))
        _aptitud=_aptitud+1500;
 if ((_juego->obtenerNumBloqueaEnDiagonal2()==3)&&
    (!(_juego->obtenerPenDiagonal23())))
        _aptitud=_aptitud+1500;
 if (_juego->obtenerNumEnRayaVertical()==3)
        _aptitud=_aptitud+850;
 if (_juego->obtenerNumEnRayaHorizontal()==3)
        _aptitud=_aptitud+850;
 if (_juego->obtenerNumEnRayaDiagonal1()==3)
        _aptitud=_aptitud+850;
 if(_juego->obtenerNumEnRayaDiagonal2()==3)
        _aptitud=_aptitud+850;
 if (_juego->obtenerPlusVertical())
        _aptitud=_aptitud+800;
 if (_juego->obtenerPlusHorizontal())
        _aptitud=_aptitud+800;
 if (_juego->obtenerPlusDiagonal1())
        _aptitud=_aptitud+800;
 if(_juego->obtenerPlusDiagonal2())
        _aptitud=_aptitud+800;
```

```
  _aptitud=_aptitud+25*_juego->obtenerNumBloqueaEnVertical()+
        25*_juego->obtenerNumBloqueaEnHorizontal()+25*_juego-
>obtenerNumBloqueaEnDiagonal1()+
        25*_juego->obtenerNumBloqueaEnDiagonal2()+
        50*_juego->obtenerNumEnRayaVertical()+50*_juego-
>obtenerNumEnRayaHorizontal()+
        50*_juego->obtenerNumEnRayaDiagonal1()+50*_juego-
>obtenerNumEnRayaDiagonal2();

 /* recogemos los valores */
 numEnRayaDiagonal1=_juego->obtenerNumEnRayaDiagonal1();
 numEnRayaDiagonal2=_juego->obtenerNumEnRayaDiagonal2();
 numEnRayaVertical=_juego->obtenerNumEnRayaVertical();
 numEnRayaHorizontal=_juego->obtenerNumEnRayaHorizontal();
}
```

-Como podemos deducir, los cálculos son un tanto tediosos. Se pueden conseguir bloqueos y casillas en líneas en varias direcciones, en horizontal, en vertical, en una diagonal y en otra. Y además la ficha que se coloca, no siempre tiene que estar en un extremo, puede ponerse entre medias otras fichas ya puestas, lo que dificulta el cálculo aún más. Una vez que pones una ficha en una casilla, debemos mirar en todas las direcciones para ver cuál es la nueva situación.

-Pasamos a explicar dos tipos de variables (una con el mismo significado por cada orientación) que aparecen en las sentencias anteriores:

obtenerPenVertical3(),*es un procedimiento que te devuelve si se está dando el caso anterior, el de que la jugada esté taponando un tres en raya sin posibilidades de que la jugada prospere, porque el usuario puede conseguir los tres en raya tanto por un extremo como por otro.(lo mismo para obtenerPenHorizontal3()…).*

obtenerPlusVertical(), *procedimiento que se pondrá a true si la jugada evita la situación anterior, en la que el usuario se ha garantizado un cuatro en raya haga lo que haga el ordenador.*

-Observar como una misma jugada puede meterse por varias ramas del if, de esta manera la aptitud= aptitud + plus por buena jugada, es decir una jugada puede acumular, por decirlo de alguna manera, varios premios.

e) Función de Selección, Función de Reproducción y Función de Mutación.

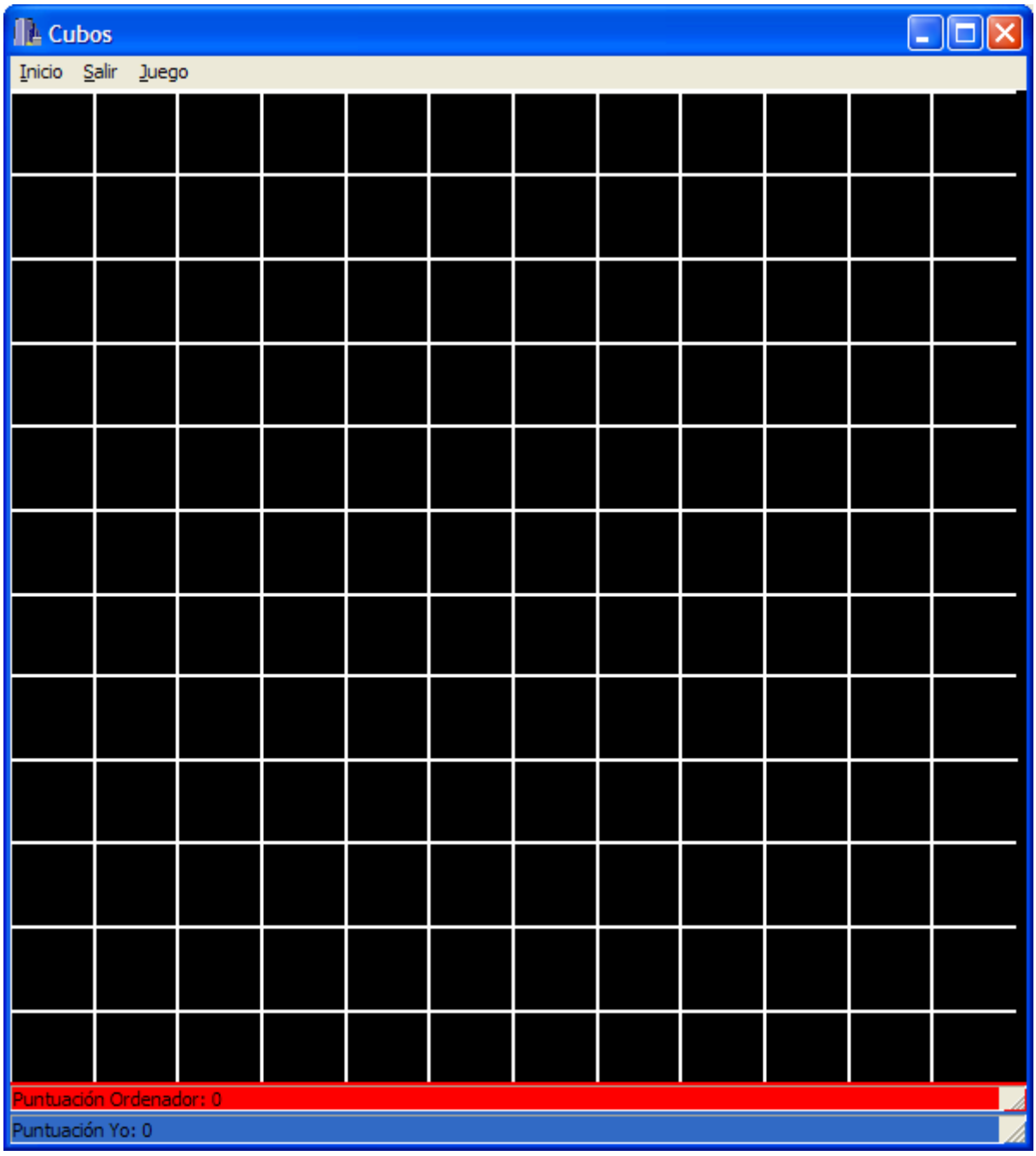
-Como hemos ido viendo hasta el momento, poco sentido tiene una función de selección, una función de reproducción y una función de mutación. La selección no tiene sentido porque dos números no se pueden mutar ni reproducir, y además en cada tirada sólo hay una iteración, donde en esa primera y única iteración se elige al mejor de los individuos. Es por ello que habíamos pensado en tres tiradas para poder darle sentido a cada uno de estas partes del algoritmo genético, pero pronto vimos que era

imposible si queríamos un ordenador competitivo. Lo único que tiene sentido es la función de adaptación y la función de evaluación.

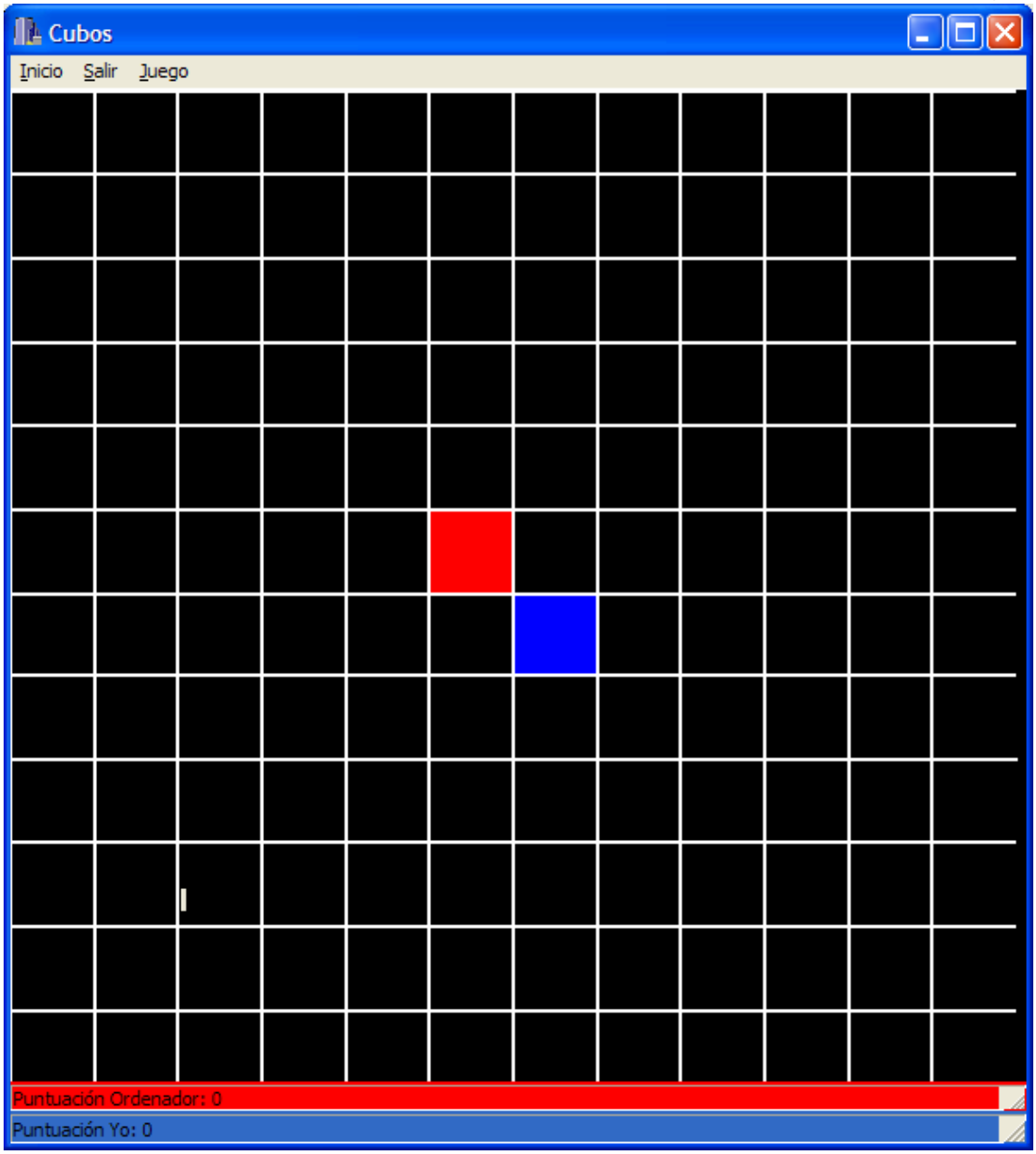
3. Presentación de la solución (entorno gráfico).

-Ya hemos mostrado un poco antes, pero vamos a dar ahora, en esta su sección más detalles:

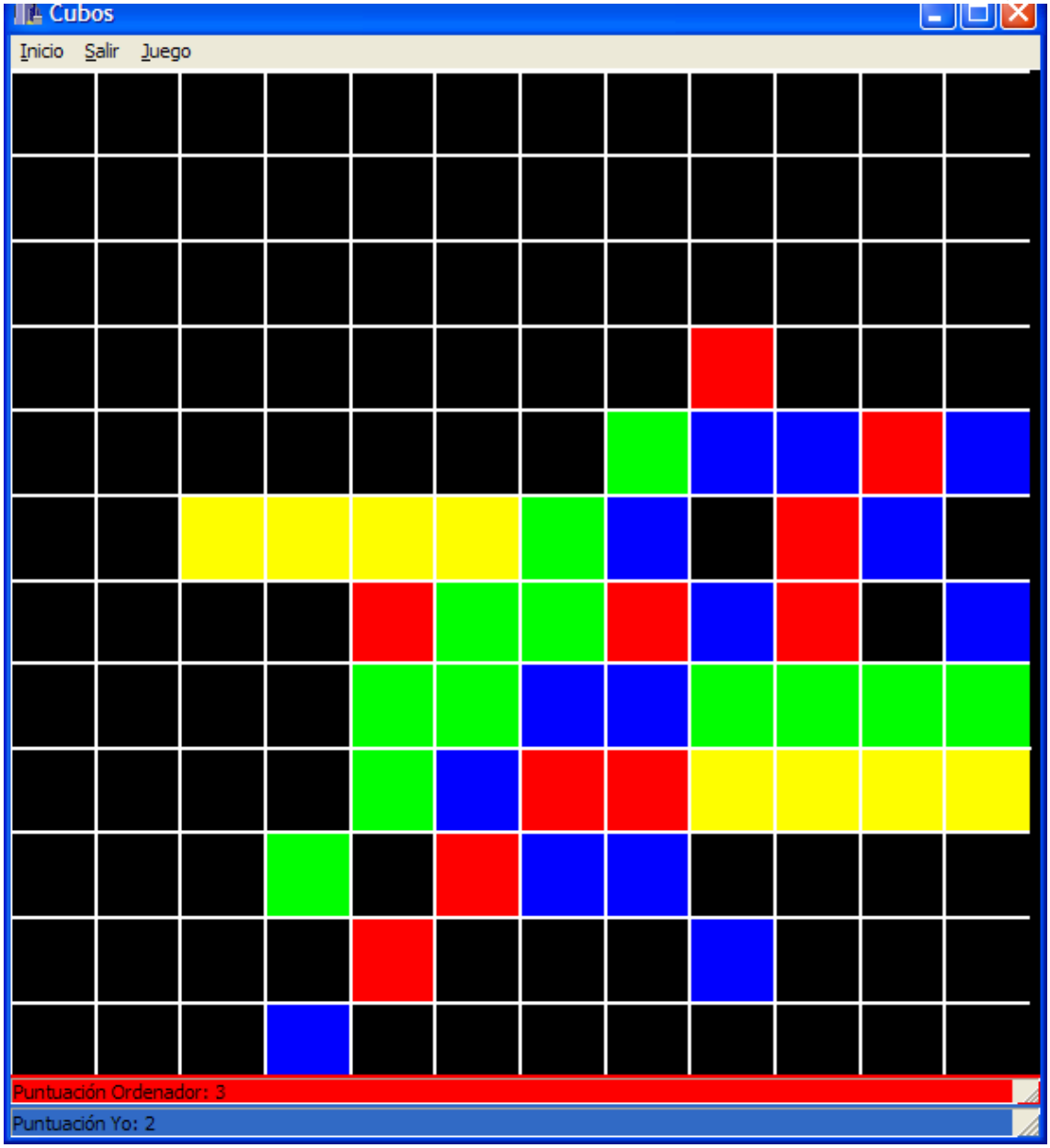
El tablero:



Bastante claro: pinchando en una casilla pones una ficha. En todo momento queda visible tu puntuación y la del ordenador.



Las casillas del ordenador se marcan en rojo (en el mismo color que su marcador) y las del usuario en color azul (también como su marcador).



Las casillas que se muestran en color amarillo, son casillas que el ordenador ha conseguido poner como cuatro en línea(dado2) y las de color verde las que ha conseguido poner el usuario(dado1). De esta manera se distingue perfectamente lo que lleva conseguido cada usuario. A efectos prácticos esas casillas son como paredes, no se pueden ocupar con nuevas fichas y tampoco sirven para enlazar con otras del tablero y conseguir así alguna combinación de casillas en línea.

4.-Versiones de la aplicación.

-Como ya hemos dicho con anterioridad, de este juego hay dos versiones. Una que se acerca más al cuatro en raya original, en el sentido en el que la partida se acaba en cuanto alguno de los dos jugadores consiga las cuatro en raya. En esta primera versión, a nuestro modo de ver un poco deslucida por lo rápido que se acababan las partidas, premiaba evitar que el contrincante realizara las cuatro en raya a través de las

jugadas de sobra conocidas, aunque esto supusiera olvidar un poco tu jugada propia, pero era esencial que el usuario, que además siempre es el que inicia la partida, no consiguiera las cuatro en raya antes que tú. En la segunda versión se deja que el usuario haga con más facilidad esas cuatro en raya, siempre que el ordenador garantice que le puede empatar en la siguiente jugada, porque al final lo que cuenta es tener mayor número de éxitos posibles, no obtenerlos antes que el contrincante, que era el caso de la primera versión.

5.-Tecnologías empleadas.

-Las tecnologías utilizadas han sido básicamente **C++** y **Open GL.** El Open GL es un SW(en forma de librería C) que permite la comunicación entre el programador y el HW de la máquina para el diseño de gráficos.

-En este punto vamos a explicar todo lo que podamos sobre la interacción entre el **C++** y **Open Gl** de la parte gráfica de nuestra aplicación, sin tampoco pretender ahondar en los conceptos propios de **Open GL.**

-Lo primero que hemos de resaltar que todo lo que se pinta en una escena creada por Open GL se hace a través de un método princial que es:

```
void __fastcall TGLForm2D::GLScene(){
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// comandos para dibujar la escena

glFlush();
if(juego !=NULL) {
    JUEGO->DRAW();
     }
SwapBuffers(hdc);
}
```

-Lo principal de este método es la llamada marcada en rojo, la clase juego dispone de un método draw que es el que se encarga de pintar el tablero en la escena, pero esto no es del todo verdad, porque lo único que se encarga este procedimiento es ahondar un poco mas y acceder hacia capas mas interiores mediante la llamada al método draw de su atributo privado Ladrillos. Este objeto a su vez no es mas que una matriz de objetos Ladrillo, es a este nivel de profundidad donde realmente aparecen las llamadas a las librería de Open GL.

-Es en este punto donde podemos aclarar porque decidimos implementar la clase ladrillos con esta estructura de datos vector<vector<void*> > *matriz y no optamos por lo que a simple vista parece mas trivial, que es implementar una matriz de objetos ladrillo. Como ya hemos comentado arriba, realmente es a ese nivel, ósea al nivel de los objetos que guarda la matriz donde realmente se interactúa con las librerías gráficas de Open GL. De esta forma lo que conseguimos es separar el C++ del Open Gl, con las ventajas que ello conlleva. El tablero es de 12 por 12, porque vimos oportuno

crear un tablero suficientemente grande como para que no fuera tan trivial el jugar. El tamaño del tablero se denota a través de la constructora del objeto *Ladrillos* que representa una matriz de n*m de objetos cada uno de los cuales tiene su método para pintarse en la escena. .Nos hemos aprovechado del diseño utilizado para representar la parte gráfica en el juego de los cuadrados y lo hemos utilizado íntegro, realmente no se ha cambiado casi nada, ahora un juego es otra matriz de objetos ladrillo, solo que ahora el objeto ladrillo solo puede tener tres posibles estados, negro si la casilla está vacia, azul denota que esa casilla ha pinchada por el usuario en cuestión y rojo que denota que la casilla marcada por el algoritmo evolutivo.

-No hemos creído conveniente entrar en más detalles sobre el OpenGL, más que nada hemos querido mostrar la estructura tan interesante utilizada.

6.-Estudio de los parámetros del algoritmo.

-En ningún momento dejamos al usuario que configure los parámetros del tamaño de la población, ni el de tamaño del elitismo. Está claro que con valores muy bajos de casillas que pueda barajar el ordenador, es decir con un número muy pequeño de individuos, tiene todas las de ganar el usuario. Hemos establecido dos valores fijos en ambos casos, tanto para el tamaño de la población como para el tamaño del elitismo. En el primero de los casos 250 y en el segundo 50. Se puede pensar que 250 es mucho, pero si tenemos en cuenta que esos 250 se pueden concentrar en una única casilla, la cifra se queda pequeña. Esto es así, porque la casilla se elige de manera aleatoria entre todas las casillas del tablero, y en una misma generación no se puede ver la casilla que los demás individuos representan. En ningún momento hemos querido manipular la generación de individuos, y que de las 250 posibilidades genere 250 distintas (más que nada porque hay sólo 144 y en una misma iteración puede repetir una misma casilla las veces que quiera), de esta manera siempre elegiríamos con total seguridad el mejor de los individuos con respecto a nuestra función de adaptación, pero le quitaríamos toda la salsa que tiene. De esta manera el ordenador en algunas ocasiones, como el humano, se despista y no escoge la mejor casilla para ocupar, así es más bonito…

3.2. Juego de las Cifras.

Índice:

6. Estudio de los parámetros del algoritmo.

1. Introducción

El juego de las Cifras consiste en generar una expresión aritmética que se aproxime lo máximo posible a un número calculado al azar. Dicho número está comprendido entre 1 y 999, y la expresión aritmética se obtiene de combinar seis números obtenidos al azar usando operaciones básicas (suma, resta, multiplicación y división). No hace falta usar los seis números y no se pueden repetir ninguno de los seis números.

Los seis números aleatorios, que usaremos como operando, los obtenemos de la siguiente forma:

-Tres entre 1 y 9.
-Dos entre 1 y 99.
-Uno entre 1 y 999.

El objetivo del juego consiste en minimizar el valor absoluto de la diferencia del número a calcular y el valor de evaluar la expresión aritmética obtenida. Lo ideal sería que este valor fuera cero, con lo que habríamos obtenido el número exacto.

2. Aspectos del Algoritmo Evolutivo

a) Representación de la Población

La población está compuesta por:

- Un vector de punteros a individuos. En este vector se almacenan los punteros a los individuos que forman la población. El hecho de usar punteros mejora el rendimiento y velocidad de la aplicación, por el contrario se necesita tener más cuidado con la destrucción de los punteros, para que no se quede memoria ocupada de forma inútil. En el destructor de la población hay que eliminar uno a uno todos los punteros del vector. Este vector es de tipo clase ***Vector***, predefinida en el C++ Builder, y que proporciona un interfaz cómodo y eficaz para almacenar punteros a objetos, aumentando así el rendimiento de la aplicación
- El número de individuos de la población. Número de individuos que formaran la población
- Un puntero a un vector de operandos disponibles para generar la expresión aritmética. Este puntero será añadido a cada individuo que forma la población.
- Tamaño del elitismo. Este parámetro es esencial para obtener la solución óptima, ya que representa el número de mejores elementos que conservamos en cada iteración, para que no se pierdan dichos individuos.

b) Generación de la Población.

La generación de la población se realiza creando cada uno de los individuos y añadiendo el puntero de cada uno de los individuos al vector que representa la población.

Un aspecto importante es la destrucción de los individuos que forman el vector, es decir, no sólo hay que liberar los punteros del vector, sino que también hay que liberar el objeto que apunta cada uno de los punteros. En la liberación de memoria en el C++ Builder es importante estar atento a la destrucción de los punteros que ya no sean útiles, puesto que a diferencia con Java no tiene recopilación automática de memoria no útil.

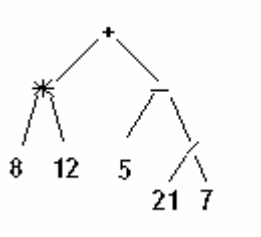
La recopilación de memoria no usada y el uso de punteros cobra especial relevancia en los algoritmos evolutivos puesto que en ellos se usa una gran cantidad de memoria y tiempo de cómputo, por lo que no conviene desperdiciarlas.

c) Representación de los Individuos.

En este problema cada individuo posee un árbol que representa la expresión aritmética, un vector con los operandos disponibles, el número a obtener, la adaptación, la puntuación y la puntuación acumulada.

El árbol es la representación de la expresión aritmética. Los nodos internos representan los operadores (+, -, *, /), y las hojas representan los operandos.

Ejemplo:



En este árbol hay 4 nodos internos (operadores), 5 hojas (operandos). El recorrido del arbol de forma infija nos da la expresión arimética, en este caso es : 8 * 12 + (5 – 21 / 7), cuyo valor es : 98.

El número máximo de hojas del árbol es ocho, que es el número de operandos disponibles, ya que no se pueden repetir, y el número mínimo de hojas es una. La evaluación del árbol de forma infija da como resultado el valor de la expresión aritmética. La evaluación en infija consiste en evaluar primero el hijo izquierdo, luego la raíz y por ultimo el hijo derecho. Recorriendo el árbol de forma infija también obtenemos la expresión aritmética del árbol, que una vez evaluada nos da el valor representado por el árbol.

El vector de operandos representa los números que se pueden usar para formar la expresión aritmética, sin poder repetirlos, y sin necesidad de usarlos todos.

d) Generación de los Individuos.

La parte más importante en la generación del individuo es la creación del árbol, que representa la expresión aritmética. Antes de invocar al constructor hay que inicializar algunas variables: un vector de operandos usados y el número de operandos usados, para que no se repitan operandos. Como es obvio el vector de usados hay que inicializarlo a falso y el número de operandos usados a cero. En este punto cabe destacar que ambas variables son punteros, referencias a memoria que permanecerán constantes a lo largo de la construcción de forma recursiva del árbol. Si no se hubiera hecho así los valores no se hubieran actualizado de forma correcta. La creación del árbol se realiza de forma aleatoria y recursiva.

Además de crear el árbol hay que actualizar una serie de variables:

- La lista de operadores disponibles.
- Puntuación.
- Puntuación acumulada.
- Número a obtener.

Por último hay que realizar la adaptación para calcular la aptitud del individuo. La adaptación consiste en evaluar el árbol, calculando el valor de la expresión aritmética.

e) Función de Adaptación

La función de adaptación sirve para calcular la aptitud del individuo. La aptitud nos sirve para comparar lo bueno que es un individuo con respecto a los demás individuos de la población. Mediante este valor podemos saber cual es la mejor solución y en el elitismo poder elegir los mejores individuos.

En este caso la función de adaptación consiste en evaluar el árbol de forma infija, (hijo izquierdo, raíz, hijo derecho). Este recorrido se hace de forma recursiva, explorando primero el hijo izquierdo del árbol, luego la raíz y por último el hijo derecho. El caso base es cuándo nos encontramos en una hoja. Una vez hecha la evaluación del árbol hay que hacer el valor absoluto de la diferencia entre valor buscado y el valor obtenido en la evaluación del árbol. Con este valor se actualiza la aptitud del individuo.

Durante el recorrido del árbol para calcular la adaptación se pueden producir excepciones, como la división entre cero o la división no entera. En dichos caso se acaba el recorrido del árbol y se penaliza el individuo con una aptitud elevada para que sea desechado en el proceso de selección.

f) Función de Selección.

Dependiendo de si se usa el elitismo se usa una función distinta en la selección. Los elementos que se seleccionan son aquellos que van a sufrir los procesos de reproducción y cruce.

1. Usando elitismo. Al usar elitismo se eligen los individuos de la población con mayor aptitud. Ya que la población se ordena de mayor a menor en función de la aptitud sólo hay que seccionar los primeros de la población. En el caso del elitismo los individuos seleccionados se duplican para evitar que se pierdan en la reproducción o en la mutación. Este proceso nos garantiza que no se pierdan los que tienden a parecer ser los mejores individuos, pero nos puede hacer caer en máximos locales. Esto se evita teniendo una población amplia y un porcentaje de cruce y mutación óptimo.

2. Sin usar elitismo. Se intentan elegir los mejores individuos, aunque también se seleccionan algunos individuos que en principio no parece que vayan a conducir a la solución óptima, pero que podría ser que al cambiar alguna de sus propiedades resulten dar la solución óptima. El hecho de elegir algún individuo que en principio no sea el más adecuado, se hace para evitar quedarnos en máximos locales, con lo que nunca podríamos alcanzar el máximo global, que sería la solución óptima. Para realizar este cálculo usamos la ***puntuación acumulada*** del individuo, que han sido calculadas en la ***adaptación*** y un factor aleatorio para elegir los individuos.

```
void TPoblacion::seleccion(){
  float aleatorio;
  int aux,i,pos_super;
  int *sel_super=new int[_tam_poblacion];
  vector<TIndividuo*> pob_aux;
  float prob;
  for(i=0;i<_tam_poblacion;i++){
    aleatorio=rand();
    prob=(aleatorio/(RAND_MAX+1));
    pos_super=0;
    while ((pos_super<_tam_poblacion)&&
      (prob>_poblacion.at(pos_super)->obtenerPunt_acu()))
        pos_super ++;
      sel_super [i] =pos_super;}
    for(i=0;i<_tam_poblacion;i++){
     TIndividuo *ind = new TIndividuo(_poblacion.at(sel_super[i]));
     pob_aux.push_back(ind);}
      /*eliminar _poblacion*/
    vector<TIndividuo*>::iterator it;
    for(it= _poblacion.begin();it!=_poblacion.end(); ++it)
      delete (*it);
    _poblacion.clear();
     for(i=0;i<_tam_poblacion;i++){
       TIndividuo *ind2 = new TIndividuo(pob_aux.at(i));
      _poblacion.push_back(ind2);}
     /*eliminar _poblacion auxiliar*/
    for(it= pob_aux.begin();it!=pob_aux.end(); ++it)
     delete (*it);
    pob_aux.clear(); }
```

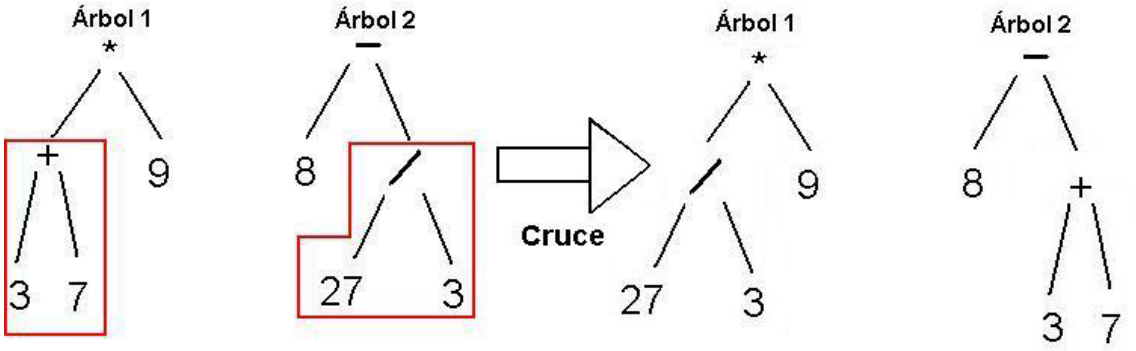
g) Función de Reproducción

La función de reproducción consiste en el cruce de dos árboles, reajustar los dos árboles y posteriormente calcular la aptitud de cada uno de los individuos por medio de la adaptación. El reajuste de cada uno de los dos árboles es necesario puesto que se pueden repetir operandos debido al cruce por lo que se crea un árbol no válido. El reajuste se realiza recorriendo el árbol y llevando la cuenta de los operandos usados, y en caso de encontrar uno repetido sustituirlo por uno no usado.

La decisión de cruzar o no dos árboles depende de un parámetro del algoritmo evolutivo, dicho parámetro es la probabilidad de cruce. Esto se hace calculando un número entre 0 y 100, se divide entre 100 y si es menor que la probabilidad de cruce entonces se decide cruzar los dos árboles. Este parámetro es muy importante en el algoritmo evolutivo, si el porcentaje es muy alto se cruzaría en exceso con el consiguiente gasto en tiempo de cómputo que acarrea, pues que la reproducción suele ser una operación costosa; también existe la posibilidad de perder individuos buenos, pero con el elitismo se soluciona este problema. Por el contrario, si el porcentaje es pequeño no se regenera la población, con lo que es difícil encontrar distintas soluciones que pueden ser mejor que las actuales. Un valor aproximado, que suele dar buenos resultados, es el 20%.

➢ Función de Cruce

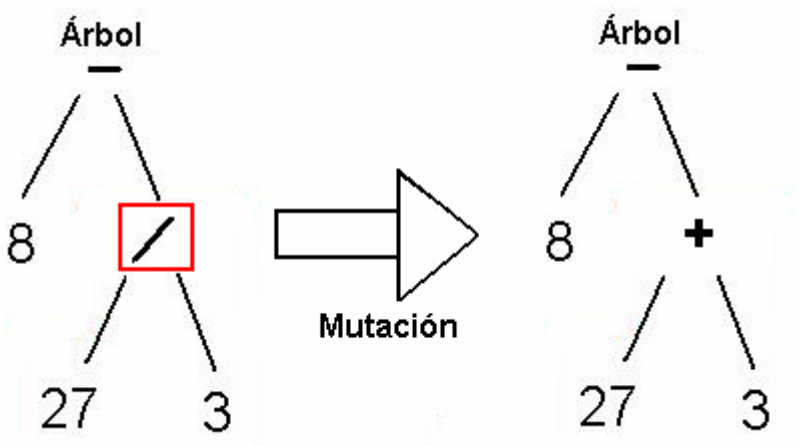
El cruce entre dos árboles consiste en intercambiar dos ramas de cada árbol. Más en concreto en intercambiar los punteros, ya que los árboles están implementados con punteros. Cuando se eligen los nodos a cruzar (dicha elección se realiza al azar), se intercambian los punteros a uno de los hijo de cada uno, y además también hay que recalcular el puntero al padre, el número de hojas, de nodos , de hijos y de operadores del los dos nuevos árboles creados. Una vez realizado el cruce hay que realizar el reajuste del árbol y volver a calcular su adaptación.



h) Función de Mutación

La mutación consiste en elegir un nodo interno del árbol y cambiar el operador del nodo entre todos los operadores disponibles. Los operadores disponibles son: la suma, la resta, la multiplicación y la división entera (+, -, *, /). Es evidente que sólo se pueden mutar los nodos internos, ya que son los que tienen los operadores. Si se elige una hoja, entonces dicho árbol no se muta. Una vez realizada la mutación hay que recalcular la aptitud del individuo, pues que el árbol ha cambiado, y lo más probable es que el valor de la expresión aritmética también haya cambiado.

Al igual que ocurre con el cruce existe una parámetro del algoritmo genético que indica si se va a mutar o no el árbol. Este parámetro se denomina porcentaje de mutación y su valor suele ser muy pequeño, entorno al 5%. Una vez que se ha producido la mutación es preciso volver a recalcular la aptitud del individuo, puesto que su árbol ha cambiado.



i) Función para Revisar la Aptitud

En los problemas en los que hay que minimizar el valor de la aptitud hay que usar la función de revisar la aptitud mínima después de calcular la adaptación. Esta función consiste en invertir la aptitud de los individuos, es decir, en el caso de funciones de adaptación en las que se trata de minimizar la aptitud los mejores individuos son los que menor aptitud tienen, por lo que es necesario invertir este valor dándoles la mayor aptitud a los mejores individuos y la menor a los peores.

La implementación de esta función consiste en recorrer toda la población y elegir la aptitud máxima, este valor lo llamaremos CMax. Posteriormente volveremos a recorrer la población actualizando la aptitud de cada individuo con:
CMax-poblacion.at(i)->obtenerAptitud().

El código de la función sería el siguiente:

```
CMax=-MAXLONG;
int i;
for (i=0;i<poblacion.size();i++)
   poblacion.at(i)->adaptacion();
for (i=0;i<(poblacion.size());i++)
  if (poblacion.at(i)->obtenerAptitud()>CMax)
    CMax= poblacion.at(i)->obtenerAptitud();
for (i=0;i<(_poblacion.size());i++)
  poblacion.at(i)->act_aptitud(CMax-poblacion.at(i)>obtenerAptitud());
```

j) Elitismo.

El elitismo es un factor esencial en los algoritmos evolutivos y consiste en preservar a lo largo de las sucesivas iteraciones los individuos mejores. Se considera que los mejores individuos son aquellos que tiene mayor aptitud, que se calcula gracias a la función de adaptación. Sin el elitismo es muy difícil conseguir buenas soluciones. Sin embargo, hay que tener en cuenta del elitismo es que hay que usar mucha más memoria y tiempo de cómputo.

Preservar los mejores individuos en cada iteración favorece que ciertas características buenas, que han aparecido en algunos individuos se preserven e incluso puedan evolucionar para obtener mejores individuos.

La implementación del elitismo se realiza en varios pasos.

- Primer Paso: consiste en calcular la adaptación de todos los miembros de la población, para así poder saber cual son los mejores.

- Segundo Paso: consiste en ordenar la población en función de la aptitud, quedando los individuos con mayor aptitud al principio de la población.

- Tercer Paso: consiste en seleccionar "n" de los primeros individuos de la población, que son los mejores ya que la población ha sido ordenada en función de la aptitud. El número "n" se conoce como tamaño de la población elitista, y es otro parámetro importante en los algoritmos evolutivos. Si el tamaño de la población elitista es elevado el tiempo de ejecución se dispara, por el contrario si es pequeño podemos desperdiciar individuos con características muy interesantes. La selección de estos "n" individuos consiste en crear una copia de ellos y almacenarlos en vector auxiliar para que no se vean afectados por la mutación y por el cruce.

- Cuarto Paso: una vez realizada la mutación y la reproducción se fusionan las dos poblaciones, la que resulta de la mutación y la reproducción, y la población auxiliar elitista.

- Quinto Paso: después de fusionar las dos poblaciones se ordena la población resultante en función de la aptitud, y se eliminan los peores individuos, para que la población tenga el número de individuos especificados.

k) Código adicional desarrollado para el programa.

Para el correcto funcionamiento del algoritmo hay que usar clases específicas para cada programa. Estas clases tienen poco que ver con la estructura esencial del algoritmo evolutivo, pero son necesarias para el correcto funcionamiento del algoritmo o para representar soluciones.

En este juego se podrían considerar dos clases adicionales a la estructura esencial del algoritmo evolutivo. Estas dos clases son: la *clase árbol* y la *clase pila*. De la clase árbol ya hemos hablado en profundidad en puntos anteriores. Ahora nos vamos a centrar en la *clase pila.*

La clase pila ha sido implementada para poder generar al expresión aritmética a partir del árbol que la representa. Es muy importante el uso adecuado de los paréntesis porque, de un mismo árbol se podrían obtener multitud de expresiones aritméticas dependiendo del posicionamiento de los paréntesis.

En este caso el árbol se evalúa de forma infija, (hijo izquierdo, raíz, hijo derecho). Para saber cuando hay que poner paréntesis es necesario el uso de una pila donde se van guardando los operandos. El funcionamiento es el siguiente, al principio se apila un operador con poca prioridad, luego se va recorriendo el árbol de forma infija. Si se encuentra un nodo hoja se escribe su valor, si se encuentra un nodo interno se comprueba la prioridad de la cima de la pila, si es tiene mayor prioridad la cima entonces se apila el operador, se escribe la expresión con los paréntesis y se desapila el operador. En caso de que la prioridad de la cima de la pila sea menor no se escriben los paréntesis. Los operadores con mayor prioridad son la multiplicación y la división, y los q menos la suma y la resta.

El siguiente código muestra como se genera la expresión aritmética.

En la clase principal:

```
TPila *pila=new TPila();
pila->apilar('#');
Edit3->Text=poblacion->obtener_individuo(pos_mejor)- >obt_arbol()>cal_expre(pila);
```

En la clase Arbol:

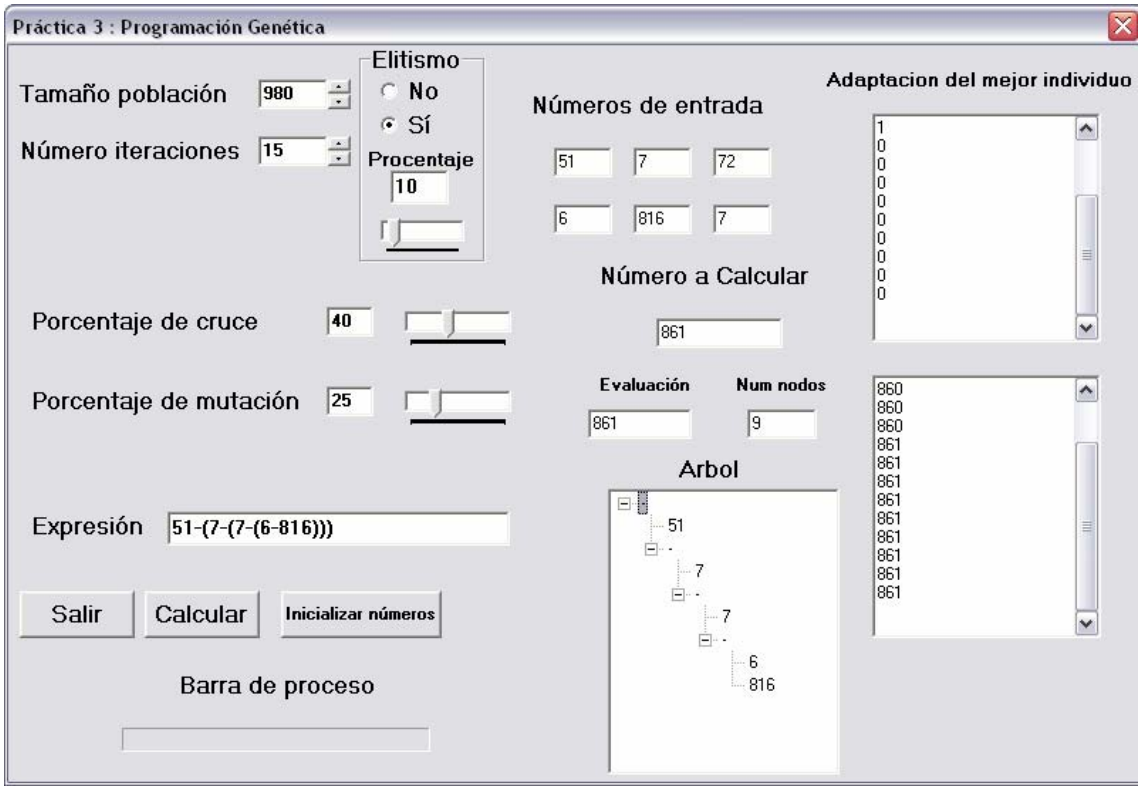
```
String TArbol::cal_expre(TPila *pila){
   String result="";
   if (num_hijos==0)
    result= String(valor);
   else {
    if(prioridad (pila->get_cima())>=prioridad(caracter)){
     pila->apilar(caracter);
     result="("+hijos.at(0)->cal_expre(pila) + String(caracter)+
                 hijos.at(1)->cal_expre(pila) +result+=')';
     pila->desapilar();
     }
     else{
       pila->apilar(caracter);
       result=hijos.at(0)->cal_expre(pila) + String(carácter) +
              hijos.at(1)->cal_expre(pila);
       pila->desapilar();
     }
   }
  return result;
}
```

3. Presentación de la solución (entorno gráfico).

Para representar la solución del juego se ha implementado un formulario, en el cuál se pueden configurar todos los parámetros del algoritmo evolutivo (número de iteraciones, tamaño de la población, porcentajes de mutación y cruce,…), donde en cada iteración se puede ver la aptitud y el valor de la expresión del mejor individuo. Al final del algoritmo se puede ver el resultado en forma de expresión aritmética o en forma de árbol. Para generar la expresión aritmética usamos la clase *TPila,* descrita anteriormente. La representación del árbol se ha realizado usando la clase *TreeView,* que viene incluida en el C++ Builder. Mediante esta clase se pueden representar árboles en forma de una manera muy profesional.

Además de todo lo descrito, en el formulario se ofrece la posibilidad de que tanto los operandos como el número buscado sean introducidos por el usuario. De esta manera se pueden repetir búsquedas y buscar expresiones que el usuario quiera.

La aplicación posee una barra de estado, que nos indica el número de iteraciones que se llevan realizadas del algoritmo evolutivo.



Además del formulario del ejecutable, hemos implementado un acceso web para poder acceder a la aplicación. Este apartado se comentará en la sección: Tecnologías Empleadas.

4. Versiones de la aplicación

A lo largo del desarrollo de la aplicación se han desarrollado varias versiones.

Una **primera versión**, en la que se permitía la división entera, es decir se podían obtener decimales en la división. De esta forma se conseguían resultados muy buenos. En esta primera versión ya se generaba el *TreeView* que mostraba el árbol obtenido.

En una **segunda versión** se decidió impedir la división no entera, impidiendo que hubiera decimales. Esto hace que muchos individuos se desprecien por no cumplir este requisitito, pero aún así la aplicación proporciona unos resultados óptimos en un tiempo record. Esto se consigue mediante el buen uso de los punteros, tanto en la población como en los árboles. En esta segunda versión también se generó la expresión aritmética, que represente al árbol del mejor individuo.

La **última versión** consiste en la posibilidad de ejecutar el juego a través de Internet. Este aspecto a llevado una gran cantidad de tiempo por la poca información que hay de cómo generar dinámicamente paginas web en el C++ Builder. Finalmente optamos por usar las CGI-ASAPI, que genera un archivo *.dll*. Todo este proceso se describirá en el punto siguiente.

5. Tecnologías empleadas

Las tecnologías de interés usadas en la práctica han sido para la representación del entorno gráfico, tanto en la generación del ejecutable, cómo en el acceso web.

En la generación del ejecutable una de las tecnologías más relevantes que se han usado ha sido el uso de la clase *TreeView*. Con esta clase se consigue implementar un árbol en forma de carpetas de Windows, permitiendo representar árboles con cualquier número de hijos.

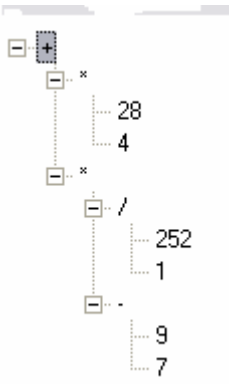
El código necesario sería:

```
void TForm1::mostrar_arbol (TArbol *arbol,TTreeNode *arbol_graf ){
 int i;
 TTreeNode *arbol_graf_rec;
 if (arbol->es_hoja())
   arbol_graf_rec=TreeView1->Items->AddChild(arbol_graf,AnsiString
(arbol->obt_valor()));
 else
   arbol_graf_rec=TreeView1->Items->AddChild(arbol_graf,AnsiString
(arbol->obt_nodo()));
  for (i=0;i<arbol->obt_num_hijos();i++)
   mostrar_arbol(arbol->obt_hijo(i),arbol_graf_rec);
}
```

Llamada al procedimiento:

```
  TreeView1->Items->Clear();
    TTreeNode *arbol_graf;
    arbol_graf=NULL;
    mostrar_arbol(poblacion->obtener_individuo(pos_mejor)->obt_arbol(),
                  arbol_graf);
```

Resultado gráfico:



Otra de las tecnologías empleadas ha sido la utilización de ***ASAPI/NSAPI Dynamic Link Library*** implementar el acceso web al juego. La tecnología ASAPI es similar a las CGI, y a partir de los datos de entrada enviados desde el cliente, mediante un método ***POST*** o ***GET***, procesa dichos datos y genera una página web dinámica, en función de los datos de entrada.

Para poder ejecutarse la dll ASAPI es necesario instalar en el servidor un ***servidor web*** que soporte esta tecnología. Nosotros nos decidimos por instalar el ***servidor web IIS de Microsoft***, sobre un ***Windows XP Profesional***. Este servidor es una utilidad integrada dentro del CD del Windows XP Profesional. Los resultados obtenidos con este servidor web han sido bastante buenos, lo único negativo es la cantidad de ataques que recibe por el puerto 80, aunque todos ellos han sido detectados y eliminados por el firewall del equipo.

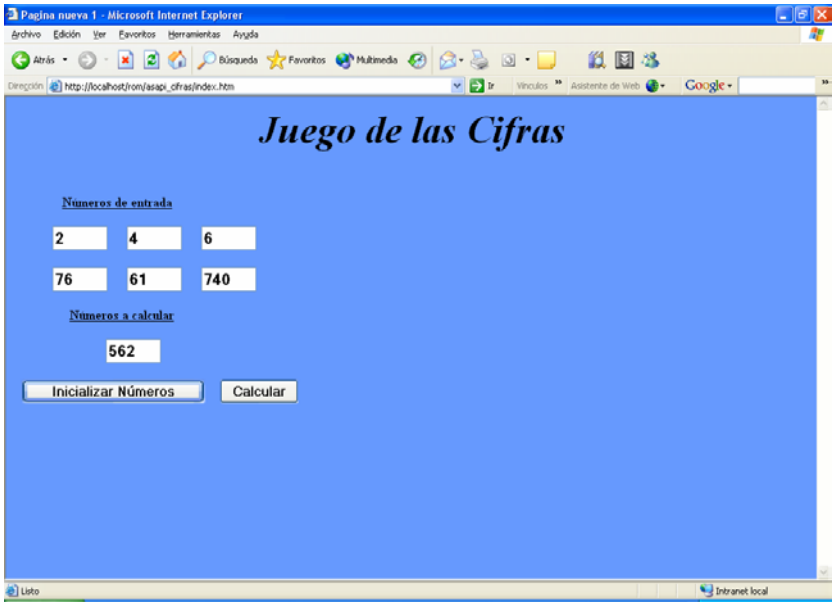
La generación de los números aleatorios, tanto los números operando cómo el número a generar son calculados en el navegador del cliente, usando ***javascrpit***. El código asociado es:

```
<SCRIPT LANGUAGE="JavaScript">
function inicializar( ){
   document.forms[0].num1.value =Math.floor(Math.round(9)*Math.random()+1);
   document.forms[0].num2.value =Math.floor(Math.round(9)*Math.random()+1);
   document.forms[0].num3.value =Math.floor(Math.round(9)*Math.random()+1);
   document.forms[0].num4.value =Math.floor(Math.round(90)*Math.random()+10);
   document.forms[0].num5.value =Math.floor(Math.round(90)*Math.random()+10);
   document.forms[0].num6.value=Math.floor(Math.round(900)*Math.random()+100);
   document.forms[0].numcal.value=Math.floor(Math.round(999)*Math.random()+1);
 }
</SCRIPT>
```

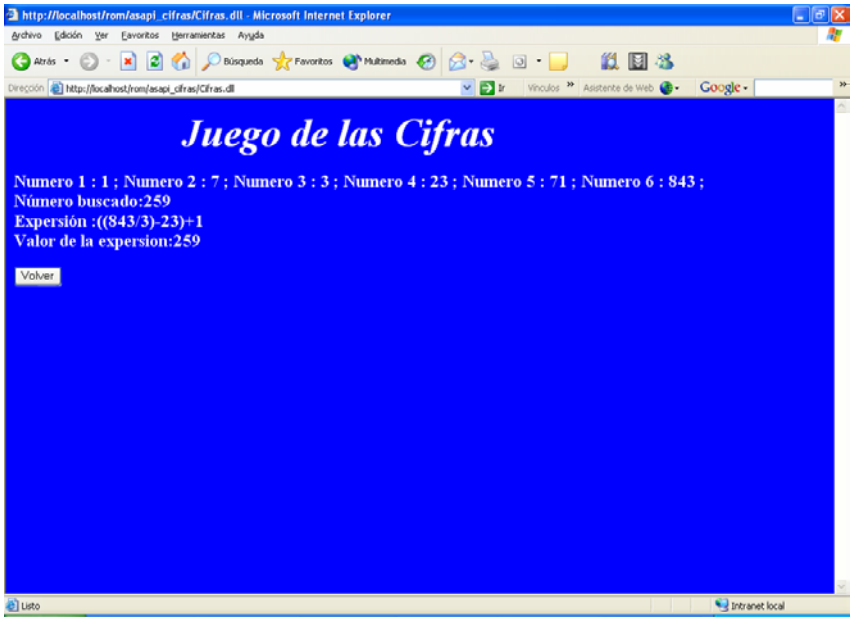
Una vez inicializados los números se produce el envío de los datos al servidor, para enviar los datos se pueden usar los métodos ***POST*** y ***GET***. La diferencia entre ambos métodos es que el método POST la información va encriptada y en el método GET no. La llamada desde el cliente al servidor sería la siguiente:

```
<FORM ACTION="http://localhost/rom/asapi_cifras/Cifras.dll" METHOD="POST" >
```

El servidor recibe los datos, ejecuta el algoritmo evolutivo con dicho datos y genera una página dinámica que es enviada al cliente, donde se muestran los resultados obtenidos.



Página solicitada al servidor por el cliente, donde mediante JavaScript se generan los números y de envían al servidor con el método POST.



Esta es la página que devuelve el servidor, mostrando los resultados obtenidos en el algoritmo evolutivo.

El código del C++ Builder que implementa la ***ASAPI/NSAPI*** sería el siguiente:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request, TWebResponse *Response,
      bool &Handled)
{
 time_t t;
 srand((unsigned) time(&t));
 char carac='"';
 String com=String(carac);
 String aux =  Request->ContentFields->Values["num1"];
 if (aux!=""){
   leer_datos(Request);
   CMax= num_buscado;
   poblacion=new TPoblacion(num_ind,lista_operadores,lista_operandos,
                            tam_pob_elit,num_buscado);
   poblacion->revisar_adapt_min(CMax);
   poblacion->evaluar(pos_mejor,suma_aptitud);
   for (int i=0;i< num_iterac;i++){
     poblacion->seleccion2 ();
     poblacion->reproduccion(porc_cruce);
     poblacion->mutacion(porc_mutac);
     poblacion->revisar_adapt_min(CMax);
     poblacion->fusionar();
     poblacion->evaluar(pos_mejor,suma_aptitud);
       }


  poblacion->obtener_individuo(pos_mejor)->obt_arbol()->obt_nodos ();
  TPila *pila=new TPila();
  pila->apilar('#');
    Response->Content=Response->Content+"<html><body      text='#FFFFFF'
bgcolor='#0000FF'>";
     Response->Content=Response->Content+"<p>"
         +"<marquee scrollamount='10' scrolldelay='30' style='font-size:
36 pt; font-style: italic; font-weight: bold'>"+
          "Juego de las Cifras</marquee></p>";
    Response->Content=Response->Content+"<b><font size='5'>";
    for (int i=0;i<lista_operandos->size();i++)
       Response->Content=Response->Content+"Numero       "+(i+1)+"       :
"+lista_operandos->at(i)+ " ; ";
```

```
          Response->Content=Response->Content+"</br>";
          Response->Content=Response->Content+"Número                buscado:"+
num_buscado+"</br>";
          Response->Content=Response->Content+"Expersión    :"+       poblacion-
>obtener_individuo(pos_mejor)->obt_arbol()->cal_expre(pila)+"</br>";
          Response->Content=Response->Content+"Valor    de    la    expersion:"+
poblacion->obtener_individuo(pos_mejor)->obt_arbol()->evaluar_Fun()+"</br>";
          Response->Content=Response->Content+"                          <SCRIPT
LANGUAGE="+com+"JavaScript"+com+">"+
           "function
vuelta(){this.location.replace("+com+"http://localhost/rom/asapi_cifras/index.
htm"+com+");}"+
           "</SCRIPT>"
           + "<form>"+
           "   <input   type=button   value=Volver   name=Boton   onClick=vuelta()
></b></font>"+
           "</form>"+"</body></html>";
         delete pila;
         delete poblacion;
         delete lista_operadores;
         delete lista_operandos;
         }
          else
            Response->Content="Rellene las casillas";

       }
       //---------------------------------------------------------------------------

       void TWebModule1::leer_datos(TWebRequest *Request){
         lista_operadores=new vector <char>();
         lista_operandos=new vector <int>();
         lista_operadores->clear();
         lista_operandos->clear();
         lista_operadores->push_back ('+');
         lista_operadores->push_back ('-');
         lista_operadores->push_back ('*');
         lista_operadores->push_back ('/');
         lista_operandos->push_back           (StrToInt(Request->ContentFields-
>Values["num1"]));
         lista_operandos->push_back           (StrToInt(Request->ContentFields-
>Values["num2"]));
         lista_operandos->push_back           (StrToInt(Request->ContentFields-
>Values["num3"]));
         lista_operandos->push_back           (StrToInt(Request->ContentFields-
>Values["num4"]));
         lista_operandos->push_back           (StrToInt(Request->ContentFields-
>Values["num5"]));
         lista_operandos->push_back           (StrToInt(Request->ContentFields-
>Values["num6"]));
         num_buscado=StrToInt(Request->ContentFields->Values["numcal"]);
         num_ind =2500;
         num_iterac=25;
         porc_cruce= 0.2;
         porc_mutac=0.05;
         tam_pob_elit=100;
       }
```

Como se puede observar, lo primero que se hace es inicializar la lista de operadores y extraer los operandos y el número a calcula de la variable ***Request***, que ha sido rellenada en el método ***POST*** del cliente. Se comprueba que las casillas han sido rellenadas y posteriormente se ejecuta el algoritmo evolutivo con dicho datos, y por último se genera la página dinámica con los resultados obtenidos. Esto se realiza rellenando la variable **Response**, con el código HTML asociado.

6. Estudio de los parámetros del algoritmo

El estudio de los parámetros en un algoritmo evolutivo es un aspecto esencial, ya que aunque el algoritmo evolutivo este bien implementado puede ser que una mala elección de dichos parámetros haga que se encuentren malas soluciones o que el tiempo de cómputo sea excesivo.

Los parámetros de un algoritmo evolutivo son muy concretos:

- **Tamaño de la población**, parámetro que indica el número de individuos que va a poseer la población. Si el tamaño es excesivo se ralentiza mucho la ejecución del algoritmo, pero si es demasiado pequeño no se generan suficientes individuos para encontrar la solución óptima.
- **Número de iteraciones**, número de veces que se ejecuta el algoritmo evolutivo. Muchas veces se puede parar si vemos que ya hemos encontrado la solución óptima, con lo que ahorramos tiempo de cómputo.
- **Porcentaje de cruce**, indica que porcentaje de la población se va a cruzar. Este factor es muy importante porque el cruce suele consumir mucho tiempo de cómputo y si se cruza muchos individuos se ralentiza el algoritmo y además podríamos perder alguna solución óptima, en el caso de no usar elitismo.
- **Porcentaje de mutación**, indica el porcentaje de individuos de la población que se van a mutar y es un factor menos determinante que el de cruce, ya que suele requerir menos tiempo de CPU, pero también tiene su importancia en la medida en que cada vez que se produce una mutación, hay que recalcular la aptitud, como en el caso del cruce, y el cálculo de la aptitud puede resultar muy costoso.
- **Tamaño de la población elitista**, indica el número de individuos que se conservan en cada iteración, si el número es muy elevado se produce un exceso de uso de memoria, al guardar temporalmente los individuos mejores, y de CPU al tener que fusionar y reordenar la población para elegir los mejores.

En el caso particular del juego de las cifras después de hacer una serie de pruebas, hemos llegado a la conclusión de que los valores ideales para los parámetros del algoritmo serían:

- **Tamaño de la población**: 2000 individuos.
- **Número de iteraciones**: 20 iteraciones.
- **Porcentaje de cruce**: 20 %
- **Porcentaje de mutación:** 5 %
- **Tamaño de la población elitista:** 10 % del Tamaño de la población.

Usando estos valores como parámetros del algoritmo se consiguen resultados espectaculares, consiguiendo siempre el número exacto, en caso de que exista, y en un tiempo record. En caso de querer mayor precisión, lo único que habría que hacer es aumentar el número de individuos y de iteraciones. Esto no supondría un aumento significativo en el tiempo de ejecución, ya que el algoritmo es muy veloz, gracias al uso de punteros, tanto en la población, como en los árboles.

3.3. Juego de los Cuadrados

Índice:

1. Introducción

-El juego de los cuadros es un juego clásico de intuición y estrategia, que se puede encontrar muy a menudo en portales de internet, en maquinas recreativas, en sistemas operativos, en móviles etc…

Existen un par de versiones diferenciadas del mismo, lo que varia de unas a otras es el objetivo del juego no en si las reglas del mismo.

-El juego de los cuadros no es mas que una representación de un tablero en 2D de dimensiones N*N dividido en casillas o cuadros de diferentes colores, en nuestro caso N=6. Es decir para nuestro caso en concreto disponemos de una matriz de 36 cuadros coloreados al azar.

-En el caso concreto del juego que hemos implementado, el objetivo es deshacer el tablero totalmente, o en su defecto (porque no todas las configuraciones de tableros originales tienen una solución óptima) conseguir quitar el máximo posible de cuadros.

-El juego puntúa de acuerdo al siguiente criterio, por cada cuadro que se elimina se suma un punto. Con este criterio la solución óptima es conseguir 36 puntos, o lo que es lo mismo, hacer un pleno y vaciar todo el tablero.

-Para poder eliminar un cuadro, este tiene que tener o bien en horizontal, o bien en vertical, **no en diagonal** , un cuadro del mismo color que el mismo. Nosotros optamos por disponer de 4 colores (Azul, Blanco, Morado y Rosa) , esto suele ser variable de algunas versiones a otras.

-Otro detalle importante que hay que mencionar es la recolocación que sufre el tablero tras una jugada, los huecos que se van creando tras una jugada , son ocupados por la columna que tiene justamente por encima , de igual manera los cuadros se van reagrupando hacia la derecha, es decir que nunca puede existir un tablero dividido a la mitad, las columnas que se vacían van quedándose a la derecha. (***Ver figura 6 y 7***).

-A continuación mostramos un ejemplo ilustrativo de la dinámica del juego para que sea más fácil la comprensión del mismo.

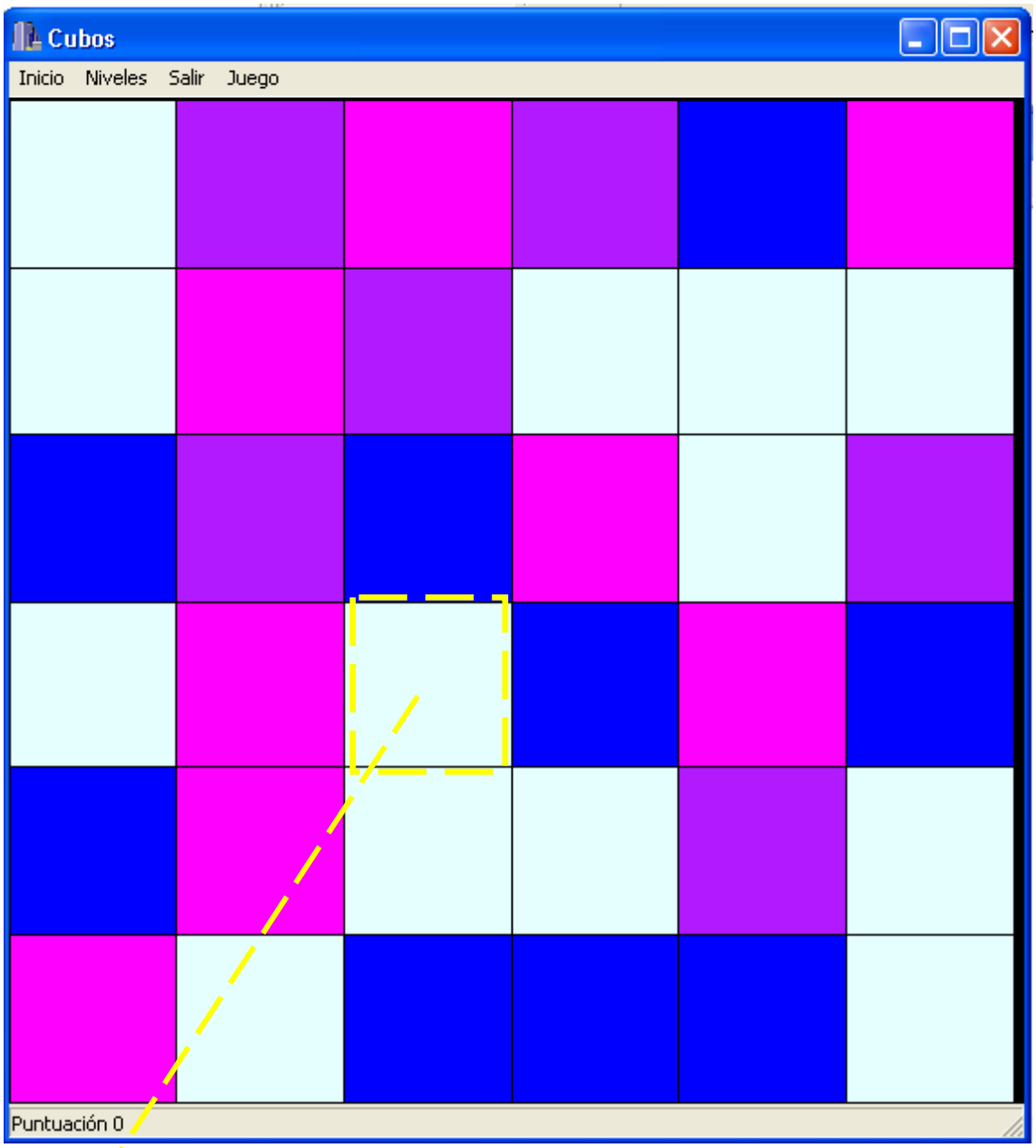


figura 1

-Supongamos que pinchamos en este cuadrado marcado en amarillo, a continuación se muestra el efecto que produciría sobre el tablero esta jugada. Los cuadros blancos contiguos al cual pinchamos desaparecen y se produce la recolocación del tablero.

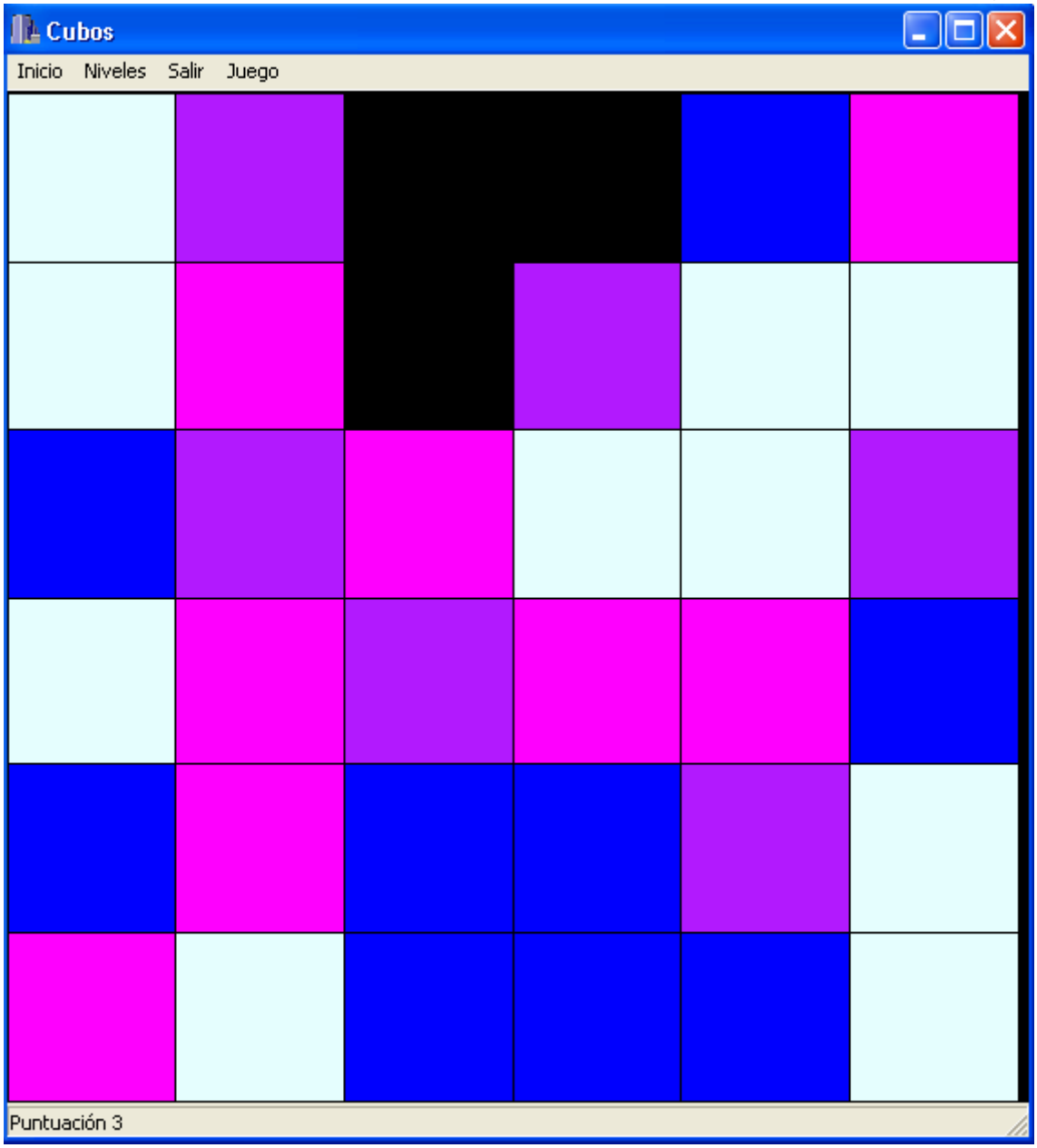


figura 2

-Una vez que se ha explicado la dinámica del juego vamos a mostrar un par de jugadas que dejen entrever la estrategia del juego.

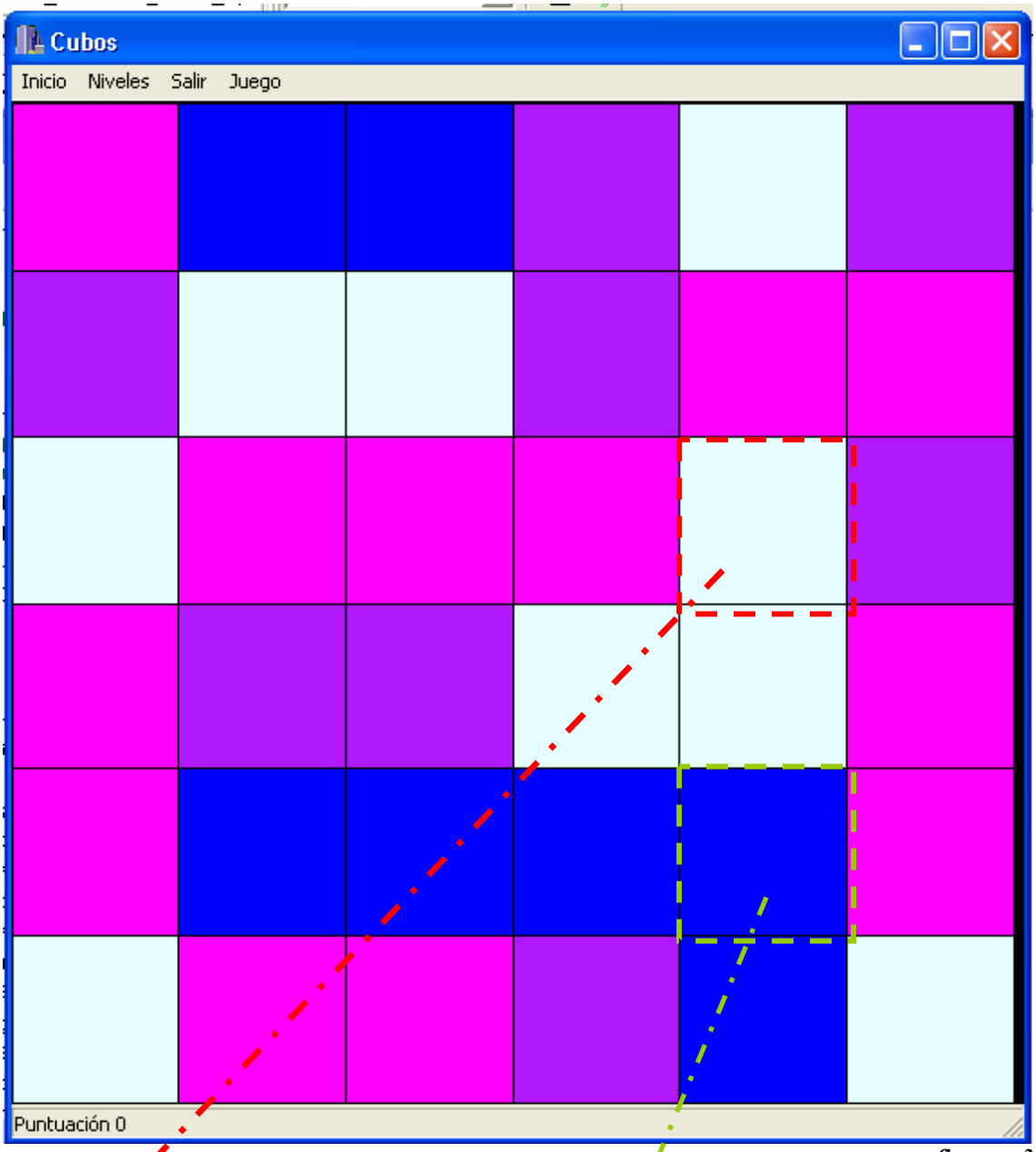


figura 3

- En rojo marcamos cual seria una posible jugada no del todo satisfactoria en lo que sería la resolución del juego, si optamos por la jugada denotada con el color rojo analicemos cual serán efectos, las tres blancas contiguas desaparecerían, pero para ello en principio debemos pagar un coste demasiado elevado, puesto como se puede ver a continuación dejamos las dos blancas (marcadas en amarillo en la siguiente figura) cerradas. Para llegar a una resolución optima normalmente no conviene aislar ningún cubo lejos de cubos del mismo color porque sino se nos acabaran cerrando y no será posible su eliminación.

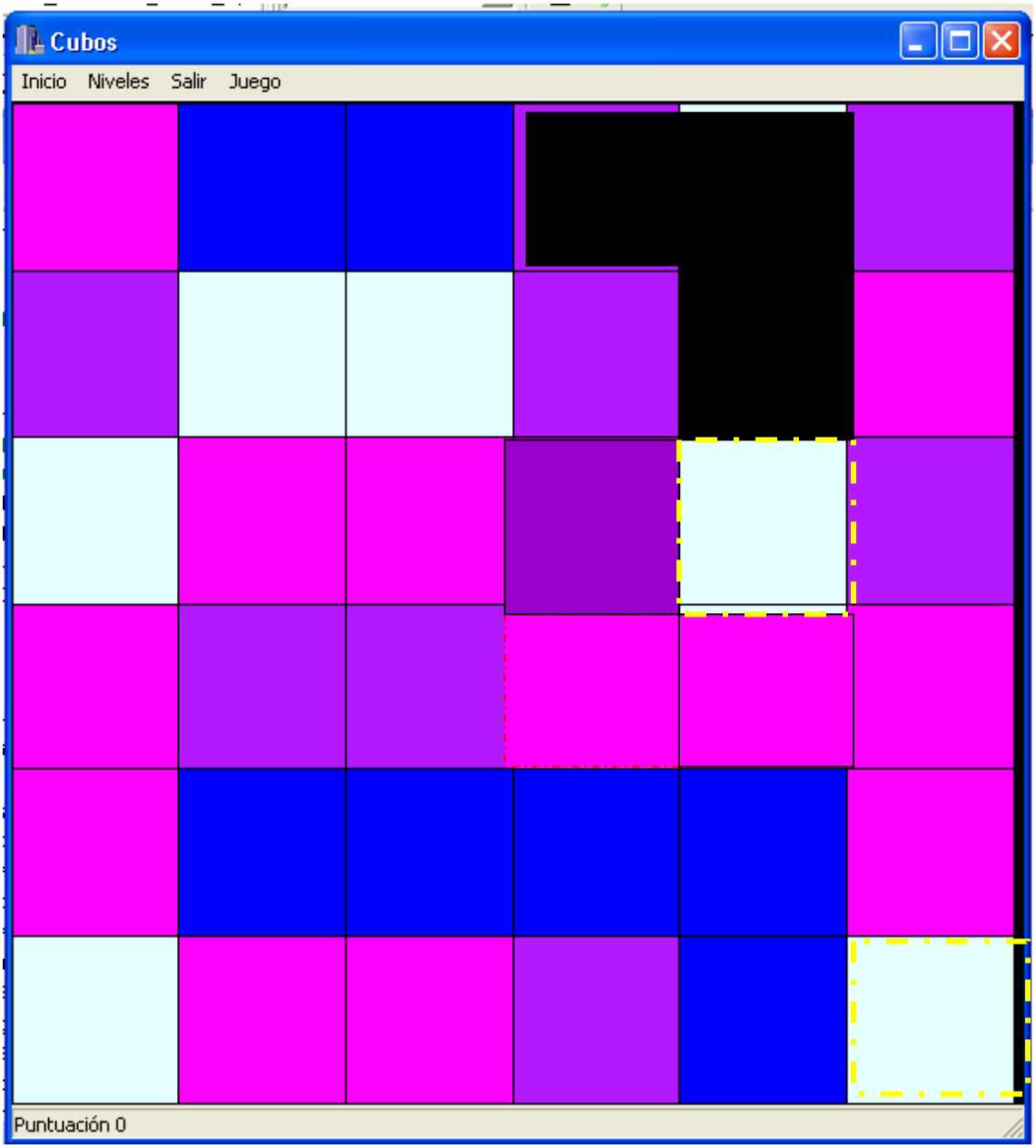


figura 4

- En verde marcamos una jugada que a priori parece ser mas positiva para la lógica humana. Ahora evitamos la situación negativa que se nos creaba al realizar la jugada roja, no nos quedan fichas blancas aisladas con lo cual esta jugada parece ser mas positiva para la resolución del juego.

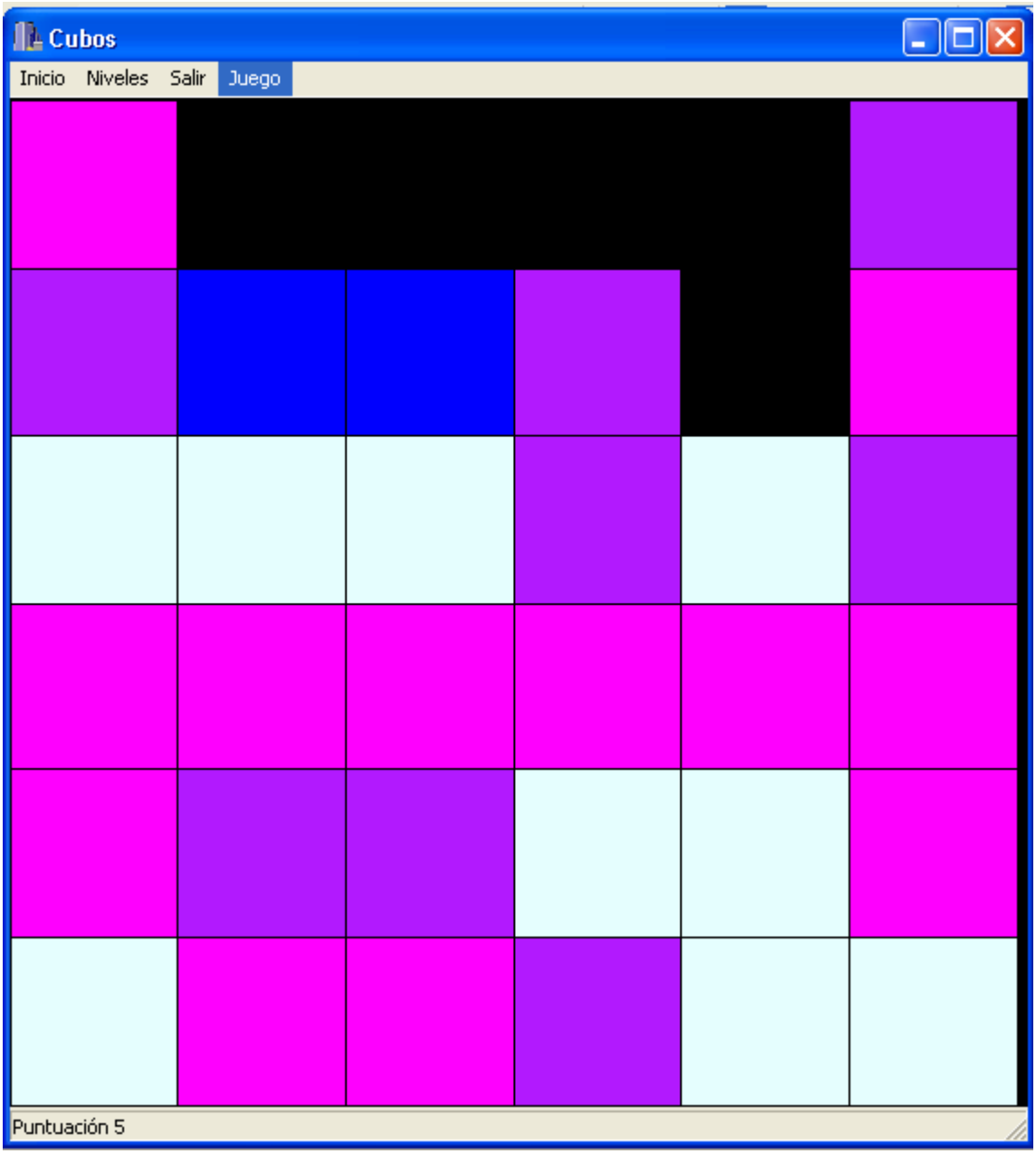


figura 5

-Como podemos ver ahora, al contrario que antes las blancas no quedan cerradas.

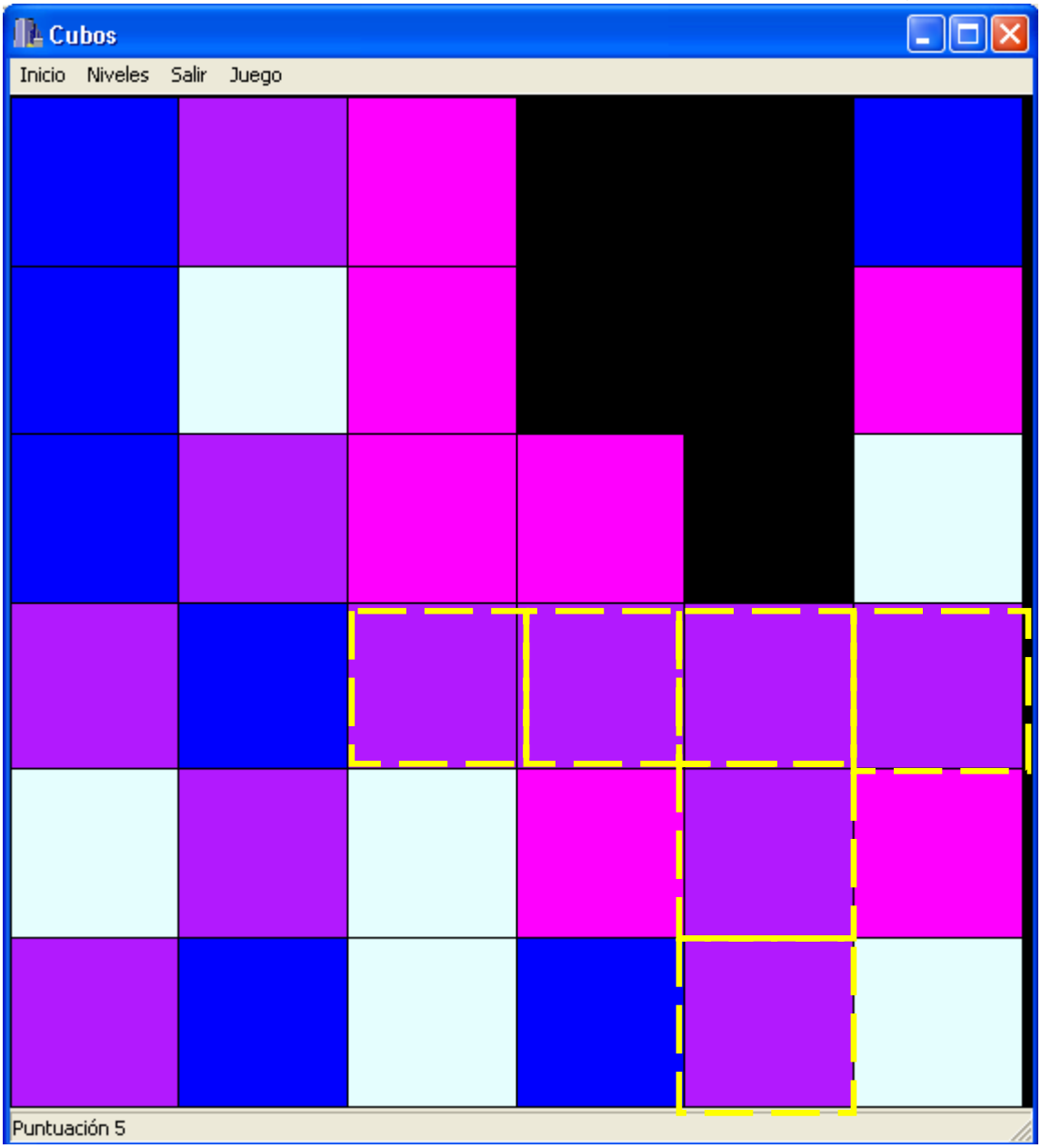


figura 6

-Aquí se muestra un ejemplo de recolocación de las columnas. Tras pinchar cualquiera de los cuadros marcados en amarillo, se nos vacía una columna.
El resultado de la recolocación se muestra en la siguiente captura:

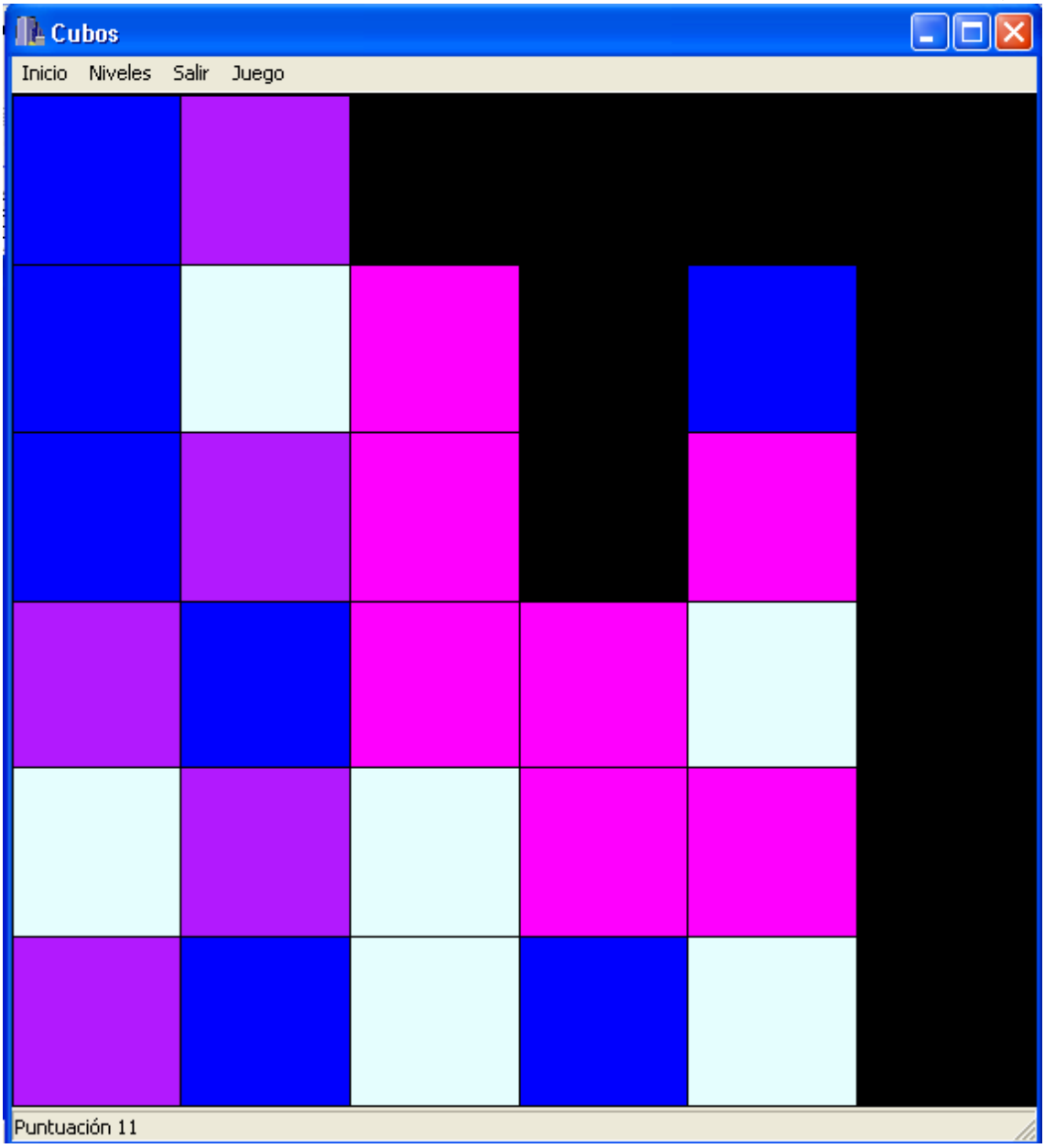


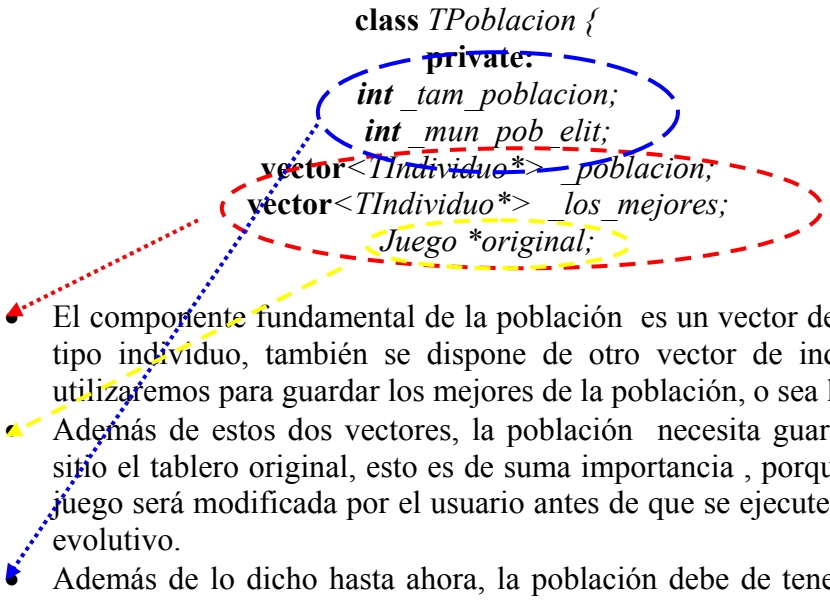
figura 7

-Como podemos ver, la recolocación evita que se nos cree una columna vacía, dividiéndonos el tablero en dos mitades, evitando así la posibilidad de que los cuadros se queden encerrados y no sea posible su destrucción.

2. Aspectos del Algoritmo Evolutivo

a) Representación de la Población

La población está compuesta por:



- El componente fundamental de la población es un vector de punteros de tipo individuo, también se dispone de otro vector de individuos que utilizaremos para guardar los mejores de la población, o sea la élite.
- Además de estos dos vectores, la población necesita guardar en algún sitio el tablero original, esto es de suma importancia , porque la variable juego será modificada por el usuario antes de que se ejecute el algoritmo evolutivo.
- Además de lo dicho hasta ahora, la población debe de tener variables que se encarguen de indicar el numero de individuos que tiene esa población y cuantos de esos individuos de la población formarán parte de la población elite.

b) Generación de la Población.

-Generar la población llama a generar cada uno de los individuos y después de crearlos se meten de manera adecuada en el vector población.

-Reseñar que además de crear todos los individuos, al generar la población debemos de guardar en la población la configuración del tablero inicial de la partida. Debido a esto hacemos una copia profunda(no compartida), del juego inicial que se aloja en la variable pública juego.

c) Representación de los individuos.

```
class TIndividuo {
private:
vector<int> *list_mov;

float _aptitud;
float _punt;
```

```
float _punt_acu;
Juego *_juego;
```

-Vamos a detenernos en describir cada individuo generado. Como podemos ver arriba un individuo consta además de los parámetros típicos utilizados en los algoritmos evolutivos como son la aptitud, puntuación y puntuación acumulada, también utilizamos un puntero a un vector de enteros, que representa una ristra de pulsaciones sobre nuestro tablero. Al principio pensamos en utilizar una lista de pares de enteros, porque a priori parece necesario que para acceder a un tablero o matriz de cuadrados se necesitan dos componentes, una coordenada *X* y una coordenada *Y*, pero como veremos a continuación esta representación aunque tampoco sería mucho mas difícil de implementar que por la que hemos optado, no será necesaria.

0	1	2	3	4	5
1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	25	26	27	28	29
30	31	32	33	34	35

Y

-Podemos optar por ver así nuestro tablero de juego, en vez de ver la casilla marcada como X=2, Y=3, pues la vemos como la casilla 16. Pero nos surgió un problema, aunque nosotros veamos la casilla marcada como la 16, la estructura física de los datos no se corresponde, no se puede acceder a juego[16], puesto que nuestro tablero es de la forma vector<vector<void*> > *matriz;(***mas adelante se explicara porque se utilizo una matrid de void)** entonces para solucionar este conflicto lo que hacemos es a través de la casilla 16 , sacamos su componentes X e Y de la siguiente forma :

```
int x = casilla/6;
int y = casilla % 6; // el operador % es el resto de la división entera.
```

-Por último reseñar que cada individuo también debe tener una copia profunda del tablero inicial del juego, sobre ese tablero es donde va a realizar su lista de movimientos.

d) Generación de los Individuos.

-De la generación de individuos constatar que los individuos los creamos con una longitud fija, también barajamos la posibilidad de mantener una longitud fija y sumarle un random(5) para que los individuos no tuvieran todos la misma longitud, esto no influye en nada en los resultados del algoritmo por la siguiente cuestión. La longitud el individuo da un poco igual en cuanto que los movimientos que son malos el algoritmo no los tiene en cuenta, para el algoritmo los individuos A y B son iguales suponiendo que la casilla 12,6,31 y 9 son movimientos que aportan algo productivo a la resolución del juego:

23	4	8	21	12	15	6	30	35	31	7	15	5	9

Individuo A

12	6	31	9										

Individuo B

e)Función de Adaptación

-En este juego fue bastante sencillo elegir la función de adaptación adecuada para poder ver la calidad es un individuo. La función de adaptación tan solo se encarga de contar los cuadros eliminados por el individuo. Para ello un individuo se encarga de realizar los movimientos en su tablero de juego, y una vez realizados todos ellos, se cuentan el número de huecos que existen. Un individuo será mejor que otro si el número de huecos que crea es mayor.

f) Función de Selección.

-En este caso, hemos utilizado elitismo, por lo tanto la selección deja intactos para la siguiente iteración los mejores individuos (un tanto por ciento dentro del número de individuos de la población que permita notar la mejora y que no ralentice en demasía el cómputo). Los seleccionados son los que se van a reproducir y mutar.

h) Función de Reproducción

La función de reproducción utilizada es sumamente simple y consiste en lo siguiente; se eligen dos puntos de cruce al azar uno en cada individuo que se va a reproducir (*marcados en rojo*) y el proceso es el siguiente

Individuo A

23	4	8	21	12	15	6	30	35	31	7	15	5	9

Individuo B

3	7	34	11	27	16	9	0	3	21	1	5	30	29

Hijo 1

23	4	8	21	12	5	30	29

Hijo2

3	7	34	11	27	16	9	0	3	21	1	15	6	30	35	31	7	15	5	9

-Como podemos apreciar, los nuevos hijos que se crean son de longitud variable, puesto que el punto de cruce se elige al azar.

i) Función de Mutación

-Mutar un individuo significa cambiar una parte de sus genes (en este caso concreto cambiar una parte de su lista de movimientos). La técnica utilizada es muy parecida a la de la función de reproducción, se elige un punto arbitrario de mutación, luego se elige al azar el sentido de la mutación y finalmente a partir de este punto se crean nuevos genes en un sentido u otro de la cadena en función del sentido elegido de la mutación que sustituyen a los antiguos.

Individuo

3	7	34	11	27	16	9	0	3	21	1	5	30	29

Individuo Mutado en un sentido

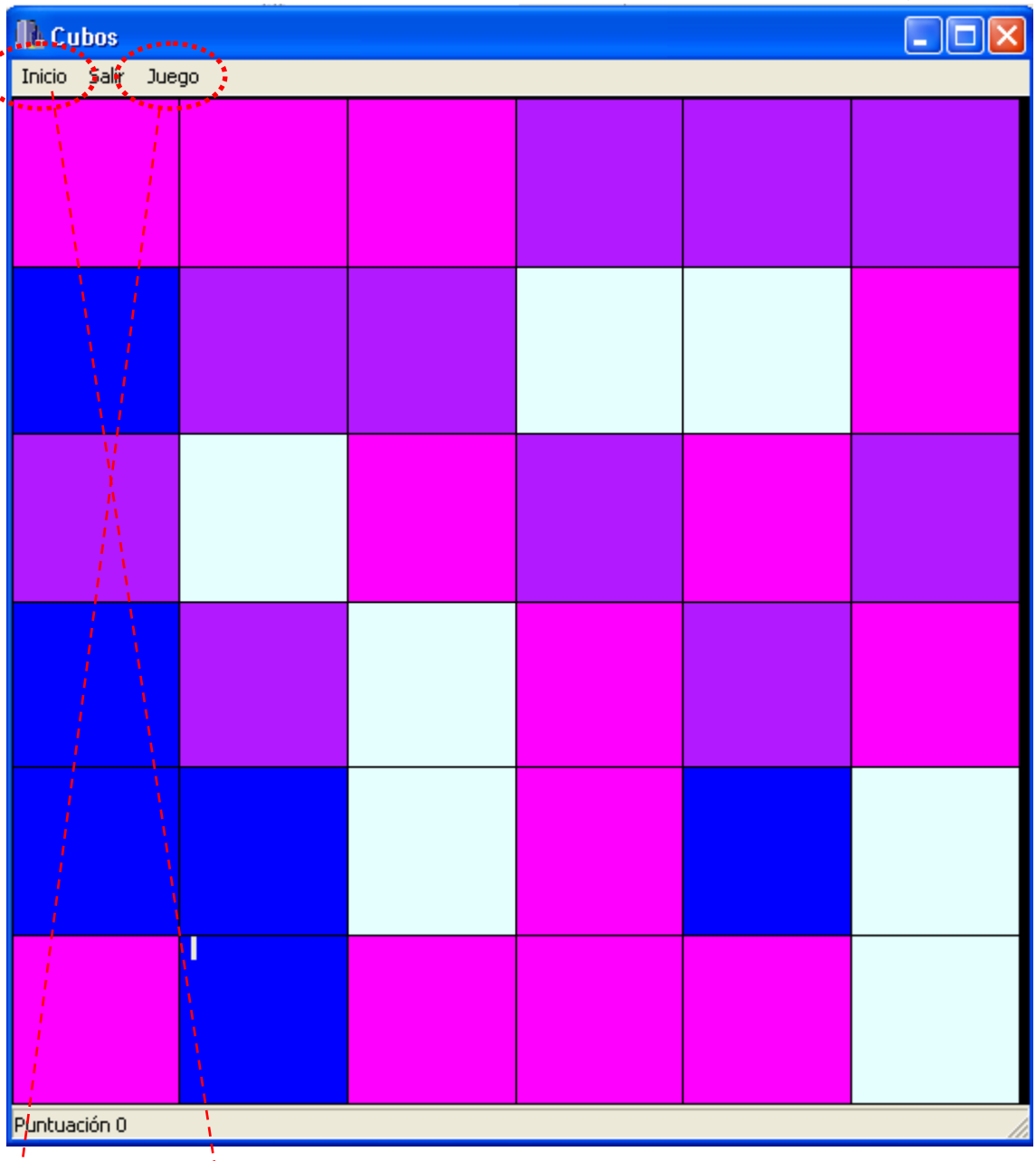
3	7	34	11	27	6	7	34	1	28	15	12	16	2

Individuo Mutado en el sentido contrario

27	16	31	12	32	16	9	0	3	21	1	5	30	29

3.-Presentación de la solución (entorno gráfico).

-Hemos reproducido fielmente el entorno gráfico que suele tener este juego en los diversos medios en que aparece.



-El juego no es más que un tablero de diversos colores. Hemos de resaltar una pequeña peculiaridad y a nuestro modo de entender un gran acierto de la aplicación y es que se ofrece la posibilidad de que el usuario se eche una partida para que así se pueda ver mediante la comparación de resultados la potencia de cálculo del algoritmo evolutivo. Una vez que el usuario ya se ha entretenido y ha dado muestra de su ingenio, es cuando nuestro algoritmo evolutivo entra en juego y busca una solución que el 95% de las veces supera la encontrada por la capacidad humana. También hemos de resaltar que para que la aplicación sea mucho más dinámica y menos engorrosa hemos utilizado los eventos del mouse para que se pueda jugar con el ratón y no sea necesario introducir la casilla elegida mediante el teclado. Por último, decir que para empezar a jugar hay que pulsar sobre *inicio*, esto es mas que nada y sin entrar en mas detalles para que se activen los eventos del mouse. Una vez que el usuario ha acabado su partida, entonces es cuando se ejecuta el algoritmo evolutivo, para ello hemos de pulsar sobre *juego*.

4.Versiones de la aplicación.

-Las versiones de la aplicación podriamos agruparlas a modo groso en 3. Las 1.X, en la que la aplicación se hizo con un entorno gráfico más simple y sin cuidar los detalles de la misma, además no se implemento la reproducción ni la mutación, y aún así los resultados obtenidos siempre y cuando el número de individuos de la población fuera lo suficientemente elevado eran bastante buenos. Las 2.X, en la que ya se implementa la reproducción y la mutación, con lo cual se consigue que aunque el número de individuos de la población no sea sumamente grande, el algoritmo da resultados bastantes buenos, y luego las 3.x , en las que ademas de implementar el elitismo se incluye los eventos del ratón que le hacen a la aplicación mas dinámica y amigable para el usuario.

5.-Tecnologías empleadas.

-Las tecnologías utilizadas han sido básicamente **C++** y **Open GL.** El Open GL es un SW(en forma de librería C) que permite la comunicación entre el programador y el HW de la máquina para el diseño de gráficos.

-En este punto vamos a explicar la interacción entre el **C++** y **Open Gl** de la parte gráfica de nuestra aplicación, sin tampoco pretender ahondar en los conceptos propios de **Open GL.**

-Lo primero que hemos de resaltar que todo lo que se pinta en una escena creada por **Open GL** se hace a través de un método princial que es:

```
void __fastcall TGLForm2D::GLScene(){
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// comandos para dibujar la escena

glFlush();
if(juego !=NULL) {
JUEGO->DRAW();
}
SwapBuffers(hdc);
}
```

-Lo principal de este método es la llamada marcada en rojo. La clase juego dispone de un método ‘draw’ que es el que se encarga de pintar el tablero en la escena, pero esto no es del todo verdad, porque de lo único que se encarga este procedimiento es ahondar un poco mas y acceder hacia capas mas interiores mediante la llamada al método draw de su atributo privado Ladrillos. Este objeto a su vez no es mas que una

matriz de objetos Ladrillo. Es a este nivel de profundidad donde realmente aparecen las llamadas a las librería de Open GL.

-En este punto donde podemos aclarar porque decidimos implementar la clase ladrillos con esta estructura de datos vector<vector<void*> > *matriz y no optamos por lo que a simple vista parece mas trivial, que es implementar una matriz de objetos ladrillo. Como ya hemos comentado arriba, realmente es a ese nivel, ósea al nivel de los objetos que guarda la matriz donde realmente se interactúa con las librerías gráficas de Open GL. De esta forma lo que conseguimos es separar el C++ del Open Gl, con las ventajas que ello conlleva. Hemos de decir que esto se pensó para poder hacer futuras mejoras en la aplicación .Debido a nuestro diseño, y como se ha explicado aquí no sería muy costoso el cambiar el interfaz de la aplicación, por ejemplo en vez de cuadros, podíamos optar por octógonos, triángulos o cualquier otra forma en 2D, tan solo cambiando el tipo de objeto que asignemos a la matriz

-Creemos que no es necesario el entrar en más detalles sobre aspectos del Open Gl. puesto que el proyecto no trata de ello, pero en caso de verlo conveniente se puede acudir al código fuente de la clase *ladrillo* donde aparecen los detalles técnicos mas en profundidad.

6.-Estudio de los parámetros del algoritmo.

-La experiencia de haber realizado numerosísimas ejecuciones del mismo algoritmo, habiendo fijado una semilla para los números aleatorios nos lleva a las siguientes conclusiones:

- La reproducción y la mutación, incluso más esta ultima, son de suma importancia para este juego .

- Como también se comentó con anterioridad en las primeras versiones, no se implementó ni la selección ni la mutación, por lo cual teníamos que crear un número excesivamente alto para que el algoritmo nos devolviera resultados satisfactorios. Una vez implementadas estas, y al disponer de elitismo, nos dimos cuenta que aunque se disponga de una población pequeña, si los porcentajes de reproducción y mutación son elevados, se pueden alcanzar soluciones óptimas. Esto es muy lógico, al disponer de elitismo (por lo que no vamos a perder a los individuos buenos que ya teníamos), los individuos que mutan no son mas que nuevas listas de movimientos que pueden ser buenos o malos.

- A colación de esto, otra cosa a resaltar que si el número inicial de individuos es alto, el algoritmo evolutivo se ejecuta pocas iteraciones,

porque es muy probable que dentro de la población inicial se encuentre el individuo que aporte la solución óptima. Si la población es pequeña, el número de iteraciones crece bastante, pero puede llegar a encontrar la misma solución.

- Los parámetros que hemos utilizado nosotros en este juego que creemos que son los mas adecuados son los siguientes:

Individuos de la población:100 Elitismo: 25 Probabilidad de cruce: 0.35 Probabilidad de mutación: 0.35

- Como podemos observar los porcentajes de mutación y cruce son más elevados de lo normal.

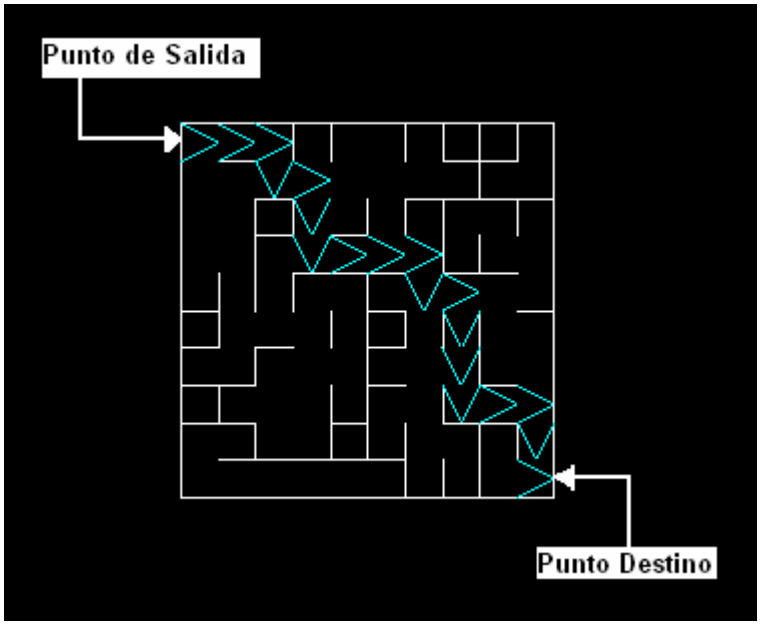
3.4. Juego del Laberinto

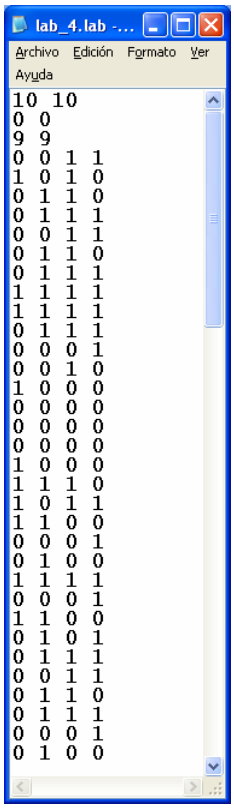
Índice:

6. Estudio de los parámetros del algoritmo.

1. Introducción

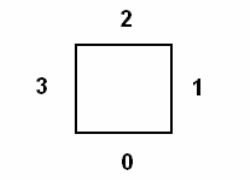
El juego del laberinto consiste en encontrar el camino de mínima longitud desde un punto de salida a otro punto destino. Como es evidente, no se pueden atravesar las paredes del tablero, y siempre deberá haber al menos un camino desde el punto de salida al punto destino. En este juego se ha decidido que el punto de partida sea la esquina superior izquierda, y el punto de llegada en la parte inferior izquierda.



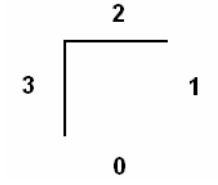


El laberinto se carga de un archivo de tipo texto con un formato especial:

- Los dos primeros números indican las dimensiones del laberinto: número de filas y de columnas.
- Los siguientes dos números indican el punto de partida.
- Los otros dos indican el punto de llegada.
- Por último se listan todas las casillas del laberinto, con el siguiente formato: si hay pared en la posición ***i*** se pone un 0, en caso de que haya pared se pone un 1. La numeración de los cuadrados es la siguiente:

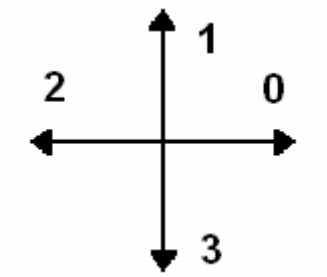


Ejemplo: código de esta casilla: 0 0 1 1.



Lo complicado de generar las casillas que forman el laberinto es que cada una de ellas está relacionada con sus vecinos, es decir, si una casilla tiene pared en la posición 0, entonces la casilla que está debajo de ella también tiene que tener pared en su posición 2. Este aspecto es muy importante ya que sino se podrían atravesar las paredes dependiendo del sentido en el que se afrontasen. En la primera versión de la aplicación el cálculo de dicho laberinto se calculaba a mano, con la dificultad que ello conlleva. En una posterior versión se implementó un programa que genera laberintos aleatorios de forma automática.

Los movimientos sólo pueden ser en sentido de los cuatro puntos cardinales, no pudiéndose realizar movimientos en diagonal, y se representan de la siguiente forma:



2. Aspectos del Algoritmo Evolutivo

a) Representación de la Población

La población está compuesta por:

- Un vector de punteros a individuos. En este vector se almacenan los punteros a los individuos que forman la población. El hecho de usar punteros mejora el rendimiento y velocidad de la aplicación, por el contrario se necesita tener más cuidado con la destrucción de los punteros, para que no se quede memoria ocupada de forma inútil. En el destructor de la población hay que eliminar uno a uno todos los punteros del vector. Este vector es de tipo clase ***Vector***, predefinida en el C++ Builder, y que proporciona un interfaz cómodo y eficaz para almacenar punteros a objetos, aumentando así el rendimiento de la aplicación.

- El número de individuos de la población. Número de individuos que formaran la población.

- Tamaño del elitismo. Este parámetro es esencial para obtener la solución óptima, ya que representa el número de mejores elementos que conservamos en cada iteración, para que no se pierdan dichos individuos.

- Un vector auxiliar de punteros a individuos, donde se almacenan los mejores individuos (población elitista), para que no se pierdan en el proceso de mutación y cruce.

b) Generación de la Población.

La generación de la población se realiza creando cada uno de los individuos y añadiendo el puntero de cada uno de los individuos al vector que representa la población.

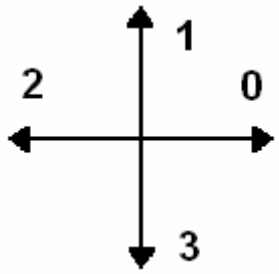
Un aspecto importante es la destrucción de los individuos que forman el vector, es decir, no sólo hay que liberar los punteros del vector, sino que también hay que liberar el objeto que apunta cada uno de los punteros. La liberación de memoria en el C++ Builder es importante estar atento a la destrucción de los punteros que ya no sean útiles, puesto que a diferencia con Java no tiene recopilación automática de memoria no útil.

La recopilación de memoria no usada y el uso de punteros cobra especial relevancia en los algoritmos evolutivos puesto que en ellos se usa una gran cantidad de memoria y velocidad de cómputo, por lo que no conviene desperdiciarlas.

c) Representación de los Individuos.

En este juego cada individuo posee las siguientes características:

- Un *Vector* de enteros con la lista de movimientos. Los movimientos sólo pueden ser en cuatro direcciones, los cuatro puntos cardinales. Estas cuatro direcciones se representan por cuatro números enteros: 0, 1, 2, 3, de la siguiente forma:



- Un puzzle inicial de partida, que ha sido generado aleatoriamente. A partir del puzzle inicial y aplicando la lista de movimientos podemos llegar al puzzle final. El puzzle final es el que representa la solución del algoritmo.

- Una variable ***TPunto,*** que indica el punto de partida del laberinto.

- Una variable ***TPunto,*** que indica el punto de llegada del laberinto.

- Una variable ***TPunto,*** que indica el punto actual donde nos encontramos dentro del laberinto, al aplicar la lista de movimientos al punto inicial.

- Una variable ***bool***, que indica si el individuo está atrapado, es decir, que el único movimiento posible es volver por el camino por el que se llegó.

- Una variable ***TLaberinto***, que representa la matriz de casillas que forman el laberinto.

- Una variable **float**, que representa la puntuación del individuo.

- Una variable **float**, que representa la puntuación acumulada del individuo.

d) Generación de los Individuos.

Lo primero en la generación del individuo es calcular el número máximo de movimientos que van a realizar. Este número se calcula al azar, habiendo un valor máximo y otro mínimo. Este valor se calcula en función de las dimensiones del laberinto de la siguiente forma:

```
int num_mov=((rand()%2)+2)*(lab->get_num_fil()+lab->get_num_col());
```

Una vez calculado el número máximo de movimientos hay que generar la lista de movimientos dentro del laberinto. Estos movimientos se calculan de forma aleatoria, comprobándose que son correcto, y en caso de no serlo se calcula otro .Cada movimiento se representa con un número entre 0 y 3, dependiendo de la dirección del mismo.

A la vez que se generan movimientos hay que ir avanzando a lo largo del laberinto. Avanzar por el laberinto consiste en ir actualizando la variable *punto_actua*l, con el movimiento seleccionado. No se permite volver para atrás y existe la posibilidad de quedarse atrapado, con lo cual se finaliza el proceso de generación de movimientos y se indica que el individuo ha quedado atrapado. Otra posibilidad por la cual se termine la generación de movimientos, antes de llegar al número máximo de movimientos, es por haber llagado ya ha la salida, con lo que hemos encontrado una solución.

Durante este proceso se guarda el movimiento anterior para impedir que se vulva hacia atrás y se pueda generar un bucles que no conducen a la solución del problema.

El código que implementa este proceso es el siguiente:

```
TIndividuo::TIndividuo(TLaberinto *_lab){
 lab=_lab;
 punt_salida=lab->get_punt_salida();
 punt_llegada=lab->get_punt_llegada();
 punt_act=new TPunto(punt_salida);
 atrapado=false;
 int num_mov=((rand()%2)+2)*(lab->get_num_fil()+lab->get_num_col());
 int direc_ant=-1;
 bool seguir=true;
 int i=0;
 int dir;
 list_mov=new vector <int>;
 while ((seguir)&&( i<num_mov)) {
   dir=lab->cal_mov(punt_act,direc_ant);
   direc_ant=dir;
   if (dir==-1) {
    seguir=false;
    atrapado=true;
   }
   else {
      punt_act->mover_punto(dir);
      list_mov->push_back(dir);
      i++;
    }
   if ((punt_act->get_x()==punt_llegada->get_x())&&
        (punt_act->get_y()==punt_llegada->get_y()))
     seguir=false;
   }
 _punt=0;
 _punt_acu=0;
 adaptacion();
}
```

A cada individuo se le asigna un puntero a una variable laberinto para poder recorrerlo y saber la posición exacta en la que se encuentra el punto actual. Además se crean las variables que indican el punto de salida y de llegada del laberinto, y a medida

que se van generando los movimientos se va actualizando el punto actual, a partir del punto de salida.

Hay que tener en cuenta que si partimos del punto inicial y aplicamos la lista de movimientos, en cualquier momento, podemos calcular el punto actual. Aunque dicho punto actual está guardado en una variable para facilitar los cálculos.

Por ultimo habría que calcular la aptitud del individuo mediante la función de adaptación, evaluando la distancia desde el punto actual a la salida del laberinto.

e) Función de Adaptación

La función de adaptación sirve para calcular la aptitud del individuo. La aptitud nos sirve para comparar lo bueno que es un individuo con respecto a los demás individuos de la población. Mediante este valor podemos saber cual es la mejor solución y en el elitismo poder elegir los mejores individuos.

El cálculo de la aptitud de un individuo se obtiene evaluando la distancia que hay desde el punto actual en que se encuentra, hasta el punto de llegada del laberinto. Esta distancia se mide contando las casillas que separan el punto de salida y el de llegada, de forma vertical y horizontal.

Aparte de este cálculo existen también factores de penalización, que ayudan a evaluar de forma más objetiva los individuos, ya que hay que tener en cuenta que el objetivo del juego no es sólo encontrar un camino desde la salida hasta la llegada, sino que también se trata de encontrar el camino mínimo.

En este problema hay que tratar de minimizar la aptitud, es decir, minimizar la distancia del punto actual al punto de llegada, por lo cual para penalizar hay que sumar una cierta cantidad a la aptitud. Al tener que minimizar la función hay que usar la función ***revisar_aptitud_min***, reajustar la función de aptitud.

Las dos penalizaciones son:

1. Penalización por el número de movimientos, se intenta penalizar las soluciones con mayor número de movimientos.
2. Penalización por quedar atrapado, se intenta penalizar los individuos cuyo punto actual se encuentra atrapado, y el único movimiento posible es volver por donde ha venido. Estas soluciones se desprecian dándoles un valor alto de aptitud.

```
void TIndividuo::adaptacion(){
      int penal_atrap=1000;
      int penal_camino=list_mov->size()*0.1;
      _aptitud=abs(punt_llegada->get_x()-punt_act->get_x())+
             abs(punt_llegada->get_y()-punt_act->get_y());
      _aptitud=penal_camino+_aptitud;
     if(! atrapado)
       _aptitud=_aptitud+penal_atrap;
   }
```

f) Función de Selección

En este juego se ha optado por dar sólo la opción de usar sólo elitismo, ya que las funciones heurísticas están bastante adaptadas al problema, y hacen que los mejores individuos converjan a la solución. Por lo tanto sería desaconsejable usar la otra opción.

3. Usando elitismo. Al usar elitismo se eligen los individuos de la población con mayor aptitud. Ya que la población se ordena de mayor a menor en función de la aptitud sólo hay que seccionar los primeros de la población. En el caso del elitismo los individuos seleccionados se duplican para evitar que se pierdan en la reproducción o en la mutación. Este proceso nos garantiza que no se pierdan los que tienden a parecer ser los mejores individuos, pero nos puede hacer caer en máximos locales. Esto se evita teniendo una población amplia y un porcentaje de cruce y mutación óptimo. El elitismo, como es obvio, ralentiza la aplicación, pero los beneficios obtenidos son muy notables, y en muchos problemas es muy difícil obtener la solución óptima sin utilizar esta técnica.

g) Función de Reproducción

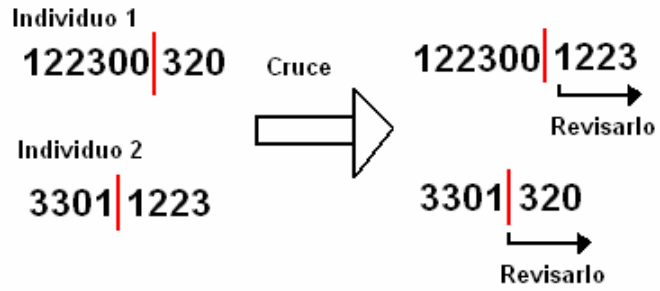
La función de reproducción consiste en elegir dos individuos de la población para ejecutar el cruce de los dos individuos elegidos. La proporción de individuos de la población que se van a reproducir viene en función de un parámetro del algoritmo evolutivo. Este parámetro es el ***porcentaje de cruce***, este factor es esencial para el correcto funcionamiento del algoritmo, puesto que si se cruza poco puede ser que no converjamos a la solución óptima, y si este valor es muy alto se pueden perder soluciones buenas y se consumiría mucho tiempo de CPU, puesto que la después de realizar la reproducción hay que volver a calcular la aptitud de cada uno de los dos individuos, y como hemos comentado anteriormente, resuelta un proceso costoso en tiempo.

Para implementar la reproducción se recorre la población eligiendo los individuos a cruzar. La elección se realizar al azar en función del porcentaje de cruce. Una vez elegidos los individuos cruzar, se cruzan de dos en dos, hay que comprobar que el individuo resultante es correcto, cumpliendo las reglas del juego, es decir que los nuevos movimientos sean correctos, en caso contrario hay que reajustar la lista de movimientos. Una vez realizado este proceso hay que volver a calcular la aptitud del individuo, puesto que el punto actual habrá cambiado.

- **Función de Cruce**

La función de cruce consiste en elegir un punto de corte en el vector de movimientos de cada individuo e intercambiar los movimientos a partir de dicho punto. Este punto se elige al azar dentro del rango posible de los respectivos vectores. Una vez intercambiados los movimientos hay que proceder al reajuste de dichos movimientos desde el punto de cruce, puesto que puede ser que muchos de los nuevos movimientos introducidos no sean correctos o sean repetidos.

Para reajustar la lista de movimientos hay que empezar partiendo desde el punto de salida e irle aplicando cada uno de los movimientos, actualizando el punto actual. En caso de que un movimiento sea incorrecto, se calcula un nuevo movimiento posible. Puede darse el caso de que se encuentre el punto de salida o que acabemos encerrados, con lo que se eliminan el resto de los movimientos, ya que son innecesarios.



h) Una vez realizada la revisión y corrección de los dos individuos, se procede a recalcular la aptitud de cada uno de ellos, ya que el punto actual habrá cambiado.

i) Función de Mutación

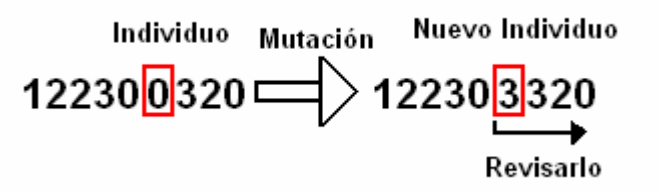
La función de mutación consiste en elegir un punto de la lista de movimientos, y cambiar dicho movimiento por otro posible. La decisión de mutar un individuo depende de un factor del algoritmo evolutivo llamado ***porcentaje de mutación***, y suele tener un valor muy bajo.

El punto de mutación de la lista de movimientos se elige al azar entre la longitud del vector. Al igual que ocurre en el cruce, una vez modificado el vector de movimientos hay que revisar a partir del nuevo movimiento para comprobar que todos los movimientos son correctos y en caso de alguno no lo sea se reemplaza por uno movimiento posible. Además hay que ir actualizando el punto actual dentro del laberinto, con la lista de movimientos, a partir del punto de partida. Si se consigue llegar a la salida o se queda encerrado se elimina los movimientos restantes.

El código que implementa el reajuste de la lista de movimientos de un individuo es el siguiente:

```
void TIndividuo::reajustar(){
  int i,j,dir;
   delete punt_act;
   bool seguir=true;
   punt_act=new TPunto(punt_salida);
   i=0;
   while ((seguir)&&(i<list_mov->size())) {
     if ( ! lab->mov_posible(punt_act,list_mov->at(i)))
       dir=lab->cal_mov(punt_act,dir);
     else
       dir=list_mov->at(i);
     if (dir==-1) {
       seguir=false;
       atrapado=true;
        }
     else {
        punt_act->mover_punto(dir);
        list_mov->at(i)=dir;
        i++;
       }
     if ((punt_act->get_x()==punt_llegada->get_x())&&
         (punt_act->get_y()==punt_llegada->get_y()))
        seguir=false;
  }
  int longit=list_mov->size();
  for(j=i;j<longit;j++)
    list_mov->erase(&list_mov->at(list_mov->size()-1));
}
```

Además, una vez acabado este proceso hay que realizar el cálculo de la aptitud del individuo, por medio de la adaptación, puesto que el punto actual ha cambiado, y su distancia con el punto de salida.



i) Función para Revisar la Aptitud

En los problemas en los que hay que minimizar el valor de la aptitud hay que usar la función de revisar la aptitud mínima después de calcular la adaptación. Esta función consiste en invertir la aptitud de los individuos, es decir, en el caso de funciones de adaptación en las que se trata de minimizar la aptitud los mejores individuos son los que menor aptitud tienen, por lo que es necesario invertir este valor dándoles la mayor aptitud a los mejores individuos y la menor a los peores.

La implementación de esta función consiste en recorrer toda la población y elegir la aptitud máxima, este valor lo llamaremos CMax. Posteriormente volveremos a recorrer la población actualizando la aptitud de cada individuo con:
`CMax-poblacion.at(i)->obtenerAptitud().`

El código de la función sería el siguiente:

```
CMax=-MAXLONG;
int i;
for (i=0;i<poblacion.size();i++)
   poblacion.at(i)->adaptacion();
for (i=0;i<(poblacion.size());i++)
  if (poblacion.at(i)->obtenerAptitud()>CMax)
    CMax= poblacion.at(i)->obtenerAptitud();
for (i=0;i<(_poblacion.size());i++)
  poblacion.at(i)->act_aptitud(CMax-poblacion.at(i)>obtenerAptitud());
```

j) Elitismo

El elitismo es un factor esencial en los algoritmos evolutivos y consiste en preservar a lo largo de las sucesivas iteraciones los individuos mejores. Se considera que los mejores individuos son aquellos que tiene mayor aptitud, que se calcula gracias a la función de adaptación. Sin el elitismo es muy difícil conseguir buenas soluciones, lo único del elitismo es que hay que usar mucha más memoria y tiempo de cómputo.

Preservar los mejores individuos en cada iteración favorece que ciertas características buenas, que han aparecido en algunos individuos se preserven e incluso puedan evolucionar para obtener mejores individuos.

La implementación del elitismo se realiza en varios pasos.

- Primer Paso: consiste en calcular la adaptación de todos los miembros de la población, para así poder saber cual son los mejores.

- Segundo Paso: consiste en ordenar la población en función de la aptitud, quedando los individuos con mayor aptitud al principio de la población.

- Tercer Paso: consiste en seleccionar "n" de los primeros individuos de la población, que son los mejores ya que la población ha sido ordenada en función de la aptitud. El número "n" se conoce como tamaño de la población elitista, y es otro parámetro importante en los algoritmos evolutivos. Si el tamaño de la población elitista es elevado el tiempo de ejecución se dispara, por el contrario si es pequeño podemos desperdiciar individuos con características muy interesantes. La selección de estos "n" individuos consiste en crear una copia de ellos y almacenarlos en vector auxiliar para que no se vean afectados por la mutación y por el cruce.

- Cuarto Paso: una vez realizada la mutación y la reproducción se fusionan las dos poblaciones, la que resulta de la mutación y la reproducción, y la población auxiliar elitista.

- Quinto Paso: después de fusionar las dos poblaciones se ordena la población resultante en función de la aptitud, y se eliminan los peores individuos, para que la población tenga el número de individuos especificados.

K) Código adicional desarrollado para el programa.

Para poder desarrollar el juego ha sido necesaria la creación de una clase auxiliar para representar el laberinto. El laberinto contiene una matriz de casillas, donde cada casilla está representada por cuatro paredes, donde cada una de las cuales puede o no estar presente.

Clase ***TLaberinto***: el laberinto se carga desde un archivo de texto con un formato especial, creado por nosotros mismos, y a partir de dicho archivo se rellenan las dimensiones del laberinto, el punto de salida, el punto de llaga y todas las casillas del laberinto.

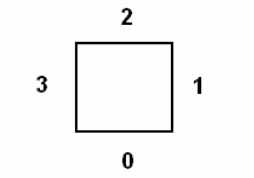
Variables de la clase ***TLaberinto***:

- Una matriz de punteros a cuadrados, que representa la estructura del laberinto.
- Una ***int*** que representa el número de filas.
- Una ***int*** que representa el número de filas.
- El punto de salida del laberinto.
- El punto de llegada del laberinto.

Métodos de la clase ***TLaberinto***:

- `TLaberinto(String ruta);`: constructor del laberinto a partir de un archivo.
- `void draw();`: dibuja los cuadrados del laberinto.
- `int get_num_fil();`: devuelve el número de filas del laberinto
- `int get_num_col();`:devuelve el número de columnas del laberinto
- `int cal_mov(TPunto *punt_act,int direc_ant);`: calcula un posible movimiento, a partir del punto actual y la dirección anterior.
- `TPunto* get_punt_salida();`: devuelve el punto de partida del laberinto.
- `TPunto* get_punt_llegada();`:devuelve el punto de llegada del laberinto.
- `void dibujar_lista(vector<int> *get_list);`: dibuja la lista de movimientos.
- `void draw2(TPunto* punt,TColor color);`: dibuja un aspa en el laberinto en el punto indicado, con el color especificado.
- `bool mov_posible(TPunto* punt,int direc);`: indica si un movimiento es posible.
- `void draw2(TPunto* punt,int direc,TColor color);` dibuja una flecha en el laberinto en el punto indicado y con el color especificado. La dirección nos sirve para saber la dirección de la flecha.
- `int get_long_lado();`: devuelve la longitud del lado de cada cuadrado
- `~TLaberinto();`: destructor del laberinto.

Clase ***cuadrado***: esta clase representa cada una de las casillas del laberinto. Cada cuadrado dispone de cuatro lados, los cuales pueden estar ocupados o no, para representar si hay lado o no se usa un vector de ***bool*** y el cuadrado se representa de la siguiente forma:



Variables de la clase ***cuadrado***:

- `TPunto *centro;`: indica el centro del cuadrado.
- `int long_lado;`: longitud de cada uno de los lados del cuadrado.
- `bool* lados_ocupados;`: lista de lados ocupados.
- `TPunto* list_puntos;`: lista con los cuatro punto que forman el cuadrado.

Métodos de la clase ***cuadrado***:

- `cuadrado(TPunto *cent, int _long_lado, bool* _lados_ocupados);`: constructor del cuadrado, teniendo como parámetros el centro del cuadrado, la longitud de los lados y los lados ocupados.
- `void draw(int tipo,TColor color_linea);`:en caso de *tipo*=0 dibuja el cuadrado con los lados ocupados, y en caso de *tipo*=1, dibuja un aspa en el cuadrado, con el color indicado.
- `bool get_lado(int i);`: indica si un lado está ocupado.
- `void draw_2(int direc, TColor color_linea);`: dibuja una flecha en el cuadrado indicado, con una dirección especificada.
- `int get_long_lado();`: devuelve la longitud de de un lado.
- `~cuadrado();`: destructor del cuadrado.

Clase ***TPunto***: representa las dos componentes de un punto: ***x*** e ***y***.

Variables de la clase ***TPunto***:

- `GLint x;`: componente x del punto.
- `GLint y;`: componente y del punto.

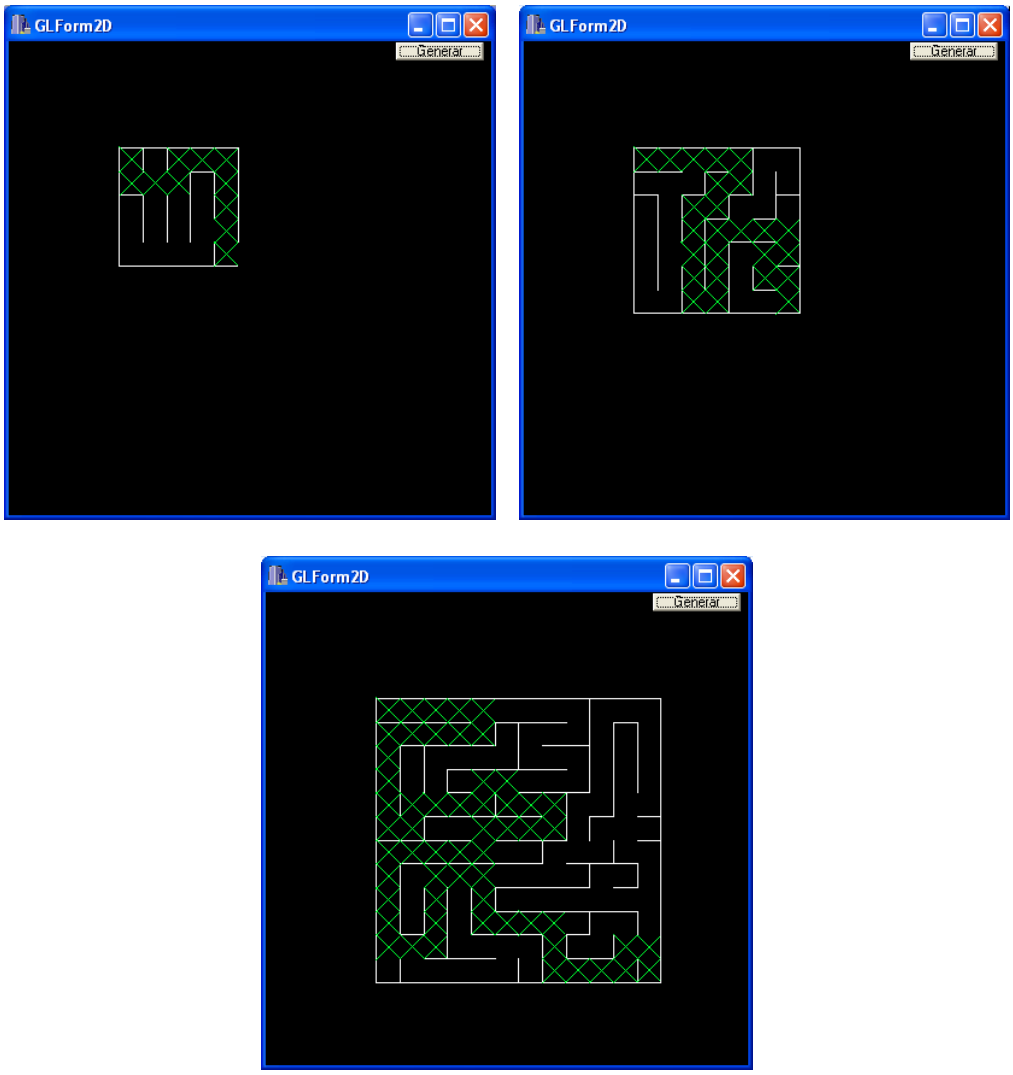
Métodos de la clase ***TPunto***:

- `TPunto();`: constructor por defecto.
- `TPunto(GLint x, GLint y);`: constructor a partir de las componentes x e y.
- `TPunto(TPunto *p),`: constructor a partir de otro punto.
- `GLint get_x();`: devuelve la componente x del punto.
- `GLint get_y();`:devuelve la componente y del punto.
- `void draw();` : método que dibuja un punto.
- `void set_punto(GLint x, GLint y);`: actualiza las componentes x e y del punto.
- `void mover_punto(int direc);`: método que hace mover una posición el punto, en la dirección indicada.
- `int cal_mov (TPunto *punto);`: a partir de un punto y comparándolo con el punto actual calcula cual ha sido la dirección para llegar al punto actual.

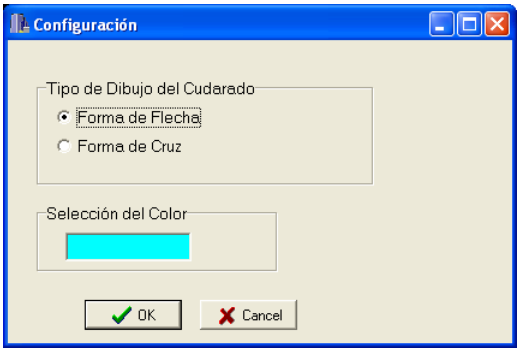
3. Presentación de la solución (entorno gráfico).

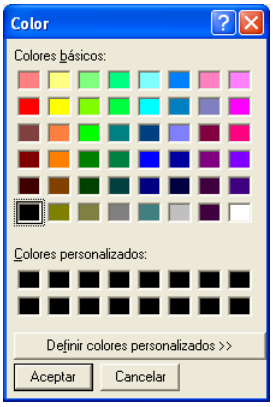
Para representar el entorno gráfico hemos optado por la utilización de las librerías que proporciona el ***OpenGL***, para poder mostrar el laberinto y los movimientos que condicen desde el punto de partida al punto destino. Sobre el uso del ***OpenGL***, hablaremos en el apartado de *Tecnologías Empleadas*.

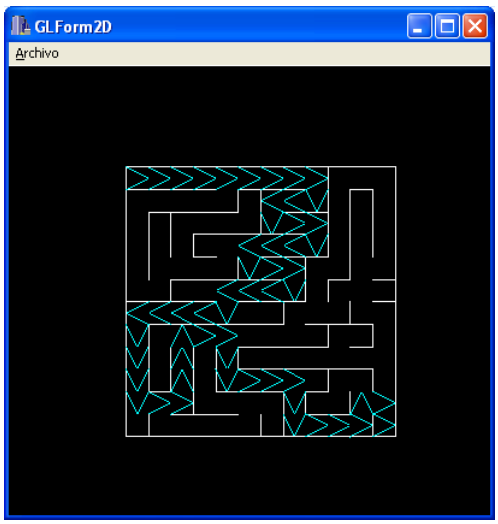
En una primera versión se ofrecía la posibilidad de abrir un laberinto, creado a mano, a partir de un archivo de texto con un formato especial, y generar la solución correcta. Para esta versión existían tres tipos de laberintos, dependiendo de la dificultad.

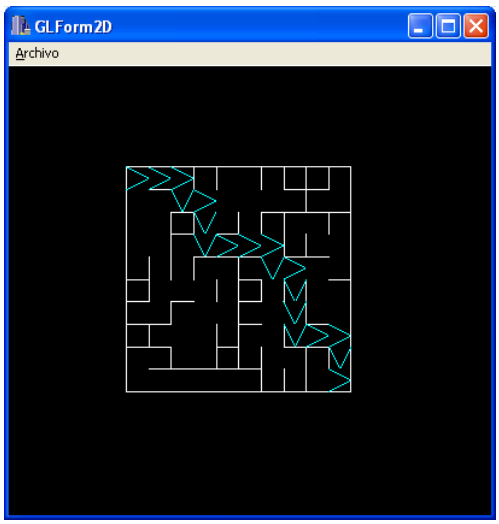


En una versión posterior se ofrece la posibilidad de configurar el color con el que se pintan los movimientos, y también se puede elegir entre representar los movimientos cómo flechas o cómo aspas. Además se implementó un programa adicional que genera de forma aleatoria laberintos, con lo que se pudieron hacer muchas más pruebas con otros laberintos.


Configuración
Tipo de Dibujo del Cudarado
Forma de Flecha
Forma de Cruz
Selección del Color
OK
Cancel


Color
Colores básicos:
Colores personalizados:
Definir colores personalizados >>
Aceptar
Cancelar


GLForm2D
Archivo

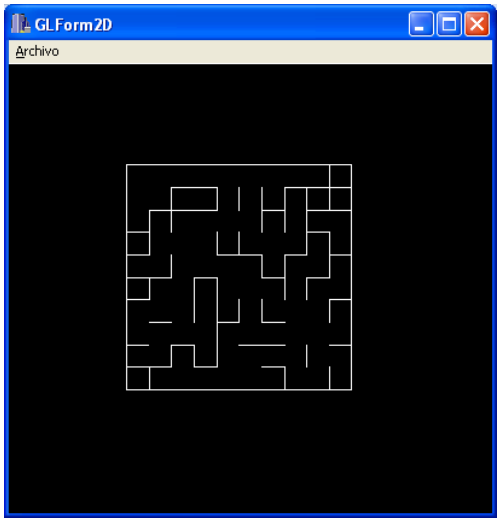

GLForm2D
Archivo

4. Versiones de la aplicación

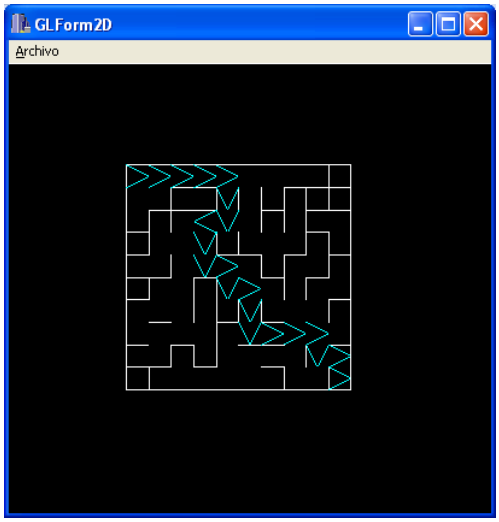
Además de las dos versiones hechas para mejorar el entorno gráfico, se ha desarrollado una ampliación del juego que genera laberintos de forma aleatoria, y los guarda en un archivo de tipo texto con el formato apropiado.

Como ya hemos comentado antes, la creación de los laberintos no resulta trivial, puesto que las casillas adyacentes comparten lados y tienen que tener los mismos valores en los lados que comparten.

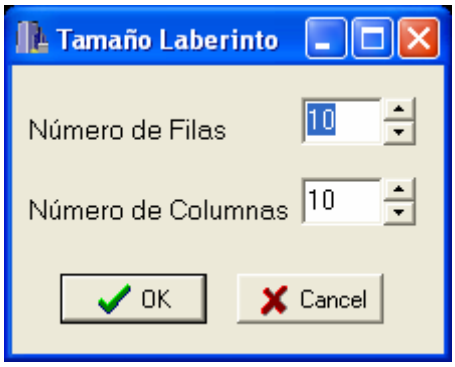
La idea inicial de esta ampliación era generar una población de laberintos, aplicarles el proceso evolutivo, y elegir aquel o aquellos que tuvieran salida. Pero el cálculo de la aptitud de cada laberinto resultó muy complicado y se optó por dejar una versión que simplemente genera el laberinto. Lo bueno de esta versión es que nuestra el resultado del laberinto de forma gráfica, así que de forma intuitiva podemos ver si tiene solución; y además se puede configurar el número de filas y de columnas del laberinto.



Laberinto generado de forma automática, y como se puede ver tiene solución.



Solución encontrada por el juego.



5. Tecnologías empleadas

Una de las tecnologías más importantes que han sido usadas en la implementación del juego el uso de las librerías del ***OpenGL***. El ***OpenGL*** proporciona métodos para poder pintar cualquier figura, tanto en 3 y en 2 dimensiones. En nuestro caso sólo hemos usado las 2 dimensiones.

Lo primero que hay que hacer es crear el marco donde se pintará:

```
//---------------------------------------------------------------------------
void __fastcall TGLForm2D::FormCreate(TObject *Sender)
{
    // inicialización de las variables del programa
    time_t t;
    srand((unsigned) time(&t));
    hdc = GetDC(Handle);
    SetPixelFormatDescriptor();
    hrc = wglCreateContext(hdc);
    if(hrc == NULL)
       ShowMessage(":-)~ hrc == NULL");
    if(wglMakeCurrent(hdc, hrc) == false)
       ShowMessage("Could not MakeCurrent");
    w = ClientWidth;
    h = ClientHeight;
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    color= clAqua;
}
//---------------------------------------------------------------------------
void __fastcall TGLForm2D::SetPixelFormatDescriptor()
{
    PIXELFORMATDESCRIPTOR pfd = {
       sizeof(PIXELFORMATDESCRIPTOR),
         1,
         PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER,
         PFD_TYPE_RGBA,
         24,
         0,0,0,0,0,0,
         0,0,
         0,0,0,0,0,
         32,
         0,
         0,
         PFD_MAIN_PLANE,
         0,
         0,0,0
    };
    int PixelFormat = ChoosePixelFormat(hdc, &pfd);
    SetPixelFormat(hdc, PixelFormat, &pfd);
}
//---------------------------------------------------------------------------
void __fastcall TGLForm2D::FormResize(TObject *Sender)
{
    GLfloat nRange = 200.0;
    w = ClientWidth;
    h = ClientHeight;
    if (h==0) h=1;
    if (w==0) w=1;
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        {
         gluOrtho2D (-nRange, nRange, -nRange*h/w, nRange*h/w);
         XLeft=-nRange;
```

```
         XRight=nRange;
         YLeft=-nRange*h/w;
         YRight=nRange*h/w;
        }
    else
       {
         gluOrtho2D (-nRange*w/h, nRange*w/h, -nRange, nRange);
         XLeft=-nRange*w/h;
         XRight=nRange*w/h;
         YLeft=-nRange;
         YRight=nRange;
        }
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    GLScene();
}
```

Cada vez que queramos pintar el laberinto y los movimientos, invocaremos al método: **GLScene()**. Cuyo código es:

```
void __fastcall TGLForm2D::GLScene()
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// comandos para dibujar la escena
if (laberinto!=NULL)
 laberinto->draw();
if( ind!=NULL) {
// laberinto->dibujar_lista(ind->get_list_mov());
 if (opcion_dib==0){
    int direc=1;
    TPunto *anterior=NULL;
    TPunto *punt=new TPunto(laberinto->get_punt_salida());
    for (int i=0;i< ind->get_list_mov()->size();i++){
      if(anterior!=NULL) {
        direc=punt->cal_mov(anterior);
        delete anterior;
      }
    laberinto->draw2(punt,direc,color);
    anterior= new TPunto(punt);
     punt->mover_punto(ind->get_list_mov()->at(i));
     glFlush();
     SwapBuffers(hdc);
    Sleep(150);
    }
   laberinto->draw2(punt,1,color);
   Sleep(150);
 }
 else if (opcion_dib==1){
    TPunto *punt=new TPunto(laberinto->get_punt_salida());
    for (int i=0;i< ind->get_list_mov()->size();i++){
     laberinto->draw2(punt,color);
     punt->mover_punto(ind->get_list_mov()->at(i));
     glFlush();
     SwapBuffers(hdc);
     Sleep(150);
    }
  laberinto->draw2(punt,color);
   Sleep(150);
  }
 }
glFlush();
SwapBuffers(hdc);
}
```

Para dibujar las casillas del laberinto:

```
void cuadrado::draw(int tipo,TColor color_linea) {
 if(tipo==1){
  //pinta las paredes de la casilla
  glColor3f(1.0,1.0,1.0);
  if (lados_ocupados[0]){
    glBegin(GL_LINES);
     glVertex2i(list_puntos[0].get_x(),list_puntos[0].get_y());
     glVertex2i(list_puntos[1].get_x(),list_puntos[1].get_y());
    glEnd();
   }
  if (lados_ocupados[1]){
    glBegin(GL_LINES);
     glVertex2i(list_puntos[1].get_x(),list_puntos[1].get_y());
     glVertex2i(list_puntos[2].get_x(),list_puntos[2].get_y());
    glEnd();
   }
  if (lados_ocupados[2]){
    glBegin(GL_LINES);
     glVertex2i(list_puntos[2].get_x(),list_puntos[2].get_y());
     glVertex2i(list_puntos[3].get_x(),list_puntos[3].get_y());
    glEnd();
   }
  if (lados_ocupados[3]){
    glBegin(GL_LINES);
     glVertex2i(list_puntos[3].get_x(),list_puntos[3].get_y());
     glVertex2i(list_puntos[0].get_x(),list_puntos[0].get_y());
    glEnd();
   }
 }
 else{
  //pinta un aspa dentro de la casilla
  int color= color_linea;
  float f_rojo= (color % 256)/256.0;
  color=(color/256);
  float f_verde=(color % 256)/256.0;
  color = (color/256);
  float f_azul=(color%256)/256.0;
  glColor3f(f_rojo, f_verde, f_azul);
     glBegin(GL_LINES);
       glVertex2i(list_puntos[0].get_x(),list_puntos[0].get_y());
       glVertex2i(list_puntos[2].get_x(),list_puntos[2].get_y());

      glEnd();
    glBegin(GL_LINES);
      glVertex2i(list_puntos[3].get_x(),list_puntos[3].get_y());
      glVertex2i(list_puntos[1].get_x(),list_puntos[1].get_y());
     glEnd();

  }
}
```

Para poder dibujar una flecha:

```
void cuadrado::draw_2(int direc,TColor color_linea) {
 //  dibuja una flecha en la direccion indicada
 //  con la dirección indicada
 TPunto* nuevo,vertic1,vertic2;
  int color= color_linea;
  float f_rojo= (color % 256)/256.0;
  color=(color/256);
  float f_verde=(color % 256)/256.0;
  color = (color/256);
  float f_azul=(color%256)/256.0;
```

```
  glColor3f(f_rojo, f_verde, f_azul);
 switch(direc){
  case 0 :
   nuevo = new TPunto( list_puntos[0].get_x()+(long_lado/2),
                                                 list_puntos[0].get_y());
    vertic1= list_puntos[3];
    vertic2= list_puntos[2];
    glBegin(GL_LINES);
      glVertex2i(  vertic1.get_x(), vertic1.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();

    glBegin(GL_LINES);
      glVertex2i(  vertic2.get_x(), vertic2.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();
    break;

  case 1 :
   nuevo = new TPunto( list_puntos[2].get_x(),

list_puntos[1].get_y()+(long_lado/2));
    vertic1= list_puntos[3];
    vertic2= list_puntos[0];
    glBegin(GL_LINES);
      glVertex2i(  vertic1.get_x(), vertic1.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();
    glBegin(GL_LINES);
      glVertex2i(  vertic2.get_x(), vertic2.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();
    break;
   case 2 :
    nuevo = new TPunto( list_puntos[3].get_x()+(long_lado/2),
                                                 list_puntos[3].get_y());
    vertic1= list_puntos[0];
    vertic2= list_puntos[1];
    glBegin(GL_LINES);
      glVertex2i(  vertic1.get_x(), vertic1.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();

    glBegin(GL_LINES);
      glVertex2i(  vertic2.get_x(), vertic2.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();
    break;
   case 3 :
    nuevo = new TPunto( list_puntos[3].get_x(),

list_puntos[0].get_y()+(long_lado/2));
    vertic1= list_puntos[2];
    vertic2= list_puntos[1];
    glBegin(GL_LINES);
      glVertex2i(  vertic1.get_x(), vertic1.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();

    glBegin(GL_LINES);
      glVertex2i(  vertic2.get_x(), vertic2.get_y());
      glVertex2i(  nuevo->get_x(), nuevo->get_y());
    glEnd();
    break;
 }
}
```

6. Estudio de los parámetros del algoritmo

El estudio de los parámetros en un algoritmo evolutivo es un aspecto esencial, ya que aunque el algoritmo evolutivo este bien implementado puede ser que una mala elección de dichos parámetros haga que se encuentren malas soluciones o que el tiempo de cómputo sea excesivo.

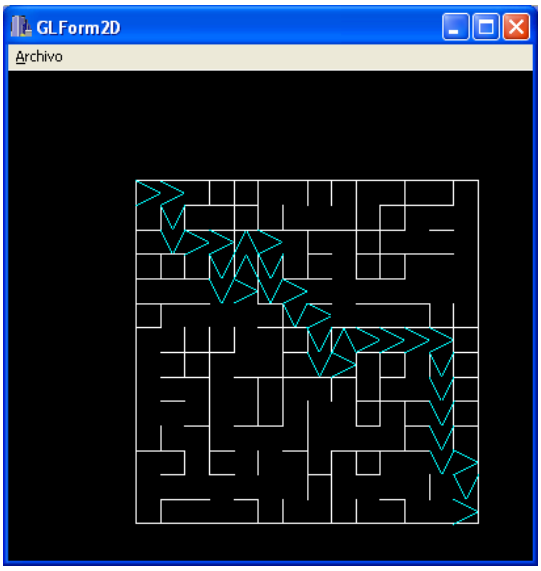
Los parámetros de un algoritmo evolutivo son muy concretos:

- **Tamaño de la población**, parámetro que indica el número de individuos que va a poseer la población. Si el tamaño es excesivo se ralentiza mucho la ejecución del algoritmo, pero si es demasiado pequeño no se generan suficientes individuos para encontrar la solución óptima.
- **Número de iteraciones**, número de veces que se ejecuta el algoritmo evolutivo. Muchas veces se puede parar si vemos que ya hemos encontrado la solución óptima, con lo que ahorramos tiempo de cómputo.
- **Porcentaje de cruce**, indica que porcentaje de la población se va a cruzar. Este factor es muy importante porque el cruce suele consumir mucho tiempo de cómputo y si se cruza muchos individuos se ralentiza el algoritmo y además podríamos perder alguna solución óptima, en el caso de no usar elitismo.
- **Porcentaje de mutación**, indica el porcentaje de individuos de la población que se van a mutar y es un factor menos determinante que el de cruce, ya que suele requerir menos tiempo de CPU, pero también tiene su importancia en la medida en que cada vez que se produce una mutación, hay que recalcular la aptitud, como en el caso del cruce, y el cálculo de la aptitud puede resultar muy costoso.
- **Tamaño de la población elitista**, indica el número de individuos que se conservan en cada iteración, si el número es muy elevado se produce un exceso de uso de memoria, al guardar temporalmente los individuos mejores, y de CPU al tener que fusionar y reordenar la población para elegir los mejores.

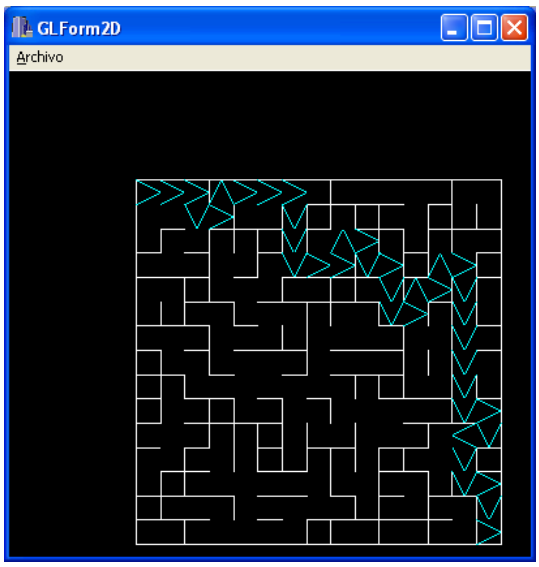
El tamaño de los parámetros del algoritmo evolutivo viene influenciado por el tamaño del laberinto, a mayor tamaño del laberinto mayor número de individuos y de iteraciones.

Para laberintos de dimensiones considerables, (14x14) o (15x15), se consigue encontrar el camino óptimo con los siguientes parámetros:

- **Tamaño de la población**: 1500 individuos.
- **Número de iteraciones**: 50 iteraciones.
- **Porcentaje de cruce**: 25 %
- **Porcentaje de mutación:** 8 %
- **Tamaño de la población elitista:** 140 individuos.



Puzzle de 14x14.



Puzzle de 15x15.

3.5. Juego de los Cuadrados

Índice:

6. Estudio de los parámetros del algoritmo.

1. Introducción

El juego de las letras consiste en dadas 9 letras combinarlas para encontrar la palabra que tenga el máximo número de letras y que tenga significado. Se puede elegir el número de vocales y de consonantes que se usarán para formar la palabra. Como en el cado de las cifras, las letras se eligen al azar en función del número de vocales y de consonantes elegidas. Esta claro que si elegimos un número muy elevado de cualquiera de los dos tipos de letras nos resultará muy difícil encontrar una palabra larga, es decir si elegimos siete vocales y 2 consonantes es imposible encontrar una palabra de 9 letras. En el juego no se pueden repetir las letras que se escogen para formar la palabra, es decir, sólo se puede escoger una vez cada letra generada. Se pueden generar letras repetidas ya que el proceso de generación de las letras es aleatorio.

El idioma elegido para implementar el juego ha sido el español, que posee cinco vocales y veintidós consonantes, en total veintisiete letras para elegir. Dichas letras se extraen de forma aleatoria, teniendo todas las mismas posibilidades de ser elegidas. Cómo es obvio para comprobar que una palabra es correcta no se tiene en cuenta las tildes, ya que cuando se generan aleatoriamente las vocales, estas en ningún caso llevan tilde.

El juego permite que las letras a utilizar para generar la palabra sean introducidas por el usuario, pudiéndose así compara los resultados con otras implementaciones del juego o repetir búsquedas.

Es obvio que no sólo vale con elegir las letras apropiadas, sino que también es importante su ordenación para dar lugar a una palabra con significado. Para comprobar que una palabra existe, es decir, que tiene significado en la lengua española, se busca dicha palabra en una base de datos. Esta base de datos contiene 295938 palabras en español, y esta guarda en mySQL para agilizar el acceso, ya que a lo largo del juego hay que acceder en muchas ocasiones. Además de la base de datos de palabras en español, tenemos muchos más idiomas, como son: inglés, catalán, italiano, francés y alemán, con lo que se podría adaptar el juego a otros idiomas. Lo único que habría que hacer es cambiar las letras del alfabeto, dependiendo del idioma elegido.

Una funcionalidad adicional del juego es la posibilidad de comprobar si una palabra está en el diccionario, es decir, si encuentra en la base de datos. Esta funcionalidad es muy interesante para saber si se ha conseguido encontrar una palabra de mayor longitud que la obtenida por el algoritmo evolutivo.

2. Aspectos del Algoritmo Evolutivo

a) Representación de la Población

La población está compuesta por:

- Un vector de punteros a individuos. En este vector se almacenan los punteros a los individuos que forman la población. El hecho de usar punteros mejora el rendimiento y velocidad de la aplicación, por el contrario se necesita tener más cuidado con la destrucción de los punteros, para que no se quede memoria ocupada de forma inútil. En el destructor de la población hay que eliminar uno a uno todos los punteros del vector. Este vector es de tipo clase ***Vector***, predefinida en el C++ Builder, y que proporciona un interfaz cómodo y eficaz para almacenar punteros a objetos, aumentando así el rendimiento de la aplicación
- El número de individuos de la población. Número de individuos que formaran la población
- Un puntero a un vector de operandos disponibles para generar la expresión aritmética. Este puntero será añadido a cada individuo que forma la población.
- Tamaño del elitismo. Este parámetro es esencial para obtener la solución óptima, ya que representa el número de mejores elementos que conservamos en cada iteración, para que no se pierdan dichos individuos.

b) Generación de la Población.

La generación de la población se realiza creando cada uno de los individuos y añadiendo el puntero de cada uno de los individuos al vector que representa la población..

Un aspecto importante es la destrucción de los individuos que forman el vector, es decir, no sólo hay que liberar los punteros del vector, sino que también hay que liberar el objeto que apunta cada uno de los punteros. La liberación de memoria en el C++ Builder es importante estar atento a la destrucción de los punteros que ya no sean útiles, puesto que a diferencia con Java no tiene recopilación automática de memoria no útil.

La recopilación de memoria no usada y el uso de punteros cobra especial relevancia en los algoritmos evolutivos puesto que en ellos se usa una gran cantidad de memoria y tiempo de cómputo, por lo que no conviene desperdiciarlas.

c) Representación de los Individuos.

En este juego cada individuo posee las siguientes características:

- Una variable **String** que representa la palabra que hemos formado con las letras seleccionadas para formar la palabra. No se pueden repetir las letras seleccionadas.
- Una variable **float**, que representa la aptitud del individuo, representando lo adaptado que esta el individuo con respecto a los demás individuos de la población.
- Una variable **float**, que representa la puntuación del individuo.
- Una variable **float**, que representa la puntuación acumulada del individuo.
- Una variable **int**, que representa la longitud de la palabra.
- Una variable **String,** que representa las letras con las que podemos formar la palabra
- Un ***Vector*** de ***bool***, que representa las letras que hemos usado en la palabra respecto a la variable anterior. Esta variable es específica para cada individuo.

d) Generación de los Individuos.

En la generación del individuo tenemos que elegir las letras que vamos a elegir para formar la palabra. Lo primero es calcular la longitud de la palabra, que se hace de forma aleatoria, siendo el valor mínimo de la palabra 3 y el máximo 9. Se acota el valor de la palabra puesto que palabras de poca longitud tienen poco interés y también porque al haber solo 9 letras para elegir y no pudiéndose repetir letras es imposible generar palabras con más de 9 letras.

Una vez elegida la longitud de la palabra, tenemos que elegir las letras que formaran dicha palabra. Las letras se irán eligiendo de forma aleatoria hasta completar la longitud correspondiente, siempre sin elegir letras ya usadas. Para comprobar que esto no suceda usamos el vector de boléanos.

Una vez generada la palabra, inicializamos las variables generales, como son las que nos indican las posibles letras a usar, la longitud de la palabra y la puntuación.

Por último realizamos la adaptación para calcular la aptitud del individuo, dicho cálculo se explica en la siguiente sección,

e) Función de Adaptación.

La función de adaptación sirve para calcular la aptitud del individuo. La aptitud nos sirve para comparar lo bueno que es un individuo con respecto a los demás individuos de la población. Mediante este valor podemos saber cual es la mejor solución y en el elitismo poder elegir los mejores individuos.

En este caso la función de adaptación consiste en buscar la palabra en la base de datos, y en caso de estar asignar a la aptitud la longitud de la palabra. En caso de que dicha palabra no esté en la base de datos se asigna a la aptitud el valor 0. Esta función de aptitud no es la más óptima, puesto que no tiene en cuenta la posible existencia de sílabas dentro de la palabra. Es decir que una aunque tenga varias sílabas correctas, en el idioma correspondiente, tendrá aptitud 0, igual que otra palabra que no tenga ninguna sílaba correcta. Con esto lo que se pasa es que se desperdician posibles buenas soluciones. Lo complicado de esta función de aptitud es su implementación y el excesivo gasto de tiempo en el cálculo de las sílabas y su comprobación en la base de datos.

En el caso básico de usar la función de adaptación de consulta de la palabra en la base de datos se pierde mucho tiempo en dicha consulta, puesto que la base de datos tiene un volumen enorme. Si se usara la función de aptitud basada en sílabas el tiempo de cómputo se multiplicaría, ya que además de buscar la palabra en la base de datos habría que calcular y buscar las posibles sílabas dentro de la palabra.

Para combatir el gasto el tiempo en acceso el acceso de tatos hemos optado por el uso del sistema gestor de base de datos mySQL, que proporciona unos resultados excelentes y es de distribución gratuita.

En resumen, el cálculo de la aptitud de toda la población resulta costoso en tiempo debido a la necesidad de acceder a la base de datos y el tiempo que esto conlleva.

f) Función de Selección.

Dependiendo de si se usa el elitismo se elije- una función distinta en la selección. Los elementos que se seleccionan son aquellos que van a sufrir los procesos de reproducción y cruce.

4. Usando elitismo. Al usar elitismo se eligen los individuos de la población con mayor aptitud. Ya que la población se ordena de mayor a menor en función de la aptitud sólo hay que seccionar los primeros de la población. En el caso del elitismo los individuos seleccionados se duplican para evitar que se pierdan en la reproducción o en la mutación. Este proceso nos garantiza que no se pierdan los que tienden a parecer ser los mejores individuos, pero nos puede hacer caer en máximos locales. Esto se evita teniendo una población amplia y un porcentaje de cruce y mutación óptimo.

5. Sin usar elitismo. Se intentan elegir los mejores individuos, aunque también se seleccionan algunos individuos que en principio no parece que vayan a conducir a la solución óptima, pero que podría ser que al cambiar alguna de sus propiedades resulten dar la solución óptima. El hecho de elegir algún individuo que en principio no sea el más adecuado, se hace para evitar quedarnos en máximos locales, con lo que nunca podríamos alcanzar el máximo global, que sería la solución óptima. Para realizar este cálculo usamos la ***puntuación acumulada*** del individuo, que han sido calculadas en la ***adaptación*** y un factor aleatorio para elegir los individuos.

```
void TPoblacion::seleccion(){
  float aleatorio;
  int aux,i,pos_super;
  int *sel_super=new int[_tam_poblacion];
  vector<TIndividuo*> pob_aux;
  float prob;
  for(i=0;i<_tam_poblacion;i++){
    aleatorio=rand();
    prob=(aleatorio/(RAND_MAX+1));
    pos_super=0;
    while ((pos_super<_tam_poblacion)&&
      (prob>_poblacion.at(pos_super)->obtenerPunt_acu()))
        pos_super ++;
      sel_super [i] =pos_super;}
    for(i=0;i<_tam_poblacion;i++){
     TIndividuo *ind = new TIndividuo(_poblacion.at(sel_super[i]));
     pob_aux.push_back(ind);}
      /*eliminar _poblacion*/
    vector<TIndividuo*>::iterator it;
    for(it= _poblacion.begin();it!=_poblacion.end(); ++it)
      delete (*it);
    _poblacion.clear();
     for(i=0;i<_tam_poblacion;i++){
       TIndividuo *ind2 = new TIndividuo(pob_aux.at(i));
      _poblacion.push_back(ind2);}
     /*eliminar _poblacion auxiliar*/
    for(it= pob_aux.begin();it!=pob_aux.end(); ++it)
     delete (*it);
    pob_aux.clear(); }
```

g) Función de Reproducción

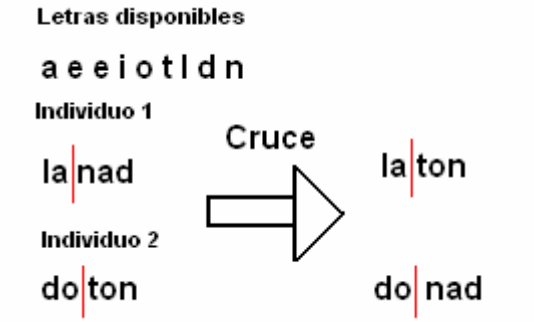
La función de reproducción consiste en elegir dos individuos de la población para ser cruzados. La proporción de individuos de la población que se van a reproducir viene en función de un parámetro del algoritmo evolutivo. Este parámetro es el ***porcentaje de cruce***, este factor es esencial para el correcto funcionamiento del algoritmo, puesto que si se cruza poco puede ser que no converjamos a la solución óptima, y si este valor es muy alto se pueden perder soluciones buenas y se consumiría mucho tiempo de CPU, puesto que después de realizar la reproducción hay que volver a calcular la aptitud de cada uno de los dos individuos, y como hemos comentado anteriormente, resuelta un proceso costoso en tiempo.

Para implementar la reproducción se recorre la población eligiendo los individuos a cruzar. La elección se realizar al azar en función del porcentaje de cruce. Una vez elegidos los individuos cruzar, se cruzan de dos en dos, se comprueba que el individuo resultante es correcto, cumpliendo las reglas del juego, se actualiza el vector de letras usadas y por último se recalcula la aptitud de cada uno.

- **Función de Cruce**

En este juego la función de cruce es muy sencilla, y consiste en elegir un punto de cruce en cada uno de los dos individuos a cruzar e intercambiar las letras. Este proceso es sencillo, aunque como resultado de él se pueden producir desajustes que hay que reparar. Dicho desajustes son la posibilidad de que se repitan letras en los nuevos individuos, con lo que no cumplirían las reglas del juego, con lo que hay que recorrer cada una de las palabras de los dos individuos comprobando que no se produce este error y solucionarlo en caso necesario. Para poder comprobar y solucionar este error se usa el vector de '*bool'* del individuo que nos indica las letras usadas para formar la palabra. De esta forma además de comprobar y reparar la palabra, actualizamos dicho vector que es una variable esencial.

El punto de cruce es aleatorio y, por lo general, diferente para cada uno de los individuos, pudiéndose generar palabras de más de 9 letras. Este error se subsana con el reajuste de la palabra, puesto que si una palabra tiene más de nueve letras es porque alguna se repite.



h) Función de Mutación

La función de mutación consiste en elegir un punto de la palabra y cambiar la letra correspondiente por otra que no haya sido usada. La decisión de los individuos de la población que se mutan viene determinada por el porcentaje de mutación. Este valor suele ser muy bajo, alrededor del 5 %. Si la palabra tiene nueve sílabas, entonces no se puede mutar ya que no hay ninguna letra libre para elegir. El punto de mutación es elegido de forma aleatoria dentro del rango permitido, es decir dentro de la longitud de la palabra.

Una vez realizada la mutación hay que actualizar el vector de letras usadas, marcando como usada la nueva letra que hemos seleccionado y desmarcando la letra remplazada.

Por último hay que recalcular la aptitud del individuo por medio de la adaptación, consultando la nueva palabra en la base de datos, comprobando si existe la palabra, y en caso afirmativo asignar la aptitud del individuo la longitud de la nueva palabra.

i)

j) Función para Revisar la Aptitud

En los problemas en los que hay que minimizar el valor de la aptitud hay que usar la función de revisar la aptitud mínima después de calcular la adaptación. Esta función consiste en invertir la aptitud de los individuos, es decir, en el caso de funciones de adaptación en las que se trata de minimizar la aptitud los mejores individuos son los que menor aptitud tienen, por lo que es necesario invertir este valor dándoles la mayor aptitud a los mejores individuos y la menor a los peores.

La implementación de esta función consiste en recorrer toda la población y elegir la aptitud máxima, este valor lo llamaremos CMax. Posteriormente volveremos a recorrer la población actualizando la aptitud de cada individuo con:
CMax-poblacion.at(i)->obtenerAptitud().

El código de la función sería el siguiente:

```
CMax=-MAXLONG;
int i;
for (i=0;i<poblacion.size();i++)
   poblacion.at(i)->adaptacion();
for (i=0;i<(poblacion.size());i++)
  if (poblacion.at(i)->obtenerAptitud()>CMax)
    CMax= poblacion.at(i)->obtenerAptitud();
for (i=0;i<(_poblacion.size());i++)
  poblacion.at(i)->act_aptitud(CMax-poblacion.at(i)>obtenerAptitud());
```

j) Elitismo

El elitismo es un factor esencial en los algoritmos evolutivos y consiste en preservar a lo largo de las sucesivas iteraciones los individuos mejores. Se considera que los mejores individuos son aquellos que tiene mayor aptitud, que se calcula gracias a la función de adaptación. Sin el elitismo es muy difícil conseguir buenas soluciones, lo único negativo del elitismo es que hay que usar mucha más memoria y tiempo de cómputo.

Preservar los mejores individuos en cada iteración favorece que ciertas características buenas, que han aparecido en algunos individuos se preserven e incluso puedan evolucionar para obtener mejores individuos.

La implementación del elitismo se realiza en varios pasos.

- Primer Paso: consiste en calcular la adaptación de todos los miembros de la población, para así poder saber cual son los mejores.

- Segundo Paso: consiste en ordenar la población en función de la aptitud, quedando los individuos con mayor aptitud al principio de la población.

- Tercer Paso: consiste en seleccionar "n" de los primeros individuos de la población, que son los mejores ya que la población ha sido ordenada en función de la aptitud. El número "n" se conoce como tamaño de la población elitista, y es otro parámetro importante en los algoritmos evolutivos. Si el tamaño de la población elitista es elevado el tiempo de ejecución se dispara, por el contrario si es pequeño podemos desperdiciar individuos con características muy interesantes. La selección de estos "n" individuos consiste en crear una copia de ellos y almacenarlos en vector auxiliar para que no se vean afectados por la mutación y por el cruce.

- Cuarto Paso: una vez realizada la mutación y la reproducción se fusionan las dos poblaciones, la que resulta de la mutación y la reproducción, y la población auxiliar elitista.

- Quinto Paso: después de fusionar las dos poblaciones se ordena la población resultante en función de la aptitud, y se eliminan los peores individuos, para que la población tenga el número de individuos especificados.

k) Código adicional desarrollado para el programa.

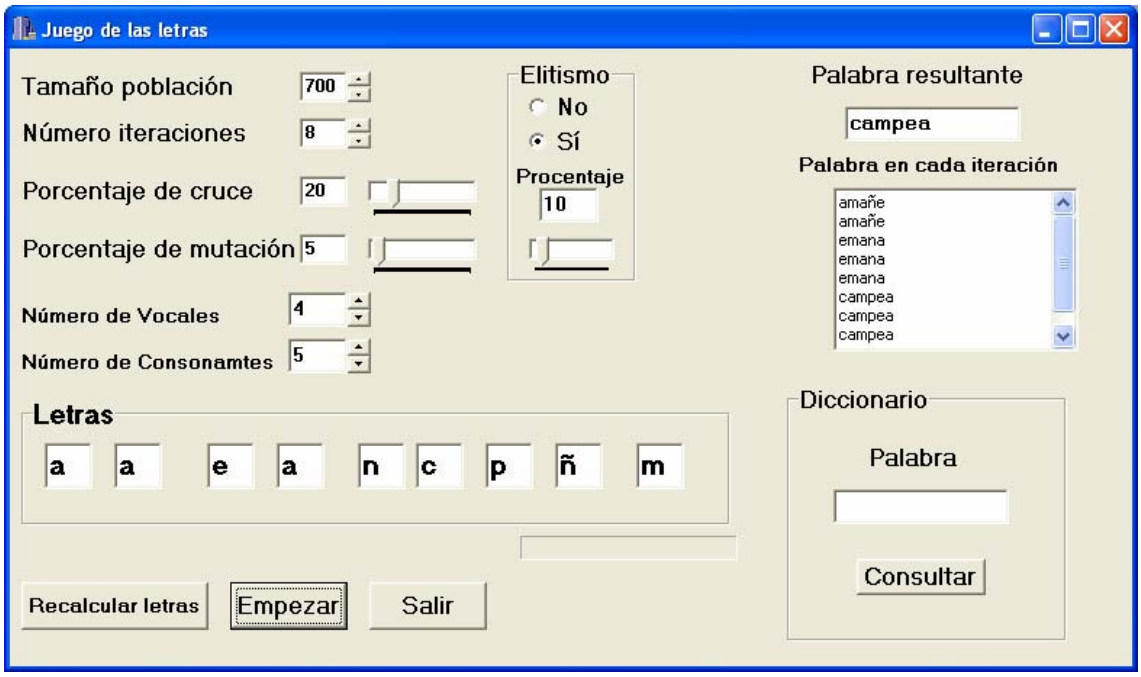
En el desarrollo del juego el código adicional al algoritmo evolutivo usado ha sido el uso del acceso a la base de datos. Este aspecto se comentará más adelante en el apartado de *Tecnologías Utilizadas*.

3. Presentación de la solución (entorno gráfico).

Para este juego se ha desarrollo un entorno grafico en forma de formulario en C++ Builder. Como en el juego anterior se intento implementar un acceso web al juego, pero es imposible la compatibilidad entre el ***ASAPI/NSAPI*** y el ***TQuery*** que se usa para acceder a la base de datos, es decir, no es compatible en las ***ASAPI/NSAP*** el acceso a la base de datos, por medio de las ***TQuery***.

En el entorno gráfico del ejecutable se ha usado los formularios del C++ Builder. En este entorno gráfico se pueden configurar todos los parámetros del algoritmo evolutivo, se muestran las letras disponibles para formar la palabra, la mejor palabra encontrada en cada iteración y la mejor palabra final. Las letras para formar la palabra pueden ser introducidas por el usuario, para poder comparar y repetir resultados. Además se ofrece la posibilidad de configurar el número de vocales y consonantes que formaran las letras disponibles para generar la palabra. Existe una barra de proceso que nos indica la fase en la que se encuentra el algoritmo evolutivo.

También se ofrece la posibilidad de que el usuario introduzca una palabra y compruebe si esta en la base de datos, para poder hacer comparativas con los resultados obtenidos por el algoritmo.



4. Versiones de la aplicación

A lo largo del desarrollo del juego se han propuesto varias versiones, pero sólo una ha podido ser implementada. Estas versiones adicionales son:

- La modificación de la función de adaptación, teniendo en cuenta las silabas de una palabra. Esta opción se desestimó por la dificultad de calcular las silabas de la palabra y el exceso de tiempo de cómputo consumido en acceder a la base de datos, que unido al tiempo para consultar la palabra entera, harían que la aplicación fuera muy lenta.
- Acceso web a la aplicación. Opción descartada por la imposibilidad de hacer compatible el ***ASAPI/NSAPI*** y el ***TQuery.***

Otras posibles versiones que si que se podrían implementar de forma fácil sería la extensión del juego a otros idiomas, ya que poseemos una base de datos con varios idiomas: inglés, catalán, francés, italiano, alemán. Lo único que habría que cambiar es el alfabeto de letras disponibles, adaptándolo a cada idioma.

7. Tecnologías empleadas

Entre las tecnologías usadas en la aplicación cabe destacar el uso del acceso a la base de datos. Este apartado ha sido bastante costoso y ha supuesto varias fases:

1. Encontrar un diccionario en español que se pudiera exportar a una base datos. Para conseguirlo se decidió realizar una búsqueda por Internet, con poco éxito, hasta encontrar un programa llamado ***WordBox***, que posee un archivo de texto con todas las palabras asociadas a un idioma. La funcionalidad del ***WordBox*** es la consulta de palabra en un diccionario, justamente la necesidad requerida por nosotros. Otra peculiaridad de ***WordBox*** es que esta disponible en varios idiomas: español, inglés, catalán, francés, alemán e italiano, con lo que es posible implementar el juego de las cifras en varios idiomas.

2. Decidir el sistema gestor de base de datos a usar. Viendo la magnitud de los datos a magnitud de los datos a usar ha sido necesario usar un SGBD eficaz y rápido. No era posible usar por ejemplo el ***Microsoft Access***, debido a su bajo rendimiento. La decisión tomada fue la de usar el ***mySQL*** bajo ***Windows XP***, la versión adquirida del ***mySQL*** fue la ***1.4.18***, que nos ha dado excelente resultado teniendo en cuenta la cantidad de accesos que se hace a la base de datos y el volumen de la misma. El ***mySQL*** es un SGBD de libre distribución, muy rápido y eficaz, soportando el acceso concurrente. Posteriormente hablaremos más en detalle del ***mySQL***.

3. Crear la base de datos donde guardar las palabras. Esto se realizó de forma sencilla mediante dos procesos:
   a. Creación del Database, mediante la siguiente sentencia SQL:
   ```
   CREATE DATABASE idioma;
   ```
   b. Creación del la Tabla:
   ```
   CREATE TABLE español(palabra char(30) binary not null primary Key);
   ```

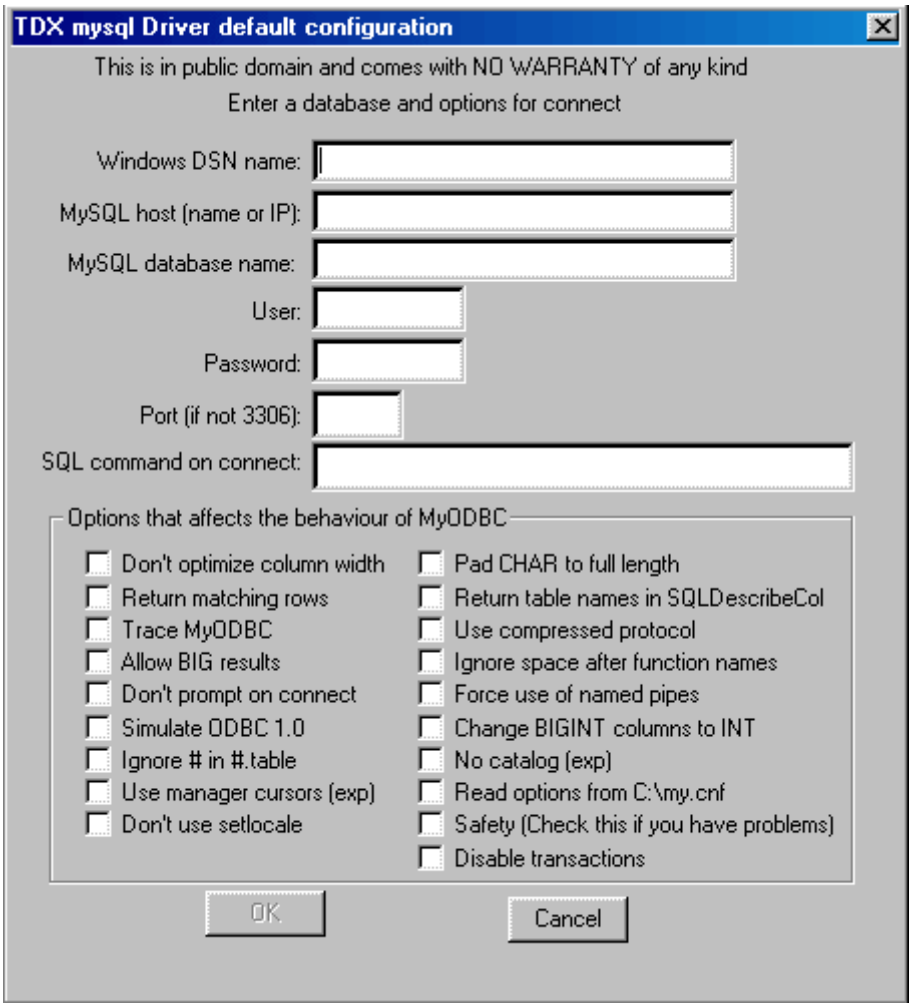
4. Cargar las palabras desde el archivo de texto del ***WordBox***, a la base de datos.

5. Crear el ***puente ODBC*** para que desde el C++ Builder se pueda acceder a la base de datos. Este apartado fue muy laborioso porque hubo que instalar actualizaciones de Windows y el ***driver OBDC*** para ***mySQL***. La documentación y los archivos necesarios para realizar esta operación se encontraron en la siguiente página: http://www.desarrolloweb.com/articulos/897.php?manual=34 , donde se describen los siguientes pasos:
   a. Lo primero que hay que hacer en el ordenador que tiene el sistema Windows XX y Access 2000 es actualizar a la versión 6 de Microsoft Jet: http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q239114&
   b. Una vez se ha hecho esto, hay que descargar la última versión de Myodbc de la página de Mysql: http://www.mysql.com/downloads/api-myodbc-2.50.html
   c. Cuando ya tenemos todo, instalamos la actualización de Microsoft Jet, y descomprimimos e instalamos el driver OBDC de Mysql.

Cuando pregunta en la pantalla de "Data Sources" haz clic en "Close" para terminar.

d. Una vez se ha instalado el driver ODBC, acceda al panel de control de OBDC de 32 Bits (Botón Inicio-> Configuración-> Panel de control-> Fuentes de datos ODBC 32 bits).

En este punto, tendría que elegir si quieres utilizar el driver para un solo usuario (DSN de usuario), o para cualquier usuario del ordenador (DSN de Sistema). Una vez hayas elegido uno, haz clic en el botón de "Agregar" para añadir una nueva fuente de datos y a continuación, selecciona el driver de Mysql. Aparecerá la siguiente pantalla:



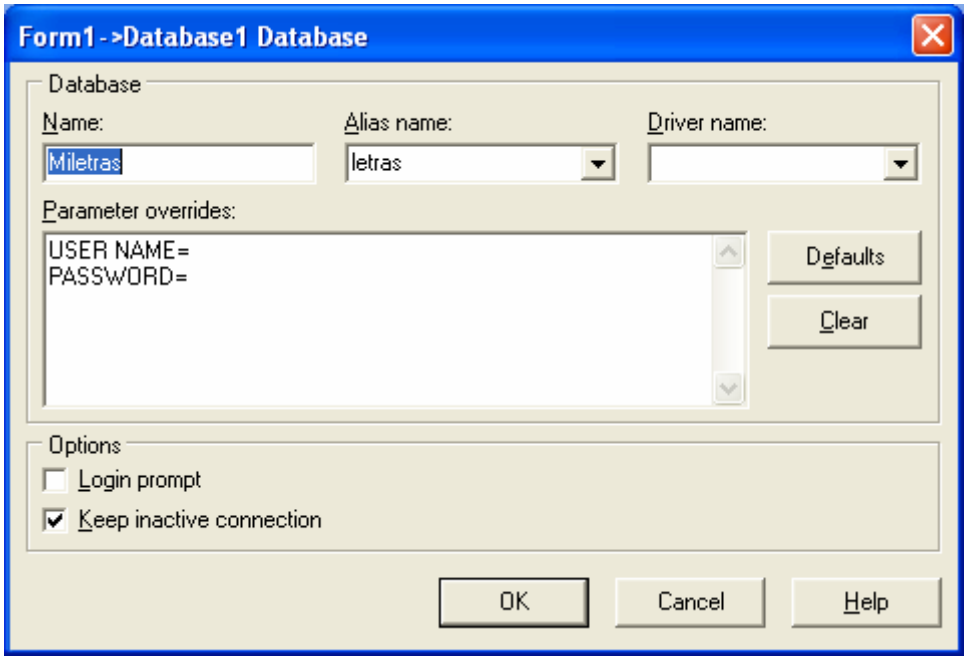
En ella tendrás que rellenar los siguientes campos:

Windows DSN name: Nombre de la fuente de datos que estará disponible desde Windows.
Mysql host (name or IP): Nombre o dirección IP del ordenador donde se encuentra instalado el servidor Mysql.
Mysql Database Name: Nombre de la base de datos con la que se trabajará desde la fuente de datos
User: Nombre de usuario con el que se accederá al servidor de bases de datos.
Password: Contraseña del usuario.
Port: Sirve para especificar el puerto en el que se encuentra el servidor

Mysql, hay que poner un valor en caso de que no se esté utilizando el predeterminado, que es el3306.

Una vez están estas opciones configuradas, se puede hacer clic en "OK" para cerrar las ventanas.

6. Crear el código en el C++ Builder para poder acceder a la base de datos. Es necesario usar dos variables para poder implementar el acceso a la base de datos, estas son:
   a. Una es de tipo ***TDatabase***, que nos permite establecer la conexión a la base de datos por medio del puente ODBC, descrito anteriormente. En esta variable se configura el nick y la contraseña con la que acceder a la base de datos y el puente ODBC elegido.

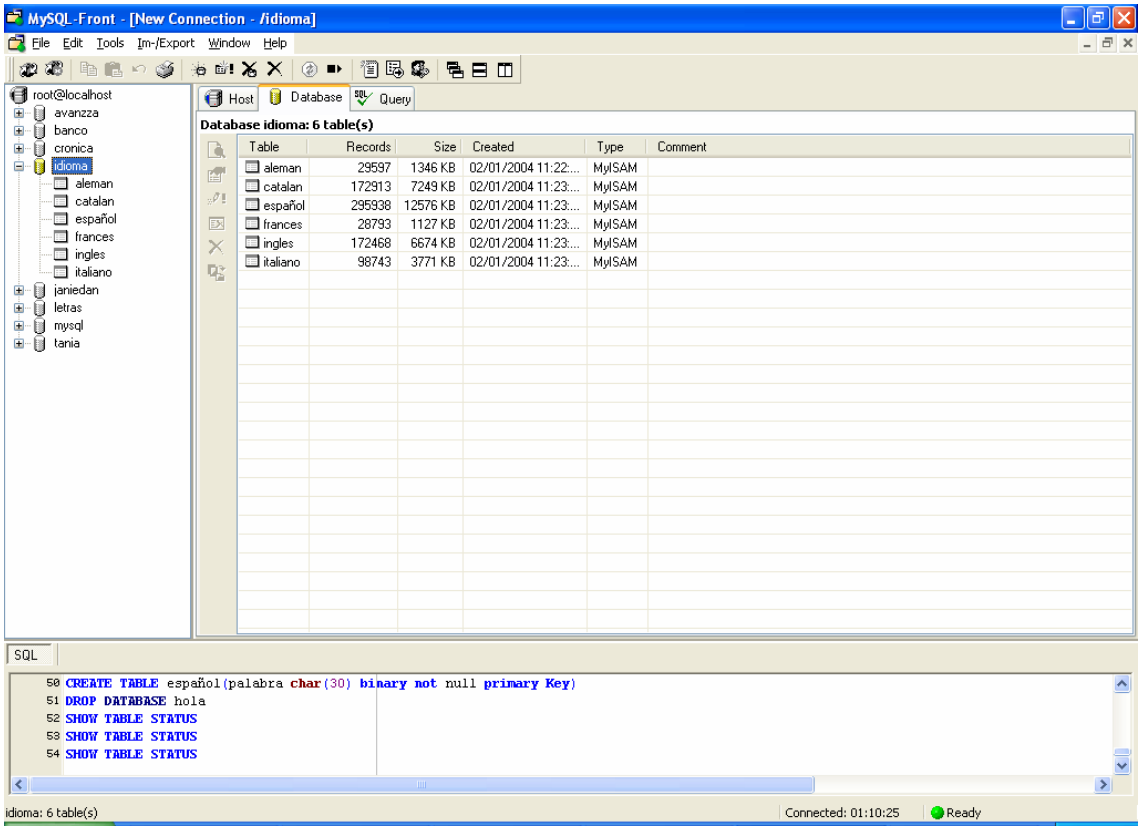


   b. La otra variable es de tipo ***TQuery***, y es donde se insertan y se ejecutan las consultas, y también donde se recogen los datos de la consulta SQL a la base de datos. A cada TQuery se le asocia su Database correspondiente, de manera que cada vez que se ejecute no hay que pedir ni nick ni contraseña. El código asociado a la comprobación de si una palabra está en la base de datos sería el siguiente:

```
bool TForm1::esta_palabra(String palabra){
  bool result=false;
  Query1->SQL->Clear();
  Query1->SQL->Add("select  palabra from tpalabras where palabra =
:Valor");
  Query1->ParamByName("Valor")->AsString =palabra;
  Query1->Open();
  if( Query1->RecordCount>0)
   result=true;
  Query1->Close();
  return result;
}
```

Para profundizar más en el mundo de las bases de datos mostraremos como se configura y se administra el mySQL de forma gráfica. Para esto se dispone de un

programa muy útil llamado MySQL-Front, y que suministra un entorno gráfico de acceso al motor de la base de datos, facilitando todo el proceso de gestión. En la siguiente figura se puede observar como se administra la base de datos idioma, que es a la que accedemos para comprobar si una palabra pertenece a un idioma concreto.

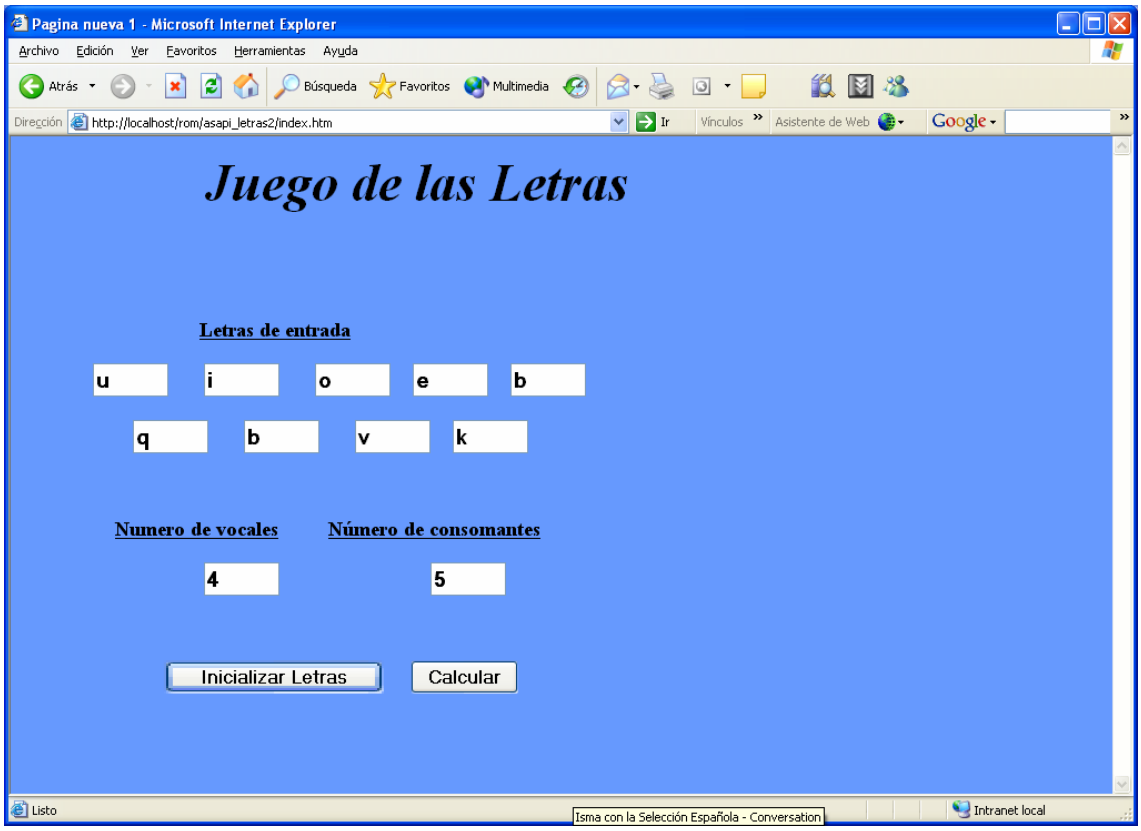


En el CD asociado a la memoria del proyecto se pueden encontrar todos los siguientes archivos:

1. Archivo de texto con todas las palabras en español del WordBox.
2. El ejecutable del WorBox en español.
3. El ejecutable del MySQL-Front.
4. El mySQL 1.4.18
5. El driver ODBC para mySQL (MyODBC-3.51.06).
6. La actualización de Windows XP para poder instalar el driver ODBC para mySQL(WindowsXP-KB282010-x86-ESN)

Como en el juego de las cifras se intentó generar un acceso web al juego por medio de una ***ASAPI/NSAPI***, pero resultó incompatible con el acceso a la base de datos. En cuanto se accedía dentro de la ***ASAPI/NSAPI*** al método que consulta la palabra en la base de datos se producía una excepción, diciendo que no encontraba la base de datos. Debido a esto no se puede generar la página con la solución, aunque todo el códido está implementado.

El entorno gráfico en el cliente sería el siguiente:



La generación aleatoria de las letras se hace mediante el uso de ***JavaScript***, el código sería el siguiente:

```
<SCRIPT LANGUAGE="JavaScript">
function inicializar( ){
  var alfabeto= new Array(27);
  var sol= new Array(9);
  var conson,vocal,i,j;
  conson=parseInt(document.forms[0].conson.value);
  vocal=parseInt(document.forms[0].vocal.value);
  for (i=0;i<vocal;i++){
   sol[i]=Math.floor(Math.round(5)*Math.random());
  }
  for (j=vocal;j<9;j++){
   sol[j]=Math.floor(Math.round(27-5)*Math.random()+5);
  }
  alfabeto [0]="a";
  alfabeto [1]="e";
  alfabeto [2]="i";
  alfabeto [3]="o";
  alfabeto [4]="u";
  alfabeto [5]="b";
  alfabeto [6]="c";
  alfabeto [7]="d";
  alfabeto [8]="f";
  alfabeto [9]="g";
  alfabeto [10]="h";
  alfabeto [11]="j";
  alfabeto [12]="k";
  alfabeto [13]="l";
  alfabeto [14]="m";
  alfabeto [15]="n";
  alfabeto [16]="ñ";
```

```
         alfabeto [17]="p";
         alfabeto [18]="q";
         alfabeto [19]="r";
         alfabeto [20]="s";
         alfabeto [21]="t";
         alfabeto [22]="v";
         alfabeto [23]="w";
         alfabeto [24]="x";
         alfabeto [25]="y";
         alfabeto [26]="z";
         document.forms[0].letra1.value =  alfabeto [sol[0]];
         document.forms[0].letra2.value =  alfabeto [sol[1]];
         document.forms[0].letra3.value =  alfabeto [sol[2]];
         document.forms[0].letra4.value =  alfabeto [sol[3]];
         document.forms[0].letra5.value =  alfabeto [sol[4]];
         document.forms[0].letra6.value =  alfabeto [sol[5]];
         document.forms[0].letra7.value =  alfabeto [sol[6]];
         document.forms[0].letra8.value =  alfabeto [sol[7]];
         document.forms[0].letra9.value =  alfabeto [sol[8]];
        }
       </SCRIPT>
```

La llamada desde el cliente al servidor, una vez rellenadas las casillas, sería:

```
<FORM ACTION="http://localhost/rom/asapi_letras2/Letras.dll" METHOD="POST" >
```

El código que se ejecuta en el servidor para generar la página web dinámica, con los datos rellenados en el cliente, es el siguiente:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request, TWebResponse *Response,
      bool &Handled)
{
  int i;
  inicilialicar( Request) ;
  poblacion=new TPoblacion(tam_pob,letras_selec,tam_pob_elit);
  poblacion->evaluar(pos_mejor,suma_aptitud);
  for (i=0 ;i<num_iterac;i++){
    poblacion->seleccion ();
    poblacion->reproduccion(porc_cruce);
    poblacion->mutacion(porc_mutac);
    poblacion->fusionar();
    poblacion->evaluar(pos_mejor,suma_aptitud);
  }
  Response->Content="La palabra encontada es:";
  Response->Content=Response->Content+( *letras_selec);
  Response->Content=Response->Content+
(*(poblacion>obtener_individuo(pos_mejor)->get_palabra()));
  delete letras_selec;
  delete poblacion;

}
//---------------------------------------------------------------------------

bool TWebModule1::esta_palabra(String palabra){
  bool result=false;
  Query1->SQL->Clear();
  Query1->SQL->Add("select  palabra from tpalabras where palabra = :Valor");
  Query1->ParamByName("Valor")->AsString =palabra;
  Query1->Open();
  if( Query1->RecordCount>0)
   result=true;
  Query1->Close();
  return result;
}
```

```
void TWebModule1::inicilialicar(TWebRequest *Request){
 tam_pob=700;
 num_iterac=20;
 porc_cruce=0.25;
 porc_mutac=0.07;
 tam_pob_elit=20;
 letras_selec= new String("");
 *letras_selec="";
 *letras_selec+=Request->ContentFields->Values["letra1"];
 *letras_selec+=Request->ContentFields->Values["letra2"];
 *letras_selec+=Request->ContentFields->Values["letra3"];
 *letras_selec+=Request->ContentFields->Values["letra4"];
 *letras_selec+=Request->ContentFields->Values["letra5"];
 *letras_selec+=Request->ContentFields->Values["letra6"];
 *letras_selec+=Request->ContentFields->Values["letra7"];
 *letras_selec+=Request->ContentFields->Values["letra8"];
 *letras_selec+=Request->ContentFields->Values["letra9"];
}
```

No puede ver la página web dinámica generada por el servidor puesto que en el método ***esta_palabra***, se produce una excepción de acceso a la base de datos, y no por lo tanto no genera la página web resultado.

6. Estudio de los parámetros del algoritmo

El estudio de los parámetros en un algoritmo evolutivo es un aspecto esencial, ya que aunque el algoritmo evolutivo este bien implementado puede ser que una mala elección de dichos parámetros haga que se encuentren malas soluciones o que el tiempo de cómputo sea excesivo.

Los parámetros de un algoritmo evolutivo son muy concretos:

- **Tamaño de la población**, parámetro que indica el número de individuos que va a poseer la población. Si el tamaño es excesivo se ralentiza mucho la ejecución del algoritmo, pero si es demasiado pequeño no se generan suficientes individuos para encontrar la solución óptima.
- **Número de iteraciones**, número de veces que se ejecuta el algoritmo evolutivo. Muchas veces se puede parar si vemos que ya hemos encontrado la solución óptima, con lo que ahorramos tiempo de cómputo.
- **Porcentaje de cruce**, indica que porcentaje de la población se va a cruzar. Este factor es muy importante porque el cruce suele consumir mucho tiempo de cómputo y si se cruza muchos individuos se ralentiza el algoritmo y además podríamos perder alguna solución óptima, en el caso de no usar elitismo.
- **Porcentaje de mutación**, indica el porcentaje de individuos de la población que se van a mutar y es un factor menos determinante que el de cruce, ya que suele requerir menos tiempo de CPU, pero también tiene su importancia en la medida en que cada vez que se produce una mutación, hay que recalcular la aptitud, como en el caso del cruce, y el cálculo de la aptitud puede resultar muy costoso.
- **Tamaño de la población elitista**, indica el número de individuos que se conservan en cada iteración, si el número es muy elevado se produce un exceso de uso de memoria, al guardar temporalmente los individuos mejores, y de CPU al tener que fusionar y reordenar la población para elegir los mejores.

En el caso particular del juego de las letras después de hacer una serie de pruebas, hemos llegado a la conclusión de que los valores ideales para los parámetros del algoritmo serían:

- **Tamaño de la población**: 800 individuos.
- **Número de iteraciones**: 8 iteraciones.
- **Porcentaje de cruce**: 20 %
- **Porcentaje de mutación:** 5 %
- **Tamaño de la población elitista:** 6 % del Tamaño de la población.

Se puede observar que el número de individuos y el de iteraciones es bastante bajo debido al consumo de tiempo que se gasta en el acceso a la base de datos. Es decir, no se puede aumentar los parámetros del algoritmo de forma significativa, porque significaría un descenso muy acusado del rendimiento del juego. Esto está en clara contraposición con lo que ocurre en el juego de las cifras, donde el rendimiento es espectacular y no se ve afectado por el aumento de los parámetros del algoritmo.

Como consecuencia de esta problemática, los resultados no son los óptimos, puesto que los parámetros del algoritmo evolutivo no son muy elevados. Sin embargo, dichos resultados se pueden considerar aceptables.

3.6. Juego de Master Mind.

Índice:

1. Introducción

-El Master Mind es un juego clásico. Consiste en adivinar una combinación de colores o números o imágenes o cualquier combinación de cosas propuesta por alguien. No sólo se tiene que adivinar la combinación, sino también la posición exacta de cada componente de la combinación oculta. El jugador que propone la combinación debe guiar, a través de pistas, al segundo jugador, una vez que este último ha hecho una propuesta. Las pistas tan sólo pueden decirle al segundo jugador cuantas piezas de la combinación oculta ha acertado y cuantas, además de adivinarlas, las ha puesto en el lugar correcto.

-En nuestro caso somos nosotros, el usuario, el que introduce una combinación de 4 colores, peces de colores, coches o famosos (son los 4 temas que le damos a elegir al usuario) que se eligen de un grupo, de cada uno de los temas, de 5 componentes, y es el ordenador, el que a través de cálculos basados en la programación evolutiva, debe adivinar la combinación oculta.

-El ordenador, basándose en las reglas del juego, sólo puede trabajar con las pistas que nosotros como usuarios le damos. Al principio, la primera de las combinaciones es totalmente aleatoria, pero a medida que va evolucionando el proceso y el ordenador va proponiendo nuevas propuestas, cuenta con más pistas para seleccionar de manera más adecuada a los mejores individuos de la población. Por ello, guardamos en un vector todas las propuestas que el ordenador hace y lo próximas que están esas propuestas a la solución, nos sirve para ordenar de manera adecuada dicho vector. De esta manera, cada individuo de la población es comparado con estas propuestas para saber así cuál es el más adecuado para presentar como nueva propuesta al usuario. Esto lo conseguimos a través de una función de adaptación adecuada y de darle más peso a las mejores propuestas. Si después de 10 iteraciones trabajando con la misma población no hemos conseguido ningún fruto, procedemos a la renovación de la misma y vuelta a empezar.

2. Aspectos del Algoritmo Evolutivo

a) Representación de la Población

-La población está compuesta por:

- El componente fundamental de la población es un puntero al tipo población, que es un vector de punteros al conjunto de individuos.

typedef vector<TIndividuo*> TPoblacion;

- Junto con el puntero al vector de punteros a individuos, el conjunto de propuestas es la otra parte importante de la población. Todas las propuestas que se hacen quedan almacenadas y ordenadas con respecto a los criterios establecidos, en un vector de TPropuestas, que es una clase creada para almacenar el código de la propuesta junto con el número de casillas bien colocadas y de casillas, que sin estar bien colocadas, si están en la combinación original.
- Además de las dos variables anteriores, se encuentran las típicas variables: la posición en la que se encuentra el mejor individuo, el tamaño de la población, el tamaño del elitismo, porcentaje de cruce, de mutación…,todos estos son parámetros que se incluyen en una clase TParámetros dentro de población y que se configuran fácilmente.

b) Generación de la Población.

-Generar la población llama a generar cada uno de los individuos y después de crearlos se meten de manera adecuada en el vector población.

-Señalar que cada 10 iteraciones sin que se encuentra la solución hemos tomado la determinación de renovar la población. Esto lo hemos hecho porque puede que la población generada al inicio no nos aporte nada positivo, incluso después de mutar, cruzar…..

c) Representación de los individuos.

-Vamos a detenernos en describir cada individuo generado. Un individuo tiene como parte fundamental el código que lo distingue. Dicho código se almacena en un puntero a una clase TCodigo, clase que a su vez define un tipo numerado con todos los posibles valores para los componentes de una posible combinación (blanco, negro, verde, azul y rojo). Un individuo, por lo tanto, no es ni más ni menos que una combinación de colores, la que lo determina y también es importante llevar una variable que nos indique lo que se parece el nuevo individuo a las propuestas utilizadas.

```
typedef enum {blanco,negro,rojo,verde,azul,ninguno1,ninguno2} TColores;

vector<TColores>* codigo; /*Un puntero a un vector de colores*/
```

-Los valores ninguno1 y ninguno2 nos sirven, únicamente y exclusivamente, para poder hacer comparaciones entre distintas combinaciones (para más detalle ir al código).

d) Generación de los Individuos.

-La generación de los individuos no tiene nada que reseñar. Lo único que se hace es establecer de manera aleatoria la combinación de colores que va a representar a dicho individuo.

e) Función de Adaptación

-Es bastante importante elegir de manera adecuada la función de adaptación para que las soluciones, en modo de propuestas, converjan de manera adecuada a la combinación secreta. Son muchas las funciones de adaptación que hemos probado, eso sí, siempre teniendo en cuenta las pistas que el usuario nos ha dado con anterioridad.

-Está claro que de algún modo, hay que tener en cuenta las propuestas que se han realizado y lo buenas que han sido estas. Para ello, el vector de propuestas con el que contamos se ordena atendiendo al número de casillas bien puestas y en el lugar correcto. Esto quiere decir que al final no hemos optado por tener en cuenta las casillas que coinciden con las casillas de la combinación secreta pero no están en el mismo lugar (vimos que no aportaban una mayor velocidad de convergencia). También hemos ignorado, por razones obvias, posibles combinaciones repetidas. La función de adaptación utilizada tiene la siguiente forma:

$$\text{adaptación} = \frac{\sum \text{puntuación(propuesta)} * |\text{lc-parecido(propuesta-individuo)}/\text{lc}}{\text{número de propuestas}}$$

donde lc es la longitud del código a adivinar y parecido(propuesta-individuo) nos da la diferencia entre las bien colocadas que tiene la propuesta en cuestión y el individuo. El sumatorio se extiende a cada propuesta.

g) Función de Selección.

-En este caso, hemos utilizado elitismo, por lo tanto la selección deja intactos para la siguiente iteración los mejores individuos (un tanto por ciento dentro del número

de individuos de la población que permita notar la mejora y que no ralentice en demasía el cómputo). Los seleccionados son los que se van a reproducir y mutar.

g) Función de Reproducción

-La reproducción consiste en el cruce entre las combinaciones de dos individuos. Los dos individuos tienen sus correspondientes combinaciones, se elige un punto al azar (entre 1 y 4) y se generan dos individuos nuevos en los que los colores de cada uno de ellos resultan de la combinación de los colores de ambos padres.

-El porcentaje de cruce para el que se obtienen buenos valores no deber pasar el 10%, así conseguimos frutos en el cruce y no se producen cruces innecesarios y que no llevan a ningún sitio, sólo a entorpecer el cálculo.

h) Función de Mutación

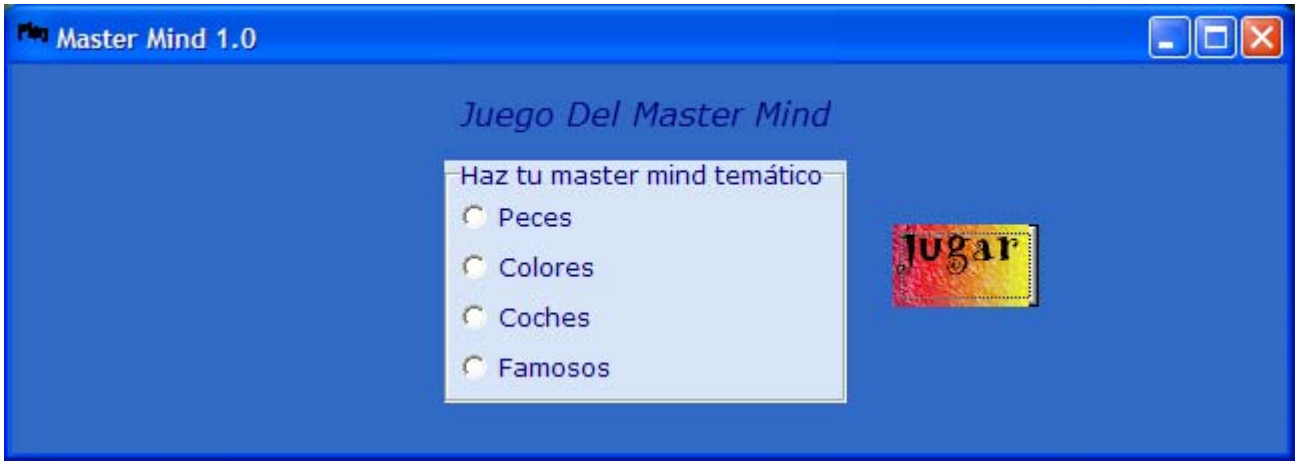
-Mutar un individuo significa cambiar uno de sus genes (colores). Es decir, se elige un gen dentro de la cadena de genes del individuo y se cambia su valor, en resumidas cuentas se cambia un componente de la combinación por otro color. Esto, claro está, supone un recalculo de la adaptación del individuo.

3.-Presentación de la solución (entorno gráfico).

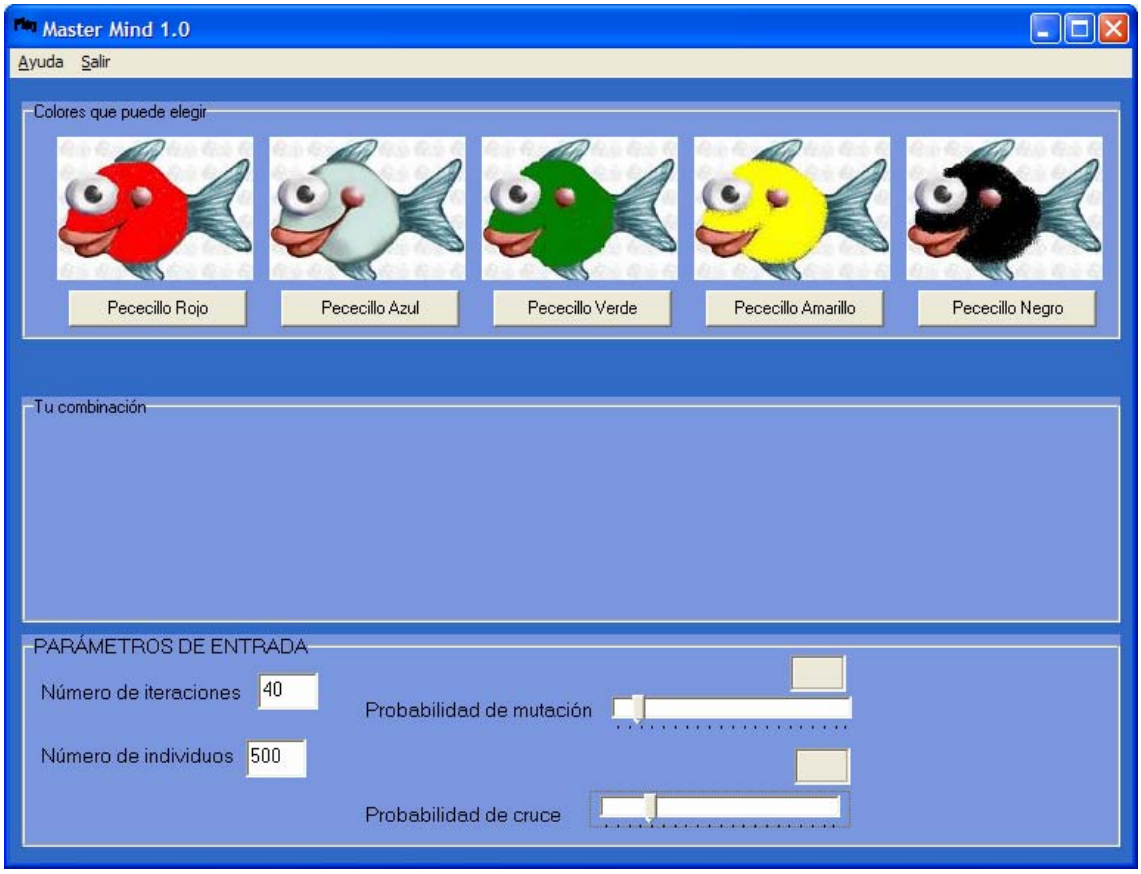
-Hemos elegido un entorno gráfico que hiciera lo más ameno posible un juego que de por sí, tampoco se caracteriza por necesitar un entorno gráfico excesivo.

-Al principio al usuario se le presenta la posibilidad de seleccionar entre cuatro "master minds distintos". Lo ponemos entre comillas, porque se pueden elegir entre cuatro temas distintos en cuanto a la presentación se refiere, pero el master mind es el mismo en todos los casos. Una vez seleccionado el tema se le presenta al usuario, una pantalla principal en la que debe elegir la combinación que el usuario quiere que el ordenador adivine, y si quiere configurar el tamaño de la población, la probabilidad de mutación, la probabilidad de cruce y el número de generaciones. Si el usuario está listo activa el ordenador y el cómputo empieza. Es al final de este cómputo cuando se le presenta al usuario la solución, si es que la ha encontrado, y todas y cada una de las propuestas que el ordenador ha ido haciendo hasta llegar a la propuesta final, que en la mayoría de los casos es la que el usuario esconde. Para que todo fuera más rápido y ameno, el ordenador una vez que hace una propuesta ya sabe el número de bien colocadas y bien puestas, es decir el usuario no tiene que meter nada, así es más rápido y ameno.

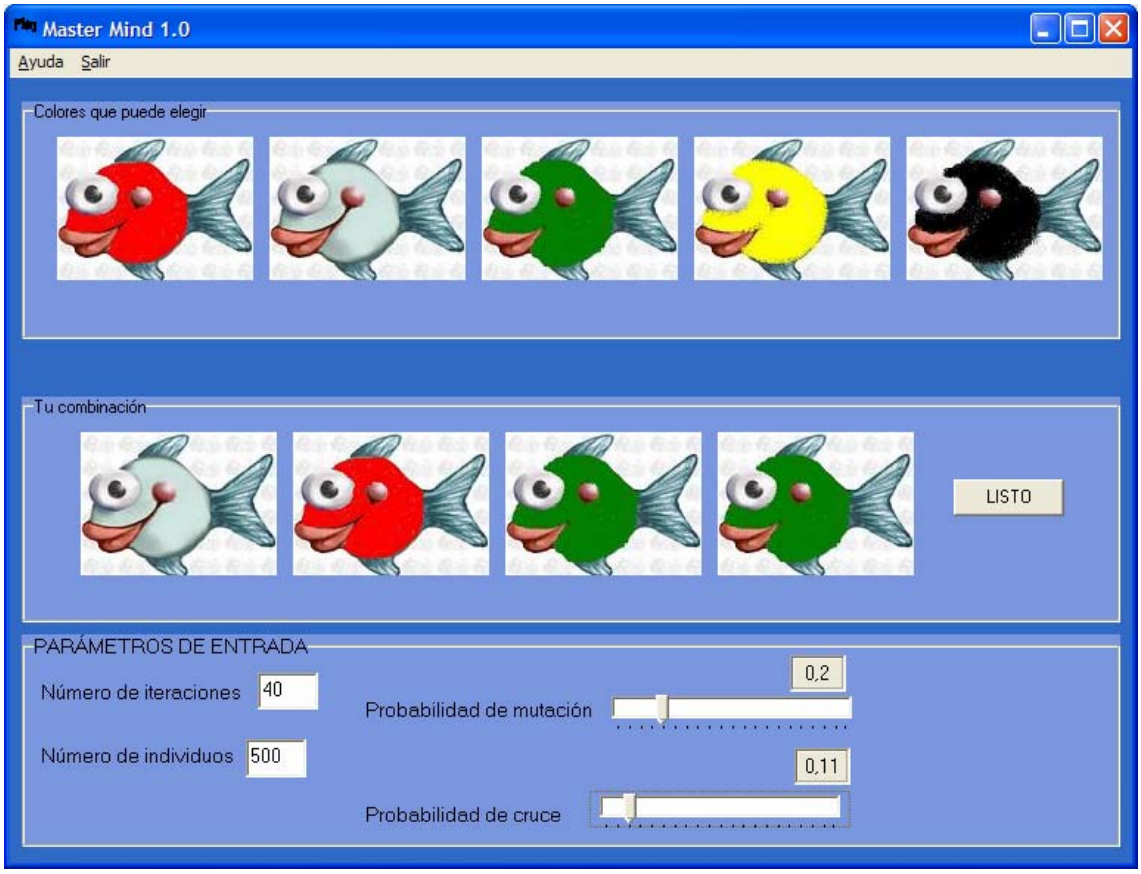
-A continuación presentamos algunas capturas de pantalla que muestran el juego y el entorno gráfico:



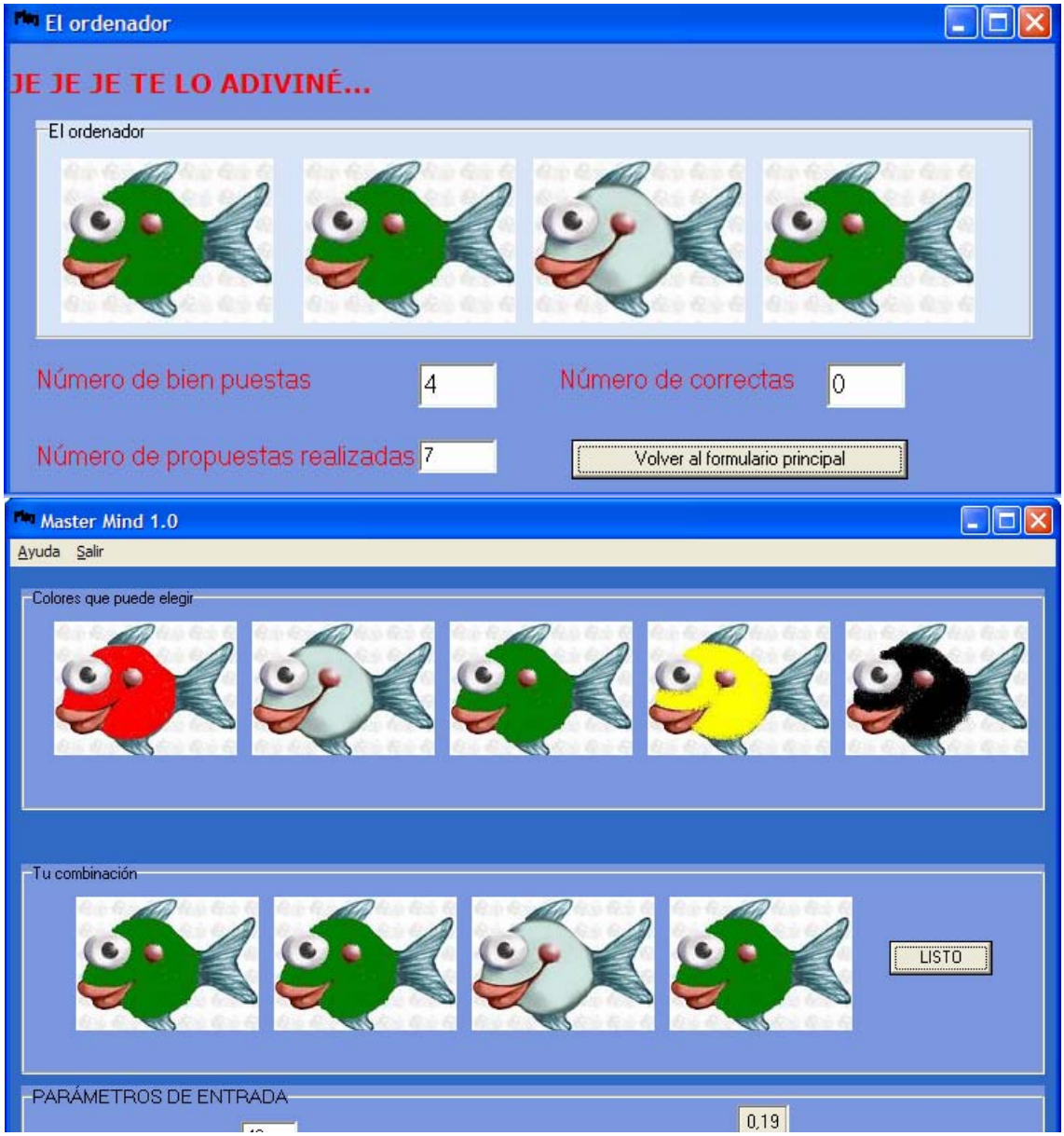
1.-Pantalla inicial del juego. El usuario puede elegir entre adivinar una combinación de peces, famosos, coches o colores.



2.-Una vez que el usuario se ha decidido por los peces, por ejemplo, tiene que introducir la combinación que desea que el ordenador adivine, así como modular los parámetros del algoritmo evolutivo.



3.-Esta sería una posible combinación y una introducción de los parámetros adecuada. Al usuario sólo le falta pulsar el botón LISTO para ponerse a jugar.



4.-Una vez que el ordenador ha terminado, te presenta todas las propuesta que ha ido realizando y el número de bien puestas y de correctas para cada propuesta en particular, hasta llegar a la última propuesta. En todo momento podemos ver, en la misma pantalla, nuestra combinación, y las que ha ido generando el ordenador hasta llegar a la propuesta final.

4.Versiones de la aplicación.

-Si contabilizamos una versión por cada una de las funciones de adaptación que hemos probado, podríamos a haber superado la decena. Sin embargo podemos contabilizar dos versiones. Una primera en la que la función de adaptación utilizada si conseguía converger en algunos casos, casos que nosotros considerábamos pocos y por eso evolucionamos hasta la versión definitiva, la 2.0, en la que en el 99% de los casos se

consigue adivinar la combinación.

5.-Tecnologías empleadas.

-Básicamente, C++ para el desarrollo íntegro de la práctica.

6.-Estudio de los parámetros del algoritmo.

-La experiencia de haber realizado numerosísimas ejecuciones del mismo algoritmo, nos lleva a concluir que un tamaño de la población mayor de 500 no conduce a mejores resultados y si a un aumento insostenible del tiempo de cálculo. Tengamos en cuenta, que puede que después de 10 iteraciones de mutaciones, cruces…se renueve la población íntegra y empecemos de nuevo todos los cálculos. El cruce y la mutación deben moverse en valores muy bajos de porcentaje, es decir, el número de cruces y de mutaciones tampoco tiene que ser muy grande para arrojar buenos resultados. Un número muy grande de iteraciones tampoco aporta mejoras. El óptimo lo hemos establecido en unas 40 iteraciones.

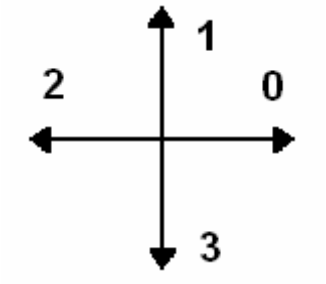
3.7. Juego del Puzzle.

Índice:

6. Estudio de los parámetros del algoritmo.

1. Introducción

El juego del puzzle de ocho consiste en partir un tablero inicial, donde hay ocho piezas descolocadas y un hueco, y la idea es ir moviendo el hueco para conseguir que la figura quede colocada, con el hueco en la parte inferior derecha. Las direcciones en las que se puede mover el hueco son: Norte, Sur, Este y Oeste, los cuatro puntos cardinales. No esta permitido el movimiento del hueco de forma oblícua. Hay que intentar minimizar el número de movimientos. Para representar los posibles movimientos del hueco, a cada uno se le ha signado un número entero:



El juego genera al azar un puzzle descolocado, he intentar calcular una serie de movimientos del hueco que conduzcan a colocar todas las partes de la figura.

Se ha intentado que el juego no repita movimientos, para evitar ciclos y movimientos innecesarios que generasen malas soluciones, mediante la consideración del movimiento anterior.

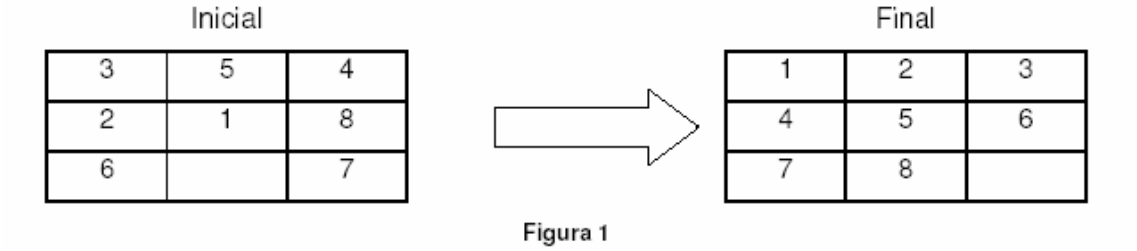


Figura 1

Para este juego existe la posibilidad de ampliar el tablero, a 4x4, es decir, un puzzle de 15 fichas y un hueco. Esto complica en gran medida el cálculo porque como es evidente es mucho más costoso encontrar la solución óptima. En una de las versiones de este juego se ha implementado esta ampliación.

2. Aspectos del Algoritmo Evolutivo

a) Representación de la Población

La población está compuesta por:

- Un vector de punteros a individuos. En este vector se almacenan los punteros a los individuos que forman la población. El hecho de usar punteros mejora el rendimiento y velocidad de la aplicación, por el contrario se necesita tener más cuidado con la destrucción de los punteros, para que no se quede memoria ocupada de forma inútil. En el destructor de la población hay que eliminar uno a uno todos los punteros del vector. Este vector es de tipo clase ***Vector***, predefinida en el C++ Builder, y que proporciona un interfaz cómodo y eficaz para almacenar punteros a objetos, aumentando así el rendimiento de la aplicación
- El número de individuos de la población. Número de individuos que formaran la población
- Tamaño del elitismo. Este parámetro es esencial para obtener la solución óptima, ya que representa el número de mejores elementos que conservamos en cada iteración, para que no se pierdan dichos individuos.

b) Generación de la Población.

La generación de la población se realiza creando cada uno de los individuos y añadiendo el puntero de cada uno de los individuos al vector que representa la población.

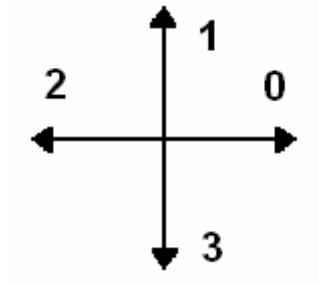
Un aspecto importante es la destrucción de los individuos que forman el vector, es decir, no sólo hay que liberar los punteros del vector, sino que también hay que liberar el objeto que apunta cada uno de los punteros. La liberación de memoria en el C++ Builder es importante estar atento a la destrucción de los punteros que ya no sean útiles, puesto que a diferencia con Java no tiene recopilación automática de memoria no útil.

La recopilación de memoria no usada y el uso de punteros cobra especial relevancia en los algoritmos evolutivos puesto que en ellos se usa una gran cantidad de memoria y velocidad de cómputo, por lo que no conviene desperdiciarlas.

c) Representación de los Individuos.

En este juego cada individuo posee las siguientes características:

- Un Vector de enteros con la lista de movimientos del hueco. El hueco sólo se puede mover en cuatro direcciones, el la dirección de los cuatro puntos cardinales. Estas cuatro direcciones se representan por cuatro números enteros: 0, 1, 2, 3, de la siguiente forma:



- Un puzzle inicial de partida, que ha sido generado aleatoriamente, A partir del puzzle inicial y aplicando la lista de movimientos podemos llegar al puzzle final. El puzzle final es el que representa la solución del algoritmo.

- Un puzzle final, para saber el resultado del algoritmo. Podemos llegar a él, partiendo desde el puzzle inicial y aplicando la lista de movimientos, pero resultaría muy costoso, con lo que guardamos dicho puzzle en una variable, de forma que pueda ser acceder a él de forma sencilla. A partir de el se calcula la aptitud del individuo.

- Una variable **float**, que representa la aptitud del individuo, representando lo adaptado que esta el individuo con respecto a los demás individuos de la población.

- Una variable **float**, que representa la puntuación del individuo.

- Una variable **float**, que representa la puntuación acumulada del individuo.

- Una variable **bool**, que nos indica si en la función de adaptación usamos la distancia Manhatan o contar el número de casillas correctas.

d) Generación de los Individuos.

Lo primero en la generación del individuo es calcular el número máximo de movimientos que va a realizar el hueco. Este número se calcula al azar, habiendo un valor máximo y otro mínimo. El valor máximo es 25 y el mínimo 50. El código asociado es:

```
int num_mov=(rand()% 25)+25;
```

Lo siguiente es crear tanto el puzzle inicial, como el final a partir del puzzle generado aleatoriamente. El puzzle inicial no se modificará en todo el algoritmo, y nos servirá de punto de partida en caso de tener que recalcular el puzzle final. El puzzle final va a ir cambiando según vayamos generando movimientos, puesto que cada vez que se genera un movimiento el hueco cambia en esa dirección.

Una vez elegido el número de movimientos máximo, se van generando movimientos hasta llegar a dicho número o a encontrar el puzzle solución buscado. Si se encuentra el puzzle solución entonces se detiene el proceso de generación de movimientos. Este proceso consiste en elegir un moviendo posible del hueco, mover el hueco actualizando el puzzle final y almacenar dicho movimiento en la lista de movimientos. El cálculo del posible movimiento tiene en cuenta la posición actual del hueco, ya que puede ser que este en el borde del tablero y hay algunos movimientos que los tiene restringidos ya que no se puede salir del tablero, además se tiene en cuenta el anterior movimiento para que no se repitan movimientos. Esto se consigue guardando en una variable auxiliar el anterior movimiento. El código sería:

```
int dir;
 list_mov=new vector <int>;
 puz_inic=new Puzle(puz_inicial);
 puz_final=new Puzle(puz_inic);
 while ((seguir)&&( i<num_mov))
   if (puz_final->num_cas_corectas()==9)
    seguir=false;
   else {
      dir=puz_final->cal_mov();
      puz_final->mover_hueco(dir);
      list_mov->push_back(dir);
      i++;
    }
```

Por ultimo habría que calcular la aptitud del individuo por medio la adaptación, evaluando el puzzle final con la heurística seleccionada.

e) Función de Adaptación

La función de adaptación sirve para calcular la aptitud del individuo. La aptitud nos sirve para comparar lo bueno que es un individuo con respecto a los demás individuos de la población. Mediante este valor podemos saber cual es la mejor solución y en el elitismo poder elegir los mejores individuos.

El cálculo de la aptitud de un individuo se obtiene evaluando el puzzle final, ofreciéndose la posibilidad de usar dos funciones de aptitud distintas:

1. Contar el número de casillas bien colocadas, es una heurística muy sencilla y fácil de implementar, y además consume poco tiempo de cómputo. Los resultados obtenidos con esta opción son peores y se necesita aumentar los parámetros del algoritmo genético, es decir aumentar el número de individuos y de iteraciones. Al producirse dicho aumento se produce un incremento en el tiempo que tarda el algoritmo en calcular la solución.

3	5	4
2	1	8
6		7

Número de casillas correctas: 0

1	2	3
4	5	6
7	8	

Número de casillas correctas: 9

2. La distancia Manhatan, tiene en cuenta la posición de cada casilla y es la suma de la distancia cada casilla a su posición correcta, pero no pudiéndose mover en diagonal. Esta heurística da mejores resultados puesto que tiene en cuenta información más concreta. El tiempo de cálculo es superior, pero se compensa al tener que usar menos individuos y menos iteraciones.

3	5	4
2	1	8
6		7

Distancia Maniatan: 2+1+3+2+2+2++3+1+2=18

1	2	3
4	5	6
7	8	

Distancia Maniatan: 0

Como se puede observar las dos funciones de adaptación son muy distintas, y una de las cosas que tienen distinta y que es más importante es que en una hay que maximizar la función de adaptación y en la otra minimizarla. En el caso de número de casillas bien colocadas hay que maximizar la función de aptitud, intentando que haya el máximo de casillas bien colocadas posible. En el caso de la distancia Manhatan hay que intentar minimizar su valor, procurando que la distancia de cada casilla a su sitio correcto sea cero. Por tanto, para la distancia Manhatan hay que usar la función ***Revisar la Aptitud***, para recalcular la aptitud de forma correcta. Para la otra heurística no haría falta.

Además de encontrar la solución correcta hay que minimizar el número de movimientos, es decir, el puzzle solución hay que tratar de encontrarlo en el menor número de movimientos posible. Por tanto hay que penalizar el número de movimientos. Dado que las dos heurísticas se comportan de forma distinta hay que tratar esta penalización de forma separada:

- En el caso del número de casillas bien colocadas, se resta una pequeña cantidad, para que en el caso de que haya dos soluciones con el mismo número de casillas bien colocadas elija la que en menos movimientos se haya conseguido. Esta cantidad se calcula multiplicando un valor constante por el número de movimientos.
- En el caso de la distancia Maniatan, ya que el objetivo es minimizar, para penalizar el número de movimientos hay que sumar una penalización por el número de movimientos.

```
void TIndividuo::adaptacion(){
  if (Manhattan){
    _aptitud=puz_final->dist_manhattan();
    _aptitud=_aptitud+(0.001* list_mov->size());
    }
  else{
    _aptitud=puz_final->num_cas_corectas();
    _aptitud=_aptitud-(0.001* list_mov->size());
    }
}
```

f) Función de Selección

En este juego se ha optado por dar sólo la opción de usar sólo elitismo, ya que las funciones heurísticas están bastante adaptadas al problema, y hacen que los mejores individuos converjan a la solución. Por lo tanto sería desaconsejable usar la otra opción.

6. Usando elitismo. Al usar elitismo se eligen los individuos de la población con mayor aptitud. Ya que la población se ordena de mayor a menor en función de la aptitud sólo hay que seccionar los primeros de la población. En el caso del elitismo los individuos seleccionados se duplican para evitar que se pierdan en la reproducción o en la mutación. Este proceso nos garantiza que no se pierdan los que tienden a parecer ser los mejores individuos, pero nos puede hacer caer en máximos locales. Esto se evita teniendo una población amplia y un porcentaje de cruce y mutación óptimo. El elitismo, como es obvio, ralentiza la aplicación, pero los beneficios obtenidos son muy notables, y en muchos problemas es muy difícil obtener la solución óptima sin utilizar esta técnica.

g) Función de Reproducción

La función de reproducción consiste en elegir dos individuos de la población para ejecutar el cruce de los dos individuos elegidos. La proporción de individuos de la población que se van a reproducir viene en función de un parámetro del algoritmo evolutivo. Este parámetro es el ***porcentaje de cruce***, este factor es esencial para el correcto funcionamiento del algoritmo, puesto que si se cruza poco puede ser que no converjamos a la solución óptima, y si este valor es muy alto se pueden perder soluciones buenas y se consumiría mucho tiempo de CPU, puesto que la después de realizar la reproducción hay que volver a calcular la aptitud de cada uno de los dos individuos, y como hemos comentado anteriormente, resuelta un proceso costoso en tiempo.

Para implementar la reproducción se recorre la población eligiendo los individuos a cruzar. La elección se realizar al azar en función del porcentaje de cruce. Una vez elegidos los individuos cruzar, se cruzan de dos en dos, hay que comprobar que el individuo resultante es correcto, cumpliendo las reglas del juego, hay que actualizar el vector de letras usadas y por último se recalcula la aptitud de cada uno.

- **Función de Cruce**

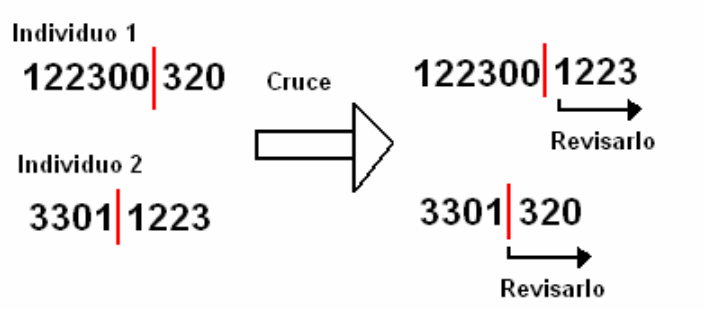
La función de cruce consiste en elegir un punto de corte en el vector de movimientos de cada individuo e intercambiar los movimientos a partir de dicho punto. Este punto se elige al azar dentro del rango posible de los respectivos vectores. Una vez intercambiados los movimientos hay que proceder al reajuste de dichos movimientos desde el punto de cruce, puesto que puede ser que muchos de los nuevos movimientos introducidos no sean correctos o sean repetidos.

Lo primero que hay que hacer es partir del puzzle inicial e irle aplicando los movimientos que no han sido modificados, que sabemos que son correctos, hasta generar el puzzle que correspondería al punto de corte.

A partir de este puzzle, podemos irle aplicando la lista de movimientos nueva, y en caso de alguno de los movimientos sea incorrecto o repetido se calcula uno nuevo correcto, y se prosigue con la comprobación del siguiente movimiento. A la vez que se va comprobando que los movimientos son correctos, se va actualizando el puzzle final, con cada movimiento.

También puede ocurrir que antes de llegar al final de la comprobación de la nueva lista de movimientos hayamos encontrado la solución buscada, con lo que se para el reajuste y se eliminan los movimientos sobrantes. Este proceso de reajuste el bastante complejo, pero es necesario para comprobar que todos los movimientos son correctos y para generar el puzzle final.

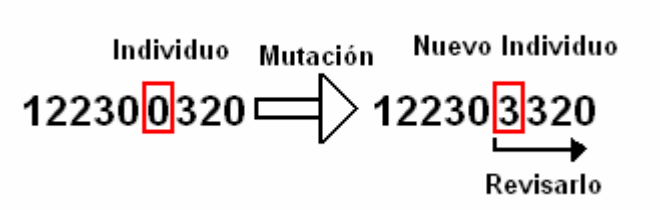
Después de realizar el cruce hay que volver a calcular la adaptación del individuo, puesto que el puzzle final al aplicarle los nuevos movimientos ha cambiado, y el valor de la aptitud depende de la posición de las casillas en el puzzle final.


Individuo 1
122300 320
Individuo 2
3301 1223
Cruce
122300 1223
Revisarlo
3301 320
Revisarlo

h) Función de Mutación

La función de mutación consiste en elegir un punto de la lista de movimientos, y cambiar dicho movimiento por otro posible. La decisión de mutar un individuo depende de un factor del algoritmo evolutivo llamado ***porcentaje de mutación***, y suele tener un valor muy bajo. El punto de mutación de la lista de movimientos se elige al azar entre la longitud del vector. Al igual que ocurre en el cruce, una vez modificado el vector de movimientos hay que revisar los nuevos movimientos para comprobar que son correctos y en caso de no ser se reemplaza por uno movimiento posible. Además hay que ir generando el puzzle final correcto, a partir del puzzle inicial y aplicando los viejos movimientos y los nuevos. Podría ser que al comprobar los nuevos movimientos o al generar algunos nuevos, en caso de que fueran erróneos, se encontrara el puzzle solución, con lo que se pararía la comprobación y se eliminarían los movimientos restantes.

Además, una vez acabado este proceso hay que realizar el cálculo de la aptitud del individuo, por medio de la adaptación, puesto que el puzzle final ha cambiado con los nuevos movimientos y la aptitud del individuo se basa en la posición de las casillas en el puzzle final.



i) Función para Revisar la Aptitud

En los problemas en los que hay que minimizar el valor de la aptitud hay que usar la función de revisar la aptitud mínima después de calcular la adaptación. Esta función consiste en invertir la aptitud de los individuos, es decir, en el caso de funciones de adaptación en las que se trata de minimizar la aptitud los mejores individuos son los que menor aptitud tienen, por lo que es necesario invertir este valor dándoles la mayor aptitud a los mejores individuos y la menor a los peores.

La implementación de esta función consiste en recorrer toda la población y elegir la aptitud máxima, este valor lo llamaremos CMax. Posteriormente volveremos a recorrer la población actualizando la aptitud de cada individuo con:
CMax-poblacion.at(i)->obtenerAptitud().

El código de la función sería el siguiente:

```
CMax=-MAXLONG;
int i;
for (i=0;i<poblacion.size();i++)
   poblacion.at(i)->adaptacion();
for (i=0;i<(poblacion.size());i++)
  if (poblacion.at(i)->obtenerAptitud()>CMax)
    CMax= poblacion.at(i)->obtenerAptitud();
for (i=0;i<(_poblacion.size());i++)
  poblacion.at(i)->act_aptitud(CMax-poblacion.at(i)>obtenerAptitud());
```

j) Elitismo

El elitismo es un factor esencial en los algoritmos evolutivos y consiste en preservar a lo largo de las sucesivas iteraciones los individuos mejores. Se considera que los mejores individuos son aquellos que tiene mayor aptitud, que se calcula gracias a la función de adaptación. Sin el elitismo es muy difícil conseguir buenas soluciones, lo único del elitismo es que hay que usar mucha más memoria y tiempo de cómputo.

Preservar los mejores individuos en cada iteración favorece que ciertas características buenas, que han aparecido en algunos individuos se preserven e incluso puedan evolucionar para obtener mejores individuos.

La implementación del elitismo se realiza en varios pasos.

- Primer Paso: consiste en calcular la adaptación de todos los miembros de la población, para así poder saber cual son los mejores.

- Segundo Paso: consiste en ordenar la población en función de la aptitud, quedando los individuos con mayor aptitud al principio de la población.

- Tercer Paso: consiste en seleccionar "n" de los primeros individuos de la población, que son los mejores ya que la población ha sido ordenada en función de la aptitud. El número "n" se conoce como tamaño de la población elitista, y es otro parámetro importante en los algoritmos evolutivos. Si el tamaño de la población elitista es elevado el tiempo de ejecución se dispara, por el contrario si es pequeño podemos desperdiciar individuos con características muy interesantes. La selección de estos "n" individuos consiste en crear una copia de ellos y almacenarlos en vector auxiliar para que no se vean afectados por la mutación y por el cruce.

- Cuarto Paso: una vez realizada la mutación y la reproducción se fusionan las dos poblaciones, la que resulta de la mutación y la reproducción, y la población auxiliar elitista.

- Quinto Paso: después de fusionar las dos poblaciones se ordena la población resultante en función de la aptitud, y se eliminan los peores individuos, para que la población tenga el número de individuos especificados.

k) Código adicional desarrollado para el programa.

Para el desarrollo de la práctica ha sido necesario implementar la clase ***Puzzle***, que proporciona todas las funciones y variables para poder administrar los dos puzzles con los que cuenta cada individuo, el puzzle inicial y el puzzle final.

La clase ***Puzzle*** es la clase primordial del juego, ya que tiene todas las operaciones que se realizan sobre los dos puzzles de cada individuo.

Las **variables** que posee son:

1. Una matriz de dimensiones 3x3 de enteros, donde se guarda la representación del puzzle
   int matriz [3] [3];

2. La posición del hueco, mediante su posición ***x*** e ***y***.
   int hueco_x,hueco_y;

3. La dirección anterior, para no repetir movimientos.
   int dir_ant;

Los **métodos** que posee son:

1. Un constructor por defecto, que crea de forma aleatoria un puzzle.
   Puzle();

2. Un constructor a partir de otro puzzle, creando una copia del objeto que se le pasa:
   Puzle(Puzle *puz);

3. Una función que pasado un punto, con componentes ***i*** e ***j*** devuelve el valor del puzzle para ese punto:
   int obt_casilla(int i,int j);

4. Una función que calcula el número de casillas bien colocadas.
   int Puzle::num_cas_corectas ();

5. Un procedimiento que dada una dirección mueve el hueco:
   void Puzle::mover_hueco(int dir);

6. Una función que dado el puzzle calcula un posible movimiento, teniendo en cuenta el movimiento anterior y la matriz que representa al puzzle:
   int Puzle::cal_mov();

7. Una función que indica si un movimiento es posible:
   bool Puzle::mov_posible (int dir);

8. Una función que calcula la distancia Manhatan:
   int Puzle::dist_manhattan();

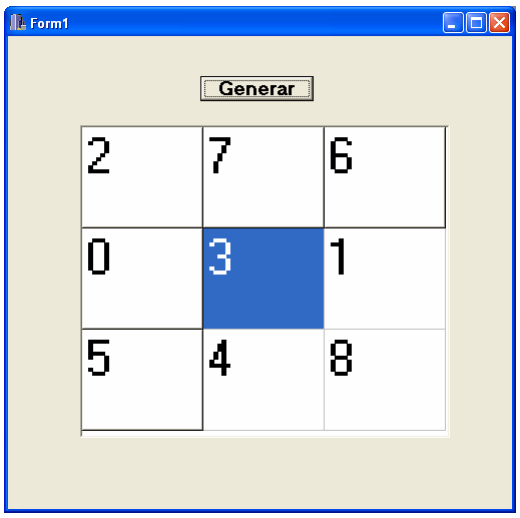
9. Una función que calcula la posición correcta de una casilla en el puzzle, por ejemplo, la posición del *0* es la x=0 e y=0; y la del *5* es x=1 e y=1.
   void Puzle::cal_comp(int num,int &pos_x,int &pos_y);

3. Presentación de la solución (entorno gráfico).

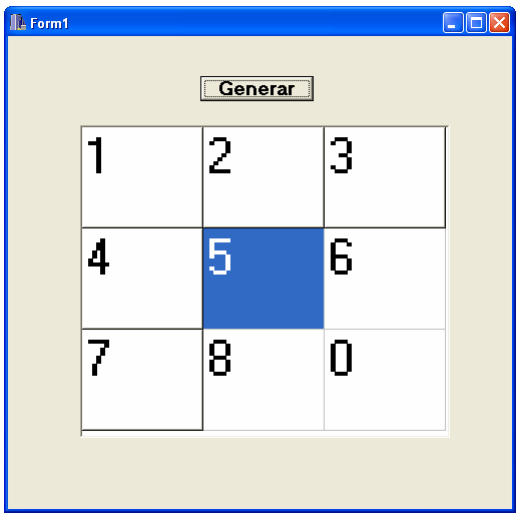
A medida que se han ido generando nuevas versiones del juego se ha ido mejorando el entorno gráfico.

En la primera versión la representación del puzzle se hacía mediante el uso de números. La forma de mostrar los números era muy sencilla, mediante el uso de una tabla (***TStringGrid***) y escribiendo en cada casilla de la tabla el número correspondiente. La forma de ir repintando la tabla y que así se muestre cómo se mueve el hueco, se hace mediante el uso de un temporizador (***TTimer***), el cuál se ejecuta cada un cierto tiempo, tras el cual se actualiza la el puzzle con el siguiente movimiento.

Puzzle Inicial.



Puzzle Final.

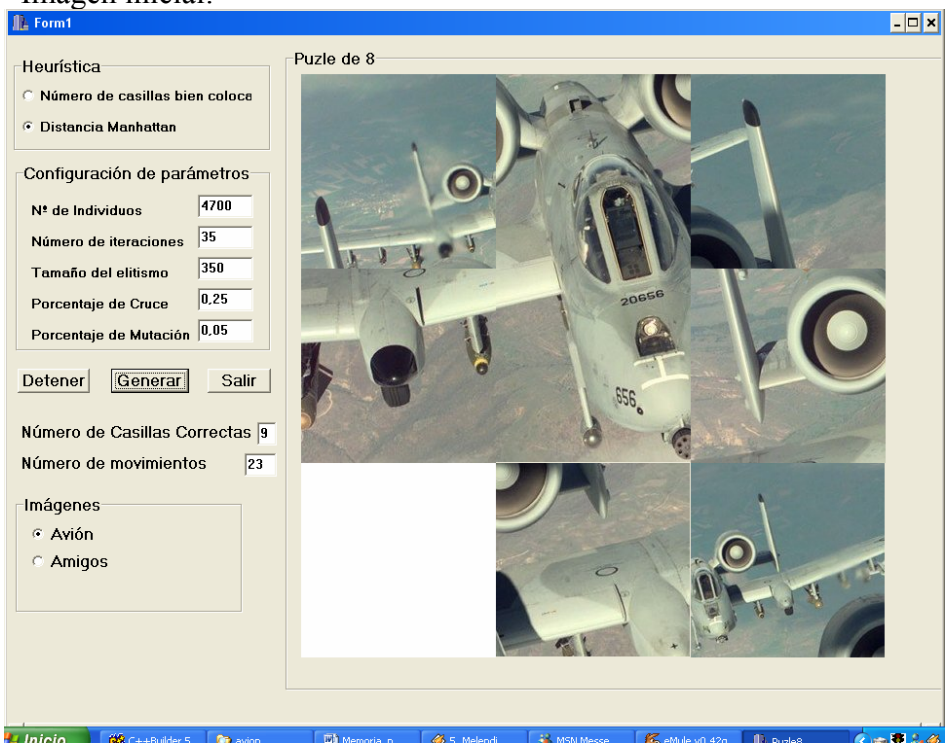


Código para esta primera versión:

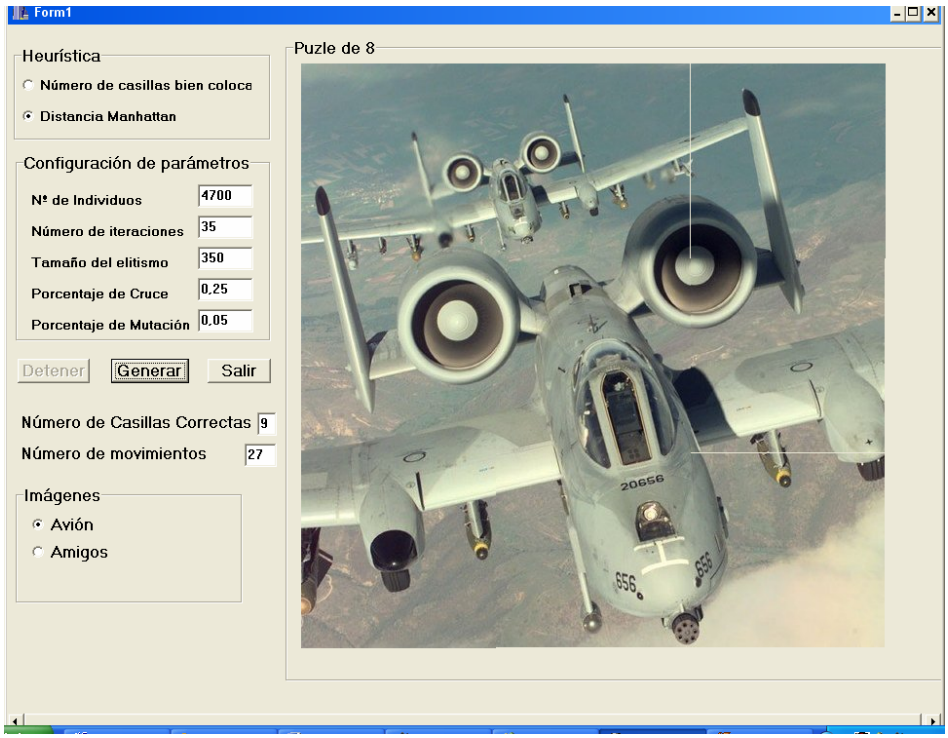
```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
if (mostrar){
   puz_mostr->mover_hueco(ind->get_list_mov()->at(pos));
   mostrar_puzle(puz_mostr);
   pos++;
   if (pos==ind->get_list_mov()->size()){
    mostrar=false;
    delete poblacion;
   }
 }
}

//---------------------------------------------------------------------------

void TForm1::mostrar_puzle(Puzle *puz){
 int i,j;
 for(j=0;j<3;j++)
  for(i=0;i<3;i++)
    StringGrid1->Cells[i][j]=puz->obt_casilla(i,j);
}
```

En la siguiente versión se sustituyó la tabla por una matriz de imágenes (***TImage***). Los números fueron sustituidos por fotos en formato ***jpg*** almacenadas en el disco duro. Para rellenar estas imágenes primero hay que cargar las fotos. Las fotos se cargan en variables de tipo ***TPicture***. Posteriormente se asignan las fotos (***TPicture***) a la matriz de imágenes (***TImage***).

Imagen inicial:

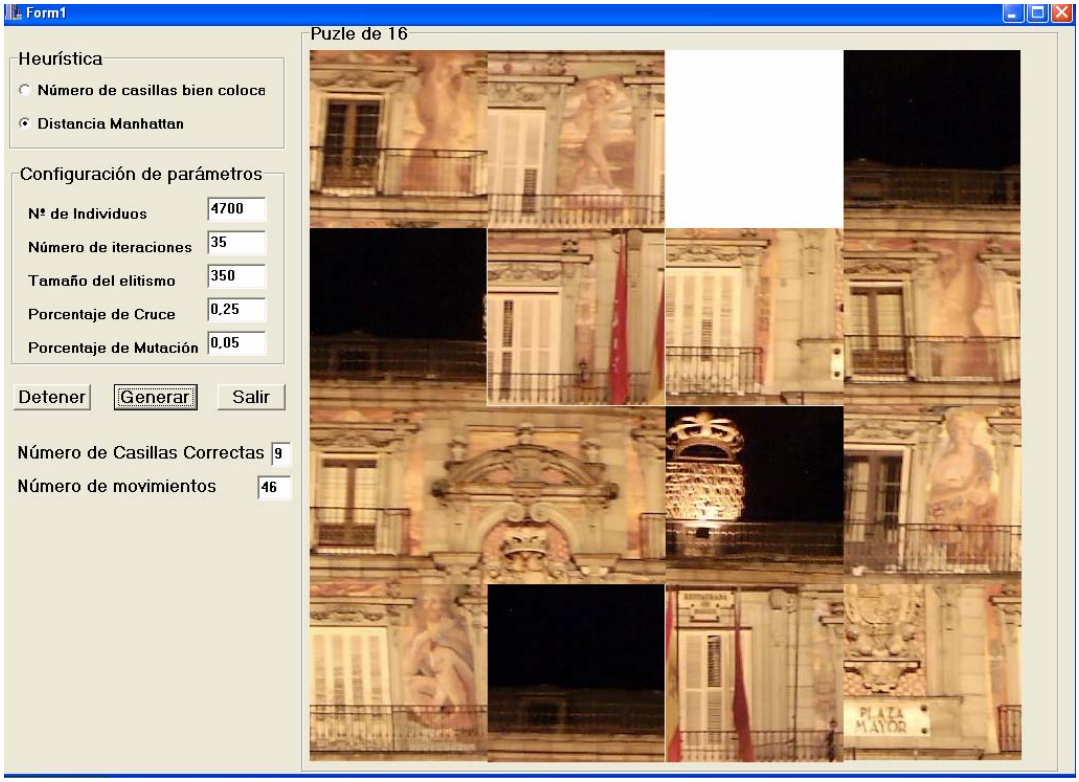


Resultado final:

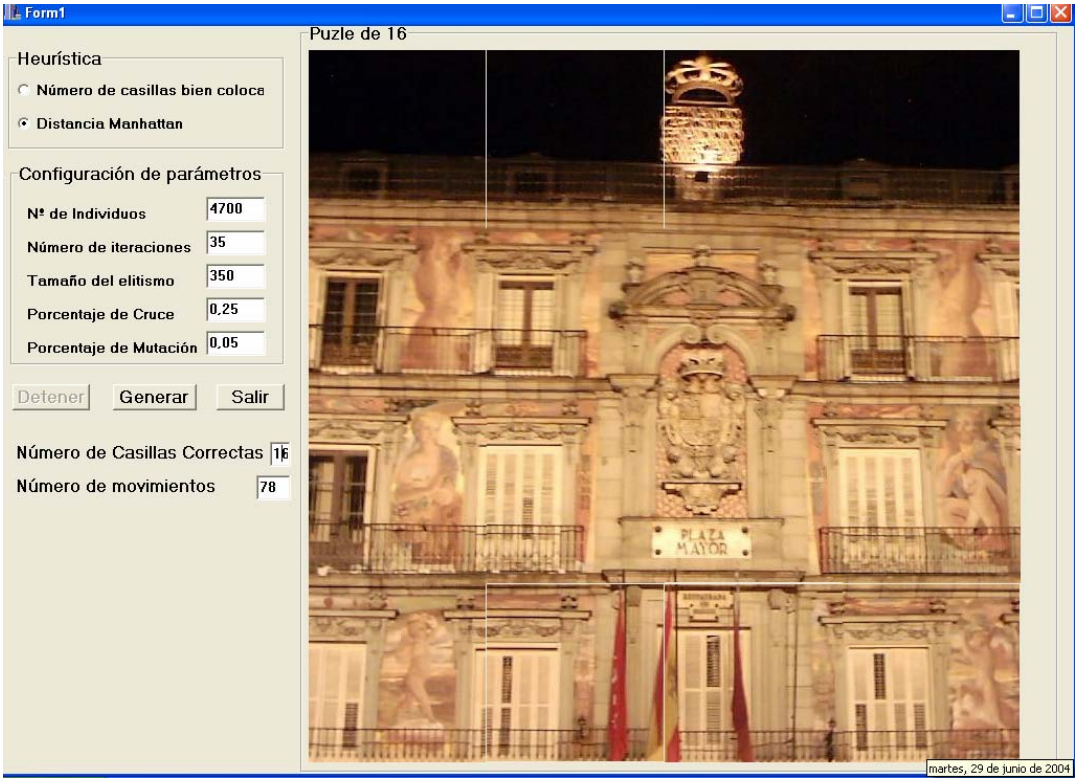


Por último se implementó una ampliación del juego donde se ampliaron las casillas del tablero a 16. Por lo que también se amplio la matriz de imágenes y se tuvo que reducir el tamaño de cada imagen.

Puzzle Inicial:



Puzzle Final:



4. Versiones de la aplicación

Como ya hemos hablado anteriormente a lo largo del desarrollo del juego se han ido creando varias versiones en las cuales se ha mejorando el entorno gráfico. Pero además de estas mejoras gráficas existen dos mejoras fundamentales que han dado lugar a diferentes versiones del juego. Las dos mejoras o ampliaciones más significativas han sido las siguientes: el uso de la distancia Manhatan en la función de adaptación, y la ampliación del puzzle a 16 casillas.

1. Uso de la distancia Manhatan: en una primera versión del juego se usaba como función de aptitud el número de casillas bien colocadas. Esta función de aptitud era bastante pobre, ya que tiene en cuenta pocos aspectos de la situación de las casillas en el puzzle. Para mejorarlo se decidió usar una función de adaptación muy conocida, que es la distancia Manhatan, la cual da mejores resultados con menos individuos, ya que tiene en cuenta la distancia de cada casilla a su posición correcta. La única desventaja de la distancia Manhatan es que su cálculo es mayor que el cálculo del número de casillas bien colocadas, pero esto se compensa al tener que usar menos individuos, y además se consiguen mejores resultados.

2. Ampliación del puzzle a 16 casillas: viendo que el algoritmo funcionaba bastante bien para un puzzle de *3x3* se decidió ampliar el puzzle a uno de *4x4*. A priori parece un proceso sencillo, pero hubo que rehacer muchas funciones de la clase ***Puzzle***, generar un nuevo entorno gráfico reduciendo el tamaño de las imágenes y volver a recortar la foto original para dar lugar a las 16 piezas.

   Con este nuevo puzzle los resultados han sido peores, ya que se necesitan muchísimos más movimientos para encontrar la solución correcta, ya que el espacio de búsqueda y de movimientos es mayor. Para solucionar este problema se intento el uso de un ***cluster***, formado por varias máquinas donde se ejecutara el algoritmo en cada una de ellas y cada cierto número de iteraciones se agruparan los mejores resultados y se volviera a distribuir esta nueva población elitista en cada una de las máquinas. El mecanismo de comunicación entre los hosts integrantes del cluster es el uso del protocolo ***mpi***, consistente en el paso de mensajes.

   Para la implementación del cluster en Windows se instalaron dos herramientas: ***mpich.nt.1.2.5*** *(http://www-unix.mcs.anl.gov/mpi/mpich/mpich-nt/)* y ***nt-mpich1.3.1***.(http://www.lfbs.rwth-aachen.de/content/index.php?ctl_pos=312).

El problema por el cual no se ha usado el ***mpi*** es que los ejecutables tienen que ser generados en el ***Visual C++ de Microsoft***, con unas características especiales, donde se indica que se va a ejecutar en varias máquinas usando el protocolo ***mpi***. El problema de usar el ***Visual C++ de Microsoft*** consiste en que hay que aprender a usar una nueva herramienta, hay que exportar el código del ***Builder C++*** al ***Visual C++***, y los formularios del Builder no son compatibles con los del ***Visual*** con lo que se perdería todo el entorno gráfico.

En el apartado de ***Tecnologías Empleadas***, se explicará en detalle el funcionamiento del ***mpich***.

5. Tecnologías empleadas

En esta práctica se han utilizado varías librerías y clases del C++ Builder interesantes, estas son: ***TTimer***, ***TImage*** y ***TPicture***.

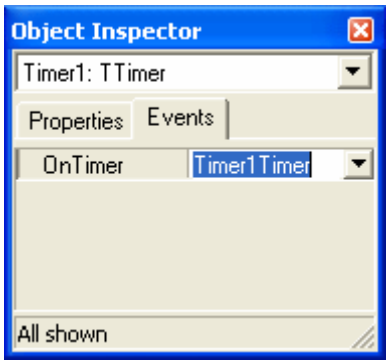
En el caso del ***TTimer*** consiste en una variable que funciona de temporizador. La clase tiene un campo en el cual se indica cada cuántos milisegundos se ejecuta el método ***OnTimer***, esta cantidad se indica en el campo ***Inteval***, en nuestro caso se ejecuta el temporizador cada 800 milisegundos.

El temporizador nos ha servido mostrar los distintos movimientos del hueco, una vez calculada la solución. Cada vez que se ejecuta el temporizador se repinta la matriz de imágenes, mostrándose puzzle actual.

Método ***OnTimer***:

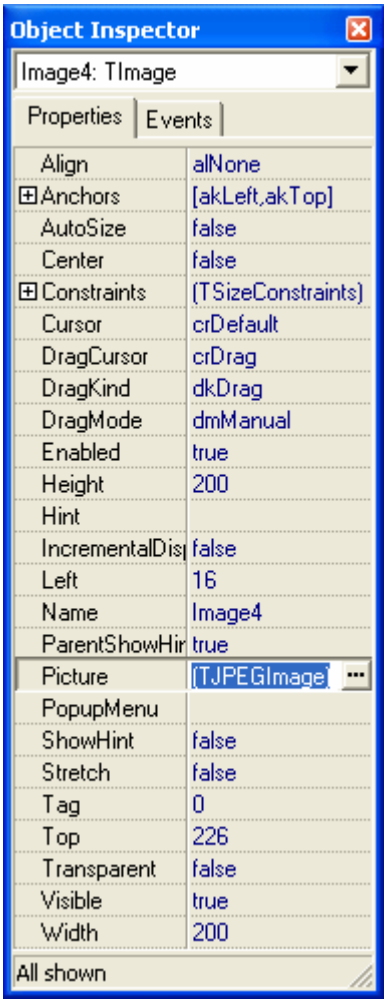
```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
if (mostrar){
  if (pos==ind->get_list_mov()->size()){
    if (num_casillas_correc==9) {
      Image9->Picture=foto8_B;
    mostrar=false;
    delete poblacion;
    Button1->Enabled=false;
    }
   }
   else{
   Button1->Enabled=true;
   puz_mostr->mover_hueco(ind->get_list_mov()->at(pos));
   mostrar_puzle(puz_mostr);
   pos++;
   }
  }
}
```

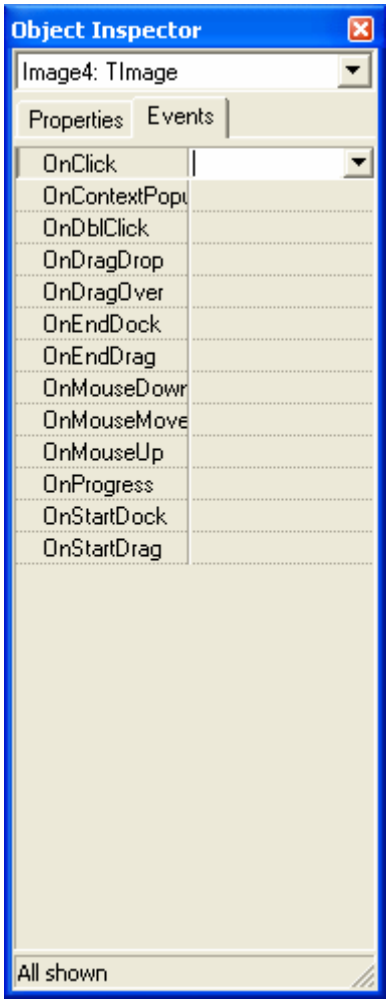
Propiedades y eventos de la clase ***TTimer***:



Para representar la solución hemos usado dos clases del ***C++ Builder***, la clase ***TImage*** y la clase ***TPicture***. La clase ***TImage*** nos ha servido para construir la matriz de imágenes. Esta matriz se inserta en el formulario principal, y servirá para mostrar el puzzle. En los métodos de la clase ***TImage*** podemos especificar el tamaño de la imagen, su posición en el formulario,…

Propiedades y métodos de la clase ***TImage***:





La clase ***TPicture*** nos sirve para cargar las fotografías guardas en el disco duro. Posteriormente las fotos se asignan a las variables de tipo TPicture, mostrándose en el formulario. El formato de fotos que se pueden cargar en las variables de tipo ***TPicture*** es muy diverso: jpg, bmp, gif, … En nuestro caso hemos usado fotografías en formato jpg.

Ejemplo:

```
TImage *imagen;
TPicture *foto0= new TPicture;
foto0->LoadFromFile(PATH+direc+"\\a022_0.jpg");
imagen->Picture=foto0;
```

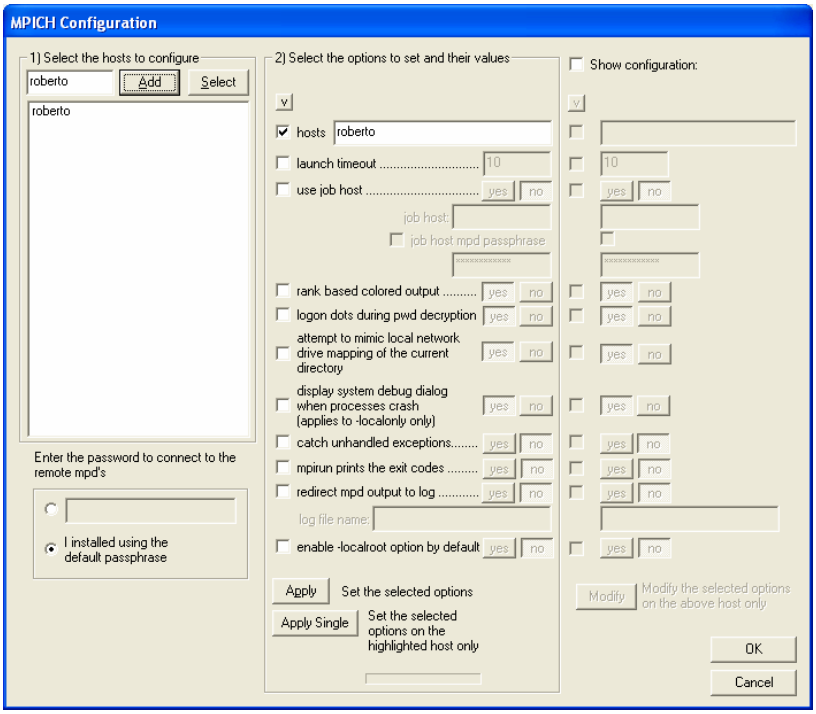
Para poder crear fotos que forman el puzzle ha sido necesario recortar una imagen inicial y generar cada uno de los pedazos. Para ello hemos usado el ***Paint Shop Pro***. El tamaño de cada foto depende del tamaño de cada una de las ***TImage*** que formaban la matriz. En el caso del puzzle de 8 el tamaño de la foto es de 200x200 pixeles, y en el caso del puzzle de 16 casillas el tamaño es de 170 pixeles. EL recorte de cada una de las piezas que forma el puzzle ha sido un trabajo arduo, puesto que el tamaño tenía que ser exacto para que no aparezcan huecos, ni deformaciones, y para que al mostrar el puzzle correcto se consiga la impresión de uniformidad en la imagen.

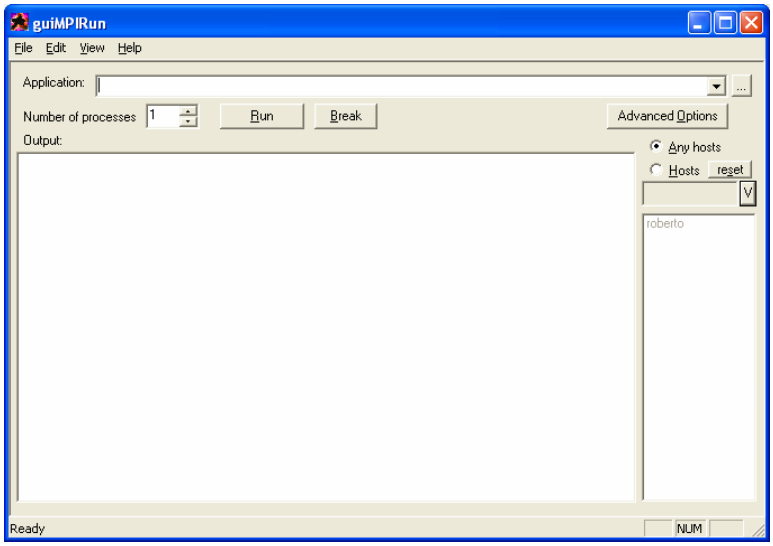
Otra tecnología que se intentó usar fue el uso de protocolo ***mpi***, para poder crear un cluster con varios hosts y ejecutar el algoritmo evolutivo en varios ordenadores, y fusionando los mejores resultados cada cierto número de iteraciones.

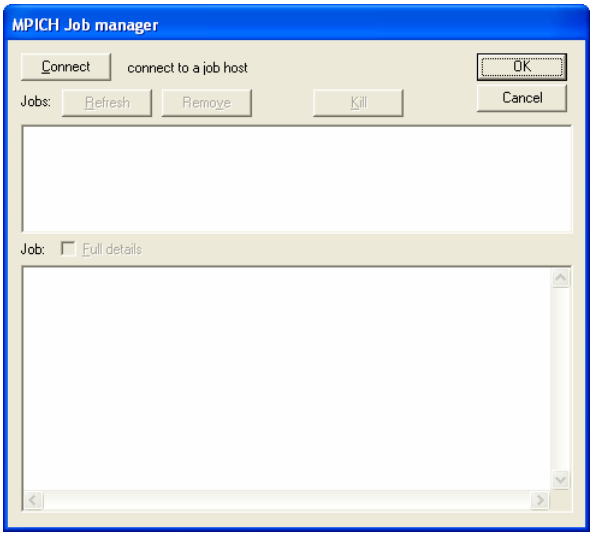
Como hemos comentado antes el problema fundamental es la migración de las clases del ***C++ Builder*** al ***Visual C++***, ya que el ***mpi*** sólo ejecuta ejecutable compilados en el Visual C++, con la inclusión de las librerías del ***mpi*** y con las características especiales para poder ejecutarse en varias máquinas.

El protocolo ***mpi*** consiste en el paso de mensajes entre las máquinas que forman el cluster. Estos mensajes puede ser para pasar información, sincronización, indicar el fin de algún proceso,… Sobre Windows existen dos versiones que simulan el ***mpi***:

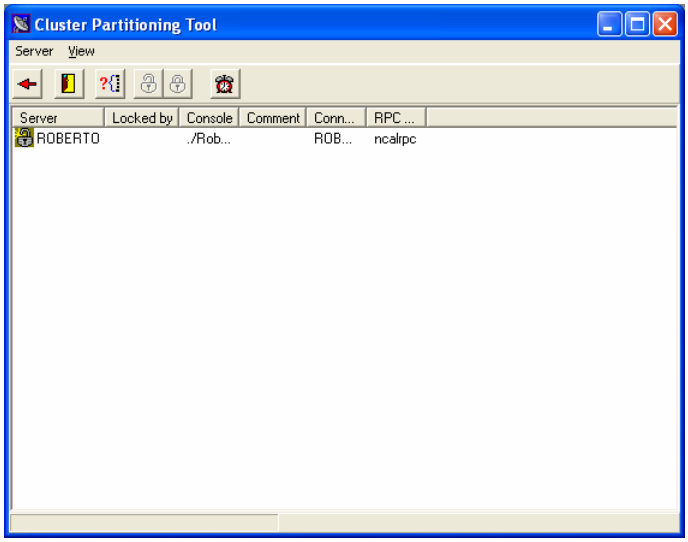
1. ***mpich.nt.1.2.5*** *(http://www-unix.mcs.anl.gov/mpi/mpich/mpich-nt/)*

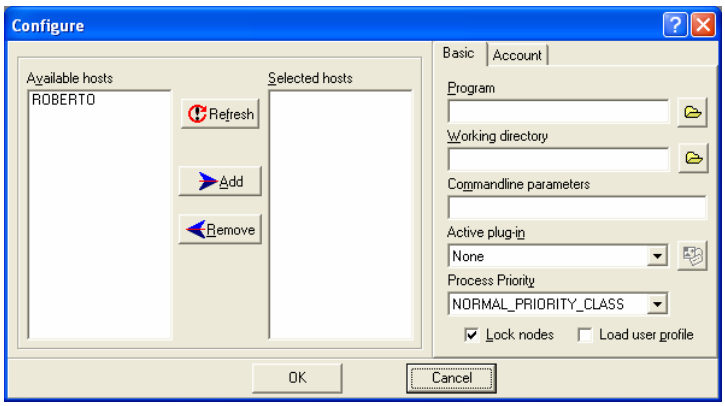






2. ***nt-mpich1.3.1***(http://www.lfbs.rwth-aachen.de/content/index.php?ctl_pos=312).






Rexec Shell
File Window Help
Hosts
Ready

6. Estudio de los parámetros del algoritmo

El estudio de los parámetros en un algoritmo evolutivo es un aspecto esencial, ya que aunque el algoritmo evolutivo este bien implementado puede ser que una mala elección de dichos parámetros haga que se encuentren malas soluciones o que el tiempo de cómputo sea excesivo.

Los parámetros de un algoritmo evolutivo son muy concretos:

- **Tamaño de la población**, parámetro que indica el número de individuos que va a poseer la población. Si el tamaño es excesivo se ralentiza mucho la ejecución del algoritmo, pero si es demasiado pequeño no se generan suficientes individuos para encontrar la solución óptima.
- **Número de iteraciones**, número de veces que se ejecuta el algoritmo evolutivo. Muchas veces se puede parar si vemos que ya hemos encontrado la solución óptima, con lo que ahorramos tiempo de cómputo.
- **Porcentaje de cruce**, indica que porcentaje de la población se va a cruzar. Este factor es muy importante porque el cruce suele consumir mucho tiempo de cómputo y si se cruza muchos individuos se ralentiza el algoritmo y además podríamos perder alguna solución óptima, en el caso de no usar elitismo.
- **Porcentaje de mutación**, indica el porcentaje de individuos de la población que se van a mutar y es un factor menos determinante que el de cruce, ya que suele requerir menos tiempo de CPU, pero también tiene su importancia en la medida en que cada vez que se produce una mutación, hay que recalcular la aptitud, como en el caso del cruce, y el cálculo de la aptitud puede resultar muy costoso.
- **Tamaño de la población elitista**, indica el número de individuos que se conservan en cada iteración, si el número es muy elevado se produce un exceso de uso de memoria, al guardar temporalmente los individuos mejores, y de CPU al tener que fusionar y reordenar la población para elegir los mejores.

Los parámetros para cada una de las versiones varían según la función de aptitud. Si se usa cómo función de aptitud la distancia Manhatan hay que usar un menor número de iteraciones y de individuos para encontrar la solución correcta. Además en el puzzle de 16 casillas es necesario poner un mayor número de iteraciones y de individuos, y se producen peores resultados, ya que el espacio de búsqueda es bastante amplio.

1. Puzzle de 9 casillas, función de aptitud:

    a) Número de casillas bien colocadas:
        - **Tamaño de la población**: 5700 individuos.
        - **Número de iteraciones**: 38 iteraciones.
        - **Porcentaje de cruce**: 20 %
        - **Porcentaje de mutación:** 5 %
        - **Tamaño de la población elitista:** 450 individuos.

    b) Distancia Manhatan:
        - **Tamaño de la población**: 4900 individuos.
        - **Número de iteraciones**: 34 iteraciones.

- **Porcentaje de cruce**: 20 %
- **Porcentaje de mutación:** 5 %
- **Tamaño de la población elitista:** 350 individuos.

2. Puzzle de 16 casillas, función de aptitud:

    a) Número de casillas bien colocadas:
        - **Tamaño de la población**: 15000 individuos.
        - **Número de iteraciones**: 60 iteraciones.
        - **Porcentaje de cruce**: 20 %
        - **Porcentaje de mutación:** 5 %
        - **Tamaño de la población elitista:** 720 individuos.

    b) Distancia Manhatan:
        - **Tamaño de la población**: 9000 individuos.
        - **Número de iteraciones**: 50 iteraciones.
        - **Porcentaje de cruce**: 20 %
        - **Porcentaje de mutación:** 5 %
        - **Tamaño de la población elitista:** 650 individuos.