Algoritmos Genéticos (I)

© 2001 Javier Ventoso Reigosa
www.texelfactory.com

Artículo publicado en la revista ***Sólo Programadores*** Num. 89

Este primer artículo será de introducción a los *Algoritmos Genéticos*, veremos cómo surgieron, su estructura y funcionamiento básico, sus virtudes y limitaciones.

Se le atribuye a John Holland ser el descubridor (junto con sus colegas y alumnos) de los algoritmos genéticos en la década de los sesenta en la Universidad de Michigan, en donde impartía un curso llamado "*Teoría de sistemas adaptativos*". En un principio denominó a esta técnica "*planes reproductivos*" pero más tarde se popularizó con el nombre de "*algoritmos genéticos*".
Lo cierto es que otros destacados personajes ya habían intuido antes que Holland la relación beneficiosa entre el concepto de evolución y la computación.
Charles Babbage en su libro "*Ninth Bridgewater Treatise*" (1813) aprovecha su modelo teórico de computadora para intentar ofrecer una prueba matemática de que Dios tenía programada a la naturaleza para generar a todas las especies; habría creado las leyes que generaban a las especies en lugar de crearlas directamente. Charles Darwin, padre de la teoría de la selección natural (1859), conocía a Babbage y su libro y asistió en varias ocasiones a sus "*parties*" en Londres, y es muy probable que ambos debatiesen sobre el tema.
Después de Babbage, el propio John von Newmann estudió el problema abstracto de la autorreplicación, pensaba que debía existir un código que describiera como se construye un ser vivo y que dicho ser vivo tuviese la facultad de reproducirse. También Alan Turing realizó en 1952 varios trabajos pioneros sobre uno de los problemas básicos de la embriología y morfogénesis: *¿cómo puede la compleja topología de un organismo surgir de la simple topología de una única célula fertilizada desde la que crece?*.

John Holland fue el descubridor de los AGs.

Conceptos Básicos.

Antes de continuar es necesario repasar muy brevemente algunos conceptos que es importante conocer para trabajar con AGs.

Gen: Es la unidad básica de material hereditario.
Cromosoma: Es un cuerpo en forma de filamento constituido principalmente por ADN y proteína.
Genotipo: Es el conjunto de material genético de un organismo.
Fenotipo: Es el conjunto de caracteres observables de un organismo en contraposición con el conjunto de genes que posee.
Locus: Es el lugar que ocupa un gen en un cromosoma.
Para comprenderlo mejor podemos imaginar que si el genotipo de un organismo estuviese escrito en una gran enciclopedia cada tomo sería un cromosoma, cada página de cada tomo sería un gen y el locus de un gen sería su número de página.
En cuanto al genotipo y fenotipo, es importante apuntar que un mismo genotipo puede dar lugar a distintos fenotipos debido a variaciones en el ambiente. Lo cierto es que no todos los genes se manifiestan en el fenotipo, algunos pueden estar enmascarados o neutralizados por otros que los dominan o por el ambiente y aparecer sólo al cabo de varias generaciones.

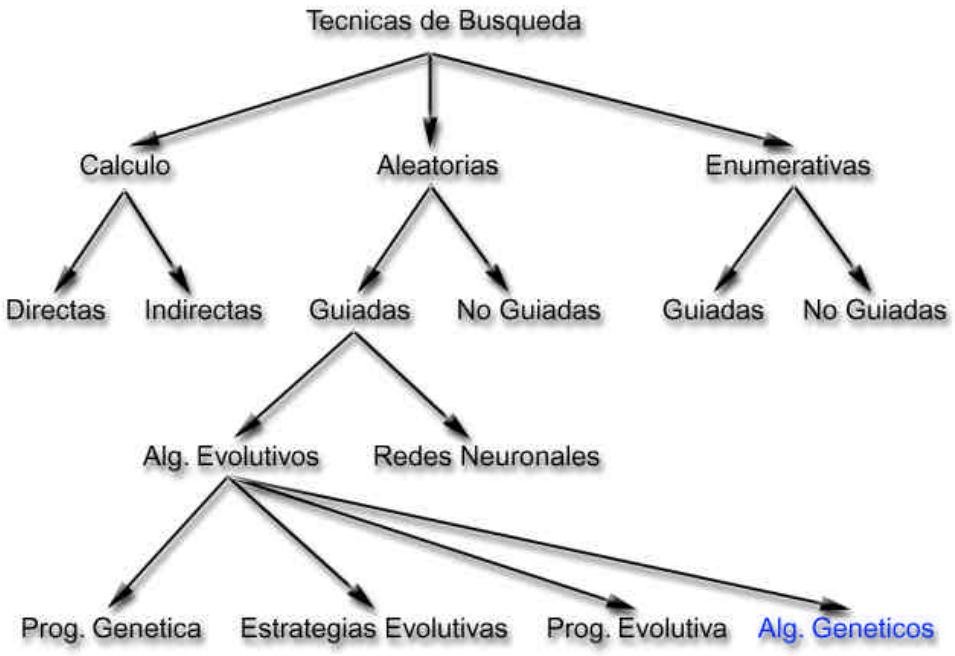


Imagen 1 - Esquema de las principales Técnicas de Búsqueda.

¿Qué es un Algoritmo Genético?

Los algoritmos genéticos (en adelante *AGs*) son métodos adaptativos que se emplean principalmente para la resolución de problemas de búsqueda y optimización. Se enmarcan dentro de la rama de Inteligencia Artificial conocida como *Computación Evolutiva* o *Algoritmos Evolutivos*. Esta rama trata el estudio de los fundamentos y aplicaciones de técnicas heurísticas de búsqueda que emplean los principios de la evolución natural. Existen otras técnicas que junto con los AGs perteneces a esta rama, las más importantes son; *Programación Evolutiva, Estrategias Evolutivas, Programación Genética* y *Sistemas Clasificadores.* Las diferencias entre estas técnicas se basan principalmente en los diferentes operadores que emplean y en las distintas técnicas de selección y reproducción de los individuos de la población.

Los AGs son muy potentes y efectivos sobre todo para aquellos problemas con un gran espacio de búsqueda y para los que no existe una técnica específica para resolverlo.

Podemos aprovecharnos de los AGs para realizar tareas muy complejas que de otra forma sería muy difícil, o imposible, llevarlas a cabo con un enfoque más determinista; problemas basados en variables con interdependencias muy complejas, en ingeniería para optimizaciones numéricas de algunos problemas no lineales que resultan muy difíciles de resolver por métodos analíticos, son muy eficientes para el control de robots, e incluso resultan muy útiles para el ajuste de los *pesos* de las conexiones de redes neuronales, etc.

Son muy efectivos para problemas de búsqueda y optimización.

Funcionamiento.

Básicamente un AG se basa en la utilización de los mismos mecanismos que emplea la naturaleza para la evolución de las especies. Los individuos que mejor se adaptan a su entorno sobreviven sobre los que están peor adaptados. La parte interesante es que cada individuo de la población de un AG representa una posible solución al problema que intentamos resolver, por supuesto para ello tendremos que codificar cada individuo como una representación válida del problema y necesitaremos también crear una función que permita determinar hasta que punto cada individuo evoluciona correctamente para poder actuar en consecuencia.

Los individuos con una mejor adaptación tendrán más posibilidades de reproducirse que aquellos que posean un código genético peor, de esta forma las mejores secuencias genéticas se propagarán en cada nueva generación por medio de los cruces entre los mejores individuos. Si el AG está bien diseñado la población convergerá hacia la solución del problema, o al menos hacia una solución óptima en un plazo de tiempo razonablemente corto.

El esquema básico del funcionamiento de un AG simple es el siguiente:

- Crear la población inicial de forma aleatoria.
- Evaluar a cada uno de los individuos de la población.
- Crear una nueva población a partir del cruzamiento y mutación de los mejores individuos de la población actual.
- Repetir proceso con la nueva población hasta que el algoritmo converja.

Los AGs emplean los mismos mecanismos que la evolución natural.

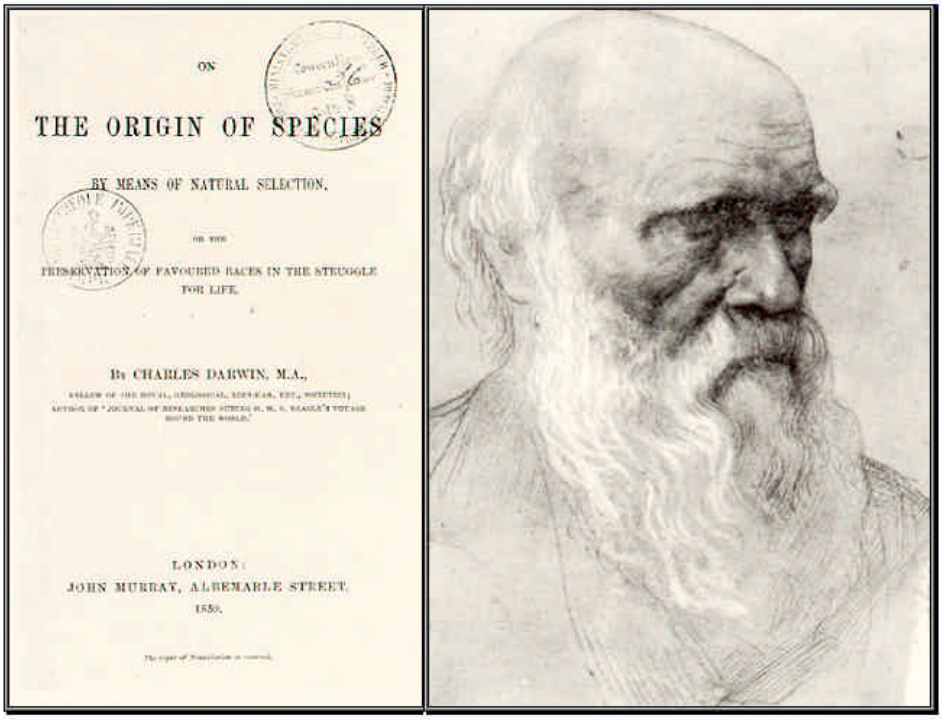


Imagen 2 - Charles Darwin, padre de la teoría de la selección natural.

Función de Evaluación.

Un buen diseño de la función de evaluación (también conocida como *función objetivo* o *función de adaptación*) resulta extremadamente importante para el correcto funcionamiento de un AG. Esta función determina el grado de adaptación o aproximación de cada individuo al problema y por lo tanto permite distinguir a los mejores individuos de los peores. A esta puntuación en función de su proximidad a la mejor solución del problema se le denomina *fitness*.

Veamos un ejemplo; supongamos que hacemos un AG de prueba para que encuentre un punto determinado en un espacio tridimensional. Cada individuo tendrá codificado en su código genético un punto de ese espacio 3D, y la función de evaluación hallaría la inversa de la distancia del punto buscado al punto codificado en cada individuo. De esta forma los mejores individuos serían aquellos que estuviesen más próximos al punto a encontrar (los que tengan un *fitness* mayor). Sobra mencionar que este problema es sólo de ejemplo, ya sabemos de antemano cuál es el punto a buscar, pero nos sirve para ilustrar la utilidad y funcionamiento de la función de evaluación.

Para ciertos problemas, la evaluación de cada individuo puede suponer un coste computacional demasiado elevado, para estos casos se puede implementar una función de evaluación aproximada.

Una población de 25 a 100 individuos es suficiente para muchos problemas.

Selección.

Existen distintos métodos para la selección de los padres que serán cruzados para la creación de nuevos individuos, uno de los más utilizados se denomina *función de selección proporcional a la función de evaluación*. Se basa en que la probabilidad de que un individuo sea seleccionado como padre es proporcional al valor de su función de evaluación. Esto hace que los mejor individuos sean los seleccionados para el proceso de reproducción. Esta técnica puede producir el inconveniente de que la población converja prematuramente hacia un resultado óptimo local. Esto significa que pueden aparecer "superindividuos" muy similares entre si, de forma que la diversidad genética sea bastante pobre y el algoritmo se estanque en una solución buena (óptimo local) pero no la mejor. Trataremos éste y otros problemas más adelante con mayor detalle.

Otro método de selección es el de la *Ruleta*, consiste en asignar una porción de la "ruleta" a cada individuo de forma que el tamaño de cada porción sea proporcional a su *fitness*. Los mejores individuos dispondrán de una porción mayor y por lo tanto de más posibilidades de ser seleccionados.

El método de *Torneo* consiste en hacer competir a los individuos en grupos aleatorios (normalmente parejas), el que tenga el *fitness* más elevado será el ganador. En el caso de competición por parejas se deben realizar dos torneos. Con este método de selección nos aseguramos de que al menos dos copias del mejor individuo de la población actuarán como progenitores para la siguiente generación. Este método tiene evidentes similitudes con el mundo animal, en donde los machos combaten por el control del grupo.

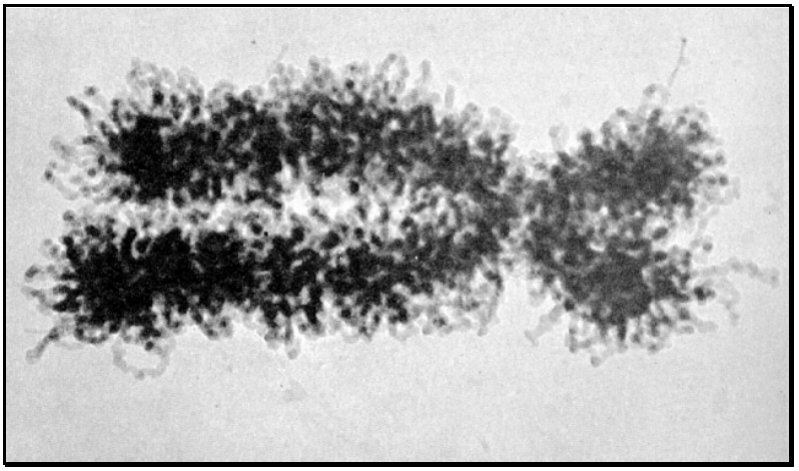


Imagen 3 - Fotografía de un cromosoma.

Reproducción.

Después de seleccionar a los individuos que engendrarán a la siguiente generación, llega el momento de cruzarlos (*crossover*) entre ellos, y al igual que en la naturaleza, el proceso consiste en crear a los descendientes a partir del intercambio de material genético de sus padres. Existen dos técnicas principales; *intercambio a partir de un solo punto de cruce* y a partir de *dos puntos de cruce*. En ambas técnicas se obtienen dos descendientes fruto del cruce.

Para ciertos problemas es necesario emplear una técnica de cruzamiento *especializada* que controle el proceso de intercambio genético evitando que se codifiquen en él soluciones inválidas, un ejemplo de problema que necesita un cruce controlado es el del *problema del viajante* que trataremos más adelante.

Independientemente de la técnica de cruce que se emplee, se suele implementar la reproducción en un AG como un valor porcentual que indica la frecuencia con la que se deben realizar los cruces, y los que no se crucen pasarán como réplicas de si mismos a la siguiente generación (hay que tener también en cuenta que existe la posibilidad de que se produzca una mutación durante la replicación del código).
Esto nos lleva a una técnica muy importante desarrollada hace unos cuantos años conocida como *elitismo*. Consiste en que el mejor individuo de la población permanece inalterable generación tras generación (ni siquiera se le aplica ningún tipo de mutación) hasta que aparece un individuo con un *fitness* mejor que lo sustituye. Su importancia reside en que de esta forma nunca se pierde la mejor solución encontrada hasta el momento.

Los descendientes se crean por intercambio de material genético de sus padres.

Mutación.

Durante la fase de cruce se aplica el operador de mutación; aleatoriamente se modifican uno o más genes de los individuos descendientes de la anterior generación. Esto se realiza para aumentar la diversidad genética que favorece la aparición de individuos con un código genético distinto con la posibilidad de que sea mejor, es especialmente importante la mutación cuando la población, después de un cierto número de generaciones, tiende a converger hacia un óptimo local. No es conveniente abusar del operador de mutación si no queremos que el AG se convierta en un algoritmo de búsqueda al azar.
Igual que sucede en la fase de cruce (reproducción), el proceso de mutación suele implementarse como un valor porcentual y se ha comprobado que el ajuste correcto del porcentaje de mutación es de vital importancia para el correcto funcionamiento del AG. En la mayoría de las implementaciones de AGs se suele emplear unos porcentajes de cruce y mutación estáticos que no varían durante el proceso, sin embargo se han obtenido buenos resultados modificando la probabilidad de mutación a medida que aumenta el número de generaciones.

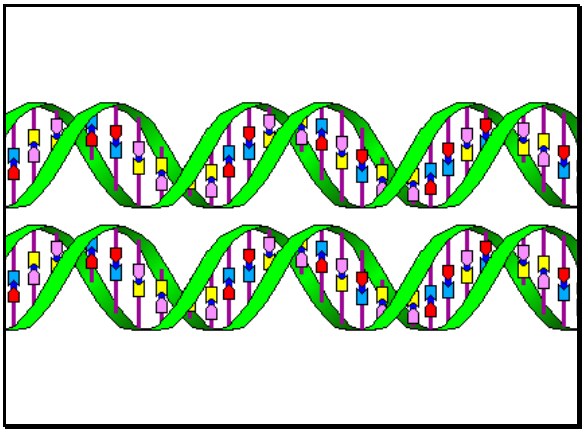


Imagen 4 - Representación de una cadena de ADN.

Población Inicial.

El código genético de la primera generación de individuos debe crearse de forma aleatoria para establecer una probabilidad uniforme. Se puede crear también utilizando alguna técnica heurística pero los pocos estudios que existen sobre este tema indican que el AG tenderá a converger más rápidamente pero con el riesgo de que el algoritmo converja hacia un óptimo local.
En cuanto al tamaño de la población hay que decir que no existen reglas fijas, normalmente una población de 25 a 100 individuos es perfectamente válida para la mayoría de los casos. Si nos excedemos en el tamaño, el coste computacional será muy elevado, y si nos quedamos cortos corremos el riesgo de que el AG no explore correctamente el espacio de búsqueda.

El *elitismo* permite que no se pierda la mejor solución encontrada.

Codificación.

Siempre que implementemos un AG para que resuelva un problema, necesitaremos obligatoriamente que el cromosoma de cada individuo sea una representación de la solución del problema. Volvamos al ejemplo anterior de un AG que busque un punto en un espacio 3D; una implementación perfectamente válida consistiría en que cada individuo estaría formado por tres genes, cada gen sería una variable (entera o flotante, según convenga) y representarían a las coordenadas *X, Y, Z*. De esta forma la estructura de cada individuo (en C o C++) podría ser:

```
typedef struct INDIVIDUO
{
  double fitness;
  int cromosoma[3];
};
```

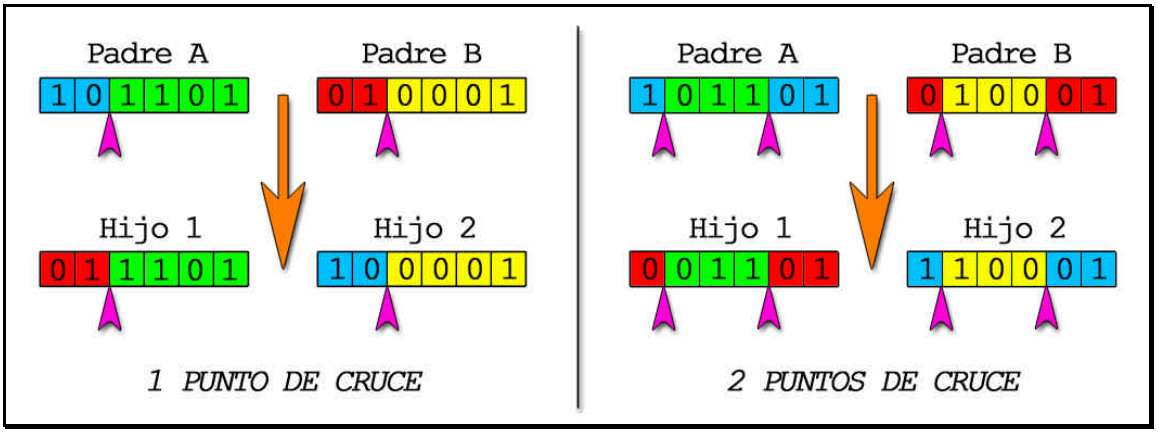
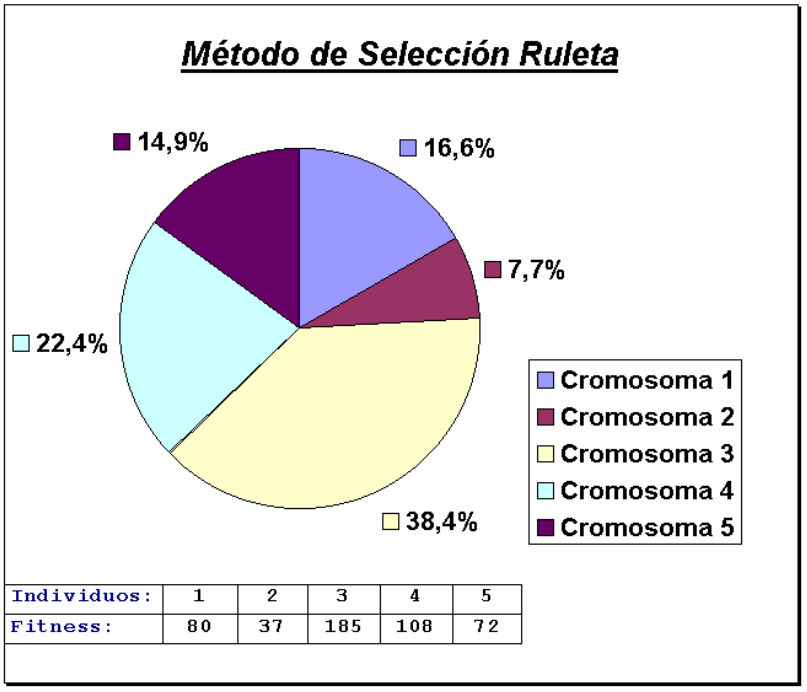


Imagen 5 - Cruzamiento por 1 y 2 puntos de cruce.

Observamos que la estructura contiene la variable *fitness* que almacena el resultado de la evaluación del individuo.
Normalmente el cromosoma es mucho más extenso que en este ejemplo y casi siempre se codifica como un *array*.
Vamos con otro ejemplo más complicado y que realmente es el típico problema para el cual no existen métodos deterministas para su resolución. El problema es el conocido con el nombre de *problema del viajante;* un viajante debe recorrer un determinado número de ciudades pasando una sola vez por cada una de ellas y debe regresar a la ciudad de origen empleando la ruta más corta. A simple vista podría parecer un problema no demasiado complicado, pero hay que tener en cuenta que sólo para 10 ciudades existen 3.628.800 rutas posibles (10!), para 50 ciudades 3e64, para 100 existen 9,3e157... Está claro que el espacio de búsqueda de este problema es gigantesco; se estima que el número total de partículas del universo visible es de 10e80. Si intentásemos resolver este problema explorando una por una todas la rutas entre sólo 20 ciudades, a una velocidad de un millón de rutas por segundo, tardaríamos en explorarlas todas 77.146 años!!!.
En el siguiente artículo haremos un programa que emplee un AG para resolver este problema, pero de momento veamos cómo podríamos codificar el código genético de cada individuo para este caso concreto, una opción sería esta:

```
typedef struct INDIVIDUO
{
  float fitness;
  int *cromosoma;
};
```

El puntero *cromosoma* contendrá la dirección de un *array* generado dinámicamente para cada individuo con los índices a otro *array* en el que se almacenan las posiciones *X* e *Y* de cada una de las ciudades. Por lo tanto cada entero de la lista *cromosoma* corresponde a un gen que representa el índice a una ciudad. Por ejemplo, si un individuo contiene los siguientes genes [9,1,3,5,8,2,0,7,4,6] significa que la ruta que representa es: de la ciudad número 9 viajar hasta la ciudad número 1, de ahí hasta la 3, luego a la 5, 8, 2... para terminar regresando de la 6 a la 9 que es de donde ha partido. Todo esto se comprenderá mejor en el siguiente artículo con el ejemplo práctico que vamos a hacer.



Individuos:	1	2	3	4	5
Fitness:	80	37	185	108	72

Imagen 6 - Selección de individuos por el método de la Ruleta.

En los primeros trabajos de Holland las soluciones se representaban como cadenas binarias y los operadores de cruce y mutación operaban a nivel de *bits*. Para ciertos problemas puede resultar buena esta opción pero para los problemas que exigen una representación más complicada este método resulta bastante engorroso; imaginemos sólo la codificación de las rutas del ejemplo anterior en binario (00001001, 00000001, 00000011...), además nos encontramos con el problema de tener que controlar los *bits* que se pudieran salir del rango durante los procesos de cruce y mutación.
En principio podemos codificar los cromosomas como mejor nos convenga, siempre y cuando apliquemos correctamente los operadores de selección, cruce y mutación y la función de evaluación.

La primera generación de individuos debe crearse al azar.

Convergencia.

Cuando implementamos un AG para que resuelva un problema determinado, se supone que precisamente empleamos un AG porque no existe un método directo para resolverlo. Esto implica que no podemos saber si el AG ha encontrado la mejor solución posible o sólo una solución aproximada (óptimo local), a no ser que en un caso concreto nosotros mismo sepamos cuál es la mejor solución; por ejemplo, en el problema del viajante sabemos sin necesidad de realizar ningún cálculo que la ruta más corta entre una serie de ciudades dispuestas de forma circular es precisamente realizar el viaje entre una y otra ciudad de forma circular. Pero, ¿qué pasa cuando buscamos la mejor ruta entre bastantes ciudades que están situadas de forma totalmente aleatoria?, en ese caso no podemos estar

completamente seguros de que la solución que nos ofrezca un AG sea la mejor de todas las posibles.
La correcta convergencia de un AG es un tema abierto para el que no existen métodos que nos garanticen que un determinado AG convergerá hacia la mejor solución o se quedará atrapado en un óptimo local. Dependiendo del problema, podemos emplear algunas técnicas que nos pueden resultar útiles; una de ellas es dejar que el AG procese un determinado número de generaciones y entonces detenerlo y emplear el cromosoma del mejor individuo de la población como el mejor resultado. Esta técnica es útil cuando el tiempo de proceso es un factor determinante y nos vale con una buena solución aunque no sea la mejor de todas; por ejemplo, en un juego en el se que emplee uno o más AGs para ciertas tareas de inteligencia artificial y no podemos esperar diez minutos a que el AG converja.

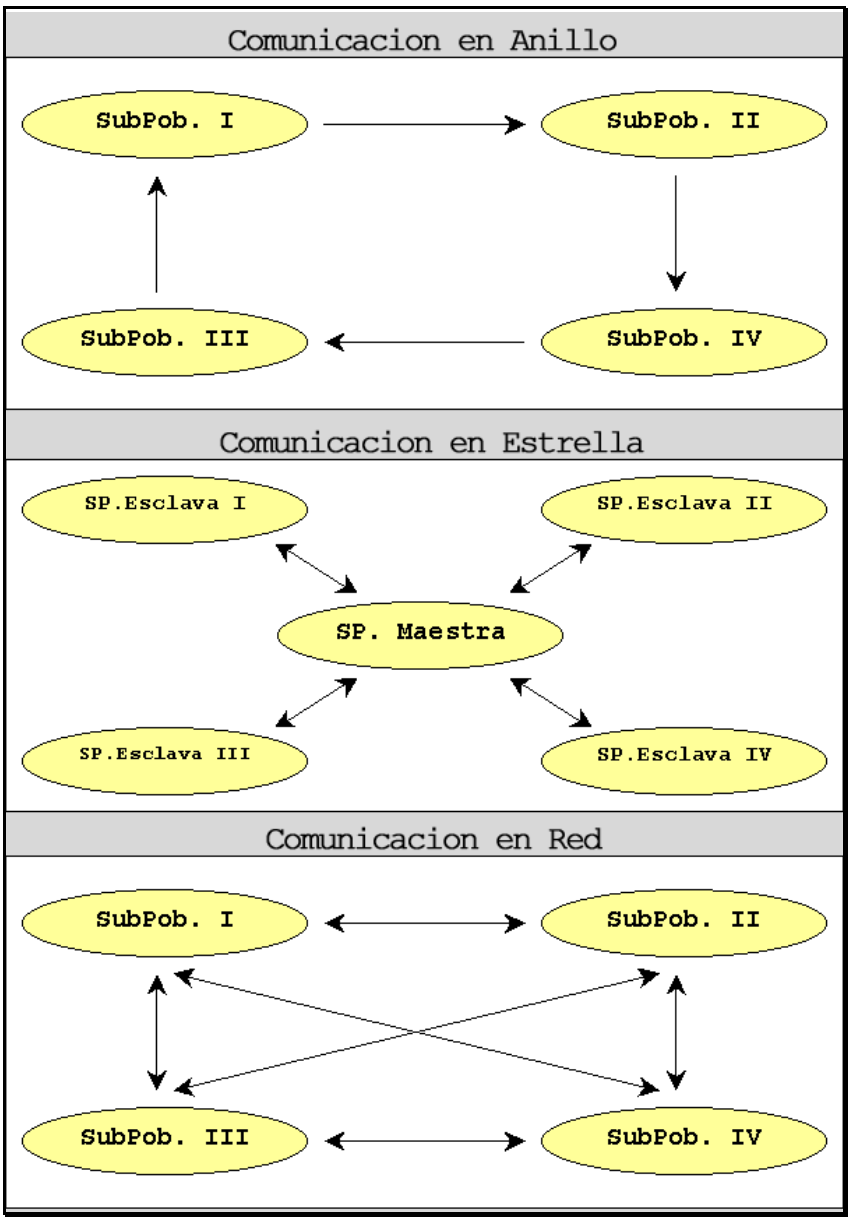


Imagen 7 - Principales configuraciones de Modelos de Islas.

Otro método que podemos utilizar es procesar el AG hasta que la mayoría de la población converja, y sabemos que esto ha sucedido porque la mayor parte de sus individuos tienen un código genético muy similar y por lo tanto un valor de adaptación (*fitness*) próximo.
El funcionamiento ideal de un AG es que converja lo mas rápidamente posible hacia la mejor solución del problema, sin embargo lo normal es que el AG se quede atrapado

durante el proceso (al menos temporalmente) en uno o más óptimos locales. Ya sabemos que no hay ninguna técnica "mágica" que dirija directamente al AG hacia la mejor solución pero por lo menos podemos intentar minimizar los estancamientos en óptimos locales implementando correctamente los operadores de selección, cruce y mutación e incluso algún otro más especializado si resulta beneficioso para la resolución del problema; podemos incluir por ejemplo un operador de *transposición* que intercambie dos genes de un cromosoma, o un operador *creep* que aumente o disminuya el valor de un gen en 1 para conseguir una variación suave y controlada. Es también muy importante ajustar correctamente los porcentajes en los operadores de cruzamiento y mutación, sólo con esto un AG puede pasar de ser penosamente lento a increíblemente rápido. Lo mejor es experimentar con distintas opciones hasta encontrar la fórmula ideal para cada problema.

El operador de mutación aumenta la diversidad genética.

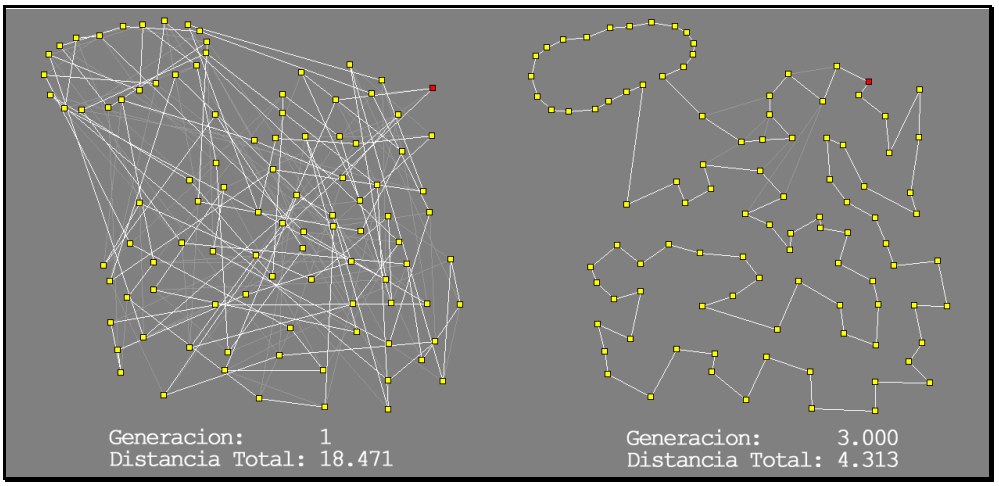


Imagen 8 - Ejemplo del Problema del Viajante con 100 ciudades.

Implementación en Paralelo.

Los AGs son candidatos perfectos para implementarlos en paralelo, a esta técnica se la conoce como *Modelo de Islas*, y consiste en crear varias subpoblaciones procesando en cada una de ellas un AG, cada cierto tiempo se realiza una *migración* entre las subpoblaciones para conseguir un intercambio de información. Se denomina "*isla*" a cada subpoblación y "*migración*" al proceso de intercambio. No hay que esforzarse mucho para ver las similitudes que existen con los procesos migratorios que se producen en la naturaleza.
Al igual que sucede con las tasas de los otros operadores, aquí también es de vital importancia fijar una tasa de migración apropiada para que el AG converja de la mejor forma posible.
Veamos ahora las principales configuraciones que existen en los modelos de islas:

Comunicación en anillo: Cada subpoblación envía a uno o más de sus mejores individuos a la siguiente subpoblación vecina, en esta configuración, la migración se produce en un solo sentido.
Comunicación en estrella: Se selecciona a la subpoblación que posea el mejor *fitness* de media como *subpoblación maestra*, el resto de subpoblaciones serán las *esclavas*. Cada subpoblación esclava envía a uno más de sus mejores individuos a la subpoblación maestra, y la maestra envía a cada subpoblación esclava a uno o más de sus mejores individuos.
Comunicación en red: Cada subpoblación envía a uno o más de sus mejores individuos a todas y cada una de las demás subpoblaciones.
Los AGs son muy fácilmente implementados en paralelo, sólo tenemos que asignar cada uno de ellos a un proceso (*thread*) y controlarlos. Aunque lo ideal para obtener una respuesta lo más rápidamente posible es emplear un equipo con varios procesadores.

La codificación del cromosoma debe representar una posible solución del problema.

Virtudes y Limitaciones.

A lo largo de esta introducción ya hemos visto muchas de las virtudes y alguna que otra limitación de los AGs. La primera ventaja destacable es que un AG trabaja simultáneamente con varias soluciones en lugar de hacerlo de una forma secuencial, y no necesita conocer el problema para intentar solucionarlo, nosotros sólo le aportamos la información y será, principalmente, la función de evaluación la encargada de "guiar" el desarrollo del algoritmo según su aproximación al problema. Otra importante virtud que acabamos de repasar, es que resulta muy sencillo implementar un AG en arquitecturas paralelas. No podemos olvidar tampoco de que son realmente buenos para la exploración de soluciones en problemas con un espacio de búsqueda enorme (el problema del viajante, por ejemplo) y para los que no existe un método determinista, porque de existir, lo más probable es que fuese más rápido y preciso que un AG, todo hay que decirlo.

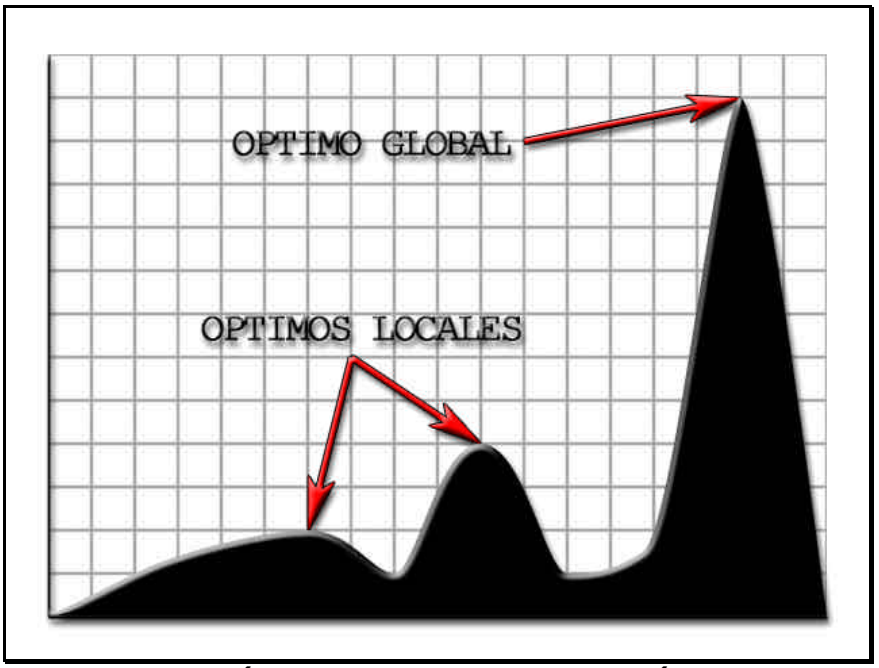


Imagen 9 - Ejemplo de Óptimos Locales (*X=6 y X=10*) y Óptimo Global (*X=16*).

En cuanto a las limitaciones ya sabemos que no tenemos garantías de que la solución obtenida por un AG en un momento determinado sea la mejor solución de todas (óptimo global), corremos el riesgo de que el algoritmo se quede atrapado en un óptimo local y tarde mucho en salir, o incluso puede pasar que no salga nunca. Depende en gran parte de los operadores de selección, cruce, mutación y migración (en el modelo de islas), que empleemos, las tasas que fijemos para cada uno de ellos, el tamaño de la población, cómo inicializamos la población inicial, la función de evaluación, la codificación de la información en el cromosoma, etc.

Los AGs se pueden implementar fácilmente en paralelo.

Conclusión.

Este primer artículo sobre AGs ha sentado unas bases mínimas para poder realizar nuestros propios proyectos con AGs. Un aspecto muy interesante es que es un tema abierto, nada nos impide diseñar e implementar nuestros propios operadores de cruce o mutación, por ejemplo. No hay reglas fijas. Lo comprobaremos de una forma práctica en el siguiente artículo.