# ALGORITMOS GENETICOS

***NATYHELEM GIL LONDOÑO***
*<e-mail:nggil@unal.edu.co>*

27 de Noviembre de 2006

Universidad Nacional de Colombia
Escuela de Estadística
Sede Medellín
2006

# Índice

# 1. GENERALIDADES

## 1.1. INTRODUCCIÓN

### 1.1.1. Reseña Historica

Los algoritmos genéticos (AG), fueron inventados en 1975 por John Holland, de la Universidad de Michigan. Los AG son, simplificando, algoritmos de optimización, es decir, tratan de encontrar la mejor solución a un problema dado entre un conjunto de soluciones posibles. Los mecanismos de los que se valen los AG para llevar a cabo esa búsqueda pueden verse como una metáfora de los procesos de evolución biológica.

John Holland desde pequeño, se preguntaba cómo logra la naturaleza, crear seres cada vez más perfectos. 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.

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 AG.

Por tanto, cuando Holland se enfrentó a los AG, 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 AG, conoció a Holland, y se convirtió en su estudiante. Goldberg era un ingeniero industrial trabajando en diseño de pipelines, y fue uno de los primeros que trató de aplicar los AG a problemas industriales. Aunque Holland trató de disuadirle, porque pensaba que el problema era excesivamente complicado como para aplicarle AG, Goldberg consiguió lo que quería, escribiendo un AG en un ordenador personal Apple II. Estas y otras aplicaciones creadas por estudiantes de Holland convirtieron a los AG en un campo con bases suficientemente aceptables como para celebrar la primera conferencia en 1985, ICGA´85.

### 1.1.2. ¿Que son los Algoritmos Genéticos?

Los AG son métodos adaptativos que pueden usarse para resolver problemas de búsqueda y optimización. Están basados en el proceso genético de los organismos vivos. A lo largo de las generaciones, las poblaciones evolucionan en la naturaleza de acorde con los principios de la selección natural y la supervivencia de los mas fuertes, postulados por Darwin (1859). Por imitación de este proceso, los AG son capaces de ir creando soluciones para problemas del mundo real. La evolución de dichas soluciones hacia valores óptimos del problema depende en buena medida de una adecuada codificación de las mismas.

En la naturaleza los individuos de una población compiten entre si en la búsqueda de recursos tales como comida, agua y refugio. Incluso los miembros de una misma especie compiten a menudo en la búsqueda de un compañero. Aquellos individuos que tienen mas éxito en sobrevivir y en atraer compañeros tienen mayor probabilidad de generar un gran numero de descendientes. Por el contrario individuos poco dotados producirán un menor numero de descendientes. Esto significa que los genes de los individuos mejor adaptados se propagaran en sucesivas generaciones hacia un número de individuos creciente. La combinación de buenas características provenientes de diferentes ancestros, puede a veces producir descendientes "superindividuos", cuya adaptación es mucho mayor que la de cualquiera de sus ancestros. De esta manera, las especies evolucionan logrando unas características cada vez mejor adaptadas al entorno en el que viven.

El poder de los AG proviene del hecho de que se trata de una técnica robusta, y pueden tratar con éxito una gran variedad de problemas provenientes de diferentes áreas, incluyendo aquellos en los que otros métodos encuentran dificultades. Si bien no se garantiza que el AG encuentre la solución optima del problema, existe

evidencia empírica de que se encuentran soluciones de un nivel aceptable, en un tiempo competitivo con el resto de algoritmos de optimización combinatoria.

En el caso de que existan técnicas especializadas para resolver un determinado problema, lo mas probable es que superen al AG, tanto en rapidez como en eficacia. El gran campo de aplicación de los AG se relaciona con aquellos problemas para los cuales no existen técnicas especializadas. Incluso en el caso en que dichas técnicas existan, y funcionen bien, pueden efectuarse mejoras de las mismas hibridandolas con los AG.

### 1.1.3. ¿Por qué utilizar Algoritmos Genéticos en la Optimización?

La razón del creciente interés por los AG es que estos son un método global y robusto de búsqueda de las soluciones de problemas. La principal ventaja de estas características es el equilibrio alcanzado entre la eficiencia y eficacia para resolver diferentes y muy complejos problemas de grandes dimensiones.

Lo que aventaja a los AG frente a otros algoritmos tradicionales de búsqueda es que se diferencian de estos en los siguientes aspectos:

- Trabajan con una codificación de un conjunto de parámetros, no con los parámetros mismos.
- Trabajan con un conjunto de puntos, no con un único punto y su entorno (su técnica de búsqueda es global.) Utilizan un subconjunto del espacio total, para obtener información sobre el universo de búsqueda, a través de las evaluaciones de la función a optimizar. Esas evaluaciones se emplean de forma eficiente para clasificar los subconjuntos de acuerdo con su idoneidad.
- No necesitan conocimientos específicos sobre el problema a resolver; es decir, no están sujetos a restricciones. Por ejemplo, se pueden aplicar a funciones no continuas, lo cual les abre un amplio campo de aplicaciones que no podrían ser tratadas por los métodos tradicionales.
- Utilizan operadores probabilísticos, en vez de los tipicos operadores determinísticos de las técnicas tradicionales.
- Resulta sumamente fácil ejecutarlos en las modernas arquitecturas masivas en paralelo.

- Cuando se usan para problemas de optimización, resultan menos afectados por los máximos locales que las técnicas tradicionales (i.e., son métodos robustos).

Ahora bien; un esquema del funcionamiento general de un algoritmo genético podría ser el siguiente:
*Algoritmo Genetico* :
- Generar una población inicial.
- Iterar hasta un criterio de parada.
- Evaluar cada individuo de la población.
- Seleccionar los progenitores.
- Aplicar el operador de cruce y mutación a estos progenitores.
- Incluir la nueva descendencia para formar una nueva generación.

### 1.1.4. Ventajas

- El primer y más importante punto es que los AG son intrínsecamente paralelos. La mayoría de los otros algoritmos son en serie y sólo pueden explorar el espacio de soluciones hacia una solución en una dirección al mismo tiempo, y si la solución que descubren resulta subóptima, no se puede hacer otra cosa que abandonar todo el trabajo hecho y empezar de nuevo. Sin embargo, ya que los AG tienen descendencia múltiple, pueden explorar el espacio de soluciones en múltiples direcciones a la vez. Si un camino resulta ser un callejón sin salida, pueden eliminarlo fácilmente y continuar el trabajo en avenidas más prometedoras, dándoles una mayor probabilidad en cada ejecución de encontrar la solución.

  Sin embargo, la ventaja del paralelismo va más allá de esto. Considere lo siguiente: todas las cadenas binarias (cadenas de ceros y unos) de 8 dígitos forman un espacio de búsqueda, que puede representarse como ******** (donde * significa “o 0 o 1”). La cadena 01101010 es un miembro de este espacio. Sin embargo, también es un miembro del espacio 0*******, del espacio 01******, del espacio 0******0, del espacio 0*1*1*1*, del espacio 10*01**0, etc. Evaluando la aptitud de esta cadena particular, un AG estaría sondeando cada uno de los espacios a los que pertenece. Tras muchas evaluaciones, iría obteniendo un valor cada vez más preciso de la aptitud media de cada uno de estos espacios, cada uno de los cuales contiene muchos miembros. Por tanto, un AG que evalúe explícitamente un número pe-

queño de individuos está evaluando implícitamente un grupo de individuos mucho más grande -de la misma manera que un encuestador que le hace preguntas a un cierto miembro de un grupo étnico, religioso o social espera aprender algo acerca de las opiniones de todos los miembros de ese grupo, y por tanto puede predecir con fiabilidad la opinión nacional sondeando sólo un pequeño porcentaje de la población. De la misma manera, el AG puede dirigirse hacia el espacio con los individuos más aptos y encontrar el mejor de ese grupo. En el contexto de los algoritmos evolutivos, esto se conoce como teorema del esquema, y es la ventaja principal de los AG sobre otros métodos de resolución de problemas.

- Debido al paralelismo que les permite evaluar implícitamente muchos esquemas a la vez, los AG funcionan particularmente bien resolviendo problemas cuyo espacio de soluciones potenciales es realmente grande -demasiado vasto para hacer una búsqueda exhaustiva en un tiempo razonable. La mayoría de los problemas que caen en esta categoría se conocen como "no lineales". En un problema lineal, la aptitud de cada componente es independiente, por lo que cualquier mejora en alguna parte dará como resultado una mejora en el sistema completo. No es necesario decir que hay pocos problemas como éste en la vida real. La no linealidad es la norma, donde cambiar un componente puede tener efectos en cadena en todo el sistema, y donde cambios múltiples que, individualmente, son perjudiciales, en combinación pueden conducir hacia mejoras en la aptitud mucho mayores. La no linealidad produce una explosión combinatoria: el espacio de cadenas binarias de 1.000 dígitos puede examinarse exhaustivamente evaluando sólo 2.000 posibilidades si el problema es lineal, mientras que si no es lineal, una búsqueda exhaustiva requiere evaluar 21.000 posibilidades -un número que, escrito, ocuparía más de 300 dígitos.

  Afortunadamente, el paralelismo implícito de los AG les permite superar incluso este enorme número de posibilidades, y encontrar con éxito resultados óptimos o muy buenos en un corto periodo de tiempo, tras muestrear directamente sólo regiones pequeñas del vasto paisaje adaptativo. Por ejemplo, un AG desarrollado en común por ingenieros de General Electric y el Rensselaer Polytechnic Institute produjo el diseño de la turbina de un motor a reacción de altas prestaciones que era tres veces mejor que la configuración diseñada por humanos, y un 50 % mejor que una configuración diseñada por un sistema experto que recorrió con éxito un espacio de soluciones que contenía más de 10.387 posibilidades. Los métodos convencionales para di-

señar estas turbinas son una parte fundamental de proyectos de ingeniería que pueden durar hasta cinco años y costar más de 2.000 millones de dólares; el AG descubrió esta solución en dos días, en una estación de trabajo de escritorio típica en ingeniería.

- Otra ventaja notable de los AG es que se desenvuelven bien en problemas con un paisaje adaptativo complejo -aquéllos en los que la función objetivo es discontinua, ruidosa, cambia con el tiempo, o tiene muchos óptimos locales. La mayoría de los problemas prácticos tienen un espacio de soluciones enorme, imposible de explorar exhaustivamente; el reto se convierte entonces en cómo evitar los óptimos locales -soluciones que son mejores que todas las que son similares a ella, pero que no son mejores que otras soluciones distintas situadas en algún otro lugar del espacio de soluciones. Muchos algoritmos de búsqueda pueden quedar atrapados en los óptimos locales: si llegan a lo alto de una colina del paisaje adaptativo, descubrirán que no existen soluciones mejores en las cercanías y concluirán que han alcanzado la mejor de todas, aunque existan picos más altos en algún otro lugar del mapa.

  Los algoritmos evolutivos, por otro lado, han demostrado su efectividad al escapar de los óptimos locales y descubrir el óptimo global incluso en paisajes adaptativos muy escabrosos y complejos. (Debe decirse que, en la realidad, a menudo no hay manera de decir si una cierta solución a un problema es el óptimo global o sólo un óptimo local muy alto. Sin embargo, aunque un AG no devuelva siempre una solución perfecta y demostrable a un problema, casi siempre puede devolver al menos una muy buena solución). Todos los cuatro componentes principales de los AG -paralelismo, selección, mutación y cruzamiento- trabajan juntos para conseguir esto. Al principio, el AG genera una población inicial diversa, lanzando una "red" sobre el paisaje adaptativo. (Koza 2003[42], p. 506) compara esto con un ejército de paracaidistas cayendo sobre el paisaje del espacio de búsqueda de un problema, cada uno de ellos con órdenes de buscar el pico más alto). Pequeñas mutaciones permiten a cada individuo explorar sus proximidades, mientras que la selección enfoca el progreso, guiando a la descendencia del algoritmo cuesta arriba hacia zonas más prometedoras del espacio de soluciones.

  Sin embargo, el cruzamiento es el elemento clave que distingue a los AG de los otros métodos como los trepacolinas y el recocido simulado. Sin el

cruzamiento, cada solución individual va por su cuenta, explorando el espacio de búsqueda en sus inmediaciones sin referencia de lo que el resto de individuos puedan haber descubierto. Sin embargo, con el cruzamiento en juego, hay una transferencia de información entre los candidatos prósperos -los individuos pueden beneficiarse de lo que otros han aprendido, y los esquemas pueden mezclarse y combinarse, con el potencial de producir una descendencia que tenga las virtudes de sus dos padres y ninguna de sus debilidades. Este punto está ilustrado en Koza et al. 1999[41], p. 486, donde los autores analizan el problema de sintetizar un filtro de paso bajo utilizando programación genética. En una generación se seleccionaron dos circuitos progenitores para llevar a cabo el cruzamiento; un padre tenía una buena topología (componentes como inductores y condensadores colocados en el sitio correcto) pero malos tamaños (valores demasiado bajos de inductancia y capacidad para los componentes). El otro padre tenía mala topología pero buenos tamaños. El resultado de aparearlos mediante cruzamiento fue una descendencia con la buena topología de un padre y los buenos tamaños del otro, dando como resultado una mejora sustancial de la aptitud sobre sus dos padres.

El problema de encontrar el óptimo global en un espacio con muchos óptimos locales también se conoce como el dilema de la exploración versus explotación, "un problema clásico de todos los sistemas que pueden adaptarse y aprender". Una vez que un algoritmo (o un diseñador humano) ha encontrado una estrategia para resolver problemas que parece funcionar satisfactoriamente, ¿debería centrarse en hacer el mejor uso de esa estrategia, o buscar otras? Abandonar una estrategia de probada solvencia para buscar otras nuevas casi garantiza que supondrá una pérdida y degradación del rendimiento, al menos a corto plazo. Pero si uno se queda con una estrategia particular excluyendo a todas las demás, corre el riesgo de no descubrir estrategias mejores que existen pero no se han encontrado. De nuevo, los AG han demostrado ser muy buenos en dar con este equilibrio y descubrir buenas soluciones en un tiempo y esfuerzo computacional razonables.

- Otra área en el que destacan los AG es su habilidad para manipular muchos parámetros simultáneamente. Muchos problemas de la vida real no pueden definirse en términos de un único valor que hay que minimizar o maximizar, sino que deben expresarse en términos de múltiples objetivos, a menudo involucrando contrapartidas: uno sólo puede mejorar a expensas de otro. Los AG son muy buenos resolviendo estos problemas: en particular, su uso del

paralelismo les permite producir múltiples soluciones, igualmente buenas, al mismo problema, donde posiblemente una solución candidata optimiza un parámetro y otra candidata optimiza uno distinto y luego un supervisor humano puede seleccionar una de esas candidatas para su utilización. Si una solución particular a un problema con múltiples objetivos optimiza un parámetro hasta el punto en el que ese parámetro no puede mejorarse más sin causar una correspondiente pérdida de calidad en algún otro parámetro, esa solución se llama óptimo paretiano o no dominada.

- Finalmente, una de las cualidades de los AG que, a primera vista, puede parecer un desastre, resulta ser una de sus ventajas: a saber, los AG no saben nada de los problemas que deben resolver. En lugar de utilizar información específica conocida a priori para guiar cada paso y realizar cambios con un ojo puesto en el mejoramiento, como hacen los diseñadores humanos, son "relojeros ciegos"; realizan cambios aleatorios en sus soluciones candidatas y luego utilizan la función objetivo para determinar si esos cambios producen una mejora.

  La virtud de esta técnica es que permite a los AG comenzar con una mente abierta, por así decirlo. Como sus decisiones están basadas en la aleatoriedad, todos los caminos de búsqueda posibles están abiertos teóricamente a un AG; en contraste, cualquier estrategia de resolución de problemas que dependa de un conocimiento previo, debe inevitablemente comenzar descartando muchos caminos a priori, perdiendo así cualquier solución novedosa que pueda existir. Los AG, al carecer de ideas preconcebidas basadas en creencias establecidas sobre "cómo deben hacerse las cosas" o sobre lo que "de ninguna manera podría funcionar", los AG no tienen este problema. De manera similar, cualquier técnica que dependa de conocimiento previo fracasará cuando no esté disponible tal conocimiento, pero, de nuevo, los AG no se ven afectados negativamente por la ignorancia. Mediante sus componentes de paralelismo, cruzamiento y mutación, pueden viajar extensamente por el paisaje adaptativo, explorando regiones que algoritmos producidos con inteligencia podrían no haber tenido en cuenta, y revelando potencialmente soluciones de asombrosa e inesperada creatividad que podrían no habérseles ocurrido nunca a los diseñadores humanos. Un ejemplo muy gráfico de esto es el redescubrimiento, mediante la programación genética, del concepto de retroalimentación negativa -un principio crucial para muchos componentes electrónicos importantes de hoy en día, pero un concepto que, cuando fue descubierto en primera instancia, se le denegó

una patente de nueve años porque el concepto era demasiado contrario a las creencias establecidas. Por supuesto, los algoritmos evolutivos no están enterados ni preocupados de si una solución va en contra de las creencias establecidas -sólo de si funciona.

### 1.1.5. Desventajas

Aunque los AG han demostrado su eficiencia y potencia como estrategia de resolución de problemas, no son la panacea. Los AG tienen ciertas limitaciones; sin embargo, se demostrará que todas ellas pueden superarse y que ninguna de ellas afecta a la validez de la evolución biológica.

- La primera y más importante consideración al crear un AG es definir una representación del problema. El lenguaje utilizado para especificar soluciones candidatas debe ser robusto; es decir, debe ser capaz de tolerar cambios aleatorios que no produzcan constantemente errores fatales o resultados sin sentido.

  Hay dos maneras principales para conseguir esto. La primera, utilizada por la mayoría de los AG, es definir a los individuos como listas de números -binarios, enteros o reales- donde cada número representa algún aspecto de la solución candidata. Si los individuos son cadenas binarias, un 0 o 1 podría significar la ausencia o presencia de una cierta característica. Si son listas de números, estos números podrían representar muchas cosas distintas: los pesos de las conexiones en una red neuronal, el orden de las ciudades visitadas en un recorrido dado, la situación espacial de componentes electrónicos, los valores con los que se alimenta a un controlador, los ángulos de torsión de los enlaces péptidos de una proteína, etc. Así, la mutación implica cambiar estos números, cambiar bits o sumar o restar valores aleatorios. En este caso, el propio código del programa no cambia; el código es lo que dirige la simulación y hace un seguimiento de los individuos, evaluando sus aptitudes y quizá asegurando que sólo se producen valores realistas y posibles para el problema dado.

  En otro método, la programación genética, el propio código del programa sí cambia. Como ya se dijo en la sección “Métodos de representación”, la PG representa a los individuos como árboles de código ejecutables que pueden mutar cambiando o intercambiando subárboles. Ambos métodos producen

representaciones robustas ante la mutación, y pueden representar muchos tipos diferentes de problemas y, como se dice en la sección “Algunos ejemplos específicos”, ambas han tenido un éxito considerable.

El problema de representar a las soluciones candidatas de manera robusta no surge en la naturaleza, porque el método de representación utilizado por la evolución, a saber, el código genético, es inherentemente robusto: con muy pocas excepciones, como una cadena de codones de parada, no existe una secuencia de bases de ADN que no pueda traducirse en una proteína. Por lo tanto, virtualmente, cualquier cambio en los genes de un individuo siempre producirá un resultado inteligible, y por tanto las mutaciones en la evolución tienen mayor probabilidad de producir una mejora. Esto entra en contraste con los lenguajes creados por el hombre como el inglés, donde el número de palabras con significado es pequeño comparado con el número total de formas en las que se pueden combinar las letras del alfabeto, y por tanto, es probable que un cambio aleatorio en una frase en inglés produzca un sinsentido.

- El problema de cómo escribir la función objetivo debe considerarse cuidadosamente para que se pueda alcanzar una mayor aptitud y verdaderamente signifique una solución mejor para el problema dado. Si se elige mal una función objetivo o se define de manera inexacta, puede que el AG sea incapaz de encontrar una solución al problema, o puede acabar resolviendo el problema equivocado. (Esta última situación se describe a veces como la tendencia del AG a “engañar”, aunque en realidad lo que está pasando es que el AG está haciendo lo que se le pidió hacer, no lo que sus creadores pretendían que hiciera). Como por ejemplo: unos investigadores utilizaron un algoritmo evolutivo en conjunción con una serie de chips reprogramables, haciendo que la función objetivo recompensara al circuito en evolución por dar como salida una señal oscilatoria. Al final del experimento, se producía efectivamente una señal oscilatoria -pero en lugar de actuar como un oscilador, como pretendían los investigadores, ¡descubrieron que el circuito se había convertido en un receptor de radio que estaba recibiendo y retransmitiendo una señal oscilatoria de un componente electrónico cercano!

  Sin embargo, esto no es un problema en la naturaleza. En el laboratorio de la evolución biológica, sólo hay una función objetivo que es igual para todos los seres vivos -la carrera por sobrevivir y reproducirse, sin importar qué adaptaciones hagan esto posible. Los organismos que se reproducen

con más abundancia que sus competidores están más adaptados; los que fracasan en reproducirse no están adaptados.

- Además de elegir bien la función objetivo, también deben elegirse cuidadosamente los otros parámetros de un AG -el tamaño de la población, el ritmo de mutación y cruzamiento, el tipo y fuerza de la selección. Si el tamaño de la población es demasiado pequeño, puede que el AG no explore suficientemente el espacio de soluciones para encontrar buenas soluciones consistentemente. Si el ritmo de cambio genético es demasiado alto o el sistema de selección se escoge inadecuadamente, puede alterarse el desarrollo de esquemas beneficiosos y la población puede entrar en catástrofe de errores, al cambiar demasiado rápido para que la selección llegue a producir convergencia.

  Los seres vivos también se enfrentan a dificultades similares, y la evolución se ha encargado de ellas. Es cierto que si el tamaño de una población cae hacia un valor muy bajo, los ritmos de mutación son muy altos o la presión selectiva es demasiado fuerte (una situación así podría ser resultado de un cambio ambiental drástico), entonces la especie puede extinguirse. La solución ha sido “la evolución de la evolutividad” -las adaptaciones que alteran la habilidad de una especie para adaptarse. Un ejemplo. La mayoría de los seres vivos han evolucionado una elaborada maquinaria celular que comprueba y corrigue errores durante el proceso de replicación del ADN, manteniendo su ritmo de mutación a unos niveles aceptablemente bajos; a la inversa, en tiempos de fuerte presión ambiental, algunas especies de bacterias entran en un estado de hipermutación en el que el ritmo de errores en la replicación del ADN aumenta bruscamente, aumentando la probabilidad de que se descubrirá una mutación compensatoria. Por supuesto, no pueden eludirse todas las catástrofes, pero la enorme diversidad y las adaptaciones altamente complejas de los seres vivos actuales muestran que, en general, la evolución es una estrategia exitosa. Igualmente, las aplicaciones diversas y los impresionantes resultados de los AG demuestran que son un campo de estudio poderoso y que merece la pena.

- Un problema con el que los AG tienen dificultades son los problemas con las funciones objetivo ”engañosas”, en las que la situación de los puntos mejorados ofrecen información engañosa sobre dónde se encuentra probablemente el óptimo global. Por ejemplo: imagine un problema en el que el espacio de búsqueda esté compuesto por todas las cadenas binarias de

ocho caracteres, y en el que la aptitud de cada individuo sea directamente proporcional al número de unos en él es decir, 00000001 sería menos apto que 00000011, que sería menos apto que 00000111, etcétera -, con dos excepciones: la cadena 11111111 resulta tener una aptitud muy baja, y la cadena 00000000 resulta tener una aptitud muy alta. En este problema, un AG (al igual que la mayoría de los algoritmos) no tendría más probabilidad de encontrar un óptimo global que una búsqueda aleatoria.

La solución a este problema es la misma para los AG y la evolución biológica: la evolución no es un proceso que deba encontrar siempre el óptimo global. Puede funcionar casi igual de bien alcanzando la cima de un óptimo local alto y, para la mayoría de las situaciones, eso será suficiente, incluso aunque el óptimo global no pueda alcanzarse fácilmente desde ese punto. La evolución es como un "satisfactor" -un algoritmo que entrega una solución "suficientemente buena", aunque no necesariamente la mejor solución posible, dada una cantidad razonable de tiempo y esfuerzo invertidos en la búsqueda. La "FAQ de la evidencia de diseño improvisado en la naturaleza" proporciona ejemplos de la naturaleza con estos resultados. (También hay que tener en cuenta que pocos o ningún problema real es tan engañoso como el ejemplo algo forzado dado arriba. Normalmente, la situación de las mejoras locales proporciona alguna información sobre la situación del óptimo global).

- Un problema muy conocido que puede surgir con un AG se conoce como convergencia prematura. Si un individuo que es más apto que la mayoría de sus competidores emerge muy pronto en el curso de la ejecución, se puede reproducir tan abundantemente que merme la diversidad de la población demasiado pronto, provocando que el algoritmo converja hacia el óptimo local que representa ese individuo, en lugar de rastrear el paisaje adaptativo lo bastante a fondo para encontrar el óptimo global. Esto es un problema especialmente común en las poblaciones pequeñas, donde incluso una variación aleatoria en el ritmo de reproducción puede provocar que un genotipo se haga dominante sobre los otros.

  Los métodos más comunes implementados por los investigadores en AG para solucionar este problema implican controlar la fuerza selectiva, para no proporcionar tanta ventaja a los individuos excesivamente aptos. La selección escalada, por rango y por torneo, discutidas anteriormente, son tres de los métodos principales para conseguir esto; algunos métodos de selección

escalada son el escalado sigma, en el que la reproducción se basa en una comparación estadística de la aptitud media de la población, y la selección de Boltzmann, en la que la fuerza selectiva aumenta durante la ejecución de manera similar a la variable "temperatura" en el recocido simulado. La convergencia prematura ocurre en la naturaleza (los biólogos la llaman deriva genética). Esto no debe sorprender; como ya se dijo arriba, la evolución, como estrategia de resolución de problemas, no está obligada a encontrar la mejor solución, sólo una que sea lo bastante buena. Sin embargo, en la naturaleza, la convergencia prematura es menos común, ya que la mayoría de las mutaciones beneficiosas en los seres vivos sólo producen mejoras en la aptitud pequeñas e incrementales; son raras las mutaciones que producen una ganancia de aptitud tan grande que otorgue a sus poseedores una drástica ventaja reproductiva.

- Finalmente, varios investigadores aconsejan no utilizar AG en problemas resolubles de manera analítica. No es que los AG no puedan encontrar soluciones buenas para estos problemas; simplemente es que los métodos analíticos tradicionales consumen mucho menos tiempo y potencia computacional que los AG y, a diferencia de los AG, a menudo está demostrado matemáticamente que ofrecen la única solución exacta. Por supuesto, como no existe una solución matemática perfecta para ningún problema de adaptación biológica, este problema no aparece en la naturaleza.

### 1.1.6. ¿Cómo Saber si es Posible usar un Algoritmo Genético?

La aplicación más común de los AG ha sido la solución de problemas de optimización, en donde han mostrado ser muy eficientes y confiables.
Sin embargo, no todos los problemas pudieran ser apropiados para la técnica, y se recomienda en general tomar en cuenta las siguientes características del mismo antes de intentar usarla:

- Su espacio de búsqueda (i.e., sus posibles soluciones) debe de estar delimitado dentro de un cierto rango.
- Debe permitir definir una función de aptitud que nos indique que tan buena o mala es una cierta respuesta.
- Las soluciones deben codificarse de una forma que resulte relativamente fácil de implementar en el computador.

El primer punto es muy importante, y lo más recomendable es intentar resolver problemas que tengan espacios de búsqueda discretos aunque éstos sean muy grandes. Sin embargo, también podrá intentarse usar la técnica con espacios de búsqueda continuos, pero preferiblemente cuando exista un rango de soluciones relativamente pequeño.

### 1.1.7. Algunas Aplicaciones de los Algoritmos Genéticos

Como hemos podido observar, el área de aplicación de los AG es muy amplia, y en general sus aplicaciones se pueden implementar a muchos de los problemas de la vida cotidiana, de igual forma, se hayan aplicado a diversos problemas y modelos en ingeniaría, y en la ciencia en general cabe destacar entre ellos:

- ***Optimización:*** Se trata de un campo especialmente abonado para el uso de los AG, por las características intrínsecas de estos problemas. No en vano fueron la fuente de inspiración para los creadores estos algoritmos. Los AG se han utilizado en numerosas tareas de optimización, incluyendo la optimización numérica, y los problemas de optimización combinatoria.
- ***Programación automática:*** Los AG se han empleado para desarrollar programas para tareas específicas, y para diseñar otras estructuras computacionales tales como el autómata celular, y las redes de clasificación.
- ***Aprendizaje máquina:*** Los AG se han utilizado también en muchas de estas aplicaciones, tales como la predicción del tiempo o la estructura de una proteína. Han servido asimismo para desarrollar determinados aspectos de sistemas particulares de aprendizaje, como pueda ser el de los pesos en una red neuronal, las reglas para sistemas de clasificación de aprendizaje o sistemas de producción simbólica, y los sensores para robots.
- ***Economía:*** En este caso, se ha hecho uso de estos Algoritmos para modelizar procesos de innovación, el desarrollo estrategias de puja, y la aparición de mercados económicos.
- ***Sistemas inmunes:*** A la hora de modelizar varios aspectos de los sistemas inmunes naturales, incluyendo la mutación somática durante la vida de un individuo y el descubrimiento de familias de genes múltiples en tiempo evolutivo, ha resultado útil el empleo de esta técnica.

- ***Ecología:*** En la modelización de fenómenos ecológicos tales como las carreras de armamento biológico, la coevolución de parásito-huesped, la simbiosis, y el flujo de recursos.
- ***Genética de poblaciones:*** En el estudio de preguntas del tipo "¿Bajo qué condiciones será viable evolutivamente un gene para la recombinación?".
- ***Evolución y aprendizaje:*** Los AG se han utilizado en el estudio de las relaciones entre el aprendizaje individual y la evolución de la especie.
- ***Sistemas sociales:*** En el estudio de aspectos evolutivos de los sistemas sociales, tales como la evolución del comportamiento social en colonias de insectos, y la evolución de la cooperación y la comunicación en sistemas multi-agentes.

Aunque esta lista no es, en modo alguno, exhaustiva, sí transmite la idea de la variedad de aplicaciones que tienen los AG. Gracias al éxito en estas y otras áreas, los AG han llegado a ser un campo puntero en la investigación actual.

# 2. ALGORITMOS GENÉTICOS

## 2.1. ALGORITMOS GENÉTICOS SIMPLES

### 2.1.1. Tipos de Representación

Durante los primeros años el tipo de representación utilizado era siempre binario, debido a que se adapta perfectamente al tipo de operaciones y el tipo de operadores que se utilizan en un AG. Sin embargo, las representaciones binarias no son siempre efectivas por lo que se empezaron a utilizar otro tipo de representaciones. En general, una representación ha de ser capaz de identificar las características constituyentes de un conjunto de soluciones, de forma que distintas representaciones dan lugar a distintas perspectivas y por tanto distintas soluciones. Podemos considerar tres tipos básicos de representaciones:

*Representación binaria:* Cada gen es un valor 1 ó 0.

1 0 1 1 0 1

*Representación entera:* Cada gen es un valor entero.

1 0 3 -1 0 4

*Representación real:* Cada gen es un valor real.

1,78 2,6 7 0 -1,2 6,5

**Ejemplo:**

Primero considere como se puede usar una cadena de bits para describir una restricción sobre un atributo, por ejemplo, el atributo cielo con sus tres valores posibles: *soleado, nublado, lluvia.* Una manera obvia de codificar una restricción sobre este atributo, consiste en usar una cadena de 3 bits, en donde cada posición

de la cadena corresponde a un valor en particular. Colocar un 1 en alguna posición, indica que el atributo puede tomar el valor correspondiente. Por ejemplo, la cadena 010, representa la restricción *cielo = nublado.* De manera similar, la cadena 011 representa la restricción mas general *cielo = nublado* $\vee$ *cielo = lluvia.*
Observe que 111 representa la restricción mas general posible, indicando que no importa que valor tome el atributo cielo.

Dado este método para representar restricciones sobre atributos, las conjunciones de restricciones sobre múltiples atributos pueden representarse por concatenación. Por ejemplo, considere un segundo atributo, *viento,* con valores posibles *fuerte y débil.* La precondición de una regla como: *(cielo = nublado* $\vee$ *lluvia)* $\wedge$ *(viento = fuerte)* puede representarse por la siguiente cadena de bits de longitud 5:

| *cielo* | *viento* |
|---|---|
| 011 | 10 |

Las post-condiciones, como *jugar-tenis? = si,* pueden representarse de la misma manera.
La regla completa puede describirse concatenando también la cadena que representa la post-condición. Por ejemplo, la regla: *Si viento = fuerte* Entonces *jugar-tenis? = si,* puede representarse como:

| *cielo* | *viento* | *jugar-tenis?* |
|---|---|---|
| 111 | 10 | 10 |

Los primeros tres bits indican que no nos importa el valor del atributo cielo; los siguientes dos describen la restricción sobre viento; los dos últimos bits representan la post-condición de la regla. Aquí asumimos que *jugar-tenis?* Puede tomar valores *si o no.* Esto nos da una representación donde las reglas que nos interesan se codifican como cadenas de bits de longitud fija.

Es útil considerar que toda cadena de bits que es sintacticamente valida, representa una hipótesis bien definida. Por ejemplo, en el código anterior, la cadena 111 10 11, representa una regla cuya post-condición no establece restricciones sobre el atributo jugar tenis?. Para evitar esta posibilidad, se puede usar un código diferente, por ej., para el atributo *jugar-tenis?* usar un solo bit que indique los valores si o no; o bien, se pueden alterar los operadores genéticos para evitar explícitamente la construcción de tales cadenas de bits; o simplemente asignar un valor de adaptación muy bajo a estas cadenas.

### 2.1.2. Tamaño de la Población

Una cuestión que se puede plantear es la relacionada con el tamaño idóneo de la población. Parece intuitivo que las poblaciones pequeñas corren el riesgo de no cubrir adecuadamente el espacio de búsqueda, mientras que el trabajar con poblaciones de gran tamaño puede acarrear problemas relacionados con el excesivo costo computacional. Goldberg efectuó un estudio teórico, obteniendo como conclusión que el tamaño óptimo de la población para ristras de longitud I, con codificación binaria, crece exponencialmente con el tamaño de la ristra.
Este resultado traería como consecuencia que la aplicabilidad de los AG en problemas reales sería muy limitada, ya que resultarían no competitivos con otros métodos de optimización combinatoria. Alander, basándose en evidencia empírica sugiere que un tamaño de población comprendida entre l y 21 es suficiente para atacar con éxito los problemas por el considerados.

### 2.1.3. Población Inicial

Habitualmente la población inicial se escoge generando ristras al azar, pudiendo contener cada gen uno de los posibles valores del alfabeto con probabilidad uniforme. Se podria preguntar que es lo que sucedería si los individuos de la población inicial se obtuviesen como resultado de alguna técnica heurística o de optimización local. En los pocos trabajos que existen sobre este aspecto, se constata que esta inicialización no aleatoria de la población inicial, puede acelerar la convergencia del AG. Sin embargo en algunos casos la desventaja resulta ser la prematura convergencia del algoritmo, queriendo indicar con esto la convergencia hacia óptimos locales.
La población inicial de un AG puede ser creada de muy diversas formas, desde generar aleatoriamente el valor de cada gen para cada individuo, utilizar una función ávida o generar alguna parte de cada individuo y luego aplicar una búsqueda local.

### 2.1.4. Función Objetivo

Dos aspectos que resultan cruciales en el comportamiento de los AG son la determinación de una adecuada función de adaptación o función objetivo, así como la codificación utilizada.

Idealmente nos interesaría construir funciones objetivo con “ciertas regularidades”, es decir funciones objetivo que verifiquen que para dos individuos que se encuentren cercanos en el espacio de búsqueda, sus respectivos valores en las funciones objetivo sean similares. Por otra parte una dificultad en el comportamiento del AG puede ser la existencia de gran cantidad de óptimos locales, así como el hecho de que el óptimo global se encuentre muy aislado.
La regla general para construir una buena función objetivo es que ésta debe reflejar el valor del individuo de una manera “real”, pero en muchos problemas de optimización combinatoria, donde existe gran cantidad de restricciones, buena parte de los puntos del espacio de búsqueda representan individuos no válidos.
Para este planteamiento en el que los individuos están sometidos a restricciones, se han propuesto varias soluciones. La primera sería la que se podria denominar absolutista, en la que aquellos individuos que no verifican las restricciones, no son considerados como tales, y se siguen efectuando cruces y mutaciones hasta obtener individuos válidos, o bien, a dichos individuos se les asigna una función objetivo igual a cero.
Otra posibilidad consiste en reconstruir aquellos individuos que no verifican las restricciones. Dicha reconstrucción suele llevarse a cabo por medio de un nuevo operador que se acostumbra a denominar reparador.
Otro enfoque está basado en la penalización de la función objetivo. La idea general consiste en dividir la función objetivo del individuo por una cantidad (la penalización) que guarda relación con las restricciones que dicho individuo viola. Dicha cantidad puede simplemente tener en cuenta el número de restricciones violadas ó bien el denominado costo esperado de reconstrucción, es decir el coste asociado a la conversión de dicho individuo en otro que no viole ninguna restricción.
Otra técnica que se ha venido utilizando en el caso en que la computación de la función objetivo sea muy compleja es la denominada evaluación aproximada de la función objetivo. En algunos casos la obtención de n funciones objetivo aproximadas puede resultar mejor que la evaluación exacta de una única función objetivo (supuesto el caso de que la evaluación aproximada resulta como mínimo n veces más rápida que la evaluación exacta).

Un problema habitual en las ejecuciones de los AG surge debido a la velocidad con la que el algoritmo converge. En algunos casos la convergencia es muy rápida, lo que suele denominarse convergencia prematura, en la cual el algoritmo converge hacia óptimos locales, mientras que en otros casos el problema es justo el contrario, es decir se produce una convergencia lenta del algoritmo. Una posible solución a estos problemas pasa por efectuar transformaciones en la función

objetivo. El problema de la convergencia prematura, surge a menudo cuando la selección de individuos se realiza de manera proporcional a su función objetivo. En tal caso, pueden existir individuos con una adaptación al problema muy superior al resto, que a medida que avanza el algoritmo "dominan" a la población. Por medio de una transformación de la función objetivo, en este caso una comprensión del rango de variación de la función objetivo, se pretende que dichos "superindividuos" no lleguen a dominar a la población.
El problema de la lenta convergencia del algoritmo, se resolvería de manera análoga, pero en este caso efectuando una expansión del rango de la función objetivo.
La idea de especies de organismos, ha sido imitada en el diseño de los AG en un método propuesto por Goldberg y Richardson, utilizando una modificación de la función objetivo de cada individuo, de tal manera que individuos que estén muy cercanos entre sí devalúen su función objetivo, con objeto de que la población gane en diversidad.

Por ejemplo, si denotamos por $d(I_t^j, I_t^i)$ a la distancia de Hamming entre los individuos $I_t^j$ e $I_t^i$, y por K un parámetro rela positivo, podemos definir la siguiente función:

$$h(d(I_t^j, I_t^i)) = \begin{cases} K - d(I_t^j, I_t^i) & si \quad d(I_t^j, I_t^i) < K, \\ 0 & si \quad d(I_t^j, I_t^i) \geq K. \end{cases} \quad (1)$$

A continuación para cada individuo $I_t^j$ definimos $\sigma_j^t = \sum_{i \neq j} h(d(I_t^j, I_t^i))$, valor que utilizaremos para devaluar la función objetivo del individuo en cuestión. Es decir, $g*(I_t^j) = g(I_t^j)/\sigma_j^t$. De esta manera aquellos individuos que están cercanos entre sí verán devaluada la probabilidad de ser seleccionados como padres, aumentándose la probabilidad de los individuos que se encuentran más aislados.

(*la distancia de Hamming* se define como el número de bits que tienen que cambiarse para transformar una palabra de código válida en otra palabra de código válida.
Si dos palabras de código difieren en una distancia d, se necesitan d errores para convertir una en la otra.

Por ejemplo:

La distancia Hamming entre 1011101 y 1001001 es 2.
La distancia Hamming entre 2143896 y 2233796 es 3.
La distancia Hamming entre "toned" y "roses" es 3.)

### 2.1.5. Operador de Selección

El operador de Selección es el encargado de transmitir y conservar aquellas características de la soluciones que se consideran valiosas a lo largo de las generaciones. El principal medio para que la información útil se transmita es que aquellos individuos mejor adaptados (mejor valor de función de evaluación) tengan más probabilidades de reproducirse. Sin embargo, es necesario también incluir un factor aleatorio que permita reproducirse a individuos que aunque no estén muy bien adaptados, puedan contener alguna información útil para posteriores generaciones, con el objeto de mantener así también una cierta diversidad en cada población. Algunas de las técnicas de las cuales se dispone son las siguientes:

*Ruleta o Selección Proporcional:* Con este método la probabilidad que tiene un individuo de reproducirse es proporcional a su valor de función de evaluación, es decir, a su adaptación. En este método se define un rango con las características de la selección por sorteo. El número al azar será un número aleatorio forzosamente menor que el tamaño del rango. El elemento escogido será aquel en cuyo rango esté el número resultante de sumar el número aleatorio con el resultado total que sirvió para escoger el elemento anterior. El comportamiento es similar al de una ruleta, donde se define un avance cada tirada a partir de la posición actual. Tiene la ventaja de que no es posible escoger dos veces consecutivas el mismo elemento, y que puede ser forzado a que sea alta la probabilidad de que no sean elementos próximos en la población -esto último no es una ventaja de por sí; salvo que algunos de los otros operadores genéticos, es mejor utilizar un método de selección directa basado en la posición relativa de los individuos de la población-.

*Selección por Ranking:* Desarrollado por Whitley(1989) consiste en calcular las probabilidades de reproducción atendiendo a la ordenación de la población por el valor de adaptación en vez de atender simplemente a su valor de adecuación. Estas probabilidades se pueden calcular de diversas formas, aunque el método habitual es el ranking lineal (Baker (1985)).

*Selección por Torneo:* Reporta un valor computacional muy bajo debido a su sencillez. Se selecciona un grupo de t individuos (normalmente t = 2, torneo binario) y se genera un número aleatorio entre 0 y 1. Si este número es menor que un cierto umbral K (usualmente 0,75), se selecciona para reproducirse al individuo con mejor adaptación, y si este número es menor que K, se selecciona, por el contrario, al individuo con peor adaptación.
Esta técnica tiene la ventaja de que permite un cierto grado de elitismo -el mejor

nunca va a morir, y los mejores tienen más probabilidad de reproducirse y de emigrar que los peores- pero sin producir una convergencia genética prematura, si la población es, al menos, un orden de magnitud superior al del número de elementos involucrados en el torneo. En caso de que la diferencia sea menor no hemos observado mucha diferencia entre emplear el torneo o no.

### 2.1.6. Operador de Cruce

El operador de cruce permite realizar una exploración de toda la información almacenada hasta el momento en la población y combinarla para crear mejores individuos. Dentro de los métodos habituales destacamos los siguientes:

*Cruce de un punto:* Es el método de cruce más sencillo. Se selecciona una posición en las cadenas de los progenitores, y se intercambian los genes a la izquierda de esta posición.

*Cruce de n puntos:* Es una generalización del método anterior. Se seleccionan varias posiciones (n) en las cadenas de los progenitores y se intercambian los genes a ambos lados de estas posiciones.

*Cruce Uniforme:* Se realiza un test aleatorio para decidir de cual de los progenitores se toma cada posición de la cadena.

*Cruces para permutación:* Existe una familia de cruces específicas para los problemas de permutación, siendo algunos de ellos:

- *Cruce de mapeamiento parcial:* Toma una subsecuencia del genoma del padre y procura preservar el orden absoluto de los fenotipos -es decir, orden y posición en el genoma- del resto del genoma lo más parecido posible de la madre.
- *Cruce de orden:* toma una subsecuencia del genoma del padre y procura preservar el orden relativo de los fenotipos del resto del genoma lo más parecido posible de la madre.
- *Cruce de ciclo:* Tomamos el primer gen del genoma del padre, poniéndolo en la primera posición del hijo, y el primer gen del genoma de la madre, poniéndolo dentro del genoma del hijo en la posición que ocupe en el genoma del padre. El fenotipo que está en la posición que ocupa el gen del genoma del padre igual al primer gen del genoma de la madre se va a colocar en la

posición que ocupe en el genoma del padre, y así hasta rellenar el genoma del hijo.

Es una buena idea que, tanto la codificación como la técnica de cruce, se hagan de manera que las características buenas se hereden; o, al menos, no sea mucho peor que el peor de los padres. En problemas en los que, por ejemplo, la adaptación es función de los pares de genes colaterales, el resultante del cruce uniforme tiene una adaptación completamente aleatoria.

### 2.1.7. Operador de Mutación

La mutación se considera un operador básico, que proporciona un pequeño elemento de aleatoriedad en la vecindad (entorno) de los individuos de la población. Si bien se admite que el operador de cruce es el responsable de efectuar la búsqueda a lo largo del espacio de posibles soluciones, también parece desprenderse de los experimentos efectuados por varios investigadores que el operador de mutación va ganando en importancia a medida que la población de individuos va convergiendo (Davis). El objetivo del operador de mutación es producir nuevas soluciones a partir de la modificación de un cierto número de genes de una solución existente, con la intención de fomentar la variabilidad dentro de la población. Existen muy diversas formas de realizar la mutación, desde la más sencilla ( Puntual ), donde cada gen muta aleatoriamente con independencia del resto de genes, hasta configuraciones más complejas donde se tienen en cuanta la estructura del problema y la relación entre los distintos genes.

Schaffer y col. encuentran que el efecto del cruce en la búsqueda es inferior al que previamente se esperaba. Utilizan la denominada evolución primitiva, en la cual, el proceso evolutivo consta tan sólo de selección y mutación. Encuentran que dicha evolución primitiva supera con creces a una evolución basada exclusivamente en la selección y el cruce. Otra conclusión de su trabajo es que la determinación del valor óptimo de la probabilidad de mutación es mucho más crucial que el relativo a la probabilidad de cruce. Si bien en la mayoría de las implementaciones de AG se asume que tanto la probabilidad de cruce como la de mutación permanecen constantes, algunos autores han obtenido mejores resultados experimentales modificando la probabilidad de mutación a medida que aumenta el número de iteraciones.

### 2.1.8. Reemplazo de la Población y Condición de Parada

Cada vez que se aplica el operador de cruce, nos encontramos con un número de nuevos individuos (la descendencia) que se han de integrar en la población para formar la siguiente generación. Esta operación se puede hacer de diversas formas, pero en general existen tres métodos fundamentales para realizar el reemplazo:

- Cuando el número de individuos llega a un cierto número, se elimina un subconjunto de la población conteniendo a los individuos peor adaptados.
- Cada vez que se crea un nuevo individuo, en la población se elimina el peor adaptado para dejar su lugar a este nuevo individuo.
- Cada vez que se crea un nuevo individuo, en la población se elimina aleatoriamente una solución, independientemente de su adaptación.

En cuanto a el criterio de parada, generalmente viene determinado por criterios a priori sencillos, como un número máximo de generaciones o un tiempo máximo de resolución, o más eficientemente por estrategias relacionadas con indicadores del estado de evolución de la población, como por la pérdida de diversidad dentro de la población o por no haber mejora en un cierto número de iteraciones, siendo por lo general una condición mixta lo más utilizado, es decir, limitar el tiempo de ejecución a un número de iteraciones y tener en cuenta algún indicador del estado de la población para considerar la convergencia antes de alcanzar tal limitación.

El problema que confrontan los AG consiste en identificar dentro de un espacio de hipótesis candidato, la mejor, donde la mejor hipótesis es aquella que optimiza una medida numérica predefinida para el problema, llamada adaptación ($fitness$) de la hipótesis. Por ejemplo, si la tarea de aprendizaje es el problema de aproximar una función desconocida dado un conjunto de entrenamiento de entradas y salidas, la adaptación puede definirse como la precisión de la hipótesis sobre el conjunto de entrenamiento (porcentaje de éxitos al predecir la salida). Si la tarea de aprendizaje tiene la forma de un juego, la adaptación puede medirse en términos del porcentaje de partidas ganadas. Aunque los detalles de implementación varían entre diferentes AG, todo comparten en general la siguiente estructura: El algoritmo opera iterativamente, actualizando un conjunto de hipótesis llamada población. En cada iteración, todos los miembros de la población son evaluados de acuerdo a una función de adaptación. Una nueva población es generada, seleccionando probabilísticamente los individuos de mayor adaptación en la población presente. Algunos de estos individuos pasan intactos a la siguiente generación.

Otros son seleccionados para crear una nueva generación, aplicando operaciones genéticas como el cruce y la mutación.

El prototipo de un AG simple se muestra a continuación:

---

*generación de la población inicial*
***while*** *no se cumpla la condición de parada do*
. *evaluación de los individuos*
. *selección*
. *cruce*
. *mutación*
***end*** *while*

---

Las entradas de este algoritmo incluyen una función de adaptación para evaluar los candidatos a hipótesis, un umbral definiendo el nivel aceptado de adaptación para dar por terminado el algoritmo, el tamaño que debe mantener la población, y los parámetros necesarios para determinar como evoluciona la población, esto es, la fracción de la población que será remplazada en cada generación y la tasa de mutación presente.
Observen que cada iteración de este algoritmo produce una nueva generación de hipótesis, con base en la población actual de hipótesis. Primero, un cierto número de hipótesis en la población $((1-r)\times p)$ es seleccionado probabilísticamente para su inclusión en la prole. La probabilidad de una hipotesis $h_i \in pob$ de ser seleccionada esta dada por:

$$P_r(h_i) = \frac{adaptacion(h_i)}{\sum_{j=1}^{p} adaptacion(h_i)} \tag{2}$$

Por lo tanto, la probabilidad de que una hipótesis sea seleccionada es proporcional a su propio indice de adaptación, e inversamente proporcional a la adaptación de las hipótesis concurrentes en la misma población.
Una vez que estos miembros de la población son seleccionados para ser incluidos en la generación, el resto de los miembros de esta se genera usando el operador de cruce.
Este operador, selecciona $(r \times p)/2$ pares de hipótesis en la población y genera dos descendientes para cada par, combinando porciones de ambos padres. Los padres son elegidos probabilísticamente con la $P_r(h_i)$ definida como hasta ahora.

En este momento, la generación tiene el tamaño de población deseado p, así que solo resta seleccionar m elementos de la población para aplicar la operación de mutación. Estos elementos son elegidos al azar y los cambios efectuados en ellos son también aleatorios.

Algunos enfoques de AG practican un elitismo al evitar explícitamente que las mejores hipótesis encontradas hasta ese momento, puedan mutar.
Este AG lleva a cabo una búsqueda por barrido (*beam*), paralela y aleatoria de hipótesis que tienen un buen desempeño de acuerdo a la función de adaptación.

### 2.1.9. Otros Operadores

Estos operadores no son utilizados en todos los problemas, solamente son empleados en algunos, y en principio su variedad es infinita. Generalmente son operadores que exploran el espacio de soluciones de una forma más ordenada, y que actúan más en las últimas fases de la búsqueda, en la cuál se pasa de soluciones çasi buenas.ª "buenas"soluciones.

- ***Cromosomas de Longitud Variable:*** Hasta ahora se han descrito cromosomas de longitud fija, donde se conoce de antemano el número de parámetros de un problema. Pero hay problemas en los que esto no sucede. Por ejemplo, en un problema de clasificación, donde dado un vector de entrada, queremos agruparlo en una serie de clases, podemos no saber siquiera cuántas clases hay. Por ejemplo, en un perceptrón hay reglas que dicen cuantas neuronas se deben de utilizar en la capa oculta; pero en un problema determinado puede que no haya ninguna regla heurística aplicable; se tendría que utilizar los AG para hallar el número óptimo de neuronas.

  En estos casos, se necesitan operadores más: añadir y eliminar. Estos operadores se utilizan para añadir un gen, o eliminar un gen del cromosoma. La forma más habitual de añadir un gen es duplicar uno ya existente, el cuál sufre mutación y se añade al lado del anterior. En este caso, los operadores del AG simple (selección, mutación, cruce) funcionarán de la forma habitual, salvo, claro está, que sólo se hará cruce en la zona del cromosoma de menor longitud.

- ***Operadores de Nicho (Ecológico):*** Otros operadores importantes son los operadores de nicho. Estos operadores están encaminados a mantener la diversidad genética de la población, de forma que cromosomas similares

sustituyan sólo a cromosomas similares, y son especialmente útiles en problemas con muchas soluciones; un AG con estos operadores es capaz de hallar todos los máximos, dedicándose cada especie a un máximo. Más que operadores genéticos, son formas de enfocar la selección y la evaluación de la población.

Uno de las formas de llevar esto a cabo es la introducción del crowding (apiñamiento). Otra forma es introducir una función de compartición o sharing, que indica cuán similar es un cromosoma al resto de la población. La puntuación de cada individuo se dividirá por esta función de compartición, de forma que se facilita la diversidad genética y la aparición de individuos diferentes.
También se pueden restringir los emparejamientos, por ejemplo, a aquellos cromosomas que sean similares. Para evitar las malas consecuencias del inbreeding (destinado a potenciar ciertas características frente a otras) que suele aparecer en poblaciones pequeñas, estos periodos se intercalan con otros periodos en los cuáles el emparejamiento es libre.

- ***Operadores Especializados:*** En una serie de problemas hay que restringir las nuevas soluciones generadas por los operadores genéticos, pues no todas las soluciones generadas van a ser válidas, sobre todo en los problemas con restricciones. Por ello, se aplican operadores que mantengan la estructura del problema. Otros operadores son simplemente generadores de diversidad, pero la generan de una forma determinada:
*Zap:*En vez de cambiar un solo bit de un cromosoma, cambia un gen completo de un cromosoma.
*Creep:*Este operador aumenta o disminuye en 1 el valor de un gen; sirve para cambiar suavemente y de forma controlada los valores de los genes.
*Transposición:*Similar al cruce y a la recombinación genética (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.), pero dentro de un solo cromosoma; dos genes intercambian sus valores, sin afectar al resto del cromosoma. Similar a este es el operador de eliminación-reinserción, en el que un gen cambia de posición con respecto a los demás.

### 2.1.10. Aplicando Operadores Genéticos

En toda ejecución de un AG hay que decidir con qué frecuencia se va a aplicar cada uno de los AG; 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 suelen 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, 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 AG a una búsqueda aleatoria; y hay que aplicar otros operadores. En todo caso, se pueden usar operadores especializados.

**Ejemplo simple de AG:**

Se requiere calcular el máximo de una función *f(x)* en un intervalo *[a,b]*. Para esto solamente se deriva la función y se iguala a cero. Pero se presenta un pequeño inconveniente y es que no se conoce a *f(x)*, aunque sí se puede calcular o estimar su valor en cualquier punto. He aquí los pasos a seguir:

- Se estiman la resolución con la que se desea trabajar. Es decir, se elige el número de puntos que se van a examinar dentro del intervalo. Si, por ejemplo, el intervalo es el *[0,100]* y se asigna una resolución de *0.5*, entonces se obtendrán *200* puntos en el intervalo.
- Se genera una población inicial de n individuos; que serán n números (elegidos al azar). Es decir, se obtiene a $x_1, x_2, ..., x_n$. Todos ellos se encuentran dentro del intervalo *[a,b]*.
- Ahora se le debe asignar mayor capacidad de reproducción a los mejor dotados. Si se esta buscando el máximo, pues el mejor dotado será aquel cuyo valor de $f(x_i)$ sea mayor. De los n individuos que se tienen, se creara una

población intermedia, que será la población de los individuos que pasaran a ser recombinados. Luego se calcula la frecuencia de cada uno de los n genotipos en la primera población, de la siguiente forma: $p(x_i) = \frac{f(x_i)}{suma-total}$. A continuación definimos $P(x_i)$ como la función de distribución: $P(x_j)$ es la suma, desde 0 hasta $j$, de los $p(x_i)$.

**Ejempo:** Supongamos que n es igual a 4, y que, por ejemplo, se tiene:

$$f(x_0) = 10$$
$$f(x_1) = 40$$
$$f(x_2) = 30$$
$$f(x_3) = 20$$

Por lo tanto,

$$p(x_0) = 0{,}1$$
$$p(x_1) = 0{,}4$$
$$p(x_2) = 0{,}3$$
$$p(x_3) = 0{,}2$$

y también,

$$P(x_0) = 0{,}1$$
$$P(x_1) = 0{,}5$$
$$P(x_2) = 0{,}8$$
$$P(x_3) = 1{,}0$$

Es claro que los $p(x_i)$ suman 1. A continuación se generan $n$ números aleatorios entre 0 y 1. Cada uno de esos números por ejemplo, $t$ estará relacionado con un individuo de la generación intermedia; de la siguiente forma:

si t está entre $P(x_i)$ y $P(x_{i+1})$ escogemos $x_{i+1}$.
Siguiendo con el ejemplo: Si los cuatro números aleatorios son 0.359, 0.188, 0.654 y 0.399, entonces la generación intermedia será:

Como se ve, está claro que, pese al azar el individuo mejor dotado es el más favorecido.

- El siguiente paso es la recombinación. Se puede hacer de la forma que parezca más adecuada. En primer lugar hay que buscar las parejas. Se trata de

| Número Aleatorio | Individuo Seleccionado | Nuevo Valor |
|---|---|---|
| 0.1<0.359<0.5 | $x_1$ | $x_0$ |
| 0.1<0.188<0.5 | $x_1$ | $x_1$ |
| 0.5<0.654<0.8 | $x_2$ | $x_2$ |
| 0.1<0.399<0.5 | $x_1$ | $x_3$ |

obtener una nueva generación como mezcla de esta con la que se esta trabajando. Una vez que se tienen las parejas, se hace la recombinación. Una alternativa: media aritmética; otra: media geométrica. Otra, la más usada: cortar los dos números "papás"por un lugar al azar y conseguir los "hijos"intercambiando partes.

Para ejemplificar mejor veamos: si los individuos son 456 y 123, se elige al azar un número entre 1 y 2 (ya que hay tres cifras). Si, por ejemplo, se obtiene el 1, entonces un posible hijo podría ser el 4-23 y otro 1-56. Aunque lo más habitual es trabajar en base 2 (por lo menos, los números serán más largos).

- En este momento ya se tiene una nueva generación. Generalmente, mejor dotada que la inicial. Lo que se hace ahora es volver al tercer punto (cálculo de una población intermedia). Por supuesto, el algoritmo se para cuando se considere prudente o necesario. Por ejemplo, cuando se lleve un número de iteraciones determinado, o cuando la población no mejore demasiado.

- Una variante posible es permitir la existencia de mutaciones (por ejemplo, introducir cada cierto número de generaciones alguna variación en una cifra en busca de mejores soluciones. Esto es interesante debido a que, por ejemplo, si se están dando vueltas en torno a un máximo local, la variación introducida podría llevarnos hacia un genotipo mejor dotado.

- Otra variante es la elitista, consistente en que los mejores genotipos no se recombinen, sino que pasen directamente -tal cual- a la siguiente generación.

**Ejemplo:**

Un AG puede verse como un método de optimización general, que busca en un gran espacio de candidatos, a los elementos que tienen mejor desempeño con respecto a la función de objetivo. Aunque no garantizan encontrar el elemento opti-

mo, los AG generalmente tienen éxito en encontrar un elemento con alta adaptación.

Para ilustrar el uso de los AG en el aprendizaje de conceptos, se presentara brevemente el sistema GABIL que usa AG para aprender conceptos booleanos representados como un conjunto disyuntivo de reglas preposicionales.

Se tomara el parámetro $r = 0{,}6$ que determina el porcentaje de la población que será remplazada por cruce,. El parámetro $m$ que determina la tasa de mutación, fue fijado en 0.001.

Estos son valores tipicos para ambos parámetros. El tamaño de la población vario entre 100 y 1000, dependiendo de la tarea de aprendizaje específica en cada experimento. Las hipótesis en GABIL se representan como un conjunto disyuntivo de reglas proposicionales, solo que el atributo objetivo es codificado por un solo bit. Para representar el conjunto de reglas, las cadenas representando reglas individuales son concatenadas. Por ejemplo, consideren un espacio de hipótesis donde las pre-condiciones de las reglas son conjunciones restricciones sobre dos atributos $a_1$ y $a_2$. El atributo objetivo es c. Entonces, la hipótesis que consta de estas dos reglas:

Si $a_1 = T \wedge a_2 = F$ Entonces $c = T$; Si $a_2 = T$ Entonces $c = F$, se representa por la cadena:

| $a_1$ | $a_2$ | $c$ | $a_1$ | $a_2$ | $c$ |
|---|---|---|---|---|---|
| 10 | 01 | 1 | 11 | 10 | 0 |

Observen que la longitud de la cadena de bits crece con el número de reglas en la hipótesis. Esta longitud variable requiere una ligera modificación en la definición de operador de cruce. El operador de cruce usado por GABIL es una extensión estándar del operador de cruce en dos puntos.
En particular, dada la longitud variable de la cadena de bits representando las hipótesis, el operador de corte debe estar restringido a aplicarse bajo las siguientes condiciones: Dos padres son seleccionados, luego dos puntos de cruce son elegidos aleatoriamente sobre el primer padre. Sea $d_1(d_2)$ la distancia del punto mas a la izquierda (mas a la derecha) al limite de la regla inmediatamente a la izquierda. Los puntos de cruce en el segundo padre se fijan ahora aleatoriamente con la restricción de que deben tener los mismos valores $d_1(d_2)$. Por ejemplo, si los dos padres son:

$$\begin{array}{rcccccc} & a_1 & a_2 & c & a_1 & a_2 & c \\ h_1 = & 10 & 01 & 1 & 11 & 10 & 0 \end{array}$$

y:

$$\begin{array}{rcccccc} & a_1 & a_2 & c & a_1 & a_2 & c \\ h_2 = & 01 & 11 & 0 & 10 & 01 & 0 \end{array}$$

y los puntos de cruce para el primer padre son los bits 1 y 8:

$$\begin{array}{rcccccc} & a_1 & a_2 & c & a_1 & a_2 & c \\ h_1 = & 1[0 & 01 & 1 & 11 & 1]0 & 0 \end{array}$$

entonces $d_1 = 1$ y $d_2 = 3$, por lo que los pares de puntos de cruce permitidos para la hipótesis $h_2$ son: $\langle 1, 3\rangle, \langle 1, 8\rangle$ *y* $\langle 6, 8\rangle$ . Si el par $\langle 1, 3\rangle$ es elegido:

$$\begin{array}{rcccccc} & a_1 & a_2 & c & a_1 & a_2 & c \\ h_2 = & 0[1 & 1]1 & 0 & 10 & 01 & 0 \end{array}$$

Por lo que la prole resultante de estas dos hipótesis es:

$$\begin{array}{rccc} & a_1 & a_2 & c \\ h_3 = & 11 & 10 & 0 \end{array}$$

y:

$$\begin{array}{rccccccccc} & a_1 & a_2 & c & a_1 & a_2 & c & a_1 & a_2 & c \\ h_4 = & 00 & 01 & 1 & 11 & 11 & 0 & 10 & 01 & 0 \end{array}$$

Como se puede ver en el ejemplo, este operador de cruce permite que la prole contenga diferente número de reglas que sus padres, y al mismo tiempo garantiza que todas las cadenas de bits generadas de esta forma, representen conjuntos de reglas bien definidos.

La función objetivo para cada hipótesis se basa en su precisión como clasificador sobre un conjunto de ejemplos de entrenamiento. En particular, la función usada para medir la adaptación es:

$$adaptacion(h) = (correcto(h))^2$$

donde $correcto(h)$ es el porcentaje de ejemplos bien clasificados por la hipótesis $h$.

## 2.2. ALGORITMOS GENÉTICOS PARALELOS

Un programa es paralelo si en cualquier momento de su ejecución puede ejecutar más de un proceso. Para crear programas paralelos eficientes hay que poder crear, destruir y especificar procesos así como la interacción entre ellos. Básicamente existen tres formas de paralelizar un programa:

- *Paralelización de grano fino:* la paralelización del programa se realiza a nivel de instrucción. Cada procesador hace una parte de cada paso del algoritmo (selección, cruce y mutación) sobre la población común.
- *Paralelización de grano medio:* los programas se paralelizan a nivel de bucle. Esta paralelización se realiza habitualmente de una forma automática en los compiladores.
- *Paralelización de grano grueso:* se basan en la descomposición del dominio de datos entre los procesadores, siendo cada uno de ellos el responsable de realizar los cálculos sobre sus datos locales.

La paralelización de grano grueso tiene como atractivo la portabilidad, ya que se adapta perfectamente tanto a multiprocesadores de memoria distribuida como de memoria compartida. Este tipo de paralelización se puede a su vez realizar siguiendo tres estilos distintos de programación: paralelismo en datos, programación por paso de mensajes y programación por paso de datos.

- *Paralelismo en datos:* El compilador se encarga de la distribución de los datos guiado por un conjunto de directivas que introduce el programador. Estas directivas hacen que cuando se compila el programa las funciones se distribuyan entre los procesadores disponibles. Como principal ventaja presenta su facilidad de programación. Los lenguajes de paralelismo de datos mas utilizados son el estándar HPF (High Performance Fortran) y el OpenMP.
- *Programación por paso de mensajes:* El método mas utilizado para programar sistemas de memoria distribuida es el paso de mensajes o alguna variante del mismo. La forma mas básica consiste en que los procesos coordinan sus actividades mediante el envío y la recepción de mensajes. Las librerías mas utilizadas son por este orden la estándar MPI (Message Passing Interface) y PVM (Parallel Virtual Machine).

- *Programación por paso de datos:* A diferencia del modelo de paso de mensajes, la transferencia de datos entre los procesadores se realiza con primitivas unilaterales tipo put-get, lo que evita la necesidad de sincronización entre los procesadores emisor y receptor. Es un modelo de programación de muy bajo nivel pero muy eficiente, aunque en la actualidad son muy pocos los fabricantes que los soportan.

Una de las principales ventajas de los AG es que permite que sus operaciones se puedan ejecutar en paralelo. Debido a que la evolución natural trata con una población entera y no con individuos particulares, excepto para la fase de selección, durante la cual existe una competencia entre los individuos y en la fase de reproducción, en donde se presentan iteraciones entre los miembros de la población, cualquier otra operación de la población, en particular la evaluación de cada uno de los miembros de la población, pueden hacerse separadamente. Por lo tanto, casi todas las operaciones en los AG son implícitamente paralelas.

Se ha establecido que la eficiencia de los AG para encontrar una solución optima, esta determinada por el tamaño de la población. Por lo tanto, una población grande requiere de mas memoria para ser almacenada. También se ha probado que toma mayor cantidad de tiempo para converger. Si $n$ es el tamaño de la población, la convergencia esperada es $n\ log(n)$.

Al utilizar computadores en paralelo, no solamente se provee de mas espacio de almacenamiento, sino también con el uso de ellos se podrán producir y evaluar mas soluciones en una cantidad de tiempo mas pequeño. Debido al paralelismo, es posible incrementar el tamaño de la población, reducir el costo computacional y mejorar el desempeño de los AG.

Probablemente el primer intento que se hizo para implementar los AG en arquitecturas en paralelo fue en 1981, por John Grefenstette. Los primeros ensayos consistían en un paralelismo global. Esta aproximación trataba por paralelizar explícitamente las tareas paralelas implícitas de los AG secuénciales, por lo tanto la naturaleza de los problemas permanecía invariable. El algoritmo simplemente manejaba una sencilla población en donde cada individuo podía combinarse con cualquiera de los otros, pero la generación de los nuevos hijos y/o su evaluación se hacía en paralelo. La idea básica es que los diferentes procesadores puedan crear nuevos individuos y computar sus aptitudes en paralelo, sin tener que comunicarse con los otros.

La evaluación de la población en paralelo es simple de implementar. A cada procesador se le asigna un subconjunto de individuos para ser evaluados. Por ejemplo, en un computador de memoria compartida, los individuos pueden estar almacenados en la memoria, y cada uno de los procesadores puede leer los cromosomas asignados y puede grabar los resultados del computo de las aptitudes. Este método solamente supone que los AG trabajan con una generación actualizada de la población. Se necesita además, alguna sincronización entre las generaciones.

Generalmente, la mayor parte del tiempo de computo en un AG se gasta en la funcion objetivo. El tiempo que se gasta en el manejo de los cromosomas durante las fases de selección y recombinación es despreciable. En un computador de memoria distribuida se puede almacenar la población en un procesador "maestro", el cual es responsable de enviar los individuos a los otros procesadores ”esclavos". El "maestro", también es responsable por guardar los resultados de la evaluación. Una desventaja de esta implementación es que se pueden presentar cuellos de botella, cuando los esclavos están desocupados y solo el maestro esta trabajando. Pero si se hace un buen uso del procesador maestro, se puede mejorar el factor de balance, distribuyendo dinámicamente los individuos a los procesadores esclavos, cuando ellos terminen sus trabajos.
Una segunda clase de AG paralelos consiste en dividir la población en subpoblaciones, y cada una de ellas ejecutarlas en un procesador. El intercambio entre subpoblaciones es posible por medio de un operador de "migración".

Se emplea el modelo de islas para mostrar como los AG se comportan como si el mundo fuera constituido por islas que se desarrollaran en forma independiente, unas de las otras. En cada una de las islas, la población es libre de converger hacia un optimo diferente. El operador de migración permite extraer de las diferentes subpoblaciones las buenas características, para luego mezclarlas.

A continuación se muestra el seudocódigo de un algoritmo con la evaluación paralelizada:

---

*generación de la población inicial*
***while*** *no se cumpla la condición de parada do*
*do in parallel*
. *evaluación de los individuos*
*end parallel do*
. *selección*
. *cruce*

. *mutación*
***end*** *while*

---

y en el siguiente esquema se muestra el seudocódigo de un AG con varias poblaciones que evolucionan en paralelo:

---

*do in parallel*
*generación de la población inicial*
***while*** *no se cumpla la condición de parada do*
. *evaluación de los individuos*
. *selección*
. *producción de nuevos individuos*
. *mutación*
***end*** *while*
*end parallel do*
*Escoger la mejor solución*

---

## 2.3. COMPARACIÓN DE LOS ALGORITMOS GENÉTICOS CON OTRAS TÉCNICAS DE OPTIMIZACIÓN

Existen otras técnicas para resolver problemas de búsqueda y de optimización. Tanto los métodos que se describen a continuación, como los AG no requieren información relacionada con la estructura del espacio de búsqueda, ni con las propiedades de la función objetivo. Estos métodos son generalmente estocásticos y necesitan utilizar generadores de números aleatorios. Su proceso se realiza haciendo suposiciones sobre en donde las soluciones mas optimas tienen la probabilidad mas alta de ser encontradas en el espacio de búsqueda. La mayoría de las veces estos métodos no garantizan que se encuentra una solución optima global al problema a resolver. Ellos solamente son capaces de proveer buenas soluciones en una cantidad de tiempo razonable, sin verificar que es la mejor, siendo esto una de las desventajas de estos métodos. Entre otras técnicas consideradas para la solución de problemas de búsqueda y optimización se tienen:

- *Búsqueda aleatoria:* Explora el espacio de búsqueda seleccionando soluciones y evaluando sus aptitudes. Se considera una estrategia no inteligente

y se utiliza pocas veces. Sin embargo, cuando se obtiene una solución y no es la optima, se puede mejorar continuando la ejecución del algoritmo por mas tiempo. Teóricamente si el espacio de búsqueda es finito, este método garantiza encontrar la solución optima, pero en la mayoria de los problemas, explorar todo el espacio de búsqueda toma gran cantidad de tiempo.

- *Escalada por la Máxima Pendiente:* Es el método mas simple de los que utilizan una clase de gradiente para direccionar la búsqueda. En cada iteración se escoge aleatoriamente una solución, cercana a la solución actual y si la seleccionada mejora la función objetivo, se guarda. Este método converge a una solución optima si la función objetivo del problema es continua y si tiene solamente un pico (unímodal). Si la función es multimodal, el algoritmo termina en el primer pico que encuentre, aun sin ser este el mas alto. Una forma de evitar que termine en un óptimo local consiste en repetir varias veces el proceso, comenzando desde diferentes puntos seleccionados aleatoriamente.

- *Temple Simulado:* Este método se origino debido al proceso de formación de cristales en sólidos durante el proceso de enfriamiento. Este método se comporta similar al método de la Escalada por la Máxima Pendiente, pero con la posibilidad de descender, para evitar que se logre un optimo local. Cuando la temperatura es alta, la probabilidad de modificar la solución es importante y por lo tanto es posible realizar varios movimientos para explorar el espacio de búsqueda. Cuando la temperatura decrece, existe mayor dificultad para descender y el algoritmo trata de escalar desde la ultima solución encontrada. Cuando la temperatura es baja, se toma la solución actual. Usualmente, el temple simulado comienza con una temperatura alta, la cual decrece exponencialmente. El punto mas bajo de enfriamiento, es el mejor. La mayor dificultad de este método consiste en definir una tasa de decrecimiento, para que el comportamiento del algoritmo sea bueno.

  El método del Temple Simulado mezclado con algunas características de exploración del método de Búsqueda Aleatoria y del método de Máxima Pendiente, conlleva a buenos resultados. El Temple Simulado es uno de los competidores mas fuertes de los AG. Ambos se derivan de una analogía con sistemas de evolución natural. Los AG difieren del Temple Simulado en dos principales características: Primero, los AG usan una población basada en selección, mientras el Temple Simulado solamente trabaja con un individuo en cada iteración, pero de otro lado, las iteraciones en el Temple Simula-

do son mucho mas simples y a menudo se ejecutan mucho mas rápido. La mayor ventaja de los AG consiste en la habilidad excepcional para ser paralelizado, mientras que el Temple Simulado no gana mucho. Segundo, los AG usan operadores de recombinación, capaces de mezclar buenas características desde diferentes soluciones. De otro lado, el método del Temple Simulado es muy simple de implementar y genera buenos resultados.

# 3. ALGORITMOS GENÉTICOS: CÓDIGO EN R

## 3.1. PROGRAMA EN R: ALGORITMOS GENÈTICOS

```
simulaciones<-1000
tasas.t<-array(NA,dim=c(simulaciones,2))
for (im in 1:simulaciones){

library(MASS)

muestral<-mvrnorm(n=20,rep(l,2),matrix(c(l,0.3,0.3,1),2,2))
muestra2<-mvrnorm(n=20,rep(2,2),matrix(c(1,O.1,O.1,1),2,2))
población.simulada<-rbind((cbind(muestral,c(0))),(cbind(muestra2,c(1))))


muestralK-mvrnorm(n=20,rep(1,2),matrix(c(l,0.3,0.3,1),2,2))
muestra22<-mvrnorm(n=20,rep(2,2),matrix(c(l,0.1,0.1,1),2,2))
población.simuladal<-rbind((cbind(muestral1,c(0))),(cbind(muestra22,c(l))))


tamaño.p<-30
variables<-2
iteraciones<-30
Ngen<-11


cromosomas<-array(sample(c(0,1),(Ngen*variables)*(tamano.p),replace=T),
dim=c(tamano.p,(variables*Ngen)))


transforma.binario.a.decimal<-function(Ngen){
y<-length(Ngen)
pot<-2∧seq((y − 1), 0)
temp < −sum(Ngen ∗ pot)
}

generacion.de.w<-function(matriz,variables){
n<-ncol (matriz)
m<-nrow (matriz)
z<-variables-1
tamano<-(n/variables)-1
```

```
matriz.w<-array(NA,dim=c(m,variables))
for(j in 1:m){
for(i in 0:z){
x<-i*(tamano+l)+2
y<-(tamano+1)*(i+1)
w<-(tamano+1)*i+1
if(matriz[j,w]==0
matriz.w[j,(i+1)]<-(transforma.binario.a.decimal(matriz[j,x:y]))*(-1)
}
else{
matriz.w[j,(i+1)]<-transforma.binario.a.decimal(matriz[j,x:y])
}
}
}


matriz.w
matriz.w1<-apply (matriz,w,2,abs)
matriz.w2<-apply(matriz,w1,1,max)
matriz.w3<-matriz.w/matriz.w2
}


generacion.de.z<-function(matriz.x,vector.w){
tempK-apply (vector,w,1,as.vector)
matriz. x %* %temp1


operacion<-function(obj,y){
temp<-obj-y
temp1<-apply(temp,2,sum)
}


Operacion1<-function(obj,y){
temp<-abs(obj-y)
temp1<-apply(temp,2,sum)
}


indice<-function(z1 ,z2){
```

```
n<-ncol(z1)
resultado<-array(0,dim=c(nrow(zl),n))
resultado1<-array(0,dim=c(nrow(z1),n))
for(i in l:n){
if (i==i) {resultado[,i]<-apply(as.matrix(z1[,i]),1,
operacion,as.matrix(z2[,i]))
resultado1[,i]<-apply(as.matrix(z1[,i]),1,operacion1,as.matrix(z2[,i]))}}
resultado<-resultado*(-1)
resultado<-apply(resultado,2,sum)
resultado1<-apply(resultado1,2,sum)
resultadofinal<-(resultado/resultado1)
}


SortMat<-function(Mat,Sort){
m<-do.call('order',as.data.frame(Mat[,Sort]))
Mat[m,]
}


——Genera un par de hijos por la pareja——
generar.hijos.de.pareja<-function(padremadre){
n1-length(padremadre)
n<-n1/2
padre<-padremadre[1:n]
madre<-padremadre[(n+1):n1]
k<-sample(1:(n-1),1)
hijo1<-c(padre[1:k],madre[(k+1):n])
hijo2<-c(madre[1:k],padre[(k+1):n])
result<-c(hijo1,hijo2)


——Funcion de selección——
seleccion<-function(población.ordenada){
n<-nrow(población.ordenada)
nmedio<-n %/ %2


npar<-(nmedio %/ %2)*2
padres<-poblacion.ordenada[seq(1,npar,by=2),]
```

```
madres<-poblacion.ordenada[seq(2,npar,by=2),]
hijos<-apply(cbind(padres,madres),1,generar.hijos.de.pareja)
hijos<-matrix(hijos,ncol=ncol(poblacion.ordenada),byrow=T)
nueva.poblacion<-rbind(padres,madres,hijos)
nueva.poblacion
}
——Función mutación——
mutaciones<-function(Población,porcentaje=0.05){
n<-nrow(Poblacion)
n1<-ceiling(n*porcentaje)
k1<-sample(2:n,n1,replace=F)
k2<-sample(1:ncol(Poblacion),n1,replace=T)
Poblacion[k1,k2]<-((Poblacion[kl,k2]+l) % % 2)
Poblacion
}


discriminacion<-function(matriz,numero,corte){
punto.de.corte<-quantile(matriz[,numero],probs = corte)
n<-nrow(matriz)
temp<-array(NA,dim=c(n,1))
for(i in 1:n){
if(matriz[i,numero]<=punto.de.corte){
temp[i,1]<-1
}
else{
temp[i,l]<-0
}
}
matriz<-cbind(matriz,temp)
}
discriminacion1<-function(matriz,numero,corte){
punto.de.corte<-quantile(matriz[,numero],probs = 1-corte)
n<-nrow(matriz)
temp<-array(NA,dim=c(n,1))
f or(i in 1 :n){
if (matriz [i , numero]<=punto.de.corte){
temp[i,l]<-0
```

```
}

else{
temp[i,1]<-1
}
}
matriz<-cbind (matriz, temp)
}
——iteraciones——
uno<-generacion.de.w(cromosomas,variables)
z1<-generacion.de.z(muestra1,uno)
z2<-generacion.de.z(muestra2,uno)
dos<-cbind(cromosomas,indice(z1,z2))
poblacion<-SortMat(dos,(Ngen*variables)+l)

for(ic in 1 : iteraciones) {
hijos<-seleccion(poblacion[,1:(Ngen*variables)])
generacion<-mutaciones(hijos [,1:(Ngen*variables)],porcentaje=0.05)
uno1-generacion.de.w(generacion,variables)
z1<-generacion.de.z(muestra1,uno1)
z2<-generacion.de.z(muestra2,uno1)
dos1<-cbind(generacion,indice(z1,z2))
poblacion<-SortMat(dos1,(Ngen*variables)+1)
}
cromosomas<-poblacion[,l:(Ngen*variables)]
uno<-generacion.de.w(cromosomas,variables)
w.elegido<-as.matrix(uno[1,])
sco.resultados<-generacion.de.z(poblacion.simulada[,1:2],w.elegido)
población.total<-cbind(poblacion.simulada1,sco.resultados)
población.total<-SortMat(población.total,4)
resul<-discriminacion(poblacion.total,4, nrow(muestra2)
/(nrow(muestral)+nrow(muestra2)))
resul-resul[,-4]

——DISCRIMINACION LOGISTICA——
x1<-poblacion.simulada[,1]
```

```
x2<-poblacion.simulada[,2]
y<-poblacion. simulada[,3]
simulacion<-data.frame(x1,x2,y)
datos<-glm(y ˜ x1+x2,family=binomial)
predicho<-array(predict(datos),dim=c(nrow(simulacion),1))


logistica<-cbind(población.simuladal,predicho)
logistica<-SortMat(logística,4)
salida<-diseriminacionKlogística,4,nrow(muestra2)
/(nrow(muestral)+nrow(muestra2)))
salida<-salida[,-4]


——TASA DE CLASIFICACIÓN ERRONEA——
matriz. clasificacion<-function(matriz,res.final){
a<-ifelse(matriz==0&res.final==0,1,0)
b<-ifelse(matriz==l&res.final==l,1,0)
c<-ifelse(matriz==0&res.final==1,1,0)
d<-ifelse(matriz==lres.final==0,1,0)
final<-cbind(a,b,c,d)
e<-apply(final,2,sum)
}
nda<-matriz.clasificacion( resul [,3] ,resul[,4] )
logistica<-matriz.clasificacion( salida[,3],salida [, 4] )
t asa.de.error.nda<- (nda [3] +nda [4] ) /sum (nda)
tasa.de.error.logistica<-(logistica[3]+logistica[4] ) /sum (logística)


tasa.erronea<-cbind (tasa.de.error.nda,tasa.de.error.logistica)
tasas .t [im,] <-tasa. erronea
}
tasas.t
sum(tasas.t[,1])
sum(tasas.t[,2])
```

## 3.2. ALGORITMO GENÉTICO DE BÚSQUEDA ÓPTIMA: UN SUBCONJUNTO DE K-VARIABLES

### 3.2.1. Descripción

Dado un conjunto de variables, un AG del algoritmo busca un subconjunto k-variable que sea óptimo, como sustituto para el sistema del conjunto, con respecto a un criterio dado.

### 3.2.2. Uso

genetic( mat, kmin, kmax = kmin, popsize = 100, nger = 100,
mutate = FALSE, mutprob = 0.01, maxclone = 5, exclude = NULL,
include = NULL, improvement = TRUE, setseed= FALSE, criterion = RM",
pcindices = 1:kmax, initialpop=NULL )

### 3.2.3. Argumentos

**mat:** Una matriz de covarianza o de correlación de las variables de las cuales el k-subconjunto debe ser seleccionado.
**kmin:** Valor del subconjunto más pequeño que se desea.
**kmax:** Valor del subconjunto más grande que se desea.
**popsize:** variable de número entero que indica el tamaño de la población.
**nger:** Variable de número entero que da el número de las generaciones para las cuales el AG funcionará.
**mutate:** Variable lógica que indica si cada hijo experimenta una mutación, con el mutprob de la probabilidad. Por defecto, FALSE.
**mutprob:** Variable que da la probabilidad de cada hijo que experimenta una mutación, si el mutate es TRUE. Por defecto ES 0.01. Los altos valores retrasaron el algoritmo considerablemente y tienden a replegar la misma solución.
**maxclone:** Variable de número entero que especifica el número máximo de réplicas idénticas (clones) de individuos que son aceptables en la población. Sirve para asegurarse de que la población tiene suficiente diversidad genética. Sin embargo, incluso maxclone=0 no garantiza que no hayan repeticiones: solamente prueban al descendiente de pares para clones. Si cualquiera se reproduce se rechazan, y son

substituidos por un subconjunto k-variable elegido al azar, sin fomentar cloness de la copia.
**exclude:** Un vector de las variables (referidas por su fila/columna numeros en mat de la matriz) que deben ser excluidas fuertemente de los subconjuntos.
**include:** Un vector de las variables (referidas por su fila/columna numeros en mat de la matriz) que deben ser incluidas fuertemente en los subconjuntos. improvement un variable lógico que indica ”si” o ”no”, el mejor subconjunto (para cada valor) debe ser pasado como entrada a un algoritmo local de la mejora.
**setseed:** La variable lógica que indica si fijar una semilla inicial para el generador del número al azar, que será reutilizado en las llamadas futuras a esta función siempre que setseed tome nuevamente el valor de verdad, TRUE.
**Criterion:** Variable char, que indica qué criterio debe ser utilizado en la sentencia de la calidad de los subconjuntos. Actualmente, los criterios solamente de RM, RV y de GCD se apoyan, y se refieren como ”RM”, ”RV” o ”GCD”.
**pcindices:** Un vector de las filas de los componentes principales que deben ser utilizados para la comparación con los subconjuntos.
**initialpop:** Vector o matriz tridimensional de población inicial para el AG.

### 3.2.4. Detalles

Para cada valor k (con k extendiéndose de kmin al kmax), una población inicial de popsize k-variables se selecciona aleatoriamente de un sistema completo de variables de p (p que no excede de 300). En cada iteración, los pares popsize/2 se forman entre la población y cada par genera a hijo (un nuevo subconjunto de k-variable) que hereda las características de sus padres (específicamente, hereda todas las variables comunes a padres y a una selección al azar de variables en la diferencia simétrica del maquillaje genético de sus padres). Cada descendiente puede experimentar opcionalmente una mutación (bajo la forma de algoritmo local de la mejora). Alinean a los padres y al descendiente según su valor de criterio, y el mejores popsize de estos k-subconjuntos compondrán la generación siguiente, que se utiliza como la población actual en la iteración subsecuente.
La regla que para el algoritmo es el número de las generaciones (nger).
El usuario puede forzar variables para ser incluidas y/o para ser excluidas de los k-subconjuntos, y puede especificar a una población inicial.
Para cada valor k, el número total de llamadas al procedimiento que computa los valores del criterio es popsize + el nger x popsize/2.

### 3.2.5. Ejemplo

para la ilustración del uso, una pequeña base de datos con pocas iteraciones del algoritmo.

```
data(swiss)
genetic(cor(swiss),3,4,popsize=10,nger=5,criterion=”Rv")


For cardinality k=
4
there is not enough genetic diversity in generation number
5
for acceptable levels of consanguinity (couples differing by at
least 2 genes).

Try reducing the maximum acceptable number of clones (maxclone) or increasing the population size (popsize)

Best criterion value found so far:
0.9590526
$subsets
.           Var.1 Var.2 Var.3
Solution 1     1     2     3
Solution 2     1     2     3
Solution 3     1     2     5
Solution 4     1     2     6
Solution 5     3     4     6
Solution 6     3     4     5
Solution 7     3     4     5
Solution 8     1     3     6
Solution 9     2     4     5
Solution 10    1     3     4
$values
Solution 1 Solution 2 Solution 3 Solution 4 Solution 5 Solution 6
0.9141995 0.9141995 0.9098502 0.9074543 0.9034868 0.9020271
Solution 7 Solution 8 Solution 9 Solution 10
0.9020271 0.8988192 0.8982510 0.8940945
```

$bestvalues
Card.3
0.9141995

$bestsets
Var.1 Var.2 Var.3
1 2 3

## 3.3. FUNCIÓN PLOT

### 3.3.1. Descripción

Traza características del funcionamiento genético de la optimización del algoritmo. El diagrama del defecto indica el mínimo valor de la evaluación, indicando hasta dónde ha progresado el GA.
El diagrama del "hist"demuestra para el cromosoma binario la frecuencia de la selección del gene, es decir, el tiempo en el que un gen en el cromosoma fue seleccionado en la población actual. En caso de los cromosomas float, hará los histogramas para cada una de las variables para indicar los valores seleccionados en la población.
Los "vars"trazan la función de la evaluación contra el valor de la variable. Esto es útil para mirar correlaciones entre las variables y los valores de la evaluación.

### 3.3.2. Uso

plot.rbga (x, type= "default", breaks=10,...)

### 3.3.3. Argumentos

**x:** Un objeto del rbga.
**type:** Un "hist", "vars” o "default".
**breaks:** El número de breaks en un histograma.

### 3.3.4. Ejemplos

```
evaluate <- function(string=c()) {
returnVal = 1 / sum(string);
returnVal
}
```

```
rbga.results = rbga.bin(size=10, mutationChance=0.01, zeroToOneRatio=0.5, evalFunc=evaluate)
plot(rbga.results)
plot(rbga.results, type="hist")
```

## 3.4. CROMOSOMA FLOTANTE

### 3.4.1. Descripción

Un AG basado en R que optimiza, con una función determinada de la evaluación del usuario, un conjunto de valores flotantes. Toma el mínimo y los valores máximos de la entrada para los valores flotantes a optimiza. El grado óptimo es el cromosoma para el cual el valor de la evaluación es el mínimo.
Requiere un método de evalFunc , función que toma como argumento el cromosoma, y un vector de valores flotantes. Además, la optimización de un AG puede ser supervisada fijando un monitorFunc que tome un objeto del rbga como argumento.
Los resultados se pueden visualizar con plot.rbga y resumir con summary.rbga.

### 3.4.2. Uso

```
rbga(stringMin=c(), stringMax=c(),
suggestions=NULL,
popSize=200, iters=100,
mutationChance=NA,
elitism=NA,
```

monitorFunc=NULL, evalFunc=NULL,
showSettings=FALSE, verbose=FALSE)
rbga (stringMin=c (), stringMax=c (), suggestions=NULL, popSize=200, iters=100, mutationChance=NA, elitism=NA, monitorFunc=NULL, evalFunc=NULL, showSettings=FALSE,
verbose=FALSE)

### 3.4.3. Argumentos

**stringMin:** Vector con los valores mínimos para cada gen.
**stringMax:** Vector con los valores máximos para cada gen.
**suggestions:** Lista opcional de cromosomas sugeridos
**popSize:** Tamaño de la población.
**iters:** Número de iteraciones.
**mutationChance:** En el caso de un gen en la mutación del cromosoma. Por el defecto 1/ (size+1). Afecta el tipo de convergencia y el espacio de la búsqueda: resultados bajos indican una convergencia más rápida, mientras que resultados altos aumenta el espacio de la búsqueda.
**elitism:** El número de los cromosomas que se guardan en la generación siguiente. Por defecto está cerca del 20 % del tamaño de la población.
**monitorFunc:** Funcionamiento del método después de cada generación para permitir la supervisión de la optimización.
**evalFunc:** El usuario proveyó método para calcular la función de la evaluación para el cromosoma dado.
**showSettings:** Si es TRUE los ajustes serán impresos los screen. Por defecto FALSE. **verbose:** Si es TRUE el algoritmo será más verbose. Por defecto FALSE.

### 3.4.4. Ejemplos

```
# optimiza dos valores para emparejar pi y sqrt(50)
evaluate <- function(string=c()) {
returnVal = NA;
if (length(string) == 2) {
returnVal = abs(string[1]-pi) + abs(string[2]-sqrt(50));
}
```

```
else {
stop(”Expecting a chromosome of length 2!”);
}
returnVal
}


monitor <- function(obj) {
# plot de la población
xlim = c(obj$stringMin[1], obj$stringMax[1]);
ylim = c(obj$stringMin[2], obj$stringMax[2]);
plot(obj$population, xlim=xlim, ylim=ylim, xlab=”pi”, ylab=”sqrt(50)”); }


rbga.results = rbga(c(1, 1), c(5, 10), monitorFunc=monitor, evalFunc=evaluate,
verbose=TRUE, mutationChance=0.01)

plot(rbga.results) plot(rbga.results, type="hist") plot(rbga.results, type="vars")
```

## 3.5. CROMOSOMA BINARIO

### 3.5.1. Descripción

R basado en AG que optimiza, con una función determinada de la evaluación del usuario, un cromosoma binario que se puede utilizar para la selección de variables. El grado óptimo es el cromosoma para el cual el valor de la evaluación es mínimo. Requiere un método de evalFunc que tome como discusión el cromosoma binario, un vector de ceros y unos. Además, la optimización de AG puede ser supervisada fijando un monitorFunc que tome un objeto del rbga como argumento.
Los resultados se pueden visualizar con plot.rbga y resumir con summary.rbga.

### 3.5.2. Uso

```
rbga.bin(size=10,
suggestions=NULL,
popSize=200, iters=100,
mutationChance=NA,
elitism=NA, zeroToOneRatio=10,
```

monitorFunc=NULL, evalFunc=NULL,
showSettings=FALSE, verbose=FALSE)

### 3.5.3. Argumentos

**size:** El número de genes en el cromosoma.
**popSize:** El tamaño de la población.
**iters:** El número de iteraciones.
**mutationChance:** En el caso de un gen en la mutación del cromosoma. Por el defecto 1/ (size+1). Afecta el tipo de convergencia y el espacio de la búsqueda: resultados bajos en una convergencia más rápida, mientras que resultado altos aumenta el espacio de la búsqueda.
**elitism:** El número de los cromosomas que se guardan en la generación siguiente. Por defecto está cerca del 20 % del tamaño de la población.
**zeroToOneRatio:** El cambio por un cero para las mutaciones y la inicialización. Esta opción se utiliza para controlar el número de los genes del sistema en el cromosoma. Por ejemplo, al hacer la selección de la variable este parámetro debe ser fijado.
**monitorFunc:** Funcionamiento del método después de cada generación para permitir la supervisión de la optimización.
evalFunc El usuario proveyó método para calcular la función de la evaluación para el cromosoma dado.
**showSettings:** Si es TRUE los ajustes serán impresos para el screen. Por defecto FALSE.
**verbose:** Si es TRUE el algoritmo será más verbose. Por defecto FALSE.
**Suggestions:** Lista opcional de cromosomas sugeridos.

### 3.5.4. Ejemplos

```
# evalúa una optimización muy simplista
evaluate <- function(string=c()) {
returnVal = 1 / sum(string);
returnVal
}
```

```
rbga.results = rbga.bin(size=10, mutationChance=0.01, zeroToOneRatio=0.5,
evalFunc=evaluate)

plot(rbga.results)

# en este ejemplo las cuatro variables en la matriz son completadas con 36 variables al azar. La selección aleatoria debe encontrar las cuatro variables originales.
# # Not run:
data(iris)
library(MASS)
X <- cbind(scale(iris[,1:4]), matrix(rnorm(36*150), 150, 36))
Y <- iris[,5]

iris.evaluate <- function(indices) {
result = 1
if (sum(indices) >2) {
huhn <- lda(X[,indices==1], Y, CV=TRUE)$posterior
result = sum(Y != dimnames(huhn)[[2]][apply(huhn, 1,
function(x)
which(x == max(x)))]) / length(Y)
}
result
}

monitor <- function(obj) {
minEval = min(obj$evaluations);
plot(obj, type="hist");
}

woppa <- rbga.bin(size=40, mutationChance=0.05, zeroToOneRatio=10,
evalFunc=iris.evaluate, verbose=TRUE, monitorFunc=monitor)
# # End(Not run)

# # Not run:
library(pls.pcr)
data(NIR)

numberOfWavelenghts = ncol(NIR$Xtrain)
evaluateNIR <- function(chromosome=c()) {
returnVal = 100
minLV = 2
if (sum(chromosome) <minLV) {
```

```
returnVal
}
else {
xtrain = NIR$Xtrain[,chromosome == 1]; pls.model = pls(xtrain, NIR$Ytrain,
validation="CV", grpsize=1,
ncomp=2:min(10,sum(chromosome)))
returnVal = pls.model$val$RMS[pls.model$val$nLV-(minLV-1)] returnVal
}
}

monitor <- function(obj) { minEval = min(obj$evaluations);
filter = obj$evaluations == minEval;
bestObjectCount = sum(rep(1, obj$popSize)[filter]);

if (bestObjectCount >1) {
bestSolution = obj$population[filter,][1,];
} else {
bestSolution = obj$population[filter,];
}
outputBest = paste(obj$iter, " #selected=", sum(bestSolution),
"Best (Error=", minEval, "): ", sep="");
for (var in 1:length(bestSolution)) {
outputBest = paste(outputBest,
bestSolution[var], " ",
sep="");
}
outputBest = paste(outputBest, sep="");


cat(outputBest);
}

nir.results = rbga.bin(size=numberOfWavelenghts, zeroToOneRatio=10,
evalFunc=evaluateNIR, monitorFunc=monitor,
popSize=200, iters=100, verbose=TRUE)
# # End(Not run)
```

## 3.6. FUNCIÓN SUMMARY

### 3.6.1. Descripción

Resume los resultados del AG.

### 3.6.2. Uso

summary.rbga(object, echo=FALSE, ...)

### 3.6.3. Argumentos

**object:** Un objeto rbga. **echo:** Si es TRUE, el summary será impreso a STDOUT así como retornado.

### 3.6.4. Ejemplos

```
evaluate <- function(string=c()) {
returnVal = 1 / sum(string);
returnVal
}

rbga.results = rbga.bin(size=10, mutationChance=0.01, zeroToOneRatio=0.5,
evalFunc=evaluate)

summary(rbga.results)
```

# 4. APLICACIÓN

Una empresa de energía desea maximizar la producción de electricidad de una de sus plantas. En ella la electricidad se produce mediante vapor a partir de carbón y se desea maximizar la producción de vapor que se hace a partir de la combinación de dos tipos de carbón: Carbón A y carbón B. Cada tonelada de carbón A produce 24 unidades de vapor y de carbón B 20. Pero existen ciertas restricciones. La cantidad máxima de emisiones de humo por hora esta limitada a 12 kg. Cada tonelada de carbón A produce 0.5 kg de humo y de B 1 kg. El sistema de cinta transportadora que traslada el carbón de los depósitos al pulverizador tiene una capacidad de 20 ton/h. La capacidad máxima del pulverizador es de 16 ton/h para carbón A ó 24 ton/h para carbón B.

Definamos las restricciones y la función objetivo:

$$XA: ton\ de\ carbon\ A/hora$$
$$XB: ton\ de\ carbon\ A/hora$$
$$Z = 24XA+20XB$$
$$0.5XA+XB \leq 12kg/h$$

XA+XB≤ 20 Primero se debe codificar en forma binaria como se hará a continuación:

Nótese que esto es en el caso de que todos los valores binarios sean 1 ya que al sumarlos debe de dar la restricción del valor XA, tomando como valor máximo 16.

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | Total |
| 2 | 2 | 6 | 4 | 2 | 16 |

De igual forma para XB, en este caso el màximo valor es 4.

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | Total |
| 1 | 2 | 1 | 4 |

El siguiente paso es hacer competir a los individuos entre sí. Este proceso se conoce como selección.

**SELECCIÓN (XA)**

| N° del individuo | Individuo en binario | Valor de XA | Pareja aleatoria |
|---|---|---|---|
| 1 | (1,1,1,1,0) | 14 | 5 |
| 2 | (1,1,1,0,0) | 10 | 3 |
| 3 | (1,1,0,0,0) | 4 | 1 |
| 4 | (0,0,0,1,1) | 6 | 6 |
| 5 | (0,0,1,1,1) | 12 | 2 |
| 6 | (0,1,1,1,1) | 14 | 4 |

**SELECCIÓN (XB)**

| N° del individuo | Individuo en binario | Valor de XA | Pareja aleatoria |
|---|---|---|---|
| 1 | (1,1,0) | 3 | 4 |
| 2 | (1,0,1) | 2 | 1 |
| 3 | (0,1,1) | 3 | 5 |
| 4 | (0,0,1) | 1 | 3 |
| 5 | (1,0,0) | 1 | 6 |
| 6 | (0,1,0) | 2 | 2 |

Ahora se evaluan los individuos en la función:

| | Individuo XA | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Individuo XB | 1 | 396 | 300 | 156 | 204 | 348 | 396 |
| | 2 | 376 | 280 | 136 | 184 | 328 | 376 |
| | 3 | 396 | 300 | 156 | 204 | 348 | 396 |
| | 4 | 356 | 260 | 116 | 164 | 308 | 356 |
| | 5 | 356 | 260 | 116 | 164 | 308 | 356 |
| | 6 | 376 | 280 | 136 | 184 | 328 | 376 |

Como se puede observar el mejor individuo para XA puede ser 1,6 y para XB 1,3 conformando las parejas (1,1),(1,3),(6,1) y (6,3) ya que al evaluar las parejas en la función el resultado es el mayor con f = 396.

Una manera de realizar el proceso de selección es mediante un torneo entre dos. A cada individuo de la población se le asigna una pareja y entre ellos se establece un torneo: el mejor genera dos copias y el peor se desecha. La columna (4) indica

la pareja asignada a cada individuo.Después de realizar el proceso de selección, la población que tenemos es la mostrada en la columna (2) de la siguente tabla.

**CRUCE (XA)**

| N° del individuo | Individuo en binario | Pareja aleatoria | Punto de cruce |
|---|---|---|---|
| 1 | (1,1,1,1,0) | 3 | 1 |
| 2 | (1,1,1,1,0) | 6 | 2 |
| 3 | (1,1,1,0,0) | 2 | 1 |
| 4 | (1,1,1,0,0) | 1 | 1 |
| 5 | (0,1,1,1,1) | 4 | 2 |
| 6 | (0,1,1,1,1) | 5 | 2 |

**CRUCE (XB)**

| N° del individuo | Individuo en binario | Pareja aleatoria | Punto de cruce |
|---|---|---|---|
| 1 | (1,1,0) | 2 | 1 |
| 2 | (1,1,0) | 5 | 2 |
| 3 | (1,1,0) | 1 | 1 |
| 4 | (1,1,0) | 6 | 1 |
| 5 | (0,1,1) | 4 | 2 |
| 6 | (0,1,1) | 3 | 1 |

**POBLACIÓN TRAS EL CRUCE (XA)**

| N° del individuo | Individuo en binario | Valor de XA |
|---|---|---|
| 1 | (1,1,1,0,0) | 10 |
| 2 | (1,1,1,1,0) | 14 |
| 3 | (1,1,1,1,1) | 16 |
| 4 | (0,1,1,1,0) | 12 |
| 5 | (1,1,1,1,0) | 14 |
| 6 | (1,1,1,0,0) | 10 |

**POBLACIÓN TRAS EL CRUCE (XB)**

| N° del individuo | Individuo en binario | Valor de XA |
|---|---|---|
| 1 | (1,1,0) | 3 |
| 2 | (1,1,0) | 3 |
| 3 | (1,1,1) | 4 |
| 4 | (0,1,0) | 2 |
| 5 | (1,1,0) | 3 |
| 6 | (1,1,0) | 3 |

De nuevo se evaluan los individuos en la función:

| | Individuo XA | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Individuo XB | 1 | 300 | 396 | 444 | 348 | 396 | 300 |
| | 2 | 300 | 396 | 444 | 348 | 396 | 300 |
| | 3 | 320 | 416 | 464 | 368 | 416 | 320 |
| | 4 | 280 | 376 | 424 | 328 | 376 | 280 |
| | 5 | 300 | 396 | 444 | 348 | 396 | 300 |
| | 6 | 300 | 396 | 444 | 348 | 396 | 300 |

Ahora el mejor individuo para XA es 3 y para XB es 3 con un vajor de ***464***; ¿Qué quiere decir esto? Simplemente que los individuos después de la selección y el cruce son mejores que antes de estas transformaciones.
Esta manera de proceder se repite tantas veces como número de iteraciones se fijes. Y ¿cuál es el óptimo? En realidad un algoritmo genético no garantiza la obtención del óptimo como ya hemos mencionado anteriormente pero, si está bien construido, proporcionará una solución razonablemente buena. Puede que se obtenga el óptimo, pero el algoritmo no confirma que lo sea. También es buena idea ir guardando la mejor solución de todas las iteraciones anteriores y al final quedarte con la mejor solución de las exploradas.

# 5. CONCLUSIONES

Como se ha podido observar, una de las principales ventajas de los AG puede observarse en su sencillez; puesto que se necesita muy poca información sobre el espacio de búsqueda ya que se trabaja sobre un conjunto de soluciones o parámetros codificados (hipótesis o individuos). Al igual que sus campos de aplicación, se puede afirmar que es un método muy completo de optimización, puesto que sus áreas de estudio son muy amplias, y se puede ver generalizado en muchos sucesos cotidianos.

Se ha observado de igual forma que los AG están indicados para resolver todo tipo de problemas que se puedan expresar como un problema de optimización donde se define una representación adecuada para las soluciones y para la función a optimizar. Se busca una solución por aproximación de la población, en lugar de una aproximación punto a punto.

Probablemente el punto más delicado de todo se encuentra en la definición de la función objetivo, ya que de su eficiencia depende la obtención de un buen resultado. El resto del proceso es siempre el mismo para todos los casos.
La programación mediante AG supone un nuevo enfoque que permite abarcar todas aquellas áreas de aplicación donde no se sabe de ante mano como resolver el problema.

También es importante anotar que a pesar de que es una técnica muy buena, en el caso de un problema especifico en donde se sepa que para su optimización se puede utilizar otro método, pues en este caso lo más recomendable es hacerlo por el otro método, ya que con seguridad se encontrara una solución más optima (sino la más optima) de la que se hubiese podido encontrar con un AG.

En la parte final Se pueden observar las diferentes aplicaciones que tiene los AG en el paquete estadístico R, al igual que algunas de las funciones que se utilizan para su aplicación, según sea el caso.

Por ultimo se detalla una de las muchas aplicaciones que tiene este método, que es la optimización; allí se observa la forma de proceder de este método tan completo y a la ves tan ”aparentemente"sencillo.

# Referencias

[1] http://www.dii.uchile.cl/ ceges/publicaciones/ceges73.pdf

[2] http://taylor.us.es/componentes/miguelangel/algoritmosgeneticos.pdf

[3] http://www.elrincondelprogramador.com/default.asp?pag=articulos/ leer.asp&id=6

[4] http://www.utp.edu.co/php/revistas/ScientiaEtTechnica/docsFTP/ 21273285-290.pdf

[5] http://descargas.cervantesvirtual.com/servlet/SirveObras/120394186289 25940987435/002760_11.pdf

[6] http://eddyalfaro.galeon.com/geneticos.html

[7] http://www.uv.mx/aguerra/teaching/MIA/MachineLearning/ clase08.pdf

[8] http://tigre.aragon.unam.mx/geneticos/indice.htm

[9] http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t2geneticos.pdf

[10] http://147.96.80.155/sistemaoptimizacion/AlgoritmoGenetico.htm

[11] http://www.redcientifica.com/gaia/ce/agsp_c.htm

[12] http://www.frro.utn.edu.ar/isi/algoritmosgeneticos/ html_data/3algoritmos/Algoritmo.htm

[13] http://www.redcientifica.com/imprimir/doc199904260011.html

[14] http://www.divulgamat.net/weborriak/TestuakOnLine/02-03/ PG02-03-iglesias.pdf

[15] http://www.uv.es/asepuma/X/J24C.pdf

[16] http://www.uv.es/asepuma/X/J01C.pdf

[17] http://webdiis.unizar.es/j̃campos/EDA/ea/slides/9- Algoritmos %20geneticos.pdf

[18] http://pbil.univ-lyon1.fr/library/subselect/html/genetic.html