UNIVERSIDAD OBERTA CATALUNYA

PROYECTO FIN DE MASTER

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

PROFESOR : Samir Kanaan Izquierdo

AUTOR : Pablo Solar Rodríguez

OCTUBRE 2014

MEMORIA PRESENTADA POR

D. Pablo Solar Rodríguez

PARA OPTAR AL TÍTULO DE

MÁSTER Universitario de Ingeniería Informática

Documentos que acompañan al Proyecto

El presente proyecto consta de los documentos que a continuación se citan. Cada uno de ellos está correlativo y hacen referencia al volumen en el cual se pueden encontrar. Además, son independientes y poseen una numeración propia.

Volumen 1

- **Capítulo 1 :** Los Algoritmos Genéticos y la Regresión Simbólica
- **Capítulo 2 :** La Regresión Simbólica. Método de Resolución. Diseño y Codificación
- **Capítulo 3 :** Implementación
- **Capítulo 4 :** Experimentación y Resultados
- **Capítulo 5 :** Código Fuente + Bibliografía

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

Capítulo 1 : Los Algoritmos Genéticos y la Regresión Simbólica

Pablo Solar Rodríguez

ÍNDICE

LISTA DE FIGURAS

Capítulo 1 (I) : Los Algoritmos Genéticos

En la naturaleza, los individuos de una población compiten constantemente con otros por recursos tales como comida, agua y refugio. Los individuos que tienen más éxito en la lucha por los recursos tienen mayores probabilidades de sobrevivir y generalmente una descendencia mayor. Al contrario, los individuos peor adaptados tienen un menor número de descendientes, o incluso ninguno. Esto implica que los genes de los individuos mejor adaptados se propagarán a un número cada vez mayor de individuos de las sucesivas generaciones.

La combinación de características buenas de diferentes ancestros puede originar en ocasiones que la descendencia esté incluso mejor adaptada al medio que los padres. De esta manera, las especies evolucionan adaptándose más y más al medio a medida que transcurren las nuevas poblaciones.

Pero la adaptación de un individuo al medio no sólo está determinada por su composición genética. Influyen otros factores como el aprendizaje, en ocasiones adquirido por el método de prueba y error, en ocasiones adquirido por imitación del comportamiento de los padres.

Y, si algo funciona bien, ¿por qué no imitarlo?. La respuesta a esta pregunta lleva directamente a los orígenes de la computación evolutiva. Durante millones de años las diferentes especies se han adaptado para poder sobrevivir en un medio cambiante. De la misma manera se podría tener una población de potenciales soluciones a un problema de las que se irían seleccionando las mejores hasta que se adaptasen perfectamente al medio, en este caso el problema a resolver. En términos muy generales se podría definir la computación evolutiva como una familia de modelos computacionales inspirados en la evolución.

En Ingeniería Informática, la computación evolutiva es un subcampo de la Inteligencia Artificial (en particular de la Inteligencia Computacional) que implican problemas de optimización combinatioria.Utiliza progresos iterativos, como el crecimiento o desarrollo de una población, que es seleccionada al azar y se somete a una búsqueda guiada para obtener el fin deseado. Estos procesos se han inspirado en los mecanismos biológicos de la evolución.

El uso de las teorías y los principios Darwinianos para la solución de problemas se originó en los años cincuenta, aunque no fue hasta la década de los 60 cuando tres interpretaciones de esta misma idea fueron desarrolladas en campos diferentes. La programación evolutiva fue introducida por Lawrence J. Fogel, mientras que Ingo Rechenberg y Hans-Paul Schwefel idearon las estrategias evolutivas; por otra parte, John H. Holland hizo lo mismo con algo que denominó **algoritmos genéticos**. Estos tres conceptos se trabajaron de forma independiente durante años, hasta que se unificaron, junto a la programación genética, como diferentes técnicas de una misma tecnología, la ya citada computación evolutiva.

1.1.1 ¿Qué son los Algoritmos Genéticos?

Los Algoritmos genéticos (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.

Una definición bastante completa de un algoritmo genético es la propuesta por John Koza:

"Es un algoritmo matemático altamente paralelo que transforma un conjunto de objetos matemáticos individuales con respecto al tiempo usando operaciones modeladas basadas en los principios Darwinianos de reproducción y supervivencia del más apto, y tras haberse presentado de forma natural una serie de operaciones genéticas de entre las que destaca la recombinación sexual. Cada uno de estos objetos matemáticos suele ser una cadena de información (letras o números) de longitud fija que se ajusta al modelo de las cadenas de cromosomas, y se les asocia con una cierta función matemática que refleja su actitud."

La definición anterior no es sencilla, y en las siguientes líneas se tratará de explicar de manera más clara y sencilla.

Como se ha dicho, los AG son una técnica de programación que pretende imitar el proceso evolutivo como estrategia para resolver problemas. Dado un problema específico, la entrada de un AG es un conjunto de posibles soluciones para dicho problema (población del AG), codificadas de alguna manera (normalmente como tiras de 0's y 1's), y una métrica llamada *función fitness* que permite evaluar a cada individuo (denominados cromosomas o genotipos) de forma cuantitativa. Estos candidatos pueden ser soluciones que ya se saben válidas y que funcionan, con el objetivo de que el AG los mejore, o, lo que sucede más a menudo, que se generen de manera aleatoria.

En la naturaleza esto equivaldría al grado de efectividad de un organismo para competir por unos determinados recursos. Cuanto mayor sea la adaptación de un individuo al problema, mayor será la probabilidad de que el mismo sea seleccionado para reproducirse, cruzando su material genético con otro individuo seleccionado de igual forma. Este cruce producirá nuevos individuos . descendientes de los anteriores . los cuales comparten algunas de las características de sus padres. Cuanto menor sea la adaptación de un individuo, menor será la probabilidad de que dicho individuo sea seleccionado para la reproducción, y por tanto de que su material genético se propague en sucesivas generaciones.

De esta manera se produce una nueva población de posibles soluciones, la cual reemplaza a la anterior y verifica la interesante propiedad de que contiene una mayor proporción de buenas características en comparación con la población

anterior. Así a lo largo de las generaciones las buenas características se propagan a través de la población. Favoreciendo el cruce de los individuos mejor adaptados, van siendo exploradas las áreas más prometedoras del espacio de búsqueda. Si el Algoritmo Genético ha sido bien diseñado, la, población convergerá hacia una solución óptima del problema.

La aplicación más común de los algoritmos genéticos 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 estar delimitado dentro de un cierto rango.
- Debe poderse definir una función de aptitud que nos indique qué tan buena o mala es una cierta respuesta.
- Las soluciones deben codificarse de una forma que resulte relativamente fácil de implementar en la computadora.

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 preferentemente cuando exista un rango de soluciones relativamente pequeño.

La **función fitness** no es más que la función objetivo de nuestro problema de optimización. El algoritmo genético únicamente maximiza, pero la minimización puede realizarse fácilmente utilizando el recíproco de la función maximizante (debe cuidarse, por supuesto, que el recíproco de la función no genere una división por cero). Una característica que debe tener esta función es que tiene ser capaz de "castigar" a las malas soluciones, y de "premiar" a las buenas, de forma que sean estas últimas las que se propaguen con mayor rapidez.

La **codificación** más común de las soluciones es a través de cadenas binarias, aunque se han utilizado también números reales y letras. El primero de estos esquemas ha gozado de mucha popularidad debido a que es el que propuso originalmente Holland, y además porque resulta muy sencillo de implementar.

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 hibridándolas con los AG.

1.1.2 El Algoritmo Genético Simple

El Algoritmo Genético Simple, también denominado Canónico, se representa en la Figura . 1. Como se verá a continuación, se necesita una **codificación** o representación del problema, que resulte adecuada al mismo. Además se requiere una **función fitness** o de ajuste al problema, la cual asigna un número real a cada posible solución codificada. Durante la ejecución del algoritmo, los padres deben ser seleccionados (**selección**) para la reproducción, a continuación dichos padres seleccionados se cruzarán (**cruce**) generando dos hijos, sobre cada uno de los cuales actuará un operador de **mutación**. El resultado de la combinación de las anteriores funciones será un conjunto de individuos (posibles soluciones al problema), los cuales en la evolución del Algoritmo Genético formarán parte de la siguiente población.

```
BEGIN /* Algoritmo Genetico Simple */
      Generar una poblacion inicial.
      Computar la funcion de evaluacion de cada individuo.
      WHILE NOT Terminado DO
      BEGIN /* Producir nueva generacion */
            FOR Tamaño poblacion/2 DO
            BEGIN /*Ciclo Reproductivo */
                  Seleccionar dos individuos de la anterior generacion,
                  para el cruce (probabilidad de seleccion proporcional
                  a la funcion de evaluacion del individuo).
                  Cruzar con cierta probabilidad los dos
                  individuos obteniendo dos descendientes.
                  Mutar los dos descendientes con cierta probabilidad.
                  Computar la funcion de evaluacion de los dos
                  descendientes mutados.
                  Insertar los dos descendientes mutados en la nueva generacion.
            END
            IF    la poblacion ha convergido THEN
                  Terminado := TRUE
      END
END
```

Figura 1. Pseudocódigo del AG Simple

Nótese que, evidentemente, trabajando con un única población no se puede decir que se pase a la siguiente generación cuando se llene la población, pues siempre está llena. En este caso el paso a la siguiente generación se producirá una vez que se hayan alcanzado cierto número de cruces y mutaciones.

Codificación

Cualquier solución potencial a un problema puede ser presentada dando valores a una serie de parámetros. El conjunto de todos los parámetros (genes en la terminología de AG) se codifica en una cadena de valores denominada cromosoma.

El conjunto de los parámetros representado por un cromosoma particular recibe el nombre de genotipo. El genotipo contiene la información necesaria para la construcción del organismo, es decir, la solución real al problema, denominada fenotipo. Por ejemplo, en términos biológicos, la información genética contenida en el ADN de un individuo sería el genotipo, mientras que la expresión de ese ADN (el propio individuo) sería el fenotipo.

Desde los primeros trabajos de John Holland la codificación suele hacerse mediante valores binarios. Se asigna un determinado número de bits a cada parámetro y se realiza una discretización de la variable representada por cada gen. El número de bits asignados dependerá del grado de ajuste que se desee alcanzar. Evidentemente no todos los parámetros tienen porque estar codificados con el mismo número de bits. Sin embargo, también pueden existir representaciones que codifiquen directamente cada parámetro con un valor entero, real o en punto flotante. A pesar de que se acusa a estas representaciones de degradar el paralelismo implícito de las representaciones binarias, permiten el desarrollo de operadores genéticos más específicos al campo de aplicación del Algoritmo Genético.

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.

Se efectuaron estudios teóricos, obteniendo como conclusión que el tamaño óptimo de la población para ristras de longitud L, 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.

Sin embargo, otros investigadores, basándose en evidencia empírica sugieren que un tamaño de población comprendida entre L y 2L es suficiente para atacar con éxito los problemas por el considerados, aunque normalmente se realizan pruebas con poblaciones de bastante más tamaño.

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 podría preguntar qué es lo que sucedería si los individuos de la población inicial se obtuviesen como resultado de alguna técnica heurística o de optimización local, es decir, de partir inicialmente con valores ya válidos que sean soluciones factibles al problema a solucionar. 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.

Función Fitness

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

Idealmente 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 Algoritmo Genético 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 podríamos 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 Algoritmos Genéticos 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.

Selección

Los algoritmos de selección serán los encargados de escoger qué individuos van a disponer de oportunidades de reproducirse y cuáles no.

Puesto que se trata de imitar lo que ocurre en la naturaleza, 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:

- *Selección Elitista* : en este modelo, se fuerza a que el mejor individuo de la población en el tiempo t, sea elegido como padre. También existe la posibilidad de, insertar este individuo al generar una población inicial vacía al comienzo de una nueva iteración.

- *Ruleta o Selección Proporcional* : con este método, la probabilidad que tiene un individuo de reproducirse es proporcional a su valor de función fitness, es decir, a su adaptación. Se define un rango de valores desde 1 hasta la suma de los fitness de los individuos. El número al azar será un número aleatorio forzosamente comprendido entre el tamaño del rango. Cada uno de dichos individuos tendrá tantos números consecutivos en la ruleta como el valor de su fitness. En cada tirada de la ruleta, será seleccionado el individuo que posea el número que ha salido. Se trata de una selección con reemplazamiento, de modo que un mismo individuo puede ser elegido varias veces. Cuando se ha seleccionado el número de individuos requerido, se realiza un emparejamiento de los mismos de manera aleatoria para a continuación aplicar algún operador de reproducción a cada una de las parejas.

- *Selección por Ránking* : 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.

- *Selección por k-Torneo* : se selecciona un grupo de k individuos (normalmente k = 2, torneo binario) y se genera un número aleatorio entre 0 y 1. Si este número es menor que un cierto umbral X (usualmente 0,75), se selecciona para reproducirse al individuo con mejor adaptación, y si este número es menor que X, 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.

Elegir uno u otro método de selección determinará la estrategia de búsqueda del Algoritmo Genético. Si se opta por un método con una alta presión de selección, se centra la búsqueda de las soluciones en un entorno próximo a las mejores soluciones actuales. Por el contrario, optando por una presión de selección menor se deja el camino abierto para la exploración de nuevas regiones del espacio de búsqueda.

Existen muchos otros algoritmos de selección. Unos buscan mejorar la eficiencia computacional, otros el número de veces que los mejores o peores individuos pueden ser seleccionados. Algunos de estos algoritmos son muestreo determinístico, escalamiento sigma, selección por jerarquías, estado uniforme, sobrante estocástico, brecha generacional, etc.

Cruce

Una vez seleccionados los individuos, éstos son recombinados para producir la descendencia que se insertará en la siguiente generación. Tal y como se ha indicado anteriormente el cruce es una estrategia de reproducción sexual. Su importancia para la transición entre generaciones es elevada puesto que las tasas de cruce con las que se suele trabajar rondan el 90%.

Los diferentes métodos de cruce podrán operar de dos formas diferentes. Si se opta por una estrategia destructiva los descendientes se insertarán en la población temporal aunque sus padres tengan mejor ajuste. Por el contrario utilizando una estrategia no destructiva la descendencia pasará a la siguiente generación únicamente si supera la bondad del ajuste de los padres (o de los individuos a reemplazar).

La idea principal del cruce se basa en que, si se toman dos individuos correctamente adaptados al medio y se obtiene una descendencia que comparta genes de ambos, existe la posibilidad de que los genes heredados sean precisamente los causantes de la bondad de los padres. Al compartir las características buenas de dos individuos, la descendencia, o al menos parte de ella, debería tener una bondad mayor que cada uno de los padres por separado. Si el cruce no agrupa las mejores características en uno de los hijos y la descendencia tiene un peor ajuste que los padres no significa que se esté dando un paso atrás. Optando por una estrategia de cruce no destructiva garantizamos que pasen a la siguiente generación los mejores individuos. Si, aún con un ajuste peor, se opta por insertar a la descendencia, y puesto que los genes de los padres continuarán en la población - aunque dispersos y posiblemente levemente modificados por la mutación - en posteriores cruces se podrán volver a obtener estos padres, recuperando así la bondad previamente perdida.

Existen multitud de algoritmos de cruce, sin embargo los más empleados son los que se detallarán a continuación:

- Cruce en 1 punto : es la más sencilla de las técnicas de cruce. Una vez seleccionados dos individuos se cortan sus cromosomas por un punto seleccionado aleatoriamente para generar dos segmentos diferenciados en cada uno de ellos: la cabeza y la cola. Se intercambian las colas entre los dos individuos para generar los nuevos descendientes. De esta manera ambos descendientes heredan información genética de los padres, tal y como puede verse en la figura 2.

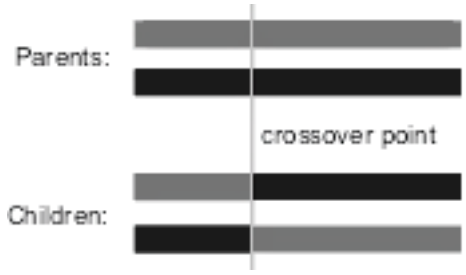


Figura 2. Ejemplo Cruce en 1 Punto

- Cruce en 2 puntos : se trata de una generalización del cruce de 1 punto. En vez de cortar por un único punto los cromosomas de los padres como en el caso anterior se realizan dos cortes. Para generar la descendencia se escoge el segmento central de uno de los padres y los segmentos laterales del otro padre (ver figura 3).

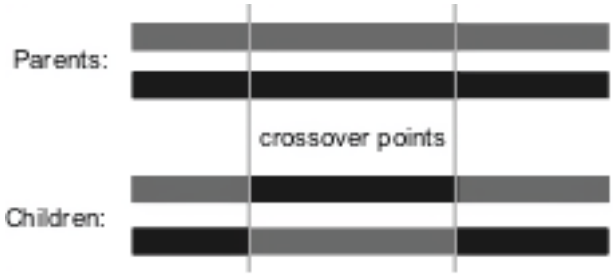


Figura 3. Ejemplo Cruce en 2 Puntos

Generalizando se pueden añadir más puntos de cruce dando lugar a algoritmos de cruce multipunto, aunque es desaconsejable, ya que es más fácil que los segmentos originados por separado quizás pierdan las características de bondad que poseían conjuntamente.

- Cruce Uniforme : El cruce uniforme es una técnica completamente diferente de las vistas hasta el momento. Cada gen de la descendencia tiene las mismas probabilidades de pertenecer a uno u otro padre. Aunque se puede implementar de muy diversas formas, la técnica implica la generación de una máscara de cruce con valores binarios. Si en una de las posiciones de la máscara hay un 1, el gen situado en esa posición en uno de los descendientes se copia del primer padre. Si por el contrario hay un 0 el gen se copia del segundo padre. Para producir el segundo descendiente se intercambian los papeles de los padres, o bien se intercambia la interpretación de los unos y los ceros de la máscara de cruce. Tal y como se puede apreciar en la figura 4, la descendencia contiene una mezcla de genes de cada uno de los padres.

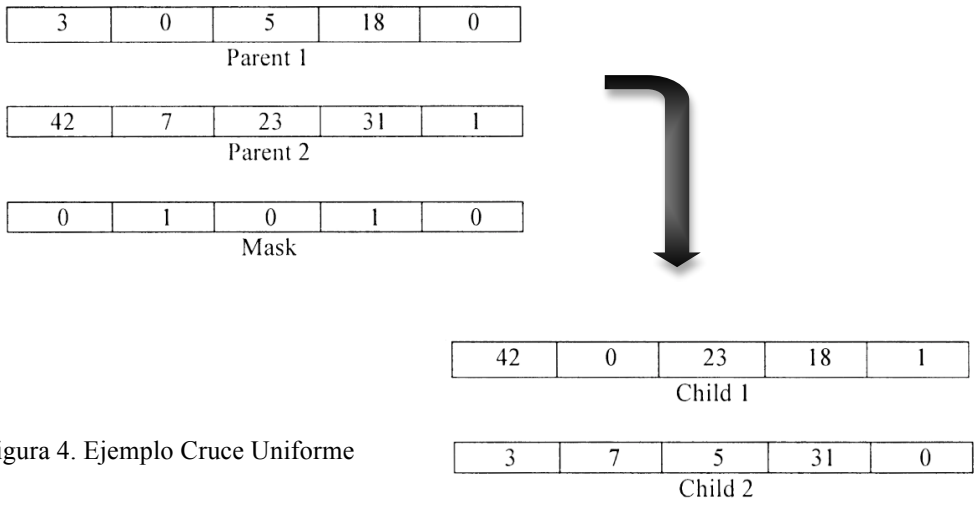


Figura 4. Ejemplo Cruce Uniforme

Algoritmos de Reemplazo

Cuando en vez de trabajar con una población temporal se hace con una única población, sobre la que se realizan las selecciones e inserciones, deberá tenerse en cuenta que para insertar un nuevo individuo deberá de eliminarse previamente otro de la población.

También se puede optar por una de estas estrategias cuando se trabaja con una población temporal auxiliar que reemplace a la anterior en cada nueva iteración del algoritmo.

Existen diferentes métodos de reemplazo:

- *Aleatorio :* el nuevo individuo se inserta en un lugar cualquiera de la población.
- *Reemplazo de padres :* se obtiene espacio para la nueva descendencia liberando el espacio ocupado por los padres.
- *Reemplazo de similares :* una vez obtenido el ajuste de la descendencia se selecciona un grupo de individuos (entre seis y diez) de la población con un ajuste similar. Se reemplazan aleatoriamente los que sean necesarios.
- *Reemplazo de los peores :* de entre un porcentaje de los peores individuos de la población se seleccionan aleatoriamente los necesarios para dejar sitio a la descendencia.

Copia

La copia es la otra estrategia reproductiva para la obtención de una nueva generación a partir de la anterior. A diferencia del cruce, se trata de una estrategia de reproducción asexual. Consiste simplemente en la copia de un individuo en la nueva generación.

El porcentaje de copias de una generación a la siguiente es relativamente reducido, pues en caso contrario se corre el riesgo de una convergencia prematura de la población hacia ese individuo. De esta manera el tamaño efectivo de la población se reduciría notablemente y la búsqueda en el espacio del problema se focalizaría en el entorno de ese individuo.

Lo que generalmente se suele hacer es seleccionar dos individuos para el cruce, y si éste finalmente no tiene lugar, se insertan en la siguiente generación los individuos seleccionados.

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.

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.

Aunque se pueden seleccionar los individuos directamente de la población actual y mutarlos antes de introducirlos en la nueva población, la mutación se suele utilizar de manera conjunta con el operador de cruce. Primeramente se seleccionan dos individuos de la población para realizar el cruce. Si el cruce tiene éxito entonces uno de los descendientes, o ambos, se muta con cierta probabilidad Pm. Se imita de esta manera el comportamiento que se da en la naturaleza, pues cuando se genera la descendencia siempre se produce algún tipo de error, por lo general sin mayor trascendencia, en el paso de la carga genética de padres a hijos.

La probabilidad de mutación es muy baja, generalmente menor al 1%. Esto se debe sobre todo a que los individuos suelen tener un ajuste menor después de mutados. Sin embargo se realizan mutaciones para garantizar que ningún punto del espacio de búsqueda tenga una probabilidad nula de ser examinado.

Tal y como se ha comentado, la mutación más usual es el reemplazo aleatorio. Este consiste en variar aleatoriamente un gen de un cromosoma. Si se trabaja con codificaciones binarias consistirá simplemente en negar un bit. También es posible realizar la mutación intercambiando los valores de dos alelos del cromosoma. Con otro tipo de codificaciones no binarias existen otras opciones:

- Incrementar o decrementar a un gen una pequeña cantidad generada aleatoriamente.

- Multiplicar un gen por un valor aleatorio próximo a 1.

Aunque no es lo más común, existen implementaciones de Algoritmos Genéticos en las que no todos los individuos tienen los cromosomas de la misma longitud. Esto implica que no todos ellos codifican el mismo conjunto de variables. En este caso existen mutaciones adicionales como puede ser añadir un nuevo gen o eliminar uno ya existente.

Criterio de parada de un AG

Los criterios de parada de un algoritmo evolutivo son múltiples. Obviamente un criterio de parada natural es la obtención del óptimo buscado, siempre que podamos conocer dicha información. Este criterio se utiliza generalmente en problemas de tipo resolución de sistemas o cuando se busca cualquier solución que satisfaga un conjunto de restricciones, en los que se puede comprobar cuándo un individuo es solución. Otros criterios de finalización de la ejecución son de tipo temporal, como la ejecución durante un número determinado de generaciones o hasta que se hayan realizado un número concreto de evaluaciones de la función fitness. Estos números se fijan al inicio de la ejecución del algoritmo evolutivo en base a una experimentación previa que nos determine los parámetros de finalización más adecuados teniendo en cuenta el tamaño del problema a resolver.

Inclusión de otros operadores

Hasta aquí se han descrito los componentes básicos que posee todo algoritmo evolutivo. Sin embargo y dependiendo de la naturaleza del problema existen procedimientos de tratamiento de los individuos para que se produzca una mejora de los mismos al margen de aquellas que puedan producirse fruto de la evolución natural del algoritmo evolutivo. Son métodos que se aplican de manera local a cada individuo explorando el espacio de búsqueda en un entorno del mismo, transformándolo si en dicho entorno se produce alguna mejora con respecto al individuo de partida. Estos métodos de búsqueda local no son más que la inclusión en el algoritmo evolutivo de estrategias que pueden ser utilizadas de manera independiente para la resolución del problema y que si bien de esta manera no se alcanza el óptimo, en combinación con el proceso de evolución que le da el algoritmo evolutivo producen buenos resultados. El mejor momento para la aplicación a un individuo de estos operadores es tras la aplicación del cruce y la posible mutación, es decir justo antes de su inserción en la población.

Aplicación de 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; el cruce, sin embargo, 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.

1.1.3 Conclusión

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 específico 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.

1.1.4 Ejemplo de AG (Problema del Viajante)

Un ejemplo gráfico de aplicación de los AG puede ser el del problema del viajante.

Se trata de, dado un conjunto de ciudades, encontrar un recorrido de tal manera que:

- Cada ciudad se visite una vez
- La distancia recorrida se minimice

La representación de los individuos podría ser algo como lo siguiente:

Ciudades	**ID**
Londres	1
Venecia	2
Madrid	3
Singapur	4
Pekín	5
Nueva York	6
Tokyo	7
El Cairo	8

Individuo	**Representación**
1	(3 5 7 2 1 6 4 8)
2	(2 5 7 6 8 1 3 4)
3	(1 4 2 5 3 6 7 8)
4	(1 2 8 7 3 4 6 5)
5	(5 6 4 7 3 8 2 1)
6	(2 3 1 4 8 6 7 5)
7	(1 2 3 7 8 6 4 5)
…	…

Suponiéndose implementado el algoritmo genético para resolver dicho problema, se muestra a continuación gráficamente como irían evolucionando las soluciones a medida que se realizan generaciones (iteraciones):

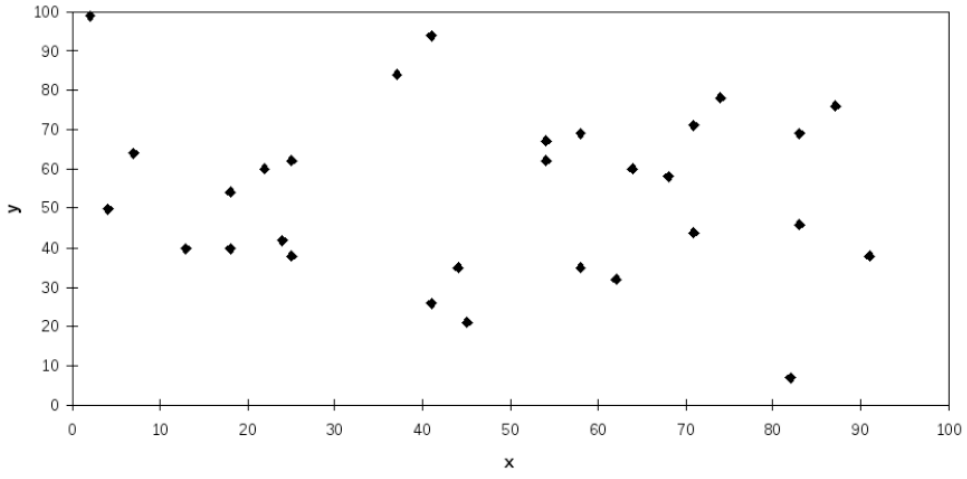


Figura 5. Generación 0

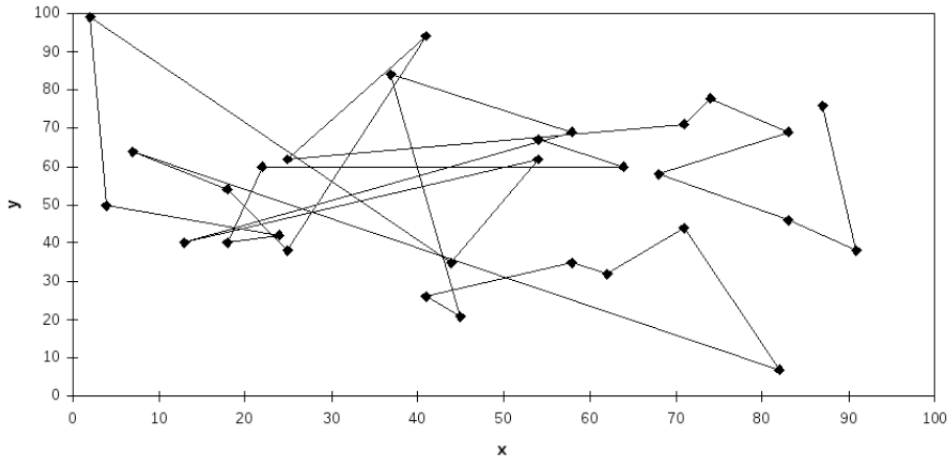


Figura 6. Generación 1

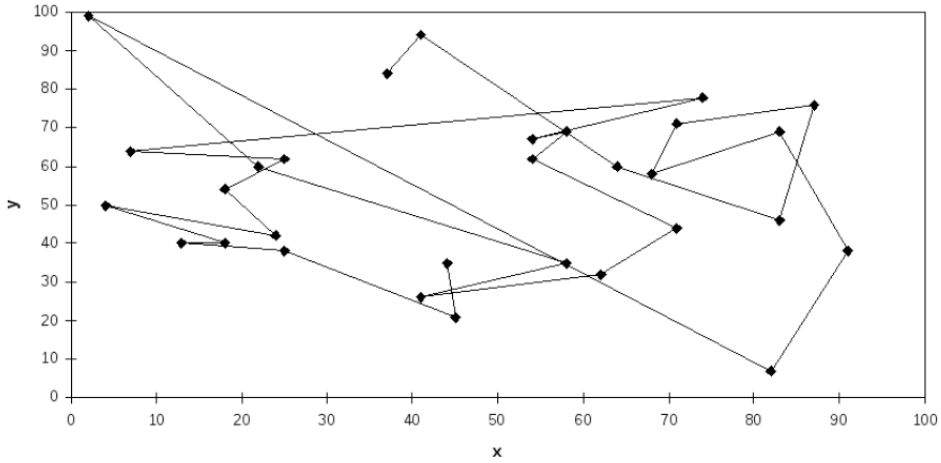


Figura 7. Generación 30

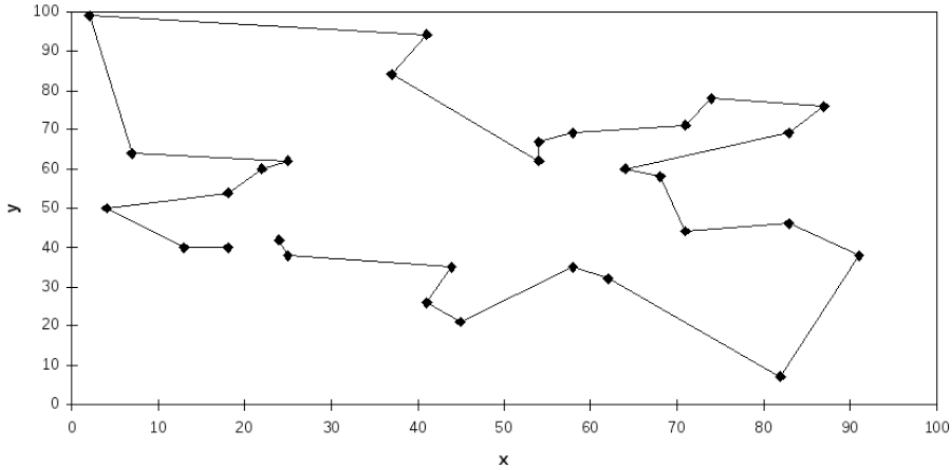


Figura 8. Generación 100

Capítulo 1 (II) : La Regresión Simbólica

Si se tiene una serie de valores históricos, ¿se puede predecir el comportamiento futuro? La respuesta es: Muy probable. ¿Y cómo? Analizando esos valores históricos por si hay algún patrón que los conecta. Una vez encontrado el patrón ya se pueden hacer predicciones realistas. ¿Y cómo se halla ese patrón? Existe un procedimiento estadístico llamado correlación entre variables, pero no funciona en todos los casos. ¿Qué otros procedimientos existen para hallar patrones cuando la correlación falla? Existe uno llamado la regresión simbólica que es una aplicación de la Programación Genética (una rama de la Inteligencia Artificial).

El término "Regresión Simbólica" (RS) representa el proceso durante el cual son medidos datos provistos por una fórmula especificada. Esta técnica es de tipo matemático y es muy utilizada cuando se tienen datos de algún proceso que se desconoce, como por ejemplo resultados de alguna experimentación científica.

Durante mucho tiempo la RS estaba limitada al campo de las matemáticas, donde llegó a ser estudiada y casi dominada, pero en las últimas décadas ha dado el salto también al ámbito informático y computacional. La idea de resolver instancias del problema de la RS a través de algoritmos evolutivos proviene de John Koza, cuando desarrolló los algoritmos genéticos dentro de la programación genética. Por eso, podría decirse que la programación genética es básicamente el intento de resolución de la RS a través de los algoritmos evolutivos, ya que nos proporciona un poder de cómputo infinitamente superior al que los seres humanos pudiéramos desarrollar.

Los problemas que hoy en día se intentan resolver son realmente complejos y pueden presentar ciertas cuestiones que hagan que la búsqueda de la solución a veces sea casi imposible, por ejemplo:

- El cálculo del resultado puede ser computacionalmente carísimo.
- El cálculo puede no ser viable ya que el dominio del problema puede ser dependiente y presentar regiones del espacio en las que sea imposible trabajar.
- El resultado puede verse modificado debido a las limitaciones computacionales.
- Las variables de las que depende el problema pueden no ser triviales ni ser necesariamente independientes

¿Qué se puede hacer entonces? Si el problema es tan complejo, se utiliza una aproximación (metamodelo) con las características requeridas o deseadas, como por ejemplo, menor coste computacional, etc. Esta aproximación, la realiza, entre otras técnicas, la regresión simbólica.

El arma principal de la RS es que se puede ejecutar por medio de algoritmos evolutivos. Hoy en día existen, principalmente, tres métodos de trabajo para la RS: la programación genética (PG), la gramática evolutiva (GE) y la llamada programación analítica (PA), siendo ésta última la herramienta más novedosa.

La capacidad de resolver problemas realmente complejos fue probada en numerosas ocasiones y, hoy en día, la RS vía programación genética tiene el nivel

suficiente como para por ejemplo sintetizar de forma realmente increíble circuitos electrónicos extremadamente sofisticados. Es evidente que la importancia de la regresión simbólica aumentará debido a la creciente complejidad de los problemas que surgen en la ciencia e industria.

1.2.1 ¿Qué es la Regresión Simbólica?

Como se ha dicho, la RS se basa en la existencia de los algoritmos evolutivos, y éstos a su vez están basados en la teoría de la evolución de Darwin, siendo su principal atributo que no se calcula una única solución, sino un conjunto de posibles soluciones simultáneamente, llamada "población" y constituida por "individuos". El principal objetivo de estos algoritmos es encontrar la mejor solución de entre todas las posibles, y difieren entre sí en muchos puntos de vista como la representación de esos "individuos" (decimal, binario…), o la creación de descendencia (cruce, operadores aritméticos, vectores, etc.).

La RS, al estar basada en lo anterior, intentará sintetizar de forma evolutiva un programa definido por el usuario cuando sea posible (fórmulas matemáticas, programas computacionales, expresiones lógicas, etc.). Mientras que el dominio de los algoritmos evolutivos es de naturaleza numérica (reales, complejos, enteros, etc.), el dominio de la RS es de carácter funcional, es decir, consiste en un conjunto de funciones como por ejemplo (seno(), coseno(), MiFuncion(), etc.) , y en otro conjunto denominado terminal (x, t, p, etc.) que contendrá las variables a utilizar por el programa.

De la mezcla de ambos conjuntos se obtiene el programa final, que puede ser bastante complicado desde el punto de vista de su estructura.

Lo descrito anteriormente, puede resultar técnico y complejo, por lo que se intentará dar otra explicación más sencilla del problema de la RS.

La RS, esencialmente, es un método que busca el espacio de expresiones matemáticas para minimizar métricas de error, es decir, dados unos datos obtenidos experimentalmente de los que se desconocen sus relaciones matemáticas, intenta aproximarlos a otros datos mediante expresiones matemáticas minimizando dicho error de aproximación. Por ejemplo, obtenidos los siguientes valores en una experimentación (1, 1.80, 3.75, 4, 5.15, 6), la RS busca mediante el conjunto de expresiones matemáticas posibles, una que minimice el error y de valores más sencillos; una posible expresión para el ejemplo podría ser $f(x) = X + 1$ siendo las soluciones (0, 1, 2, 3, 4, 5) y proporcionando un resultado aproximado pero factible de acorde con el obtenido empíricamente.

A diferencia de las tradicionales regresiones lineales y no lineales que ajustan parámetros a una ecuación de una forma determinada, la RS busca los parámetros y las ecuaciones de forma simultánea. Este proceso proporciona automáticamente ecuaciones matemáticas que son más fáciles de interpretar y que ayudan a explicar el fenómeno observado o la investigación llevada a cabo.

La regresión simbólica trabaja con dos tipos de funciones, las funciones implícitas en las que la relación con la variable dependiente no está dada de forma explícita, y las funciones explícitas en las que sí lo está, aunque es más común trabajar con el primer tipo, ya que, aunque supone un mayor estudio del problema, normalmente no se suele conocer nada acerca de las variables que se manejan. Por ejemplo, una función implícita puede darse en la forma $f(x,y)=0$, mientras que una función explícita vendría dada de la forma $y=f(x)$. Las ecuaciones implícitas pueden ser mucho más expresivas y son normalmente usadas para definir de forma concisa superficies complejas o funciones con múltiples salidas. Consideremos, por ejemplo, la ecuación de un círculo: puede ser representada implícitamente como $x^2+y^2-r^2=0$, y explícitamente usando una multi-salida con la función raíz cuadrada $y=\pm\sqrt{r^2-x^2}$, o como una ecuación paramtérica de la forma $x=\cos(t)$, $y=\sin(t)$, $t=0...2\pi$. El objetivo es obtener ecuaciones implícitas de los datos experimentales obtenidos.

Aplicar la RS a las relaciones implícitas se puede considerar como una forma de aprendizaje no supervisado. Las aplicaciones de RS tradicional eran una forma de aprendizaje supervisado, ya que cada etiqueta 'y' es proporcionada por cada vector 'x' de entrada, y se busca una relación simbólica de la forma $y=f(x)$. Sin embargo, cuando se busca una relación implícita de la forma $f(x,y)=0$, se busca un patrón que identifica los puntos en el conjunto de datos, y excluye a todos los demás puntos del espacio.

Independientemente del tipo de funciones que se utilicen, el objetivo de la RS sigue siendo el mismo: proporcionar aproximaciones válidas a unos datos a través de la búsqueda de una función matemática que sea más 'sencilla' de manejar y de comprender, todo ello minimizando el error cometido.

La RS tiene el mismo objetivo de la regresión lineal pero con un espectro mucho mayor de búsqueda y menos limitaciones: dados los datos, buscará el patrón (expresión algebraica) que identifique el comportamiento de éstos accediendo a todo tipo de funciones y combinaciones algebraicas.

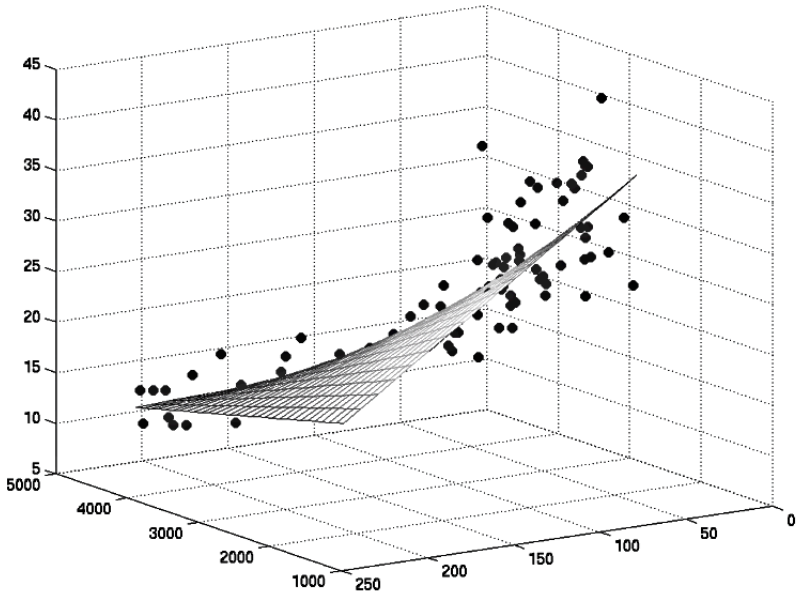


Figura 9. Ejemplo de regresión no lineal para una serie de valores. Como se observa, se intentan ajustar todos los datos en una ecuación que los represente de forma aproximada.

1.2.2 ¿Por qué utilizar la computación evolutiva en la RS?

Como se ha dicho, la RS es una aplicación directa de la programación genética, ya que, como se dijo, se pueden manejar simultáneamente las funciones implícitas y las posibles soluciones del problema.

Además de lo expuesto en el párrafo anterior, existen otros motivos por los cuales aplicar los algoritmos evolutivos a la RS, que son:

- No hay hipótesis sobre modelos prefijados
- Selección natural de las variables más importantes
- No hay supuestos sobre la independencia de las variables
- Las soluciones que proporcionan son más entendibles al ser humano
- …

También existen otras razones que tienen que ver con el coste computacional en las que no se profundizará; tan sólo decir que la aplicación de los algoritmos genéticos en lugar de otra técnica para la resolución de la RS también reduce el consumo computacional de forma considerable.

Por todas estas razones, y más, pero sobre todo por la posibilidad de explorar el espacio de funciones matemáticas y de generar al mismo tiempo soluciones factibles, la programación genética es una buena opción como método para la resolución de instancias del problema de la RS.

En resumen, algunas de las ventajas que tiene aplicar computación evolutiva al problema de la regresión simbólica son:

- Facilita la compresión del problema
- Resume el comportamiento de los múltiples datos
- Identifica las variables clave para la correcta regresión de los datos
- Permite la exploración global de las posibles soluciones para su optimización
- Visualiza los datos como si fuera una expresión analítica (matemática)
- Sugiere futuros experimentos
- Computacionalmente tiene un coste menor que otras técnicas
- Las soluciones proporcionadas suelen aproximarse mucho a los datos iniciales y se ajustan a una expresión mucho más manejable
- Se puede manejar una gran cantidad de datos
- Se garantiza el progreso de la población del algoritmo, es decir, que las posibles soluciones finales serán mejores que las iniciales.

Por último, decir que, aunque existen otras técnicas para la resolución de la RS, como las ya citadas gramáticas evolutivas y programación analítica, entre otras, sobre la que se tiene un cierto dominio así como un mayor ámbito de estudio y trabajo es con la programación genética.

1.2.3 Ejemplo de Aplicación (y técnicas para medir calidad)

A continuación se expondrá un ejemplo que intentará explicar de una forma gráfica el por qué utilizar RS, cuándo utilizarla y los resultados que puede proporcionar en un problema sencillo.

¿Cómo se comportará el mercado en los próximos meses? Este tipo de pregunta quita el sueño a todo gerente porque si supiera qué va a pasar, entonces podría tomar medidas para maximizar la rentabilidad de la empresa o minimizar las pérdidas.

Supóngase que se tienen los siguientes datos históricos:

Mes	Valor
1	3
2	5
3	7
4	9
5	11

A la vista de los datos, se podría obtener sin mucho esfuerzo que la ecuación que se ajusta a ellos es:

$$Valor = 2 \cdot mes + 1$$

y con ella, se podrían predecir acontecimientos futuros, es decir, en el mes 6 el valor del mercado será de 13, en el mes 11, 23, etc.

En el ejemplo anterior, es relativamente sencillo poder deducir la expresión que se comporte de acorde a los resultados experimentales, pero en la vida real normalmente se trabaja con otros datos de mayor dificultad. Véase por ejemplo:

Mes	Valor
1	7
2	10
3	18
4	28
5	44

En esta ocasión, ya no resulta trivial encontrar una relación entre los meses y los valores. Aquí es donde resulta importante la aplicación de la regresión, ya que nos permitirá encontrar ecuaciones que se aproximen a los valores anteriores de forma que el error que se cometa sea mínimo.

De forma gráfica, los resultados anteriores se dispondrían en una gráfica de la siguiente manera:

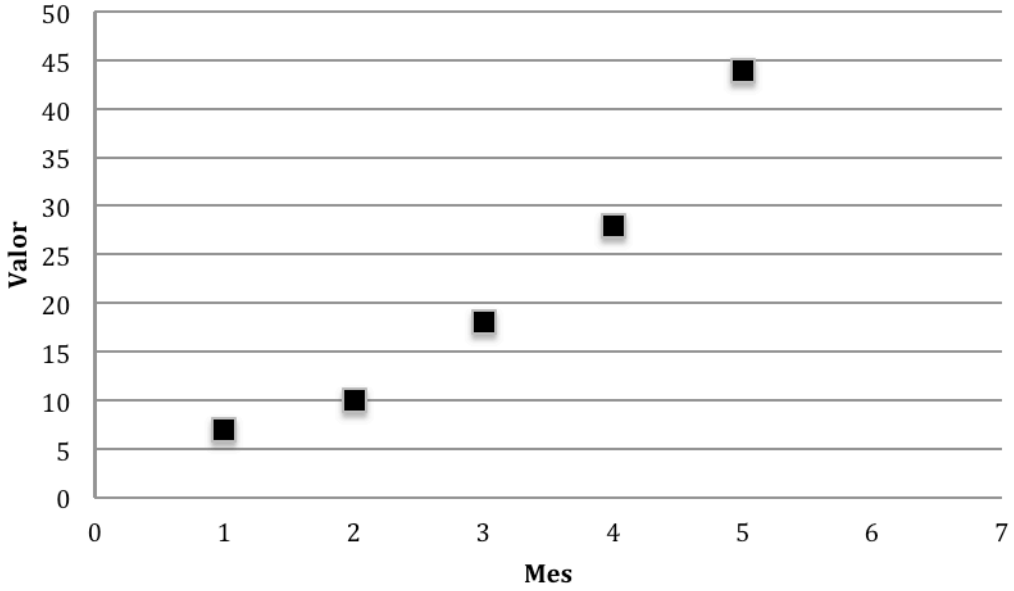


Figura 10. Gráfico Valor-Mes de acuerdo a los datos

Aplicando regresión, en este caso lineal, se obtiene la siguiente ecuación para los puntos muestra:

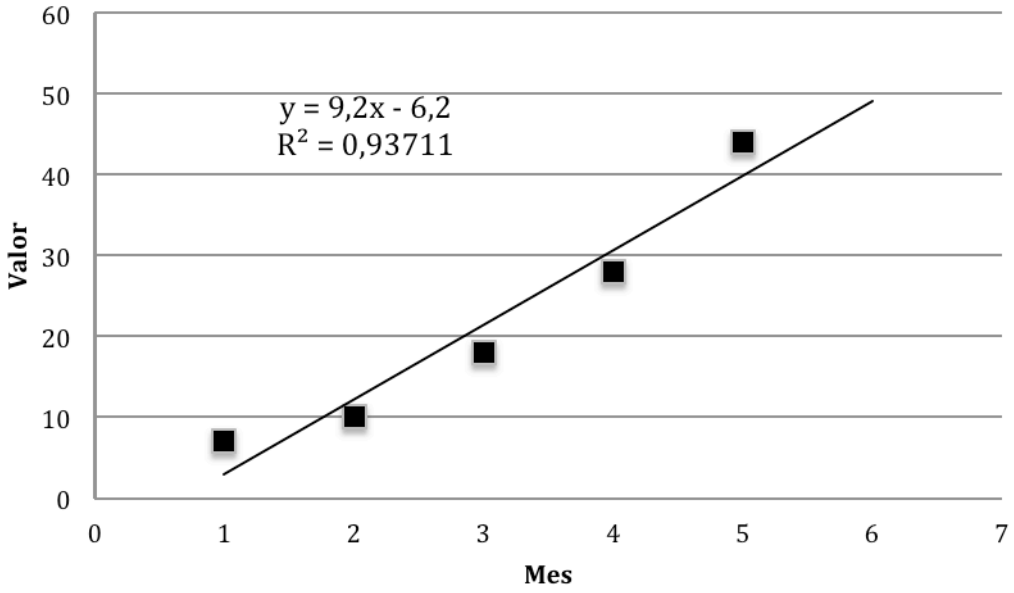


Figura 11. Gráfico Valor-Mes de acuerdo a los datos con regresión lineal

Como se puede observar, para esos datos y con una regresión lineal (que quizá no sea la que mejor resultado proporcione de todas las regresiones posibles pero sí la que más clarifica la idea), se obtiene una ecuación

$$y = 9,2 \cdot X - 6,2 \text{ , con una aproximación de } R^2 = 0,93711$$

El valor R^2 se conoce como **coeficiente de correlación lineal o de Pearson**, el cual muestra la exactitud a la que se ajusta la expresión obtenida:

"El coeficiente de Pearson es una medida cualitativa de la ecuación obtenida en la regresión. Si su valor es 1, es prefecta; si está por encima de 0,65, se considera un buen ajuste. Si se encuentra entre 0,4 y 0,649, se habla de un ajuste regular; por debajo, es malo. La regresión, por tanto, idealmente, busca acercar el coeficiente a 1".

En este punto, se hará un inciso para explicar cómo se obtiene este valor. Es necesario, debido a que es el método más utilizado para conocer si una regresión aplicada a un problema ha sido satisfactoria o no. En este proyecto también se hará uso de ello en la parte experimental, para saber si la aplicación de la regresión simbólica vía algoritmos evolutivos ha resultado válida y proporciona resultados factibles.

El coeficiente de Pearson se calcula de la siguiente manera:

- Reemplazo del punto muestra (x) en la ecuación obtenida en la regresión

X	Y_e	Y_t (*tendencia*)	$Y_e - \bar{Y}_e$	$Y_t - \bar{Y}_t$	$\left(Y_e - \bar{Y}_e * Y_t - \bar{Y}_t\right)$
1	7	3	-14,4	-18,4	264,96
2	10	12,2	-11,4	-9,2	104,88
3	18	21,4	-3,4	0	0
4	28	30,6	6,6	9,2	60,72
5	44	39,8	22,6	18,4	415,84
Total					**846,40**

$$Y_e = 21,4 \text{ (promedio)}$$
$$Y_t = 21,4 \text{ (promedio)}$$

- Se calcula la covarianza

$$Covarianza = \frac{\sum \left(Y_e - \bar{Y}_e\right) * \left(Y_t - \bar{Y}_t\right)}{n-1} = \frac{846,4}{5-1} = 211,6$$

- Desviación típica de Y_e $\rightarrow$ $S_{y_e} = \sqrt{\frac{\sum_{1}^{5}\left(y_e - \overline{y}_e\right)^2}{5-1}} = 14,54647724$

- Desviación típica de Y_t $\rightarrow$ $S_{y_t} = \sqrt{\frac{\sum_{1}^{5}\left(y_t - \overline{y}_t\right)^2}{5-1}} = 15,02664301$

- Por último, cálculo de la correlación:

$$r^2 = \left(\frac{\text{cov}\,arianza}{s_{ye} \cdot s_{yt}}\right)^2 = \left(\frac{211,6}{14,54647724 \cdot 15,02664301}\right)^2 = 0,93711$$

Vemos que el valor obtenido es el que salía en un primer momento.

Continuando con el ejemplo, si se seleccionara otro tipo de regresión, por ejemplo polinomial, que diera lugar a una ecuación del mismo tipo, mostrada en la figura 11, vemos que el coeficiente de Pearson se acerca mucho a 1, resultando un mejor ajuste que para la regresión lineal anterior.

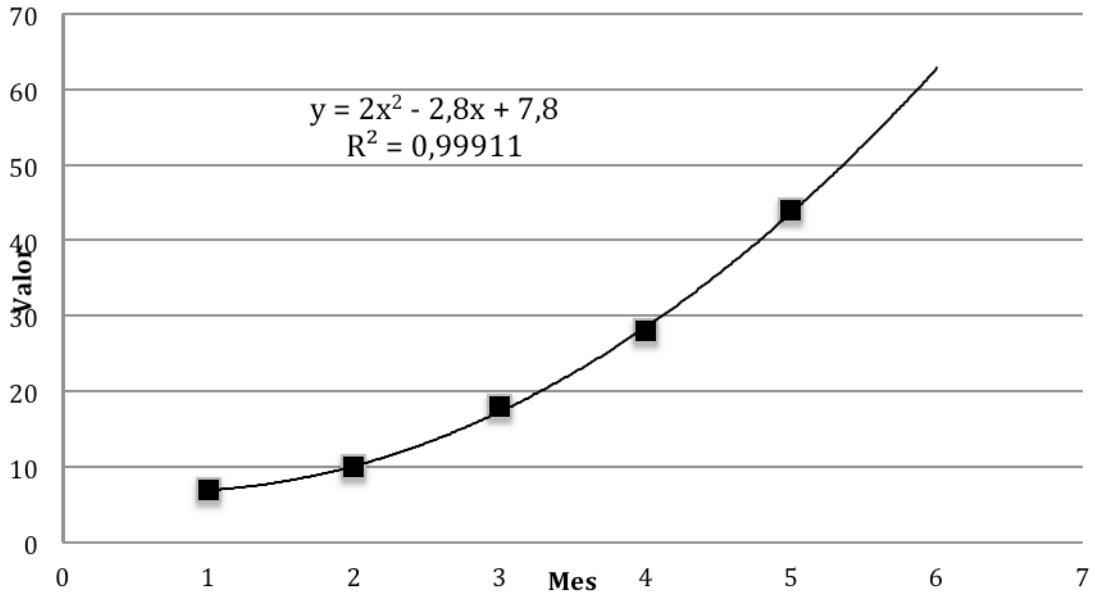


Figura 12. Gráfico Valor-Mes de acuerdo a los datos con regresión polinomial

Se explicará ahora otra segunda técnica para verificar la calidad de las expresiones matemáticas obtenidas tras aplicar una regresión. Se conoce como la suma de las diferencias absolutas, y determina que una ecuación obtenida es mejor que otra en tanto en cuanto la suma de las diferencias de la ecuación para cada valor Y sea más cercana al valor 0.

La siguiente tabla muestra este método para las dos ecuaciones obtenidas en los distintos ejemplos de regresión:

Mes	**Valor (Y)**	**9,2X – 6,2 (E1)**	**Diferencias (E1 e Y)**	**$2x^2$ - 2,8x + 7,8 (E2)**	**Diferencias (E2 e Y)**
1	7	3	4	7	0
2	10	12,2	2,2	10,2	0,2
3	18	21,4	3,4	17,4	0,6
4	28	30,6	2,6	28,6	0,6
5	44	39,8	4,2	43,8	0,2
Suma de Diferencias			**16,4**		**1,6**

Efectivamente, en este caso, la expresión polinomial de grado 2 es mucho más precisa que la ecuación lineal y por tanto es mejor usarla para predecir futuros valores.

La labor del investigador es entonces encontrar la mejor expresión para resolver el problema mediante regresión, es decir, que tenga el mejor coeficiente de Pearson posible o que la suma de las diferencias absolutas se acerque lo más posible a 0.

Sin embargo, hay veces que se tienen puntos muestra de los cuales es muy difícil encontrar una ecuación que sea factible. Véase la siguiente gráfica:

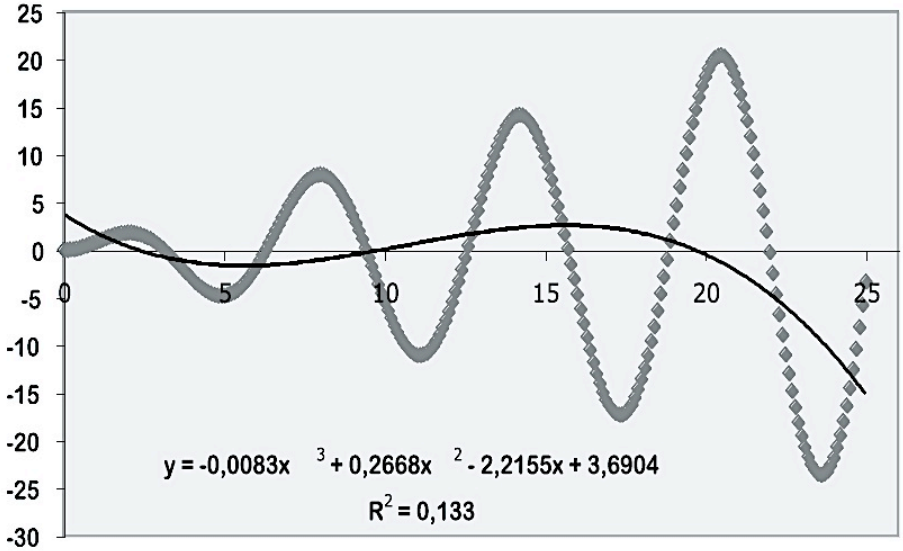


Figura 13. Intento de regresión no lineal para una serie de puntos muestra dados

Aunque se siga aumentando el grado del polinomio, la correlación de Pearson permanece en un umbral demasiado bajo que hace que aplicar este método sea prácticamente inviable.

¿Qué hacer? ¿Cómo buscar una solución? ¿Qué técnica o herramienta utilizar? Todas estas preguntas se contestan con dos palabras: **Regresión Simbólica (RS)** (vía programación genética).

Como ya se ha explicado, la RS se define como la búsqueda de funciones que se ajusten a una serie de puntos muestra dados. La RS tiene acceso a una mayor variedad de funciones (por ejemplo, funciones sinusoidales, valores absolutos, división modular, truncamiento, redondeos, polinomios de grado N, raíces, etc.), compara con la búsqueda determinística que se suele realizar, por ejemplo, en las hojas de cálculo, que sólo se limitan a trabajar con tendencias de tipo logarítmica, potencial, exponencial, lineal o polinómica de muy pocos grados. La RS no sólo abarca multitud de funciones, incluidas éstas, sino que puede trabajar simultáneamente con varias de ellas lo que hace que su capacidad para encontrar una solución válida se incremente de forma muy considerable.

Los pasos que deben darse en la RS vía programación genética para resolver este ejemplo concreto (ya que puede diferir dependiendo del objetivo que se busque y delo problema en sí) serían:

1. *Se genera una población inicial de N ecuaciones (formadas al azar).*

2. *A cada ecuación se le deduce su correlación de Pearson (su valor fitness en este caso concreto y para este ejemplo).*

3. *Ordenar las ecuaciones en orden descendente (según el valor de la correlación de Pearson) y seleccione las M primeras.*

4. *Esas M ecuaciones son reproducidas y generan Q hijas, pero a una hija al azar se le hace un pequeño cambio (mutación) o la hija es el producto combinado de dos ecuaciones padres (cruce).*

5. *Reemplazar las Q ecuaciones peores de la población inicial con las Q ecuaciones hijas.*

6. *Volver al punto 2 repitiéndose el ciclo constantemente hasta que se obtengan ecuaciones con un valor de correlación de Pearson aceptable.*

Véase un ejemplo de aplicación de la regresión simbólica a través de la programación genética. Tendrá las siguientes características:

- Población de tamaño 10
- Las ecuaciones usan funciones tales como seno, coseno o tangente
- La función fitness utilizada es la correlación de Pearson
- Sólo se genera una hija por padre
- Reemplazo por las peores ecuaciones

Los puntos muestra de los que se parte son:

X	**Y**
0	0
0,1	0,00998334
0,2	0,03973387
0,3	0,08865606
0,4	0,15576734
0,5	0,23971277
0,6	0,33878548
0,7	0,45095238
0,8	0,57388487
0,9	0,70499422
1	0,84147098

Se busca, por tanto, la mejor ecuación f(x) que dado el valor X más se ajuste a Y.

1. Se generan N = 10 ecuaciones al azar

Ecuaciones Generadas
Y = 2*seno(x)+coseno(x)
Y = 5*tangente(x/3)-coseno(x/2)
Y = 17-seno(x)+coseno(x)
Y = 4.91-tangente(x)-coseno(x)+seno(x)
Y = x*coseno(x)-seno(x) + 7.19
Y = x*x – seno(x/2) + 3
Y = 2*x – 4*seno(x) + 6*coseno(x)
Y = 1-seno(2*x)+tangente(x)-seno(x)-tangente(x)
Y = 2*seno(x)-coseno(x*x)
Y = 5-seno(x*2)-coseno(x-x)+tangente(4)

2. *Se calcula para cada ecuación su valor fitness, es decir, su coeficiente de Pearson*

Ecuaciones Generadas	**Coeficiente de Pearson**
Y = 2*seno(x)+coseno(x)	0,945542986
Y = 5*tangente(x/3)-coseno(x/2)	0,960871256
Y = 17-seno(x)+coseno(x)	0,964631111
Y = 4.91-tangente(x)-coseno(x)+seno(x)	0,510068073
Y = x*coseno(x)-seno(x) + 7.19	0,961439088
Y = x*x – seno(x/2) + 3	0,924849655
Y = 2*x – 4*seno(x) + 6*coseno(x)	0,985267147
Y = 1-seno(2*x)+tangente(x)-seno(x)-tangente(x)	0,771527756
Y = 2*seno(x)-coseno(x*x)	0,963507321
Y = 5-seno(x*2)-coseno(x-x)+tangente(4)	0,629947087

3. *Se reordenan según su coeficiente y se cogen, por ejemplo, las M = 3 primeras*

Ecuaciones Generadas	**Coeficiente de Pearson**
Y = 2*x – 4*seno(x) + 6*coseno(x)	**0,985267147**
Y = 17-seno(x)+coseno(x)	**0,964631111**
Y = 2*seno(x)-coseno(x*x)	**0,963507321**
Y = x*coseno(x)-seno(x) + 7.19	0,961439088
Y = 5*tangente(x/3)-coseno(x/2)	0,960871256
…	…

4. *Esos 3 padres generan 1 hija cada uno por mutación (no se aplica cruce)*

Ecuaciones Padre	**Ecuaciones Hijas**
Y = **2*x** – 4*seno(x) + 6*coseno(x)	Y = x – 4*seno(x) + 6*coseno(x)
Y = **17**-seno(x)+coseno(x)	Y = seno(x)+coseno(x)
Y = 2***seno(x)-**coseno(x*x)	Y = 2*coseno(x)-coseno(x*x)

5. *Se reemplazan las hijas por las peores ecuaciones de la población*

Ecuaciones Generadas	**Coeficiente de Pearson**
Y = 2*x – 4*seno(x) + 6*coseno(x)	0,985267147
Y = 17-seno(x)+coseno(x)	0,964631111
Y = 2*seno(x)-coseno(x*x)	0,963507321
Y = x*coseno(x)-seno(x) + 7.19	0,961439088
Y = 5*tangente(x/3)-coseno(x/2)	0,960871256
Y = 2*seno(x)+coseno(x)	0,945542986
Y = x*x – seno(x/2) + 3	0,924849655
Y = x – 4*seno(x) + 6*coseno(x)	
Y = seno(x)+coseno(x)	
Y = 2*coseno(x)-coseno(x*x)	

6. *Volver al paso 2 hasta encontrar una ecuación cuya correlación de Pearson sea válida, por ejemplo, 0,9988 o más.*

Ya se ha visto un ejemplo de aplicación permite hacerse una idea de la gran capacidad que posee la regresión junto con la programación genética para encontrar soluciones válidas a problemas complejos, mucho más que éste del ejemplo.

Otro punto a favor de esta técnica es que la regresión simbólica permite ir más allá de la simple relación entre dos variables, ya que puede trabajar con varias. Obsérvese la siguiente tabla:

X	**Y**	**Z**
2	7	15
4	2	18
5	1	9
7	5	11

Z es el resultado de una ecuación en la que participan X e Y. Pero, ¿cuál es la ecuación? ¿Será $Z = 3 \cdot X - 2 \cdot Y$?, ¿ $Z = 5 \cdot X \cdot X + X \cdot Y + 3 \cdot Y$? Este tipo de solución nos lo proporciona la regresión simbólica apoyada en la programación genética.

Sin embargo, no todo son ventajas. Como puntos desfavorables, tenemos que:

- Requiere alto poder de cómputo
- Hay que pensar cómo formar las ecuaciones para que haya variedad y sean algebraicamente correctas.
- El resultado depende en gran medida de la implementación del algoritmo.
- La población puede quedar paralizada en un máximo local (las ecuaciones no tienen forma de mejorar).
- Las ecuaciones pueden tener operaciones que se anulan entre sí (p.e. x-x+x-x)
- No hay garantías de que la ecuación encontrada sea la mejor.

Aun así, las ventajas que tiene aplicar esta herramienta superan a las desventajas, por lo que el uso de la regresión simbólica junto con la computación evolutiva hace que hoy en día se puedan resolver innumerables problemas de diferentes campos con resultados muy aceptables.

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

Capítulo 2 : La Regresión Simbólica. Método de Resolución. Diseño y Codificación

Pablo Solar Rodríguez

ÍNDICE

LISTA DE FIGURAS

Capítulo 2 (I) : Método de Resolución

La Programación Genética (PG) puede ser vista como un método evolutivo directo de programas computacionales con el propósito, como se dijo, del aprendizaje inductivo. Esta definición general hace a la PG independiente de las estructuras de datos que se usen para la representación de los programas. El tamaño, la forma y los contenidos de dichos programas pueden entonces cambiar dinámicamente durante el proceso evolutivo. Por lo general, suelen estar representados por LISP S-Expresiones o por árboles directos con ramas ordenadas.

Antes de continuar, se explicará brevemente en qué consisten las dos representaciones citadas anteriormente. Es conveniente citarlas, ya que son las más comunes en cuanto uso.

2.1.1 S-Expresiones

Las S-Expresiones, en lenguaje LISP, son estructuras de datos basadas en listas que representan datos semi-estructurados. Pueden ser listas jerarquizadas de S-Expresiones menores. Normalmente, suelen estar representadas mediante texto entre paréntesis, con secuencias separadas por espacios en blanco de caracteres tipo string. Véase un ejemplo de una expresión booleana escrita en C convertida a S-Expresión de LISP:

$$\left(4 == 2 + 2\right) \quad \Rightarrow \quad \left(= 4 \left(+ 2\ 2\right)\right)$$

Se suelen definir de forma recursiva, ya sea como objetos de datos aislados llamados “átomos” o como listas de S-Expresiones. Números, vectores, caracteres o símbolos son átomos.

Como ya se dijo, las S-Expresiones pueden estar constituidas de otras S-Expresiones, que a su vez pueden contener más S-Expresiones, por lo que pueden ser anidadas a cualquier profundidad. En LISP, las listas S-Expresiones son construidas a partir de pequeñas estructuras de datos llamadas “cons pairs”, escritas como $\left(x\ .\ y\right)$. El primer elemento (x) es el primer elemento de la lista y el segundo elemento, (y), el resto de ella. Las listas grandes están formadas por “cons pairs” anidados, Ejemplos:

- (1 . (2 . (3 . nil))) $\Rightarrow$ El símbolo nil marca el final de la lista $\equiv$ $\left(1\ 2\ 3\right)$
- factorial (z) $\Leftrightarrow$ $(factorial\ (z))$
- factorial $\{x - 1\}$ $\Leftrightarrow$ $(factorial\ \{x - 1\})$ $\Leftrightarrow$ $(factorial\ (-\ x\ 1))$
- f $\left\{\left\{x + 2\right\} * \left\{y - 3\right\}\right\}$ $\Leftrightarrow$ $\left(f\ \left(*\left(+\ x\ 2\right)\left(-\ y\ 3\right)\right)\right)$

2.1.2 Árboles Ordenados

Un árbol es una estructura de datos ampliamente usada que imita la forma de un árbol (un conjunto de nodos conectados). Un nodo es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él. Se dice que un nodo *a* es padre de un nodo *b* si existe un enlace desde *a* hasta *b* (en ese caso, también decimos que *b* es hijo de *a*). Sólo puede haber un único nodo sin padres, que llamaremos raíz. Un nodo que no tiene hijos se conoce como hoja. Los demás nodos (tienen padre y uno o varios hijos) se les conoce como rama.

Formalmente, podemos definir un árbol de la siguiente forma:

- Caso base: un árbol con sólo un nodo (es a la vez raíz del árbol y hoja).
- Un nuevo árbol a partir de un nodo n_r y k árboles $A_1, A_2, ..., A_n$ de raíces $n_1, n_2, ..., n_n$ con $N_1, N_2, ..., N_k$ elementos cada uno, puede construirse estableciendo una relación padre-hijo entre n_r y cada una de las raíces de los k árboles. El árbol resultante de $N = 1 + N_1 + N_2 + ... + N_k$ nodos tiene como raíz el nodo n_r, los nodos $n_1, n_2, ..., n_k$ son los hijos de n_r y el conjunto de nodos hoja está formado por la unión de los k conjuntos hojas iniciales. A cada uno de los árboles A_i se les denota ahora subárboles de la raíz.

Una sucesión de nodos del árbol, de forma que entre cada dos nodos consecutivos de la sucesión haya una relación de parentesco, decimos que es un recorrido árbol. Existen dos recorridos típicos para listar los nodos de un árbol: primero en profundidad y primero en anchura. En el primer caso, se listan los nodos expandiendo el hijo actual de cada nodo hasta llegar a una hoja, donde se vuelve al nodo anterior probando por el siguiente hijo y así sucesivamente. En el segundo, por su parte, antes de listar los nodos de nivel $n + 1$ (a distancia $n + 1$ aristas de la raíz), se deben haber listado todos los de nivel n. Otros recorridos típicos del árbol son preorden, postorden e inorden:

- El recorrido en preorden, también llamado orden previo consiste en recorrer en primer lugar la raíz y luego cada uno de los hijos $A_1, A_2, ..., A_k$ en orden previo.
- El recorrido en inorden, también llamado orden simétrico (aunque este nombre sólo cobra significado en los árboles binarios) consiste en recorrer en primer lugar A_1, luego la raíz y luego cada uno de los hijos $A_2, ..., A_k$ en orden simétrico.
- El recorrido en postorden, también llamado orden posterior consiste en recorrer en primer lugar cada uno de los hijos $A_1, A_2, ..., A_k$ en orden posterior y por último la raíz.

Para representar expresiones matemáticas mediante árboles, normalmente se consideran los operadores como raíces del árbol, y las hojas como las variables en cuestión. Véase el siguiente ejemplo:

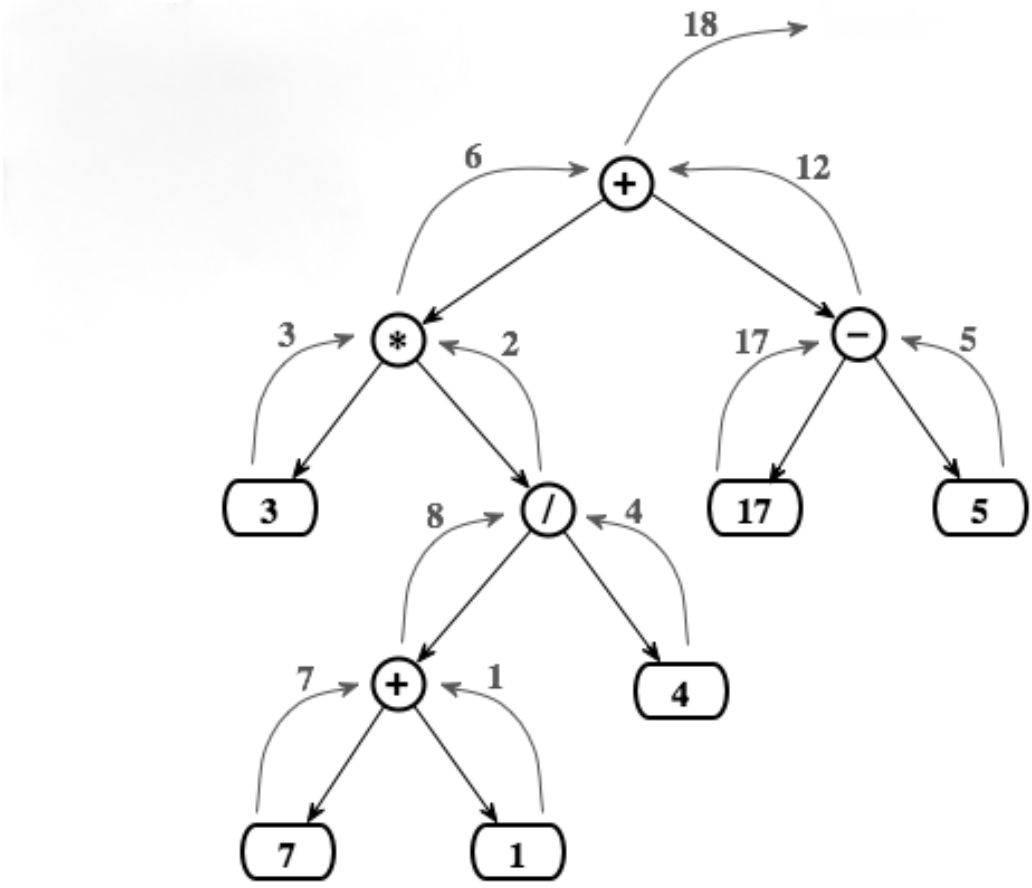


Figura 1. Representación de la expresión *3 * ((7+1) / 4) + (17 – 5)* mediante un árbol

Ya se han descrito de forma muy general las dos representaciones más habituales para las estructuras de datos que se manejan en la PG. Sin embargo, otras variantes en la PG han surgido en los último años. Además de la representación tradicional de árboles o S-Expresiones, varios modelos de representación, como la lineal o la gráfica han sido estudiadas y desarrolladas.

La **Programación Genética Lineal**, que es la que interesa en este proyecto, es una variante de la PG que evoluciona secuencias de instrucciones proporcionadas por un lenguaje de programación imperativo o por lenguaje máquina. El término *lineal*, en este caso, se refiere a las estructuras de datos usadas en la representación de los programas, compuestas por secuencias de asignación de operaciones sobre constantes, o variables sobre otras.

Con esta representación tan simple se puede generar expresiones de tipo no lineal. Numerosos estudios han sido realizados sobre este tipo de representación para la resolución de problemas mediante la PG, arrojando resultados realmente satisfactorios.

En este proyecto se va a presentar, estudiar y realizar el análisis del rendimiento de una nueva estructura de datos para representar programas en la PG lineal: los ***straight line programs (slp's)***.

Los slp's van a ser considerados como una representación lineal de programas, aunque también se pueden aceptar como representaciones graficas (véase Figura 14). Para este tipo de representación lineal se van a desarrollar operadores de recombinación *ad hoc*, que parecen ser más adecuados para las tareas de la regresión simbólica que las generalizaciones dadas por el cruce en un punto, el cruce en *k* puntos y cruce uniforme, más comúnmente usadas en las aproximaciones de la PG lineal.

Una clase particular de slp's, conocidos como *circuitos aritméticos*, constituyen el modelo de computación en el campo de la Teoría de la Complejidad Algebraica. Los *circuitos aritméticos*, con las operaciones aritméticas estándar { +, -, *, / }, son un modelo natural de computación utilizado para estudiar la complejidad computacional de los algoritmos de resolución de problemas de tipo algebraico. Han sido usados en problemas de álgebra lineal, geometría algebraica, Teoría de la Eliminación, etc…

Aquí se presentarán los slp's con la implementación más sencilla posible para evolucionarlos mediante una función fitness que intente reflejar su capacidad para resolver problemas más complejos, además del que se exige en este proyecto.

2.1.3 Straight Line Programs como Estructura de Datos

Como ya se ha dicho en el punto anterior, los straight line programs (slp's) intentan ser una nueva representación de una estructura de datos para resolver problemas de regresión simbólica (RS) mediante la programación genética (PG) lineal y mejorar los resultados que proporcionan las técnicas existentes.

A modo conceptual, los slp's pueden ser vistos como:

- Un conjunto de funciones aritméticas (+, -, *, /, seno, …)
- Un conjunto de variables que se pueden combinar con las funciones anteriores
- Un conjunto de instrucciones que utilizan los conjuntos descritos y que pueden utilizar resultados (instrucciones) previas.

Desde un punto de vista forma, se definen como:

- Sea $F = \{f_1, \ldots, f_n\}$ un conjunto de funciones donde f_i tiene una aridad a_i, para $1 \le i \le n$.
- Sea $T = \{t_1, \ldots, t_m\}$ un conjunto de terminales.

Un slp sobre F y T es una secuencia finita de instrucciones computacionales $\Gamma = \{I_1, \ldots, I_l\}$ donde:

- $I_k \equiv u_k := f_{j_k}(\alpha_1, \ldots, \alpha_{a_{jk}})$; $con \quad f_{j_k} \in F$, $\alpha_i \in T$ para cada i si $k = 1$ y $\alpha_i \in T \cup \{u_1, \ldots, u_{k-1}\}$ para $1 < k \le l$

El conjunto T satisface $T = V \cup C$, donde $V = \{x_1, \ldots, x_p\}$ es un conjunto finito de variables, y $C = \{c_1, \ldots, c_q\}$ es un conjunto finito de constantes. El número de instrucciones l constituye la longitud de Γ.

Nótese que si se considera el slp Γ como el código de un programa, en cada instrucción I_i una nueva variable u_i es introducida. Así es que, el número de variables que no pertenecen al conjunto terminal T, coindice con el número de instrucciones y también con la longitud de Γ. Por tanto, a partir de ahora denotaremos un slp $\Gamma = \{I_1, \ldots, I_l\}$ como $\Gamma = \{u_1, \ldots, u_l\}$. Cada una de las variables no terminales u_i puede ser considerada como una expresión sobre el conjunto de terminales T construida mediante una secuencia de composiciones recursivas del conjunto de funciones F. Se denotará por $SLP(F,T)$ al conjunto de todos los slp's sobre F y T.

Ejemplo : Sea F un conjunto de tres operaciones aritméticas binarias, $F = \{+, -, *\}$ y $T = \{1, x_1, x_2\}$ el conjunto de terminales. En esta situación, cualquier slp $\Gamma \in SLP(F,T)$ es una secuencia de polinomios en dos variables con coeficientes enteros. Si consideramos el siguiente straght line program con una longitud total de 5 instrucciones, tenemos:

$$\Gamma \equiv \begin{cases} u_1 := x_1 + 1 \\ u_2 := u_1 * u_1 \\ u_3 := x_2 + x_2 \\ u_4 := u_2 * u_3 \\ u_5 := u_4 - u_3 \end{cases}$$

El resultado del slp es el polinomio:

$$u_1 := x_1 + 1$$
$$u_2 := (x_1 + 1)^2$$
$$u_3 := 2x_2$$
$$u_4 := 2x_2 \cdot (x_1 + 1)^2$$
$$u_5 := 2x_2 \cdot (x_1 + 1)^2 - 2x_2$$

Observación: Cada slp $\Gamma = \{u_1, \ldots, u_l\}$ sobre F y T puede ser representado mediante un grafo dirigido $G_\Gamma = (V, E)$. El conjunto de vértices es $V = T' \cup \{u_1, \ldots, u_l\}$, donde T' contiene todos los terminales involucrados en la computación. El conjunto de arcos es construido de la siguiente manera: para cada k, $1 \le k \le l$, se dibuja un arco (u_k, α_i) para cada $i \in \{1, \ldots, a_{j_k}\}$. Nótese que T' es el conjunto de hijos de G_Γ y es a la vez un subconjunto de el conjunto T de terminales. La Figura 14 es un grafo dirigido que representa el slp descrito anteriormente.

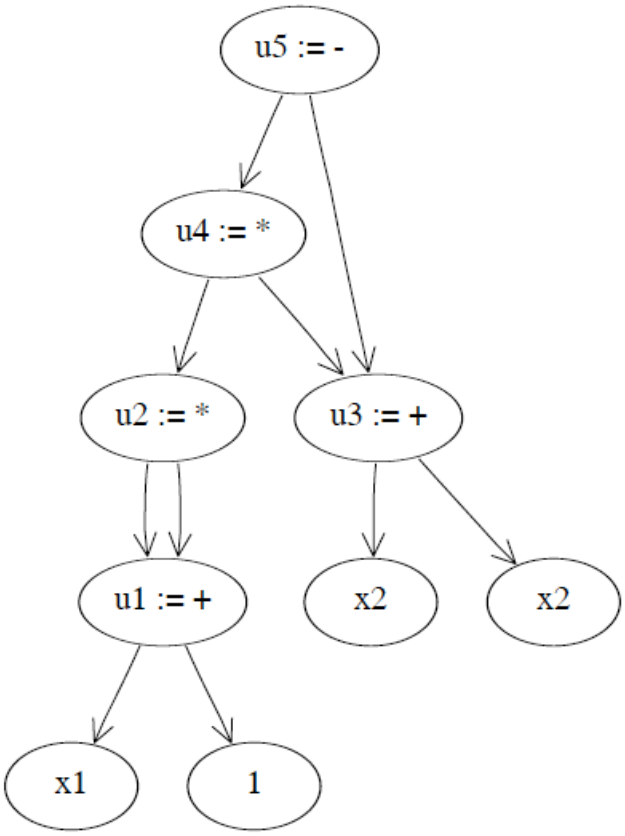


Figura 2. Grafo dirigido representando el slp del ejemplo.

Esta definición de slp no estaría completa si no se complementa con una función semántica . Para establecerla, se considera el espacio de salida como sigue:

- Sea $\Gamma = \{u_1, \ldots, u_l\}$ un slp sobre *F* y *T*.
- Un conjunto de salidas de Γ, $O(\Gamma) = \{u_{i_1}, \ldots, u_{i_t}\}$ es cualquier conjunto de variables no terminales de Γ.
- Siempre que $V = \{x_1, \ldots, x_p\} \subset T$ sea el conjunto de variables terminales, la función semántica de Γ, denotada como $\Phi_\Gamma : I^p \rightarrow O^t$, satisface que:
  - $\Phi_\Gamma(a_1, \ldots, a_p) = (b_1, \ldots, b_t)$, donde b_j representa el valor de la expresión sobre *V* de la variable no terminal u_{i_j} cuando se sustituye la variable x_k por el valor $a_k : 1 \le k \le p$.
- En nuestro caso, el conjunto de salida $O(\Gamma)$ estará constituido por una sola variable, y por tanto los slp's computarán funciones multivariadas con valores en |R.

Dados dos slp's Γ_1 y Γ_2 sobre *F* y *T*, se dirá que son equivalentes si tienen la misma función semántica, es decir, $\Phi_{\Gamma_1} \equiv \Phi_{\Gamma_2}$.

- Sea $\Gamma = \{u_1, \ldots, u_l\}$ un slp sobre *F* y *T* con un conjunto de salida $O(\Gamma) = \{u_{i_0}\}$, con $1 \le i_0 \le l$. Se puede entonces comprobar fácilmente que el slp $\Gamma' = \{u_1, \ldots, u_{i_0}\}$ es equivalente a Γ.
- Nótese que para la computación de la función semántica Φ_Γ, a lo sumo $u_1, \ldots, u_{i_0}$ son necesarios, Por lo tanto $\Phi_\Gamma \equiv \Phi_{\Gamma'}$. A partir de ahora se asume sin pérdida de generalidad que el conjunto de salida de Γ es $O(\Gamma) = \{u_l\}$.

Sea $f_0(x) = x$ *y* $f_{i+1}(x) = f_i(x) \cdot f_i(x)$. *Para evaluar, por ejemplo,* $f_3(5) = 5^8$, *se haría:*

$$f_3(5) = f_2(5) \cdot f_2(5) = \big(f_1(5) \cdot f_1(5)\big) \cdot f_2(5) = \Big(\big(f_0(5) \cdot f_0(5)\big)\Big) \cdot f_1(5) \cdot f_2(5)$$

Y ahora se evaluarían los terminales almacenando todos los resultados intermedios:

$f_0(5) = 5$
$f_1(5) = f_0(5) \cdot f_0(5) = 5 \cdot 5 = 25$
$f_2(5) = f_1(5) \cdot f_1(5) = 25 \cdot 25 = 625$
$f_3(5) = f_2(5) \cdot f_2(5) = 625 \cdot 625 = 390625$

En lugar de 8 multiplicaciones , sólo se hacen 3

Figura 3. Ejemplo de Straight Line Program

2.1.3.1 Código efectivo y no efectivo en los SLP's

Considérese el siguiente slp sobre $F = \{+, -, *\}$ y $T = \{1, x, y\}$:

$$\Gamma \equiv \begin{cases} u_1 := x * 1 \\ u_2 := u_1 + y \\ u_3 := u_2 * u_2 \\ u_4 := u_1 * y \end{cases}$$

Sea ahora $O(\Gamma) = \{u_4\}$ la salida de Γ. Nótese que si se quiere computar el valor de la función semántica de Γ para cualquier entrada $(a_1, a_2) \in \mathbb{R}^2$, no es necesario computar los valores intermedios de u_2 y u_3.

En este caso, las asignaciones de u_2 y u_3 en Γ pueden ser consideradas como código no efectivo y pueden ser eliminadas. Después de realizar esta acción, y renombrar las asignaciones restantes en Γ, un nuevo slp Γ' es obtenido:

$$\Gamma \equiv \begin{cases} u_1 := x * 1 \\ u_4 := u_1 * y \end{cases}$$

El slp Γ' es equivalente a Γ porque tienen la misma función semántica $\Phi(x, y) = x * y$, si consideramos para Γ' la salida $O(\Gamma) = \{u_2\}$.

En general, para calcular el código efectivo de un slp $\Gamma = \{u_1, \ldots, u_l\}$, se necesita en primer lugar establecer una relación en el conjunto de variables no terminales. En este sentido, se podrá decir que $u_i \Re u_k$ si u_i aparece en la expresión funcional asignada a u_k. De una manera formal:

- Sea $\Gamma = \{u_1, \ldots, u_l\}$ un slp sobre F y T. Se define la siguiente relación en el conjunto $\{u_1, \ldots, u_l\}$ (asumir que $u_i := f_{j_i}(\alpha_1, \ldots, \alpha_{a_{j_i}})$ y $u_k := f_{j_k}(\beta_1, \ldots, \beta_{a_{j_k}})$ con $i < k$):
    - $u_i \Re u_k \Leftrightarrow u_i = \beta_s$, para alguna $s, 1 \le s \le a_{j_k}$
    - Si se considera $\Re$ como el cierre transitivo y reflexivo de $\Re$, entonces todo ello constituye una relación de orden sobre $\{u_1, \ldots, u_l\}$.

Siempre que $O(\Gamma) = \{u_l\}$, el código efectivo de Γ es el conjunto de variables no terminales involucradas en el proceso de evaluación de u_l para un valor de entrada de las variables terminales. Se podría denotar este conjunto por $S = \{u_i \in \Gamma \ / \ u_i \overline{\Re} u_l\} = \{u_{i_1}, \ldots, u_{i_m}\}$, asumiendo que $i_1 < \ldots < i_m$. Para obtener el conjunto anterior, se construye una cadena no decreciente de conjuntos $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_t = S$, donde el conjunto inicial $S_0 = \{u_l\}$ y de forma general $S_k = S_{k-1} \cup \{u_i \in \Gamma \ / \ \exists u_j \in S_{k-1} \ ; \ u_i \Re u_j\}$, siendo $\Re$ la relación definida anteriormente. Es fácil de ver que la cadena anterior de conjuntos estaciona después de un número finito de pasos, así como que en el peor caso S_t será precisamente Γ. También se comprueba de manera clara que $u_{i_m} = u_l$ y que se puede construir un nuevo slp $\Gamma' = \{u_1', \ldots, u_m'\}$ considerando la asignación de instrucciones en S y una función de renombramiento ξ sobre S de tal manera que $\xi(u_{i_k}) = u'_k$. Nótese que Γ' es equivalente a Γ como consecuencia directa de la construcción y además verifica que:

$$u'_i \Re u'_m \quad \forall i \in \{1, \ldots, m\}$$

Nota : Si $\Gamma \equiv \Gamma'$ se dirá que Γ es un slp efectivo.

2.1.3.2 Cálculo (computacional) de la Función Semántica

Para el cálculo de la función semántica de un slp, hay que considerar el conjunto de funciones F y el conjunto de terminales T con un conjunto de variables $V = \{x_1, \ldots, x_n\}$. La estrategia para la computación de la función semántica de un slp $\Gamma = \{u_1, \ldots, u_l\}$ sobre F y T que consiste en la evaluación de las variables no terminales siguiendo el orden establecido en Γ, no es un buen método cuando Γ no es efectivo. En la práctica es mejor obtener el slp efectivo equivalente a Γ y evaluarlo. El siguiente algoritmo describe su obtención:

Input : *Un slp* $\Gamma = \{u_1, \ldots, u_l\}$ *sobre F y T, con un conjunto de salida* $O(\Gamma) = \{u_l\}$ *y un vector de valores* $(a_1, \ldots, a_n)$ *donde* a_i *es el valor de la variable* x_i.

Output : $\Phi_\Gamma(a_1, \ldots, a_n)$

1. *Cálculo antes descrito del conjunto* $S = \{u_{i_1}, \ldots, u_{i_m}\}$ *por medio de los conjuntos parciales Sk.*
2. *Desde j = 1 hasta m, evaluar* u_{i_j} *reemplazando cada ocurrencia de* x_i *por* a_i *y cada ocurrencia de* u_{i_k}, *con k < j, por su valor, que fue previamente calculado.*
3. *Retornar el valor de* u_{i_m}

2.1.4 Programación Genética (PG) con SLP's para la resolución del problema de la Regresión Simbólica (RS)

Como se ha dicho en varias ocasiones, el problema de la RS consiste en encontrar de forma simbólica una función que se ajuste a un conjunto finito de puntos muestra.

De manera más formal, se considera un espacio de entrada $X = \mathbb{R}^n$ y un espacio de salida $Y = \mathbb{R}$, además de un conjunto de m pares $z = (x_i, y_i)\ 1 \le i \le m$, que tienen una medida probabilística desconocida p en el producto escalar $Z = X \times Y$ y que tienen una distribución idéntica e independiente. El objetivo es construir una función $f : X \rightarrow Y$ que prediga el valor $y \in Y$ para un valor $x \in X$, utilizando un criterio que busca minimizar el error empírico, siendo éste:

$$\varepsilon_z(f) = \frac{1}{m}\sum_{i=1}^{m}\left(f(x_i) - y_i\right)^2$$

que es conocido como el error cuadrático medio (ECM).

Aunque la RS ha sido tratada mediante la PG de diferentes formas, normalmente una población con estructuras de árbol que codifican expresiones es evolucionada siguiendo el principio Darwiniano de supervivencia y reproducción de los mejores. Sin embargo, en este proyecto se adoptarán los straight line programs como las estructuras que evolucionarán dentro del proceso.

Una de las ventajas es que la estructura slp permite que el resultado de una subexpresión sea reutilizada múltiples veces durante la computación, lo que permite expresar cálculos más complejos con menos cantidad de instrucciones y obteniendo resultados individuales, en general, más compactos en términos de tamaño.

Además, las variaciones de tamaño que pueden surgir en los slp's son más fáciles de controlar que en las estructuras en árbol; de hecho, usando código no efectivo, se puede forzar a que todos los slp's del espacio de búsqueda tengan el mismo tamaño.

Sin embargo, el rendimiento que pueden proporcionar todas estas características anteriores dependen mucho de el diseño elegido para los operadores de recombinación (cruce, mutación, …).

Uno de los aspectos clave en la evolución del proceso va a ser el valor que pueda tomar la variable que se generará de manera aleatoria, ya que va a determinar si se aplicarán los operadores de recombinación a la población de individuos. Normalmente la probabilidad de que se efectúe el cruce es alta, siendo todo lo contrario la mutación. Por tanto, es importante que a la hora realizar el diseño, se tenga en cuenta la probabilidad con la que pueden evolucionar las estructuras slp.

Desde un punto de vista de un lenguaje de muy alto nivel, el algoritmo de la programación genética que se ha de implementar para este problema y estas estructuras sería:

Generar una población inicial aleatoria

Evaluar a los individuos (función fitness)

Mientras (no condición de terminación) hacer

para i = 1 hasta tamaño_población hacer

Op := valor aleatorio [0, 1]

si (Op < probabilidad_cruce)

aplicar cruce

else si (Op < probabilidad_cruce + probabilidad_mutacion)

aplicar mutación

else si (Op < probabilidad_cruce + probabilidad_mutacion + probabilidad_reproducción)

aplicar reproducción

Evaluar a los nuevos individuos (función fitness)

Insertar nuevos individuos en Nueva_Poblacion

Actualizar Población con Nueva_Población

En el pseudo-código anterior se ha introducido un nuevo concepto, la reproducción, que se tiene en cuenta en el modelo teórico, aunque en la aplicación práctica de este proyecto no se va a llevar a cabo, ya que se reserva por si se quisiera hacer más adelante una mayor profundización en las técnicas de la programación genética y los procesos evolutivos.

2.1.4.1 La Población Inicial

Sea, tal y como se dijo en la sección 2.2.3:

- $F = \{f_1, \ldots, f_n\}$ un conjunto de funciones donde f_i tiene una aridad a_i, para $1 \le i \le n$.
- $T = \{t_1, \ldots, t_m\}$ un conjunto de terminales.

La generación, entonces, de cada slp en la población inicial se realiza de la siguiente manera:

- Para la primera instrucción u_1, se selecciona $f_{j_1} \in F$ al azar
- Cuando esta función sea seleccionada, para cada argumento $i \in \{1, \ldots, a_{j_1}\}$ de f_{j_1}, un elemento α_i de T es escogido de manera aleatoria

En general, la construcción de la instrucción u_k, $k > 1$ también empieza con la selección aleatoria de $f_{j_k} \in F$. Tras ello, para $i = 1, \ldots, a_{j_k}$, se toma al azar $\alpha_i \in T \cup \{u_1, \ldots, u_{k-1}\}$.

En la práctica, es necesario fijar un límite superior L para el tamaño de los slp's involucrados en el proceso evolutivo. Así pues, introducido este parámetro, el primer paso en algoritmo podría ser la asignación aleatoria de su valor para cada slp, $l \in \{1, \ldots, L\}$.

Nótese que los slp's generados con la estrategia arriba definida podrían ser no efectivos. Sin embargo, se permite la no efectividad durante el proceso evolutivo. Por otro lado, se mantendrán poblaciones homogéneas de igual tamaño en los individuos. Con este propósito, dado un slp $\Gamma = \{u_1, \ldots, u_L\}$, $L \ge l$, se puede construir otro, tal que:

- $\Gamma' = \{u_1, \ldots, u_{l-1}, u'_l, \ldots, u'_{L-1}, u'_L\}$, donde:
    - $u'_L = u_l$ y u'_k, para $k = l$ hasta $L-1$ es cualquier instrucción que satisface las condiciones de definición de los slp's

Considerando que $O(\Gamma') = O(\Gamma)$, es fácil de ver que Γ' es equivalente al slp inicial Γ.

2.1.4.2 Función Fitness

En la PG, se mide el fitness de alguna manera para luego simular a la naturaleza y controlar las operaciones que modifican las estructuras en la población artificial.

La manera más común, asigna a cada individuo un valor fitness a través de algún procedimiento evaluativo definido explícitamente. Así que, una función fitness se define sobre el espacio de búsqueda.

En este proyecto, el procedimiento para computar el valor fitness de un slp siempre involucrará el cálculo de los valores de las correspondientes funciones semánticas sobre el conjunto de valores dado para las variables terminales. De esta manera, dado $z = (x_i, y_i) \in /R^n \times /R$, $1 \le i \le m$ para cualquier slp Γ sobre F y T, se define el valor fitness de Γ como:

$$\Im_z(\Gamma) = \varepsilon_z(\Gamma) = \frac{1}{m}\sum_{i=1}^{m}\left(\Phi_\Gamma(x_i) - y_i\right)^2$$

Es decir, el valor fitness es el error empírico de la función semántica de Γ para el conjunto de puntos z. Se utilizará el algoritmo del apartado 2.1.3.2.

2.1.4.3 Operadores de Recombinación

Dado que la representación por medio de los slp's consiste en una secuencia finita de instrucciones y que todas tienen la misma longitud, los métodos de cruce, como el cruce uniforme, el cruce en un punto y el cruce en varios puntos, puede ser adaptados a esta situación de una manera natural y sencilla.

Por eso, se ha diseñado una nueva operación de cruce "ad-hoc" que produce otro tipo de intercambio de información entre individuos padres; el objetivo es llevar subexpresiones de un padre al otro.

Una subexpresión está representada por una instrucción u_i y todas las instrucciones que son usadas para evaluar u_i . Esta es justo la parte de código efectiva del slp que se necesita para calcular la expresión, sobre las variables terminales, asociada a u_i.

De una manera formal, aunque teórica, se describen a continuación estos operadores de recombinación adaptados, esto es, el cruce, la mutación y la reproducción (de forma muy general este último), ya que a la hora del diseño práctico (implementación en la programación) variarán en algunos aspectos, pues es difícil ajustarse fielmente a la explicación que se detallará a partir de ahora.

CRUCE

Sea $\Gamma = \{u_1, \ldots, u_L\}$ y $\Gamma' = \{u'_1, \ldots, u'_L\}$ dos slp's sobre *F* y *T*.

Primero, una posición *k* en Γ es seleccionada al azar, $1 \le k \le L$. Se considera de nuevo la relación definida $\overline{\Re}$ para la descripción del conjunto:

$$S_{u_k} = \{u_j \in \Gamma \ / \ u_j \, \overline{\Re} \, u_k\} = \{u_{j_1}, \ldots, u_{j_m}\}, \quad j_1 < \ldots < j_m$$

Como se ha dicho, el conjunto S_{u_k} es la parte de código efectiva de Γ relacionada con el cálculo de u_k.

En segundo lugar, se selecciona una posición *t* en Γ' con $m \le t \le L$, y se modifica Γ', sustituyendo el subconjunto de instrucciones $\{u'_{t-m+1}, \ldots, u'_t\}$ en Γ' por las instrucciones de Γ en S_{u_k} y llamando a una función de renombramiento $\Omega(u_{j_i}) = u'_{t-m+i}$, para cada $i \in \{1, \ldots, m\}$.

Con este proceso, se obtiene el primer hijo de Γ y Γ'. Para conseguir el segundo, se **repite** esta estrategia de forma simétrica, pero empezando por seleccionar aleatoriamente una posición *k'* en Γ'.

Como ejemplo, considérense los siguientes slp's:

$$\Gamma \equiv \begin{cases} u_1 := x + y \\ u_2 := u_1 * u_1 \\ u_3 := u_1 * x \\ u_4 := u_3 + u_2 \\ u_5 := u_3 * u_2 \end{cases} \qquad \Gamma' \equiv \begin{cases} u_1 := x * x \\ u_2 := u_1 + y \\ u_3 := u_1 + x \\ u_4 := u_2 * x \\ u_5 := u_1 + u_4 \end{cases}$$

Si $k = 3$, entonces $S_{u_3} = \{u_1, u_3\}$, y t debe ser seleccionado en el rango {2, …, 5}. Si se asume que t = 3, el primer hijo entonces sería:

$$\Gamma_1 \equiv \begin{cases} u_1 := x * x \\ u_2 := x + y \\ u_3 := u_2 * x \\ u_4 := u_2 * x \\ u_5 := u_1 + u_4 \end{cases}$$

MUTACIÓN

La mutación es asexual y actúa en sólo un padre. Esta operación introduce cambios aleatorios en el individuo. Puede ser beneficiosa, ya que permite más diversidad en una población que puede converger prematuramente en un óptimo local.

El primer paso cuando la mutación se aplica a un slp Γ consiste en la selección de una instrucción $u_i \in \Gamma$ al azar. Tras ello, se hace una nueva selección aleatoria dentro de los argumentos de la función $f \in F$ que constituye la instrucción u_i. El paso siguiente, y final, es la sustitución del argumento seleccionado por otro en $T \in \{u_1, \ldots, u_{i-1}\}$ escogido también azarosamente.

No obstante, la definición formal de la mutación se describe como:

- Sea $\Gamma = \{u_1, \ldots, u_L\}$ un slp sobre *F* y *T*.
- Sea $u_i = f(\alpha_1, \ldots, \alpha_n)$ el punto de mutación seleccionado, donde $f \in F$, $\alpha_k \in T \cup \{u_1, \ldots, u_{i-1}\}$.

La mutación de Γ en el punto i queda como:

- $\Gamma' = \{u_1, \ldots, u_{i-1}, u'_i, u_{i+1}, \ldots, u_L\}$, donde:
  - $u'_i = f(\alpha_1, \ldots, \alpha_{j-1}, \alpha'_j, \alpha'_{j+1}, \ldots, \alpha_n)$, $\alpha_j \neq \alpha'_j \in T \cup \{u_1, \ldots, u_{i-1}\}$ con $j \in \{1, \ldots, n\}$. j y α'_j son seleccionados aleatoriamente.

REPRODUCCIÓN

La reproducción consiste en la copia de un individuo de la población actual en la nueva población. Se suele utilizar reemplazamiento generacional, pero en el proceso de construcción de la nueva población, los hijos generados no necesariamente reemplazan a los padres.

Después del cruce, se tienen cuatro individuos: dos padres y dos hijos, que se ordenan por su valor fitness. Tras ello, se escoge un individuos de cada uno de los dos niveles del ranking. Si, por ejemplo, tres de los individuos tienen el mismo valor fitness, sólo se selecciona uno de ellos y el otro sería el que tuviera peor fitness de los cuatro.

Esta estrategia previene la convergencia prematura y mantiene la diversidad en la población. También con el proceso anterior se obtienen mejores resultados si los individuos involucrados en los operadores de recombinación son seleccionados aleatoriamente, en lugar de los métodos de selección basados en los valores de fitness.

2.1.5 Adaptación de la teoría de SLP's para la resolución práctica del problema de la Regresión Simbólica

En los anteriores apartados se ha introducido, explicado y detallado de una manera formal el concepto de *straight line program*, así como sus características y su aplicación en el problema de la RS. En dicha descripción, se exponen ciertos aspectos que desde un punto de vista teórico son total y absolutamente correctos, pero que su aplicación práctica puede resultar compleja, confusa y con poca utilidad.

Por ello, será necesario readaptar de alguna manera todas estas ideas para que, lo que en un principio pueda resultar complicado, resulte más claro y manejable y pueda ser sometido a un estudio práctico de manera más sencilla.

La idea principal de slp no va a cambiar: seguirá constando de un conjunto de instrucciones que se apoyan en otras para conseguir algún valor, resultado de operaciones y expresiones aritméticas entre instrucciones. Los cambios se efectuarán sobre todo en la representación de cada instrucción y en el cálculo de la salida final. La diferencia entre el modelo teórico y esta adaptación práctica supone algún cambio, aunque si se analiza de una manera detenida, el planteamiento principal permanece y no difieren demasiado entre ellos.

2.1.5.1 Parámetros Iniciales

Tal y como está descrito en los apartados teóricos anteriores, la aplicación constará de dos conjuntos principales; V, que representa el número de variables que se ven involucradas en el proceso evolutivo, y F, que será el conjunto de funciones que utilizarán.

De una manera formal, los dos conjuntos anteriores se describen como:

- Sea $V = \{X_1, \ldots, X_n\} = \{u_{-n+1}, \ldots, u_0\}$ el conjunto de variables del proceso.
- Sea $F = \{+, -, *, /\}$ el conjunto de funciones que se utilizarán en el proceso para el cálculo de resultados.

Nótese que para el conjunto *F* ya se definen y se fijan las funciones que se van a utilizar en este proyecto, que no son ni más ni menos que +, -, *, /. Dado que este proyecto es un entrenamiento de los slp's, si se quisiera realizar un estudio más en profundidad en base a este, se podrían utilizar un conjunto de operaciones más elaboradas, aunque la complejidad de la aplicación aumentaría también de forma considerable.

Además de estos dos conjuntos arriba descritos, también se presenta un parámetro que va a tener una importancia capital durante el desarrollo de todo el proceso, ya que aparte de desempeñar su función, realiza otras funciones indirectas que tienen una gran repercusión en el desarrollo del proceso evolutivo. Este parámetro se define como $\angle$, y representa el **número máximo de operaciones no escalares o diferentes de {+, -}**. Dado que la suma y la resta son operaciones aritméticas básicas, una instrucción va a tener realmente importancia cuando en ella se ejecuten operaciones aritméticas no básicas, en este caso {*, /}. Es decir, $\angle$ está definiendo las instrucciones que de alguna manera incrementan la complejidad de la función representada mediante el slp.

La idea, en este caso no es realmente construir slp's de determinada longitud, sino una estructura capaz de representar todos los slp's que poseen a lo sumo L instrucciones u operaciones no escalares. Sería algo así como una especie de conjunto de instrucciones genéricas, dependientes de parámetros, de manera que la especialización de dichos parámetros trae como resultado un slp concreto con L operaciones no escalares y de cualquier longitud. El algoritmo genético trabajará con individuos que codificarán los parámetros de dicha estructura estática y mediante el proceso evolutivo tratará de obtener los mejores valores para los mismos. La motivación para considerar sólo las instrucciones no escalares procede del concepto de complejidad de una función. Si tratamos con polinomios, una medida de la complejidad es sin duda el grado. El incremento de grado nunca se producirá mediante sumas o restas en monomios. En la estructura genérica SLP que se define en el siguiente apartado se puede observar que las instrucciones escalares (suma o resta) que pudiesen existir en el slp representado están agrupadas al dar los valores a los parámetros que figuran en los sumatorios.

2.1.5.2 Definición de SLP universal de longitud $\angle$

Como ya se ha dicho, es necesaria una redefinición de slp y de las instrucciones que lo componen para poder llevar a cabo la práctica del proceso evolutivo. En el apartado anterior se han expuesto los parámetros iniciales y que son necesarios para la construcción de los slp's. Una vez fijados éstos, se define el **slp universal de longitud no escalar acotada por $\angle$,** como un conjunto de instrucciones u_i no escalares con $i = 1 \ldots \angle$ donde:

$$u_i = \alpha_{-n}^{i} \left(\sum_{j=-n+1}^{i-1} \alpha_j^i \cdot u_j \right) * \left(\sum_{j=-n+1}^{i-1} \beta_j^i \cdot u_j \right) + \left(1 - \alpha_{-n}^{i}\right) \left[\frac{\left(\sum_{j=-n+1}^{i-1} \alpha_j^i \cdot u_j \right)}{\left(\sum_{j=-n+1}^{i-1} \beta_j^i \cdot u_j \right)} \right]$$

$$\alpha_{-n}^{i} \in \{0,1\}, \quad \alpha_j^i, \beta_j^i \in \mathbb{Z}$$

siendo n el número máximo de variables. Nótese, además, tal y como viene en el apartado anterior, que los u_i tal que $i = -n+1, \ldots, 0$ son las variables de entrada $x_1, \ldots, x_n$ respectivamente.

Obsérvese que las instrucción U_i además de depender de las variables del conjunto V, tiene una dependencia de las instrucciones no escalares determinadas con anterioridad (de la misma forma que ocurre en la definición del concepto de slp). Por otro lado, el parámetro α_{-n} toma valores en {0,1}, con lo que su valor seleccionará la operación que finalmente define la instrucción no escalar. Teniendo en cuenta nuestro conjunto F, dicha operación puede ser un producto o una división. El resto de parámetros toma valores enteros y determina la dependencia de instrucciones anteriores y variables, para los dos argumentos de la operación seleccionada. Además, como se mencionó con anterioridad estos últimos parámetros también determinan las posibles operaciones de suma y resta que se pueden haber producido en el slp representado.

Cada una de las instrucciones definidas así constituyen un slp de tamaño $\angle$, que tendrá un valor correspondiente a la relación que haya entre todas ellas. Dado que esta definición hace que el slp sea efectivo, todas las variables y las instrucciones u_i intervienen en la salida (resultado) del slp. Esta salida, u *output*, es necesaria para el cálculo del valor de fitness asociado al slp, y por tanto necesaria para llevar a cabo el proceso evolutivo. Dicho output no será una instrucción no escalar, sino que estará formado por una combinación lineal sobre los enteros, de las instrucciones no escalares definidas anteriormente. Así pues, será representado mediante la siguiente expresión:

$$U = \sum_{j=-n+1}^{\angle} \alpha_j^U \cdot u_j$$

Nótese que se han introducido nuevos parámetros α's.

Así pues, en la construcción de el slp universal de longitud no escalar L, se introducen un conjunto de parámetros, que además aumentan en número cuanto más instrucciones haya en dicha estructura. Fijándose en la expresión de la instrucción, y suponiendo que hay L instrucciones y n variables, el número de parámetros de un slp universal de longitud no escalar L es:

$$\angle + 2n + 2(n+1) + 2(n+2) + \ldots + 2\left(n+(\angle-1)\right) + (\angle+n) =$$

$$= \angle + \angle 2n + \frac{2(\angle-1)\angle}{2} + (\angle+n) = \angle\left[1+2n+(\angle-1)\right] + (\angle+n) =$$

$$= \underline{\underline{\angle\left(2n+\angle\right) + \left(\angle+n\right)}}$$

Nótese que tampoco se puede establecer un parámetro $\angle$ con un valor muy alto, ya que el número de parámetros sería excesivamente grande y podría no ser rentable desde el modo de vista computacional, o, en su defecto, producir resultados no satisfactorios.

2.1.5.3 Ejemplo de SLP

Véase el siguiente ejemplo, que clarifica el método de adaptación de slp's. Para ello, supóngase que $V = \{x, y\}$, que $F = \{+, -, *, /\}$ y que $\angle = 3$. Utilizando la expresión para cada instrucción del slp, nos queda:

$$u_1 = \alpha_{-2}^{1}\left(\alpha_{-1}^{1} x + \alpha_{0}^{1} y\right) * \left(\beta_{-1}^{1} x + \beta_{0}^{1} y\right) + \left(1-\alpha_{-2}^{1}\right)\left[\frac{\left(\alpha_{-1}^{1} x + \alpha_{0}^{1} y\right)}{\left(\beta_{-1}^{1} x + \beta_{0}^{1} y\right)}\right]$$

$$u_2 = \alpha_{-2}^{2}\left(\alpha_{-1}^{2} x + \alpha_{0}^{2} y + \alpha_{1}^{2} u_1\right) * \left(\beta_{-1}^{2} x + \beta_{0}^{2} y + \beta_{1}^{2} u_1\right) +$$
$$+ \left(1-\alpha_{-2}^{2}\right)\left[\frac{\left(\alpha_{-1}^{2} x + \alpha_{0}^{2} y + \alpha_{1}^{2} u_1\right)}{\left(\beta_{-1}^{2} x + \beta_{0}^{2} y + \beta_{1}^{2} u_1\right)}\right]$$

$$u_3 = \alpha_{-2}^{3}\left(\alpha_{-1}^{3} x + \alpha_{0}^{3} y + \alpha_{1}^{3} u_1 + \alpha_{2}^{3} u_2\right) * \left(\beta_{-1}^{3} x + \beta_{0}^{3} y + \beta_{1}^{3} u_1 + \beta_{2}^{3} u_2\right) +$$
$$+ \left(1-\alpha_{-2}^{3}\right)\left[\frac{\left(\alpha_{-1}^{3} x + \alpha_{0}^{3} y + \alpha_{1}^{3} u_1 + \alpha_{2}^{3} u_2\right)}{\left(\beta_{-1}^{3} x + \beta_{0}^{3} y + \beta_{1}^{3} u_1 + \beta_{2}^{3} u_2\right)}\right]$$

$$Output = U = \alpha_{-1}^{U} x + \alpha_{0}^{U} y + \alpha_{1}^{U} u_1 + \alpha_{2}^{U} u_2 + \alpha_{3}^{U} u_3$$

$$\bar{\alpha}_{-2} \in \{0,1\}, \quad \bar{\alpha}_{-1}, \ldots, \bar{\alpha}_{3} \in Z$$

Como se observa, con solo un valor $\angle = 3$, se tienen que manejar, aplicando la fórmula:

$$N\_parametros = \angle\left(2n+\angle\right) + \left(\angle + n\right) = 3\left(2\cdot 2+3\right) + \left(3+2\right) = 21 + 5 = 26$$

Por tanto, como se ha dicho, es aconsejable no utilizar un valor muy elevado debido a la gran cantidad de información que habría que manejar. Salvo esto, este modo de aplicación de slp's ofrece una gran flexibilidad a la hora de trabajar en el proceso de evolutivo, ya que habría muchas estructuras con las que poder representar toda la información que se necesita.

En el ejemplo, se sustituirían los valores de las variables del conjunto V y se calcularían las diferentes instrucciones para obtener el output final. Los parámetros serán generados de manera aleatoria en la población inicial del algoritmo evolutivo.

Capítulo 2 (II) : Diseño y Codificación

En las secciones anteriores se ha explicado el problema de la Regresión Simbólica, objetivo de este proyecto, y se ha presentado el método de resolución teórico que se va a utilizar para intentar resolverlo.

Sin embargo, lo anterior quedaría incompleto si no se detallara la manera en la que se van a diseñar las diferentes partes del algoritmo genético, así como una pseudo-codificación (a muy alto nivel, puesto que la codificación en el lenguaje en el que se desarrollará será expuesto en otro capítulo). Esto es precisamente el cometido de esta parte del capítulo. En él se van a explicar, por ejemplo, cómo se van a representar los individuos de la población, cómo va a estar formada la misma, qué estructura tiene el patrón del slp, cómo se relaciona con los individuos, los tipos de cruce y cómo se van realizar, cómo funciona la mutación, etc.

No se pretende entrar en detalles de programación, si no explicar la representación que se utilizará de las partes más relevantes que entran en juego en el proceso evolutivo.

2.2.1 Diseño y Codificación de un Individuo

Como se ha explicado en el capítulo anterior, un individuo de la población no es más que un straight line program completo, cuyos parámetros serán generados de forma aleatoria en el rango especificado por el usuario *[± a, ± b]*, de longitud (nº máximo de operaciones no escalares) $\angle$ y con un valor fitness (error cometido) asociado.

La representación que se va a fijar para un individuo de la población que utiliza *k* variables, es la que sigue:

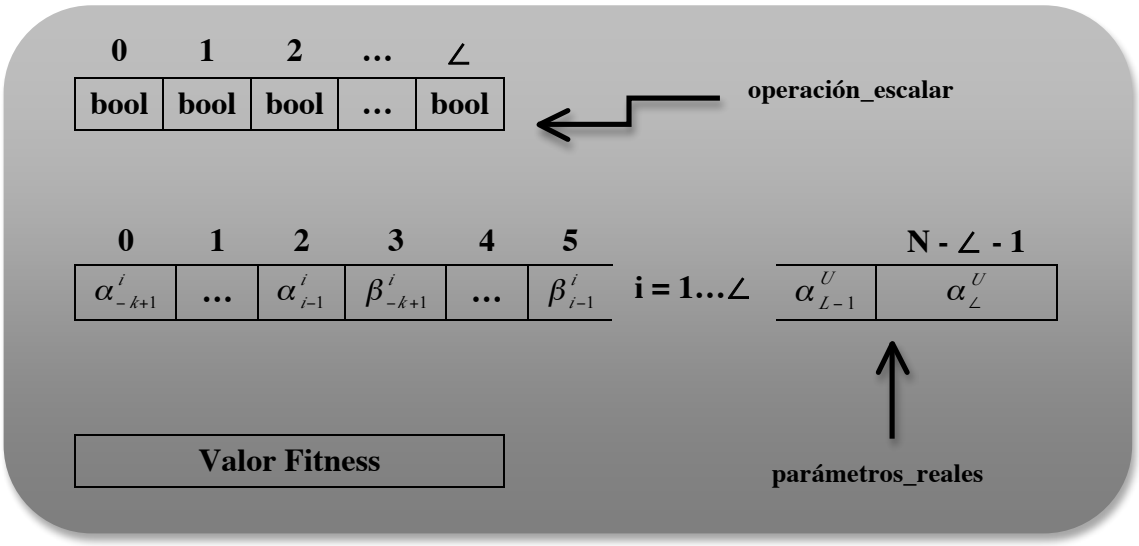


Figura 4. Ejemplo de codificación de un Individuo

Siendo *operación_escalar* un vector de booleanos de tamaño máximo la longitud del slp, que indica para cada U_i si la operación a realizar es una multiplicación (true) o una división (false), y *parámetros_reales* un vector de los reales generados en el rango antes descrito de tamaño máximo *N - ∠ - 1*, con *N = numero total de parámetros*, fácilmente calculable con la fórmula descrita en la parte 2, que almacena todos los α y β de todos los U_i. Además, cada individuo tendrá asociado un valor fitness o error.

2.2.2 Diseño y Codificación de una Población

Por lógica, una población de tamaño *T* va a estar compuesta por *T* individuos. En este caso, el diseño es sencillo, ya que una población no será más que un vector de individuos de tamaño *T – 1*. Su estructura sería de la siguiente manera:

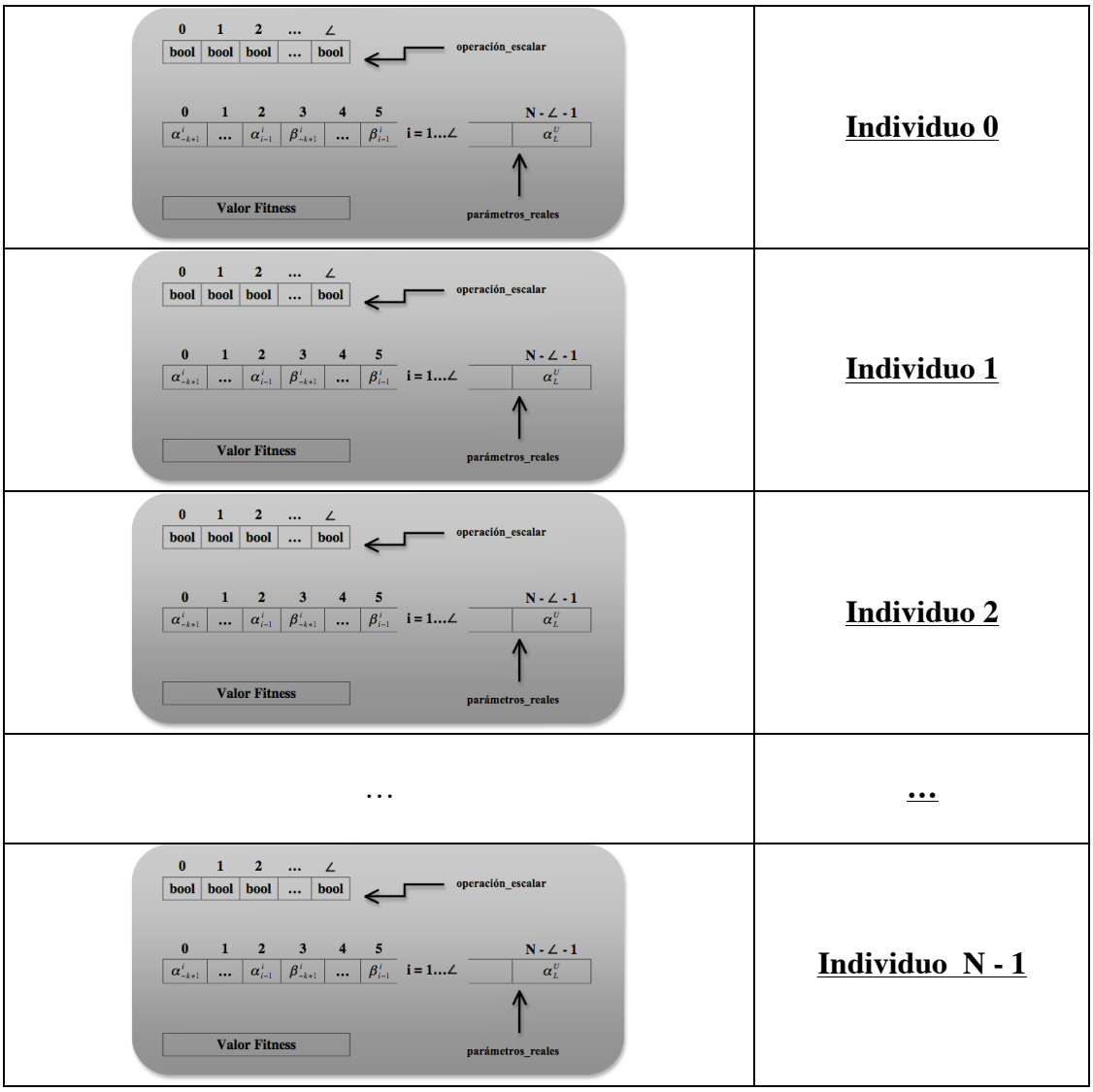


Figura 5. Ejemplo de codificación de una Población

2.2.3 Diseño y Codificación del SLP Patrón

La estructura del SLP universal o patrón resulta crucial en el diseño del algoritmo evolutivo, puesto que cada individuo será puesto en él para poder calcular el Output que produce. Se podría decir que es como una plantilla en la cual se irán procesando los individuos para calcular su Output asociado.

Puesto que un individuo (slp) va a tener $\angle$ instrucciones no escalares, el slp patrón tendrá tamaño $\angle + 1$, ya que también necesita la descripción de la instrucción que constituye el output. Supuesto n el número de variables, la representación es la siguiente:

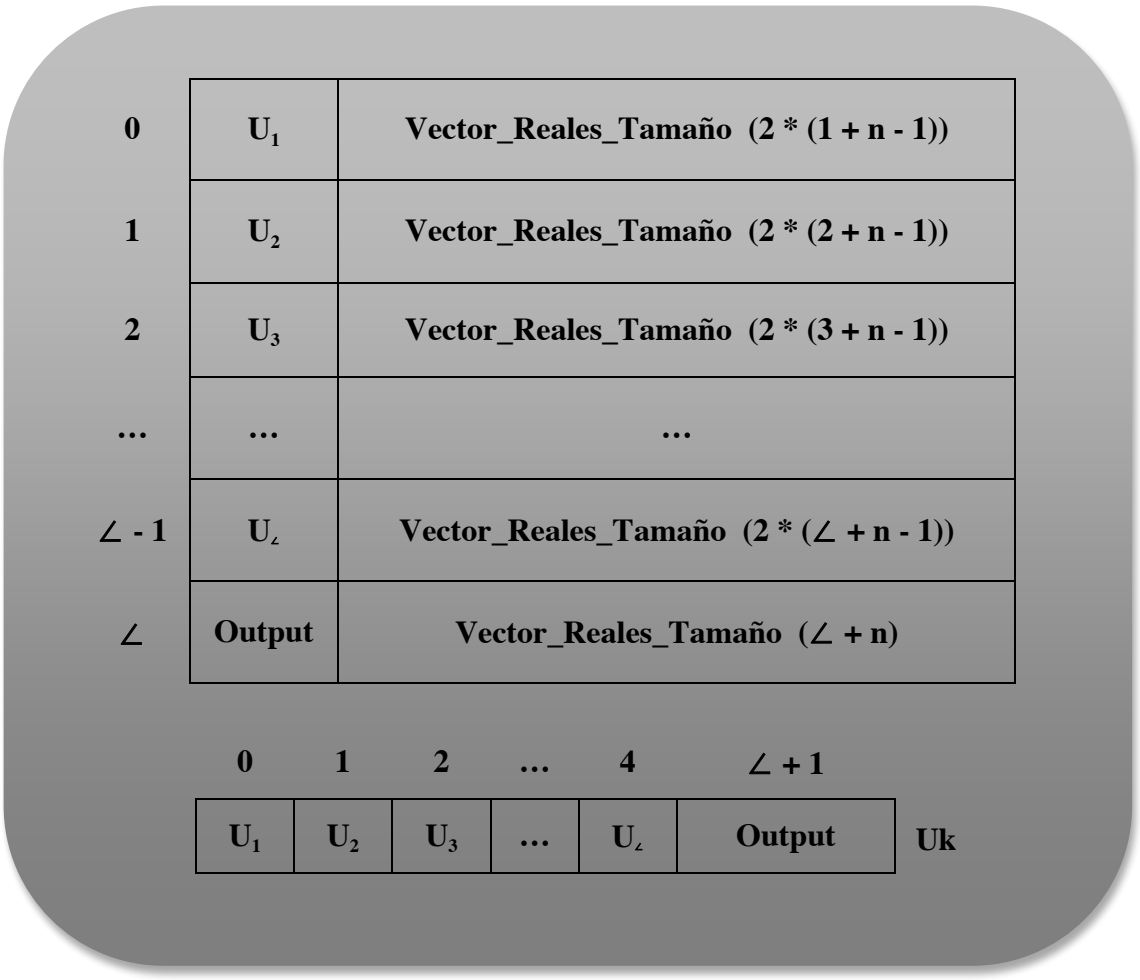


Figura 6. Ejemplo de codificación del SLP Patrón

Donde cada vector asociado a U_i contendrá los parámetros α y β de dicha instrucción. Además, el vector correspondiente a la salida tendrá un tamaño igual al número de variables mas al de instrucciones, y el vector *Uk* contendrá los valores de las instrucciones U_i así como el valor final del slp, el Output.

2.2.4 Utilización del SLP Patrón

En este apartado se explicará el cometido del SLP patrón y la relación que se establece entre él y un individuo para poder obtener su valor (output).

Nada más comenzar, se creará el patrón tal y como se detalla en la figura 18 de acuerdo a los valores que el usuario haya definido para el número de variables del programa, *n*, y el número de instrucciones o longitud, ∠. A continuación, será necesario calcular el output o valor asociado a cada individuo (es decir, a cada slp) que conforman la población, el número de parámetros (fácilmente calculable como se ha dicho con la fórmula del capítulo anterior), y aquí es donde el patrón toma todo su sentido. Para ello, el vector de parámetros reales del individuo se partirá en ∠ + *1* trozos, cuyos tamaños serán los tamaños que el patrón tenga para cada U_i. Una vez esté partido, los parámetros de cada trozo se moverán al correspondiente vector asociado a U_i del patrón. Esto se ve de forma más clara en la siguiente figura:

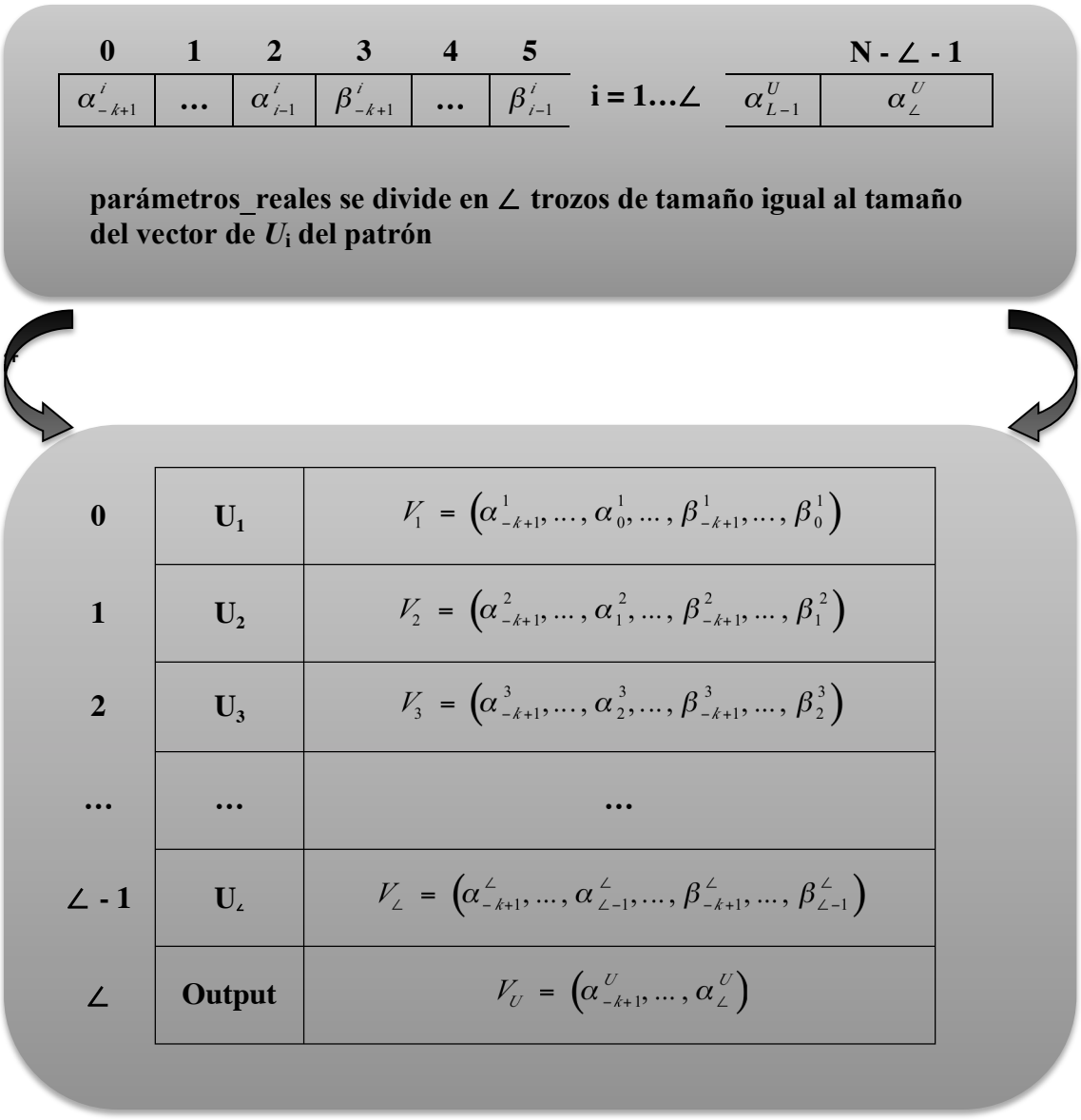


Figura 7. Utilización del SLP Patrón

Una vez queden en el patrón los parámetros del individuo asociados a su U_i correspondiente, se podrá comenzar a calcular el valor de cada instrucción, que se almacenará en el vector U_k, así como el valor final del individuo (output).

2.2.5 Ejemplo de transformación Individuo → SLP Patrón

La visión anterior puede no quedar suficientemente clara, por lo que el objetivo de este apartado es crear un pequeño ejemplo en el que se muestre cómo un individuo se transforma (inserta) en el esquema general del SLP.

Supóngase un número de variables $n = 2$ y un número de instrucciones no escalares $\angle = 2$. Con la definición de SLP Universal del capítulo anterior, obtenemos que el número total de parámetros es $3 * (2 * 2 + 3) + (3 + 2) = 26$. Con esto, y el diseño definido anteriormente, un individuo tendría la siguiente representación:

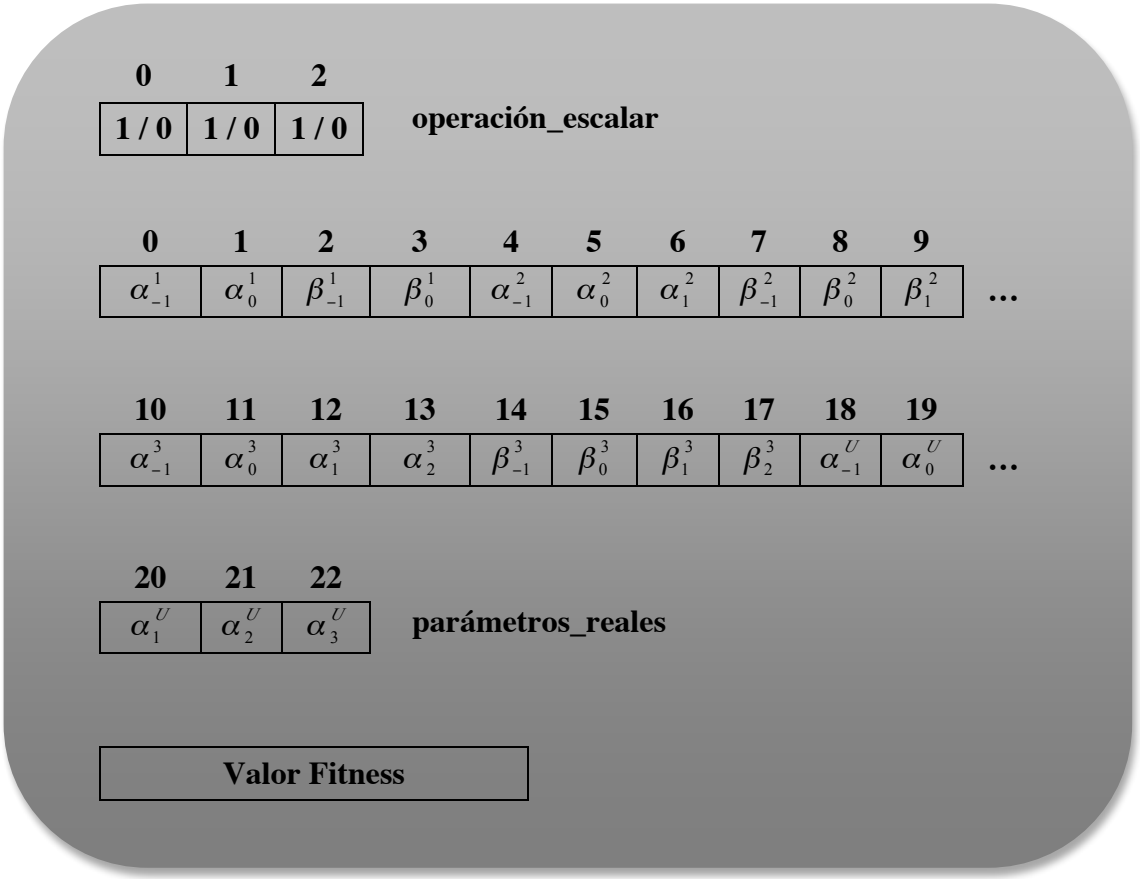


Figura 8. Ejemplo de codificación de un Individuo

Una vez tenemos el individuo representado, el SLP partiría el vector de reales en 4 trozos: el primero, de tamaño $(2 * (1 + 2 - 1)) = 4$; el segundo, de tamaño $(2 * (2 + 2 - 1)) = 6$, el tercero, de tamaño $(2 * (3 + 2 - 1)) = 8$; y el último, de tamaño $(3 + 2) = 5$. La forma que toma entonces el SLP patrón se muestra a continuación:

0	**U_1**	$V_1 = \left(\alpha_{-1}^1, \alpha_0^1, \beta_{-1}^1, \beta_0^1\right)$
1	**U_2**	$V_2 = \left(\alpha_{-1}^2, \alpha_0^2, \alpha_1^1, \beta_{-1}^2, \beta_0^2, \beta_1^2\right)$
2	**U_3**	$V_3 = \left(\alpha_{-1}^3, \alpha_0^3, \alpha_1^3, \alpha_2^3, \beta_{-1}^3, \beta_0^3, \beta_1^3, \beta_2^3\right)$
3	**Output**	$V_U = \left(\alpha_{-1}^U, \alpha_0^U, \alpha_1^U, \alpha_2^U, \alpha_3^U\right)$

Figura 9. Transformación Individuo → SLP Patrón

Una vez que el individuo es insertado en este SLP plantilla, éste último se encargaría, mediante las operaciones necesarias y siguiendo la fórmula desarrollada para calcular el valor U_i vista en el capítulo anterior (definición de SLP Universal), de calcular la salida asociada a dicho individuo..

2.2.6 Cálculo del Fitness de un Individuo

Sea el fichero de puntos muestra proporcionado por el usuario para realizar su aproximación, cuyo formato es el siguiente:

$$x_1 \quad y_1 \quad z_1 \quad \ldots \quad w_1 \quad f\left(x_1, y_1, z_1, \ldots, w_1\right)$$

$$x_2 \quad y_2 \quad z_2 \quad \ldots \quad w_2 \quad f\left(x_2, y_2, z_2, \ldots, w_2\right)$$

$$x_3 \quad y_3 \quad z_3 \quad \ldots \quad w_3 \quad f\left(x_3, y_3, z_3, \ldots, w_3\right)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$x_n \quad y_n \quad z_n \quad \ldots \quad w_n \quad f\left(x_n, y_n, z_n, \ldots, w_n\right)$$

Figura 10. Ejemplo de fichero de puntos muestra

Para calcular el error cometido o fitness del individuo, es necesario en primer lugar calcular el Output que produce el individuo, a través del SLP patrón, para las *n* diferentes componentes, a excepción del valor de la función en esos puntos, es decir:

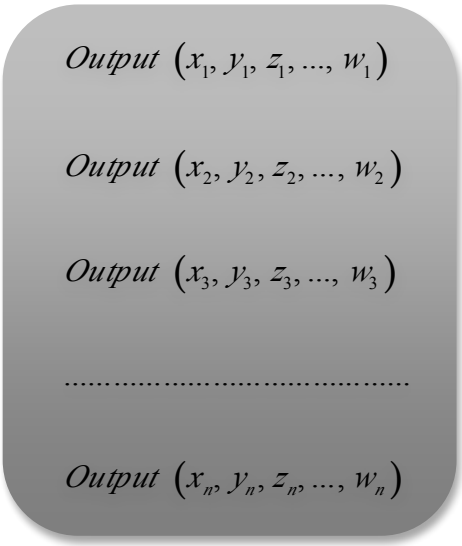


Figura 11. Cálculos necesarios para el fitness

Una vez calculadas todas las salidas que produce el individuo para las diferentes componentes, no hay más que aplicar la fórmula del fitness presentada en el capítulo anterior, trasladada a esta situación, de la siguiente manera:

$$Fitness\ (Individuo) = \frac{1}{n} \sum_{i=1}^{n} \left(Output\ (x_i, y_i, z_i, ..., w_i) - f(x_i, y_i, z_i, ..., w_i)\right)^2$$

2.2.7 Selección por K-Torneo

La selección por K-Torneo es de las más habituales y, a la vez, de las selecciones más sencillas de diseñar.

Se generan *K* números (enteros) comprendidos entre *[1, Tamaño_Población]*, que se corresponderán con los índices de algunos individuos de la población. Y, para finalizar, de entre esos individuos seleccionados, se escoge el que menor valor fitness tenga.

2.2.8 Selección por Ruleta

La selección por Ruleta que se necesita en este proyecto es ligeramente diferente al método tradicional, por lo que es necesaria la explicación de su diseño. Para empezar, en una variable *F* se almacena la suma de los valores fitness de todos los individuos de la población, tal y como se muestra:

$$Fitness\ (Individuo\,1)$$

$$Fitness\ (Individuo\,2)$$

$$Fitness\ (Individuo\,3)$$

$$F = \sum_{i=1}^{n} Fitness\ (Individuo\,i)$$

Figura 12. Cálculos para la Ruleta tradicional

Hasta este punto, los cálculos son suficientes para una Ruleta tradicional si se buscan los valores fitness mayores. Pero, en este proyecto, interesan los fitness más pequeños, ya que es el error que se comete, por lo que es necesario dar la vuelta de alguna manera a los cálculos anteriores. Por ello, se define una variable *F_auxiliar* que contendrá el valor de *F – fitness de cada individuo*, es decir:

$$F - Fitness\ (Individuo\,1)$$

$$F - Fitness\ (Individuo\,2)$$

$$F - Fitness\ (Individuo\,3)$$

$$F\_auxiliar = \sum_{i=1}^{n} F - Fitness\ (Individuo\,i)$$

Figura 13. Cálculos para la Ruleta de este proyecto

Despues de calcular esta nueva variable, el siguiente paso sería el de generar un número aleatorio entre *[1, F_auxiliar]*. Y, para finalizar, el último paso consiste en ir sumando los valores fitness de los individuos, desde el primero, y en el momento en que un individuo haga que con su valor fitness la suma sea mayor o igual que el número aleatorio, éste se escoge como seleccionado.

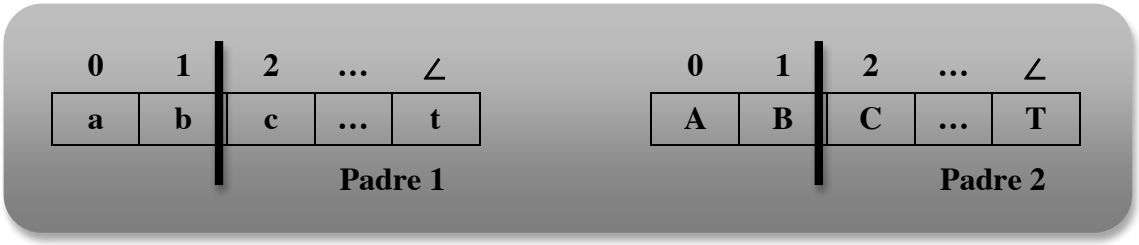
2.2.9 Tipos de Cruce

Es este proyecto se van a abarcar 4 tipos de cruce: en 1 punto, en 1 punto selectivo, uniforme y uniforme selectivo. La explicación que se va a realizar sobre cada tipo de cruce va a ser en su mayoría gráfica, ya que es como mejor puede entenderse.

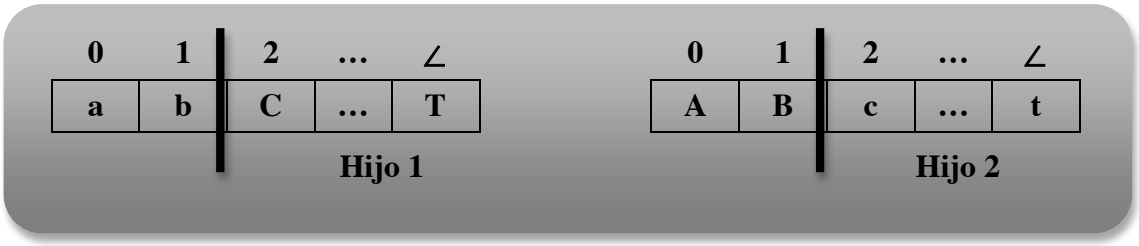
2.2.9.1 Cruce en 1 Punto

En este tipo de cruce se van a cruzar, por un lado los vectores booleanos de los individuos padre, y por otro lado el de los parámetros reales.

Para empezar, se genera un número aleatorio entre *[1, ∠]*, para saber el punto de cruce a aplicar en el vector de booleanos, y, después se aplica el cruce normal visto en capítulos anteriores. Por ejemplo, sean los siguientes vectores booleanos de dos individuos padre, con el punto de cruce en ellos:



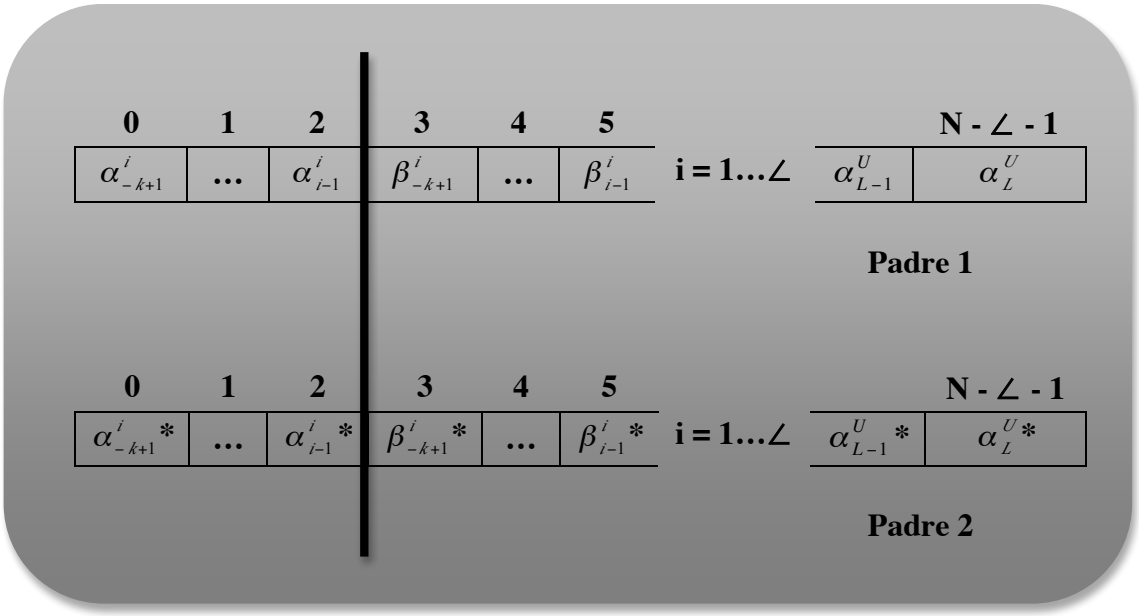
Tras aplicar el cruce en ese punto, se generarán los siguientes hijos:



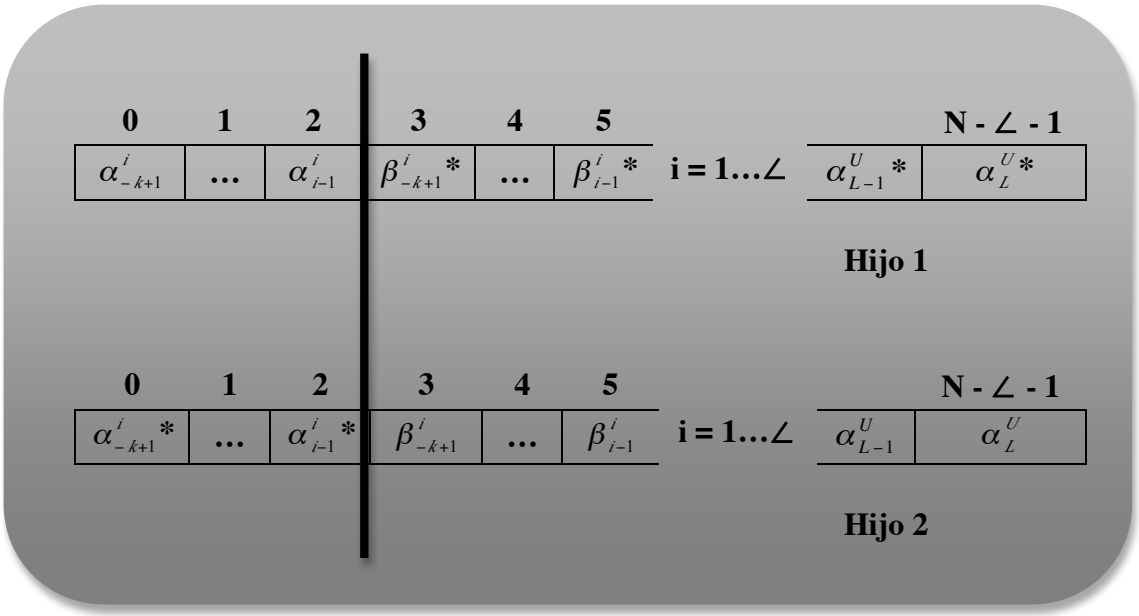
Figuras 14-15. Cruce en un punto para los vectores booleanos

Como se puede observar, es un cruce en 1 punto exactamente igual al visto en capítulos anteriores.

Después de aplicar el cruce para los vectores booleanos, se generará un valor aleatorio entre *[1, N - ∠ - 1]*, siendo *N* el número de parámetros totales y se aplicará el cruce de igual forma para los vectores de reales. Para acabar con el ejemplo, supónganse los siguientes vectores de dos individuos padres, con el punto de cruce:



Tras aplicar el cruce en ese punto, se obtienen los siguientes hijos:



Figuras 16-17. Cruce en un punto para los vectores de reales

2.2.9.2 Cruce en 1 Punto Selectivo

El cruce en 1 punto selectivo está basado en el cruce en 1 punto normal, pero restringe las posibles posiciones del punto de cruce.

En esta técnica, el cruce que se realiza tanto para los vectores booleanos de los padres, como para los parámetros reales correspondientes al Output (últimos $n + \angle$ parámetros siendo n el nº de variables) es el cruce en 1 punto normal explicado en el apartado anterior, ya que aquí solo entran en juego los diferentes parámetros reales de los distintos U_i's del vector de reales.

Recuérdese que el vector de reales está formado por ristras de α's y β's correspondientes, como se ha dicho, a las diferentes instrucciones del programa. Pues bien; en este tipo de cruce, el **punto de cruce** que se generé aleatoriamente **no podrá** de ninguna manera **cortar ninguna de estas ristras**. De ser así, el punto de cruce será recolocado de tal forma que se cumpla la premisa anterior. Además, el punto de cruce será movido a una posición elegida de entre dos posibles de tal forma que la diferencia de la posición actual a la posible posición válida sea la menor de las dos. Dado que esta recolocación que puede ser llevada a cabo, es la misma para los dos padres, se van a explicar las posibles posibilidades de forma gráfica para un sólo uno de ellos.

Sea el individuo padre del ejemplo anteriormente mostrado, y los siguientes puntos de cruce:

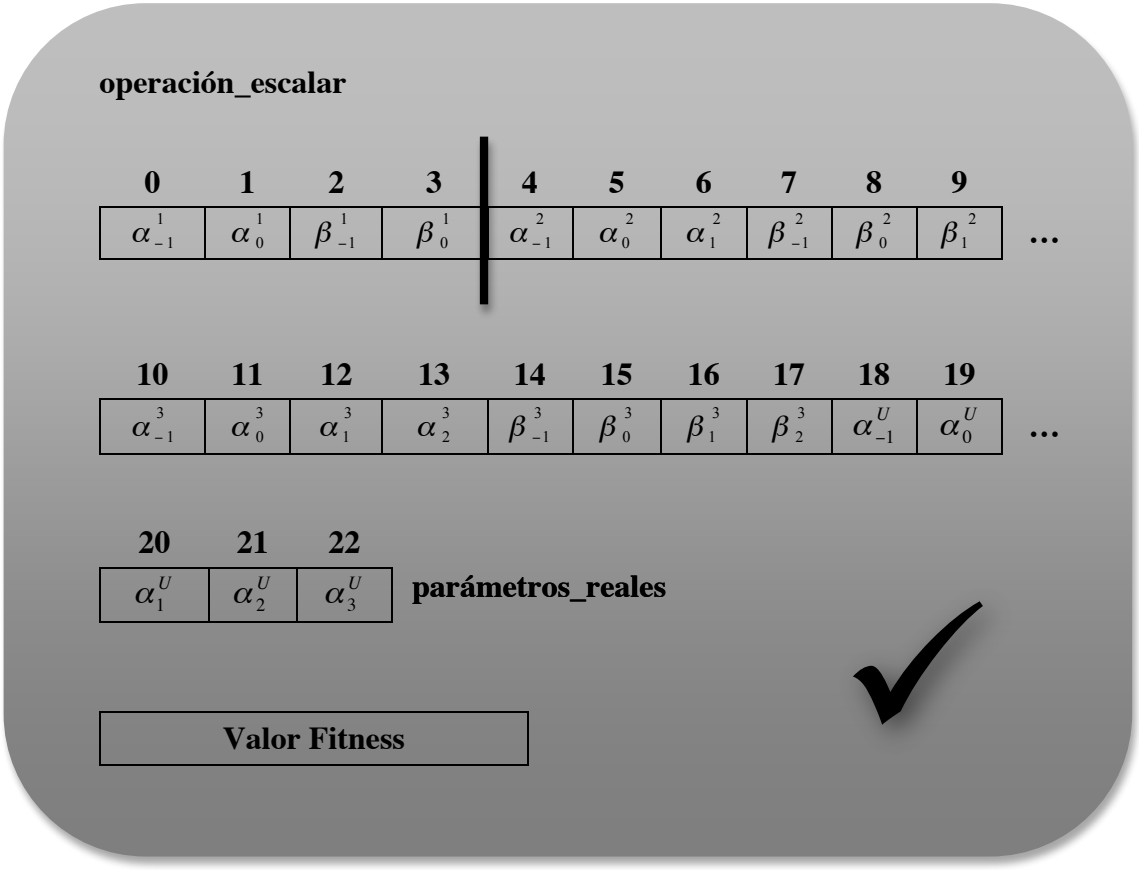


Figura 18. Cruce en un punto selectivo válido

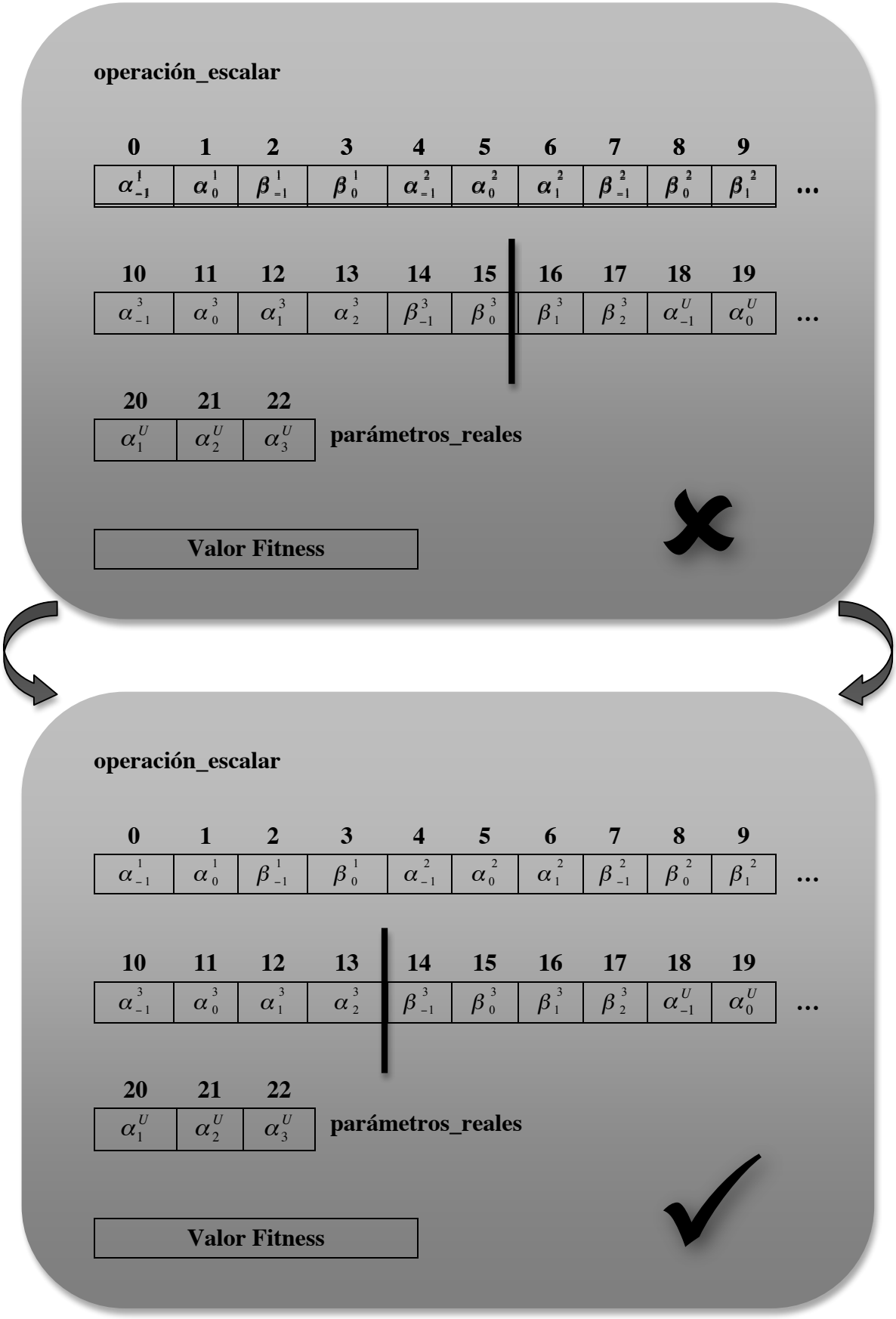


Figura 19. Recolocación del punto de cruce

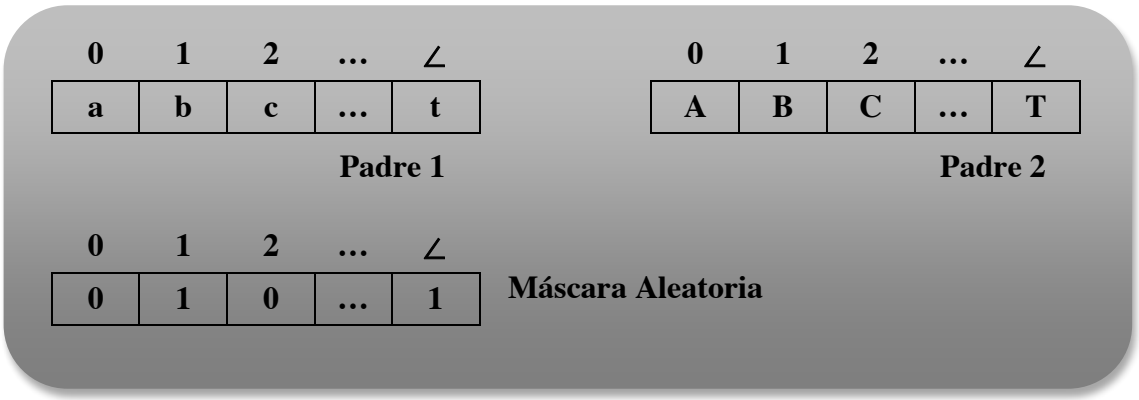
Como se puede observar, el punto de cruce es recolocado de tal forma que no corta la ristra de β's. Ésta política anterior se aplicaría a cualquier punto que no fuera válido.

NOTA : Nótese que el punto de cruce no válido de la figura 30 está en la posición 16 y se recoloca a la 14 puesto que la diferencia entre la posición actual y las posibles para recolocar (14 y 18) es la misma. Si, por ejemplo, el punto de cruce hubiera estado en la posición 17, el punto de cruce se hubiera recolocado a la posición 18.

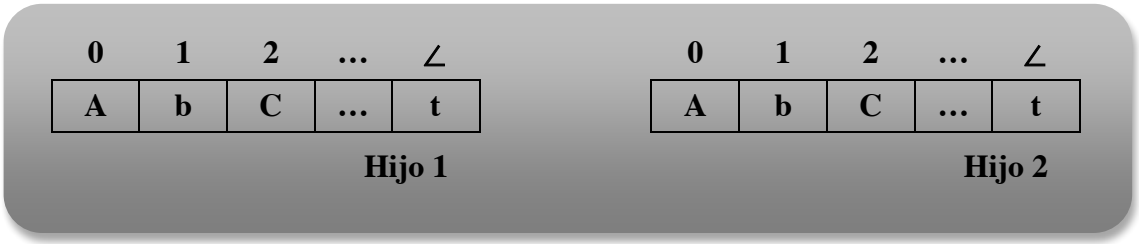
2.2.9.3 Cruce Uniforme

En este tipo de cruce se van a cruzar, por un lado los vectores booleanos de los individuos padre, y por otro lado el de los parámetros reales.

Para empezar, se genera una máscara de tamaño $\angle$ de valores booleanos y, después se aplica el cruce uniforme visto en capítulos anteriores. Por ejemplo, sean los siguientes vectores booleanos de dos individuos padre, con una máscara aleatoria cualquiera:



Tras aplicar el cruce en ese punto, se generarán los siguientes hijos:



Figuras 20-21. Cruce en un punto para los vectores booleanos

Como se puede observar, es un cruce uniforme exactamente igual al visto en capítulos anteriores.

Después de aplicar el cruce para los vectores booleanos, se generará otra máscara aleatoria de tamaño $N - \angle - 1$ de valores booleanos, siendo N el número de parámetros totales y se aplicará el cruce uniforme de manera de igual forma para los vectores de reales. No se pondrá explicación gráfica pues se supone que la explicación teórica es suficiente; ya se ha puesto para el cruce en 1 punto, y es algo que se realiza de manera análoga con las características del cruce uniforme.

2.2.9.4 Cruce Uniforme Selectivo

El cruce uniforme selectivo se basa en el cruce uniforme normal, y mantiene la idea del cruce en 1 punto selectivo de no cortar ristras de α's y β's.

En esta técnica, el cruce que se realiza tanto para los vectores booleanos de los padres, como para los parámetros reales correspondientes al Output (últimos $n + \angle$ parámetros siendo n el nº de variables) es el cruce uniforme normal explicado en el apartado anterior, ya que aquí solo entran en juego los diferentes parámetros reales de los distintos U_i's del vector de reales.

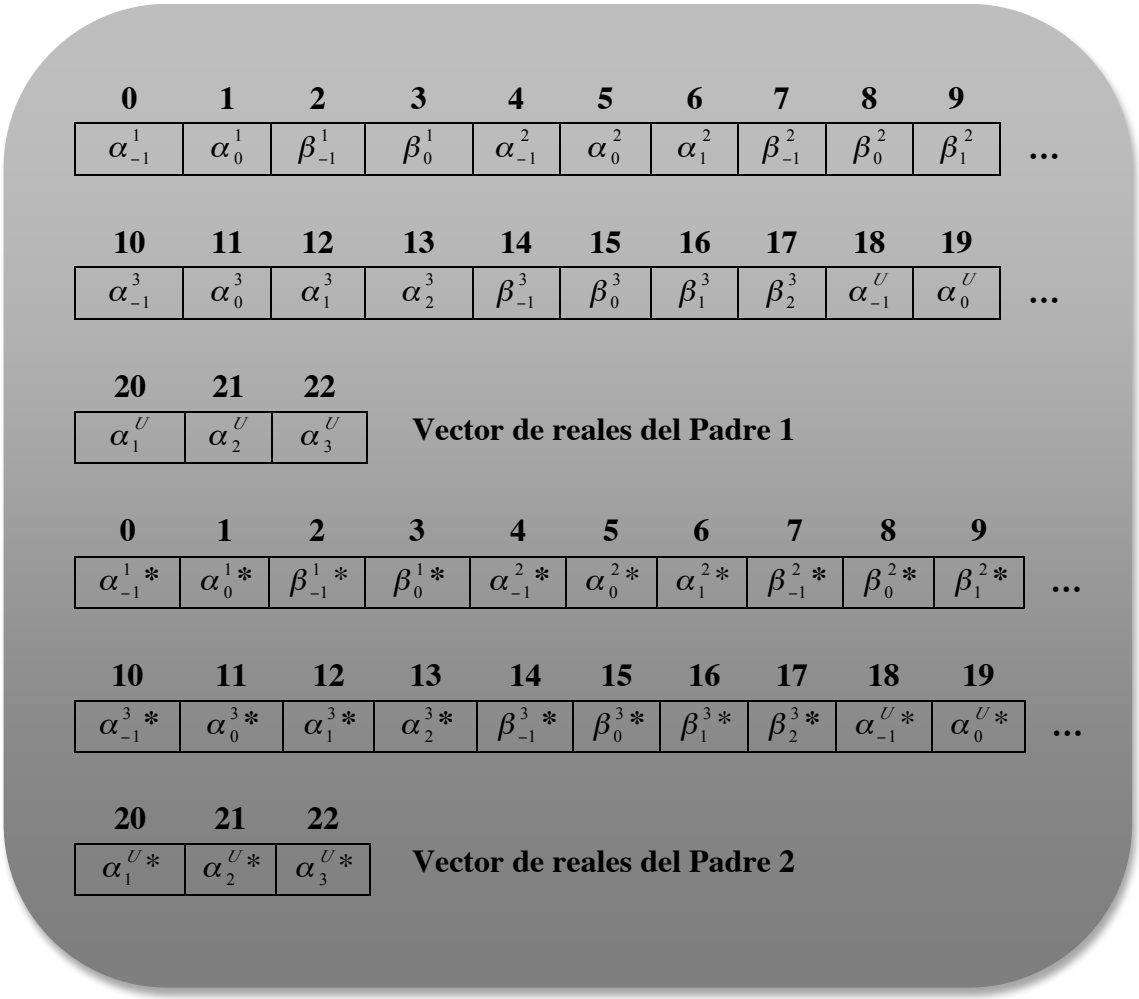
Recuérdese que el vector de reales está formado por ristras de α's y β's correspondientes, como se ha dicho, a las diferentes instrucciones del programa. Pues bien; en este tipo de cruce, la manera de generar aleatoriamente la máscara para los parámetros reales es un tato distinta: **por cada ristra de α's y β's se producirá un valor booleano 1 ó 0**. Dado que en cada U_i hay dos ristras de igual tamaño de α's y β's y luego el Output tiene parámetros independientes, el tamaño de ésta máscara será de *(2 * nº de U_i's) + Tamaño_Output*. Una vez esté creada ésta máscara **se aplicará el cruce normal**. La explicación gráfica se realizará para un individuo hijo, aunque para el otro sería análogo aunque cambiando el significado de la máscara tal y como se vió en la explicación del cruce uniforme en anteriores capítulos.

Sea la siguiente máscara aleatoria generada, tal y como se acaba de explicar, y los vectores de reales individuos padre para el ejemplo anteriormente mostrado :

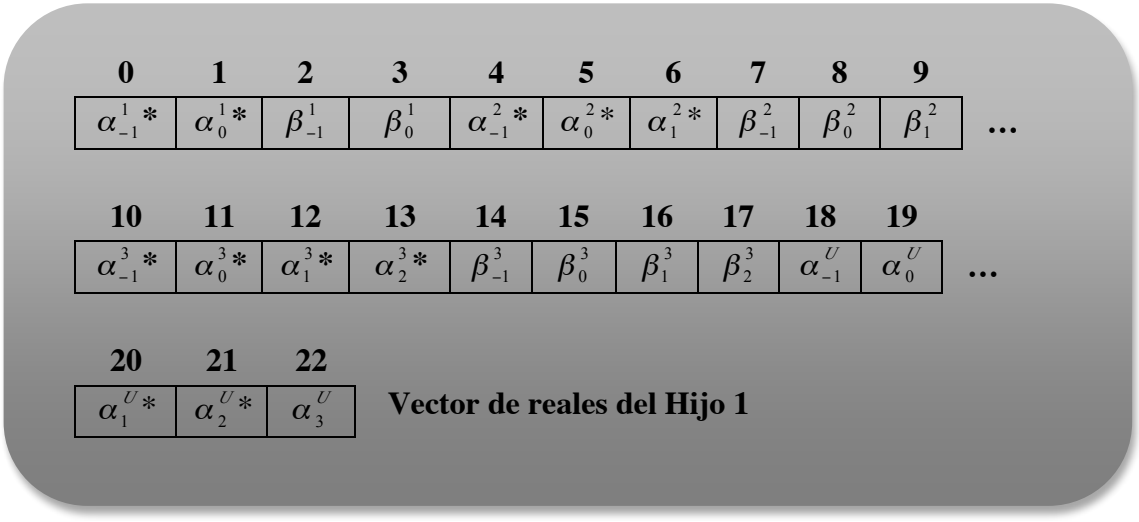
0	1	2	3	4	5	6	7	8	9	10
0	1	0	1	0	1	1	1	0	0	1

Máscara Aleatoria de tamaño (2 * 3) + 5 = 11

Figura 22. Máscara aleatoria generada para el ejemplo



Tras aplicar el cruce utilizando la máscara anterior para cada ristra de parámetros, obtenemos el siguiente vector de reales para el hijo:



Figuras 23-24. Ejemplo Cruce Uniforme Selectivo

El hijo 2 podría calcularse tal y como se ha dicho, interpretando los valores de la máscara de manera inversa.

Como se puede observar, salvo por la pequeña diferencia de generación de la máscara aleatoria para los parámetros reales de los distintos U_i's, el cruce uniforme se realiza igual.

2.2.10 Tipos de Mutación

En este proyecto se han diseñado dos tipos de mutación: la normal, y la sesgada. Tanto en una como otra no se van a exponer ejemplos gráficos explicativos, ya que son lo suficientemente sencillas como para poder explicarse de manera escrita.

2.2.10.1 Mutación Normal

En este tipo de mutación, todos los parámetros, tanto booleanos como reales, tienen la misma posibilidad de ser escogidos para sufrir el cambio.

La idea es tener la visión de que todos ellos se disponen en un único vector (aunque en la implementación no haga falta dicho vector) y generar un número aleatorio entre *[1, N]* siendo *N* el número de parámetros totales calculado con la fórmula del capítulo anterior. Ese número será el índice que indica en el vector cuál de todos los parámetros ha sido elegido para mutar.

2.2.10.2 Mutación Sesgada

Ahora, los parámetros booleanos y los reales no tienen la mismos probabilidad de ser elegidos para mutar. Además, se sigue conservando la visión de los dos vectores independientes.

De inicio se escoge, generando un nuevo número aleatorio, si se muta un booleano (*número_aleatorio* < 0.5) o un real (*número_aleatorio* ≥ 0.5). El parámetro a mutar entonces, se escogerá de la siguiente forma: si el resultado anterior fue el de mutar un booleano, se genera otro número aleatorio entre *[1, ∠]* que indicará el parámetro a mutar; si, por el contrario, se decidió que se mutara un real, el número aleatorio a producir estará comprendido entre *[1, N - ∠ - 1]*, que será el índice del parámetro a mutar en el vector de reales.

2.2.11 Otras características del Algoritmo Genético

Además de todo el diseño anterior, el proceso evolutivo va a tener otras características que le harán mas variado en cuanto a formas de resolución y de resultados.

REEMPLAZAMIENTO

El algoritmo va a soportar dos tipos de reemplazamiento en la nueva población tras haberse aplicado el cruce elegido de entre los anteriores:

- *Reemplazamiento de hijos :* siempre se transmiten los hijos a la nueva población, tanto si tienen mejor valor fitness que los padres o peor.
- *Reemplazamiento de los mejores :* se eligen qué dos individuos tienen mejor valor fitness de entre padres e hijos y se insertan en la nueva población

TIPO DE ELITISMO

El algoritmo va a considerar o no el elitismo en función de las necesidades del problema:

- *Elitista :* antes de empezar a aplicar el proceso evolutivo, cuando se crea la nueva población, se inserta en ella el mejor individuo (menor valor fitness) de la población antigua.
- *No Elitista :* lo anterior no se realiza y se aplica el algoritmo evolutivo sin más.

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

Capítulo 3 : Metodología e implementación

Pablo Solar Rodríguez

ÍNDICE

LISTA DE FIGURAS

Capítulo 3: Implementación

Este capítulo estará dedicado a la explicación de las características y herramientas necesarias para poder llevar a cabo este proyecto, es decir, decir cómo ha sido desarrollado, utilizando qué cosas y por qué.

Se empezará por exponer por qué se ha utilizado Java para desarrollar el algoritmo genético, así como las partes de las que constará el mismo sin entrar en demasiados detalles de código.

A continuación, se presentará el programa gnuplot, que ha sido utilizado en una parte del proyecto para la representación gráfica de funciones y resultados.

Para finalizar, se detallarán las características físicas y lógicas del equipo en el que se ha desarrollado el proyecto.

3.1 Java como Lenguaje de Programación

Este proyecto ha sido desarrollado en Java ya que tiene ciertas características que hacen que su uso hoy en día esté muy extendido y se considere como uno de los lenguajes de programación más potentes que existen. Las más notables son:

- *Es orientado a objetos*: si bien existen detractores de esta modalidad, la programación orientada a objetos resulta muy conveniente para la mayoría de las aplicaciones, y es esencial para los videojuegos. Entre las ventajas más evidentes que ofrece se encuentra un gran control sobre el código y una mejor organización, dado que basta con escribir una vez los métodos y las propiedades de un objeto, independientemente de la cantidad de veces que se utilicen.

- *Es muy flexible*: Java es un lenguaje especialmente preparado para la reutilización del código; permite a sus usuarios tomar un programa que hayan desarrollado tiempo atrás y actualizarlo con mucha facilidad, sea que necesiten agregar funciones o adaptarlo a un nuevo entorno.

- *Funciona en cualquier plataforma*: a diferencia de los programas que requieren de versiones específicas para cada sistema operativo (tales como Windows o Mac), las aplicaciones desarrolladas en Java funcionan en cualquier entorno, dado que no es el sistema quien las ejecuta, sino la máquina virtual (conocida como JVM).

- *Su uso no acarrea inversiones económicas*: programar en Java es absolutamente gratis; no es necesario adquirir ninguna licencia, sino simplemente descargar el kit de desarrollo (Java Development Kit o JDK) y dar riendas sueltas a la imaginación.

- *Es código abierto*: Java ofrece el código de casi todas sus librerías nativas para que los desarrolladores puedan conocerlas y estudiarlas en profundidad, o bien ampliar su funcionalidad, beneficiándose a ellos mismos y a los demás.

- *Es un lenguaje expandible*: continuando con el punto anterior, cada programador tiene la libertad de revisar y mejorar el código nativo de Java, y su trabajo puede convertirse en la solución a los problemas de muchas personas en todo el mundo. Infinidad de desarrolladores han aprovechado esta virtud del lenguaje.

Este proyecto se ha desarrllado sobre una versión de Java 1.8, sin ninguna librería externa de apoyo salvo la ejecución del programa gnuplot que se explicará más adelante con detalle.

3.2 Implementación del Algoritmo Genético

El Algoritmo Genético de este proyecto va a intentar reproducir lo más fielmente posible las diferentes partes vistas en la teoría en capítulos anteriores. Por ello, se dividirá en 4 grandes partes a programar: Individuo, Población, SLP y Algoritmo.

El objetivo de este apartado es presentar cada clase, mostrando su especificación y explicando a grandes su cometido, así como una breve explicación del funcionamiento de cada función

3.2.1 Clase Individuo

La clase Individuo va implementar en Java la codificación que se ha visto en la última parte del capítulo anterior. Su **objetivo** será la de representar a un individuo de la población acorde a los requisitos necesarios para este proyecto.

Su correcta programación es vital, puesto que una población se basa en individuos y ésta, a su vez, es el sustento básico del proceso evolutivo.

En la Figura 1 se puede observar la definición que se ha dado a esta clase. Nótese que en las funciones no se ha puesto comentarios puesto que se van a explicar brevemente aquí; esto se mostrará en el capítulo 5 correspondiente al código fuente.

Figura 1. Especificación de la clase Individuo

Fijándose en dicha figura, se puede ver que un individuo consta los siguientes campos:

```
// Vector cuyo tamanio sera igual al numero de Ui's del individuo y que nos
// indicara si en el Ui se produce una multiplicacion (true) o una
// division (false)
Vector<Boolean> operacionEscalar;

// Vector de alphas y betas (parametros) de valores reales
Vector <Double> parametrosReales;

// Variable que almacena el valor de la funcion fitness del individuo
double fitness;

// Rango de valores de los parametros reales
double limiteInferior, limiteSuperior;

// Tamaño del individuo
int tamanio;

// Numero maximo de operaciones no escalares
int maxoperationsL;
```

Como funciones constructoras, se tiene la que viene por defecto y que inicializa los vectores vacíos y otro que crea un individuo con unos parámetros específicos, tales como su tamaño, el número de operaciones no escalares, etc.

3.2.2 Clase Población

La clase población es, posiblemente, la más importante de todas a la hora de programarla, ya que tiene por **objetivo**:

- Implementar el "Cruce en 1 un punto"
- Implementar el "Cruce en 1 un punto selectivo"
- Implementar el "Cruce Uniforme"
- Implementar el "Cruce Uniforme Selectivo"
- Implementar los métodos de selección "Ruleta" y "K-Torneo"
- Implementar la "Mutación Normal" y la "Mutación Sesgada"
- Calcular el valor fitnes de la población de individuos
- Tener constancia de qué individuo es considerado "el mejor"

La siguiente figura muestra la especificación de la clase:

```
tfm.genetics.poblacion.Poblacion
{
        population : Vector<Individuo>
        Poblacion()
        Poblacion(int, int, int, double, double)
        calculaFitnessPoblacion(String, int, SLP)
        mejorIndividuo()
        getPopulation()
        ruleta()
        kTorneo(int)
        cruce1Punto(Individuo, Individuo, Individuo, Individuo)
        cruce1PuntoSelectivo(Individuo, Individuo, Individuo, Individuo, SLP)
        cruceUniforme(Individuo, Individuo, Individuo, Individuo)
        cruceUniformeSelectivo(Individuo, Individuo, Individuo, Individuo, SLP)
        mutacion(Individuo)
        mutacionSesgada(Individuo)
};
```

Figura 2. Especificación de la clase Población

Como se puede observar, una población es tremendamente simple: un vector de individuos. Las maneras de inicializarla son: por defecto, sin individuos; pasándole el tamaño de población, el de un individuo, el número de operaciones no escalares y dos

variables que indican el rango de valores que pueden tomar los parámetros reales del individuo. Esta función se encargará de crear tantos individuos como tamaño se especifique.

Otras funciones son la de calcular el fitness de cada individuo de la población, apoyándose en la ruta del archivo donde se encuentran los puntos muestra (los cuales leerá), el número de variables y la estructura del SLP patrón, y una función que devuelve el mejor individuo de la población, es decir, su menor valor fitness.

También se programa en esta clase los dos métodos de selección, k-torneo y ruleta, así como los cuatro tipos de cruce que se han visto en capítulos anteriores, que necesitarán tanto los individuos padres como los hijos, los cuales se modificarán.

Por último, se implementan los dos tipos de mutación vistos, que necesitan un individuo elegido para su modificación.

Como se ha comentado anteriormente, no se profundiza mucho en cómo se programa cada función puesto que queda reflejado en el capítulo 5 (código fuente). Sólo se pretende dar una visión general de lo que se incluye en cada clase del algoritmo.

3.2.3 Clase SLP

Otra de las clases más importantes, junto con la de Población, es sin duda la que tiene por **objetivo** implementar la estructura del straight line program patrón, puesto que casi todas las operaciones que se realizan en los individuos tienen lugar en ésta clase.

Aquí se realizarán operaciones tales como el cálculo de una instrucción de un individuo (recuérdese que un individuo es un straight line program), la obtención del output, etc.

En la siguiente figura se comprueba que el SLP Patrón está implementado con un diccionario formado por un entero para indicar el U_i, es decir, el vector de parámetros asociado al mismo. Además, consta de otro vector para almacenar los valores de éstas instrucciones.

```
// Variable que establece el numero de operaciones distintas de {+,-}
private int maxOperationsL;

// Variable que establece el numero de variables que utilizara el programa
private int nVariables;

// Diccionario que representara al slp
HashMap <Integer, Vector<Double>> slp = new HashMap <Integer,
Vector<Double>>();

// Vector que contendra los valores calculados de cada Ui
Vector <Double> Uk;
```

```
tfm.genetics.slp.SLP
{
        maxOperationsL : int
        nVariables : int
        slp : HashMap<Integer, Vector<Double>>
        Uk : Vector<Double>
        SLP(int, int)
        calculaAlphasBetas(int, int)
        calculaParametros(int, int)
        calculaOutput(Vector<Double>, Vector<Boolean>, Vector<Double>)
        tamanioUi(int)
        calculaOperandoVariables(int, Vector<Double>, Vector<Double>)
        getSlp()
        getUk()
};
```

Figura 3. Especificación de la clase SLP

No existe posibilidad de crear un SLP Patrón vacío; es necesario especificar el número de variables que maneja el programa así como el número de operaciones no escalares para poder inicializar los campos de la clase. Dado que en este SLP 'plantilla' se van a sustituir todos los individuos para calcular su output y demás operaciones, es necesario limpiar los campos del SLP cada vez que un individuo deja de estar en el SLP.

La función *calculaParametros*, obviamente, calcula el número de parámetros totales que maneja la estructura SLP, dados el número de variables y el número de operaciones no escalares. Por su parte, *Tamanio_Ui,* devuelve el número de parámetros que tiene el U_i que le indiquemos.

Las funciones *calculaAlphasBetas* y *calculaOperandoVariables*, se utilizan para calcular, en un caso, el número de α y β que tiene cada instrucción del straight line program, y en otro caso, el valor de un operando de cada instrucción.

Por último, la función más importante de esta clase, *calculaOutput*, tiene como objetivo calcular el valor final del straight line program (individuo) apoyándose en los valores que tengan los distintos U_i*'s* calculados con el resto de operaciones de la case, para los valores de los diferentes puntos muestra que proporcione el usuario.

3.2.4 Clase Algoritmo

La clase Algoritmo es la **encargada** de realizar todo el proceso evolutivo para encontrar una buena solución al problema presentado. Si todas las anteriores clases están bien definidas y programadas, esta clase no debería suponer mucha dificultad, pues es, en esencia, la aplicación del pseudo-código de la programación genética visto en capítulos anteriores, con algún que otro añadido.

La clase algoritmo consta de un constructor vacío y de la clase *inicializar* que inicializa todas las variables implicadas en el algoritmo, como por ejemplo, el número de variables, el número de operaciones escalares, el tamaño de la población, de los individuos, la ruta de los puntos muestra, el número de iteraciones que se realizarán en el algoritmo, etc.

```
// Numero de variables
private int nVariables;

// Numero de operaciones no escalares (tamanio eficaz)
private int L;

// Tamanio de la poblacion
private int tamPoblacion;

// Tamanio del individuo
private int tamIndividuo;

// Rango minimo
private double lower;

// Rango maximo
private double upper;

// Ruta del fichero
private String ruta;

// Tipo de seleccion
private int selection;

// Tipo de cruce
private int cross;

// Tipo de mutacion
private int mutation;

// Tipo de reemplazamiento
private int replacement;

// Probabilidad de cruce
private double crossProb;
```

```
// Probabilidad de mutacion
private double mutationProb;

// Test de parada
private int test;

// Elitismo
private boolean elitism;

// Contador
private int contador = 0;

// K-Torneo
private int kTorneo;

// Variables que indicaran los individuos que han resultado elegidos en el
// proceso de seleccion
private int father1;
private int father2;

// Variable que indicara el individuo elegido para mutar
private int fatherM;

// Variable que contendra la probabilidad aleatoria de cruce y mutacion
private double probability;
```

La funcion *inicializar* recoge los valores que el usuario haya insertado en la interfaz del programa y los asocia a los distintos parámetros del algoritmo (que se han visto en el párrafo anterior).

La función *ejecutar* de esta clase lanza el proceso evolutivo y, cuando termina, se invoca a la clase *generaSalida*, la cual crea un archivo con los puntos aproximados para la función original que se han obtenido durante el proceso y que será utilizado para pintar las gráficas (obviamente esto sólo si se trabaja en 1 o 2 variables).

Cuando se termina el proceso evolutivo, se retorna el resultado final al hilo principal del programa, el cual se encargará de mostrarlo, a través de la interfaz, junto con las gráficas resultantes.

La siguiente figura muestra la especificación de la clase:

tfm.genetics.algoritmo.Algoritmo
{
nVariables : int
L : int
tamPoblacion : int
tamIndividuo : int
lower : double
upper : double
ruta : String
selection : int
cross : int
mutation : int
replacement : int
crossProb : double
mutationProb : double
test : int
elitism : boolean
contador : int
kTorneo : int
father1 : int
father2 : int
fatherM : int
probability : double
Algoritmo()
inicializar(PanelPrincipalBase)
ejecutar()
generaSalida(SLP, Individuo)
redimensiona(Poblacion, Poblacion)
};

Figura 4. Especificación de la clase Algoritmo

Aparte de estas clases, existen otras dos que no van a ser detalladas ya que no intervienen en el propio proceso evolutivo. Una es *tfm.genetics.panel.base.PanelPrincipalBase*, que es la encargada de crear la interfaz para que el usuario introduzca los datos necesarios para el proceso evolutivo (los cualves verifica), y la clase *tfm.genetics.principal.Principal* que es la clase encargada de arrancar el programa y el hilo principal. Además, hay también una clase estática *tfm.genetics.utilities.Utilidades* que sirve, entre otras cosas, para generar números aleatorios en un rango, o rellenar con valores 0.0 un vector recibido con un tamaño específico.

En el capítulo 5 se encuentra toda la programación completa de estas clases, junto con el programa principal, en el cual se le pide al usuario todos los valores anteriormente citados necesarios para la resolución del problema.

Como se puede ver, se ha intentado seguir los principios básicos de la programación genética, estableciendo una clase por cada parte importante de la misma, como son los individuos, la población, la estructura objetivo de este proyecto que es el SLP, y el algoritmo propiamente dicho.

3.3 El programa GNUPlot

Aparte de las clases anteriores, este proyecto utiliza un programa auxiliar denominado GNUPlot, y que se utiliza para dibujar las gráficas de las funciones aproximadas obtenidas junto con las originales. Cabe de decir que esto sólo es posible hacerlo, como se ha dicho antes, cuando el proceso maneja una o dos variables, puesto que no se soporta la representación de espacios de más variables.

Fue necesario el uso de este programa puesto que Java carece de posibilidades de representación gráfica en su definición estándar. Y, aunque existen un gran número de librerías gráficas tanto públicas como comerciales que permiten la salida gráfica, se utilizará este programa público como utilidad de representación gráfica, realizando el proceso en dos etapas: el programa imprime los resultados en fichero de datos de salida que posteriormente se lee con GNUPlot para representarlo gráficamente.

Se ha decidido así ya que GNUPlot es un poderoso programa *freeware* para hacer gráficas con datos en 2D y en 3D que es frecuentemente usado en ámbitos científicos. GNUPlot puede usarse en muchos ambientes computacionales, incluyendo Linux, IRIX, Solaris, Mac OS X, Windows y DOS. Requiere las mínimas capacidades gráficas y puede usarse aún en una terminal de tipo vt100, lo que muestra su gran posibilidad de ser ejecutado en diferentes entornos. Tiene una amplia variedad de opciones de salidas para que el usuario pueda usar las gráficas resultantes como lo desee, ya sea para ser visualizados o para incluirlos en sus propios documentos. Este curso está basado en la versión gnuplot 4.4.2., que está disponible para muchos tipos de sistemas operativos en la página oficial de GNUPlot *http://www.gnuplot.org*.

3.4 Equipo de Desarrollo

El algoritmo genético ha sido bajo las siguientes condiciones:

- Versión del sistema: Mac OS X Yosemite 10.10
- Versión del kernel: Darwin 14.0.0
- Entorno de desarrollo Eclipse Luna 4.4.1

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

Capítulo 4 : Experimentación y Resultados

Pablo Solar Rodríguez

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABLAS

Capítulo 4 : Experimentación y Resultados

Una vez explicado el objetivo de este proyecto y su método de resolución, es necesario conocer cómo se comporta y si los resultados que proporciona son adecuados y factibles para una aplicación práctica real. Este capítulo se centra en eso, en experimentar con el algoritmo evolutivo y varios puntos muestra de diferentes funciones (en este caso conocidas) y ver cómo difiere la aproximación obtenida de los datos originales.

Para ello, primero se escogerá una función tipo para calibrar el algoritmo y saber qué parámetros son los que pueden proporcionar **a priori** mejor resultados en las diferentes funciones a analizar. Se ha resaltado la expresión 'a priori', ya que, como se ha explicado durante toda esta memoria, **este es un algoritmo que se basa en su mayoría en la probabilidad y en el azar**, por lo que se puede tener una configuración que proporcione una solución muy buena a una función, y ese mismo ajuste de una solución poco factible para otra función, o incluso para una nueva ejecución de la misma función.

No obstante, esta configuración fija para todas las distintas funciones podrá proporcionar una visión general del comportamiento del algoritmo genético en la resolución del problema de la Regresión Simbólica (RS).

4.1 Calibrado del algoritmo

El calibrado del algoritmo consiste en obtener los valores de los parámetros involucrados en el mismo con el fin de conseguir el mejor resultado posible. Lógicamente, el calibrado se probará sobre unas pocas modificaciones de los parámetros, por lo que no se puede asegurar que el ajuste que se consigue sea el óptimo siempre.

Para realizarlo, se va a utilizar la siguiente función:

$$f(x) = x^4 + x^3 + x^2 + x$$

Los parámetros a calibrar son los siguientes:

- Tamaño de la población (TP) → 10, 50, 100, 150, 200
- Probabilidad de cruce (PC) → 0.1, 0.25, 0.50 0.75, 0.90
- Probabilidad de mutación (PM) → 0.005, 0.025, 0.05, 0.075, 0.1
- Tipo de selección (TS) → Ruleta o K-Torneo
- Número de generaciones (NG) → 10, 50, 100, 500, 1000

El método que se va a seguir para calibrar cada parámetro será el siguiente**: para cada posible valor del parámetro se realizarán 10 ejecuciones del algoritmo**, de las cuales se anotará el mejor resultado (menor fitness) y la media aritmética de esos diez valores. Es decir, para un tamaño de población 10, se van a realizar 10 ejecuciones y anotar lo antes descrito; para la población de 50 individuos, otras 10 ejecuciones, y así sucesivamente. Al final, el valor de ajuste que se le dará al parámetro será aquel que tenga una menor media aritmética de los errores.

Para la función *f (x)* se van a utilizar un total de 30 puntos muestra que están en el rango [-1.75, 1.75] y en el fichero 'p_calibrado.dat'. Esta función, en el rango [-5, 5], idealmente (sin tener en cuenta los puntos muestra) presenta la siguiente gráfica:

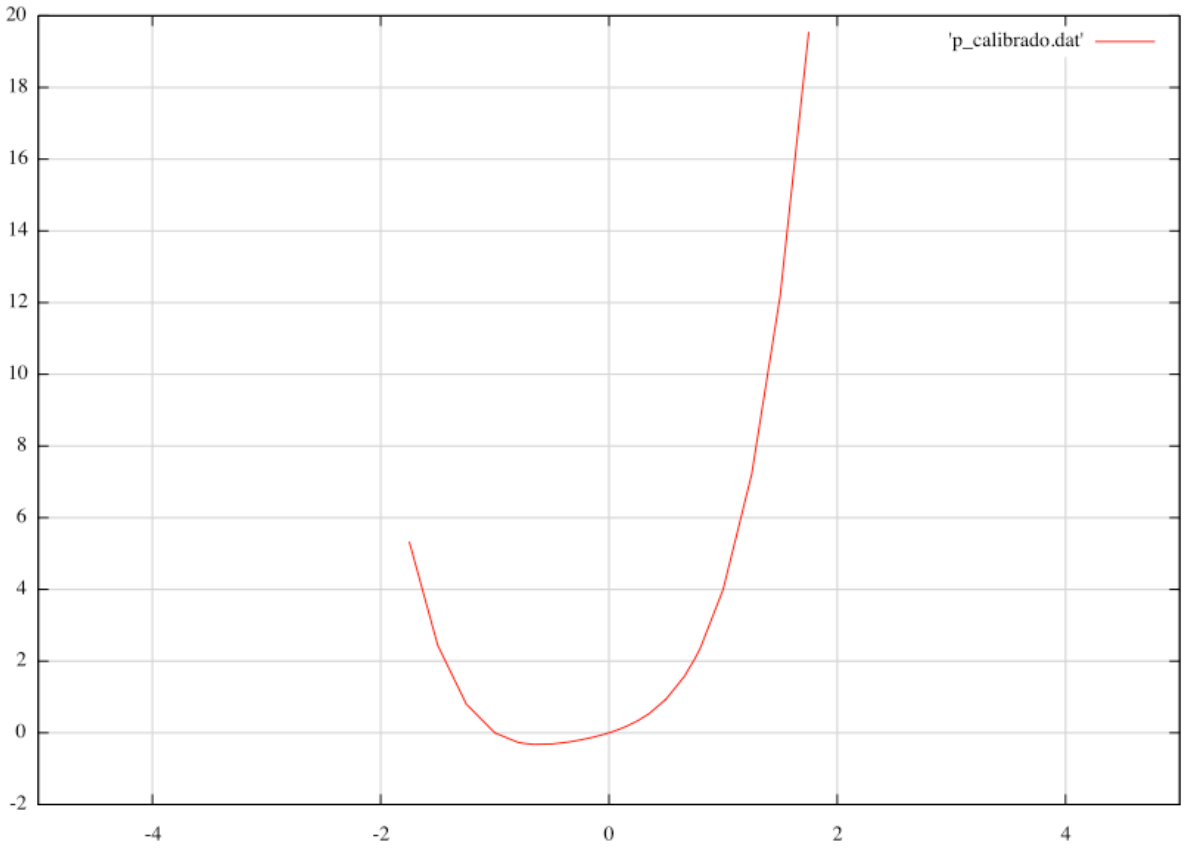


Figura 1. Gráfica de la función tipo para calibrado

El calibrado se hará de forma ordenada conforme a la lista anterior, empezando por el tamaño de la población y acabando por el número de iteraciones del algoritmo. Además, se van a fijar previo ajuste los parámetros con valores:

- Probabilidad de cruce → 0.75
- Probabilidad de mutación → 0.05
- Tipo de selección → K-Torneo (K = 2)
- Número de generaciones (test de parada) → 100

- Parámetro ∠ (nº operaciones no escalares) → 6 *
- Rango de generación aleatoria de parámetros → [-0.5, 0.75] *
- Tipo de cruce → En 1 punto *
- Tipo de mutación → Normal *
- Tipo de Reemplazamiento → Siempre Hijos *
- Tipo de algoritmo → Elitista *
- Fichero de puntos muestra → 'p_calibrado.dat' *

* (*Estos parámetros van a permanecer fijos durante todo el calibrado, es decir, no entran en juego en el ajuste)*

4.1.1 Calibrado del Tamaño de la Población

Los valores escogidos para ajustar el tamaño de la población van a ser 10, 50, 100 y 150. En teoría, a menor tamaño de población menor variedad en los individuos y por tanto peores resultados, lo que no quiere decir que a mayor número de individuos mejores soluciones, puesto que está demostrado que a veces un tamaño muy grande puede producir incluso peores resultados. Se toman, por tanto, valores típicos y comúnmente usados que arrojan resultados mostrados en la siguiente tabla:

TP	*10*	*50*	*100*	*150*	*200*
PC	0.75	0.75	0.75	0.75	0.75
PM	0.05	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	100	100	100	100	100
Mejor Fitness	7.507250	0.234134	0.140675	0.218291	0.532931
Media Fitness	11.903886	1.732168	0.879949	**0.490208**	0.750827
Elegido	Tamaño de Población = 150				

Tabla 1. Calibrado del tamaño de la población

Como se puede observar, no tiene por qué darse que a mayor tamaño mejores resultados. Normalmente, una población que oscila entre 100 y 200 individuos suele ser suficiente para que un problema resuelto por algoritmos genéticos produzca soluciones bastante válidas, aunque todo depende de cómo estén codificados los mismos y de la estructura que puedan tener.

La siguiente gráfica muestra el resultado de aplicar la RS mediante un proceso evolutivo a la función *f (x)* previamente definida para los parámetros fijados y el nuevo tamaño de la población ya calibrado:

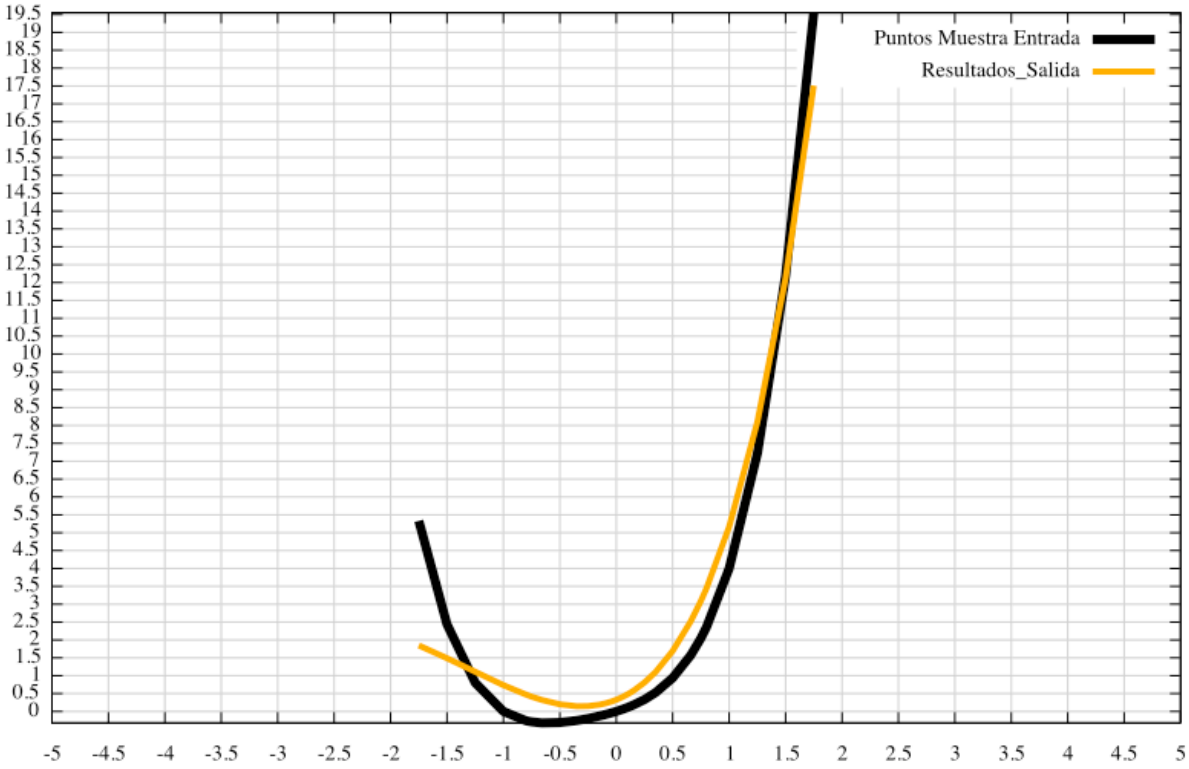


Figura 2. Gráfica con tamaño de población ya calibrado a valor 150

4.1.2 Calibrado de la Probabilidad de Cruce

Para la probabilidad de cruce se manejan como posibles valores 0.1, 0.25, 0.50, 0.75 y 0.90. Al igual que en una población de individuos del mundo real, si éstos no se reproducen con frecuencia, ésta tiende ser menos variada y escasa. Lo mismo pasa en la programación genética; una pequeña probabilidad de cruce no hace evolucionar a los individuos para que los nuevos hijos sean mejores que sus ancestros, por lo que la población queda estancada. Por el contrario, una alta probabilidad de cruce asegura más variedad y más posibilidad de manejar mejores resultados. Los resultados obtenidos en el calibrado del cruce son:

TP	150	150	150	150	150
PC	*0.1*	*0.25*	*0.50*	*0.75*	*0.90*
PM	0.05	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	100	100	100	100	100
Mejor Fitness	1.010940	1.303380	0.042005	0.122550	0.147942
Media Fitness	7.573338	1.780990	0.935815	0.720766	**0.411511**
Elegido	Probabilidad de Cruce = 0.90				

Tabla 2. Calibrado de la Probabilidad de Cruce

Como era de esperar, cuanta más probabilidad hay de que se crucen los individuos de la población, más variedad hay en el conjuntos de posibles soluciones y por tanto el abanico de soluciones es mejor. Fijándose en la tabla, se observa que para otras probabilidades el mejor fitness individual es mejor como en el caso de la probabilidad = 0.5; como ya se ha dicho, no significa que esa probabilidad sea mejor, ya que pudo arrojar resultados mucho más elevados que hacen que la media aumente. Nótese además que el resultado final es mejor que en la Tabla 1.

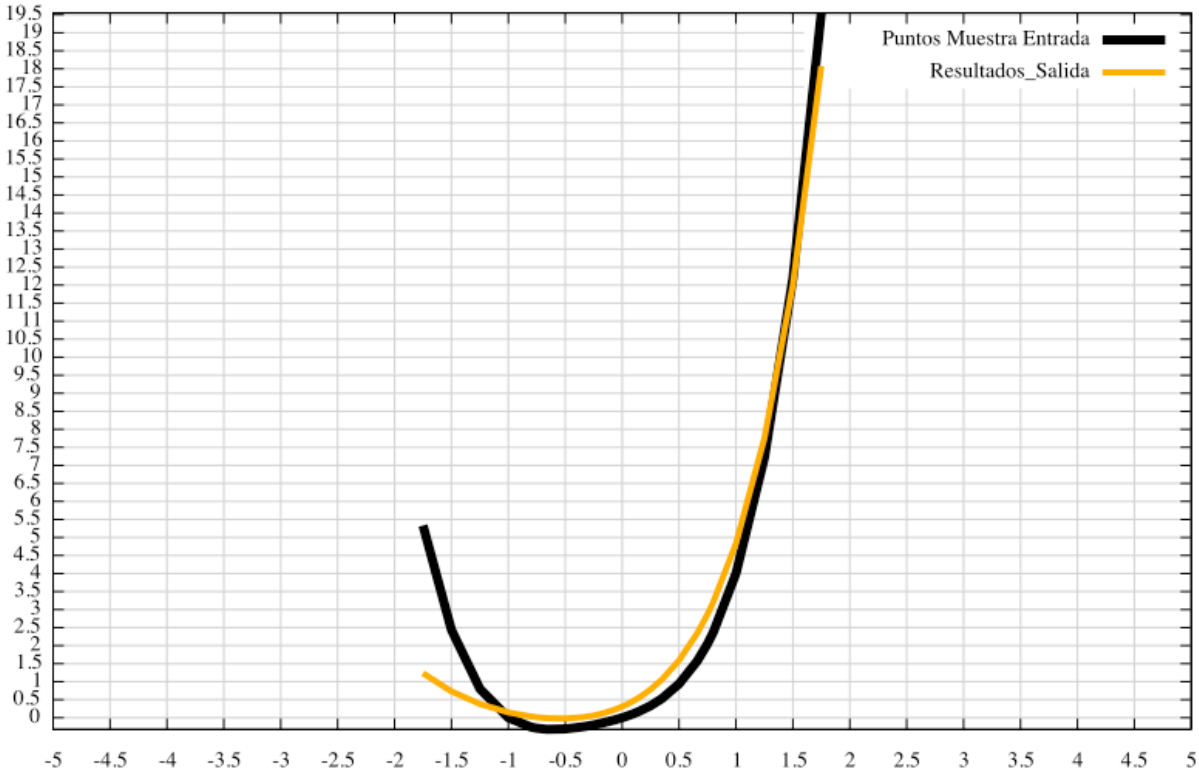


Figura 3. Gráfica con probabilidad de cruce ya calibrada a valor 0.90

4.1.3 Calibrado de la Probabilidad de Mutación

Los valores a estudiar en la probabilidad de cruce son 0.005, 0.025, 0.05, 0.075, 0.1. Dado que son valores muy pequeños, y, además, sólo se modifica (muta) un individuo de entre toda la población en el caso de que el número generado aleatoriamente sea menor o igual que éstos valores, los resultados que se obtienen son muy similares a los de la Tabla 2. Véanse en la siguiente tabla:

TP	150	150	150	150	150
PC	0.90	0.90	0.90	0.90	0.90
PM	*0.005*	*0.025*	*0.05*	*0.075*	*0.1*
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	100	100	100	100	100
Mejor Fitness	0.034150	0.203626	0.0154474	0.144509	0.147942
Media Fitness	0.7332734	0.636640	**0.559955**	0.741672	0.561507
Elegido	Probabilidad de Mutación = 0.05				

Tabla 3. Calibrado de la Probabilidad de Mutación

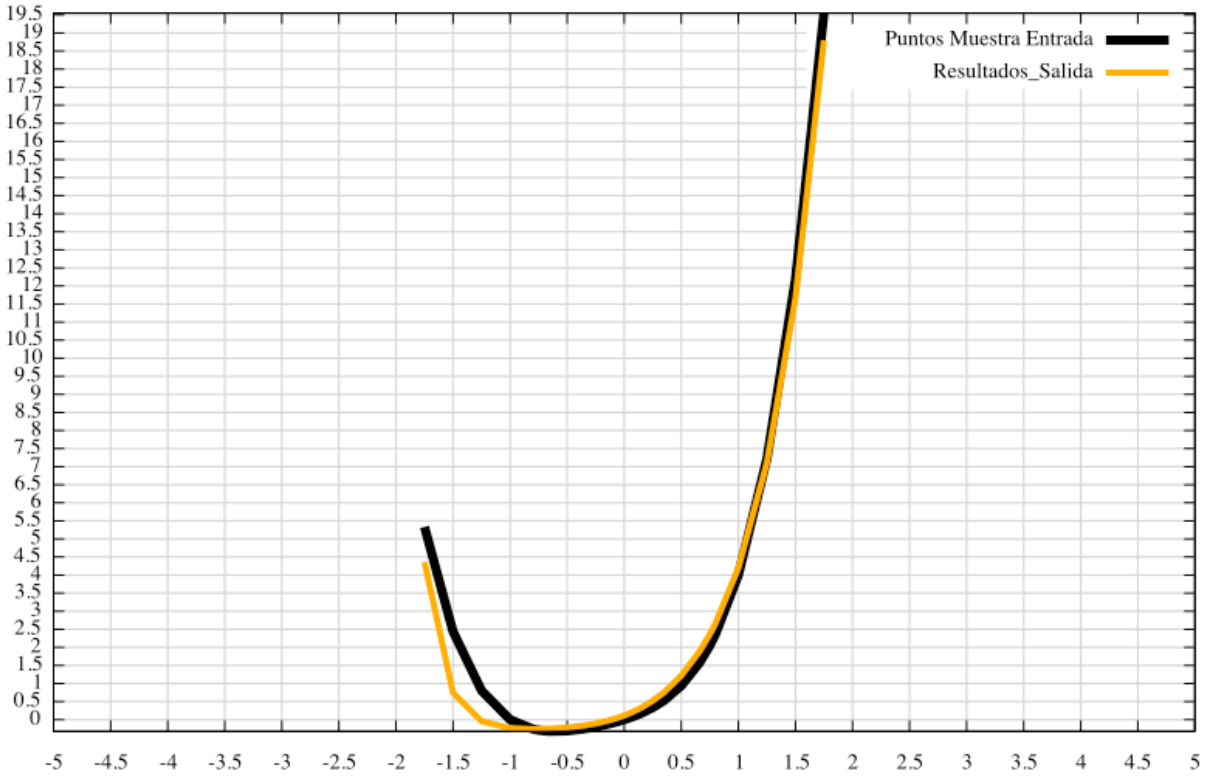


Figura 4. Gráfica con probabilidad de mutación ya calibrada a valor 0.05

4.1.4 Calibrado del Tipo de Selección

Dado que en este proyecto sólo se trabaja con dos tipos de selección, la Ruleta y el K-Torneo, el modo de ajuste del tipo de selección se realizará de distinta manera a cómo se ha hecho en los anteriores apartados; para cada tipo, se va ejecutar un total de 25 veces el algoritmo genético.

La teoría señala que el K-Torneo suele comportarse mejor que el método de la Ruleta, aunque como siempre, todo dependerá de la función a aproximar y de los valores que se generen. El K-Torneo utilizará de parámetro $k = 2$, que es el valor que se suele usar de manera genérica, ya que valores altos hacen que los individuos tiendan a converger a un óptimo local de manera prematura.

Cabe decir que se va a determinar uno de los dos métodos con un número de iteraciones = 100, que pueden ser quizá demasiado pocas para dilucidar el verdadero potencial de cada método, pero este proyecto pretende ser, como se ha dicho, un entrenamiento de cara a posibles ampliaciones para una aplicación más seria y real.

Tras ejecutar el programa el número de veces ya comentado, se ha obtenido:

TP	150	150
PC	0.90	0.90
PM	0.05	0.05
TS	*K-Torneo (k =2)*	*Ruleta*
NG	100	100
Mejor Fitness	0.0283498	1.02017
Media Fitness	**0.783032**	1.899377
Elegido	Tipo de Selección = K-Torneo (k = 2)	

Tabla 4. Calibrado del Tipo de Selección

Como se puede observar, el error cometido por el método de K-Torneo es sensiblemente inferior al cometido por el de la Ruleta. Se cumplen, por tanto, los estudios teóricos realizados al respecto, que, como ya se ha indicado anteriormente, señalaban al K-Torneo como técnica con más calidad y mejores resultados. Puede que un número de generaciones mucho mayor que 100, los dos métodos se aproximen en cuanto a resultados, ya que Ruleta tiende a evolucionar y K-Torneo a converger hacia una buena solución, pero es algo que no se estudiará aquí.

4.1.5 Calibrado del Número de Generaciones (iteraciones)

El ajuste óptimo del número de generaciones, iteraciones o test de parada, es casi imposible de definir, ya que un algoritmo genético podría repetirse un número enorme de veces si el computador tuviera potencial suficiente. Por eso, aquí se van a utilizar una serie de valores (que no tienen por qué ser los utilizados normalmente) para ver las diferencias que puede haber entre iterar pocas o muchas veces.

Es lógico que si el proceso se somete a pocas repeticiones, éste no tiene manera de evolucionar, por lo que debería de haber resultados peores. Sin embargo, al igual que sucede con el tamaño de la población no se puede asegurar que los resultados sean mejores cuantas más iteraciones se realicen, ya que, por ejemplo, los individuos podrían mutar hacia valores con menor calidad que proporcionasen peores resultados.

Para calibrar la función *f (x)* se van a utilizar como test de parada los valores 10, 50, 100, 500, y 1000, siendo éste último un valor suficientemente grande como para que el ordenador en el que se experimenta presente ya dificultades de cómputo. Con los parámetros anteriores calibrados, los datos que proporciona la experimentación son:

TP	150	150	150	150	150
PC	0.90	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	*10*	*50*	*100*	*500*	*1000*
Mejor Fitness	2.10159	0.373274	0.189895	0.002159	0.212883
Media Fitness	4.227974	0.914185	0.508957	**0.367080**	0.724921
Elegido	Número de Generaciones = 500				

Tabla 5. Calibrado del Número de Generaciones

A la vista de los resultados, se puede observar que efectivamente un número de iteraciones grande no asegura el mejor de los resultados. Además, se puede sacar como conclusión que un número de iteraciones comprendido entre, más o menos, 150 y 500 (quizá el mejor rango), puede ser suficiente como para hacer evolucionar el proceso evolutivo de forma que las soluciones que proporcione sean válidas para la resolución del problema.

Puede ser que esto sea así para estos problemas “sencillos”, pero que para los más complejos no y lo mejor fuera un número mucho mas elevado de generaciones. De todas formas, es algo que se escapa al ámbito de estudio de este proyecto.

Nótese además (véase la tabla anterior) que en este último calibrado se ha producido el mejor resultado (mejor media de fitness) de entre todos los anteriores. Lógico, ya que se supone que se han ajustado todos los parámetros de forma que proporcionen la mejor solución posible.

Con el algoritmo ya calibrado, se ha vuelto a ejecutar el algoritmo para ver qué resultado produce, obteniendo un error fitness = 0.001974, que, casualmente, ha sido el mejor valor obtenido hasta este punto de la experimentación. La gráfica con la aproximación a los puntos muestra es la siguiente:

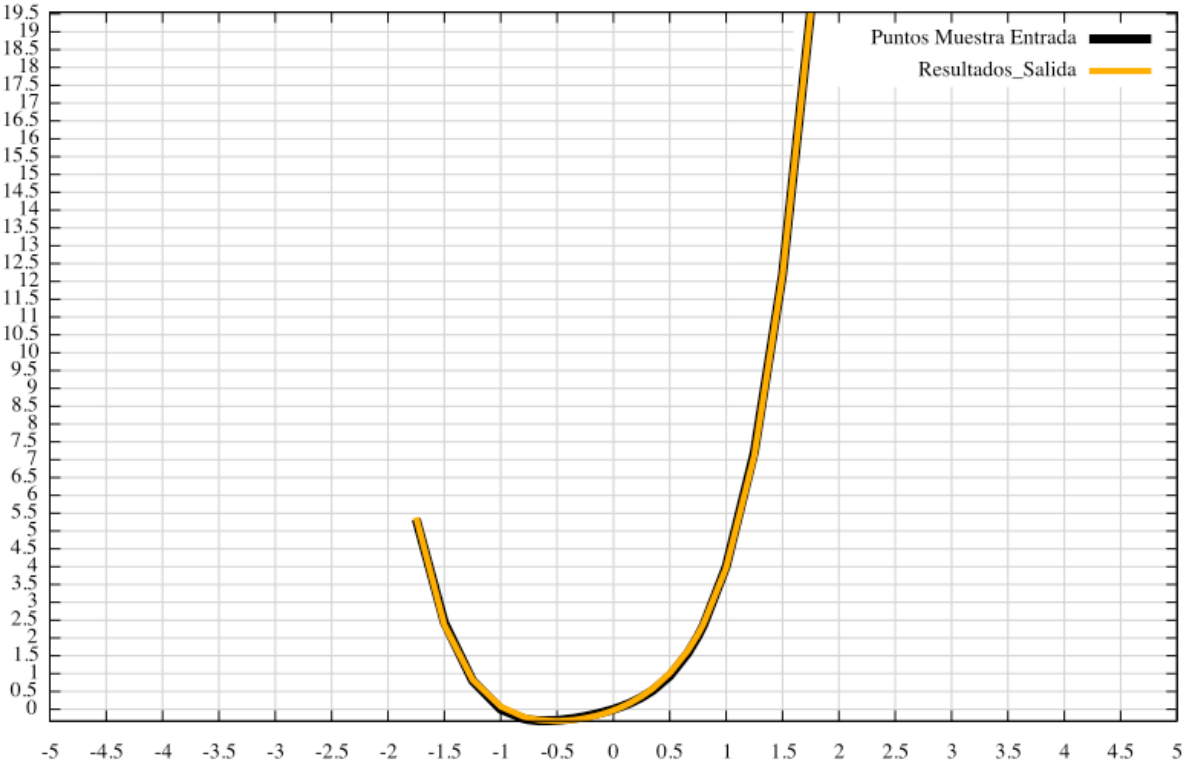


Figura 5. Gráfica con los resultados del algoritmo calibrado

Se comprueba, por tanto, que el calibrado del algoritmo parece ser bastante bueno, ya que la aproximación es casi perfecta. Es posible que éste ajuste sea de los mejores para la función que se ha utilizado, lo que no implica que lo sea para el resto de las cuales se quiera obtener una aproximación. El problema es que es inviable establecer un calibrado para cada una de las funciones, por lo que es necesario establecer un "esquema" general a seguir para todas aquellas que se estudien, y éste calibrado puede ser suficiente para proporcionar soluciones factibles.

En el resto del capítulo se van a presentar una serie de funciones de 1 y 2 variables (el algoritmo soporta más variables, pero sólo estás 2 proporcionan una gráfica asociada a la solución), y sobre ellas se va a ejecutar el proceso evolutivo utilizando el calibrado anterior. Además, se proporcionarán tablas y gráficos con los resultados obtenidos para cada función.

4.2 Aproximación de Funciones

Para comprobar hasta qué punto el calibrado anterior del algoritmo resulta válido y fiable en una utilización más general para realizar la RS, se van a presentar una serie de funciones escogidas (funciones objetivo) y se van a tratar utilizando el proceso evolutivo para ver si los resultados pueden ser factibles o no.

Éstas funciones objetivo son las siguientes:

$$1. \rightarrow f(x) = x^3 + x^2 + \frac{x}{2}$$

$$2. \rightarrow f(x) = \cos(x) + \frac{x}{2}$$

$$3. \rightarrow f(x) = \frac{\sin(x)}{\sqrt{\frac{1}{x^2}}}$$

$$4. \rightarrow f(x, y) = \left(\left(x * y\right) + y\right) * \left(x - y\right)$$

$$5. \rightarrow f(x, y) = \sin(x) + \cos(y) + \left(0.5 * (x + y)\right)$$

$$6. \rightarrow f(x, y) = \frac{\cos(\pi\, x)}{2\, y}$$

Se van a analizar, por tanto, un total de 6 funciones, 3 de ellas de una variable y las otras 3 en dos variables. El motivo por el que no se realizan en más variables, es porque no es posible representar gráficamente, en este proyecto, funciones de más de dos variables, y se considera que un apoyo gráfico clarifica los resultados. No obstante, al final de capítulo se pondrán los resultados que producen unas pocas funciones de más variables para que se vea que el algoritmo funciona perfectamente igual.

Para cada una de éstas funciones objetivo, se va a aplicar el algoritmo calibrado, y además, se va a comprobar cuál de los cuatro tipos de cruce es mejor en la función, ejecutándose un total de 5 veces para cada cruce (en 1 punto, 1 punto selectivo, uniforme y uniforme selectivo) y, como en el capítulo anterior, anotando el mejor resultado arrojado así como la media aritmética.

Salvo el cruce, el resto de parámetros que permanecían fijos en el anterior apartado del capítulo lo van a seguir estando y no se van a someter a ningún tipo de modificación.

4.2.1 Aproximación de (1) $f(x) = x^3 + x^2 + \frac{x}{2}$

Los puntos muestra de esta función se encuentran en el archivo 'p11.dat'. Una vez aplicado el proceso evolutivo a esta función realizando los pasos descritos en el punto anterior, se obtiene:

TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.013609	0.0140987	0.00458213	0.0198195
Media Fitness	0.0459006	0.0487905	**0.0184006**	0.031447
Elegido	Mejor Cruce = Uniforme			

Tabla 6. Resultados de la Función 1 aplicando el algoritmo

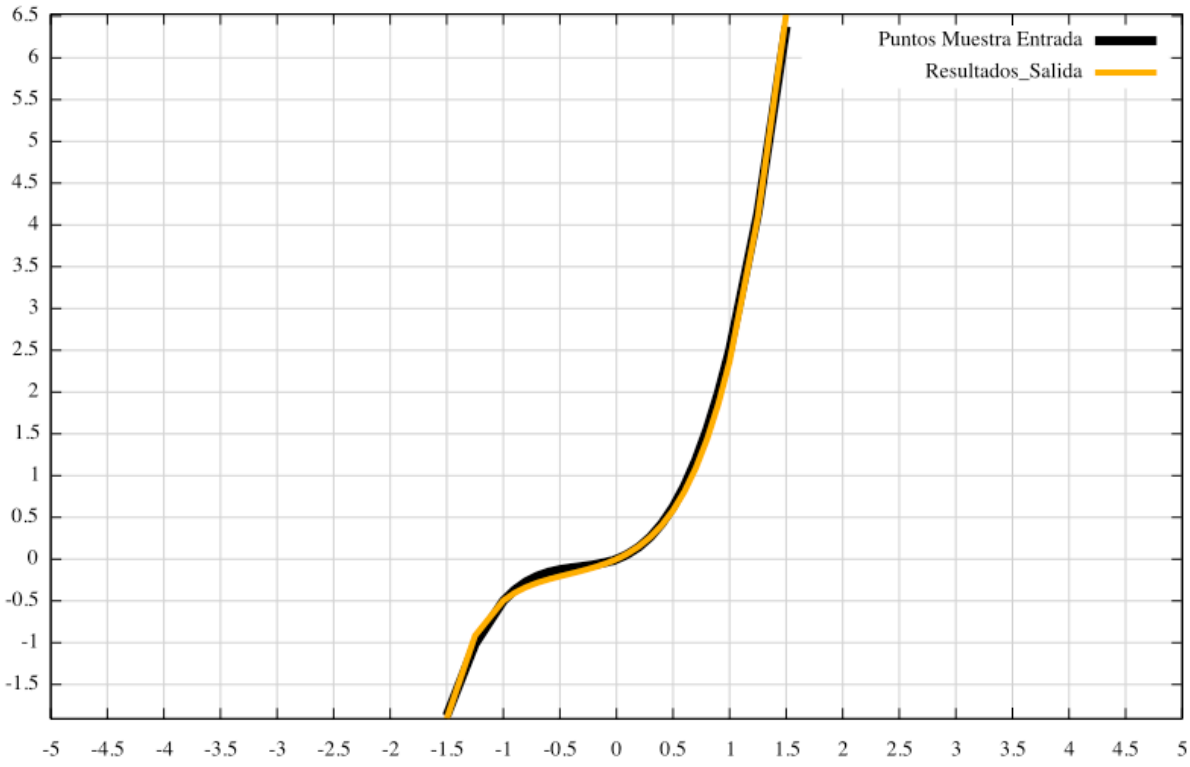


Figura 6. Gráfica correspondiente a la Función 1 con el mejor resultado de la tabla 6

Con los resultados de la tabla, y observando la gráfica (que se corresponde con el mejor valor fitness individual de la tabla, se puede deducir que, para esta función, los cruces uniformes producen mejores resultados que los cruces en 1 punto. Además, proporcionan soluciones mejores los cruces que no son selectivos.

Como la solución se ve de manera más clara mirando la gráfica, se comprueba que la regresión que ha calculado el algoritmo genético es excelente.

4.2.2 Aproximación de (2) $f(x) = \cos(x) + \frac{x}{2}$

Los puntos muestra de esta función se encuentran en el archivo 'p12.dat'. Tras ejecutar el algoritmo genético con dichos datos, se consiguen los siguientes resultados:

TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.157325	0.148309	0.115693	0.141774
Media Fitness	0.189415	0.199042	**0.1673624**	0.192716
Elegido	Mejor Cruce = Uniforme			

Tabla 7. Resultados de la Función 2 aplicando el algoritmo

De nuevo se vuelven a obtener resultados en los que el cruce uniforme sale ganador, aunque esta vez todos los métodos ofrecen una solución bastante similar. Eso sí, siguen siendo mejores los cruces normales frente a los selectivos.

Nótese que, al ser una función bastante más compleja que la anterior, el error que se comete es mayor aunque la solución sigue siendo factible.

Una representación gráfica del mejor resultado se observa en la siguiente tabla:

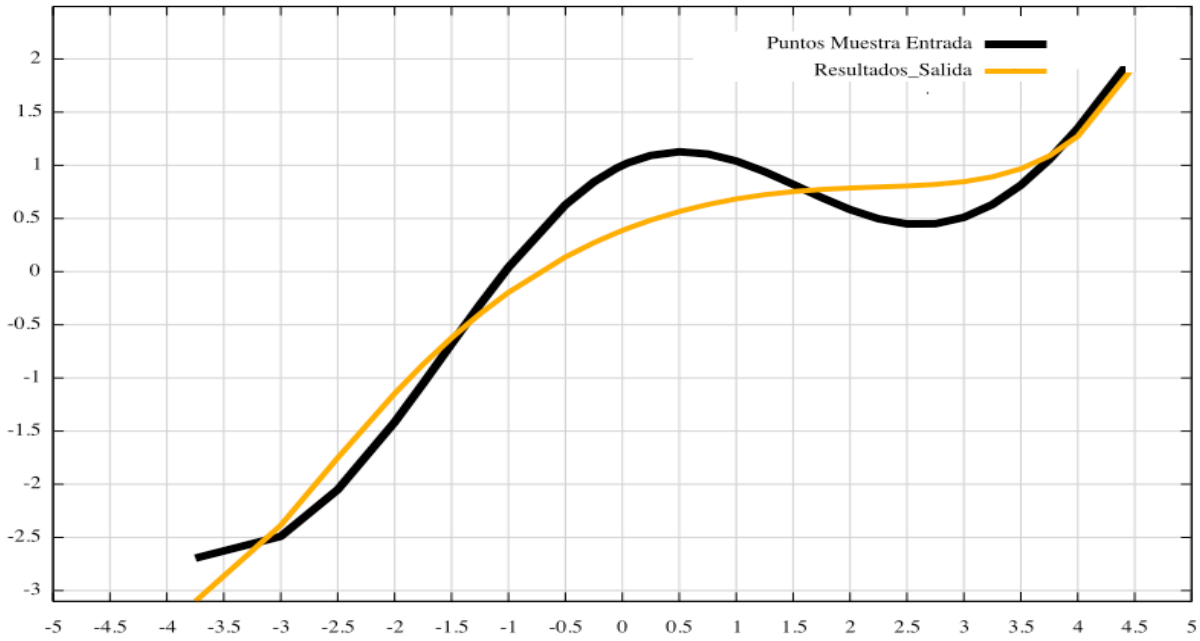


Figura 7. Gráfica correspondiente a la Función 2 con el mejor resultado de la tabla 7

4.2.3 Aproximación de (3) $f(x) = \dfrac{\sin(x)}{\sqrt{\frac{1}{x^2}}}$

Los datos iniciales de esta función se localizan en el fichero 'p13.dat'. Después de tratarlos con el algoritmo evolutivo, se obtienen los siguientes datos para analizar:

TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.045494	0.055327	0.0328402	0.0479082
Media Fitness	**0.066753**	0.0817076	0.084424	0.837355
Elegido	Mejor Cruce = En 1 Punto			

Tabla 8. Resultados de la Función 3 aplicando el algoritmo

En esta ocasión el mejor resultado se consigue con el cruce en 1 punto, siendo sensiblemente menor al resto de métodos, los cuales tienen resultados bastante similares. Esa es la razón como por la que no se puede sacar una conclusión acerca de qué cruce es el ideal para esta función, ya que todos parecen ser bastante válidos. Pero, si hubiera que decantarse, quizá sean algo más eficaces los que se aplican en 1 punto.

En la siguiente gráfica, que se corresponde con un resultado obtenido para ésta función, quizá se vea mejor que en todas las anteriores la verdadera filosofía de la regresión: la función original presenta bastantes curvas (y eso que se ha acotado el intervalo de representación, que si no serían muchas más), mientras que la aproximación obtenida se representa con una única curva a la vez que se aproxima a todos los puntos con un error pequeño que hace que la solución sea muy válida. La gráfica es la siguiente:

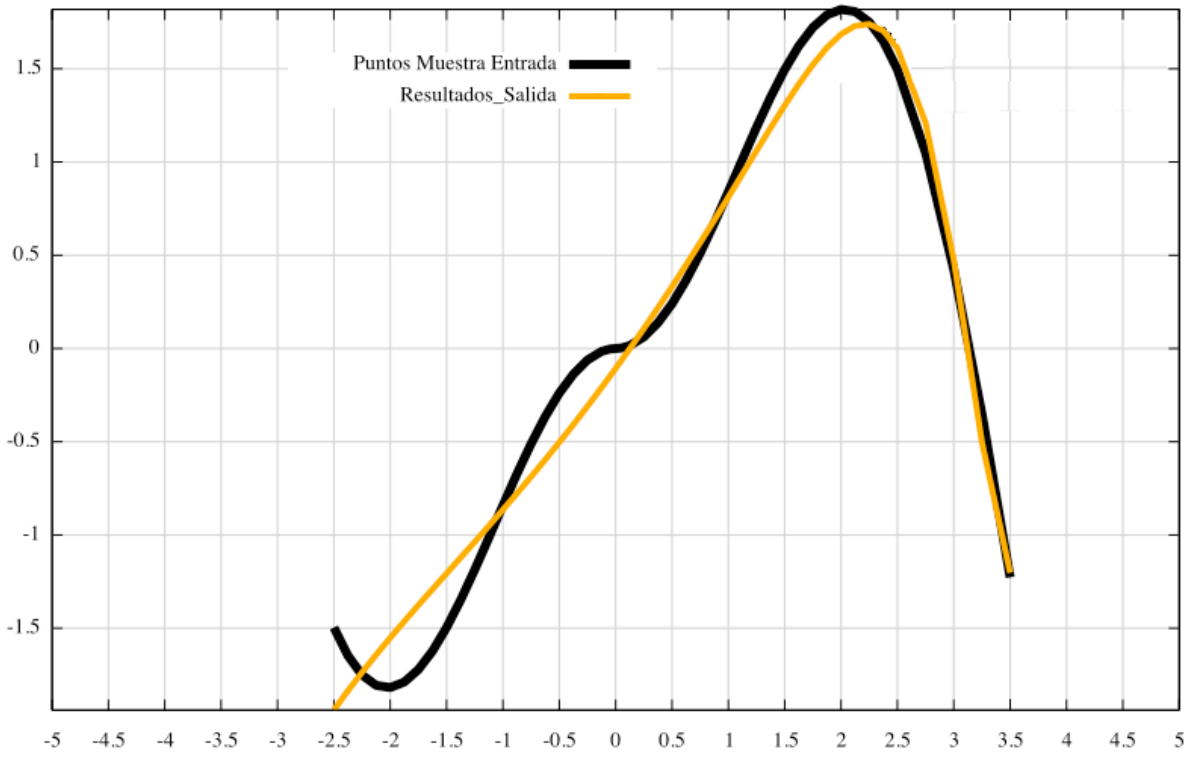


Figura 8. Gráfica correspondiente a la Función 3 con el mejor resultado de la tabla 8

Una vez presentadas y estudiadas las funciones de una sola variable, se puede decir como conclusión que el calibrado realizado en el algoritmo genético resulto ser satisfactorio, puesto que los resultados obtenidos en cada una de ellas han sido bastante válidos.

Si hubiera que decidir, además, qué método de cruce habría que utilizar para analizar otras posibles funciones, el estudio anterior indica que los cruces selectivos, puesto que parece ser que se comportan mejor que sus contrarios, y dentro de ellos, quizá el uniforme, aunque ambos tienen resultados parecidos y todos factibles.

4.2.4 Aproximación de (4) $f(x, y) = \big((x * y) + y\big) * \big(x - y\big)$

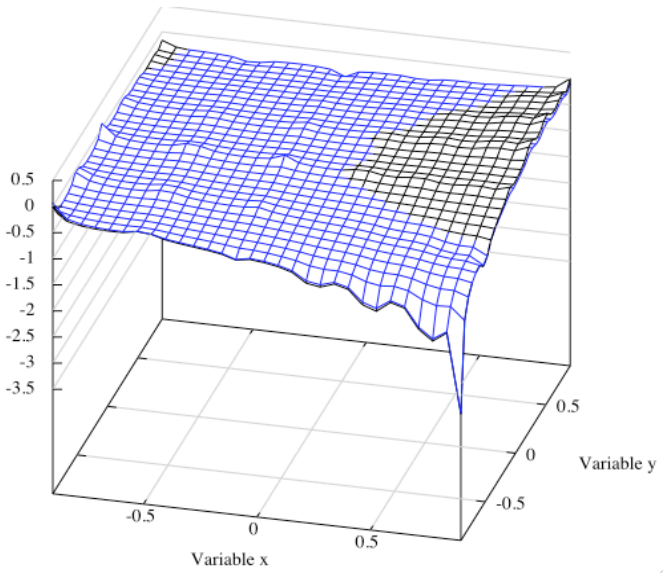
A partir de ahora, se van a analizar funciones en dos variables, algo que resulta bastante más complejo que para una sola, que es lo más habitual. Al trabajar con dos, **puede** que el error cometido aumente, ya que hay que considerar el error se comete para la *'x'* y para la *'y'*.

Los puntos de inicio para esta función se encuentran en el archivo 'p21.dat', y, después de aplicar el algoritmo calibrado indicando que se trabaja en 2 variables, se obtienen los siguientes datos:

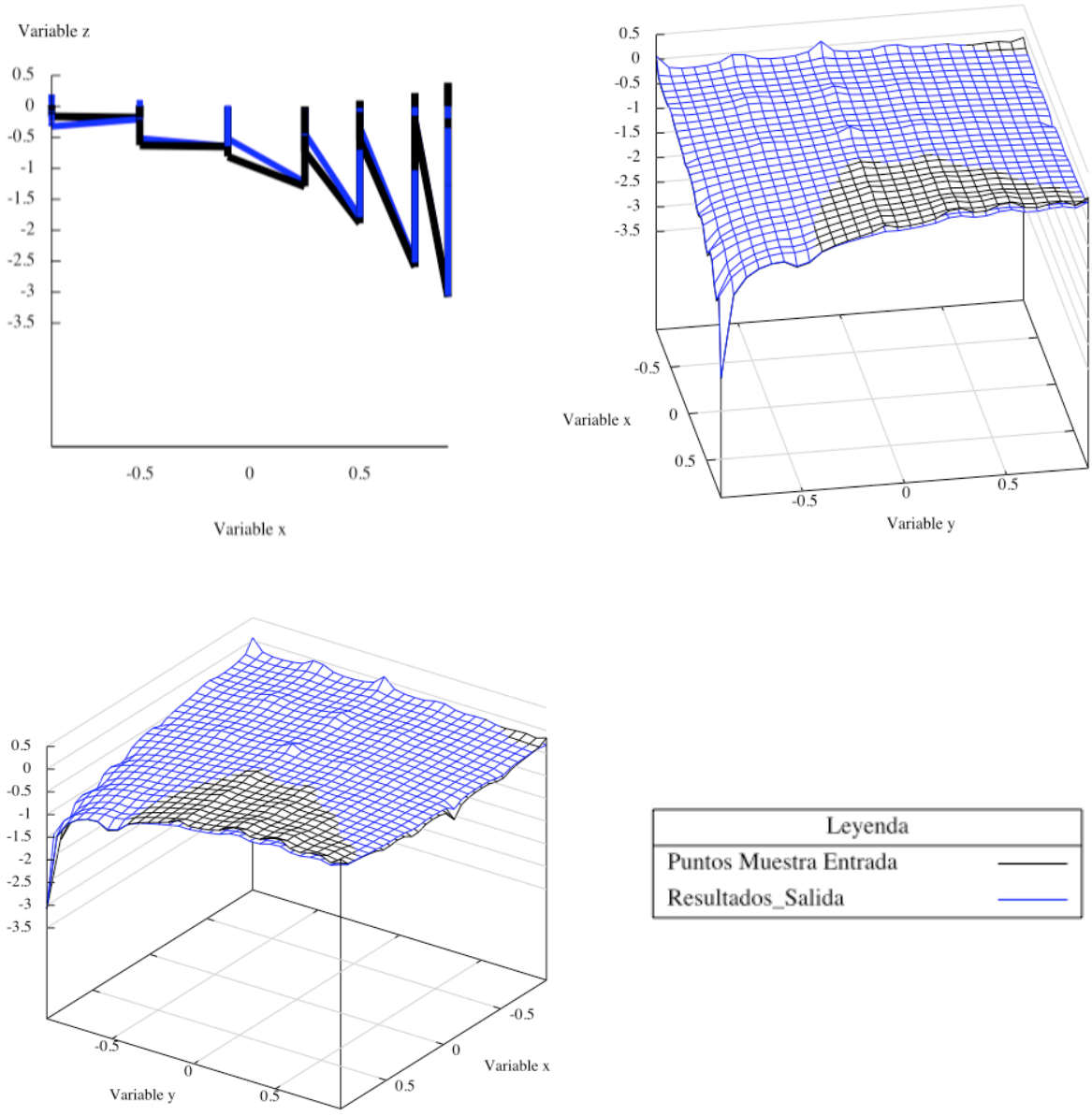
TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.025944	0.052315	0.015752	0.021200
Media Fitness	0.082704	0.139345	**0.023953**	0.051530
Elegido	Mejor Cruce = Uniforme			

Tabla 9. Resultados de la Función 4 aplicando el algoritmo

Los resultados son muy satisfactorios debido a que el error no ha aumentado casi nada y la aproximación, por tanto, es realmente buena para trabajar en un sistema de dos variables. Por otro lado, vuelve a salir ganador el cruce uniforme frente a los selectivos y, esta vez, por una sensible diferencia . Además, vuelve a cumplirse que resultan más fiables y eficaces los métodos que no son selectivos.



Las siguientes cuatro gráficas corresponden al mejor resultado obtenido para esta función, en diferentes perspectivas:



Figuras 9-12. Gráficas correspondientes a la Función 4 con el mejor resultado de la tabla 9

4.2.5 Aproximación de (5) $f(x, y) = \sin(x) + \cos(y) + \left(0.5 * (x + y)\right)$

Los puntos muestra de esta función se encuentran en el fichero fuente 'p22.dat'. Se trata de unos puntos más grandes en número que para el resto de funciones anteriores, porque eran necesarios para una correcta representación y aproximación de la función original.

Una vez ejecutado el algoritmo para estos datos, se obtiene:

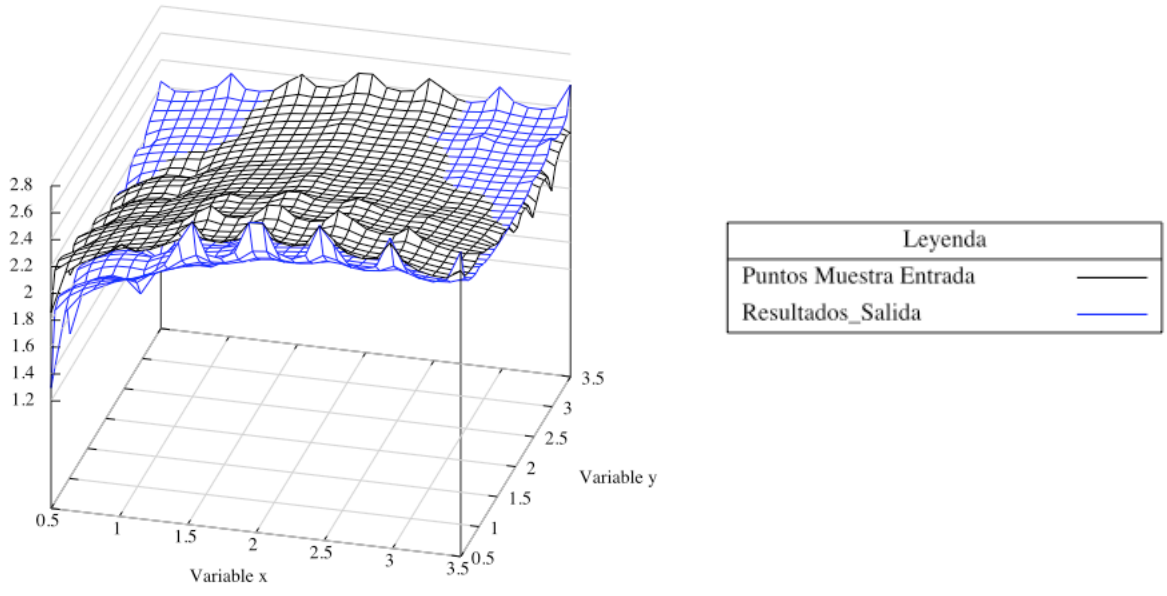
TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.027197	0.052335	0.065078	0.095574
Media Fitness	0.147901	0.122005	**0.120095**	0.175353
Elegido	Mejor Cruce = Uniforme			

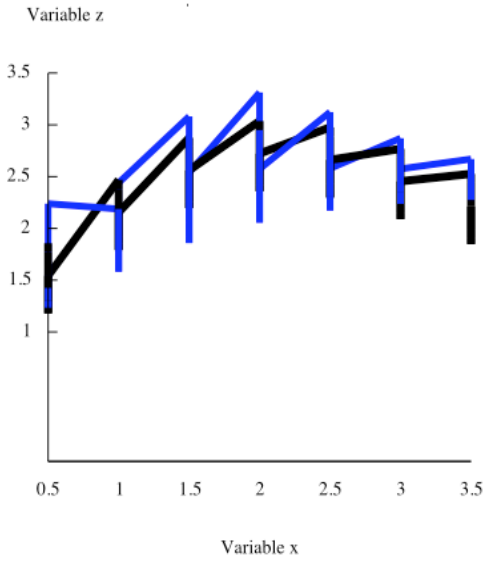
Tabla 10. Resultados de la Función 5 aplicando el algoritmo

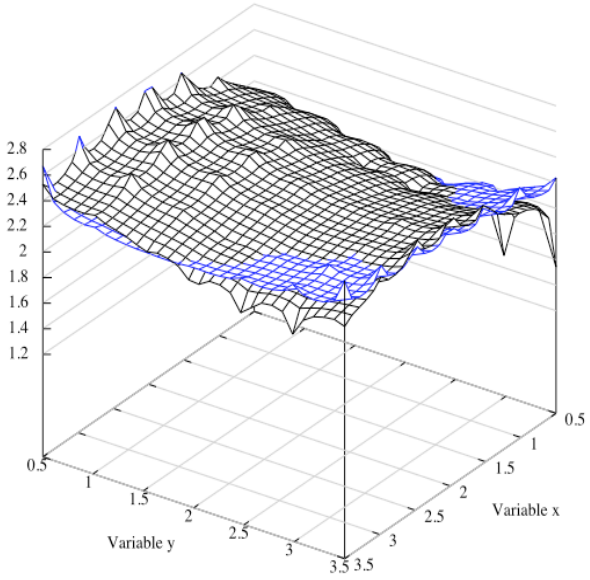
Otra vez, vuelve a obtener mejores resultados el cruce uniforme, aunque poco más se puede sacar en conclusión de los resultados obtenidos para ésta función. Lo único, que en esta ocasión, tanto el cruce uniforme como el de en 1 punto selectivo presentan resultados muy similares, y con bastante diferencia respecto al resto.

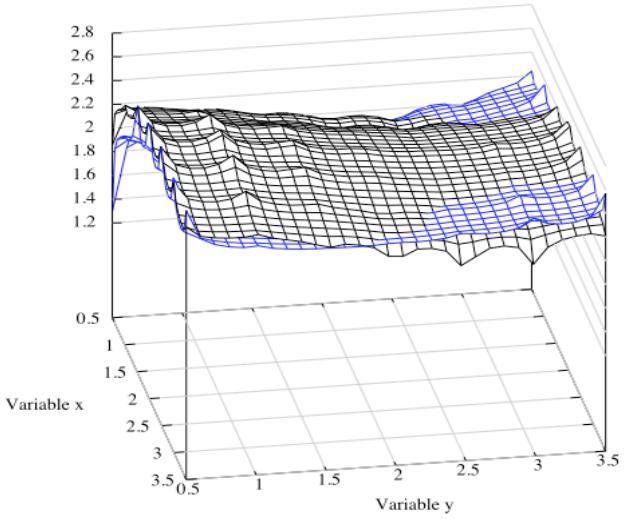
Cabe decir que, al ser una función más compleja que la anterior, el error aumenta de forma notable en comparación al resultado obtenido para la función 4, aunque la solución obtenida sigue siendo factible.

Una representación de una buena aproximación de la función original se muestra a continuación:









Figuras 13-16. Gráficas correspondientes a la Función 5 con el mejor resultado de la tabla 10

4.2.6 Aproximación de (6) $f(x, y) = \dfrac{\cos(\pi x)}{2y}$

Esta última función a analizar, puede que sea a priori (habrá que ver resultados) la más compleja de las vistas en dos variables, puesto que sólo utiliza multiplicaciones y divisiones, junto con funciones trigonométricas y constantes.

Los puntos iniciales se encuentran en el archivo 'p23.dat', que tiene una extensión similar al anterior y unos puntos que han sido elegidos para que la representación gráfica sea clara *(y ≠ 0)*, ya que en un intervalo de puntos mayor la función toma un aspecto un tanto difícil de plasmar en una gráfica. Una vez aplicado el proceso evolutivo, se obtienen los siguientes resultados:

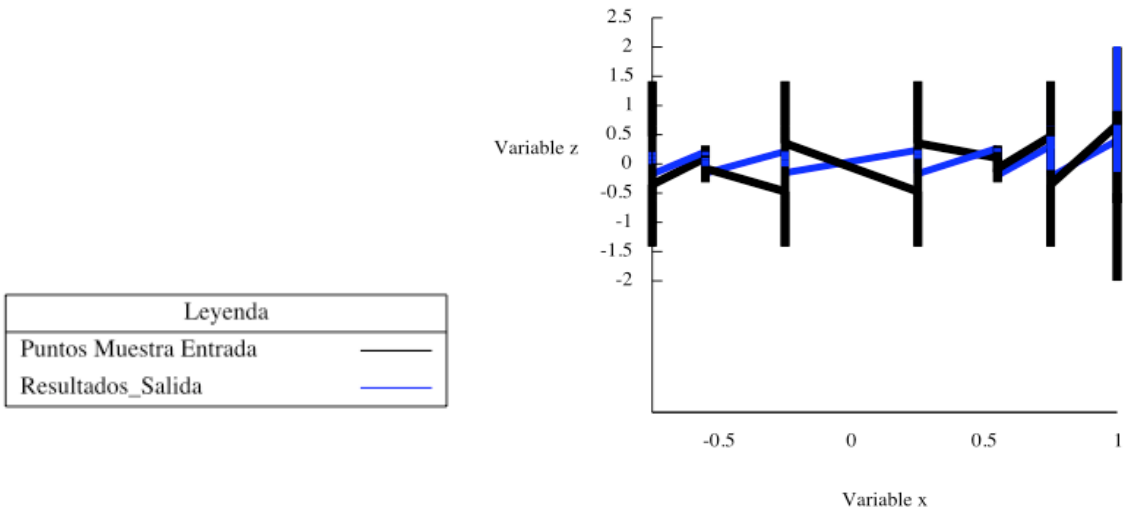
TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.393022	0.373548	0.394070	0.401664
Media Fitness	0.453654	**0.446521**	0.476787	0.473989
Elegido	Mejor Cruce = 1 punto (selectivo)			

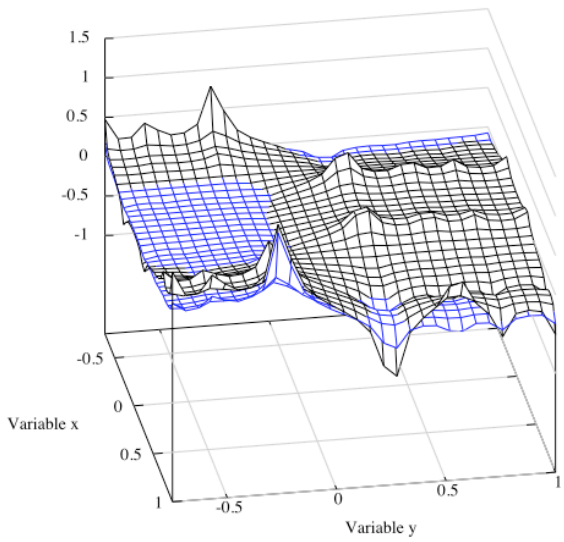
Tabla 11. Resultados de la Función 6 aplicando el algoritmo

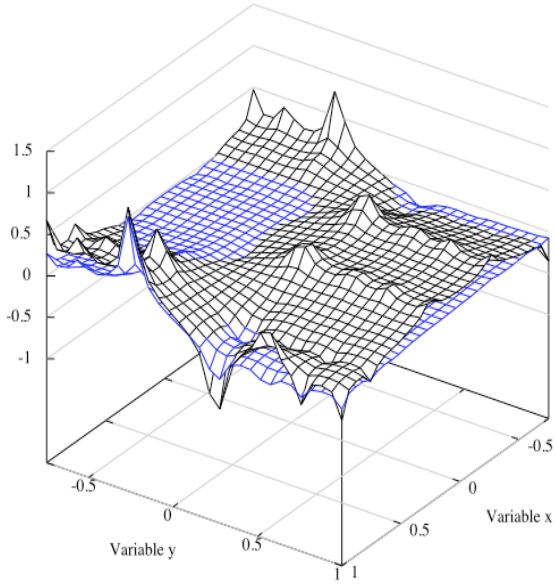
Esta vez, curiosamente, se ha obtenido el mejor resultado en el método del cruce en 1 punto selectivo, cuando en todos los anteriores siempre era de los peores. Es cierto que, los resultados que se han conseguido en esta ocasión son muy similares, pero deja patente que no se puede establecer una conclusión final acerca de qué técnica utilizar. Todo dependerá de la función a aproximar. Eso sí, en todos se consiguen soluciones factibles, puesto que en esta última función obtenemos un error pequeño, de 0.xxxx.

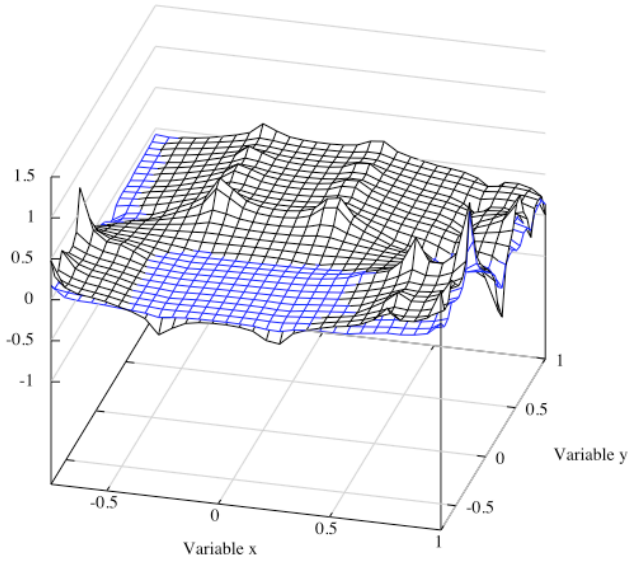
Nótese, además, que el error ahora ha aumentado considerablemente, pero es algo lógico, ya que se ha dicho que la función era más compleja que las anteriores. En esta función, trabajan mejor los cruces en 1 punto, que los uniformes, con una diferencia no muy grande, pero, al fin y al cabo, se busca el menor error cometido posible.

La representación gráfica asociada a esta función, de la cual no se sabe decir si la aproximación ha sido buena o no, se muestra a continuación:









Figuras 17-20. Gráficas correspondientes a la Función 6 con el mejor resultado de la tabla 11

Una vez analizadas estas funciones objetivo, se puede sacar en conclusión que el algoritmo genético funciona bien, proporcionando unas soluciones factibles y válidas para el problema a resolver.

Para funciones de una sola variable, el algoritmo parece arrojar mejores soluciones utilizando el cruce uniforme, mientras que para funciones de dos, se obtienen resultados buenos y similares para cualquiera de los cuatro métodos.

Estas deducciones se han obtenido en base al algoritmo calibrado, pero hubieran podido variar con la utilización de otros valores para los parámetros que han permanecido fijos durante todo este estudio, con otro número de generaciones, con otro tamaño de población, con otro intervalo para la generación de los parámetros de los individuos, etc. Pero, lo importante, es que el algoritmo genético cumple su cometido.

4.3 Otros resultados

A continuación, se van a presentar los resultados obtenidos de ejecutar el algoritmo genético a otras funciones de más de dos variables, para saber si el proceso evolutivo también se comporta correctamente. Para ello, se va a seguir manteniendo el mismo algoritmo calibrado, así como los mismos parámetros fijos.

Además, el algoritmo sólo se ejecutará una vez por método. Sólo se pretende demostrar que el algoritmo funciona de forma válida para funciones de más variables.

4.3.1 Aproximación de (7) $f(x, y, z) = (x + y + z)^2 + 1$

Los puntos muestra para esta función de tres variables se encuentran en el fichero 'p31.dat'. Una vez ejecutado el algoritmo, obtenemos:

TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.223544	0.475111	0.0711322	**0.030765**
Elegido	Mejor Cruce = Uniforme (selectivo)			

Tabla 12. Resultados de la Función 7 aplicando el algoritmo

4.3.2 Aproximación de (8) $f(x, y, z, w) = \frac{1}{2}x + \frac{1}{4}y + \frac{1}{6}z + \frac{1}{8}w$

Los datos iniciales para procesar esta función de cuatro variables están en el archivo 'p41.dat'. Tras aplicarle el proceso evolutivo, se consiguen los siguientes resultados:

TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	0.0691783	0.0637275	**0.00696535**	0.072556
Elegido	Mejor Cruce = Uniforme			

Tabla 13. Resultados de la Función 8 aplicando el algoritmo

4.3.3 Aproximación de (9) $f(x, y, z, w, t) = \dfrac{(x * y) + (z * w)}{t}$

Los puntos muestra de esta función de cinco variables, con $t > 0$, se encuentran en el archivo 'p51.dat'. Procesados esos datos con el algoritmo genético, se obtienen:

TP	150	150	150	150
PC	0.90	0.90	0.90	0.90
PM	0.05	0.05	0.05	0.05
TS	K-Torneo	K-Torneo	K-Torneo	K-Torneo
NG	500	500	500	500
Tipo de Cruce	*1 punto*	*1 punto (select)*	*Uniforme*	*Uniforme (select.)*
Mejor Fitness	**0.0676449**	0.228856	0.0904584	0.193806
Elegido	Mejor Cruce = 1 punto			

Tabla 14. Resultados de la Función 9 aplicando el algoritmo

4.4 Conclusiones

Una vez realizado el estudio con las funciones objetivo, se pueden sacar cierto tipo de conclusiones.

Para empezar, el calibrado que se ha efectuado al algoritmo ha resultado ser satisfactorio para todos los casos analizados, ofreciendo en todos ellos una solución factible y buena. No obstante, puede que haber utilizado ese mismo algoritmo fijando otros valores para los parámetros hubiera proporcionado mejores resultados, o peores. En cualquiera de los casos, el objetivo era establecer una pauta general que ofreciera resultados válidos, y se ha conseguido de forma notable.

Destacar, además, que en el algoritmo se ha utilizado la opción de reemplazar a los hijos en la nueva población cada vez que se aplicara el cruce, habiendo la opción de seleccionar los mejores entre ellos y sus padres, lo que puede que hubiera arrojado soluciones algo mejores a las obtenidas.

Otro aspecto importante es el de dilucidar cuál de los cuatro tipos de cruce suele ofrecer mejores soluciones. En este capítulo, el cruce uniforme se ha comportado la mayoría de veces como la mejor técnica de todas, obteniendo mejores soluciones que el resto y en ocasiones con una diferencia más que notable. Tras éste, el siguiente que se podría utilizar sería el cruce en 1 punto, siendo desaconsejables los dos tipos de cruce selectivo, que en la mayor parte de los casos ofrecen las peores soluciones.

Decir, no obstante, que para funciones complejas se obtienen soluciones, válidas todas, muy similares entre ellas, por lo que a veces no importa utilizar un método u otro, ya que el algoritmo proporciona buenos resultados igualmente.

Recalcar, por último, que este estudio ha sido realizado en base al calibrado del algoritmo que se estableció, pero los resultados podrían verse alterados en función de otro tipo de ajustes y valores que se pudieran asignar a los parámetros que intervienen en la ejecución del proceso evolutivo.

4.5 Posibles Ampliaciones

Este proyecto puede ser ampliado y/o continuado desarrollando o sustituyendo conceptos por otros nuevos. Algunas de las posibles modificaciones, podrían ser, entre otras:

- Utilización de más funciones candidatas para consturir los straight line programs. Esto es, aumentar el número de opciones para construir las instrucciones no escalares. En este proyecto se ha dicho que se utiliza el conjunto de operaciones {+, -, *, /} y podría ampliarse, por ejemplo, a el conjunto {+, -, *, /, sin, cos, log, sqrt (raíz), …}

- Abordar la resolución del problema de la regresión simbólica en el que los ejemplos del conjunto muestra tienen ruido. En toda medición de campo se pueden producir errores o ruido, de manera que el conjunto muestra o de entrenamiento no responda en algunos puntos exactamente a la función que se pretende "aprender." En esos casos un sobreentrenamiento para ese conjunto muestra podría no proporcionar un buen modelo. Así pues sería muy interesante diseñar procesos evolutivos con funciones de fitness adecuadas, para obtener buenos modelos en presencia de ruido.

- Diseño de nuevos operadores de recombinación para la estrategia evolutiva

- Considerar en el slp universal o patrón, valores reales en el intervalo convexo [0,1] para aquellos parámetros que en el presente proyecto hemos considerado como booleanos. Obsérvese que en la estructura de slp patrón, los parámetros que determinan la operación (ya sea división o producto) toman valores en {0,1} para cada instrucción no escalar. Sería muy interesante permitir para dichos parámetros valores en [0,1]. Intuitivamente estaríamos considerando instrucciones no escalares que tienen un cierto porcentaje de producto y otro porcentaje de división.

- Explorar nuevas funciones fitness, con términos de penalización basados en la complejidad del modelo considerado.

- …

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

Capítulo 5 : Código Fuente

Pablo Solar Rodríguez

ÍNDICE

Capítulo 5 : Código Fuente

A continuación se expone el código fuente de todas las clases java que intervienen en el TFM.

5.1 tfm.genetics.individuo.Individuo

```
package tfm.genetics.individuo;

import java.util.Vector;

import tfm.genetics.utilities.Utilidades;

public class Individuo {

	// Vector cuyo tamanio sera igual al numero de Ui's del individuo y que nos
	// indicara si en el Ui se produce una multiplicacion (true) o una
	// division (false)
	Vector<Boolean> operacionEscalar;

	// Vector de alphas y betas (parametros) de valores reales
	Vector <Double> parametrosReales;

	// Variable que almacena el valor de la funcion fitness del individuo
	double fitness;

	// Rango de valores de los parametros reales
```

```
double limiteInferior, limiteSuperior;

// Tamaño del individuo
int tamanio;

// Numero maximo de operaciones no escalares
int maxoperationsL;

public Individuo()
{
        this.fitness = 0.0;
        this.limiteInferior = 0.0;
        this.limiteSuperior = 0.1;
        operacionEscalar = new Vector<Boolean>();
        parametrosReales = new Vector<Double>();
}

public Individuo (int tamanio, int L, double limiteInf, double limiteSup)
{
        this.tamanio = tamanio;
        this.maxoperationsL = L;
        this.limiteInferior = limiteInf;
        this.limiteSuperior = limiteSup;

        // Vector booleano de tamaño L
        Vector<Boolean> aux  = new Vector<Boolean> (L);

        operacionEscalar = aux;

        // Para generar los valores de los parametros, primero se ha de saber cuantos
```

```
// parametros tendra el individuo. Se sabe el tamanio total del mismo, y se sabe
// cuantos Uk tiene (max_ops_L). Por tanto,
//  Nº parametros = Tamanio_Total - max_ops_L (nº de Uk)
int nParametrosReales = tamanio - maxoperationsL;

Vector<Double> aux2  = new Vector<Double> (nParametrosReales);

parametrosReales = aux2;

// Se rellena el vector de booleanos
// Se sabe que habra tantos parametros booleanos como Ui, y que hay tantos Ui
// como valor tenga max_ops_L (es decir, el tamanio eficaz). Por tanto
for (int j = 0; j < maxoperationsL; j++)
{
        double k = Utilidades.numerosAleatoriosEntre01();
        if (k >= 0.5)

                operacionEscalar.add (j, true);
        else

                operacionEscalar.add (j, false);
}
// Y ahora se genera un vector de ese tamaño con valores entre a y b
for (int  j = 0; j < nParametrosReales; j++)
{
        double parametroJ = Utilidades.numerosAleatoriosEntreAB (limiteInferior, limiteSuperior);
        parametrosReales.add (j, parametroJ);
}

// Se inicializa el valor fitness a cero
```

```
		fitness = 0.0;
	}

	public Vector<Boolean> getOperacionEscalar() {
		return operacionEscalar;
	}

	public Vector<Double> getParametrosReales() {
		return parametrosReales;
	}

	public double getFitness() {
		return fitness;
	}

	public void setFitness(double fitness) {
		this.fitness = fitness;
	}

	public double getLimiteInferior() {
		return limiteInferior;
	}

	public double getLimiteSuperior() {
		return limiteSuperior;
	}

}
```

5.2 tfm.genetics.poblacion.Poblacion

```
package tfm.genetics.poblacion;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Collections;
import java.util.Vector;

import tfm.genetics.individuo.Individuo;
import tfm.genetics.slp.SLP;
import tfm.genetics.utilities.Utilidades;

public class Poblacion {

	// Vector que albergara a la poblacion
	Vector<Individuo> population = null;


	public Poblacion ()
	{
		population = new Vector<Individuo> ();
	}


	public Poblacion (int tamanioPob, int tamanioInd, int L, double limInf, double limSup)
	{
		population = new Vector<Individuo> (tamanioPob);
```

```
        for (int i = 0; i < tamanioPob; i++)
        {
                // Se llama al constructor de individuos para construir individuos aleatorios
                Individuo ind  = new Individuo (tamanioInd, L, limInf, limSup);
                population.add(i, ind);
        }
}


public void calculaFitnessPoblacion (String ruta, int nVariables, SLP straightLineProgram)
{
        // Ahora hay que calcular el valor fitness de cada individuo de la poblacion, y
        // para ello es necesario leer todos los valores de los puntos muestra del
        // fichero para cada uno de ellos

        // La variable x leera las componentes
        // La variable f leera el valor de la funcion
        double x;
        double f;



        for (int i = 0; i < (int) population.size(); i++)
        {
                try
                {
                        // Ahora se abre el archivo con esa ruta
                        FileReader file = new FileReader(ruta);
                        BufferedReader fileStream = new BufferedReader(file);
```

```
if (fileStream.ready())
{
	// Vector que recoge el numero total de puntos muestra
	int nPuntosMuestra = 0;

	// Valor que recogera el valor fitness del individuo i
	double valorFitness = 0;

	// Se procesa la primera componente
	String temp = fileStream.readLine();

	while (temp!=null)
	{
		// Vector que almacenara cada conjunto de componentes
		Vector<Double> componentes = new Vector<Double> ();

		// Variable que recogera el output del individuo i asociado a Ui
		double output;

		String[] contenido = temp.split("\\s+");

		for (int j = 0; j < nVariables; j++)
			componentes.add(j, Double.parseDouble(contenido[j]));

		// Procesamos el valor de la funcion en esos puntos
		f = Double.parseDouble(contenido[contenido.length-1]);

		output = straightLineProgram.calculaOutput (componentes, population.get(i).getOperacionEscalar(),
population.get(i).getParametrosReales());
```

```
                                        // Se actualiza el valor del fitnes
                                        double t = output - f;
                                        valorFitness += Math.pow (t, 2);

                                        // Incrementamos el numero de puntos muestra leidos
                                        nPuntosMuestra++;

                                        // Se lee el siguiente conjunto de puntos
                                        temp = fileStream.readLine();
                                }

                                // Se calcula el valor final del fitness diviendolo entre el numero de
                                // puntos muestra y se asocia al individuo i
                                valorFitness /= nPuntosMuestra;

                                // Se asigna dicho valor fitness al individuo
                                population.get(i).setFitness(valorFitness);

                                fileStream.close();
                        }
                }
                catch (Exception e)
                {
                        System.out.println("No se ha podido abrir el archivo o ha ocurrido algun error");
                }

        }
}
```

```
public Individuo mejorIndividuo ()
{
	// Vector que almacenara los fitness de los individuos
	Vector<Double> fitnessIndividuos = new Vector<Double>();
	Individuo mejor = new Individuo();

	for (int i = 0; i < (int) population.size(); i++)
	{
		fitnessIndividuos.add(population.get(i).getFitness());
	}

	double mejorFitness;

	// Se busca el mejor fitness del vector
	mejorFitness = Collections.min(fitnessIndividuos);

	for (int i = 0; i < (int) population.size(); i++)
	{
		if (mejorFitness == population.get(i).getFitness())
		{
			mejor = population.get(i);
		}
	}

	return mejor;
}


public Vector<Individuo> getPopulation() {
```

```
        return population;
}


public int ruleta()
{
        // Se calcula la suma de todos los fitness de los indivudos, que se sera un valor
        // al que se llamara F
        double F = 0.0;

        int elegido = 0;;

        for (int i = 0; i < (int)population.size(); i++)
                F = F + population.get(i).getFitness();

        // Se crea un vector de valores que albergaran para cada individuo i, un valor
        // F - fitness(i)
        Vector<Double> fitnessAux = new Vector<Double>();

        for (int i = 0; i < (int)population.size(); i++)
        {
                fitnessAux.add (F - population.get(i).getFitness());
        }

        // Es necesario establecer este "nuevo fitness" porque el método de la ruleta
        // tradicional, supone que a mayor valor del fitness, mejor es el individuo.
        // Sin embargo en nuestro problema, como el fitness es el error, lo que ocurre
        // es que los mejores individuos son los de fitness más pequeños. Esa es la razón
        // de calcular un fitnes_aux. El fitness_aux sí que verifica que cuanto mayor es,
        // mejor es el individuo. Así pues, se aplica el método de la ruleta pero
```

```
        // considerando el fitness_aux

        // Se calcula la suma de todos los fitness_aux de los indivudos, que se sera un
        // valor al que se llamara F_aux
        double FAux = 0.0;

        for (int i = 0; i < (int)fitnessAux.size(); i++)
                FAux = FAux + fitnessAux.get(i);

        // Se genera un numero al azar entre 0 y F
        double aleatorio = Utilidades.numerosAleatoriosEntreAB (0, FAux);

        // Se recorre de nuevo la poblacion acumulando los fitness. En el momento en que
        // la suma sea igual o mayor que el numero aleatorio, ese es el individuo elegido
        double contador = 0.0;

        for (int i = 0; i < (int)population.size(); i++)
        {
                contador = contador + fitnessAux.get(i);

                if (contador >= aleatorio)
                {
                        elegido = i;
                        break;
                }
        }

        return elegido;
}
```

```
public int kTorneo (int k)
{
	// Se generan al azar K numeros de entre el tamaño de la poblacion, donde el
	// numero i generado estara asociado al individuo i de la poblacion
	double seleccionado = 0.0;

	int elegido = 0;

	int k_torneo = k;

	int aux;

	// El vector almacenara los indices de los Individuos seleccionables de la
	// poblacion
	Vector<Integer> seleccionables = new Vector<Integer>();

	for (int i = 0; i < k_torneo; i++)
	{
		// Se generan K numeros al azar
		aux = (int) Utilidades.numerosAleatoriosEntreAB (0, population.size()-1);

			// Los insertamos en el vector
			seleccionables.add (aux);
		}

	Vector<Double> fitnessSeleccionables = new Vector<Double>();

	for (int i = 0 ; i < k_torneo; i++)
		fitnessSeleccionables.add (population.get(seleccionables.get(i)).getFitness()) ;
```

```
		// Se elige el mejor de esos, que sera el que tenga menor fitness asociado
		for (int i = 0; i < (int) fitnessSeleccionables.size()-1; i++)
			seleccionado = Math.min (fitnessSeleccionables.get(i), fitnessSeleccionables.get(i+1));


		for (int i = 0; i < (int) seleccionables.size(); i++)
			if (population.get(seleccionables.get(i)).getFitness() == seleccionado)
			{
					elegido = seleccionables.get(i);
					break;
			}

		return elegido;
}


public void cruce1Punto(Individuo padre1, Individuo padre2, Individuo hijo1, Individuo hijo2)
{
		// Se genera un numero al azar para calcular el punto de cruce de los
		// parametros booleanos de los individuos
		int tamanioBool = padre1.getOperacionEscalar().size();

		int puntoCruceBooleanos = (int) Utilidades.numerosAleatoriosEntreAB(0, tamanioBool);

		// Se cruzan ambos parametros de los individuos padre
		for (int i = 0; i < puntoCruceBooleanos; i++)
		{
			hijo1.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
			hijo2.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));
```

```
        }

        for (int i = puntoCruceBooleanos; i < tamanioBool; i++)
        {
                        hijo1.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));
                        hijo2.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
        }

        // Se genera un numero al azar para calcular el punto de cruce de los
        // parametros reales de los individuos
        int tamanio_reales = padre1.getParametrosReales().size();

        int puntoCruceReales = (int) Utilidades.numerosAleatoriosEntreAB (0, tamanio_reales);

        // Se cruzan ambos parametros de los individuos padre
        for (int i = 0; i < puntoCruceReales; i++)
        {
                hijo1.getParametrosReales().add (padre1.getParametrosReales().get(i));
                hijo2.getParametrosReales().add (padre2.getParametrosReales().get(i));
        }

        for (int i = puntoCruceReales; i < tamanio_reales; i++)
        {
                        hijo1.getParametrosReales().add (padre2.getParametrosReales().get(i));
                        hijo2.getParametrosReales().add (padre1.getParametrosReales().get(i));
        }
}


public void cruce1PuntoSelectivo(Individuo padre1, Individuo padre2, Individuo hijo1, Individuo hijo2, SLP straightLineProgram)
```

```
{
        // Para los parametros booleanos la manera de cruce es exactamente igual al cruce
        // no selectivo

        // Se genera un numero al azar para calcular el punto de cruce de los
        // parametros booleanos de los individuos
        int tamanioBool = padre1.getOperacionEscalar().size();

        int puntoCruceBooleanos = (int) Utilidades.numerosAleatoriosEntreAB (0, tamanioBool);

        // Se cruzan ambos parametros de los individuos padre
        for (int i = 0; i < puntoCruceBooleanos; i++)
        {
                hijo1.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
                hijo2.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));
        }

        for (int i = puntoCruceBooleanos; i < tamanioBool; i++)
        {
                        hijo1.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));
                        hijo2.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
        }

        // Se aplica ahora el cruce selectivo, diferente al cruce no selectivo

        // Se genera un numero al azar para calcular el punto de cruce de los
        // parametros reales de los individuos
        int tamanioTotalReales = padre1.getParametrosReales().size();

        // Los ultimos parametros reales corresponden al output y da igual
```

```
// como se crucen tambien, por lo que solo interesa hacer cruce selectivo
// en los Ui's, y habra que calcular los parametros que tendran
int tamanioOutput = straightLineProgram.tamanioUi(straightLineProgram.getSlp().size());

int tamanioRealesUi = tamanioTotalReales - tamanioOutput;


int puntoCruceReales = (int) Utilidades.numerosAleatoriosEntreAB (0, tamanioTotalReales);

if (puntoCruceReales < tamanioRealesUi)
{
	// El punto de cruce cae en algun Ui y hay que actualizarlo aplicando
	// el cruce selectivo

	// Ahora es necesario saber en que Ui se encuentra el punto generado
	int index1 = 0;
	int index2 = 0;
	for (int i = 1; i <= straightLineProgram.getUk().size(); i++)
	{
		index2 += straightLineProgram.tamanioUi(i);

		if ((index1 < puntoCruceReales) && (puntoCruceReales < index2))
		{
		// Se necesita una referencia para saber en que parte del Ui cae el punto
			// generado, en este caso sera la mitad del Ui con el valor del indice
			int mitadUi = index1 + (straightLineProgram.tamanioUi (i) / 2);

			if (puntoCruceReales < mitadUi)
			{
				// Esta partiendo una ristra de alphas
```

```
                                if ((mitadUi - puntoCruceReales) < (puntoCruceReales - index1))
                                // El punto esta mas cerca del inicio de la ristra de betas que del
                                // inicio de la ristra de alphas
                                        puntoCruceReales = mitadUi;

                                else
                                        puntoCruceReales = index1;
                        }

                        if (puntoCruceReales > mitadUi)
                        {
                                // Esta partiendo una ristra de betas
                                if ((index2 - puntoCruceReales) < (puntoCruceReales - mitadUi))
                                // El punto esta mas cerca del inicio de la ristra de alphas que del
                                // inicio de la ristra de betas
                                        puntoCruceReales = index2;

                                else
                                        puntoCruceReales = mitadUi;
                        }

                        break;
                }

                else
                        index1 += straightLineProgram.tamanioUi (i);
        }
}

// El punto de cruce cae en los parametros correspondientes al output y
```

```
		// el cruce seria normal
		// Se cruzan ambos parametros de los individuos padre
		for (int i = 0; i < puntoCruceReales; i++)
		{
			hijo1.getParametrosReales().add (padre1.getParametrosReales().get(i));
			hijo2.getParametrosReales().add (padre2.getParametrosReales().get(i));
		}

		for (int i = puntoCruceReales; i < tamanioTotalReales; i++)
		{
				hijo1.getParametrosReales().add (padre2.getParametrosReales().get(i));
				hijo2.getParametrosReales().add (padre1.getParametrosReales().get(i));
		}

}


public void cruceUniforme(Individuo padre1, Individuo padre2, Individuo hijo1, Individuo hijo2)
{
		// Lo primero de todo sera generar dos mascaras aleatorias, una para los
		// parametros booleanos y otro para los reales
		int tamanioBool = padre1.getOperacionEscalar().size();
		int tamanioReales = padre1.getParametrosReales().size();
		Vector<Boolean> mascaraBooleana = new Vector<Boolean>(padre1.getOperacionEscalar().size());
		Vector<Boolean> mascaraReales = new Vector<Boolean>(padre1.getParametrosReales().size());

		for (int i = 0; i < tamanioBool; i++)
		{
			double b = Utilidades.numerosAleatoriosEntre01();
```

```
		if (b >= 0.5)
		{
			mascaraBooleana.add(i, true);
		}
		else
		{

			mascaraBooleana.add(i, false);
		}
}

for (int i = 0; i < tamanioReales; i++)
{
		double b = Utilidades.numerosAleatoriosEntre01();

		if (b >= 0.5)
		{
			mascaraReales.add(i, true);
		}
		else
		{
			mascaraReales.add(i, false);
		}
}

// En este punto se tienen generadas de forma aleatoria unas mascaras que seran
// distintas para cada cruce.

/* La politica de cruce a uilizar sera la siguiente:
```

```
                Si 1 en Mascara -> Parametro del padre1 para el hijo1
                Si 0 en Mascara -> Parametro del padre2 para el hijo1

                Si 1 en Mascara -> Parametro del padre2 para el hijo2
                Si 0 en Mascara -> Parametro del padre1 para el hijo2
*/

// Se aplica el cruce selectivo con la politica anterior para crear el hijo1 y
// el hijo 2
for (int i = 0; i < mascaraBooleana.size(); i++)
{
        if (mascaraBooleana.get(i) == true)
        {
                hijo1.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
                hijo2.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));
        }
        else
        {
                hijo1.getOperacionEscalar().add  (padre2.getOperacionEscalar().get(i));
                hijo2.getOperacionEscalar().add  (padre1.getOperacionEscalar().get(i));
        }
}

for (int i = 0; i < mascaraReales.size(); i++)
{
        if (mascaraReales.get(i) == true)
        {
                hijo1.getParametrosReales().add (padre1.getParametrosReales().get(i));
                hijo2.getParametrosReales().add (padre2.getParametrosReales().get(i));
        }
```

```
                else
                {
                        hijo1.getParametrosReales().add (padre2.getParametrosReales().get(i));
                        hijo2.getParametrosReales().add (padre1.getParametrosReales().get(i));
                }
        }
}

public void cruceUniformeSelectivo(Individuo padre1, Individuo padre2, Individuo hijo1, Individuo hijo2, SLP straightLineProgram) {

        // Lo primero de todo sera generar dos mascaras aleatorias, una para los
        // parametros booleanos y otro para los reales
        Vector<Boolean> mascaraBooleana = new Vector<Boolean>();
        Vector<Boolean> mascaraReales = new Vector<Boolean>();

        // La manera de generar la mascara para los parametros booleanos es exactamente
        // igual que en el cruce uniforme normal
        for (int i = 0; i < padre1.getOperacionEscalar().size(); i++)
        {
                double b = Utilidades.numerosAleatoriosEntre01();

                if (b >= 0.5)
                        mascaraBooleana.add(true);
                else
                        mascaraBooleana.add(false);
        }

        // La manera de generar la mascara de parametros reales es un tanto distinta; por
        // cada ristra de alphas y de betas se generara un 1 o un 0 aleatoriamente.
```

```
//  En cada Ui hay dos ristras de igual tamaño de alphas y de betas y luego el
// output tiene parametros independientes. Por tanto, el tamanio de esta mascara
// sera de (2 * nº de Ui's) + Tamanio_Output
int nUis = straightLineProgram.getUk().size() - 1; // Tambien se almacena el output
int tamanioOutput = straightLineProgram.tamanioUi(straightLineProgram.getSlp().size());
int tamanioMascaraReales = (2 * nUis) + tamanioOutput;

for (int i = 0; i < tamanioMascaraReales; i++)
{
	double b = Utilidades.numerosAleatoriosEntre01();

	if (b >= 0.5)
		mascaraReales.add(true);
	else
		mascaraReales.add(false);
}

// En este punto se tienen generadas de forma aleatoria unas mascaras que seran
// distintas para cada cruce.

/* La politica de cruce a uilizar sera la siguiente:

	Para los parametros booleanos:

		Si 1 en Mascara -> Parametro del padre1 para el hijo1
		Si 0 en Mascara -> Parametro del padre2 para el hijo1

		Si 1 en Mascara -> Parametro del padre2 para el hijo2
		Si 0 en Mascara -> Parametro del padre1 para el hijo2
```

```
        Para los parametros reales:

                Si 1 en Mascara -> Riestra de alphas o betas del padre1 para el hijo1
                Si 0 en Mascara -> Riestra de alphas o betas del padre2 para el hijo1

                Si 1 en Mascara -> Riestra de alphas o betas del padre2 para el hijo2
                Si 0 en Mascara -> Riestra de alphas o betas del padre1 para el hijo2

                (A excepcion de que estemos en el output que se efectua cruce normal)
*/

// Aplicacion del cruce selectivo con la politica anterior para crear los
// parametros booleanos del hijo1 y el hijo 2
for (int i = 0; i < mascaraBooleana.size(); i++)
{
        if (mascaraBooleana.get(i) == true)
        {
                hijo1.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
                hijo2.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));

        }
        else
        {
                hijo1.getOperacionEscalar().add (padre2.getOperacionEscalar().get(i));
                hijo2.getOperacionEscalar().add (padre1.getOperacionEscalar().get(i));
        }
}

// Aplicacion del cruce selectivo con la politica anterior para crear los
```

```
// parametros reales del hijo1 y el hijo 2

// Variables que nos acotaran rangos en los que se encuentran los alpas y los
// betas de cada Ui
int index1 = 0;
int index2 = 0;
int indiceMascara = 0;

for (int i = 0; i < nUis; i++)
{
	// se sabe que por cada Ui va a haber 2 valores en la mascara de parametros,
	// uno para la ristra de alphas y otra para la ristra de betas
	int tamanioUi = straightLineProgram.tamanioUi(i+1);

	// index2 aqui tiene el tamo del ui, para acotar el rango a solo alphas o
	// betas lo dividimos entre 2
	index2 += tamanioUi / 2;

	// Tratamos la ristra de alphas
	for (int j = index1; j < index2; j++)
	{
		if (mascaraReales.get(indiceMascara) == true)
		{
			hijo1.getParametrosReales().add (padre1.getParametrosReales().get(j));
			hijo2.getParametrosReales().add (padre2.getParametrosReales().get(j));
		}

		else
		{
			hijo1.getParametrosReales().add (padre2.getParametrosReales().get(j));
```

```
                        hijo2.getParametrosReales().add (padre1.getParametrosReales().get(j));
                }
        }

        // Siguiente rango, que sera el correspondiente a las betas del Ui actual
        index1 += tamanioUi / 2;
        index2 += tamanioUi / 2;
        indiceMascara++;

        // Tratamiento de la ristra de betas
        for (int j = index1; j < index2; j++)
        {
                if (mascaraReales.get(indiceMascara) == true)
                {
                        hijo1.getParametrosReales().add (padre1.getParametrosReales().get(j));
                        hijo2.getParametrosReales().add (padre2.getParametrosReales().get(j));
                }

                else
                {
                        hijo1.getParametrosReales().add (padre2.getParametrosReales().get(j));
                        hijo2.getParametrosReales().add (padre1.getParametrosReales().get(j));
                }
        }

        index1 += tamanioUi / 2;

        indiceMascara++;
}
```

```
        // Y ahora se ha de tratar el output como un cruce normal
        for (int i = index2; i < padre1.getParametrosReales().size(); i++)
        {
                if (mascaraReales.get(indiceMascara) == true)
                {
                        hijo1.getParametrosReales().add (padre1.getParametrosReales().get(i));
                        hijo2.getParametrosReales().add (padre2.getParametrosReales().get(i));
                }

                else
                {
                        hijo1.getParametrosReales().add (padre2.getParametrosReales().get(i));
                        hijo2.getParametrosReales().add (padre1.getParametrosReales().get(i));
                }

                indiceMascara++;
        }
}


public void mutacion(Individuo ind) {

        int probTotal = ind.getOperacionEscalar().size() + ind.getParametrosReales().size();
        int probBooleano = ind.getOperacionEscalar().size();
        int probReal = ind.getParametrosReales().size();

        // Se Genera un numero aleatorio entre los parametros totales
        int k = (int) Utilidades.numerosAleatoriosEntreAB(0, probTotal-1);

        // El numero generado cae dentro de los parametros booleanos
```

```
        if ((0 <= k) && (k < probBooleano))
        {
                if (ind.getOperacionEscalar().get(k) == false)
                {
                        ind.getOperacionEscalar().remove(k);
                        ind.getOperacionEscalar().add(k, true);
                }


                if (ind.getOperacionEscalar().get(k) == true)
                {
                        ind.getOperacionEscalar().remove(k);
                        ind.getOperacionEscalar().add(k, false);
                }
        }

        // El numero generado cae dentro de los reales
        else if (k > probBooleano)
        {
                double parametro = Utilidades.numerosAleatoriosEntreAB(ind.getLimiteInferior(), ind.getLimiteSuperior());
                ind.getParametrosReales().remove(k - probBooleano - 1);
                ind.getParametrosReales().add(k - probBooleano - 1, parametro);
        }

}


public void mutacionSesgada(Individuo ind)
{
        // Es necesario elegir que parametro se va a mutar, si un booleano o un real
```

```
boolean mutacion;

double b = Utilidades.numerosAleatoriosEntre01();

if (b >= 0.5)
        mutacion = true;

else
        mutacion = false;

if (mutacion == false)
{
        // Mutación de un parametro booleano
        int tam = ind.getOperacionEscalar().size();
        int k = (int) Utilidades.numerosAleatoriosEntreAB(0, tam-1);

        if (ind.getOperacionEscalar().get(k) == false)
        {
                ind.getOperacionEscalar().remove(k);
                ind.getOperacionEscalar().add (k, true);
        }

        if (ind.getOperacionEscalar().get(k) == true)
        {
                ind.getOperacionEscalar().remove(k);
                ind.getOperacionEscalar().add (k, false);
        }
}

if (mutacion == true)
```

```
                {
                        // Mutación de un parametro real
                        int tam = ind.getParametrosReales().size();
                        int k = (int) Utilidades.numerosAleatoriosEntreAB(0, tam-1);

                        double parametro = Utilidades.numerosAleatoriosEntreAB(ind.getLimiteInferior(),ind.getLimiteSuperior());

                        ind.getParametrosReales().remove(k);
                        ind.getParametrosReales().add(k, parametro);
                }
        }
}
```

5.3 tfm.genetics.slp.SLP

```
package tfm.genetics.slp;

import java.util.HashMap;
import java.util.Vector;

import tfm.genetics.utilities.Utilidades;

public class SLP {

        // Variable que establece el numero de operaciones distintas de {+,-}
```

```
private int maxOperationsL;


// Variable que establece el numero de variables que utilizara el programa
private int nVariables;

// Diccionario que representara al slp
HashMap <Integer, Vector<Double>> slp = new HashMap <Integer, Vector<Double>>();


// Vector que contendra los valores calculados de cada Ui
Vector <Double> Uk;


public SLP (int nVariables, int L)
{
	this.nVariables = nVariables;
	this.maxOperationsL = L;

	for (int i = 1; i <= maxOperationsL; i++)
	{
		// Se calcula el numero de alphas que necesita el Ui
		int a = calculaAlphasBetas (i, nVariables);

		// El numero de betas sera igual que el numero de alphas
		// por lo que el numero total de ambos es
		a *= 2;
		Vector<Double> vector = new Vector<Double> ();
		Utilidades.rellenaVector(vector, a);
		slp.put(i, vector);
```

```
        }

        // Se calcula el numero de parametros que tiene el OUTPUT
        // SUM (j = -n+1 , L) (X) = L + n - 1 + 1 veces (X) = L + n
        int o = maxOperationsL + nVariables;

        // Se inserta el OUTPUT en nuestra estructura
        Vector<Double> vector = new Vector<Double> ();
        Utilidades.rellenaVector(vector, o);
        slp.put (maxOperationsL+1, vector );

        // Se inicializa el vector de resultados a valor 0 para todos los Uk
        // Tendra el mismo tamanio como tamanio eficaz tenga el SLP, es decir, el numero
        // de opraciones distintas de {-, +} + 1, para albergar tambien el valor del
        // Output
        Vector <Double> aux  = new Vector<Double> ();
        Utilidades.rellenaVector(aux, maxOperationsL+1);
        Uk = aux;

}


private int calculaAlphasBetas(int i, int nVar)
{
        // Por propiedades de los sumatorios
        // SUM (j = -n+1, i-1 ) (X) = i - 1 + n - 1 + 1 veces (x) = i + n - 1

        return i+nVar-1;
}
```

```
public int calculaParametros (int n, int L)
{
	int aux = 2*n;
	aux += L;
	aux *= L;
	aux += L + n;
	return aux;
}


public double calculaOutput(Vector<Double> components, Vector<Boolean> op, Vector<Double> parameters) {

	// Variable que almacenara el valor del Ouput actual
	double out = 0.0;

	// Se sabe que el Straight Line Program tiene tanto tamanio como tamanio tenga
	// el vector Uk de la estructura, ya que almacena el numero de Ui y el output.
	// Por tanto
	int numeroUis = Uk.size()-1;

	// Indice que recorrera el vector de parametros del individuo
	int index = -1;

	for (int i = 1; i <= numeroUis; i++)
	{
		// Bucle que simulara el recorrido de cada Ui. Se inicia con valor uno ya que
		// en el diccionario del straight line program los valores de los Uis toman
		// valores i = 1....n
```

```
// Se calcula el numero de parametros que tiene el Ui actual
int tamanio_Ui = tamanioUi (i);

// Declaracion de operandos. Al final El resultado siempre sera una operacion
// binaria de dos operandos -->
// (a1, a2,.., ak+1)(x, y, U1, ... Uk)opB(b1, b2, ..., bk+1)(x, y, U1, ... Uk)
double op1, op2;

if (i == 1)
{
	// El primer Ui se trata distinto ya que no necesita el calculo de otros
	// Ui, solo de las variables
	op1 = op2 = 0.0;
	op1 = calculaOperandoVariables (index, components, parameters);
	op2 = calculaOperandoVariables (index, components, parameters);

	// Se resta un valor al indice ya que la funcion anterior devuelve el
	// indice apuntando a un elemento posterior tras el calculo de los
	// dos operandos
	index--;

	// Comprobacion de operacion. En este caso tenemos que mirar siempre el U1
	// que se almacena en la posicion 0 del vector Uk. Por tanto
	if (op.get(i-1) == true)
	{
		Uk.remove(i-1);
		Uk.add(i-1, op1*op2);
	}

	else
```

```
		{
			Uk.remove(i-1);
			Uk.add(i-1, op1/op2);
		}
}

if (i > 1)
{
		op1 = op2 = 0.0;
		index++;
		op1 = calculaOperandoVariables (index, components, parameters);

		//  Ahora hay que sumar el valor de los Ui anteriores multiplicados por el
		// parametro escalar al que que apunte index. Hay que calcular un numero de
		// Uis equivalente al numero de parametros que tenga el Ui actual dividido
		// entre 2 (que corresponde al numero de parametros del primer operando) y
		// a ese valor restarle el numero de variables. Es decir, supongase que
		// estamos en el U2 con 3 variables, el tamanio de U2 seria
		// (3 variables_alphas + 1 U1 + 3 variables_betas + 1 U1). Habria que
		// calcular solo un Ui, el U1, por tanto quedaria
		// Nº Ui a calcular = (8/2)-3 = 1
		for (int j = 0; j < (tamanio_Ui/2)-(int)components.size(); j++)
		{
			index++;
			op1 += parameters.get(index) * Uk.get(j);
		}

		// Ahora se calcula el segundo parametro
		op2 = calculaOperandoVariables (index, components, parameters);
```

```
                // Y se suma el valor de los Ui anteriores analogamente al caso anterior
                for (int j = 0; j < (tamanio_Ui/2)-(int)components.size(); j++)
                {
                        index++;
                        op2 += parameters.get(index) * Uk.get(j);
                }

                // Comprobacion de operacion. En este caso tenemos que mirar siempre el Ui
                // que se almacena en la posicion i-1 del vector Uk. Por tanto
                if (op.get(i-1) == true)
                {
                        Uk.remove(i-1);
                        Uk.add(i-1, op1*op2);
                }

                else
                {
                        Uk.remove(i-1);
                        Uk.add(i-1, op1/op2);
                }

                // Se resta un valor al indice ya que la funcion anterior devuelve el
                // indice apuntando a un elemento posterior tras el calculo de los
                // dos operandos y luego se ha vuelto a incrementar
                index--;
        }

}
        // Aqui se tendran todos los valores de los Ui y habra que calcular
        // el valor del ouput. Ademas, el index se mantiene actualizado apuntando
```

```
                // al siguiente parametro del individuo. En la estructura del striaight line
                // program, el output se almacena en la ultima posicion del diccionario, asi
                // pues para calcular su tamanio
                int tamanio_output = slp.get(slp.size()).size();

                // Se actualiza el output con el valor aplicado a las variables
                for (int i = 0; i < components.size(); i++)
                {
                        index++;
                        out += parameters.get(index) * components.get(i);
                }

                // Se termina de actualizar el output con los valores
                for (int i = 0; i < tamanio_output - (int)components.size(); i++)
                {
                        index++;
                        out += parameters.get(index) * Uk.get(i);
                }

        return out;
}


public int tamanioUi(int i) {

        return slp.get(i).size();
}


private double calculaOperandoVariables(int ind, Vector<Double> c, Vector<Double> p) {
```

```
                // Variable auxiliar para almacenar el resultado
                double aux = 0.0;

                // Indice para recorrer el vector de componentes
                int j = 0;

                // En el caso de procesar U1, el primer Ui
                if(ind == -1) ind = 0;

                while (j < (int)c.size()) {
                        aux += p.get(ind) * c.get(j);
                        j++;
                        ind++;
                }

                return aux;
        }


        public HashMap<Integer, Vector<Double>> getSlp() {
                return slp;
        }


        public Vector<Double> getUk() {
                return Uk;
        }

}
```

5.4 tfm.genetics.algoritmo.AlgoritmoConstantes

```
package tfm.genetics.algoritmo;

public class AlgoritmoConstantes {

	/**
	 * Constantes para el cruce
	 */
	public static final int  CRUCE_1_PUNTO = 1;

	public static final int  CRUCE_1_PUNTO_SELECTIVO = 2;

	public static final int  CRUCE_UNIFORME = 3;

	public static final int  CRUCE_UNIFORME_SELECTIVO = 4;

	/**
	 * Constantes para el tipo de seleccion
	 */
	public static final int  RULETA = 1;

	public static final int  K_TORNEO = 2;

	/**
	 * Constantes para el tipo de mutacion
	 */
	public static final int  MUTACION_NORMAL = 1;
```

```
	public static final int  MUTACION_SESGADA = 2;

	/**
	 * Constantes para reemplazo de indiviuos
	 */
	public static final int  REPLACE_HIJOS = 1;

	public static final int  REPLACE_MEJORES = 2;

}
```

5.5 tfm.genetics.algoritmo.Algoritmo

```
package tfm.genetics.algoritmo;


import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Collections;
import java.util.Vector;

import tfm.genetics.individuo.Individuo;
import tfm.genetics.panel.base.PanelPrincipalBase;
import tfm.genetics.poblacion.Poblacion;
```

```
import tfm.genetics.slp.SLP;
import tfm.genetics.utilities.Utilidades;


public class Algoritmo {

	// Numero de variables
	private int nVariables;

	// Numero de operaciones no escalares (tamanio eficaz)
	private int L;

	// Tamanio de la poblacion
	private int tamPoblacion;

	// Tamanio del individuo
	private int tamIndividuo;

	// Rango minimo
	private double lower;

	// Rango maximo
	private double upper;

	// Ruta del fichero
	private String ruta;

	// Tipo de seleccion
	private int selection;
```

```
// Tipo de cruce
private int cross;

// Tipo de mutacion
private int mutation;

// Tipo de reemplazamiento
private int replacement;

// Probabilidad de cruce
private double crossProb;

// Probabilidad de mutacion
private double mutationProb;

// Test de parada
private int test;

// Elitismo
private boolean elitism;

// Contador
private int contador = 0;

// K-Torneo
private int kTorneo;

// Variables que indicaran los individuos que han resultado elegidos en el proceso de seleccion
private int father1;
private int father2;
```

```
// Variable que indicara el individuo elegido para mutar
private int fatherM;

// Variable que contendra la probabilidad aleatoria de cruce y mutacion
private double probability;



public Algoritmo() {}

public void inicializar(PanelPrincipalBase vista) {

        this.cross = vista.getCross();
        this.crossProb = vista.getCrossProb();
        this.elitism = vista.getElitism();
        this.L = vista.getL();
        this.lower = vista.getLower();
        this.mutation = vista.getMutation();
        this.mutationProb = vista.getMutationProb();
        this.nVariables = vista.getnVariables();
        this.replacement = vista.getReplacement();
        this.ruta = vista.getRoute();
        this.selection = vista.getSelection();
        this.tamPoblacion = vista.getTamanioPoblacion();
        this.test = vista.getTest();
        this.upper = vista.getUpper();
        this.kTorneo = vista.getkTorneo();
}
```

```
public double ejecutar() {

        // Se crea antes de nada el SLP
        SLP straightLineProgram = new SLP(this.nVariables, this.L);

        // Aquí se tendra la estructura de datos representada en el diccionario llamado slp de nuestro SLP, el vector de valores de cada Uk
        // asi como del Output y una serie de funciones para manejar dicha estructura. En realidad, esta estructura va a ser una plantilla en
        // la que sustituir los valores de cada individuo de la poblacion del algoritmo genetico, y nos servira de patron a la hora de realizar
        // calculos sobre los individuos.

        // Ahora se calculara el numero de parametros totales para el SLP, que no es mas que calcular el tamaño que va tener cada
        // individuo de la poblacion (con sus valores)

        this.tamIndividuo = straightLineProgram.calculaParametros(this.nVariables, this.L);

        ////////////////////////////////////////////////////////////////////////////////
        ///////////////// Comienzo del Algoritmo Genetico /////////////////
        ////////////////////////////////////////////////////////////////////////////////
        // 1) Generacion de una poblacion inicial

        // Seguidamente, con la informacion relativa al tamanio de la poblacion y del individuo, asi como del rango de valores que podran
        // tomar los parametros  escalares, se generara una poblacion con valores generados aleatoriamente para cada individuo. La
        // poblacion no sera mas que un vector que almacena individuos

        Poblacion hominum = new Poblacion(this.tamPoblacion, this.tamIndividuo, this.L, this.lower, this.upper);

        // 2) Evaluacion de la poblacion inicial (calculo del fitness)

        hominum.calculaFitnessPoblacion(this.ruta, this.nVariables,straightLineProgram);
```

```
// Aqui todos los individuos de la poblacion tienen asociado un valor fitness.

// 3) Ejecucion de iteraciones

while (contador < test) {
	// Creacion de una poblacion auxiliar
	Poblacion hominumAux = new Poblacion();

	// Contador de individuos de la nueva poblacion
	int nIndividuosNuevos = 0;

	// 3.1) Algoritmo elitista -> Se inserta el mejor

	if (this.elitism == true) {
		// El algoritmo es elitista y tenemos que meter en la poblacion
		// auxiliar
		// nueva el mejor
		if (nIndividuosNuevos >= this.tamPoblacion) {

			break;// Continua;
		}

		else
			nIndividuosNuevos++;

		hominumAux.getPopulation().add(hominum.mejorIndividuo());
	}

	// Se aplica el algoritmo general evolutivo mientras que la poblacion nueva auxiliar sea menor que la original
```

```
Continua: while (hominumAux.getPopulation().size() < tamPoblacion) {
        // 3.2) Seleccion de padres
        switch (selection) {
        case AlgoritmoConstantes.RULETA:

                // Se obtienen los padres por seleccion por ruleta
                father1 = hominum.ruleta();
                father2 = hominum.ruleta();
                fatherM = hominum.ruleta();

                // Comprobacion de que los individuos padre son distintos
                while (father1 == father2)
                        father2 = hominum.ruleta();

                break;

        case AlgoritmoConstantes.K_TORNEO:

                // Se obtienen los padres por seleccion por torneo
                father1 = hominum.kTorneo(this.kTorneo);
                father2 = hominum.kTorneo(this.kTorneo);
                fatherM = hominum.kTorneo(this.kTorneo);

                // Comprobacion de que los individuos padre son distintos
                while (father1 == father2)
                        father2 = hominum.kTorneo(this.kTorneo);

                break;
        }
```

```
// 3.3) Generacion de un numero al azar para la probabilidad de
// cruce y mutacion
probability = Utilidades.numerosAleatoriosEntre01();

// 3.4) Determinar si se produce cruce

if (probability <= this.crossProb) {
        // 3.4.1) Creacion de los hijos

        Individuo hijo1 = new Individuo();
        Individuo hijo2 = new Individuo();

        // 3.4.2) Determinar el tipo de cruce

        switch (this.cross) {
        case AlgoritmoConstantes.CRUCE_1_PUNTO:

                hominum.cruce1Punto(hominum.getPopulation()
                                .get(father1),
                                hominum.getPopulation().get(father2), hijo1,
                                hijo2);
                break;

        case AlgoritmoConstantes.CRUCE_1_PUNTO_SELECTIVO:

                hominum.cruce1PuntoSelectivo(hominum.getPopulation()
                                .get(father1),
                                hominum.getPopulation().get(father2), hijo1,
                                hijo2, straightLineProgram);
                break;
```

```
case AlgoritmoConstantes.CRUCE_UNIFORME:

        hominum.cruceUniforme(
                        hominum.getPopulation().get(father1), hominum
                                        .getPopulation().get(father2), hijo1,
                        hijo2);
        break;

case AlgoritmoConstantes.CRUCE_UNIFORME_SELECTIVO:

        hominum.cruceUniformeSelectivo(hominum.getPopulation()
                        .get(father1),
                        hominum.getPopulation().get(father2), hijo1,
                        hijo2, straightLineProgram);
        break;
}

// 3.4.3) Determinar el tipo de insercion

switch (this.replacement) {
case AlgoritmoConstantes.REPLACE_HIJOS:

        if (nIndividuosNuevos >= tamPoblacion) {

                break Continua;
        }

        else
                nIndividuosNuevos++;
```

```
                // Se inserta el primer hijo
                hominumAux.getPopulation().add(hijo1);

                if (nIndividuosNuevos >= tamPoblacion) {

                        break Continua;
                }

                else
                        nIndividuosNuevos++;

                // Se inserta el segundo hijo
                hominumAux.getPopulation().add(hijo2);

                break;

        case AlgoritmoConstantes.REPLACE_MEJORES:

                // Poblacion Auxiliar
                Poblacion k = new Poblacion();
                k.getPopulation().add(
                                hominum.getPopulation().get(father1));
                k.getPopulation().add(
                                hominum.getPopulation().get(father2));
                k.getPopulation().add(hijo1);
                k.getPopulation().add(hijo2);

                k.calculaFitnessPoblacion(this.ruta, this.nVariables,
                                straightLineProgram);
```

```
// Vector que almacenara los fitness de padres e hijos
Vector<Double> f = new Vector<Double>();
f.add(hominum.getPopulation().get(father1).getFitness());
f.add(hominum.getPopulation().get(father2).getFitness());
f.add(hijo1.getFitness());
f.add(hijo2.getFitness());

// Variable que almacenara el mejor de padres e hijos
double mejor;
mejor = Collections.min(f);

for (int i = 0; i < k.getPopulation().size(); i++) {
	if (k.getPopulation().get(i).getFitness() == mejor)
		;
	{
		if (nIndividuosNuevos >= this.tamPoblacion) {

			break Continua;
		}

		else
			nIndividuosNuevos++;

		hominumAux.getPopulation().add(
				k.getPopulation().get(i));
	}
}

// Eliminacion del vector de fitness el mejor
```

```
                f.remove(mejor);

                while (f.size() > 3)
                        f.removeElement(f.lastElement());

                // Calculo del mejor de entre el padre
                mejor = Collections.min(f);

                for (int i = 0; i < k.getPopulation().size(); i++) {
                        if (k.getPopulation().get(i).getFitness() == mejor)
                                ;
                        {
                                if (nIndividuosNuevos >= this.tamPoblacion) {

                                        break Continua;
                                }

                                else
                                        nIndividuosNuevos++;

                                hominumAux.getPopulation().add(
                                                k.getPopulation().get(i));
                        }
                }

                break;
        }

}
```

```
// 3.5) Determinar si se produce mutacion

if (probability <= this.mutationProb) {
        // 3.5.1) Determinar el tipo de mutacion

        switch (mutation) {
        case AlgoritmoConstantes.MUTACION_NORMAL:

                // Se muta el individuo
                hominum.mutacion(hominum.getPopulation().get(fatherM));

                if (nIndividuosNuevos >= this.tamPoblacion) {

                        break Continua;
                }

                else
                        nIndividuosNuevos++;

                // Se inserta en la nueva poblacion
                hominumAux.getPopulation().add(
                                hominum.getPopulation().get(fatherM));

                break;

        case AlgoritmoConstantes.MUTACION_SESGADA:

                // Se muta el individuo
                hominum.mutacionSesgada(hominum.getPopulation().get(
                                fatherM));
```

```
                        if (nIndividuosNuevos >= this.tamPoblacion) {

                                break Continua;
                        }

                        else
                                nIndividuosNuevos++;

                        // Se inserta en la nueva poblacion
                        hominumAux.getPopulation().add(
                                        hominum.getPopulation().get(fatherM));

                        break;
                }
        }
}
int diferencia = 0;

if (hominumAux.getPopulation().size() > hominum.getPopulation()
                .size())
        diferencia = hominumAux.getPopulation().size()
                        - hominum.getPopulation().size();

for (int i = 0; i < diferencia; i++)
        hominumAux.getPopulation().remove(i);

hominumAux.calculaFitnessPoblacion(this.ruta, this.nVariables,
                straightLineProgram);
```

```
                // Se aumenta el contador de generaciones
                contador++;

                // Se asigna la poblacion auxiliar nueva a la anterior
                hominum = hominumAux;

        }

        // Se obtiene el mejor individuo de la poblacion final
        Individuo mejor = hominum.mejorIndividuo();

        // Se muestra el error final cometido en el algoritmo, que no es mas que
        // el
        // fitness de ese mejor individuo
        System.out.println("Fitness del Mejor Individuo (Error cometido) : ");
        System.out.println(mejor.getFitness());

        System.out.println("Presione una tecla para finalizar.");

        // Generacion de graficas si el ejercicio tiene 1 o 2 variables
        if ((this.nVariables == 1) || (this.nVariables == 2)) {
                generaSalida(straightLineProgram, mejor);
        }

        return mejor.getFitness();
}

public void generaSalida(SLP slp, Individuo ind) {
        String rutaRelativa = Utilidades.conseguirRutaRelativa(this.ruta);
```

```
try {
        // Puntos muestra
        FileReader file = new FileReader(this.ruta);
        BufferedReader fileStream = new BufferedReader(file);

        // Puntos Salida
        // Ahora se abre el archivo con esa ruta
        PrintWriter puntosMuestraSalida = new PrintWriter(rutaRelativa
                        + "p_salida.dat");

        if (fileStream.ready()) {
                // Se procesa la primera componente
                String temp = fileStream.readLine();

                while (temp != null) {
                        temp = temp.replaceAll("\t", " ");

                        // Vector que almacenara las cada conjunto de componentes
                        Vector<Double> componentes = new Vector<Double>();

                        // Variable que recogera el output del individuo i asociado
                        // a Ui
                        double output;

                        String[] contenido = temp.split(" ");
                        String salida = "";
                        for (int j = 0; j < nVariables; j++) {
                                componentes.add(j, Double.parseDouble(contenido[j]));
                                salida = salida + contenido[j] + "\t";
                        }
```

```
                                output = slp.calculaOutput(componentes,
                                                ind.getOperacionEscalar(),
                                                ind.getParametrosReales());

                                // Se escribe el valor del outut (que es el valor
                                // aproximado)
                                salida = salida + output;

                                puntosMuestraSalida.println(salida);

                                // Se lee el siguiente conjunto de puntos
                                temp = fileStream.readLine();
                        }

                        fileStream.close();
                        puntosMuestraSalida.flush();
                        puntosMuestraSalida.close();
                }
        } catch (Exception e) {
                e.printStackTrace();
        }

}

private Poblacion redimensiona(Poblacion original, Poblacion auxiliar) {
        return original;
}

}
```

5.6 tfm.genetics.panel.base.PanelPrincipalConstantes

```
package tfm.genetics.panel.base;

public class PanelPrincipalConstantes {

	/**
	 * Variable que notificará a la clase principal cuando el algoritmo está listo para ejecutarse
	 */
	public static final String ALGORITMO_LISTO = "readyForGenetic";

	public static final String TITULO_PRINCIPAL = "Regresión Simbólica";

	public static final String RESUMEN_PRINCIPAL = "Este proyecto aborda la Regresion Simbólica mediante algoritmos" +
			" genéticos. Se manejaran para la estructura de datos dos conjuntos: \n" +
			"\t - V (numero de variables)\n" +
			"\t - F (operaciones): viene predefinido y manejará las operaciones F = {+, -, /, *}.\n\n" +
			"Para empezar a trabajar vaya rellenando y seleccionando los parámetros que desee para la realización del problema:";

	public static final String NUMERO_VARIABLES = "Nº. variables:";

	public static final String NUMERO_MAXIMO_OPERACIONES = "Nº. máximo de operaciones distintas de {+, -}:";

	public static final String TAMANIO_POBLACION = "Tamaño población:";

	public static final String RANGO_MIN_MAX = "Rango [min, max]";

	public static final String RANGO_VALORES1 = "Rango [min, max] de valores que podrán tomar los parámetros escalares de cada
individuo" + " de la población:  [";
```

```
public static final String RANGO_VALORES2 = "   ,  ";

public static final String RANGO_VALORES3 = "   ]";

public static final String RUTA_PUNTOS_MUESTRA = "Ruta del fichero de los puntos muestra originales:";

public static final String RUTA_PUNTOS_MUESTRA_ABREVIADO = "Ruta del fichero";

public static final String TIPO_SELECCION = "Defina el tipo de selección que se aplicará a los individuos de la " +
                "población para su evolución:";

public static final String K_TORNEO_SELECCION = "Defina el valor de K (< Tamaño Población):";

public static final String K_TORNEO = "K-Torneo";

public static final String RULETA = "Ruleta";

public static final String TIPO_CRUCE = "Defina el tipo de cruce al que se someterán los individuos de la población:";

public static final String CRUCE_1_PUNTO = "Cruce en 1 punto";

public static final String CRUCE_1_PUNTO_SELECTIVO = "Cruce en 1 punto selectivo";

public static final String CRUCE_UNIFORME = "Cruce uniforme";

public static final String CRUCE_UNIFORME_SELECTIVO = "Cruce uniforme selectivo";

public static final String TIPO_MUTACION = "Escoja el tipo de mutación que sufrirán los individuos de la población:";
```

```
public static final String MUTACION_NORMAL = "Mutación normal (igual probabilidad)";

public static final String MUTACION_SESGADA = "Mutación sesgada (distinta probabilidad)";

public static final String TIPO_REEMPLAZAMIENTO = "Establezca el tipo de reemplazamiento que se llevará a cabo en la población:";

public static final String REEMPLAZAMIENTO_HIJOS = "Siempre se eligen los hijos";

public static final String REEMPLAZAMIENTO_MEJORES = "Se escogen los mejores entre padres e hijos";

public static final String PROBABILIDAD_CRUCE = "Probabilidad de cruce (entre [0, 1]):";

public static final String PROBABILIDAD_MUTACION = "Probabilidad de mutación (entre [0, 1]):";

public static final String ALGORITMO_ELITISTA = "¿El algoritmo genético es Elitista? El mejor individuo de la población se introduce
" + "en cada generación/iteración:";

public static final String TEST_PARADA = "Establezca el test de parada (nº de iteraciones) del algoritmo genético:";

public static final String TEST_PARADA_ABREVIADO = "Test Parada";

public static final String TEST_RESULTADO_ALGORITMO = "El algoritmo genético ha obtenido el mejor individuo con valor fitness
(error cometido): ";

//////////////////////////////////////////////////////////////
/////////////// Constantes Algoritmo /////////////////
//////////////////////////////////////////////////////////////
/**
 * Constantes para el cruce
 */
```

```
public static final int  CROSS_1_PUNTO = 1;

public static final int  CROSS_1_PUNTO_SELECTIVO = 2;

public static final int  CROSS_UNIFORME = 3;

public static final int  CROSS_UNIFORME_SELECTIVO = 4;

/**
 * Constantes para el tipo de seleccion
 */
public static final int  SELECTION_RULETA = 1;

public static final int  SELECTION_K_TORNEO = 2;

/**
 * Constantes para el tipo de mutacion
 */
public static final int  MUTATION_NORMAL = 1;

public static final int  MUTATION_SESGADA = 2;

/**
 * Constantes para reemplazo de indiviuos
 */
public static final int  REPLACE_HIJOS = 1;

public static final int  REPLACE_MEJORES = 2;

/**
```

```
         * Comando para mostrar las gráficas a traves del GNUPlot
         */
        public static final String COMANDO_GNUPLOT = "/gnuplot/gnuplot";
}
```

5.7 tfm.genetics.panel.base.PanelPrincipalBase

```
package tfm.genetics.panel.base;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.Observable;

import javax.swing.ButtonGroup;
import javax.swing.JButton;
```

```
import javax.swing.JCheckBox;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JProgressBar;
import javax.swing.JRadioButton;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.event.CaretEvent;
import javax.swing.event.CaretListener;

import tfm.genetics.utilities.Utilidades;

public class PanelPrincipalBase extends Observable implements ActionListener {
	private JFrame interfaz;
	private JTextArea resumen;

	// Numero Variables
	private JLabel variablesLabel;
	private JTextField variables;

	// Numero operaciones
	private JLabel nMaxOperacionesLabel;
	private JTextField nMaxOperaciones;

	// Tamanio poblacion
	private JLabel tamPoblacionLabel;
	private JTextField tamPoblacion;
```

```
// Rango Valores
private JLabel rangoValoresLabel1;
private JLabel rangoValoresLabel2;
private JLabel rangoValoresLabel3;
private JTextField rangoValorMin;
private JTextField rangoValorMax;

// Ruta Puntos Muestra
private JLabel rutaLabel;
private JTextField ruta;
private JButton selectFile;

// Seleccion
private JLabel tipoSeleccion;
private JLabel ktorneoLabel;
private JTextField ktorneo;
private JRadioButton ruletaRadio;
private JRadioButton kTorneoRadio;

// Cruce
private JLabel tipoCruce;
private JRadioButton cruce1Punto;
private JRadioButton cruce1PuntoSelectivo;
private JRadioButton cruceUniforme;
private JRadioButton cruceUniformeSelectivo;

// Mutacion
private JLabel tipoMutacion;
private JRadioButton mutacionNormalRadio;
private JRadioButton mutacionSesgadaRadio;
```

```
// Reemplazamiento
private JLabel tipoReemplazo;
private JRadioButton reemplazoHijos;
private JRadioButton reemplazoMejores;

// Probabilidad de Cruce
private JLabel probabilidadCruceLabel;
private JTextField probabilidadCruce;

// Probabilidad de Mutacion
private JLabel probabilidadMutacionLabel;
private JTextField probabilidadMutacion;

// Elitismo
private JLabel elitismoLabel;
private JCheckBox elitismo;

// Test de parada
private JLabel testParadaLabel;
private JTextField testParada;

// Controles
private JButton startButton;
private JButton endButton;
private JButton iniButton;
private JProgressBar barraProgreso;

// Grupo Botones Cruce
ButtonGroup botonesSeleccion;
```

```
// Grupo Botones Cruce
ButtonGroup botonesCruce;
// Grupo Botones Cruce
ButtonGroup botonesMutacion;
// Grupo Botones Cruce
ButtonGroup botonesReemplazamiento;

// Resultado Algoritmo
private JLabel resultadoAlgoritmo;

/**
 * Variables propias para el Algoritmo Genético
 */
// Numero de variables
private int nVariables = 0;

// Numero de operaciones no escalares (tamanio eficaz)
private int L = 0;

// Tamanio de la poblacion
private int tamanioPoblacion = 0;

// Rango minimo
private double lower = 0;

// Rango maximo
private double upper = 0;

// Ruta del fichero
private String route = null;
```

```
// Tipo de seleccion
private int selection = 0;

// Tipo de cruce
private int cross = 0;

// Tipo de mutacion
private int mutation = 0;

// Tipo de reemplazamiento
private int replacement = 0;

// Probabilidad de cruce
private double crossProb = 0;

// Probabilidad de mutacion
private double mutationProb = 0;

// Test de parada
private int test = 0;

// Variable kTorneo
private int kTorneo = 0;

// Elitismo
private boolean elitism = false;

// Campos vacíos
private ArrayList<String> camposVacios = null;
```

```
public PanelPrincipalBase ()
{
	super();
	inicializar();
}


public void inicializar()
{
	// Creación de límites del panel principal
	interfaz = new JFrame(PanelPrincipalConstantes.TITULO_PRINCIPAL);
	interfaz.setSize(785,825);
	interfaz.getContentPane().add(getResumen(), BorderLayout.NORTH);

	// Resumen
	interfaz.add(getResumen());

	// Variables
	interfaz.add(getVariablesLabel(), null);
	interfaz.add(getVariables(), null);

	// Operaciones
	interfaz.add(getMaxOperacionesLabel(), null);
	interfaz.add(getMaxOperaciones(), null);

	// Tamanio Poblacion
	interfaz.add(getTamPoblacionLabel(), null);
	interfaz.add(getTamPoblacion(), null);
```

```
// Rango Valores
interfaz.add(getRangoValoresLabel1(), null);
interfaz.add(getRangoValorMin(), null);
interfaz.add(getRangoValoresLabel2(), null);
interfaz.add(getRangoValorMax(), null);
interfaz.add(getRangoValoresLabel3(), null);

// Ruta archivo de puntos muestra
interfaz.add(getRutaLabel(), null);
interfaz.add(getRuta(), null);
interfaz.add(getSelectFile(), null);

// Tipo de seleccion
interfaz.add(getTipoSeleccion(), null);
interfaz.add(getRuletaRadio(), null);
interfaz.add(getKTorneoRadio(), null);
interfaz.add(getKTorneoLabel(), null);
interfaz.add(getKTorneo(), null);
creaGrupoSeleccion();

// Tipo de cruce
interfaz.add(getTipoCruce(), null);
interfaz.add(getCruce1Punto(), null);
interfaz.add(getCruce1PuntoSelectivo(), null);
interfaz.add(getCruceUniforme(), null);
interfaz.add(getCruceUniformeSelectivo(), null);
creaGrupoCruce();

// Probabilidad de cruce
```

```
interfaz.add(getProbabilidadCruceLabel(), null);
interfaz.add(getProbabilidadCruce(), null);

// Tipo de mutacion
interfaz.add(getTipoMutacion(), null);
interfaz.add(getMutacionNormalRadio(), null);
interfaz.add(getMutacionSesgadaRadio(), null);
creaGrupoMutacion();

// Probabilidad de mutacion
interfaz.add(getProbabilidadMutacionLabel(), null);
interfaz.add(getProbabilidadMutacion(), null);

// Tipo de Reemplazo
interfaz.add(getTipoReemplazamiento(), null);
interfaz.add(getReemplazamientoHijos(), null);
interfaz.add(getReemplazamientoMejores(), null);
creaGrupoReemplazamiento();

// Elitismo
interfaz.add(getElitismoLabel(), null);
interfaz.add(getElitismo(), null);

// Elitismo
interfaz.add(getTestParadaLabel(), null);
interfaz.add(getTestParada(), null);

// Controles
interfaz.add(getBotonStart(), null);
interfaz.add(getBotonEnd(), null);
```

```
        interfaz.add(getIniButton(), null);
        interfaz.add(getBarraProgreso(), null);
        interfaz.add(getResultadoAlgoritmo(), null);

        // Se crean grupos de botones para la selección de cada tipo
        // de operacion
        botonesSeleccion = new ButtonGroup();
        botonesSeleccion.add(getRuletaRadio());
        botonesSeleccion.add(getKTorneoRadio());

        botonesCruce = new ButtonGroup();
        botonesCruce.add(getCruce1Punto());
        botonesCruce.add(getCruce1PuntoSelectivo());
        botonesCruce.add(getCruceUniforme());
        botonesCruce.add(getCruceUniformeSelectivo());

        botonesMutacion = new ButtonGroup();
        botonesMutacion.add(getMutacionNormalRadio());
        botonesMutacion.add(getMutacionSesgadaRadio());

        botonesReemplazamiento = new ButtonGroup();
        botonesReemplazamiento.add(getReemplazamientoHijos());
        botonesReemplazamiento.add(getReemplazamientoMejores());
}


public void arrancar ()
{
        interfaz.setLocationRelativeTo(null);
        interfaz.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
		interfaz.setLayout(null);
		interfaz.setVisible(true);
}

/**
 * Creación de los distintos componentes que se verán en
 * el panel principal
 */
private JTextArea getResumen() {
		if (resumen == null) {
				resumen = new JTextArea(PanelPrincipalConstantes.RESUMEN_PRINCIPAL);
				resumen.setFont(new Font("Serif", Font.PLAIN, 13));
				resumen.setOpaque(false);
				resumen.setEditable(false);
				resumen.setBounds(30, 20, 750, 90);
				resumen.setLineWrap(true);
		}
		return resumen;
}


private JLabel getVariablesLabel() {
		if (variablesLabel == null) {
				variablesLabel = new JLabel();
				variablesLabel.setText(PanelPrincipalConstantes.NUMERO_VARIABLES);
				variablesLabel.setFont(new Font("Serif", Font.PLAIN, 13));
				variablesLabel.setBounds(30, 140, 90, 15);
				variablesLabel.setName("variablesLabel");
		}
		return variablesLabel;
```

```
}


private JTextField getVariables() {
	if (variables == null) {
		variables = new JTextField();
		variables.setBounds(110, 138, 40, 20);
		variables.setName("variables");
		variables.setEnabled(true);
		variables.setAutoscrolls(true);
		variables.addCaretListener(new CaretListener() {
			public void caretUpdate(CaretEvent e) {
				try{
					if (!getVariables().getText().toString().equalsIgnoreCase(""))
						nVariables = Integer.valueOf(getVariables().getText().toString());
				}catch (Exception ex) {}
			}
		});
	}
	return variables;
}


private JLabel getMaxOperacionesLabel() {
	if (nMaxOperacionesLabel == null) {
		nMaxOperacionesLabel = new JLabel();
		nMaxOperacionesLabel.setText(PanelPrincipalConstantes.NUMERO_MAXIMO_OPERACIONES);
		nMaxOperacionesLabel.setFont(new Font("Serif", Font.PLAIN, 13));
		nMaxOperacionesLabel.setBounds(175, 140, 250, 15);
		nMaxOperacionesLabel.setName("nMaxOperacionesLabel");
```

```
		}
		return nMaxOperacionesLabel;
	}


	private JTextField getMaxOperaciones() {
		if (nMaxOperaciones == null) {
			nMaxOperaciones = new JTextField();
			nMaxOperaciones.setBounds(430, 138, 40, 20);
			nMaxOperaciones.setName("variables");
			nMaxOperaciones.setEnabled(true);
			nMaxOperaciones.setAutoscrolls(true);
			nMaxOperaciones.addCaretListener(new CaretListener() {
				public void caretUpdate(CaretEvent e) {
					try{
						if (!getMaxOperaciones().getText().toString().equalsIgnoreCase(""))
							L = Integer.valueOf(getMaxOperaciones().getText().toString());
					}catch (Exception ex) {}
				}
			});
		}
		return nMaxOperaciones;
	}


	private JLabel getTamPoblacionLabel() {
		if (tamPoblacionLabel == null) {
			tamPoblacionLabel = new JLabel();
			tamPoblacionLabel.setText(PanelPrincipalConstantes.TAMANIO_POBLACION);
			tamPoblacionLabel.setFont(new Font("Serif", Font.PLAIN, 13));
```

```
			tamPoblacionLabel.setBounds(525, 140, 125, 15);
			tamPoblacionLabel.setName("tPoblacionLabel");
		}
		return tamPoblacionLabel;
}


private JTextField getTamPoblacion() {
		if (tamPoblacion == null) {
			tamPoblacion = new JTextField();
			tamPoblacion.setBounds(640, 138, 40, 20);
			tamPoblacion.setName("tPoblacion");
			tamPoblacion.setEnabled(true);
			tamPoblacion.setAutoscrolls(true);
			tamPoblacion.addCaretListener(new CaretListener() {
				public void caretUpdate(CaretEvent e) {
					try{
						if (!getTamPoblacion().getText().toString().equalsIgnoreCase(""))
							tamanioPoblacion = Integer.valueOf(getTamPoblacion().getText().toString());
					}catch (Exception ex) {}
				}
			});
		}
		return tamPoblacion;
}


private JLabel getRangoValoresLabel1() {
		if (rangoValoresLabel1 == null) {
			rangoValoresLabel1 = new JLabel();
```

```
                rangoValoresLabel1.setText(PanelPrincipalConstantes.RANGO_VALORES1);
                rangoValoresLabel1.setFont(new Font("Serif", Font.PLAIN, 13));
                rangoValoresLabel1.setBounds(30, 160, 700, 50);
                rangoValoresLabel1.setName("rangoValoresLabel1");
        }
        return rangoValoresLabel1;
}


private JLabel getRangoValoresLabel2() {
        if (rangoValoresLabel2 == null) {
                rangoValoresLabel2 = new JLabel();
                rangoValoresLabel2.setText(PanelPrincipalConstantes.RANGO_VALORES2);
                rangoValoresLabel2.setFont(new Font("Serif", Font.PLAIN, 13));
                rangoValoresLabel2.setBounds(650, 160, 20, 50);
                rangoValoresLabel2.setName("rangoValoresLabel2");
        }
        return rangoValoresLabel2;
}


private JLabel getRangoValoresLabel3() {
        if (rangoValoresLabel3 == null) {
                rangoValoresLabel3 = new JLabel();
                rangoValoresLabel3.setText(PanelPrincipalConstantes.RANGO_VALORES3);
                rangoValoresLabel3.setFont(new Font("Serif", Font.PLAIN, 13));
                rangoValoresLabel3.setBounds(695, 160, 20, 50);
                rangoValoresLabel3.setName("rangoValoresLabel3");
        }
        return rangoValoresLabel3;
```

```
}


private JTextField getRangoValorMin() {
	if (rangoValorMin == null) {
		rangoValorMin = new JTextField();
		rangoValorMin.setBounds(617, 176, 40, 20);
		rangoValorMin.setName("rangoValorMin");
		rangoValorMin.setEnabled(true);
		rangoValorMin.setAutoscrolls(true);
		rangoValorMin.addCaretListener(new CaretListener() {
			public void caretUpdate(CaretEvent e) {
				try{
					if (!getRangoValorMin().getText().toString().equalsIgnoreCase(""))
						lower = Double.valueOf(getRangoValorMin().getText().toString());
				}catch (Exception ex) {}
			}
		});
	}
	return rangoValorMin;
}


private JTextField getRangoValorMax() {
	if (rangoValorMax == null) {
		rangoValorMax = new JTextField();
		rangoValorMax.setBounds(660, 176, 40, 20);
		rangoValorMax.setName("rangoValorMax");
		rangoValorMax.setEnabled(true);
		rangoValorMax.setAutoscrolls(true);
```

```
                rangoValorMax.addCaretListener(new CaretListener() {
                        public void caretUpdate(CaretEvent e) {
                                try{
                                        if (!getRangoValorMax().getText().toString().equalsIgnoreCase(""))
                                                upper = Double.valueOf(getRangoValorMax().getText().toString());
                                }catch (Exception ex) {}
                        }
                });
        }
        return rangoValorMax;
}


private JLabel getRutaLabel() {
        if (rutaLabel == null) {
                rutaLabel = new JLabel();
                rutaLabel.setText(PanelPrincipalConstantes.RUTA_PUNTOS_MUESTRA);
                rutaLabel.setFont(new Font("Serif", Font.PLAIN, 13));
                rutaLabel.setBounds(30, 220, 400, 15);
                rutaLabel.setName("rangoValoresLabel3");
        }
        return rutaLabel;
}


private JTextField getRuta() {
        if (ruta == null) {
                ruta = new JTextField();
                ruta.setBounds(300, 218, 300, 20);
                ruta.setName("ruta");
```

```
                ruta.setEnabled(true);
                ruta.setAutoscrolls(true);
                ruta.addCaretListener(new CaretListener() {
                        public void caretUpdate(CaretEvent e) {
                                try{
                                        if (!getRuta().getText().toString().equalsIgnoreCase(""))
                                                route = getRuta().getText().toString();
                                }catch (Exception ex) {}
                        }
                });
        }
        return ruta;
}


private JButton getSelectFile() {
        if (selectFile == null) {
                selectFile = new JButton();
                selectFile.setBounds(610, 218, 107, 19);
                selectFile.setPreferredSize(new Dimension(50,50));
                selectFile.setText("Elige Archivo");
                selectFile.setName("selectFile");
                selectFile.addActionListener(this);
        }
        return selectFile;
}


private JLabel getTipoSeleccion() {
        if (tipoSeleccion == null) {
```

```
			tipoSeleccion = new JLabel();
			tipoSeleccion.setText(PanelPrincipalConstantes.TIPO_SELECCION);
			tipoSeleccion.setFont(new Font("Serif", Font.PLAIN, 13));
			tipoSeleccion.setBounds(30, 260, 700, 15);
			tipoSeleccion.setName("tipoSeleccion");
		}
		return tipoSeleccion;
}



private JLabel getKTorneoLabel() {
		if (ktorneoLabel == null) {
			ktorneoLabel = new JLabel();
			ktorneoLabel.setEnabled(true);
			ktorneoLabel.setVisible(false);
			ktorneoLabel.setText(PanelPrincipalConstantes.K_TORNEO_SELECCION);
			ktorneoLabel.setFont(new Font("Serif", Font.ITALIC, 12));
			ktorneoLabel.setForeground(Color.BLUE);
			ktorneoLabel.setBounds(175, 315, 300, 15);
			ktorneoLabel.setName("ktorneo");
		}
		return ktorneoLabel;
}



private JTextField getKTorneo() {
		if (ktorneo == null) {
			ktorneo = new JTextField();
			ktorneo.setBounds(400, 313, 40, 20);
```

```
                ktorneo.setName("ktorneo");
                ktorneo.setEnabled(true);
                ktorneo.setVisible(false);
                ktorneo.setAutoscrolls(true);
                ktorneo.addCaretListener(new CaretListener() {
                        public void caretUpdate(CaretEvent e) {
                                try{
                                        if (!getKTorneo().getText().toString().equalsIgnoreCase(""))
                                                kTorneo = Integer.valueOf(getKTorneo().getText().toString());
                                }catch (Exception ex) {}
                        }
                });
        }
        return ktorneo;
}


private JRadioButton getRuletaRadio() {
        if (ruletaRadio == null) {
                ruletaRadio = new JRadioButton(PanelPrincipalConstantes.RULETA);
                ruletaRadio.setFont(new Font("Serif", Font.ITALIC, 13));
                ruletaRadio.setActionCommand("ruleta");
                ruletaRadio.setBounds(50, 290, 150, 17);
                ruletaRadio.setSelected(false);
                ruletaRadio.addActionListener(this);
        }
        return ruletaRadio;
}
```

```
private JRadioButton getKTorneoRadio() {
	if (kTorneoRadio == null) {
		kTorneoRadio = new JRadioButton(PanelPrincipalConstantes.K_TORNEO);
		kTorneoRadio.setFont(new Font("Serif", Font.ITALIC, 13));
		kTorneoRadio.setActionCommand("kTorneo");
		kTorneoRadio.setBounds(50, 315, 100, 17);
		kTorneoRadio.setSelected(false);
		kTorneoRadio.addActionListener(this);
	}
	return kTorneoRadio;
}


private JLabel getTipoCruce() {
	if (tipoCruce == null) {
		tipoCruce = new JLabel();
		tipoCruce.setText(PanelPrincipalConstantes.TIPO_CRUCE);
		tipoCruce.setFont(new Font("Serif", Font.PLAIN, 13));
		tipoCruce.setBounds(30, 355, 415, 17);
		tipoCruce.setName("tipoCruce");
	}
	return tipoCruce;
}


private JRadioButton getCruce1Punto() {
	if (cruce1Punto == null) {
		cruce1Punto = new JRadioButton(PanelPrincipalConstantes.CRUCE_1_PUNTO);
		cruce1Punto.setFont(new Font("Serif", Font.ITALIC, 13));
		cruce1Punto.setActionCommand("cruce1Punto");
```

```
                cruce1Punto.setBounds(50, 385, 150, 17);
                cruce1Punto.setSelected(false);
                cruce1Punto.addActionListener(this);

        }
        return cruce1Punto;
}


private JRadioButton getCruce1PuntoSelectivo() {
        if (cruce1PuntoSelectivo == null) {
                cruce1PuntoSelectivo = new JRadioButton(PanelPrincipalConstantes.CRUCE_1_PUNTO_SELECTIVO);
                cruce1PuntoSelectivo.setFont(new Font("Serif", Font.ITALIC, 13));
                cruce1PuntoSelectivo.setActionCommand("cruce1PuntoSelectivo");
                cruce1PuntoSelectivo.setBounds(50, 410, 200, 17);
                cruce1PuntoSelectivo.setSelected(false);
                cruce1PuntoSelectivo.addActionListener(this);
        }
        return cruce1PuntoSelectivo;
}


private JRadioButton getCruceUniforme() {
        if (cruceUniforme == null) {
                cruceUniforme = new JRadioButton(PanelPrincipalConstantes.CRUCE_UNIFORME);
                cruceUniforme.setFont(new Font("Serif", Font.ITALIC, 13));
                cruceUniforme.setActionCommand("cruceUniforme");
                cruceUniforme.setBounds(300, 385, 150, 17);
                cruceUniforme.setSelected(false);
                cruceUniforme.addActionListener(this);
```

```
		}
		return cruceUniforme;
}


private JRadioButton getCruceUniformeSelectivo() {
		if (cruceUniformeSelectivo == null) {
			cruceUniformeSelectivo = new JRadioButton(PanelPrincipalConstantes.CRUCE_UNIFORME_SELECTIVO);
			cruceUniformeSelectivo.setFont(new Font("Serif", Font.ITALIC, 13));
			cruceUniformeSelectivo.setActionCommand("cruceUniformeSelectivo");
			cruceUniformeSelectivo.setBounds(300, 410, 175, 17);
			cruceUniformeSelectivo.setSelected(false);
			cruceUniformeSelectivo.addActionListener(this);
		}
		return cruceUniformeSelectivo;
}


private JLabel getTipoMutacion() {
		if (tipoMutacion == null) {
			tipoMutacion = new JLabel();
			tipoMutacion.setText(PanelPrincipalConstantes.TIPO_MUTACION);
			tipoMutacion.setFont(new Font("Serif", Font.PLAIN, 13));
			tipoMutacion.setBounds(30, 450, 700, 15);
			tipoMutacion.setName("tipoMutacion");
		}
		return tipoMutacion;
}
```

```
private JRadioButton getMutacionNormalRadio() {
	if (mutacionNormalRadio == null) {
		mutacionNormalRadio = new JRadioButton(PanelPrincipalConstantes.MUTACION_NORMAL);
		mutacionNormalRadio.setFont(new Font("Serif", Font.ITALIC, 13));
		mutacionNormalRadio.setActionCommand("mutacionNormal");
		mutacionNormalRadio.setBounds(50, 480, 250, 17);
		mutacionNormalRadio.setSelected(false);
		mutacionNormalRadio.addActionListener(this);
	}
	return mutacionNormalRadio;
}


private JRadioButton getMutacionSesgadaRadio() {
	if (mutacionSesgadaRadio == null) {
		mutacionSesgadaRadio = new JRadioButton(PanelPrincipalConstantes.MUTACION_SESGADA);
		mutacionSesgadaRadio.setFont(new Font("Serif", Font.ITALIC, 13));
		mutacionSesgadaRadio.setActionCommand("mutacionSesgadaRadio");
		mutacionSesgadaRadio.setBounds(50, 505, 275, 17);
		mutacionSesgadaRadio.setSelected(false);
		mutacionSesgadaRadio.addActionListener(this);
	}
	return mutacionSesgadaRadio;
}


private JLabel getTipoReemplazamiento() {
	if (tipoReemplazo == null) {
		tipoReemplazo = new JLabel();
		tipoReemplazo.setText(PanelPrincipalConstantes.TIPO_REEMPLAZAMIENTO);
```

```
                tipoReemplazo.setFont(new Font("Serif", Font.PLAIN, 13));
                tipoReemplazo.setBounds(30, 545, 700, 15);
                tipoReemplazo.setName("tipoReemplazo");
        }
        return tipoReemplazo;
}


private JRadioButton getReemplazamientoHijos() {
        if (reemplazoHijos == null) {
                reemplazoHijos = new JRadioButton(PanelPrincipalConstantes.REEMPLAZAMIENTO_HIJOS);
                reemplazoHijos.setFont(new Font("Serif", Font.ITALIC, 13));
                reemplazoHijos.setActionCommand("reemplazoHijos");
                reemplazoHijos.setBounds(50, 575, 250, 17);
                reemplazoHijos.setSelected(false);
                reemplazoHijos.addActionListener(this);
        }
        return reemplazoHijos;
}


private JRadioButton getReemplazamientoMejores() {
        if (reemplazoMejores == null) {
                reemplazoMejores = new JRadioButton(PanelPrincipalConstantes.REEMPLAZAMIENTO_MEJORES);
                reemplazoMejores.setFont(new Font("Serif", Font.ITALIC, 13));
                reemplazoMejores.setActionCommand("reemplazoMejores");
                reemplazoMejores.setBounds(50, 600, 275, 17);
                reemplazoMejores.setSelected(false);
                reemplazoMejores.addActionListener(this);
        }
```

```
		return reemplazoMejores;
}


private JLabel getProbabilidadCruceLabel() {
		if (probabilidadCruceLabel == null) {
			probabilidadCruceLabel = new JLabel();
			probabilidadCruceLabel.setVisible(false);
			probabilidadCruceLabel.setText(PanelPrincipalConstantes.PROBABILIDAD_CRUCE);
			probabilidadCruceLabel.setFont(new Font("Serif", Font.ITALIC, 12));
			probabilidadCruceLabel.setForeground(Color.BLUE);
			probabilidadCruceLabel.setBounds(500, 395, 200, 15);
			probabilidadCruceLabel.setName("probabilidadCruceLabel");
		}
		return probabilidadCruceLabel;
}


private JTextField getProbabilidadCruce() {
		if (probabilidadCruce == null) {
			probabilidadCruce = new JTextField();
			probabilidadCruce.setBounds(695, 392, 40, 20);
			probabilidadCruce.setName("probabilidadCruce");
			probabilidadCruce.setEnabled(true);
			probabilidadCruce.setVisible(false);
			probabilidadCruce.setAutoscrolls(true);
			probabilidadCruce.addCaretListener(new CaretListener() {
				public void caretUpdate(CaretEvent e) {
					try{
						if (!getProbabilidadCruce().getText().toString().equalsIgnoreCase(""))
```

```
                                        crossProb = Double.valueOf(getProbabilidadCruce().getText().toString());
                                }catch (Exception ex) {}

                        }
                });
        }
        return probabilidadCruce;
}


private JLabel getProbabilidadMutacionLabel() {
        if (probabilidadMutacionLabel == null) {
                probabilidadMutacionLabel = new JLabel();
                probabilidadMutacionLabel.setVisible(false);
                probabilidadMutacionLabel.setText(PanelPrincipalConstantes.PROBABILIDAD_MUTACION);
                probabilidadMutacionLabel.setFont(new Font("Serif", Font.ITALIC, 12));
                probabilidadMutacionLabel.setForeground(Color.BLUE);
                probabilidadMutacionLabel.setBounds(375, 490, 200, 15);
                probabilidadMutacionLabel.setName("probabilidadMutacionLabel");
        }
        return probabilidadMutacionLabel;
}


private JTextField getProbabilidadMutacion() {
        if (probabilidadMutacion == null) {
                probabilidadMutacion = new JTextField();
                probabilidadMutacion.setBounds(575, 487, 40, 20);
                probabilidadMutacion.setName("probabilidadMutacion");
                probabilidadMutacion.setEnabled(true);
```

```
                probabilidadMutacion.setVisible(false);
                probabilidadMutacion.setAutoscrolls(true);
                probabilidadMutacion.addCaretListener(new CaretListener() {
                        public void caretUpdate(CaretEvent e) {
                                try{
                                        if (!getProbabilidadMutacion().getText().toString().equalsIgnoreCase(""))
                                                mutationProb = Double.valueOf(getProbabilidadMutacion().getText().toString());
                                }catch (Exception ex) {}
                        }
                });
        }
        return probabilidadMutacion;
}


private JLabel getElitismoLabel() {
        if (elitismoLabel == null) {
                elitismoLabel = new JLabel();
                elitismoLabel.setText(PanelPrincipalConstantes.ALGORITMO_ELITISTA);
                elitismoLabel.setFont(new Font("Serif", Font.PLAIN, 13));
                elitismoLabel.setBounds(30, 640, 700, 15);
                elitismoLabel.setName("elitismoLabel");
        }
        return elitismoLabel;
}


private JCheckBox getElitismo() {
        if (elitismo == null) {
                elitismo = new JCheckBox();
```

```
			elitismo.setBounds(645, 638, 700, 15);
			elitismo.setVisible(true);
			elitismo.setName("elitismo");
			elitismo.addItemListener(new ItemListener() {
				public void itemStateChanged(ItemEvent e) {
					if (e.getStateChange() == ItemEvent.DESELECTED)
						elitism = false;
					if (e.getStateChange() == ItemEvent.SELECTED)
						elitism = true;
				}
			});
		}
		return elitismo;
}


private JLabel getTestParadaLabel() {
		if (testParadaLabel == null) {
			testParadaLabel = new JLabel();
			testParadaLabel.setText(PanelPrincipalConstantes.TEST_PARADA);
			testParadaLabel.setFont(new Font("Serif", Font.PLAIN, 13));
			testParadaLabel.setBounds(30, 680, 400, 15);
			testParadaLabel.setName("testParadaLabel");
		}
		return testParadaLabel;
}


private JTextField getTestParada() {
		if (testParada == null) {
```

```
                testParada = new JTextField();
                testParada.setBounds(415, 678, 40, 20);
                testParada.setName("testParada");
                testParada.setAutoscrolls(true);
                testParada.addCaretListener(new CaretListener() {
                        public void caretUpdate(CaretEvent e) {
                                try{
                                        if (!getTestParada().getText().toString().equalsIgnoreCase(""))
                                                test = Integer.valueOf(getTestParada().getText().toString());
                                }catch (Exception ex) {}
                        }
                });
        }
        return testParada;
  }


  private void creaGrupoSeleccion() {
        ButtonGroup group = new ButtonGroup();
group.add(getRuletaRadio());
group.add(getKTorneoRadio());
  }


  private void creaGrupoCruce() {
        ButtonGroup group = new ButtonGroup();
group.add(getCruce1Punto());
group.add(getCruce1PuntoSelectivo());
group.add(getCruceUniforme());
group.add(getCruceUniformeSelectivo());
```

```
    }


    private void creaGrupoMutacion() {
            ButtonGroup group = new ButtonGroup();
group.add(getMutacionNormalRadio());
group.add(getMutacionSesgadaRadio());
    }


    private void creaGrupoReemplazamiento() {
            ButtonGroup group = new ButtonGroup();
group.add(getReemplazamientoHijos());
group.add(getReemplazamientoMejores());
    }


    private JButton getBotonStart() {
            if (startButton == null) {
                    startButton = new JButton();
                    startButton.setBounds(225, 730, 100, 20);
                    startButton.setPreferredSize(new Dimension(50,50));
                    startButton.setText("Comenzar");
                    startButton.setName("ComenzarB");
                    startButton.addActionListener(this);
            }
            return startButton;
    }
```

```
private JButton getBotonEnd() {
	if (endButton == null) {
		endButton = new JButton();
		endButton.setBounds(425, 730, 100, 20);
		endButton.setText("Salir");
		endButton.setName("TerminarB");
		endButton.addActionListener(this);
	}
	return endButton;
}


private JButton getIniButton() {
	if (iniButton == null) {
		iniButton = new JButton();
		iniButton.setBounds(670, 765, 55, 20);
		iniButton.setText("Inicio");
		iniButton.setName("Inicio");
		iniButton.addActionListener(this);
		iniButton.setVisible(false);
	}
	return iniButton;
}


private JProgressBar getBarraProgreso() {
	if (barraProgreso == null) {

		barraProgreso = new JProgressBar(0, 100);
		barraProgreso.setVisible(false);
```

```
                barraProgreso.setStringPainted(true);
                barraProgreso.setIndeterminate(true);
                barraProgreso.setString("Calculando...");
                barraProgreso.setBounds(325, 730, 100, 20);

        }
        return barraProgreso;
}



private JLabel getResultadoAlgoritmo() {
        if (resultadoAlgoritmo == null) {
                resultadoAlgoritmo = new JLabel();
                resultadoAlgoritmo.setBounds(75, 730, 700, 20);
                resultadoAlgoritmo.setName("resultadoAlgoritmo");
                resultadoAlgoritmo.setEnabled(true);
                resultadoAlgoritmo.setAutoscrolls(true);
                resultadoAlgoritmo.setFont(new Font("Serif", Font.BOLD, 14));
                resultadoAlgoritmo.setForeground(Color.DARK_GRAY);
                resultadoAlgoritmo.setVisible(false);
        }
        return resultadoAlgoritmo;
}


/**
 * Función que realizará una tarea y recogerá valores
 * dependiendo del coponente que se haya visto manipulado
 * en el panel principal
```

```
 */

public void actionPerformed(ActionEvent e)
{
	Object source = e.getSource();

	if(source == getCruce1Punto() || source == getCruce1PuntoSelectivo() ||
		source == getCruceUniforme() || source == getCruceUniformeSelectivo())
	{
		getProbabilidadCruceLabel().setVisible(true);
		getProbabilidadCruceLabel().setEnabled(true);
	getProbabilidadCruce().setVisible(true);

		if (source == getCruce1Punto())
			this.cross = PanelPrincipalConstantes.CROSS_1_PUNTO;

		if (source == getCruce1PuntoSelectivo())
			this.cross = PanelPrincipalConstantes.CROSS_1_PUNTO_SELECTIVO;

		if (source == getCruceUniforme())
			this.cross = PanelPrincipalConstantes.CROSS_UNIFORME;

		if (source == getCruceUniformeSelectivo())
			this.cross = PanelPrincipalConstantes.CROSS_UNIFORME_SELECTIVO;
	}

	if (source == getSelectFile())
	{
		JFileChooser fileChooser = new JFileChooser();
		int returnValue = fileChooser.showOpenDialog(null);
```

```
                if (returnValue == JFileChooser.APPROVE_OPTION) {
                        File selectedFile = fileChooser.getSelectedFile();
                   getRuta().setText(selectedFile.getPath());
                }
}

        if (source == getRuletaRadio())
        {
                getKTorneoLabel().setVisible(false);
                getKTorneo().setVisible(false);
                this.selection = PanelPrincipalConstantes.SELECTION_RULETA;
        }

        if (source == getKTorneoRadio())
        {
                getKTorneoLabel().setVisible(true);
                getKTorneoLabel().setEnabled(true);
                getKTorneo().setEnabled(true);
                getKTorneo().setVisible(true);
                this.selection = PanelPrincipalConstantes.SELECTION_K_TORNEO;
        }

        if (source == getMutacionNormalRadio() || source == getMutacionSesgadaRadio())
        {
                getProbabilidadMutacionLabel().setVisible(true);
                getProbabilidadMutacionLabel().setEnabled(true);
                getProbabilidadMutacion().setVisible(true);

                if (source == getMutacionNormalRadio())
                        this.mutation = PanelPrincipalConstantes.MUTATION_NORMAL;
```

```
		if (source == getMutacionSesgadaRadio())
			this.mutation = PanelPrincipalConstantes.MUTATION_SESGADA;
}

if (source == getReemplazamientoHijos())
		this.replacement = PanelPrincipalConstantes.REPLACE_HIJOS;

if (source == getReemplazamientoMejores())
		this.replacement = PanelPrincipalConstantes.REPLACE_MEJORES;

if (source == getBotonStart())
{
		if (camposValidos())
		{
			deshabilitarComponentes();
			setChanged();
			notifyObservers(PanelPrincipalConstantes.ALGORITMO_LISTO);
		}
		else
		{
			String salidaError = "Las siguientes opciones/campos no están seleccionados o tienen \nvalores erróneos:\n\n";

			for (int i = 0; i < camposVacios.size(); i++)
				salidaError = salidaError + "	-	" + camposVacios.get(i) + "\n";

			salidaError = salidaError + "\nPor favor, revíselos.";
			JOptionPane.showMessageDialog(null, salidaError);
			camposVacios.clear();
			camposVacios = null;
```

```
                }
        }

        if (source == getBotonEnd())
        {
                int dialogResult = JOptionPane.showConfirmDialog(
                        null,
                        "¿Estás seguro de que quieres salir del programa?",
                        "Regresión Simbólica",
                        JOptionPane.YES_NO_OPTION);

                if(dialogResult == JOptionPane.YES_OPTION)
                        System.exit(0);
        }

        if (source == getIniButton())
        {
                int dialogResult = JOptionPane.showConfirmDialog(
                        null,
                        "¿Desea volver al inicio?",
                        "Regresión Simbólica",
                        JOptionPane.YES_NO_OPTION);

                if(dialogResult == JOptionPane.YES_OPTION)
                        restauraComponentes();
        }
}


/**
```

```
 * Funcion que deshabilita los componentes durante el proceso
 * de cálculo del algoritmo
 */
private void deshabilitarComponentes()
{
        getCruce1Punto().setEnabled(false);
        getCruce1PuntoSelectivo().setEnabled(false);
        getCruceUniforme().setEnabled(false);
        getCruceUniformeSelectivo().setEnabled(false);
        getElitismo().setEnabled(false);
        getKTorneo().setEnabled(false);
        getKTorneoLabel().setEnabled(false);
        getKTorneoRadio().setEnabled(false);
        getMaxOperaciones().setEnabled(false);
        getMutacionNormalRadio().setEnabled(false);
        getMutacionSesgadaRadio().setEnabled(false);
        getProbabilidadCruce().setEnabled(false);
        getProbabilidadCruceLabel().setEnabled(false);
        getProbabilidadMutacion().setEnabled(false);
        getProbabilidadMutacionLabel().setEnabled(false);
        getRangoValorMax().setEnabled(false);
        getRangoValorMin().setEnabled(false);
        getReemplazamientoHijos().setEnabled(false);
        getReemplazamientoMejores().setEnabled(false);
        getRuletaRadio().setEnabled(false);
        getRuta().setEnabled(false);
        getSelectFile().setEnabled(false);
        getTamPoblacion().setEnabled(false);
        getTestParada().setEnabled(false);
        getVariables().setEnabled(false);
```

```
        getBotonEnd().setVisible(false);
        getBotonStart().setVisible(false);
        getIniButton().setVisible(false);
        getBarraProgreso().setVisible(true);
        interfaz.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
}

/**
 * Función que habilitará de nuevo los componentes tras realizar el
 * proceso de cálculo de los algoritmos
 */
private void habilitarComponentes ()
{
        getResultadoAlgoritmo().setVisible(false);

        getVariables().setEnabled(true);
        getVariables().setText("");

        getMaxOperaciones().setEnabled(true);
        getMaxOperaciones().setText("");

        getTamPoblacion().setEnabled(true);
        getTamPoblacion().setText("");

        getRangoValorMax().setEnabled(true);
        getRangoValorMax().setText("");
        getRangoValorMin().setEnabled(true);
        getRangoValorMin().setText("");

        getRuta().setEnabled(true);
```

```
getRuta().setText("");
getSelectFile().setEnabled(true);

botonesSeleccion.clearSelection();
botonesCruce.clearSelection();
botonesMutacion.clearSelection();
botonesReemplazamiento.clearSelection();

getRuletaRadio().setEnabled(true);
getKTorneo().setText("");
getKTorneo().setVisible(false);
getKTorneoLabel().setVisible(false);
getKTorneoRadio().setEnabled(true);

getCruce1Punto().setEnabled(true);
getCruce1PuntoSelectivo().setEnabled(true);
getCruceUniforme().setEnabled(true);
getCruceUniformeSelectivo().setEnabled(true);
getProbabilidadCruce().setText("");
getProbabilidadCruce().setVisible(false);
getProbabilidadCruceLabel().setVisible(false);


getMutacionNormalRadio().setEnabled(true);
getMutacionSesgadaRadio().setEnabled(true);
getProbabilidadMutacion().setText("");
getProbabilidadMutacion().setVisible(false);
getProbabilidadMutacionLabel().setVisible(false);

getReemplazamientoHijos().setEnabled(true);
```

```
        getReemplazamientoMejores().setEnabled(true);


        getElitismo().setEnabled(true);
        getElitismo().setSelected(false);

        getProbabilidadCruce().setEnabled(true);
        getProbabilidadCruceLabel().setEnabled(true);
        getProbabilidadMutacion().setEnabled(true);
        getProbabilidadMutacionLabel().setEnabled(true);

        getTestParada().setEnabled(true);
        getTestParada().setText("");

        getBotonStart().setVisible(true);
        getBotonEnd().setVisible(true);
        getIniButton().setVisible(false);
        getBarraProgreso().setVisible(false);
}


public int getnVariables() {
        return nVariables;
}


public int getL() {
        return L;
}
```

```
public int getTamanioPoblacion() {
	return tamanioPoblacion;
}


public double getLower() {
	return lower;
}


public double getUpper() {
	return upper;
}


public String getRoute() {
	return route;
}


public int getSelection() {
	return selection;
}


public int getCross() {
	return cross;
}
```

```
public int getMutation() {
        return mutation;
}


public int getReplacement() {
        return replacement;
}


public double getCrossProb() {
        return crossProb;
}


public double getMutationProb() {
        return mutationProb;
}


public int getTest() {
        return test;
}


public int getkTorneo() {
        return kTorneo;
}
```

```
public boolean getElitism() {
	return elitism;
}


public JFrame getInterfaz() {
	return interfaz;
}


/**
 * Función que valida los valores que se han introducido en el panel
 * principal
 */
private boolean camposValidos(){

	boolean ok = true;

	camposVacios = new ArrayList<String>();

	if (nVariables <= 0)
		camposVacios.add("Nº. variables");

	if (L <= 0)
		camposVacios.add("Nº. máximo de operaciones distintas de {+,-}");

	if (tamanioPoblacion <=0)
		camposVacios.add("Tamaño población");
```

```
if ("".equals(getRangoValorMax().getText().toString()))
                    camposVacios.add("Límite superior del rango");

if ("".equals(getRangoValorMin().getText().toString()))
          camposVacios.add("Límite inferior del rango");

if (lower >= upper)
          camposVacios.add("Límite inferior >= superior del rango");

if (route == null || "".equals(route))
          camposVacios.add("Fichero de puntos muestra originales no escogido");

if (selection == 0)
          camposVacios.add("Tipo de selección");

if (selection == 2 && kTorneo <= 0)
          camposVacios.add("Valor de K en K-Torneo");

if (selection == 2 && kTorneo > tamanioPoblacion)
          camposVacios.add("Valor de K en K-Torneo mayor que el tamaño de la población");

if (cross == 0)
          camposVacios.add("Tipo de cruce");

if (cross > 0 && cross < 5 && (crossProb <= 0 || crossProb > 1))
          camposVacios.add("Probabilidad de cruce");

if (mutation == 0)
          camposVacios.add("Tipo de mutación");
```

```
        if (mutation > 0 && mutation < 3 && (mutationProb <= 0 || mutationProb > 1))
                camposVacios.add("Probabilidad de mutación");

        if (replacement == 0)
                camposVacios.add("Tipo de reemplazamiento");

        if (test <= 0)
                camposVacios.add("Test de parada");


        if (camposVacios.size()> 0)
                ok = false;

        return ok;
}


public void restauraComponentes()
{
        // Se restauran los componentes (cursor por defecto, vista original, etc.)
        habilitarComponentes();
}


/**
 * Función que muestra el resultado del algoritmo genético
 * @param resultado
 */
public void muestraResultado(double resultado)
{
```

```
            getBarraProgreso().setVisible(false);
            getResultadoAlgoritmo().setText(PanelPrincipalConstantes.TEST_RESULTADO_ALGORITMO + resultado);
            getResultadoAlgoritmo().setVisible(true);
            getIniButton().setVisible(true);
            interfaz.setCursor(Cursor.getDefaultCursor());

            // Se muestra la gráfica
            muestraGrafica();

    }


    private void muestraGrafica()
    {
Process p;
            try {
                    String path = new java.io.File(".").getCanonicalPath() + PanelPrincipalConstantes.COMANDO_GNUPLOT;
                    String[] comando = {path};
                    p = Runtime.getRuntime().exec(comando);

                    // Ruta del archivo de entrada
                    String rutaPuntosSalida = Utilidades.conseguirRutaRelativa(getRoute()) + "p_salida.dat";


                    OutputStream out = p.getOutputStream();

                    if (getnVariables() == 1)
                    {
                            // Envio de comandos al programa gnuplot para generar la grafica 2D
                            out.write("set title 'Regresion Simbolica' \n".getBytes());
```

```
            out.write("set xtics -100, 0.5, 100 \n ".getBytes());
            out.write("set ytics -100, 0.5, 100 \n ".getBytes());
            out.write("set grid \n ".getBytes());
            out.write("set grid xtics\n ".getBytes());
            out.write("set grid ytics\n ".getBytes());
            out.write("set xlabel 'Variable x'\n ".getBytes());
            out.write("set ylabel 'Variable y'\n ".getBytes());
            out.write("set style data lines \n ".getBytes());
            out.write("set style line 1 lt 3 lc rgb 'black' lw 6 \n".getBytes());
            out.write("set style line 2 lt 1 lc rgb 'orange' lw 4 \n".getBytes());
            StringBuffer grafica = new StringBuffer ("plot [-5:5] '");
            grafica.append(getRoute());
            grafica.append("' ls 1 title 'Puntos Muestra Entrada', '");
            grafica.append(rutaPuntosSalida);
            grafica.append("' ls 2 title 'Resultados_Salida' \n");
            out.write(grafica.toString().getBytes());
    }


    if (getnVariables() == 2)
    {
            // Envio de comandos al programa gnuplot para generar la grafica 3D
            out.write("set title 'Regresion Simbolica' \n".getBytes());
            out.write("set xtics -100, 0.5, 100 \n ".getBytes());
            out.write("set ytics -100, 0.5, 100 \n ".getBytes());
            out.write("set grid \n ".getBytes());
            out.write("set grid xtics\n ".getBytes());
            out.write("set grid ytics\n ".getBytes());
            out.write("set grid ztics\n ".getBytes());
            out.write("set dgrid3d 30, 30 \n".getBytes());
```

```
out.write("set hidden3d \n".getBytes());
out.write("set xlabel 'Variable x'\n ".getBytes());
out.write("set ylabel 'Variable y'\n ".getBytes());
out.write("set style data lines \n ".getBytes());
out.write("set parametric \n ".getBytes());
out.write("set style line 1 lt 3 lc rgb 'black' lw 1 \n".getBytes());
out.write("set style line 2 lt 1 lc rgb 'blue' lw 1 \n".getBytes());
out.write("set term aqua 0 \n".getBytes());
out.write("set view 60, 15 \n".getBytes());
out.write("set key outside \n".getBytes());
out.write("set key left bottom Left title 'Leyenda' box \n".getBytes());
StringBuffer grafica = new StringBuffer ("splot [-5:5] '");
grafica.append(getRoute());
grafica.append("' ls 1 title \'Puntos Muestra Entrada',  '");
grafica.append(rutaPuntosSalida);
grafica.append("' ls 2 title 'Resultados_Salida' \n");
out.write(grafica.toString().getBytes());
out.write("set term aqua 1 \n".getBytes());
out.write("set view 60, 80 \n".getBytes());
out.write(grafica.toString().getBytes());
out.write("set term aqua 2 \n".getBytes());
out.write("set view 60, 125 \n".getBytes());
out.write(grafica.toString().getBytes());
out.write("set term aqua 3 \n".getBytes());
out.write("set view 90,0 \n".getBytes());
out.write("unset ylabel \n".getBytes());
out.write("unset ytics \n".getBytes());
out.write("unset dgrid3d \n".getBytes());
out.write("set xlabel 'Variable x' offset 0, -3 \n".getBytes());
out.write("set zlabel 'Variable z' offset 0, 0 \n".getBytes());
```

```
                        out.write("set xtics -100, 0.5, 100 offset 0, -1 \n".getBytes());
                        out.write("set offset -1, 0 \n".getBytes());
                        out.write("set grid xtics \n".getBytes());
                        out.write("unset grid \n".getBytes());
                        StringBuffer grafica2 = new StringBuffer ("splot [-5:5] '");
                        grafica2.append(getRoute());
                        grafica2.append("' lt 3 lc rgb 'black' lw 6 title 'Puntos Muestra Entrada', '");
                        grafica2.append(rutaPuntosSalida);
                        grafica2.append("' lt 1 lc rgb 'blue' lw 5 title 'Resultados_Salida' \n");
                        out.write(grafica.toString().getBytes());

                    }
                out.flush();
                out.close();
            } catch (IOException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
            }
        }

}
```

5.8 tfm.genetics.principal.Principal

```
package tfm.genetics.principal;

import java.awt.Toolkit;
import java.util.Observable;
import java.util.Observer;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import tfm.genetics.algoritmo.Algoritmo;
import tfm.genetics.panel.base.PanelPrincipalBase;
import tfm.genetics.panel.base.PanelPrincipalConstantes;

public class Principal implements Observer {

	PanelPrincipalBase vista = null;

	/**
	 * Inicializa el panel principal y añade esta clase como observadora
	 */
	public void inicializar()
	{
		vista = new PanelPrincipalBase();
		vista.addObserver(this);
	}

	/**
```

```
 * Hace visible el panel principal
 */
public void arrancar ()
{
	vista.arrancar();
}

/**
 * Método de observación que recibirá una notificación desde el panel principal indicando
 * que todo está inicializado y listo para la ejecución del algoritmo genético.
 */
public void update(Observable e, Object estado){
	if (((String)estado).equals(PanelPrincipalConstantes.ALGORITMO_LISTO))
	{
		// Se crea una tarea que ejecutará el algoritmo genético pasados unos segundos
		ScheduledExecutorService worker = Executors.newSingleThreadScheduledExecutor();
		Runnable task = new Runnable() {
		   public void run() {

			// Instancia de toda la lógica del algoritmo genético
			Algoritmo regresionSimbolica = new Algoritmo();
			regresionSimbolica.inicializar(vista);

			// Se ejecuta el algoritmo genético
			double resultado = regresionSimbolica.ejecutar();

			vista.muestraResultado(resultado);

			Toolkit.getDefaultToolkit().beep();
		   }
```

```
                    };
                    // Se fuerza  que la tarea tenga un retardo de comienzo
                    worker.schedule(task, 3, TimeUnit.SECONDS);

            }
        }

        /**
         * Programa principal. Punto de entrada.
         * @param args
         */
        public static void main(String[] args) {

                Principal principal = new Principal();
                principal.inicializar();
                principal.arrancar();
        }

}
```

5.9 tfm.genetics.utilities.Utilidades

```
package tfm.genetics.utilities;

import java.io.File;
```

```
import java.util.Vector;

public class Utilidades {


	static public double numerosAleatoriosEntre01() {
		return Math.random();
	}

	public static double numerosAleatoriosEntreAB(double limiteInferior, double limiteSuperior) {

		return (double) (limiteSuperior * Math.random()) + limiteInferior;
	}

	public static void rellenaVector (Vector<Double> vector, int size)
	{
		for (int i = 0; i < size; i++)
			vector.add(0.0);
	}

	public static  String conseguirRutaRelativa(String ruta)
	{
		File file = new File(ruta);
		//create bufferreader to wrap the file
		String path = file.getAbsolutePath();
		String filePath = path.substring(0,path.lastIndexOf(File.separator)+1);
		return filePath;
	}

}
```

Optimización y aproximación al problema de la Regresión Simbólica a través de Straight Line Programs (SLP's) y Algoritmos Genéticos. Entrenamiento genético de SLP's.

Bibliografía y Enlaces de Interés

Pablo Solar Rodríguez

Algoritmos Evolutivos y Programación Genética

- César L. Alonso, Jorge Puente. *Straight Line Programs: A new linear genetic programming approach*. Proc. 20th IEEE International Conference on Tools with Artificial Intelligence. pp.517-524. 2008
- César L. Alonso, Jose L. Montaña. *Estrategias de la Inteligencia artifical : los algoritmos genéticos*. Informe Técnico
- T.Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, Oxford. 1996.
- W. Banzhaf. *Genetic Programming for Pedestrians* (ed.) Proceedings of the Fifth International Conference on Genetic Algorithms (IGGA'93). pp.638–. Morgan Kaufmann, San Francisco, CA. 1993.
- W. Banzhaf, P. Nording, R. Keller, F. Francone. *Genetic ProgrammingAn Introduction: On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Heilderberg- San Francisco. 1998.
- S. J. Berkowitz. *On comomputing the determinant in small parallel time using a small number of processors*. Information Processing Letters 18:147-150. 1984.
- M. Brameier; W. Banzhaf. *Linear Genetic Programming*. Springer 2007.
- P. Bürguisser, M. Clausen, M. A. Shokrollahi. *Algebraic Complexity Theory*. Comprehensive Studies in Mathematics. Springer. 1997.
- C. Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London. 1859
- Darrel Whitley. *A genetic algorithm tutorial*
- Holland, J.H., *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975, 211 p.
- Koza, J.R., *Genetic Programming. On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992, 819 p.
- Buckles, B.P., and Petry, F.E., *Genetic Algorithms* IEEE Computer Society Press, 1992, 109 p.

- Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison- Wesley Publishing Company, 1989, 412 p.
- Booker, L.B.; Goldberg, D.E., and Holland, J.H., *Classifier Systems and Genetic Algorithms*. Artificial Intelligence, 40, 1989, pp. 235-282.
- Davis, L. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991, 385 p.
- Forrest, S., *Genetic Algorithms: Principles of Natural Selection Applied to Computation*. Science, Vol. 261, No. 5123, August 13, 1993, pp. 872-878.
- Smith, R.E.; Goldberg, D.E., and Earickson, J.A., *SGA-C : A C-language Implementation of a Simple Genetic Algorithm* .TCGA Report No. 91002, The Clearinghouse for Genetic Algorithms, The University of Alabama, May 14, 1991.
- Ribeiro Filho, J.L.; Treleaven, Ph.C., and Alippi, C., *Genetic-Algorithm Programming Environments*. IEEE Computer, June 1994, pp. 28-43.
- Srinivas, M., and Patnaik, L.M., *Genetic Algorithms : A Survey*. IEEE Computer, June 1994, pp. 17-26.
- Rawlins, G.J.E, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991, 341 p.
- Whitley, L.D. *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1993, 322 p.
- Kaufmann, A., y Faure, R., *Invitación a la Investigación de Operaciones*. Segunda Edición, CECSA, México, 1977, 311 p.
- Koza, J., *Genetic Programming II : Automatic Discovery of Reusable Programs*. The MIT Press, 1992, 746 p.
- Francisco J. Varedas & Francisco J.Vico , *Computación Evolutiva Basada en un modelo de codificación implícita*. Inteligencia Artificial, Nº 5, pag 20-25.
- Natyhelem Gil Londoño, *Algoritmos Genéticos*. Sede Medellín, Colombia. Noviembre 2006

- Bautu, Elena, Bautu, Andrei, and Luchian, Henri (2005). *Symbolic regression on noisy data with genetic and gene expression programming*. In Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05), pages 321–324.

- Duffy, John and Engle-Warnick, Jim (1999*). Using symbolic regression to infer strategies from experimental data*. In Belsley, David A. and Baum, Christopher F., editors, Fifth International Conference: Computing in Economics and Finance, page 150, Boston College, MA, USA. Book of Abstracts.

- Schmidt, Michael and Lipson, Hod (2007). *Comparison of tree and graph en codings as function of problem complexity*. In Thierens, Dirk, Beyer, Hans Georg, Bongard, Josh, Branke, Jurgen, Clark, John Andrew, Cliff, Dave, Congdon, Clare Bates, Deb, Kalyanmoy, Doerr, Benjamin, Kovacs, Tim, Kumar,Sanjeev, Miller, Julian F., Moore, Jason, Neumann, Frank, Pelikan, Martin, Poli, Riccardo, Sastry, Kumara, Stanley, Kenneth Owen, Stutzle, Thomas, Watson, Richard A, and Wegener, Ingo, editors, GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, volume 2, pages 1674–1679, London. ACM Press.

- http://en.wikipedia.org/wiki/Genetic_algorithm

- http://www.talkorigins.org/faqs/genalg/genalg.html

- http://eddyalfaro.galeon.com/geneticos.html

- http://sabia.tic.udc.es/mgestal/cv/AAGGtutorial/node1.html

- http://www.cs.ucl.ac.uk/staff/W.Langdon/gpdata/ch1.pdf

- http://www.cs.ucl.ac.uk/staff/W.Langdon/gpdata/ch1.pdf

- http://www.dii.uchile.cl/ ceges/publicaciones/ceges73.pdf

- http://taylor.us.es/componentes/miguelangel/algoritmosgeneticos.pdf

- http://tigre.aragon.unam.mx/geneticos/indice.htm

- http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/t2geneticos.pdf

- http://147.96.80.155/sistemaoptimizacion/AlgoritmoGenetico.htm

- http://www.frro.utn.edu.ar/isi/algoritmosgeneticos/

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

GNUPlot

- www.gnuplot.info
- http://computacion.cs.cinvestav.mx/~acaceres/courses/gnuplotCurso/gnuplotCurso.html
- http://www.forosdelweb.com/f96/gnuplot-con-c-426615/
- http://www.suiri.tsukuba.ac.jp/~asanuma/gnuplot++/

JAVA

- http://docs.oracle.com/javase/7/docs/api/
- Deitel & Deitel, *Cómo Programar en Java* 9ª Edición, Pearson, España, 2012
- Ricardo Peña Marí, *Diseño de Programas. Formalismo y Abstracción*. 3ª Edición, Pearson Educación, España, 2005
- http://es.wikipedia.org/wiki/Java_%28lenguaje_de_programaci%C3%B3n%29
- http://docs.oracle.com/javase/tutorial/
- www.javaworld.com
- http://www.oracle.com/technetwork/articles/index.html
- http://www.tutorialspoint.com/java/
- www.stackoverflow.com
- http://www.coderanch.com/forums