Programación Evolutiva

Práctica 3

Grupo 08

Miguel Ángel Castillo Moreno

Juan Pedro Olmos Rojas

# Introducción:

Para el desarrollo de esta práctica, hemos creado optimizaciones, cruces, mutaciones, y métodos de control de bloating adicionales a los pedidos. A continuación, se expone

**Cruces**

Para todos los cruces descritos a la continuación se cumple la siguiente regla: Ningún cruce hará que ninguno de los descendientes exceda la profundidad máxima.

Las variaciones de cruces desarrolladas son las siguientes:

**Cruce Profundo:** Para este cruce solo se eligen los nodos de profundidad dos para intercambiar. Es decir, solo con los nodos directamente descendientes de la raíz. Esto asegura unos cruces muy grandes, aunque las posibilidades sean menores.

**Cruce Simple:** Para este cruce se elige un nodo función cualquiera, que se intercambiará con otro cualquiera del otro individuo siempre y cuando ese intercambio no hiciera que se exceda la profundidad máxima.

**Cruce Mixto:** Este método elige aleatoriamente uno de los dos métodos anteriores. Se hace el cruce profundo con una menor frecuencia que el simple.

**Mutaciones**

Para las mutaciones que se pedían, debemos hacer algunas aclaraciones:

**Mutación funcional simple:** Para esta mutación, el número de descendientes de la función del nodo a alterar debe ser el mismo que el por el que se sustituya. Por ello, como las dos únicas funciones con el mismo número de nodos son AND y OR, son las únicas que se intercambian. Es decir, si se elige un OR se cambia por un AND y viceversa. Nunca se escogerá un IF o un NOT, ya que no se podrían intercambiar por ninguna otra función actual.

**Mutación funcional simple**: Para las funciones AND y OR, cambiar el orden de los elementos no hace nada, y para la NOT, no hay múltiples elementos que permutar. Esto quiere decir que a la única función a la que altera esta mutación es a IF. Por ello, esta mutación combinada con árboles sin IF no hace nada.

Además de las pedidas, hemos creado algunas alteraciones sencillas sobre ellas:

**Mutación terminal múltiple:** Esta mutación hace la terminal varias veces. Potencialmente, puede alterar todos los nodos terminales del árbol. Es mucho más agresiva e introduce mucha aleatoriedad.

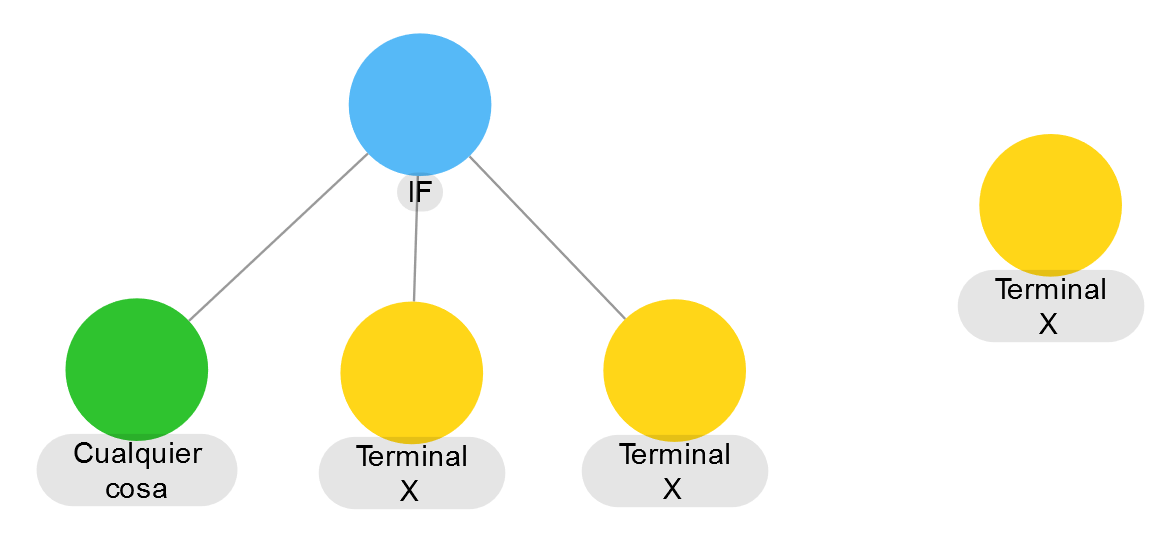
**Mutación terminal/funcional:** Esta mutación realiza la funcional y la terminal de forma aleatoria. Es ventajosa sobre ambas, ya que los árboles pueden alterarse de muchas más formas.

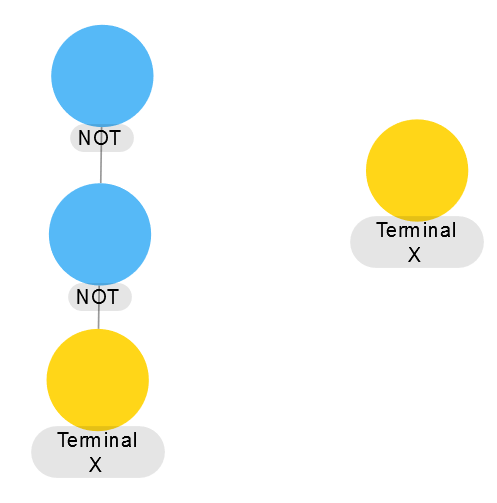
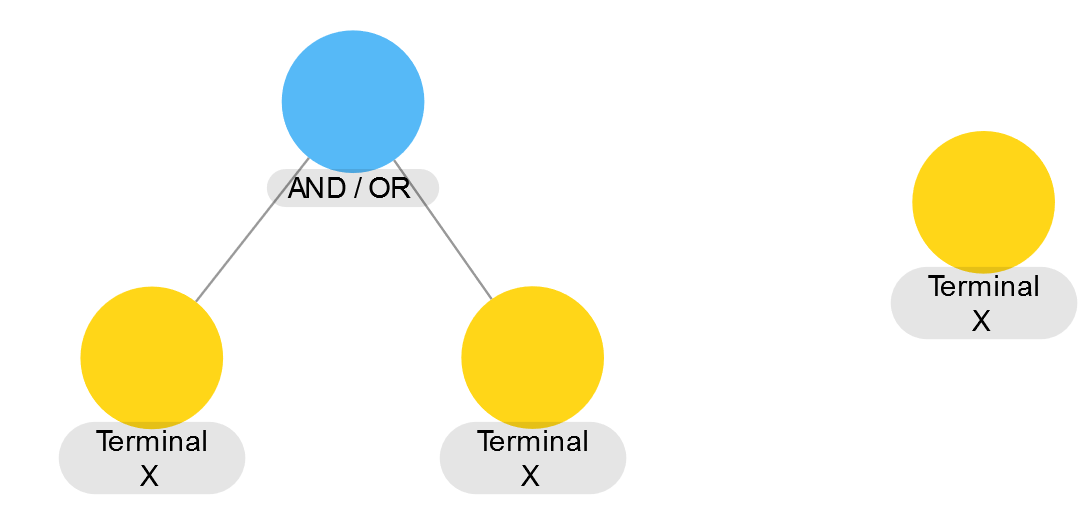
**Reducción del bloating:**

Hemos creado dos versiones del método de Tarpeian. Difieren en sobre qué se hace y compara la media. El método “Tarpeian Nº nodos” se rige por el número de nodos y el “Tarpeian Profundidad” por la profundidad total del árbol. Si el árbol a evaluar tiene la característica pertinente que sea superior a la media de la generación, este tiene una probabilidad del 25% de que su fitness sea dividido entre 10.

Además hemos creado un método propio que se puede aplicar junto a la penalización bien fundamentada o el método de Tarpeian. Lo hemos llamado simplificación.

**Simplificación:** Este método elimina y reduce trozos de árbol redundantes. A continuación, con estas imágenes podemos ver pares de árboles equivalentes:





La simplificación recorre todo el árbol sustituyendo los árboles redundantes de la izquierda por los terminales simples de la derecha. Asçi se reduce la profundidad innecesaria y se deja árboles no redundantes, o al menos sin redundancias demasiado obvias.

Una aclaración necesaria es que no se hace la simplificación si hacerla haría que el árbol se quedase con un único nodo terminal. Esto se debe a que nuestro programa está pensado para tener siempre una función en el nodo raíz, y se prefiere dejarlo redundante a generar bugs que haga que deje de funcionar.

**Optimizaciones:**

Resulta interesante comentar dos optimizaciones que han incrementado considerablemente la velocidad del programa y que no existían al comienzo del desarrollo:

**Cacheado del fitness:** La evaluación del fitness es un proceso extremadamente costoso, ya que por ejemplo para el multiplexor 11 hay que evaluar 2048 posibilidades para cada vez que se quiera saber el fitness de un individuo. Si este se guarda en una variable cuando se calcule, podemos evitar un gran número de operaciones.

Cuando se produce una variación en el genotipo, se tiene que recalcular el fitness. Estas variaciones incluyen mutaciones y cruces. Además, si se está calculando el fitness con bloating, también se debe recalcular el fitness real después. Al clonar un individuo, también obtiene el fitness del padre para.

Además del fitness, también hacemos lo mismo para la profundidad del individuo. Obtener la profundidad se hace con mucha menos frecuencia y cuesta mucho menos hacerlo, asique no aporta tanta mejora.

**Evaluación cortocircuitada de los árboles:** Esta mejora acelera mucho el proceso de evaluación del fitness. Anteriormente, se evaluaba el árbol de abajo hacia arriba, teniendo que pasar por todo el árbol sin saber si el resultado iba a ser útil.

Partiendo desde la raíz del árbol y de forma recursiva, podemos evitar evaluar ramas cuyo resultado no nos sea necesario. Esto se da en tres ocasiones:

* Con los IF, si evaluamos la rama de la condición, sabemos cuál de las otras dos debemos tomar. La que no se toma, no se evalua.
* Con los OR, si la primera rama que miremos equivale a TRUE, no es necesario mirar la otra, ya que el resultado final va a ser el mismo independientemente del resultado.
* Con los AND, si la primera rama es FALSE, no necesitamos mirar la segunda para saber que el resultado es FALSE.

# RESULTADOS