Tabla de contenido

1. Introducción(2-5paginas) 2

2. Estado del arte(20-30paginas) 3

2.1. Teoría de la evolución 3

2.2. Computación Evolutiva 5

2.3. Programación Genética 6

2.3.1. Lenguaje y Árboles 6

2.3.2. Inicialización 7

2.3.3. Evaluación 8

2.3.4. Selección 9

2.3.5. Reproducción 10

2.3.5.1. Crossover 10

2.3.5.2. Clonación 10

2.3.5.3. Mutación 11

2.3.6. La nueva población 11

2.4. El Cubo de Rubik 11

3. Objetivos del PFC 12

4. Diseño del algoritmo 12

4.1. Lenguaje 12

4.1.1. Solución 1 12

4.1.2. Solución 2 15

4.1.3. Solución 3 17

4.2. Fitness 19

4.2.1. Entropía 19

4.2.2. Cubos resueltos 20

4.2.3. Longitud de la solución 21

4.3. Evaluación 21

2. Implementación 21

2.1. ECJ 21

2.2. Cubetwister y el cubo de Rubik 22

2.3. Diagrama de clases 22

2.4. Configuración de ECJ 22

3. Resultados 23

4. Conclusiones y Futuras Líneas 23

5. Bibliografía 23

6. Anexos 23

6.1. Gestión del proyecto 23

6.2. ECJPARAMS 23

# Introducción(2-5paginas)

Este proyecto de fin de carrera intentará generar un programa que resuelva el cubo de Rubik de forma óptima mediante la programación genética. La programación genética es un rama de la computación evolutiva donde se imita el modelo de evolución natural aplicado a programas informáticos.

La computación evolutiva nace a través de diferentes vertientes: programación evolutiva (J. Fogel, Owens, & Walsh, 1966), algoritmos genéticos (Holland, 1975), evolución estratégica (Rechenberg, 1971) (Schwefel, 1974) y por último programación genética (Cramer, 1985) (Koza, 1992). Todas estas vertientes nacieron de forma independiente, pero en los noventa se unificaron para formar la computación evolutiva. Todas estas ramas coinciden en imitar la teoría de la evolución moderna en la que Darwin (Darwin, 1859) plantó sus pilares, para resolver problemas, normalmente de optimización, en la informática.

La computación evolutiva se caracteriza por encontrar soluciones inusuales a problemas de los que no existe un método resolutivo claro o viable en términos temporales. La potencia de estos sistemas reside en una multitudinaria exploración aleatoria del espacio búsqueda.

Uno de los problemas del cubo de Rubik es que debido al gran número de posibilidades de desorden no es posible determinar a priori un número mínimo de pasos que se necesitan para resolver cualquier cubo. De este modo, lo métodos que han dado mejores resultados han sido los sistemas de búsquedas exhaustivas informáticas, batiendo el record con 26 pasos (Cooperman & Kunkle, 2007) y 22 pasos (Rokicki, 2008). Sin embargo, explorar todo el extenso espacio de cubos de Rubik posibles resulta un proceso muy costoso, por lo que es posible que aún exista soluciones mas cortas inexploradas actualmente.

Esto hace que la programación genética sea ideal para la resolución óptima de este tipo de problemas.

# Estado del arte(20-30paginas)

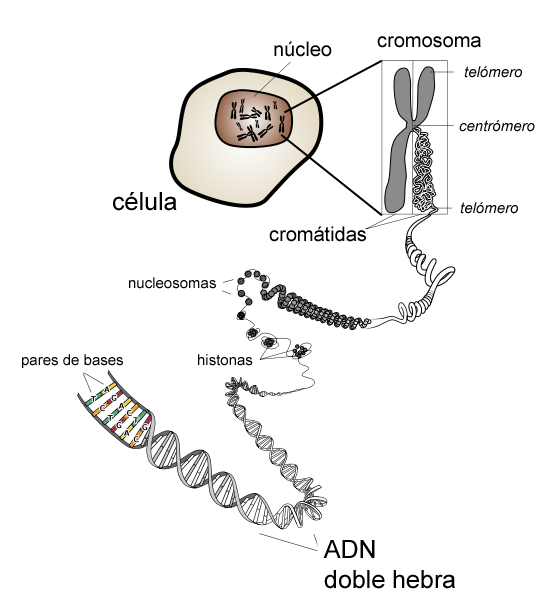
## Teoría de la evolución

Todo empezó un día de mucho calor. Tanto calor que el agua de los océanos casi hervía. Y en ese caldo se cultivaban las primeras cadenas proteínicas. Aquellas que conseguían duplicarse mas rápidamente que el resto, desbancaban a las otras sin piedad. Estas cadenas fueron adquiriendo poco a poco la capacidad de interactuar con otras cadenas, de tal forma que esta simbiosis daba una tremenda ventaja sobre el resto. Así se formaron las primeras células, la vida.

Para entender la programación genética es necesario comprender el funcionamiento de la evolución, lo que es el código genético, genotipo y fenotipo, y algunos fenómenos evolutivos. Por lo tanto, empezaremos hablando de todos estos conceptos.

¿Qué es el código genético? El código genético según es el conjunto de normas por las que la información codificada en el material genético (secuencias de ADN o ARN) se traduce en proteínas (secuencias de aminoácidos) en las células vivas (Wikipedia). Esto significa que el código genético almacena toda la información de nuestro cuerpo, de nuestras células: color de la piel, color de ojos, altura de las orejas, etc. No obstante, toda la información aprendida durante la vida no se almacena en los genes (código genético). De esta forma, nuestros hijos se parecen a nosotros porque llevan parte de nuestro código genetico.

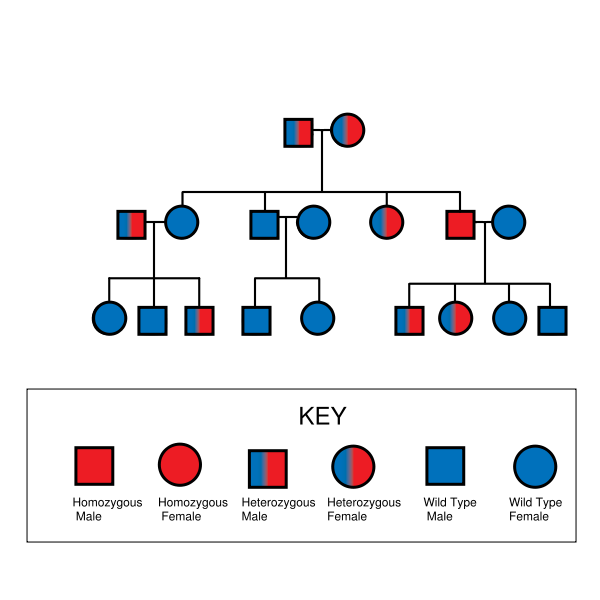
Entrando en profundidad, el código genético esta formado por una o varias cadenas de ácido desoxirribonucleico o ADN. A su vez estas cadenas estan formadas por cuatro nucleótidos cuyas bases son: adenina(A) timina(T) citosina(C) o guanina(G). Estas bases se unen entre sí formando cadenas larguísimas de información (AGGTACGTAATTCTGTCG…). Las secuencias de ADN que constituyen la unidad fundamental, física y funcional de la herencia se denominan genes. De esta forma llamaremos gen a la parte de ADN que codifica el color de los ojos. Realmente, todo esta información genética representa un lenguaje, un lenguaje con sus reglas gramaticales y sintácticas, un lenguaje que posteriormente es interpretado y expresado.

Pero existe una diferencia en la información que contiene el código genético y la expresión del mismo. Eso es lo que llamamos genotipo y fenotipo. El genotipo es la información que nuestro código genético contiene. Sin embargo, el fenotipo es la expresión del genotipo. Y esta puede variar por diferentes motivos, dominancia de genes, medio ambiente, etc. Por ejemplo, es posible que aunque tengamos la información para tener ojos azules y marrones, al final tengamos ojos marrones. Es importante que tengamos clara esta diferencia.

Cada individuo de una población tiene su propio ADN único que le define. Sin embargo, no dista mucho de otro individuo de su misma especie. Es por esto que se puede reproducir, ya que al ser ADN similar, trozos similares de ADN, con la misma función, se pueden intercambiar por los mismos trozos de el otro individuo. Esto se llama cruce o *crossover*. Así, la descendencia tendrá algunos rasgos del padre y otros rasgos de la madre: su código genético se ha mezclado.

Para que esto ocurra es necesario que exista un padre y una madre de características similares pero esencialmente diferentes. Esto es crucial para el éxito en la evolución: la diversidad.

Se ha demostrado empíricamente que la diversidad es necesaria para la supervivencia de la especie. Este echo se hizo claramente presente en la realeza a lo largo de la historia, donde solo tenían descendencia entre ellos. La consanguinidad reducía considerablemente la diversidad genética y empobrecía la calidad de lo individuos, potenciando enfermedades genéticas. Carlos II nació débil, estéril y enfermizo víctima de sucesivos matrimonios consanguíneos de la familia real.



Y toda esta diversidad tiene un origen basado en la mutación. Una mutación es un cambio, una alteración, en el código genético produciendo cambios en el fenotipo. El hecho mas importante de las mutaciones es que pueden ser heredables por la siguiente generación. No obstante, no todas las mutaciones son “buenas”, de echo, la mayoría de las mutaciones producen consecuencias fatales para el individuo. Por ejemplo….

Y para qué sirve toda esta diversidad. La diversidad sirve para crear individuos con diferentes cualidades, y estas cualidades sirven para enfrentarse a diversos problemas o situaciones. Estas situaciones pueden ser fatales si no son superadas. Por ejemplo, el ataque de un determinado virus, o el ataque de un depredador. Ciertas cualidades consiguen hacer que el individuo siga viviendo y por lo tanto tener descendencia. Por lo tanto, podríamos decir que la naturaleza va sometiendo a los individuos a unos ciertos filtros y estos filtros seleccionan a los individuos mas aptos. Es lo que se llama selección natural.

Combinando toda esta teoría evolutiva con la informática surge la computación evolutiva. La computación evolutiva convierte la evolución natural en un algoritmo de búsqueda. Imitando los propios mecanismos naturales, la computación evolutiva tiene una población de individuos que se reproducen y evoluciona a lo largo del tiempo. Dentro de la computación existen varios campos de estudio. Nosotros abordaremos la parte de algoritmos evolutivos, especialmente la parte de programación genética.

## Computación Evolutiva

Desde que Von Neumann ideó su modelo de computación moderna en los albores de la informática, el concepto de algoritmo tomó mas fuerza que nunca (una secuencias de pasos que sirve para resolver un problema). El problema ahora es encontrar la secuencia de pasos adecuada. Con la ventaja de que estas “nuevas” máquinas eran capaces de ejecutar estas secuencias de pasos infinitamente mas rápido que el ser humano, se abrió la puerta a nuevos campos de búsqueda y nuevas ideas. Desde siempre la naturaleza ha aportado a nuestra civilización multitud de ideas y soluciones y, en este ámbito, no iba a ser diferente. Por qué no imitar el mecanismo que tiene la naturaleza para dar con sus diversas y excelentes soluciones. La computación evolutiva es la realización de esta idea aplicada a la informática. La computación evolutiva es la implementación del algoritmo de la evolución.

Esta idea dio lugar a diferentes vertientes dentro de la computación evolutiva, entonces inexistente como tal. Desde USA, J. Fogel daba las primeras pinceladas a lo que se llamaría programación evolutiva. Fogel creo un sistema evolutivo donde los individuos a evolucionar eran maquinas de estado finito en el ámbito de la predicción. Henry Holland tuvo otra visión de este concepto y quiso aplicarlo a problemas de optimización, donde los individuos consistían en valores numéricos que se aplicaban de alguna forma a un determinado algoritmo. Este método recibió el nombre de algoritmos genéticos. En 1960 Ingo Rechenberg y Hans-Paul Schwefel iniciaron la rama que se llamaría Evolución Estratégica. Ya en los 90, Koza inició la última de las cuatro ramas, la programación genética. En esta última, los individuos que evolucionan son programas informaticos que luchan por conseguir la solución con mayor (o menor) puntuación. El mejor programa será capaz de procrear y tener descendencia, en cambio, los menos aptos estarán predestinados a desaparecer. De estas cuatro ramas se forma la computación evolutiva.

No obstante, los problemas evolutivos suelen venir acompañados de una carga computacional desmesurada, por lo que al principio sólo era posible resolver problemas muy simples. Con los últimos avances tecnológicos, la computación distribuida ha hecho posible que la computación evolutiva se convierta en unas de las ramas de la Inteligencia Artificial mas activas hoy en día.

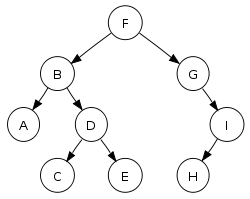
## Programación Genética

En 1992, John R. Koza publicó *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, un libro que extiende y asienta las bases que Nichael Lynn Cramer inició con su publicación *A Representation for the Adaptive Generation of Simple Sequential Programs*. Koza habla de las posibles codificaciones que un programa lineal pueden tener para adaptarlo al paradigma evolutivo. Defiende que la estructura mas adecuada es la representación en árbol. Un programa representado como árbol tiene múltiples ventajas, es posible volver al estado lineal sin ambigüedades, se asemeja más a la estructura de genes y cromosomas, las operaciones de reproducción y mutación se simplifican.

Este modelo de representación dio lugar a multitudes posibilidades de ajustes y modificaciones del algoritmo. Existen infinidad de posibilidades con los que podemos “jugar” y probar como los posibles métodos de reproducción, de mutación, de inicialización de programas, selección de nodos, etc. Hablaremos de estos aspectos en los siguiente puntos.

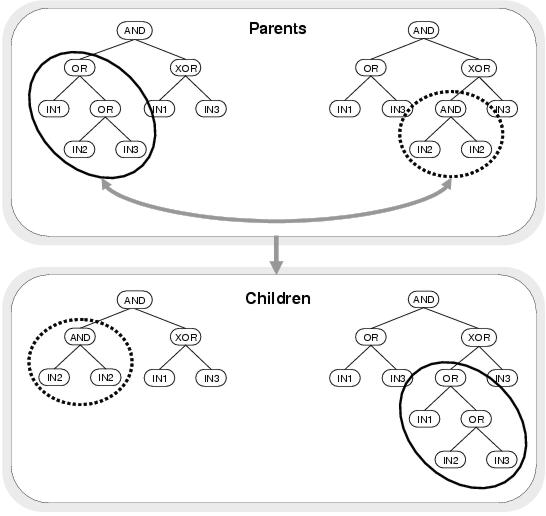
El algoritmo de la programación genética es como todos los algoritmos de la computación evolutiva. Se trata de un bucle, donde cada iteración se dice que es una generación. Si se trata de la primera iteración se generará una población de un determinado número de programas aleatoriamente. Durante la iteración, se procederá a la evaluación de los individuos, después a su selección, reproducción y/o mutación. En el proceso de evaluación, se ejecutará el programa y se puntuará de alguna forma su comportamiento. En el proceso de selección, se eligen los individuos más aptos de todas la población para procrear y generar una nueva población. Para procrear se pueden utilizar diferentes métodos, incluso clonar. Una vez que tenemos la nueva población, se mutará algunos individuos con una cierta probabilidad. El bucle llegará a su fin cuando se encuentre al individuo ideal o se llegue a un límite de generaciones.

### Lenguaje y Árboles

Existen muy diversas formas de representar el código de un programa en programación genética. Una de las mas extendidas es utilizar un árbol para este fin. Un árbol es una estructura de datos formada por nodos. Un nodo, explicándolo de una forma informal, es una caja que dentro contiene cierta información y que esta ligada a otras cajas, otros nodos. Estas ligaduras o conexiones se llaman hijos. Existen nodos que tienen varios hijos, y nodos que no tendrán ninguno, en cuyo caso se les llamará nodos hoja o terminales. Además existirá un nodo base del que partirán el resto de nodos.

En nuestro caso, cada nodo contendrá una instrucción del programa. Además, ya que la naturaleza de los programas es lineal, existen transformaciones de estos árboles en un código lineal sin ningún tipo de ambigüedad. Estas transformaciones se llaman preorder, inorder y postorder:

* Preorder: raíz, subárbol izquierdo, subárbol derecho. (F, B, A, D, C, E, G, I, H)
* Inorder: subárbol izquierdo, raíz, subárbol derecho. (A, B, C, D, E, F, G, H, I)
* Postorder: subárbol izquierdo, subárbol derecho, raíz. (A, C, E, D, B, H, I, G, F)



La decisión de utilizar un orden especifico viene dado por el lenguaje a utilizar para resolver el problema.

Esta estructura es idónea para la reproducción de los programas, ya que facilita el proceso de combinación genética que se representa por el intercambio de subárboles y, en esta estructura, este proceso resulta tremendamente trivial.

El proceso evolutivo se reduce a tres etapas: selección de individuos, reproducción y adición de los nuevos individuos. Además en la primera generación se inicializa la población. Este proceso debe ser aleatorio y debe proveer a la población de la suficiente diversidad para proporcionar el éxito del algoritmo.

### Inicialización

La parte de inicialización consiste en crear una nueva población y dotar al todos los individuos de la población de un código genético admisible y con una diversidad mínima para que el sistema pueda evolucionar. Un puede pensar multitud de métodos para cometer este fin, sin embargo, unos de los mas comunes y utilizados son los métodos *grow* y *full*. Además, estos dos métodos se suelen combinan de forma que el 50% de la población se genera por el método *grow* y el restante con el método *full*.

El método *grow* genera un árbol de programa de un individuo aleatoriamente hasta una determinada profundidad. En concreto, dado un conjunto de terminales y de no terminales de nuestro lenguaje, el método grow va rellenando el árbol aleatoriamente escogiendo indiferentemente entre terminales y no terminales hasta llegar a la máxima longitud, en cuyo caso sólo elije terminales terminando con la expansión del árbol. Aunque existe un límite estipulado de longitud máxima, es posible que los árboles generados no lleguen en profundidad a ese límite ya que al escoger entre aleatoriamente entre símbolos terminales y no terminales, puede darse el caso que aparezcan símbolos terminales podando el crecimiento del árbol. Incluso es posible que los árboles generados tengan mayor longitud, ya que si existe una gramática inherente al lenguaje, puede que en el límite de altura solo podamos escoger símbolos no terminales, prolongando el crecimiento del árbol hasta que podamos seleccionar terminales. Este método esta muy condicionado por la cantidad de nodos terminales y no terminales de nuestro lenguaje, ya que dado un lenguaje con un alto número de terminales, la probabilidad de escoger uno aumenta respeto a la probabilidad de escoger un símbolo no terminal. Esto ocurre también al contrario.

El método *full* es igual que el método *grow*, sin embargo, este método sólo elige símbolos no terminales hasta el límite de profundidad en cuyo caso sólo elegiría símbolos terminales. Es por esto que este método siempre genera árboles de la misma profundidad.

Como hemos mencionado antes, normalmente se utilizan ambos métodos para generar la población inicial. El motivo es que debido a que método grow esta condicionado por el numero de terminales y no terminales del lenguaje, es posible que con este método en un leguaje con muchos nodos terminales se generarían árboles muy cortos, ya que es muy probable que elijamos siempre nodos terminales. Este hecho puede poner en peligro el éxito de la evolución, ya que si los árboles que generamos son demasiado cortos, la diversidad de la población no sería suficiente para generar la solución deseada. Sin embargo, el método *full* garantiza un mínimo de altura en lo árboles. No obstante, el método full siempre generar árboles de la misma forma, es por esto por lo que se combinan ambos métodos para generar la población. Se suele utilizar un porcentaje de individuos que se generar mediante un método y otro porcentaje de individuos por otro método. Por último resaltar que en un lenguaje de predominancia de símbolos no terminales, los árboles que genera el método *grow* son parecidos a los que genera el método full.

### Evaluación

Antes de proceder a la selección, necesitamos evaluar a la población. El proceso de evaluación consiste en ejecutar cada individuo y probar su eficacia frente al problema. Esa eficacia tendrá que ser codificada para poder comparar a dos individuos cualquiera y decidir cuál es el mejor de ambos.

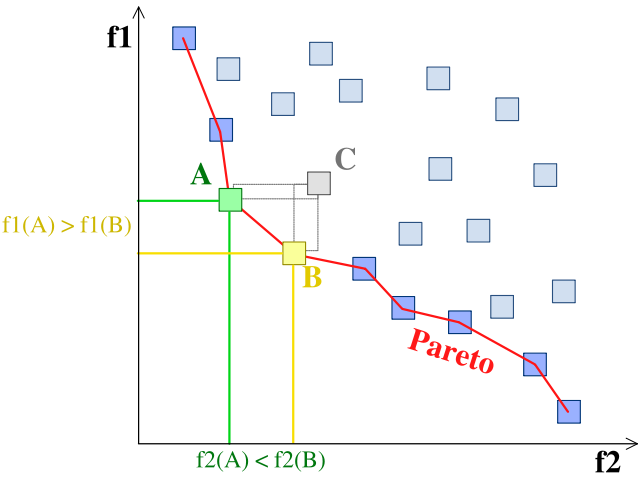
Koza defiende que solo es necesario un numero para codificar esta información, donde el valor 0 representaría al individuo mas apto e infinito para representar el peor individuo. A esta nota se la llama *standardized fitness*. Además se puede transformar a *adjusted fitness*  mediante la siguiente formula:

,



donde f es el fitness estandarizado.

Realmente nos interesa saber cuando un individuo es mejor que otro individuo. Con el sistema de Koza, hay que hacer una comprobación trivial, a<b. Sin embargo, cuando compiten muchos objetivos es difícil ponderar todos esos objetivos en una misma nota. Por ello existen diversos sistemas multiobjetivo capaces de optimizar la frontera de pareto.



La frontera de Pareto es aquella frontera formada por las soluciones óptimas de una población. Estas soluciones no se pueden comparar ya que en algún punto una solución A ofrece mejor resultado frente a cierto problema que una solución B, sin embargo la B ofrece mejor solución a otro problema que la A. Por ello no se pueden comparar. Sin embargo puede existir una solución C cuyas soluciones sean peor que la dada por la A, en ese caso, A es estrictamente superior que C. Este sistema intenta evitar que la solución se quede en un mínimo local, ampliando las fronteras a otros mínimos de interés. Además tiene la ventaja de potenciar la diversidad.

### Selección

El proceso de selección sirve para escoger los individuos de la población que será reproducidos, y por lo tanto transferidos de alguna forma a la siguiente generación. En la naturaleza existen varios factores para que un individuo pueda tener descendencia. El primero de todos es que consiga sobrevivir, ya sea por que es capaz de no ser devorado por otros, o por que sea capaz de procurarse alimento. Lo segundo es que encuentre pareja para reproducirse. Y el último factor es que el producto de esa pareja sea válido, es posible que existan combinaciones no válidas de individuos.

Un sistema muy utilizado en programación genética es el *tournament selection.* Este sistema consiste en escoger aleatoriamente de la población un cierto numero de individuos. De esos individuos se escoge el mejor de todos para ser el padre. Para escoger la madre se repite el proceso. Este sistema garantiza un mínimo de diversidad, ya que no siempre se elegirá al mejor individuo de la población.

Además existen otros métodos que tienen en cuenta cuánto mejor es un individuo que otro, ya que si la diferencia no es muy grande, quizás el vencedor de la selección no es el que tenga un fitness estrictamente superior.

### Reproducción

La fase de reproducción es la que se encarga de generar nuevos individuos, preferiblemente diferentes, a partir de los predecesores seleccionados en el proceso anterior. La parte principal de esta fase esta en el *crossover* que consiste en intercambiar parte del codigo del padre y parte de el de la madre. Sin embargo, es posible que simplemente se utilice la clonación de individuos para producir descendencia, dejando la diversidad a manos de la mutación. Incluso es posible combinar varias técnicas de crossover, utilizar a veces clonación y por último mutar algunos individuos. Aunque la mutacion no sea un operador específicamente de la reproducción, se suele aplicar después de este proceso.

La reproducción es una de las partes mas importantes, ya que será el encargado de generar un individuo nuevo y preferiblemente mejor que sus padres.

### Crossover

El *crossover* es el intercambio de material genético entre dos individuos para producir un individuo nuevo y diferente. En programación genética esto se traduce en intercambio de subárboles. Cuando dos individuos se van a reproducir se selecciona un nodo aleatoriamente en el padre y otro nodo en la madre y se realiza el intercambio de subárboles.

Sin embargo, en la naturaleza, durante la reproducción sexual de dos individuos se intercambia parte del código genético de tal forma que el código genético generado es similar al de los padres. Esto es posible debido a la organización existente del código genético en genes y cromosomas. Esto no sucede exactamente de la misma forma en la programación genética ya que al almacenarse el código genético en árboles, el intercambio de subárboles sin control mismo puede cambiar radicalmente la configuración del árbol resultante.

Existe otras maneras de intercambio de código que mantienen la posición del nodo una vez echo el intercambio llamadas *homologous.* Estas técnicas están basadas en identificar las partes de alta similitud de ambos árboles y realizar el intercambio de subárboles cerca de esas partes.

### Clonación

Como su propio nombre dice, se trata de copiar individuos de una generación a otra. Este método se llama también reproducción, aunque en este contexto el nombre puede dar lugar a ambigüedades. Clonar no aporta nueva diversidad a la población, no obstante, en ciertas ocasiones resulta conveniente aplicarlo, sobre todo en las ocasiones donde se ha conseguido una solución muy óptima.

### Mutación

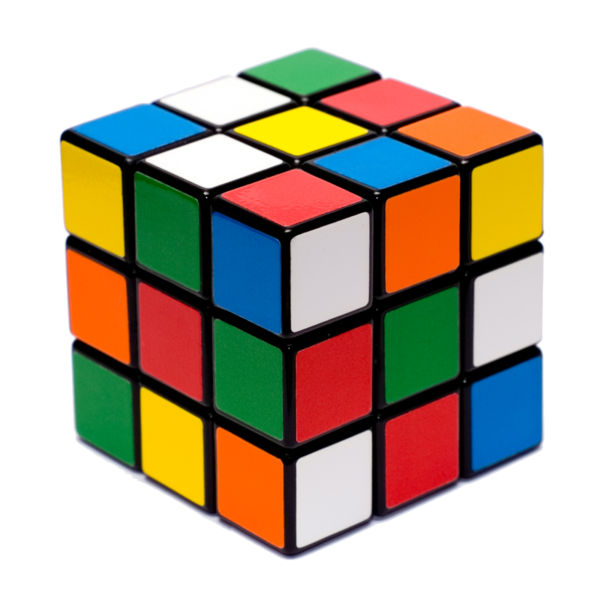
La mutación, en el ámbito de la programación genética, se define el cambio del código del programa, el código genético. Este cambio se produce de forma aleatoria. Sin embargo, la frecuencia de aparición de este fenómeno suele ser baja y normalmente de poca efectividad. Incluso se ha llegado a cuestionar si es necesario para el correcto funcionamiento del algoritmo evolutivo. Se ha demostrado que es posible resolver problemas con programación evolutiva sin llegar a utilizar la mutación. Pero para esto es necesario generar la suficiente diversidad inicial. Por el contrario, se ha visto casos que la mutación funciona igual de bien que el *crossover.*

Al igual que el *crossover*, existen múltiples maneras de hacer la mutación. Hay algunas que seleccionan un nodo aleatorio y se reemplaza por otro nodo de las mismas características. Hay otros que en vez de reemplazar el nodo, remplazan el subárbol entero por otro generado aleatoriamente. De este último método surgen diferentes ramas que difieren en la forma de generar aleatoriamente el subárbol.

### La nueva población

Una vez que hemos seleccionador, reproducido y quizás mutado la población, es momento de generar una nueva con los nuevos individuos. Normalmente se escogen los nuevos individuos mas aptos y se rellena la población hasta un número máximo de individuos. Además, se puede reservar un espacio de esa población para los individuos más aptos de todas las generaciones. Esto se llama elitismo. Aunque el elitismo vaya en contra de la diversidad, se ha comprobado que en ciertas ocasiones acelera la convergencia del algoritmo.

## El Cubo de Rubik

El cubo de Rubik fue inventado por un profesor de arquitectura Húngaro Ernö Rubik en 1974. Se trata de un puzzle en forma de cubo de 6 caras y con 9 pegatinas en cada cara. Cada cara tiene un color y estas caras se pueden rotar descolocando las posiciones de los colores. El objetivo del juego es volver a juntar las pegatinas del mismo color.

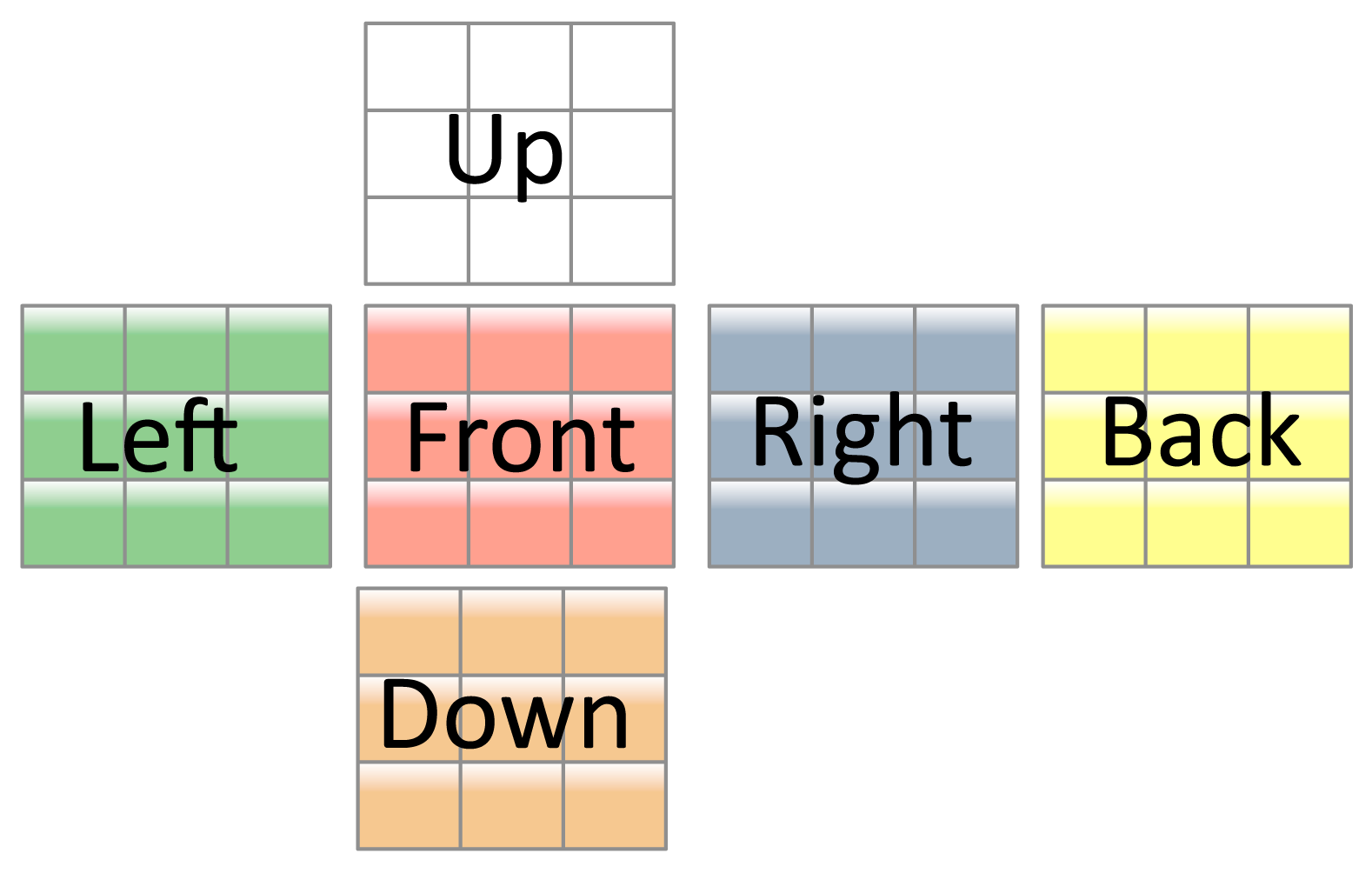
La dificultad del cubo de Rubik erradica en las millones de posibilidades de posiciones que puede llegar a tener. En total se calcula que pueden existir 43,252,003,274,489,856,000 permutaciones.

Existen diversas formas de resolución del cubo de Rubik, no obstante, el misterio erradica en que no es posible formalizar un algoritmo capaz de encontrar una solución óptima(la que menos pasos requiere) de cualquier cubo desordenado. En 2008, Tomas Rokicki batió el record y demostró que cualquier cubo de Rubik se resuelve en 22 movimientos o menos.

Para nuestro problema solo vamos a poder realizar movimientos de caras. Por lo tanto ya que tenemos 6 caras y dos sentidos de movimientos, podremos hacer 12 movimientos en cada estado. Si tenemos un cubo ordenado, podremos hacer 12 desplazamientos. Estos 12 cubos diremos que pertenecen a la dificultad 1. Por cada uno de estos 12 cubos tendremos 12 movimientos, por lo tanto 144 posibilidades. Estos cubos pertenecerán al nivel 2. Nótese que 12 de estos 144 cubos, es el cubo resuelto. Del nivel 3 tendremos 1728 cubos y así en adelante.

Estos niveles de dificultad nos servirán para clasificar los cubos de Rubiks y tener una idea a priori del número de pasos que son requeridos para su resolución.

Para referirnos a las caras del cubo utilizaremos la nomenclatura desarrollada por David Singmaster:

* F(Front): la cara que tienes enfrente.
* B(Back): la cara opuesta a la frontal.
* U(Up): la cara de encima a la frontal.
* D(Down): la cara de debajo de la frontal.
* L(Left): la cara a la izquierda de la frontal.
* R(Right): la cara a la derecha de la frontal.

TEORIA DE GRUPOS

# Objetivos del PFC

# Diseño del algoritmo

## Lenguaje

En este apartado iremos pasando por todos los lenguajes que hemos probado hasta llegar al lenguaje escogido para la representación del problema. Este lenguaje tiene que poder realizar todos los movimientos posibles del cubo de Rubik a la vez que realizar las comprobaciones suficientes para determinar qué movimiento realizar en cada momento.

### Solución 1

Un de las aproximaciones posibles, es pensar que el cubo de Rubik es una mapa. Cada casilla del cubo representa una posición del mapa. Sobre este mapa nos podemos mover hacia arriba, hacia abajo, derecha e izquierda. Al situarnos en el borde de una cara y avanzar nos situaremos en la siguiente cara. En la posición que estemos podemos comprobar de que color es la casilla que tenemos debajo. Además tendremos todos los movimientos de las caras del cubo.

Así tenemos el conjunto de nodos terminales:

|  |  |
| --- | --- |
| Nombre | Descripción |
| MovePosUp | Mueve la posición actual a la casilla superior. Si no existe tal casilla, se saltará de cara a la superior a la casilla Y2. Si estamos en la cara superior, nos desplazaremos a la cara trasera. |
| MovePosDown | Mueve la posición actual a la casilla inferior. Si no existe tal casilla, se saltará de cara a la inferior a la casilla Y0. Si estamos en la cara inferior, nos desplazaremos a la cara trasera. |
| MovePosLeft | Mueve la posición actual a la casilla izquierda. Si no existe tal casilla, se saltará de cara a la izquierda a la casilla X2. |
| MovePosRight | Mueve la posición actual a la casilla derecha. Si no existe tal casilla, se saltará de cara a la derecha a la casilla X0. |
| MoveFaceUpClockwise | Mueve la cara superior en el sentido de las agujas del reloj. |
| MoveFaceDownClockwise | Mueve la cara inferior en el sentido de las agujas del reloj. |
| MoveFaceRightClockwise | Mueve la cara derecha en el sentido de las agujas del reloj. |
| MoveFaceLeftClockwise | Mueve la cara izquierda en el sentido de las agujas del reloj. |
| MoveFaceFrontClockwise | Mueve la cara frontal en el sentido de las agujas del reloj. |
| MoveFaceBackClockwise | Mueve la cara trasera en el sentido de las agujas del reloj. |
| MoveFaceUpCounterClockwise | Mueve la cara superior en el sentido opuesto a las agujas del reloj. |
| MoveFaceDownCounterClockwise | Mueve la cara inferior en el sentido opuesto a las agujas del reloj. |
| MoveFaceRightCounterClockwise | Mueve la cara derecha en el sentido opuesto a las agujas del reloj. |
| MoveFaceLeftCounterClockwise | Mueve la cara izquierda en el sentido opuesto a las agujas del reloj. |
| MoveFaceFrontCounterClockwise | Mueve la cara frontal en el sentido opuesto a las agujas del reloj. |
| MoveFaceBackCounterClockwise | Mueve la cara trasera en el sentido opuesto a las agujas del reloj. |

Y nodos noterminales:

|  |  |  |
| --- | --- | --- |
| Nombre | Aridad | Descripción |
| IfBlue | 2 | Comprueba que la casilla de la posición actual sea azul. Si es así ejecuta lo hijos de la izquierda. Si no, ejecuta los de la derecha. |
| IfWhite | 2 | Comprueba que la casilla de la posición actual sea blanca. Si es así ejecuta lo hijos de la izquierda. Si no, ejecuta los de la derecha. |
| IfGreen | 2 | Comprueba que la casilla de la posición actual sea verde. Si es así ejecuta lo hijos de la izquierda. Si no, ejecuta los de la derecha. |
| IfOrange | 2 | Comprueba que la casilla de la posición actual sea naranja. Si es así ejecuta lo hijos de la izquierda. Si no, ejecuta los de la derecha. |
| IfRed | 2 | Comprueba que la casilla de la posición actual sea rojo. Si es así ejecuta lo hijos de la izquierda. Si no, ejecuta los de la derecha. |
| IfYellow | 2 | Comprueba que la casilla de la posición actual sea amarillo. Si es así ejecuta lo hijos de la izquierda. Si no, ejecuta los de la derecha. |
| Progn2 | 2 | Ejecuta primero la rama izquierda y luego la derecha. |
| Progn3 | 3 | Ejecuta primero la rama izquierda, luego la central y luego la derecha. |

Este lenguaje ofrece la ventaja que cada nodo es sustituible por cualquier otro. De esta manera será muy beneficioso en los procesos de *crossover* y generación de árboles.

Sin embargo este lenguaje resulta poco eficiente ya que el trabajo que cuesta comprobar dos casillas diferentes no siempre es el mismo y alcanzar el camino hacia una casilla puede ser otro problema en si. Otro factor de importancia es que depende mucho del estado actual del sistema, y el resultado de un nodo puede variar mucho en función de donde este colocado.

Vamos a ver un ejemplo de cómo se podría solucionar un cubo de rubik de nivel 1 con este lenguaje. Si por ejemplo rotamos la cara de la izquierda en el sentido de las agujas del reloj, tendremos un cubo de Rubik de nivel 1, donde la cara frontal tiene tres casillas del color de la cara superior, la cara inferior tiene 3 casillas del color de la cara frontal y así. Para dar con la solución necesitaríamos dar con el siguiente plan: IfOrange->MovePosDown->MovePosDown->MovePosDown->MovePosRight->IfOrange->MoveFaceLeftCounterClockwise. El plan puede no parecer muy complicado, pero si queremos volver a evaluar la casilla inicial, tendremos que volver hacia atrás y así con todas las casillas que se tengan que comprobar. Por otra parte, el lenguaje admite sentencias del tipo IfOrange->IfRed que carecen de sentido y que son difícilmente controlables. Esto puede desencadenar la generación de mucho código sin sentido de ninguna utilidad.

### Solución 2

Uno de los grandes fallos de la solución 1 es que comprobar el color de dos casillas diferentes se convierte en un problema en si mismo. En esta solución se intentará solventar este bache haciendo que una comprobación sea una instrucción y que da igual en que punto del código este, siempre se referirá a la misma posición.

Para ello tendremos una función Test, que comprobará si una casilla determinada es de un color determinado. Si lo es el valor devuelto de la función será verdadero. Si no será falso. Además tendremos todo el conjunto de operaciones condicionales como AND, OR y NOT. El sentencia If ahora tendrá tres hijos. El primero de los hijos representa la condición a evaluar, el segundo es la rama del árbol que se ejecutará si la condición es cierta, si no, se ejecutará la tercera rama del nodo.

De esta forma tenemos el conjunto de nodos terminales:

|  |  |  |
| --- | --- | --- |
| Nombre | Tipo | Descripción |
| X0 | XNode | Se trata de la representación de la posición x0 de una cara del cubo de Rubik. |
| X1 | XNode | Se trata de la representación de la posición x1 de una cara del cubo de Rubik. |
| X2 | XNode | Se trata de la representación de la posición x2 de una cara del cubo de Rubik. |
| Y0 | YNode | Se trata de la representación de la posición y0 de una cara del cubo de Rubik. |
| Y1 | YNode | Se trata de la representación de la posición y1 de una cara del cubo de Rubik. |
| Y2 | YNode | Se trata de la representación de la posición y2 de una cara del cubo de Rubik. |
| FaceLeft | FaceNode | Se trata de la representación de la cara izquierda del cubo de Rubik. |
| FaceUp | FaceNode | Se trata de la representación de la cara superior del cubo de Rubik. |
| FaceFront | FaceNode | Se trata de la representación de la cara frontal del cubo de Rubik. |
| FaceRight | FaceNode | Se trata de la representación de la cara derecha del cubo de Rubik. |
| FaceDown | FaceNode | Se trata de la representación de la cara inferior del cubo de Rubik. |
| FaceBack | FaceNode | Se trata de la representación de la cara trasera del cubo de Rubik. |
| Red | ColorNode | Se trata de la representación del color rojo del cubo de Rubik. |
| Orange | ColorNode | Se trata de la representación del color naranja del cubo de Rubik. |
| Green | ColorNode | Se trata de la representación del color verde del cubo de Rubik. |
| White | ColorNode | Se trata de la representación del color blanco del cubo de Rubik. |
| Yellow | ColorNode | Se trata de la representación del color amarillo del cubo de Rubik. |
| Blue | ColorNode | Se trata de la representación del color azul del cubo de Rubik. |
| Clockwise | DirectionNode | Es el sentido de rotación de las agujas del reloj de una cara. |
| CounterClockwise | DirectionNode | Es el sentido de rotación en sentido contrario a las agujas del reloj de una cara. |

Y el conjunto de nodos noterminales:

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Tipo | Aridad | Descripción |
| Test | CondNode | 4 | Comprueba que en la cara especificada (nodo 0), en la posición especificada (nodo 1 se refiere a x y nodo 2 se refiere a y) sea del color especificado (nodo 3). |
| And | CondNode | 2 | Realiza la operación booleana AND entre sus dos hijos. |
| Or | CondNode | 2 | Realiza la operación booleana OR entre sus dos hijos. |
| No | CondNode | 1 | Realiza la operación booleana NOT de su hijo. |
| Move | ActionNode | 2 | Mueve la cara (nodo 0) en la dirección (nodo 1) especificado. |
| If | ActionNode | 3 | Comprueba que la condición especificada en el hijo 0 se cumpla. Si se cumple ejecuta el subárbol del hijo 1, si no, el subárbol 2. |
| Progn2 | ActionNode | 2 | Ejecuta todos sus hijos. |
| Progn3 | ActionNode | 3 | Ejecuta todos sus hijos. |

El problema principal de este lenguaje es que cualquier nodo no es compatible con otro. Por lo tanto es posible que en determinadas ocasiones, la operación de *crossover* no se pueda producir. La consecuencia de que el crossover no se pueda producir es la copia del individuo en la nueva población, por lo tanto, el empobrecimiento de la diversidad.

Sin embargo, complejidad a la hora de comprobar el estado del cubo ha disminuido enormemente. Antes se requerían en el caso mas complejo alrededor de nueve pasos para comprobar el color de una casilla, ahora sólo se requiere un paso. Además se pueden expresar condiciones mucho más complejas.

Par resolver el ejemplo puesto en la solución anterior necesitaríamos de los siguientes operadores: If (Test FaceUp X0 Y0 Red) AND (Test FaceFront X1 Y0 Red) -> Move FaceLeft Clockwise. Como se puede observar, la sentencia resulta mucho mas simple y por otra parte fácil de entender.

Aunque hemos dado con un lenguaje bastante apropiado para representar la solución del problema, necesitamos explotar la potencia de este lenguaje. Con el árbol que hemos expuesto anteriormente solo somos capaces de resolver 1 caso de los 12 posibles de nivel 1, unas estadísticas muy poco favorables para la complejidad del árbol. El siguiente lenguaje que veremos explotará al máximo las posibilidades del lenguaje expuesto.

### Solución 3

Una de las características del cubo de Rubik es que existen estados que son equivalentes y existe una serie de rotaciones que transforman un estado en otro. Nuestro cerebro es capaz de abstraer estos estados y ver un solo estado y por lo tanto, un camino de resolución posible. Además es capaz de abstraer los colores específicos del cubo de Rubik, y puede trabajar con colores diferentes sin necesidad de adaptación alguna. De esta forma se podría reducir los 12 primeros casos, en dos simples casos: cuando debo girar en sentido de las agujas del reloj y cuando no.

Para imitar este comportamiento y adaptarlo a nuestro sistema se han realizado dos modificaciones importantes. La primera es que los colores ya no son colores concretos, como rojo o verde, sino abstracciones como color1 o color2 que pueden tomar diferentes valores en momentos concretos. Así en color1 puede referirse al color rojo en un momento determinado, y luego puede referirse al verde. Una situación que no puede ocurrir es que el color 1 se refiera al verde y el color 2 también. El color 1 es diferente al color 2.

La segunda de estas modificaciones refleja la segunda parte del proceso de abstracción. El cubo será rotado en todas sus posibilidades para cubrir todos los estados equivalentes.

Estas dos modificaciones las llevará a cabo la sentencia If y Test. If se encargará de rotar el cubo hasta que su condición se cumpla. Si no se cumple no se ejecutará el subárbol de acción. Además desaparece el subárbol *else* ya que no tiene sentido sin un previo If que coloque el cubo en la posición adecuada. Test se encargará de la asignación de los colores. Comprobará si existe algún tipo de asignación para el color que esta comprobando. Si no existe, devolverá verdadero y asignará el color especifico al color abstracto(color 1 = red). Es posible que una vez asignados todos los colores un *Test* no se verifique devolviendo *false* y por lo tanto deberán desasignarse todos los colores abstractos y probar en con otra rotación. Una vez que se ha encontrado una forma de que la condición se cumpla, el cubo de Rubik se dejará en posición deseada para que luego el grupo de acciones que vienen a continuación modifiquen el cubo en función de la condición planteada.

Por lo tanto tenemos el siguiente grupo de terminales:

|  |  |  |
| --- | --- | --- |
| Nombre | Tipo | Descripción |
| X0 | XNode | Se trata de la representación de la posición x0 de una cara del cubo de Rubik. |
| X1 | XNode | Se trata de la representación de la posición x1 de una cara del cubo de Rubik. |
| X2 | XNode | Se trata de la representación de la posición x2 de una cara del cubo de Rubik. |
| Y0 | YNode | Se trata de la representación de la posición y0 de una cara del cubo de Rubik. |
| Y1 | YNode | Se trata de la representación de la posición y1 de una cara del cubo de Rubik. |
| Y2 | YNode | Se trata de la representación de la posición y2 de una cara del cubo de Rubik. |
| FaceLeft | FaceNode | Se trata de la representación de la cara izquierda del cubo de Rubik. |
| FaceUp | FaceNode | Se trata de la representación de la cara superior del cubo de Rubik. |
| FaceFront | FaceNode | Se trata de la representación de la cara frontal del cubo de Rubik. |
| FaceRight | FaceNode | Se trata de la representación de la cara derecha del cubo de Rubik. |
| FaceDown | FaceNode | Se trata de la representación de la cara inferior del cubo de Rubik. |
| FaceBack | FaceNode | Se trata de la representación de la cara trasera del cubo de Rubik. |
| Color1 | ColorNode | Se trata de la abstracción de un color del cubo de Rubik. |
| Color2 | ColorNode | Se trata de la abstracción de un color del cubo de Rubik. |
| Color3 | ColorNode | Se trata de la abstracción de un color del cubo de Rubik. |
| Color4 | ColorNode | Se trata de la abstracción de un color del cubo de Rubik. |
| Color5 | ColorNode | Se trata de la abstracción de un color del cubo de Rubik. |
| Color6 | ColorNode | Se trata de la abstracción de un color del cubo de Rubik. |
| Clockwise | DirectionNode | Es el sentido de rotación de las agujas del reloj de una cara. |
| CounterClockwise | DirectionNode | Es el sentido de rotación en sentido contrario a las agujas del reloj de una cara. |

Y el grupo de noterminales

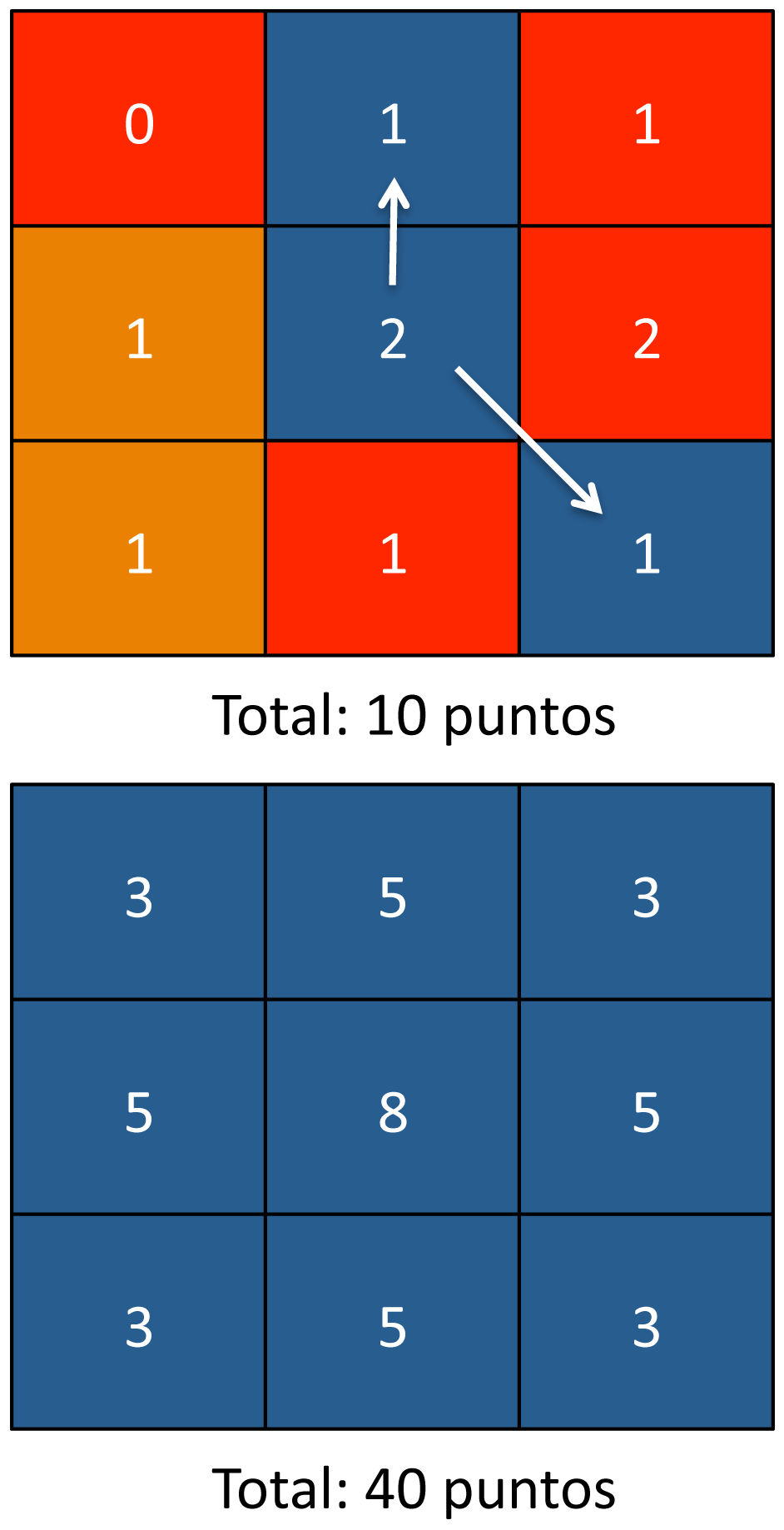
|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Tipo | Aridad | Descripción |
| Test | CondNode | 4 | Comprueba que en la cara especificada (nodo 0), en la posición especificada (nodo 1 se refiere a x y nodo 2 se refiere a y) sea del color especificado (nodo 3). |
| And | CondNode | 2 | Realiza la operación booleanas AND entre sus dos hijos. |
| Or | CondNode | 2 | Realiza la operación booleana OR entre sus dos hijos. |
| No | CondNode | 1 | Realiza la operación booleana NOT de su hijo. |
| If | IfNode | 2 | Se trata de la representación de la posición y2 de una cara del cubo de Rubik. |
| Progn2 | IfNode | 2 | Ejecuta todos sus hijos. |
| Progn3 | IfNode | 3 | Ejecuta todos sus hijos. |
| Move | ActionNode | 2 | Mueve la cara (nodo 0) en la dirección (nodo 1) especificado. |
| Progn2 | ActionNode | 2 | Ejecuta todos sus hijos. |
| Progn3 | ActionNode | 3 | Ejecuta todos sus hijos. |

Este lenguaje tiene los mismos problemas que el lenguaje anterior en el *crossover.*

En cuanto a las ventajas, si utilizamos el árbol expuesto en el lenguaje anterior, en este sistema resolvería 6 de los 12 cubos posibles de nivel 1.

## Fitness

### Entropía

La razón para necesidad de esta medición es que necesitamos saber cuanto de cerca estamos de solventar el cubo de Rubik. Podríamos saber este dato mediante la utilización de algoritmos de resolución del cubo de Rubik existentes. No obstante, estos algoritmos utilizan muchos recursos y multiplicado por cada individuo de la población, no sería viable. Además necesitamos otro valor ajeno a un sistema de búsqueda para no preenjuiciar la evolución. Por lo tanto hemos desarrollado un método basado en la entropía del cubo que nos diría con mas o menos acierto lo cerca que estamos de una solución.

El cálculo de la entropía del cubo consiste en la suma de las casillas del mismo color que estén contiguas dentro de una misma cara. Es decir, sumamos un punto por cada casilla que este contigua dentro de la misma cara y que sea del mismo color. Por consiguiente, una cara resuelta sumaría 40 puntos, por lo que un cubo de Rubik resuelto sumaría 240 puntos.

Existen estados intermedios mas cercanos a la meta que poseen menor entropía. Sin embargo, la tendencia general del cubo es a aumentar el orden de las casillas según se acerque a la solución. Además, esta medida se utilizará para comparar el estado final del cubo finalizada la ejecución de un programa frente al estado final del mismo cubo finalizada la ejecución de otro programa, por lo tanto, los estados intermedios no se deberían tener en cuenta.

### Cubos resueltos

Una vez que conseguimos tener cubos resueltos, sería muy fácil comparar programas en función del numero de cubo que resuelve uno y el número de cubos de rubik que resuelve otro. Si además añadimos la dificultad del cubo que resuelve cada programa, podremos hacer una clasificación mas exhaustiva.

La decisión mas difícil viene cuando tienes que decidir entre dos cubos que resuelven el mismo número de cubos pero de diferente dificultad. ¿Hacemos una simple suma y el que tenga más gana?, ¿es mejor que consiga resolver pocos cubos difíciles que muchos fáciles?¿no será mejor que sepa resolver todos los fáciles e ir ampliando conocimiento poco a poco?. Para comprobar que fitness puede ofrecernos mejores resultados, decidimos realizar unos experimentos. Para ello, utilizaremos las siguientes formulas en los experimentos:

* Puntuar mas las bajas dificultades:



* Puntuar más las altas dificultades:



Tras una serie de experimentos descubrimos que si puntuamos mas los cubos de nivel mas bajo perdemos la diversidad inicial, lo programas tienden a ser mas pequeños y llegado cierto punto, la convergencia del algoritmo casi se detiene.

Al puntuar mejor los niveles altos, conseguimos mantener la diversidad necesaria, ya que mientras mantenemos los programas que por algún casual han conseguido resolver algún cubo de nivel alto, los programas empiezan a resolver los cubos de niveles de nivel mas sencillos.

Como el que mejor funciona en solitario es el fitness que puntúa mejor los cubos de mayor dificultad, ese es el que utilizaremos en este proyecto. Es posible que en futuras ampliaciones y utilizando una aproximación multiobjetivo consideremos aplicar también el otro fitness.

### Longitud de la solución

Si queremos conseguir la conseguir la solución óptima

## Evaluación

* 1. Evaluación estática o dinámica
  2. Iteraciones
  3. Optimización
  4. Resolución

# Implementación

## ECJ

ECJ es una plataforma java para desarrollar aplicaciones con computación evolutiva. Se caracteriza de ser extremadamente modular y parametrizable. Esta plataforma consta de diferentes librerías para cubrir casi todos los aspectos de esta rama de IA. Nosotros nos centraremos más en las que se centran en la programación genética.

La clase principal es ec.Evolve. Esta clase se encarga de leer un fichero de configuración del sistema y empezar la evolución.

El fichero de parámetros de ECJ sirve para configurar el sistema y ejecutar la evolución. Se encarga tanto de las clases que se utilizarán, como de la estructura de los árboles de los programas, como de muchos otros parámetros personalizables. Utiliza una estructura jerárquica, de tal forma que si queremos utilizar la librería de Koza para la programación genética, simplemente tendremos que heredar del fichero de configuración de Koza y definir ciertas propiedades especificas de nuestro problema. Esto se hace mediante la siguiente línea:

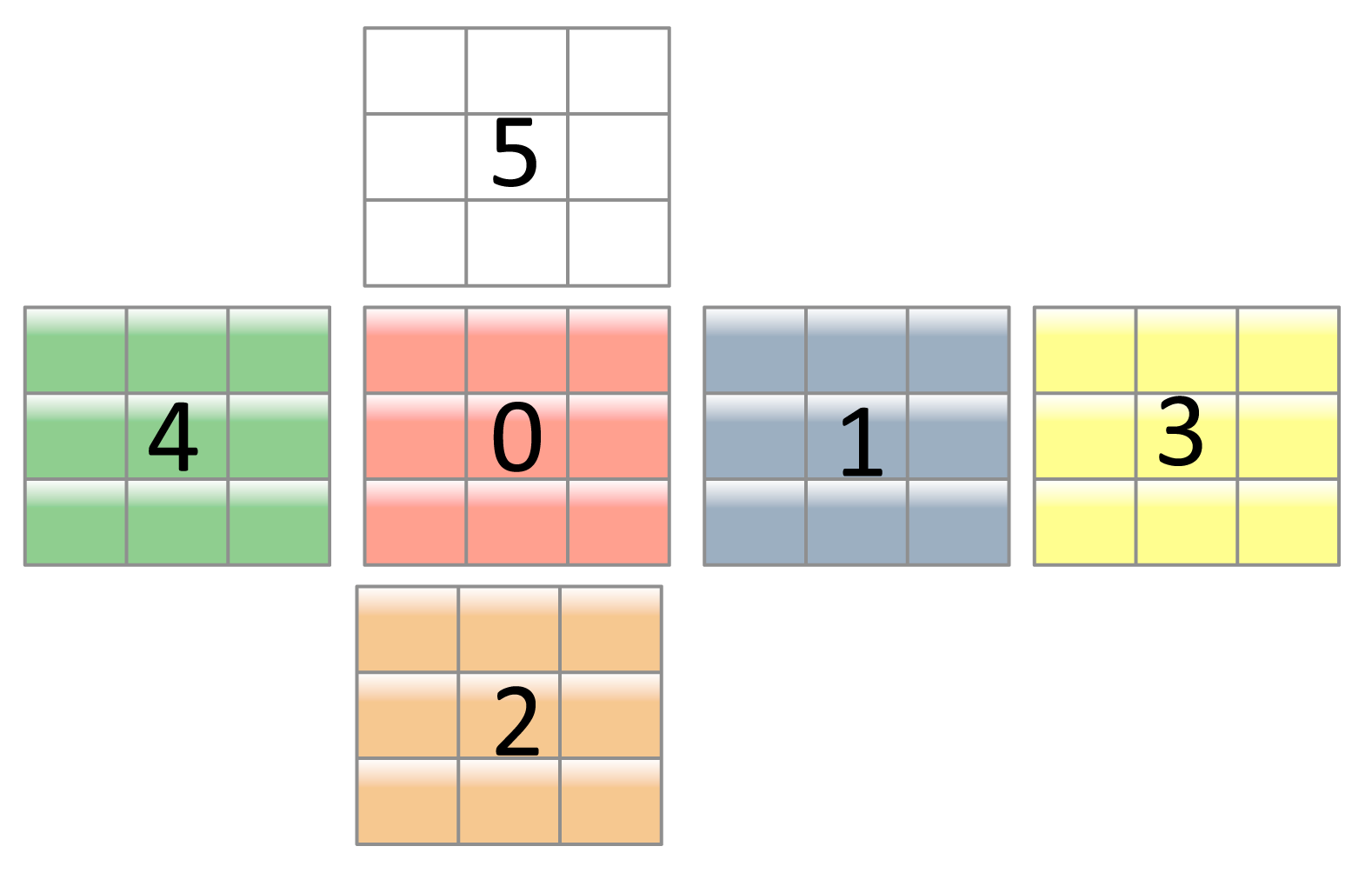
parent.0 = ../../gp/koza/koza.params

Sin embargo, para tener un control mucho más avanzado y personalizar ciertos aspectos de nuestro sistema evolutivo vamos a explorar y explicar el contenido del fichero desde el inicio de la jerarquía hasta el fichero de Koza.

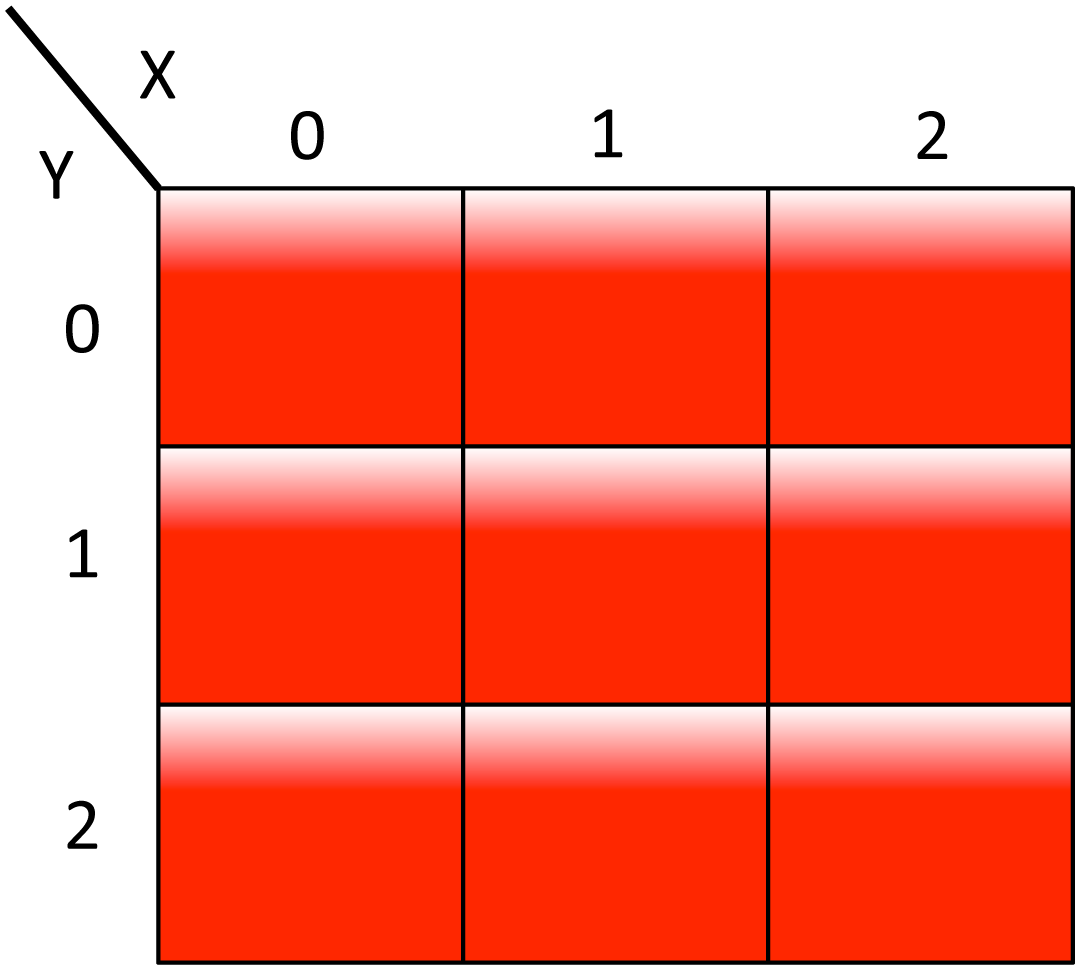
La jerarquía comienza con el fichero ec.params. Este fichero simplemente configura los parámetros de la ejecución de la evolución como número de threads, semillas, cada cuanto se hace un checkpoint:

## Cubetwister y el cubo de Rubik

-----------------------------PONER EN IMPLEMENTACION--------------------------------------

Para representar el cubo de Rubik en el ordenador utilizaremos la librería de Randelshofer(cubetwister.jar), en la versión 1.0.3.2. Existe una importante diferencia entre la versión 1 y la versión 2, también disponible en la Web, sobre todo en la nomenclatura de las caras. La numeración de las caras será la siguiente: la cara frontal será la 0, la derecha, la 1; la inferior, la 2; la trasera, la 3; la izquierda, la 4; y por último la superior, la 5.

Además, internamente, para situarnos dentro de una cara, utilizaremos la representación gráfica matemática, donde X0 e Y0 se situarán en la parte superior izquierda de la cara. No obstante el eje de ordenadas estará invertido, siendo la casilla X0 Y0 la casilla que esta en la esquina superior izquierda y la casilla X2Y2 la casilla de la esquina inferior derecha.



## Diagrama de clases

## Configuración de ECJ

# Resultados

# Conclusiones y Futuras Líneas

# Bibliografía

# Anexos

## Gestión del proyecto

## ECJPARAMS

* verbosity = 0 : Es el nivel de mensajes que el sistema sacará por pantalla. Un numero alto imprimirá mas mensajes por pantalla.
* Evalthreads=1 : Número de threads en el proceso de evaluación.
* Breedthreads=1 : Número de threads en el proceso de reproducción.
* Checkpoint=false : Activa o desactiva la opción de guardar el estado de la evolución en una generación concreta.
* Checkpoint-modulo = 1: Cada cuantas generaciones se haría el checkpoint.
* Prefix = ec : Es el prefijo del nombre del fichero de checkpoint.

El siguiente fichero en la jerarquía es simple.params.

* state = ec.simple.SimpleEvolutionState : esta clase define un modelo evolutivo generacional, es decir, cada generación es una nueva generación. Además se almacena la población de cada generación. Para poblaciones muy grandes y largas ejecuciones es mejor utilizar la clase ec.steadystate.SteadyStateEvolutionState que mantiene una población, donde sus miembros van siendo reemplazados por los nuevos.
* init = ec.simple.SimpleInitializer : es la clase que se utilizará para inicializar la población.
* Finish = ec.simple.SimpleFinisher : es la clase que usa para finalizar la población. Esta clase no hace nada actualmente.
* Exch = ec.simple.SimpleExchanger : esta clase sirve para definir la forma de intercambiarse individuos entre dos poblaciones. Esta clase tampoco hace nada.
* breed = ec.simple.SimpleBreeder : es la clase que asienta las bases de la reproducción. Se encargará de hacer el elitismo si se requiere. Sin embargo, esta clase solo escoge individuos dentro de la misma población.
* eval = ec.simple.SimpleEvaluator : es la clase que se encarga de administrar el procesos de evaluación de la población.
* stat = ec.simple.SimpleStatistics : realiza estadísticas simples de la población.
* Generations = 51 : numero máximo de generaciones que podrán ejecutarse.
* quit-on-run-complete = true : activamos la opción de que el sistema pare cuando encontremos la mejor solución.
* pop = ec.Population : la clase de población que utilizaremos.
* pop.subpops = 1 : el numero de subpoblaciones que tendremos
* pop.subpop.0 = ec.Subpopulation : la clase de subpoblación que existirá.
* pop.subpop.0.size = 1024 : el tamaño de la subpoblación 0
* pop.subpop.0.duplicate-retries = 0 : intentos de la subpoblación de eliminar los individuos repetidos de la población inicial 0.
* breed.elite.0 = 10 : espacio que dedicará para almacenar la élite de la población.
* stat.file $out.stat : el nombre del fichero de estadísticas.

El próximo fichero en la jerarquía es koza.params. Este fichero especifica unos parámetros básicos para que podamos hacer un pequeño sistema evolutivo. No obstante si se quiere hacer sistemas mas avanzados muchos de estos parámetros tendrán que ser especificados adaptándose concretamente a nuestro problema.

* pop.subpop.0.species.fitness = ec.gp.koza.KozaFitness : utilizaremos el fitness de koza.
* init = ec.gp.GPInitializer : Para inicializar la población se usará el inicializador de programación genética (GP).
* stat = ec.gp.koza.KozaStatistics : estadísticas de koza.
* pop.subpop.0.species = ec.gp.GPSpecies : especies de programación genética
* pop.subpop.0.species.ind = ec.gp.GPIndividual : individuos de programación genética.
* pop.subpop.0.duplicate-retries = 100 : se intentará 100 veces borrar a los individuos duplicados en la generación 0.
* pop.subpop.0.species.ind.numtrees = 1 : cada individuo tendrá un árbol.
* pop.subpop.0.species.ind.tree.0 = ec.gp.GPTree : este árbol será del tipo GPTree
* pop.subpop.0.species.ind.tree.0.tc = tc0 : Además tendrá la gramática tc0 (tree constrains).
* pop.subpop.0.species.pipe = ec.breed.MultiBreedingPipeline : lo individuos se someterán en la reproducción a varios sistemas de reproducción.
* pop.subpop.0.species.pipe.generate-max = false : no se intentará generar el máximo número de hijos.
* pop.subpop.0.species.pipe.num-sources = 2 : Se utilizarán dos formas de reproducción.
* pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline : la primera será *crossover*.
* pop.subpop.0.species.pipe.source.0.prob = 0.9 : Se escogerá con una probabilidad de 0.9.
* pop.subpop.0.species.pipe.source.1 = ec.breed.ReproductionPipeline : en la segunda se utilizará la clonación.
* pop.subpop.0.species.pipe.source.1.prob = 0.1: con probabilidad 0.1.
* breed.reproduce.source.0 = ec.select.TournamentSelection : para seleccionar a los candidatos de la clonación se utilizará la selección por torneo(antes explicada).
* gp.koza.xover.source.0 = ec.select.TournamentSelection : para el proceso de crossover, el padre se buscará por el método de selección por torneo.
* gp.koza.xover.source.1 = same : la madre por el mismo sistema.
* gp.koza.xover.ns.0 = ec.gp.koza.KozaNodeSelector : para seleccionar el nodo en el padre que se cambiará, se utilizará el sistema de koza.
* gp.koza.xover.ns.1 = same : para la madre se utilizará el mismo sistema.
* gp.koza.xover.maxdepth = 17 : la máxima profundidad de los subárboles a escoger para el *crossover* será de 17 alturas.
* gp.koza.xover.tries = 1 : las veces que se intentará encontrar nodos compatibles para la reproducción.
* gp.koza.mutate.source.0 = ec.select.TournamentSelection : para mutar, se seleccionará a los individuos por torneo.
* gp.koza.mutate.ns.0 = ec.gp.koza.KozaNodeSelector : la selección del nodo a mutar se realizada por el método de Koza (aleatorio).
* gp.koza.mutate.build.0 = ec.gp.koza.GrowBuilder : Una vez que se localice el nodo, se reconstruirá el árbol por el método *grow*.
* gp.koza.mutate.maxdepth = 17 : es la profundidad máxima del subárbol a escoger.
* gp.koza.mutate.tries = 1 : las veces que se intentará realizar la mutación con éxito.
* select.tournament.size = 7 : el número de individuos que entran en torneo.
* gp.koza.grow.min-depth = 5 : número de nodos mínimo de árbol del método *grow.*
* gp.koza.grow.max-depth = 5 : número máximo de nodos del método *grow.*
* gp.koza.ns.terminals = 0.1 : parámetro del selector de nodos, el cual seleccionará con probabilidad 0.1 terminales.
* gp.koza.ns.nonterminals = 0.9 : parámetro del selector de nodos, el cual seleccionará con probabilidad 0.9 noterminales.
* gp.fs.size = 1 : las siglas fs se refieren a Function Set. Sólo vamos a tener un grupo de funciones.
* gp.fs.0 = ec.gp.GPFunctionSet : nuestro grupo de funciones será del tipo GPFunctionSet.
* gp.fs.0.name = f0 : el nombre del grupo de función será f0.
* gp.type.a.size = 1 : tendremos solamente un tipo básico. Estos tipos básicos sirven para crear gramáticas y especifican el tipo de devuelta de los nodos.
* gp.type.a.0.name = nil : el nombre de este tipo básico es nil.
* gp.type.s.size = 0 : no tendremos ningún tipo compuesto.
* gp.tc.size = 1 : el tamaño de restricciones del árbol es de 1.
* gp.tc.0 = ec.gp.GPTreeConstraints : la clase para controlar estas restricciones es GPTreeConstraints.
* gp.tc.0.name = tc0 : el nombre de la restricciones sintácticas del árbol. Este debe coincidir con el especificado en el parámetro pop.subpop.0.species.ind.tree.0.tc.
* gp.tc.0.fset = f0 : el nombre del grupo de funciones.
* gp.tc.0.returns = nil : el tipo de devuelta del nodo base del árbol.
* gp.tc.0.init = ec.gp.koza.HalfBuilder : Es el constructor inicial del árbol. Mezcla el método *grow* y el método *full*.
* gp.koza.half.min-depth = 2 : es la mínima profundidad de los árboles generados.
* gp.koza.half.max-depth = 6 : es la profundidad máxima del árbol.
* gp.koza.half.growp = 0.5 : es la probabilidad que utilizará el método *grow* frente al método *full.*
* gp.nc.size = 7 : es el número de restricciones sintácticas.
* gp.nc.0 = ec.gp.GPNodeConstraints : la clase que soporta estas restricciones.
* gp.nc.0.name = nc0 : el nombre de esta restricción.
* gp.nc.0.returns = nil : el tipo de devuelta esta restricción.
* gp.nc.0.size = 0 : el número de hijos que tendrá el nodo al que se aplique esta restricción.
* gp.nc.1.child.0 = nil : cuando el número de hijos es mayor que cero hay que especificar el tipo de hijo que se requiere.

REPOSITORIO

Posibles mejoras:

ADF

Técnicas de mutación

Técnicas de breed

Técnicas de crossover

Técnicas de inicialización

Lenguaje

* consistente y noseque mas…

IDEA

Realmente el sistema evoluciona la condición del if…

Hacer un sistema con 12 árboles(cada uno mueve una cara) que sean las condiciones de movimiento. De esta forma, se evoluciona nada mas que la condición….

¿¿ADFs??

Modelo sin generaciones sino con tiempo de vida?->Tamaño de población variable.

Numero de especies variables. Dos individuos pertecenen a la misma especie cuando se pueden reproducir (nivel de crossover 50%)

Nivel de crossover, 50% código del padre y 50% de la madre!:D

Mapa 2D donde las propiedades evolutivas cambien(mutación, reproducción,etc…)

Desplazamiento de poblaciones.

Estaciones??Migraciones?? nivel máximo de individuos por zona.(limite de recursos).

Pulevas:

Lenguaje??

Metodo de evaluación??

Fitnesss??

Parámetros:

gp.koza.xover.tries

gp.koza.mutate.tries

select.tournament.size

gp.koza.grow.min-depth

gp.koza.grow.max-depth

gp.koza.fitness.factorfitness=2

pop.subpop.0.species.pipe.num-sources = 2

pop.subpop.0.species.pipe.source.0 = ec.gp.koza.CrossoverPipeline

pop.subpop.0.species.pipe.source.0.prob = 0.9

pop.subpop.0.species.pipe.source.1 = ec.breed.ReproductionPipeline

pop.subpop.0.species.pipe.source.1.prob = 0.1

gp.koza.ns.terminals = 0.1

gp.koza.ns.nonterminals = 0.9

gp.koza.ns.root = 0.0