Programación Evolutiva

Memoria de la práctica 3

David Alfonso Starry González

Daniel Pizarro Gallego

2024

Índice:

1. Introducción

El problema que vamos a optimizar es la poda de césped en una matriz toroidal de N filas por M columnas. Como la matriz es toroidal, no se puede salir del tablero. Si avanzas por la parte inferior, y sales de su límite, vuelves por la parte superior, es decir, cada fila y columna se puede entender por vectores de enteros modulo N y M, respectivamente.

El agente (un corta césped) empieza en la celda (4, 4). Tenemos que encontrar una serie de operaciones que se repite hasta una condición de finalización, que maximice el máximo número de celdas podadas en la matriz. Para ello tenemos una serie de operadores (más adelante hablaremos de los opcionales):

- Adelante: Avanza un paso y poda la casilla destino.
- Izquierda: Gira 90º a la izquierda. (No poda la casilla actual)
- Salta: Salta hasta un punto determinado, podando la casilla destino.

El agente termina su trabajo una vez ha realizado 100 operaciones (ticks). Este número se puede modificar opcionalmente.

Terminales:

- **AVANZA**: Avanza una casilla en la dirección actual, y poda la celda destino. Devuelve (0, 0)
- **IZQUIERDA**: Gira el cortacésped 90º a la izquierda. Devuelve (0, 0)
- **CONSTANTE**(X, Y): Genera dos números aleatorios. Devuelve (X, Y)

Funciones:

- **SUMA** (E1, E2): Ejecuta sus dos argumentos secuencialmente. Devuelve la suma vectorial en modulo N y M, de los resultados de sus dos variables.
- **PROGN**(E1, E2): Ejecuta sus dos argumentos secuencialmente. Devuelve el resultado de su segunda expresión.
- **SALTA**(E): Salta E.x y E.y casillas respecto a su dirección actual y poda la celda destino. Devuelve las coordenadas del salto.

Vamos a resolver este problema con **Programación Genética** (Parte obligatoria):

Cada individuo es un Árbol, pero para optimizar la implementación, además de almacenar el árbol entero con sus respectivos punteros, almacenamos una lista de strings. Así sabemos con coste constante O(1) el ciclo de operaciones, mejorando bastante el tiempo de ejecución de la función de evaluación.

Parte opcional: **Gramáticas Evolutivas.**

Esta vez los individuos son vectores de enteros generados aleatoriamente con valores de [0-255]. Cada entero se llama codón, y sirve para saber que nodo del árbol, así se puede representar el árbol con el vector. También aplicamos la misma técnica de almacenar las operaciones como en la parte anterior.

Más adelante explicaremos en mayor profundidad las características de estas implementaciones.

2. Arquitectura de la aplicación

La aplicación se aplica modelo vista controlador. Al ejecutar el programa, con la clase **Main** se inicializa la interfaz usando la clase **MainWindow**, que extiende a JFrame, y añade el controlador.

El controlador es la clase **ControlPanel** que extiende a JPanel, para crear 2 paneles dentro del principal.

- En el panel izquierdo se introducen los valores para las variables.
- El panel del medio imprime el gráfico 2D cuando termina una ejecución. El eje x es el número de generaciones, y el eje y el valor fitness. 1. Mejor absoluto (azul), 2. Mejor de la generación (rojo), 3. la media de la generación (verde) y 4. Presión selectiva (negro).
- En el panel de la derecha se encuentra la matriz con la cual se puede ver lo que el mejor individuo a podado, además de usan un botón para ver su paso por el tablero, con la dirección a la que apunta y las operaciones que ejecuta.

Una vez pulsado el botón de ejecutar, se ejecuta el algoritmo genético usando la clase **AlgoritmoGenetico**. En esta clase se guardan los valores almacenados en la interfaz, mediante una clase que almacena los datos, llamada **Valores**.

Los individuos de la población los implementamos con la clase **IndividuoArbol**, en el cual usamos la clase **Arbol**, para almacenar la raíz del árbol del individuo. Cada nodo del árbol es una expresión, que puede ser un no terminal como **Progn**, **Suma** o **Salta**, o un terminal **Avanza**, **Izquierda**, **Constante**, todos estos extienden a **Exp**. Los no terminales tiene asociado un valor *tam != 0*, además de sus hijos para formar el árbol.

La inicialización de estos individuos se puede elegir:

- Completa: Se genera el árbol con nodos no terminales hasta una cierta profundidad, en la cual solo se generan terminales. (Arboles completos)
- Creciente: Hasta una cierta profundidad se genera cualquier expresión, no terminal o terminal.
- Ramped & Half: La población se divide en *D=profundidad-1* grupos, de tamaños equitativos. Cada grupo utiliza una profundidad distinta (2, …, *D*)

La **mejora** que comentamos anteriormente, lo guardamos en un *List<String> operaciones* que, una vez creado el árbol o recorrido, para redireccionar los punteros al cruzar o mutar, almacena, en el orden correcto, las operaciones que ejecutará en el tablero para podar y calcular su fitness.

También almacenamos el número de nodos del árbol para la opción de **Bloating.**

Esta implementación es muy lenta, comparada con las implementaciones de las practicas anteriores que se gestionaban con array de reales, enteros o bits. Ahora hay que recorrer el árbol para construir o asignar valores. Sin tener en cuenta la

complejidad de las posibles funciones o que no haya problemas de punteros, y todos los individuos de una población converjan en el mismo.

Para ello almacenamos en los individuos una lista de punteros de terminales y otra de no terminales, para modificar rápida y correctamente el árbol a nuestro antojo. Y poder programar de manera más eficaz las funciones de cruce y mutación.

La función que calcula el fitness (o adaptación) de cada individuo, se implementa en la clase **Funcion**. El cálculo se implementa de la siguiente manera:

Ejecutamos un bucle *ticks (por defecto a 100)* veces recorriendo las operaciones del individuo, usando mod tam(operaciones). El agente siempre empieza en la celda 4x4, y ejecuta las operaciones. Si corta el césped suma uno a su fitness y marca dicha casilla como podada para no contabilizar más de una vez. Como avanza y salta tienen en cuenta la dirección a la que apunta, nos ayudamos de un array de vectores unitarios para saber a qué posición tiene que desplazarse el agente.

- Si avanza; tiene en cuenta *direccionAvanza* para desplazarse.
- Si salta; usa *direccionSalta* para desplazarse en la matriz en su dirección actual. *Ej Apunta al Norte y SALTA(4,2):* El agente tendrá que subir 4 celdas (restar 4 a su fila actual) y moverse a la derecha 2 celdas (sumar 2 a su columnas actual).
- Si gira a la izquierda aumenta en uno modulo 4 el iterador de dirección para los arrays de direcciones.

Los métodos de selección se implementan en la clase **Seleccion**. Se aplican igual para los dos individuos. Estos son:

- Ruleta: Selección aleatoria con la probabilidad acumulada de su fitness.
- Torneo determinístico: Se eligen ‘k’ individuos de la población de forma aleatoria y se elige el mejor. Este proceso se repite hasta llenar la población.
- Torneo probabilístico: Igual que el anterior, pero se elige peor o mejor (aleatoriamente)
- Estocástico universal: Similar al muestreo proporcional pero ahora se genera un único número aleatorio simple r y a partir de él se calculan los restantes. Los individuos se mapean en segmentos contiguos cuyo tamaño es el de su aptitud. a es un número aleatorio entre 0 y 1/tam_seleccionado. Se generan tam_seleccionados puntos en el segmento, y se eligen con sus aptitudes (probabilidades acumuladas).
  - Método1: aj=a+(j-1)/N
- Truncamiento: Se ordenan por fitness y con el porcentaje ‘trunc’ se eligen los mejores, 1/trunc veces.

- Restos: Las probabilidades acumuladas se multiplican por 'k', y se seleccionan este número redondeado para abajo veces, y los que falten con otro método.
- Ranking**:** Se basa en el ranking según la ordenación de los individuos por fitness decreciente. El valor asignado a cada individuo depende sólo de su posición en el ranking y no en su valor objetivo. Se calcula con la siguiente fórmula: $p(i) = \frac{1}{n}\left[\beta - 2(\beta.1)\frac{i-1}{n-1}\right], 1 \leq \beta \leq 2.$ β se puede interpretar como la tasa de muestreo del mejor individuo o **Presión selectiva**.

El método de cruce de programación genética se implementa en la clase **Cruce**.

- Intercambio: Al cruzar dos individuos, se genera un numero aleatorio entre [0-1]. Si este número es menor de 0.9 se hace un intercambio de nodos aleatorios teniendo en cuenta solo no terminales, es decir, funciones. En caso contrario solo se intercambias terminales.
    - Una vez se ha elegido entre no terminales y terminales se intercambian los subárboles (respetando los hijos de dichos nodos, se intercambian enteros, no solo el nodo). Se eligen de manera aleatoria, teniendo en cuenta el tamaño de nodos terminales o no terminales de cada árbol.
        - Para mejorar esta función tenemos la lista con los nodos, que para el nodo i-ésimo almacena su nodo padre. Con coste constante accedemos y modificamos los nodos en la lista.
        - Aplicamos swap para intercambiar los nodos de los padres. Y reiniciamos listas y tamaño de nodos para bloating.

Los métodos de mutación se implementan en la clase **Mutacion**.

- Terminal: Se regenera aleatoriamente un nodo terminal. Puede no surgir efecto si muta al mismo nodo terminal.
- Funcional: Se regenera aleatoriamente un nodo no terminal. Como solo cambia un nodo, tiene que permanecer sus nodos hijos, lo que quiere decir que nodos con dos hijos solo pueden mutar a otra función con dos hijos. Salto es el único no terminal con un hijo por lo que no puede mutar. Suma muta a Progn y viceversa.
- Arbol: Se elige un nodo aleatorio del árbol y se regenera dicho subárbol. Este árbol nuevo, se genera de manera aleatoria entre la inicialización completa o creciente y la profundidad se es un valor aleatorio entre [2-4].
- Permutación: Se invierte un subárbol. Es decir, se intercambia hijos. Solo surge efecto con nodos no terminales con mas de un hijo (Progn y Suma).
- Hoist: Se elige un nodo aleatorio del árbol y este pasa a ser la nueva raíz.
- Expansión: Se elige un nodo terminal aleatorio del árbol y se expande. Esta expansión se genera creando un nuevo árbol de profundidad aleatoria entre [1-3] e inicialización aleatoria.
- Contracción: Se elige un nodo no terminal aleatorio y se reduce a un nodo terminal.

Bloating: crecimiento sin control de los árboles durante el proceso evolutivo. Para no tener individuos muy grandes, hay que controlarlos y se puede aplicar varias técnicas.

- Tarpeian: Elimina aquellos programas de longitud mayor que la media, de vez en cuando (con probabilidad 1/n) (n=2, la mitad de los programas grandes muere)
- Poli and McPhee: Penalización bien fundamentada. Con esto se consigue que el tamaño medio de los individuos en todas las generaciones sea aproximadamente el mismo
  $f'(x) = f(x) + k * tam(individuo); k = Covarianza(L, F)/Varianza(L)$
  L = tamaño de los individuos; F =Fitness de los individuos
- **Ideas Propias:** Después de investigar y probar varias ejecuciones estos son algunos métodos con los que se pueden controlar el bloating. Están en el botón de Opcional.
  - Reinicio de población, cada X generaciones se reinicia la población, guardando el 10% de la población con mayor aptitud. Controla el bloating así reinicia la población y no hay tantos individuos con profundidad elevada.
  - Eliminación de superiores a la media: Cada X generaciones se elimina los que tengan un tamaño de árbol mayor a la media. Parecido a Tarpeian.

3. Mejoras implementadas

Se aplica un **algoritmo de divide** y vencerás $O(log_2 N)$ para reducir el tiempo de ejecución a la hora de elegir los valores como en ruleta y estocástica universal.

El **Elitismo** consiste en asegurar la supervivencia de un grupo con los mejores individuos de la población. En el controlador se puede especificar un porcentaje entero para el conjunto elitista que sobrevive en cada generación. Este conjunto se calcula en la etapa de evaluación, y se comparan los individuos con su fitness, almacenando los 'r' mejores en una cola de prioridad de mínimos. Así reducimos la complejidad a $O(N\ log_2 R)$ en el caso peor, siendo N el tamaño de población y R el conjunto de élite. Se compara el mínimo valor de los mejores con cada individuo, si el menor de la cola es peor que el individuo actual, se elimina de la cola y se introduce el nuevo, subiendo hasta su posición en la cola.

La **Presión Selectiva** es la mayor o menor tendencia a favorecer a los individuos más aptos. Cuantifica el número esperado de descendientes que se da al miembro más apto de la población**.** $\boldsymbol{PresSel[i] = TamPob \cdot ProbMax[i]}$. ProbMax[i] es la probabilidad de selección mayor en la generación i.

Desplazamiento de la aptitud: Tiene como finalidad hacer que la función de aptitud devuelva valores positivos. Se calcula con la siguiente fórmula: $f[i] = fmax - g[i]$. f[i] es el nuevo fitness del individuo i-ésimo, g[i] es el fitness antiguo. En lugar de utilizar fmax conviene utilizar valores ligeramente mayores (por ejemplo, un 105%) para prevenir que la adaptación revisada se haga nula.

Escalado de la adaptación: Permite controlar la diversidad de las aptitudes. Permite establecer una separación entre los valores de adaptación de distintos individuos que sea apropiada para el funcionamiento de la selección.

Implementamos **escalado Lineal**. Este tipo de escalado puede producir valores de adaptación negativos, que tenemos que evitar, pero en este problema no ocurre. El fitness nuevo se calcula de la siguiente manera f(x_i).

$$f(x_i) = a * g(x_i) + b \qquad a = \frac{(P-1)*\bar{g}}{g_{max}*\bar{g}} \qquad b = (1-a)*\bar{g}$$

Al final no lo usamos porque cambia a peor los resultados obtenidos. Seguramente lo implementamos mal.

Elegir tamaño del tablero: Se puede elegir el numero de filas y columnas antes de ejecutar una prueba.

Obstáculos: En el botón *Opcional* se puede elegir la posibilidad de añadir obstáculos al tablero. Estos se generan de manera aleatoria por toda la matriz, con una probabilidad de 5% de aparecer un obstáculo en cada celda.

Nuevos no terminales y terminales:

- No terminales:
  - **Retrocede**, igual que el operador Avanza, se desplaza una casilla, pero en sentido contrario a su dirección actual.
  - **Derecha,** igual que el operador Izquierda, pero cambiando su orientación 90º a la derecha.
  - **Mueve_A,** el agente se mueve a la primera celda adyacente (todas direcciones, incluidas las diagonales) sin podar.
- Terminales:
  - **Salta_A,** igual que el operador Salta, pero en vez de saltar x e y casillas se desplaza a una posición del tablero y poda la casilla destino.

Control de Bloating propio, comentado anteriormente.

Parte Opcional: Gramática Evolutiva (GE)

Es una técnica evolutiva que puede hacer evolucionar programas en cualquier lenguaje, y se puede considerar una forma de programación genética basada en gramáticas.

Ahora no representamos los individuos en forma de árbol, se utiliza una representación lineal del individuo (array de enteros). Cada elemento del array se llama codón, y oscila entre el intervalo [0, 255].

Los codones guardan la información para seleccionar una regla de producción en la representación BNF [1].

Las gramáticas se utilizan en el proceso de construcción de un individuo mediante la aplicación de reglas de producción, comenzando desde el símbolo inicial de la gramática (S≡<start>) y el primer elemento del array de enteros. Las reglas se eligen con la siguiente fórmula.

Regla a aplicar = c % r.
c=valor del codón. r=número de opciones en la regla.
Se aumenta el puntero al array de enteros para la siguiente regla. Si llega al final del array y la gramática no ha finalizado se vuelve a empezar desde el principio, a esto se le llama wrapping.

Gramática usada en la implementación.
(Obligamos a que la *raíz del árbol* sea un no terminal)

```
<start> := progn(<op>, <op>) | salta(<op>) | suma(<op>, <op>) | salta_a(<op>)
r=4
<op> := progn(<op>, <op>) | salta(<op>) | suma(<op>, <op>) | salta_a(<op>)
r=10   | avanza | constante(<num>, <num>) | izquierda | derecha | retrocede
       | mueve
```

Otra gramática que hemos implementado, pero da peores resultados.

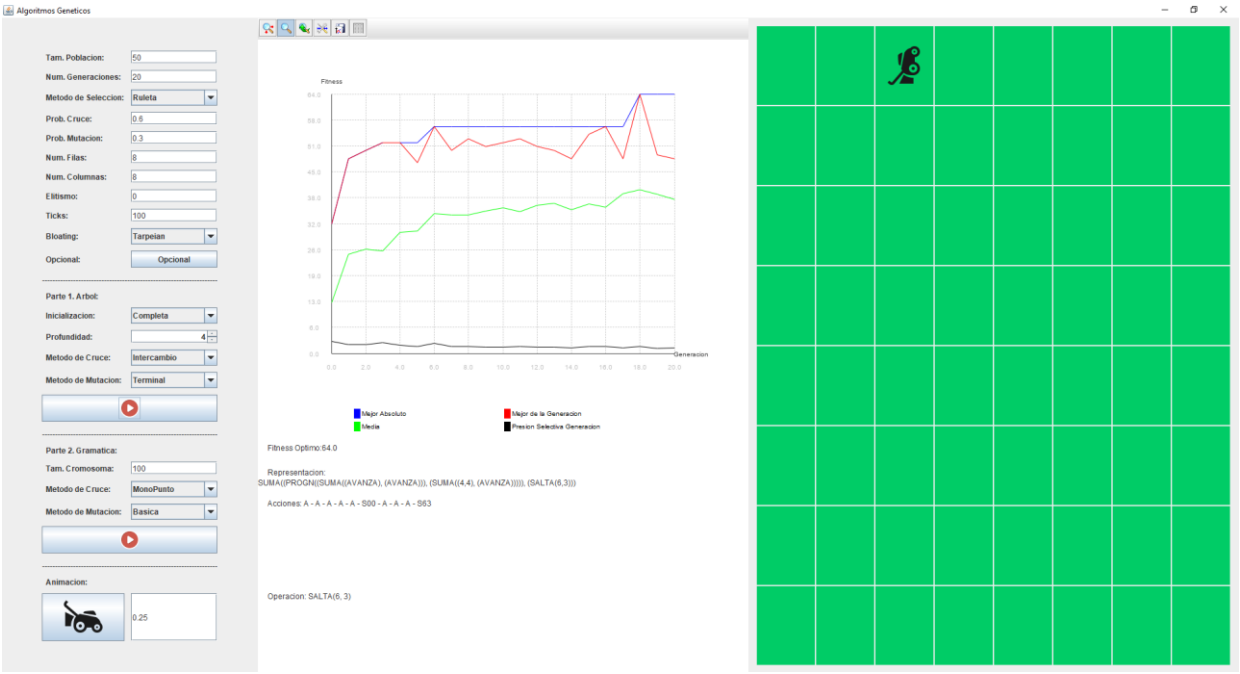
```
<start> := <op>
r=1
<op> := <no_terminal> | <terminal>
r=2
<no_terminal> := progn(<op>, <op>) | salta(<op>) | suma(<op>, <op>)
r=4            | salta_a(<op>)

<terminal> := avanza | constante(<num>, <num>) | izquierda
r=6         | derecha | retrocede | mueve
```

- Operadores de **Selección** son los mismos que en el apartado anterior.
- Operador de **Cruce**: Mono-Punto, se cruzan dos individuos por un punto aleatorio.
- Operador de **Mutación**: Básico, se elige un valor entero del array y se cambia su valor.

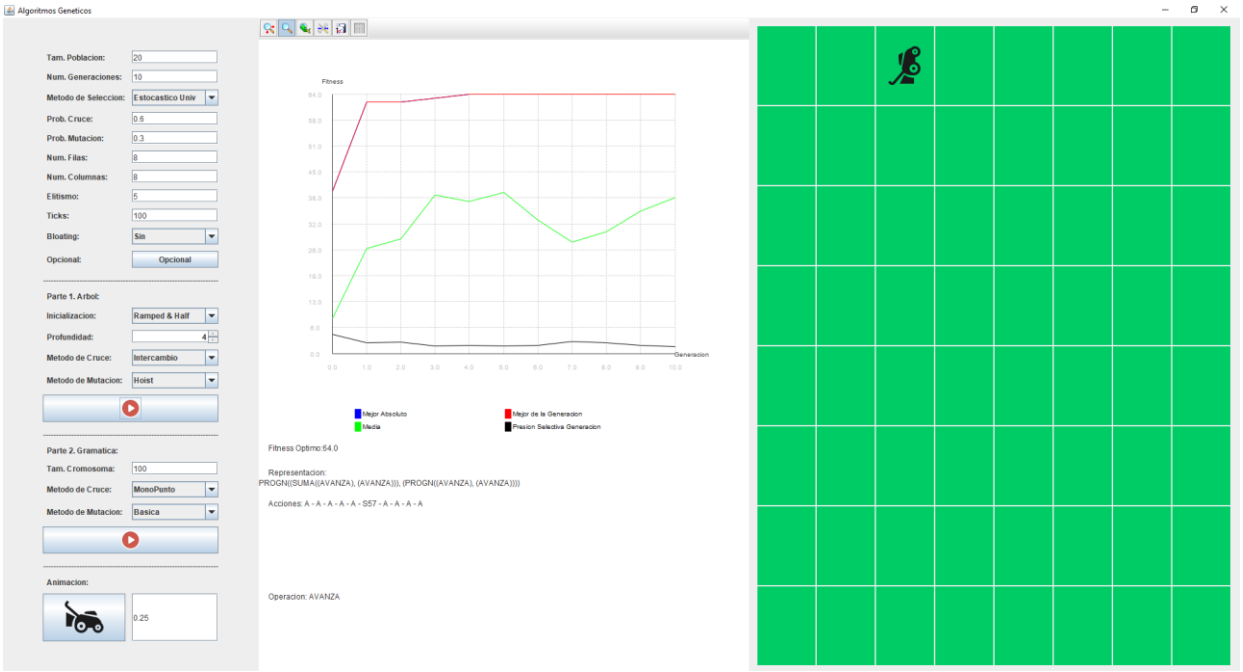
4. Ejecuciones, Análisis y Conclusión

4.1. Ejecuciones

4.1.1. Programación Genética (Árbol)

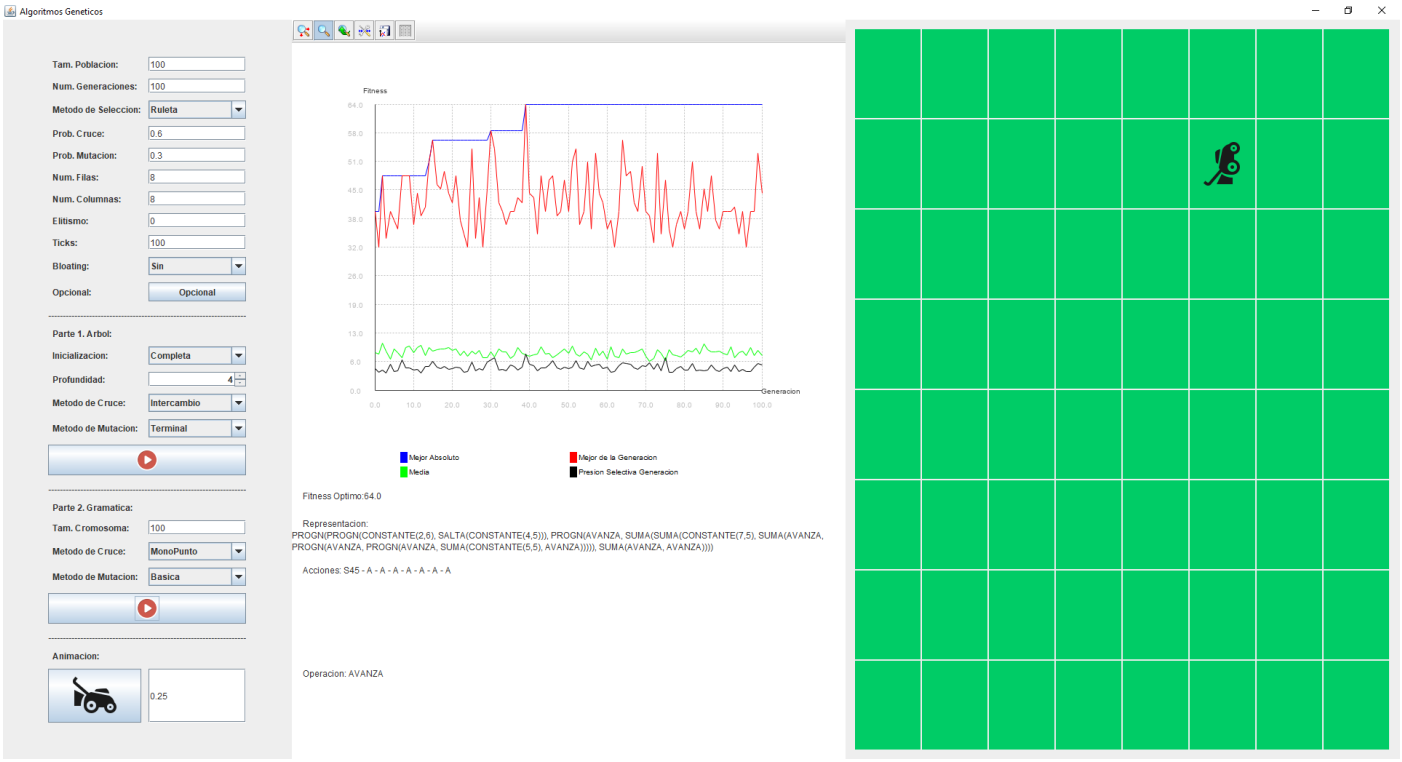


Tam. población: 50; Núm. generaciones: 20, inicialización: Completa. Selección por Ruleta y Mutación terminal. (Control de bloating: Tarpeian)



Tam. población: 20; Núm. generaciones: 10, inicialización: Ramped& Half. Selección por Estocastico y Mutación hoist. (Control de bloating: Sin)

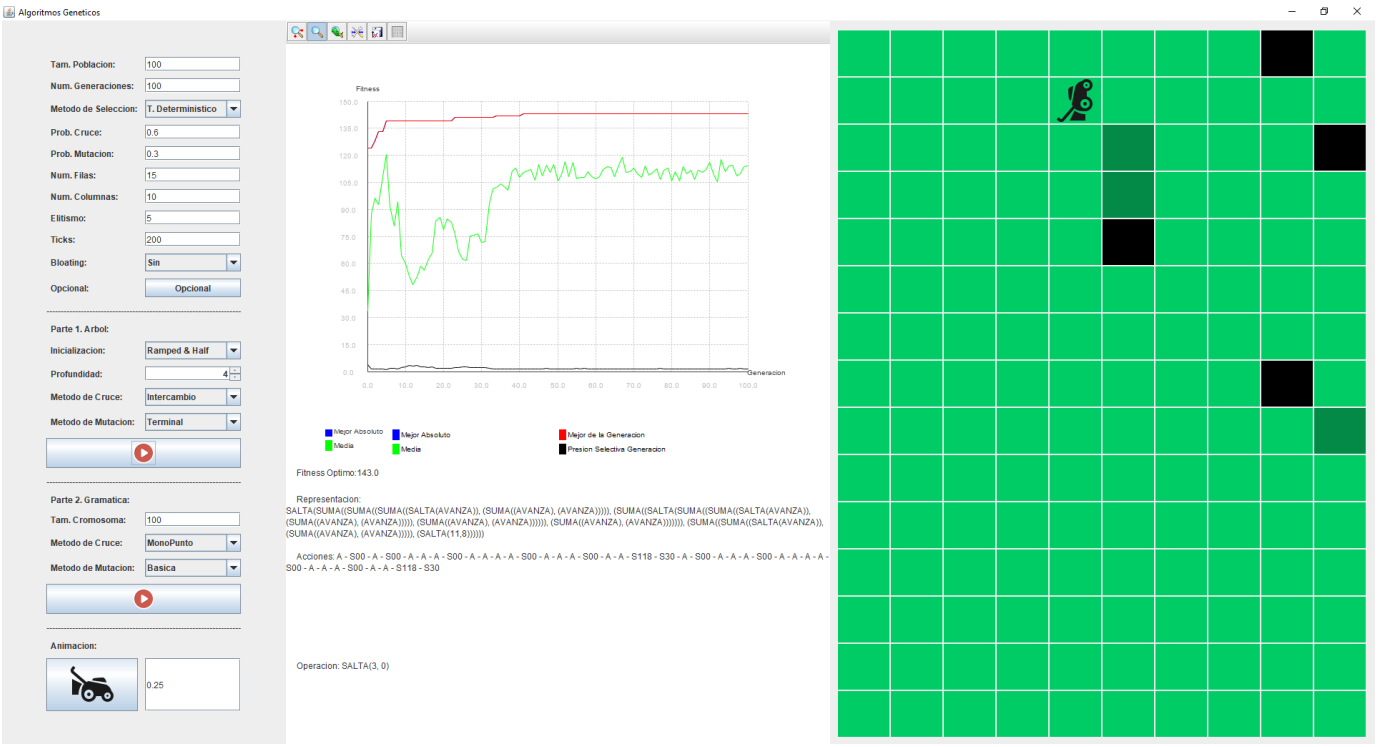
4.1.2. Gramática Evolutiva (Array de codones)



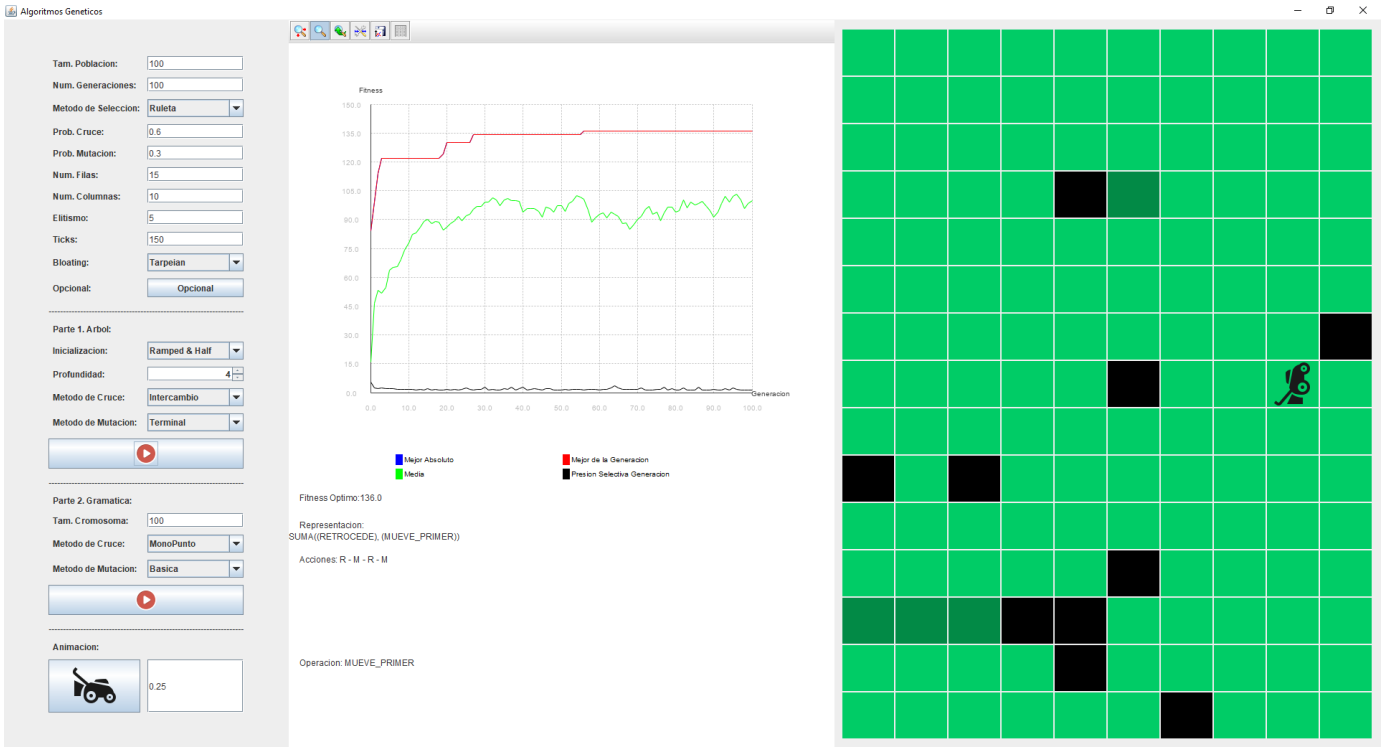
Tam. población: 20; Núm. generaciones: 10, inicialización: Ramped& Half. Selección por Estocastico y Mutación hoist. (Control de bloating: Sin)

Tam. población: 20; Núm. generaciones: 10, inicialización: Ramped& Half. Selección por Estocastico y Mutación hoist. (Control de bloating: Sin)

4.1.3 Ejecuciones Extra:



Filas=15, Columnas=10, y obstáculos, con pocos valores puede encontrar casi el óptimo.



Filas=15, Columnas=10, obstáculos, y todas las operaciones opcionales. Se puede apreciar que la operación Mueve, es muy efectiva y el agente prefiere retroceder a avanzar.

4.2. Análisis

Se pueden ver los ficheros .txt vinculados a cada prueba ejecutada.

Valor óptimo de un tablero 8x8 sin obstáculos, (valores por defecto) es de 64.

Programación Genética, todas las funciones dan buenos resultados con pocas generaciones y un tamaño de población pequeño.

- Prueba 1 (sin elitismo): Tamaño de población=10, generaciones=10, probabilidad de cruce y mutación= 0.6 y 0.3.

Con **valores** de entrada tan **bajos** da **buenos resultados**.

- Prueba 2 (sin elitismo) Tamaño de población=50, generaciones=20, probabilidad de cruce y mutación= 0.6 y 0.3.

Aumentando la población y unas pocas generaciones más **ya encuentra los valores óptimos**.

- Prueba 3 (con elitismo) Tamaño de población=50, generaciones=20, probabilidad de cruce y mutación= 0.6 y 0.3. Elitismo 5%.

Mismos resultados que la prueba anterior, pero esta vez la media aumenta un poco.

- Prueba 4 (sin elitismo) Tamaño de población=100, generaciones=100, probabilidad de cruce y mutación= 0.6 y 0.3. Elitismo 5%.

Mismos resultados que la prueba anterior, pero **aumenta** el **numero** de **nodos** y el **tiempo** de **ejecución**.

Gramáticas Evolutivas, No da muy buenos resultados, hay que subir mucho el tamaño de cromosoma, así como el tamaño de población y numero de generaciones.

- Prueba 1 (sin elitismo): Tamaño de población=10, generaciones=10, probabilidad de cruce y mutación= 0.6 y 0.3.

Con valores de entrada tan bajos los resultados son malos, el fitness medio llega a la **mitad del valor optimo**.

- Prueba 2 (sin elitismo) Tamaño de población=50, generaciones=20, probabilidad de cruce y mutación= 0.6 y 0.3.

Aumentando la población y unas pocas generaciones, no llega al óptimo, pero **mejora un 40%** respecto al anterior.

- Prueba 3 (con elitismo) Tamaño de población=50, generaciones=20, probabilidad de cruce y mutación= 0.6 y 0.3. Elitismo 5%.

Mismos resultados que la prueba anterior, **refinando** los **valores** calculados para mejorar un poco la media.

- Prueba 4 (sin elitismo) Tamaño de población=100, generaciones=100, probabilidad de cruce y mutación= 0.6 y 0.3. Elitismo 5%.

Al aumentar los valores, mejora los resultados, muchas veces **encuentra el óptimo**, pero la **media se queda un poco atrás**.

4.3. Conclusiones

Los mejores individuos suelen ser los que menos operaciones tienen, por lo que cuanto menos nodo tenga un individuo mejor será.

Cuantas mas generaciones pasen, la media de tamaño de los individuos será mayor, lo que provoca un mayor retardo en el tiempo de ejecución, y peores resultados al estar haciendo muchas operaciones distintas y no tener un patrón pequeño.

Métodos de selección, ordenados de mejor a peor:

- Los dos **torneos**, **truncamiento** y **ranking** son los mejores. Esto es así porque los mejores individuos de la población tienen más probabilidades de ser seleccionados. Por lo que encuentran antes el óptimo.
- **Restos** da resultados parecidos a los métodos anteriores (da resultados un poco peores), funciona bien porque también usa las probabilidades acumuladas.
- **Estocástico universal** y **ruleta**, al ser métodos que dependen más de la aleatoriedad dan peores resultados, pero esto se debe a que necesitan más generaciones para encontrar el óptimo.

Ocurre como en las practicas anteriores, sin embargo, para poblaciones pequeñas funciona mejor ruleta y estocásticos. A muy corto plazo encuentra valores mejores que los demás métodos de selección.

Métodos de cruce: Solo hay uno así que no podemos comparar.

Los métodos de mutación que mejores resultados obtienen son **Terminal**, **Permutación** y **Hoist**. Estos dos primeros pueden ser debido a que cambian poco los árboles y un pequeño cambio en los operadores cambia por completo el resultado. Y Hoist se debe a que, al cambiar el árbol por un subárbol de este, reduce el tamaño del individuo y controla el bloating.

En las gramáticas evolutivas son necesarias un tamaño de población mayor, así como el número de generaciones para obtener un óptimo. Esto se debe a que un cambio en el cromosoma cambia por completo el árbol, y los individuos son muy volátiles. También se debe a la aleatoriedad con el uso de codones y random de Java.

Con elitismo se puede controlar la mejora de la población ya que siempre se queda el conjunto de mejores individuos por lo que se pueden cruzar y mutar hasta encontrar el óptimo.

Los operadores extra como derecha y retrocede no aportan mucho, ya que es una matriz toroidal. Y el salto a una casilla no es útil con individuos pequeños ya que al repetir el patrón siempre van a la misma casilla.

Al modificar el tamaño de la hemos observado que hay configuraciones más fáciles de obtener el óptimo. Así como descubrir que los obstáculos dificultan mucho la obtención de un individuo optimo.

5. Guía de Uso

Antes de ejecutar el proyecto hay que comprobar si la librería externa, JMathPlot está incluida. En el classpath del proyecto está incluido correctamente.

La configuración de parámetros para la ejecución está dividida en tres partes. (Componentes de JSwing para configurar la ejecución)

Parte superior:

- Tamaño de población
- Numero de generaciones
- Método de Selección
- Probabilidad de Cruce y Mutación
- Numero de Filas y Columnas
- Porcentaje de Elitismo
- Numero de Ticks. Número de operaciones que ejecuta cada individuo en la matriz para podar y calcular el fitness.
- Método de Bloating. Para controlar el sobrecrecimiento de los individuos.
- Botón Opcional. Aparece un Dialog en el que se puede seleccionar con CheckBoxes que quieres añadir a la ejecución. Aquí están las operaciones opcionales y los obstáculos.

Parte 1:

- Método de Inicialización: Completa, Creciente o Ramped & Half.
- Profundidad de los árboles. Valores entre [2, 5]
- Método de Cruce. Solo hay uno.
- Método de Mutación
- Botón de Ejecución de Programación Genética.

Parte 2:

- Tamaño del cromosoma, numero de codones.
- Método de Cruce y Mutación, solo hay uno para cada uno.
- Botón de Ejecución de Gramática Evolutiva.

Animación:

- Botón de animación
- JTextField, Segundos por operación

6. Reparto de Tareas

Hemos hecho juntos las implementaciones para la programación genética y gramáticas evolutivas, dividiendo el trabajo de las funciones de ambas partes. David ha optimizado la implementación de los árboles. Daniel ha creado la GUI, con las animaciones.

Marcadores:

[1]. Backus-Naur form (BNF), Backus-Naur formalism o Backus normal form, es un metalenguaje usado para expresar gramáticas libres de contexto: es decir, una manera formal de describir lenguajes formales.
https://www.wikiwand.com/es/Notaci%C3%B3n_de_Backus-Naur