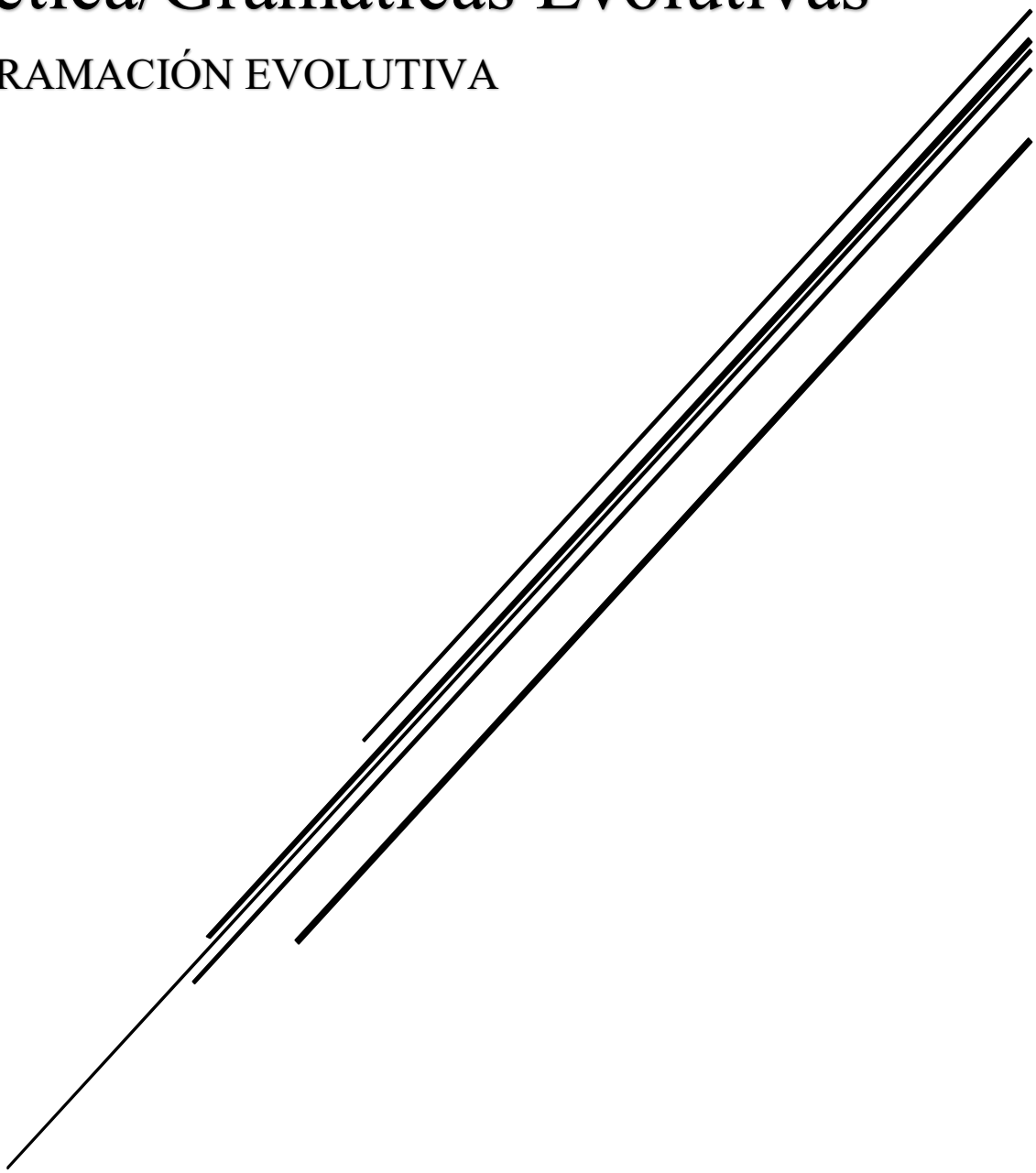
MEMORIA PRÁCTICA III: “Programación genética/Gramáticas Evolutivas”

PROGRAMACIÓN EVOLUTIVA

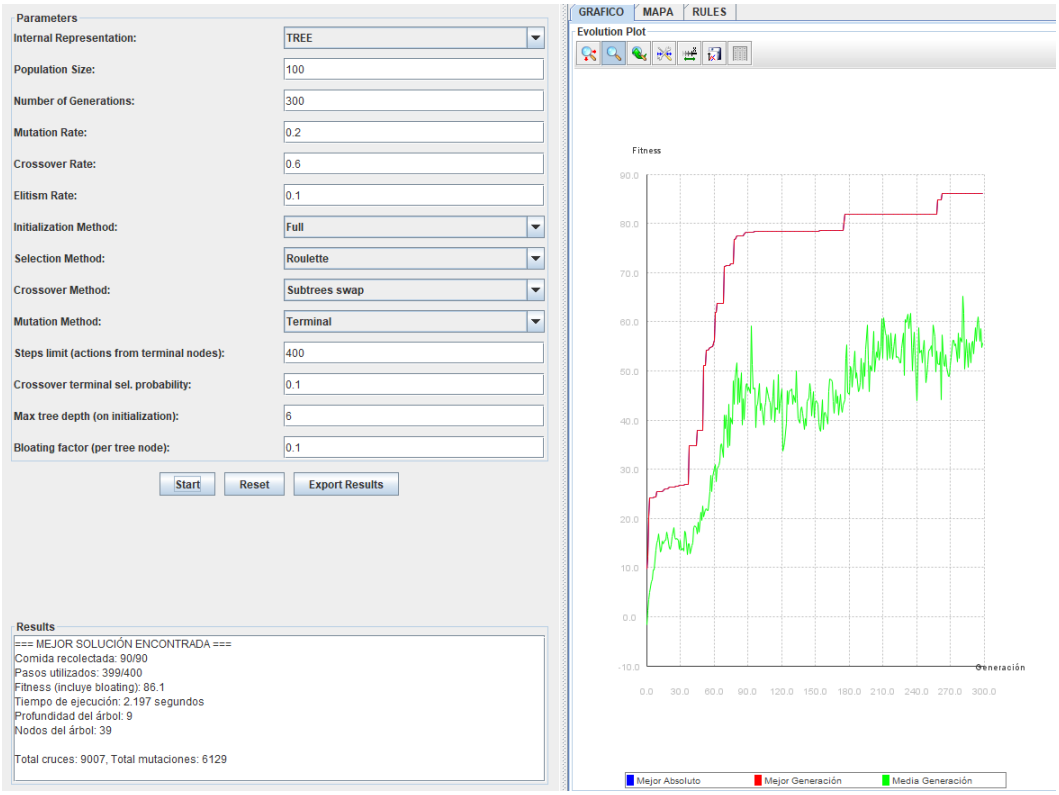


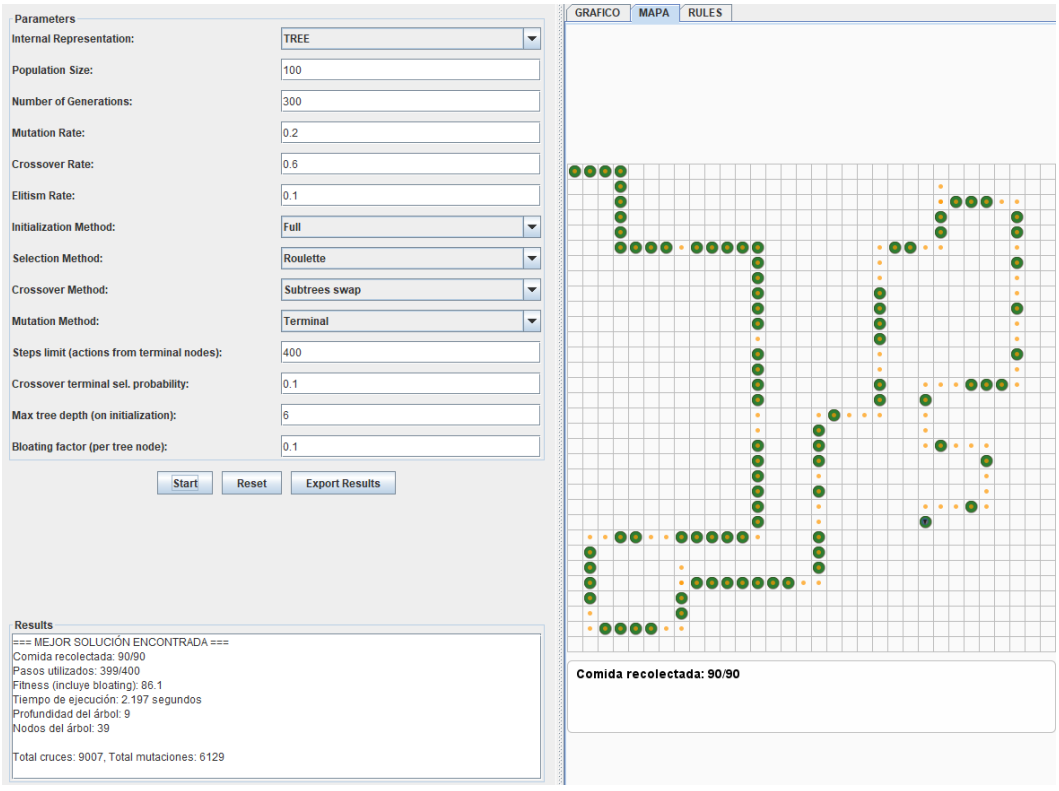
Grado en Ingeniería de Datos e Inteligencia Artificial
Artem Vartanov y Mario López Díaz

ÍNDICE

1. Capturas de resultados

PROGRAMACIÓN GENÉTICA





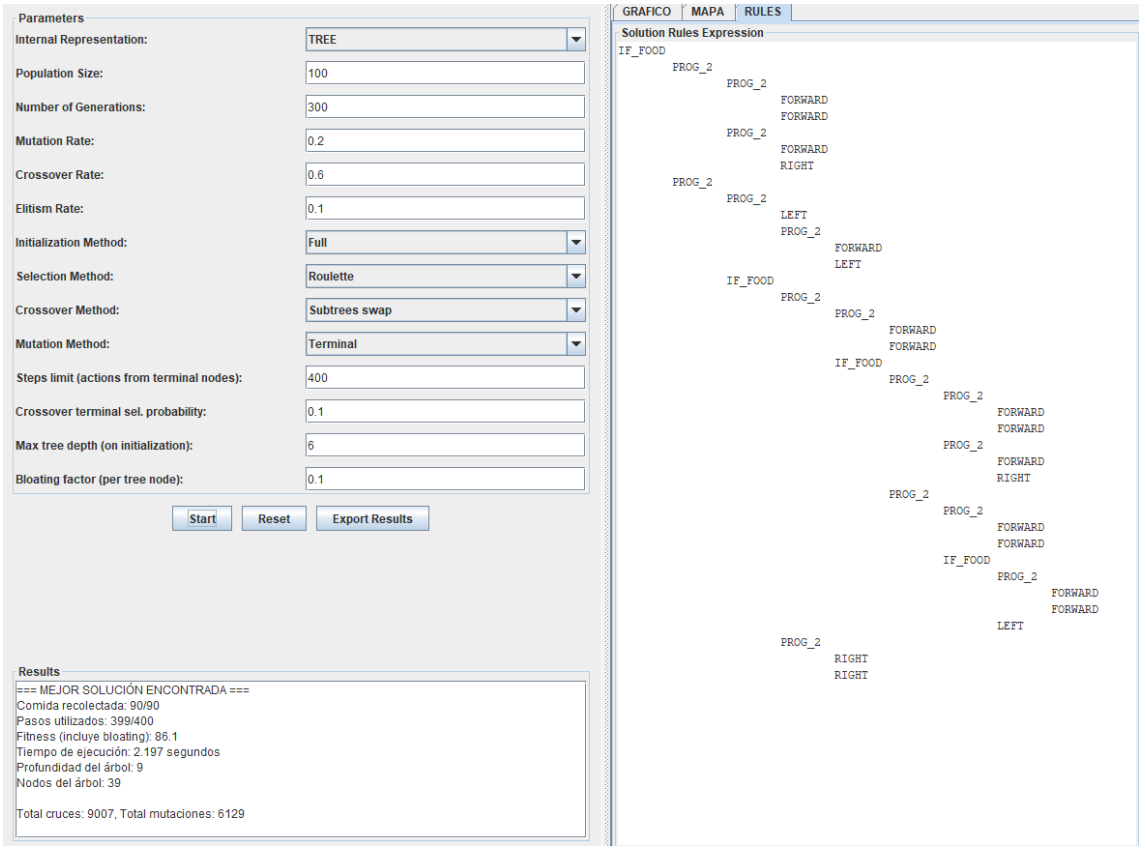
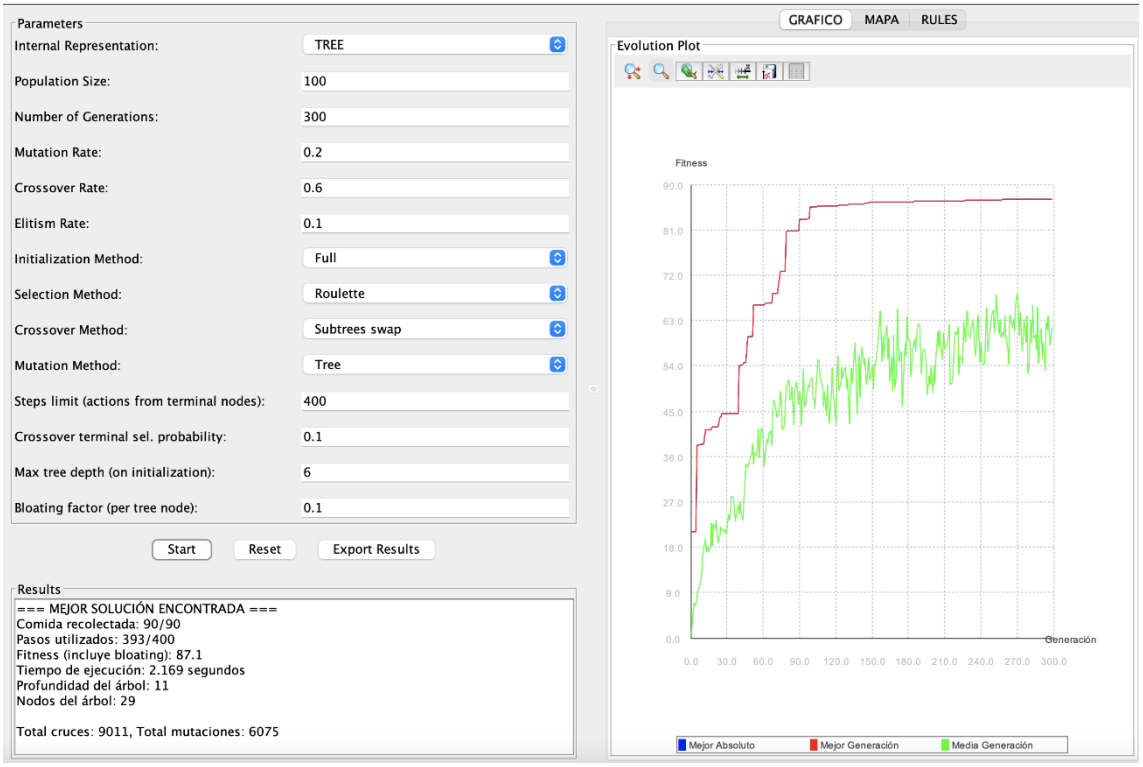
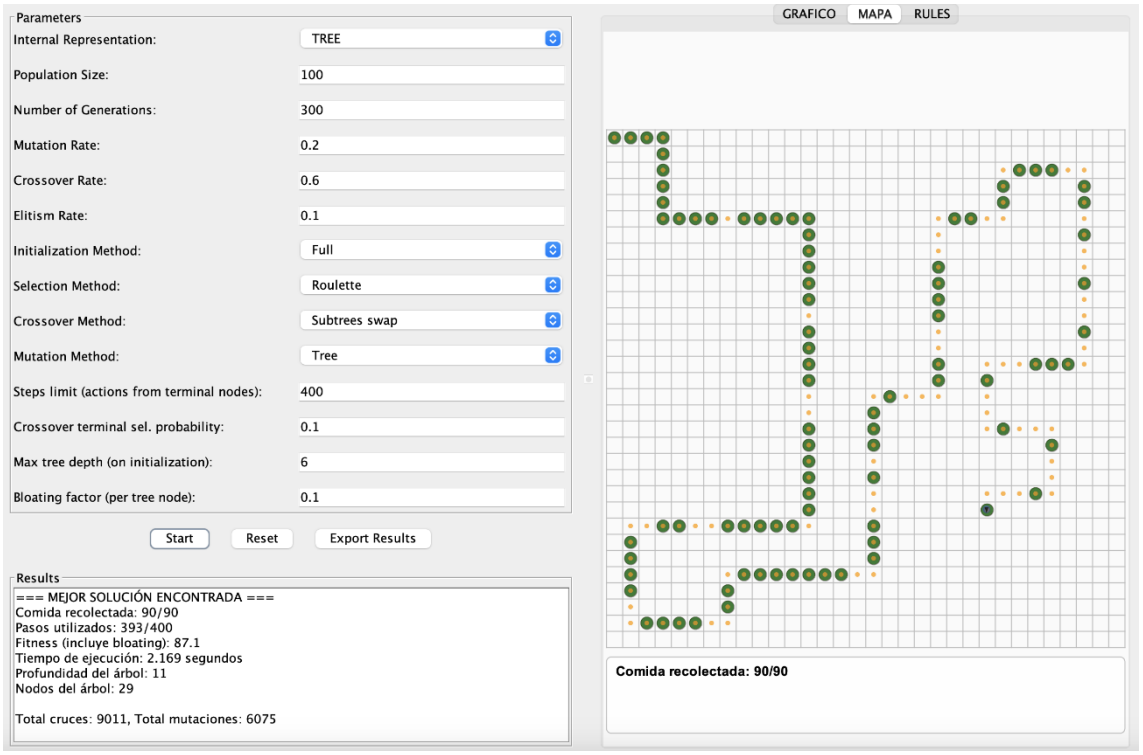


Figura 1.1: Ejecución con programación genética, método de inicialización full, selección por ruleta, cruce de subárboles y mutación terminal.


Parameters
Internal Representation: TREE
Population Size: 100
Number of Generations: 300
Mutation Rate: 0.2
Crossover Rate: 0.6
Elitism Rate: 0.1
Initialization Method: Full
Selection Method: Roulette
Crossover Method: Subtrees swap
Mutation Method: Tree
Steps limit (actions from terminal nodes): 400
Crossover terminal sel. probability: 0.1
Max tree depth (on initialization): 6
Bloating factor (per tree node): 0.1
Start
Reset
Export Results
Results
=== MEJOR SOLUCIÓN ENCONTRADA ===
Comida recolectada: 90/90
Pasos utilizados: 393/400
Fitness (incluye bloating): 87.1
Tiempo de ejecución: 2.169 segundos
Profundidad del árbol: 11
Nodos del árbol: 29
Total cruces: 9011, Total mutaciones: 6075
GRAFICO
MAPA
RULES
Evolution Plot
Fitness
Generación
Mejor Absoluto
Mejor Generación
Media Generación


Parameters
Internal Representation: TREE
Population Size: 100
Number of Generations: 300
Mutation Rate: 0.2
Crossover Rate: 0.6
Elitism Rate: 0.1
Initialization Method: Full
Selection Method: Roulette
Crossover Method: Subtrees swap
Mutation Method: Tree
Steps limit (actions from terminal nodes): 400
Crossover terminal sel. probability: 0.1
Max tree depth (on initialization): 6
Bloating factor (per tree node): 0.1
Start
Reset
Export Results
Results
=== MEJOR SOLUCIÓN ENCONTRADA ===
Comida recolectada: 90/90
Pasos utilizados: 393/400
Fitness (incluye bloating): 87.1
Tiempo de ejecución: 2.169 segundos
Profundidad del árbol: 11
Nodos del árbol: 29
Total cruces: 9011, Total mutaciones: 6075
GRAFICO
MAPA
RULES
Comida recolectada: 90/90

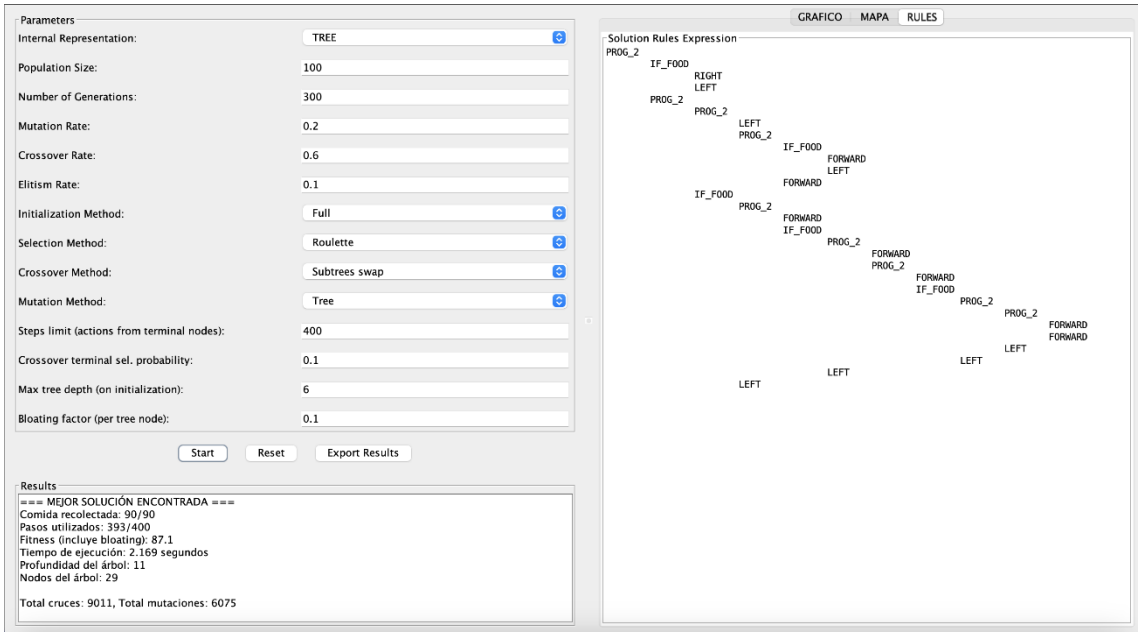
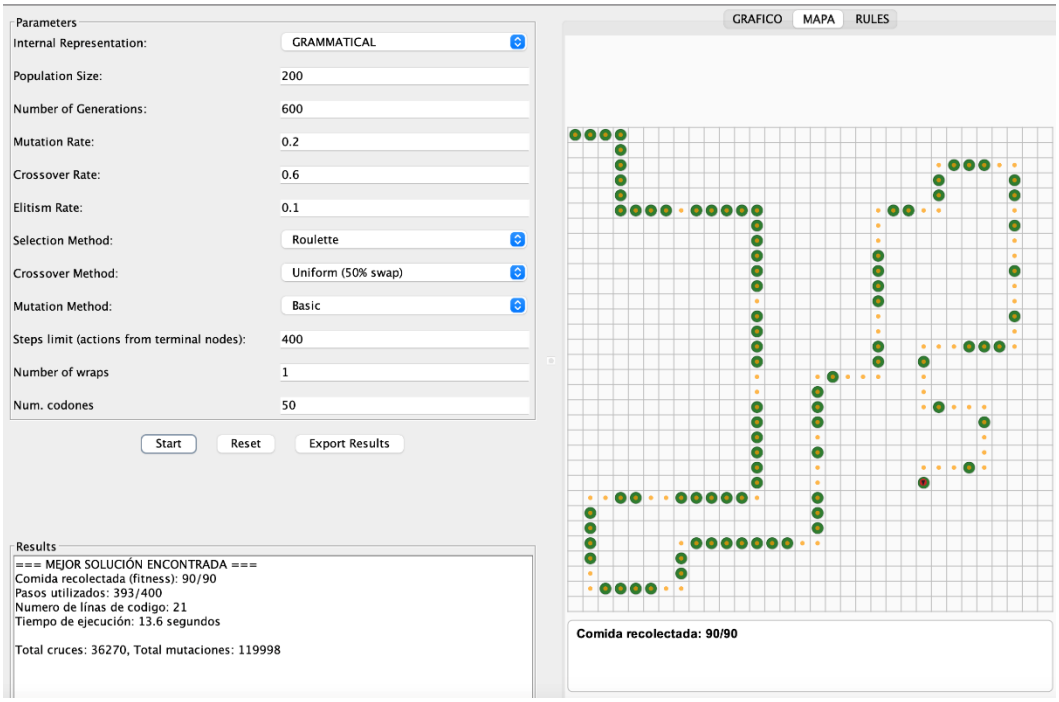


Figura 1.2: Ejecución con programación genética, método de inicialización full, selección por ruleta, cruce de subárboles y mutación árbol.

GRAMÁTICA EVOLUTIVA



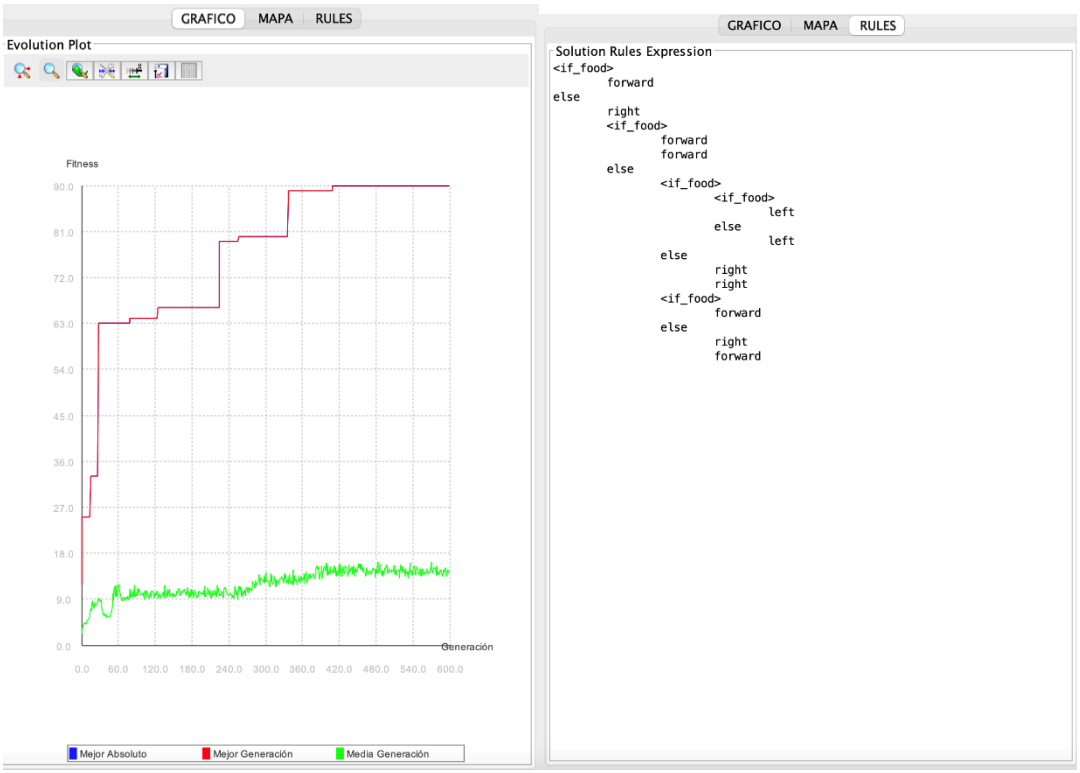
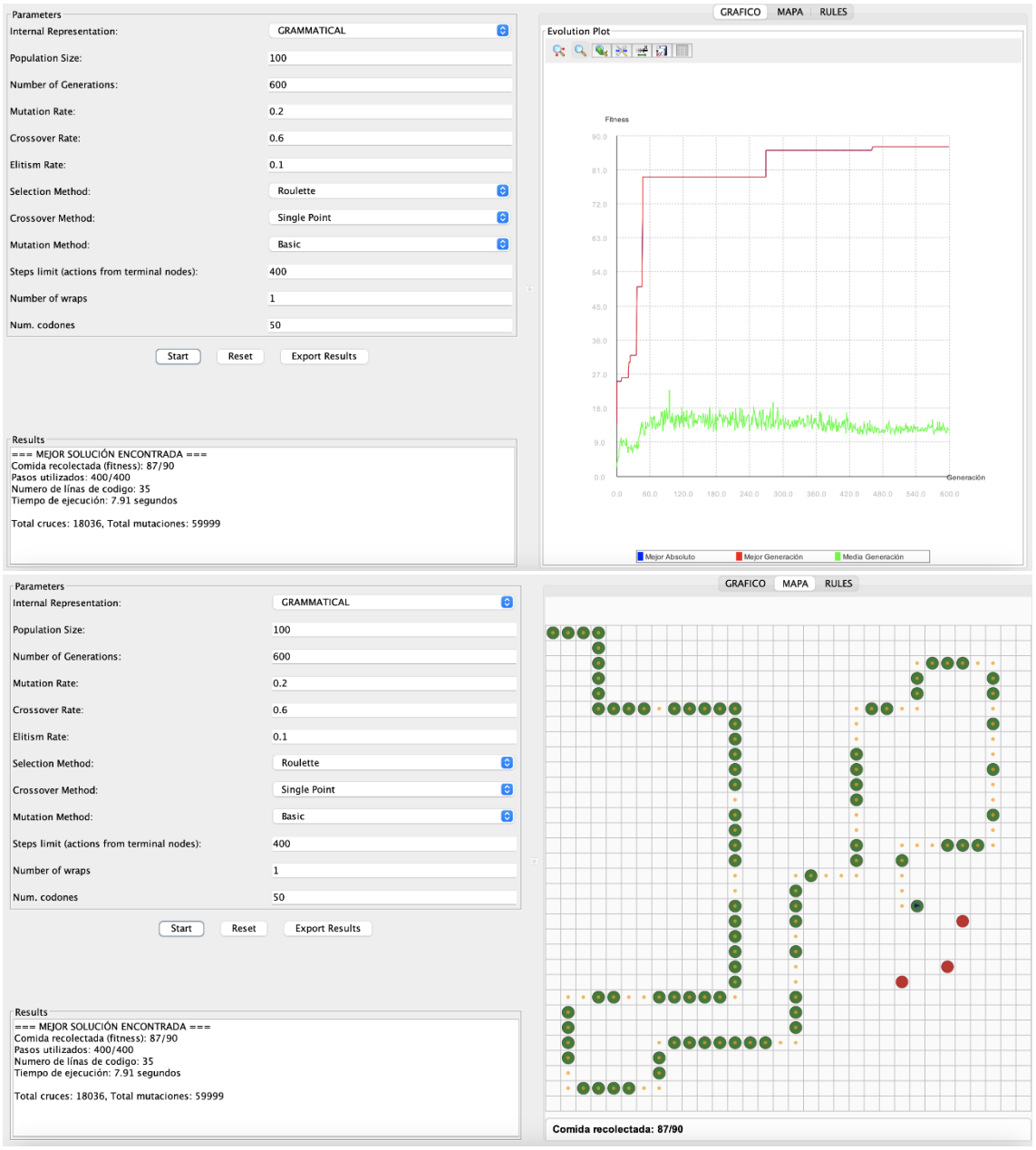


Figura 1.3: Ejecución con representación gramática, tamaño de población 200, número de generaciones 600 y número de codones 50


Parameters
Internal Representation: GRAMMATICAL
Population Size: 100
Number of Generations: 600
Mutation Rate: 0.2
Crossover Rate: 0.6
Elitism Rate: 0.1
Selection Method: Roulette
Crossover Method: Single Point
Mutation Method: Basic
Steps limit (actions from terminal nodes): 400
Number of wraps 1
Num. codones 50
Start
Reset
Export Results
Results
=== MEJOR SOLUCIÓN ENCONTRADA ===
Comida recolectada (fitness): 87/90
Pasos utilizados: 400/400
Numero de líneas de codigo: 35
Tiempo de ejecución: 7.91 segundos
Total cruces: 18036, Total mutaciones: 59999
GRAFICO
MAPA
RULES
Evolution Plot
Fitness
Generación
Mejor Absoluto
Mejor Generación
Media Generación
Comida recolectada: 87/90

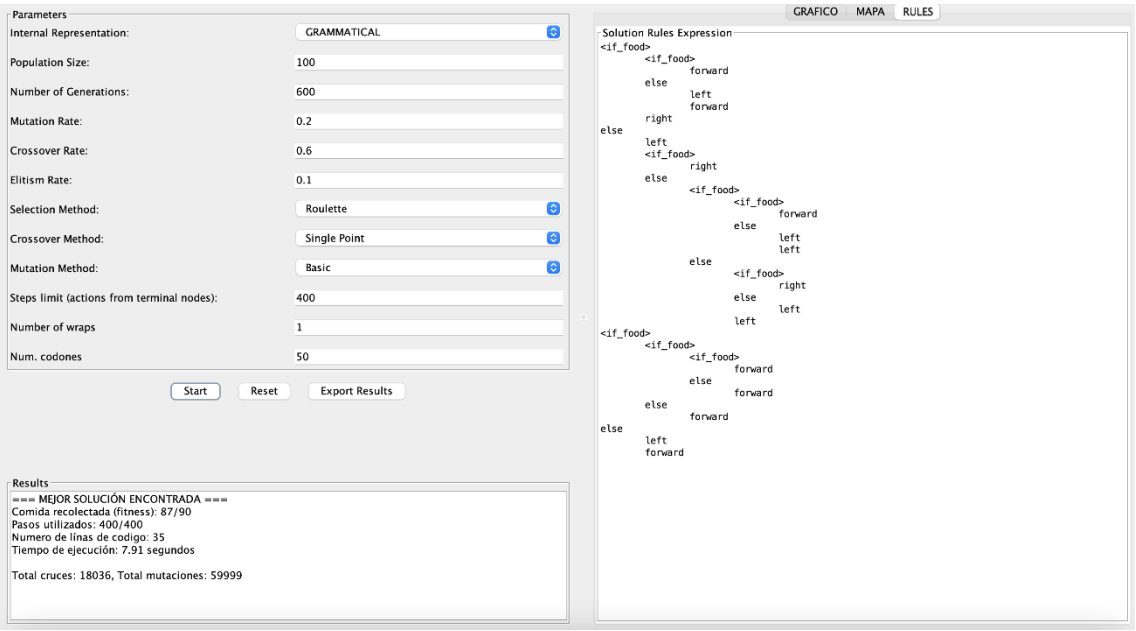


Figura 1.4: Ejecución con representación gramática, tamaño de población 100, número de generaciones 600 y número de codones 50

2. Observaciones de la práctica

Programación Genética (PG):

La PG demostró ser muy efectiva y rápida para encontrar soluciones, generando estrategias sofisticadas que permitían a la hormiga recorrer eficientemente el rastro de Santa Fe.

Se observó claramente el fenómeno de bloating, donde los programas tendían a crecer excesivamente sin mejorar significativamente su fitness.

Gramáticas Evolutivas (GE):

Las GE parecen ser menos adecuadas al problema, ya que el óptimo se alcanzaban de manera menos frecuente. Además, para evitar errores de evaluación de un código inválido, le asignábamos un fitness de 0 para descartar a ese individuo. El ajuste del número de codones lo hemos tenido que añadir en la interfaz, puesto que también era un parámetro importante para la optimización, aunque tampoco mejoraba muchísimo el resultado si se incrementaba sustancialmente.

Dificultades principales:

El principal desafío en PG fue encontrar la forma ideal para el control del bloating, que al final hemos usado una penalización por cada nodo del árbol, debido a que otras estrategias no brindaban resultados adecuados.

En GE, la definición inicial de la gramática requirió varios ajustes para lograr una buena optimización.

3. Arquitectura del código

La mayor parte de la arquitectura ha sido reutilizada de la anterior práctica. Con rojo hemos puesto lo más relevante / novedoso.

1. Paquete es.ucm

Este paquete contiene las clases principales que gestionan la ejecución del algoritmo genético y la interfaz gráfica. Dentro de la mayoría de sus subpaquetes aparecen 2 subpaquetes (grammar y tree) que hacen referencia a las 2 formas de afrontar el problema (gramáticas evolutivas y programación genética).

Clase AlgoritmoGenetico: Gestiona la ejecución del algoritmo genético.

Clase Main: Implementa la interfaz gráfica para configurar y ejecutar el algoritmo genético pasándole los parámetros y sacando de él los resultados para mostrarlos en la gráfica.

Clase FoodMapPanel: Representa un panel gráfico que dibuja el rastro con los 90 trozos de comida para la hormiga.

2. Paquete es.ucm.factories

Este paquete contiene las fábricas de individuos, que se utilizan para crear individuos de diferentes tipos (según la función a optimizar).

Clase IndividuoFactory: Define una interfaz para crear individuos.

Clase IndividuoHormigaArbolFactory: Implementa fábrica específica para el tipo de individuo (función a optimizar), en concreto el individuo de la hormiga para la programación genética.

Clase IndividuoHormigaGrammarFactory: Implementa fábrica específica para el tipo de individuo (función a optimizar), en concreto el individuo de la hormiga para la gramática evolutiva.

3. Paquete es.ucm.individuos

Este paquete contiene las clases que representan a los individuos de la población, la clase para calcular el fitness, los nodos para construir el árbol, la gramática...

Clase IndividuoHormigaArbol: representa una solución de programación genética.

Clase AbstractNode: Clase abstracta que representa el nodo del árbol de la solución. Se materializa en **ForwardNode, LeftNode, RightNode, Prog2Node, Prog3Node, IfFoodNode**. Los últimos guardan sus hijos, mientras que los primeros son nodos terminales.

Clases Coord y DirectionEnum: para representar la posición y la dirección de la hormiga.

Clases Hormiga: Clase auxiliar para calcular el fitness. Guarda el estado de la hormiga, conforme se van aplicando las acciones y evita que haga demás pasos de los establecidos.

Clase IndividuoHormigaGramatica: representa una solución de gramática evolutiva.

Clase AbstractGrammarElem: Clase abstracta que representa la acción abstracta usada en las gramáticas. Se materializa en **ForwardGrammarElem, LeftGrammarElem, RightGrammarElem, IfFoodGrammarElem.**

Clase Grammar: Clase que decodifica usando la gramática definida abajo un individuo con genotipo en forma de lista de enteros, devolviendo una especie de “código”.

La gramática BNF que hemos usado es la siguiente:

<start> ::= <code>

<code> ::= <line> | <line> <line>

<line> ::= <if_food> | <action>

<if_food> ::= <code> <code>

<action> ::= left | right | forward

Gracias al ajuste de la gramática, hemos podido alcanzar un óptimo de 90 trozos comidos. (principal factor que mejoraba el resultado). Penalizamos soluciones de código incorrecto, de modo que su fitness es 0.

4. Paquete es.ucm.genes

Este paquete contiene las clases que representan los genes de los individuos.

Clase Gen: Define una interfaz para los genes.

Clase IntegerGen: Representa un gen cuyo genotipo es entero.

Clase TreeGen: Representa un gen cuyo genotipo es un árbol. El cromosoma solo tiene un único TreeGen que se va modificando.

5. Paquete es.ucm.selection

Este paquete contiene las clases que implementan los métodos de selección de individuos.

Clase AbstractSelection: Define una interfaz para los métodos de selección.

Clases RouletteSelection, TorneoSelection, etc.: Implementan métodos de selección específicos (ruleta, torneo, truncamiento, etc.)

6. Paquete es.ucm.cross

Este paquete contiene las clases que implementan los métodos de cruce.

Clase AbstractCross: Define una interfaz para los métodos de cruce.

Clase SubtreeSwapCross: Implementa método de cruce específico para el problema de programación genética.

Clase SinglePointCross, UnifromCross: Implementan métodos de cruce específico para el problema de gramáticas evolutivas.

7. Paquete es.ucm.mutation

Este paquete contiene las clases que implementan los métodos de mutación.

Clase AbstractMutate: Define una interfaz para los métodos de mutación.

Clase ContractionMutate, ExpansionMutate, etc.: Implementan métodos de mutación específicos para el problema programación genética.

Clase BasicMutate: Implementan métodos de mutación específicos para el problema gramáticas evolutivas.

8. Paquete es.ucm.mansion

Este paquete contiene las clases para representar el mapa, calcular las rutas, calcular el fitness...

Clase AbstractFoodMap: Representa un mapa abstracto del rastro de la comida. Se usa al calcular el fitness, puesto que se simula un recorrido de comida (eliminando trozos si se come) hasta llegar al límite de pasos o comer todo.

Clase SantaFeMap: Representación del mapa del problema propuesto, definiendo la ubicación de los trozos de comida y las dimensiones del área.

9. Paquete es.ucm.utils

Clase RandomUtil: Proporciona utilidades para generar N números aleatorios diferentes.

10. Paquete es.ucm.initilizer

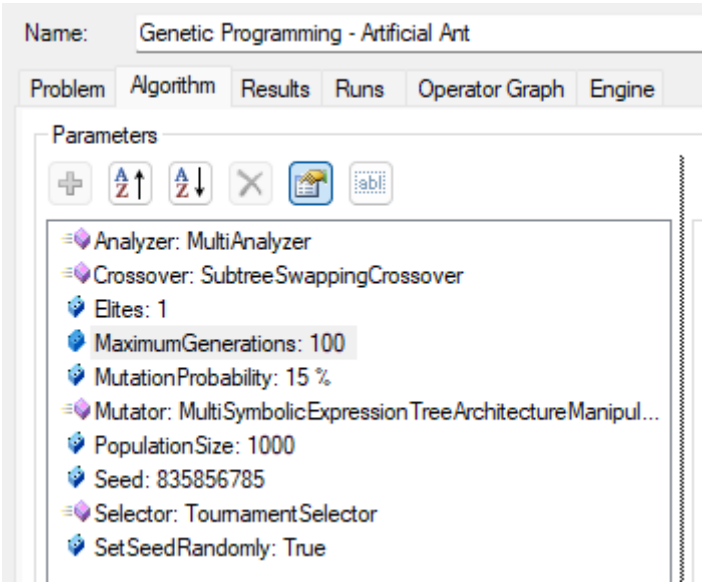
Este paquete contiene las clases que configuran el método de inicialización. (se usan en programación genética)

Clase AbstractInitializer: Define una interfaz para los métodos de inicialización.

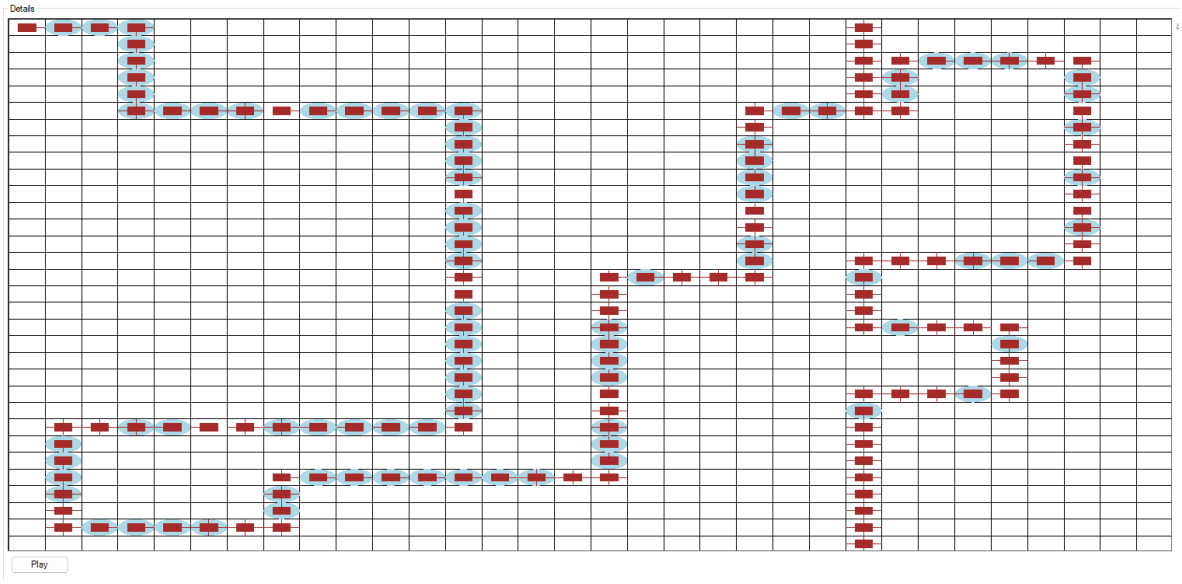
Clase FULLInitializer, GrowInitializer, RampedHalfInitializer: Implementan los métodos de inicialización específico para el problema propuesto.

4. HeuristicLab

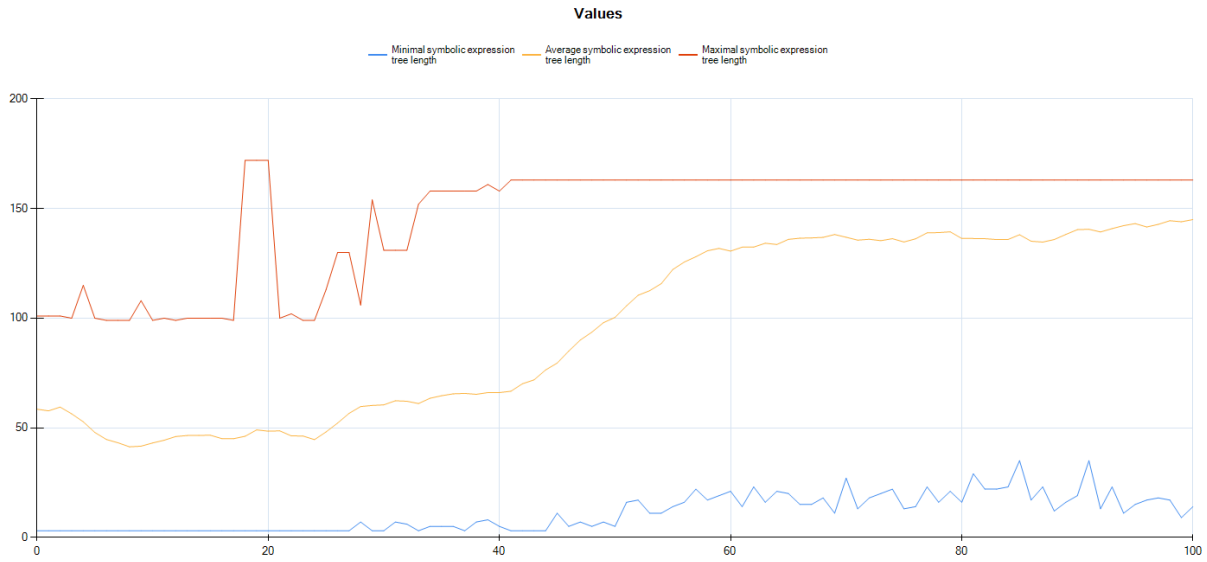
EJEMPLO PROGRAMACIÓN GENÉTICA



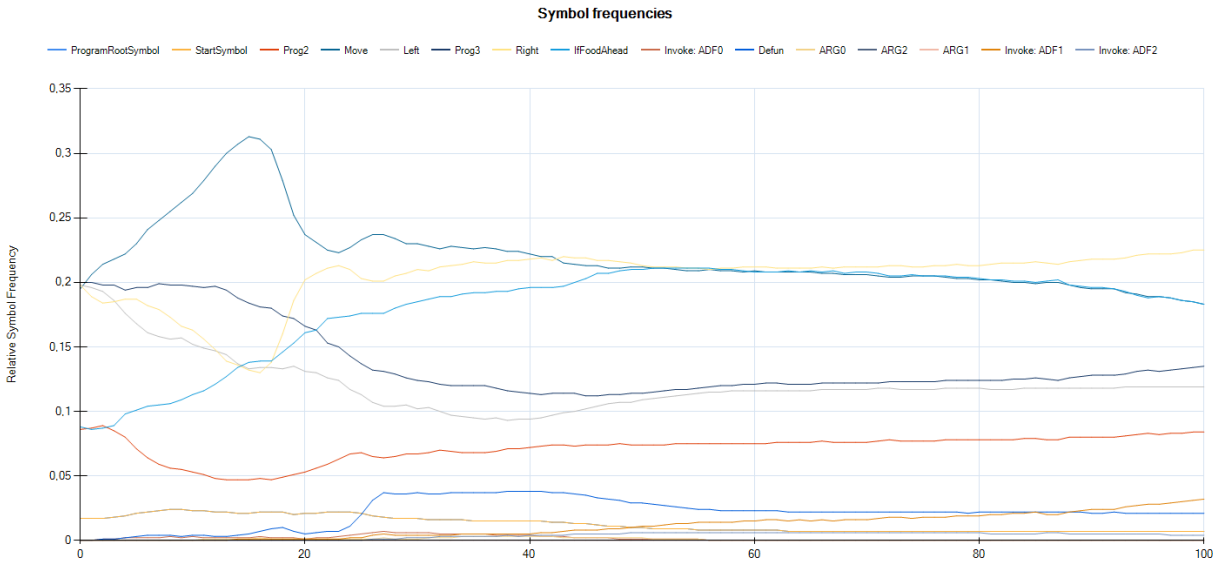
Parámetros empleados para el problema de programación genética



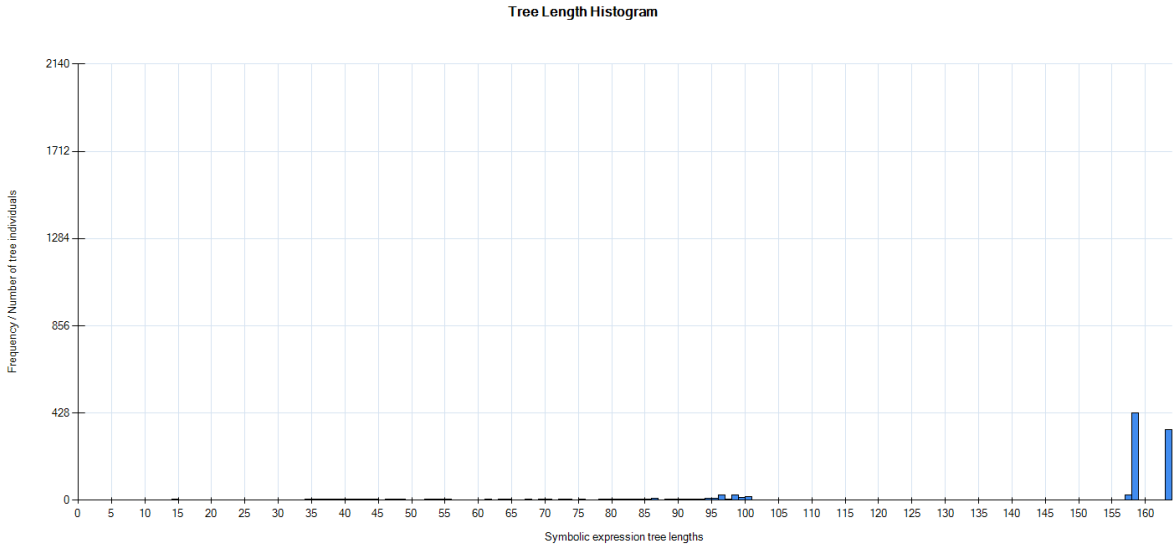
Rastro óptimo del individuo comiendo los 90 trozos de comida



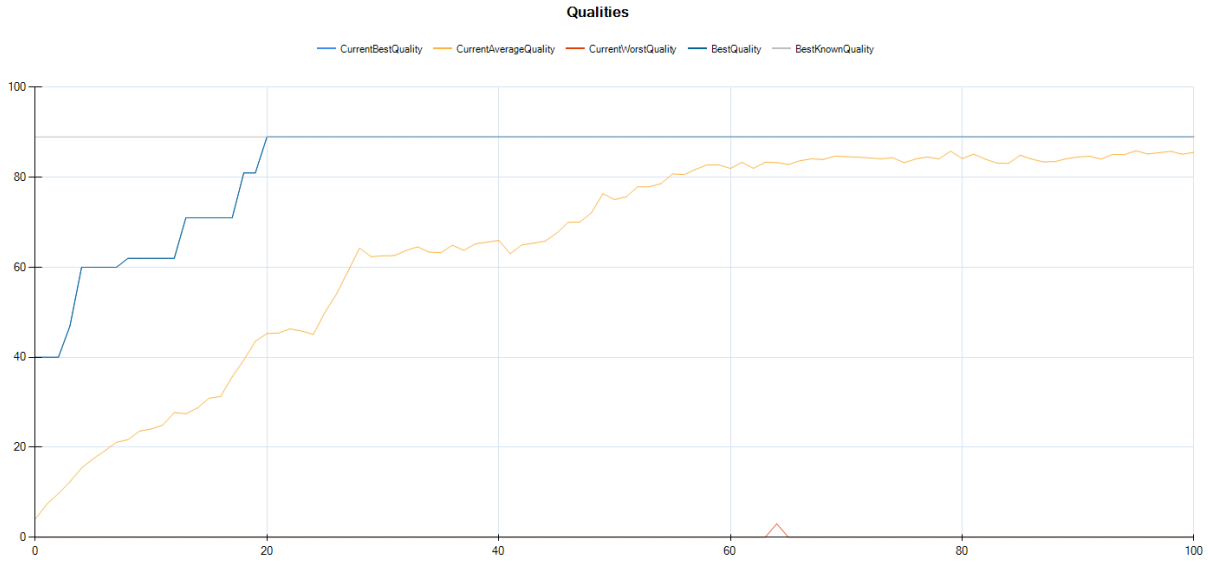
Gráfica con valores de la “symbolic expresión tree length”



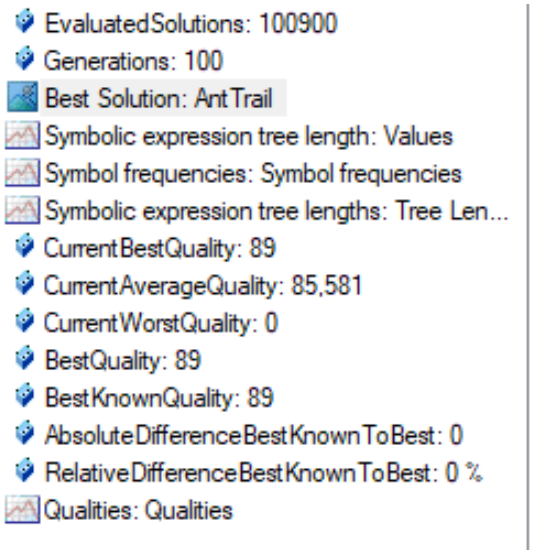
Gráfica con frecuencia de los símbolos



Histograma con longitud de los árboles generados

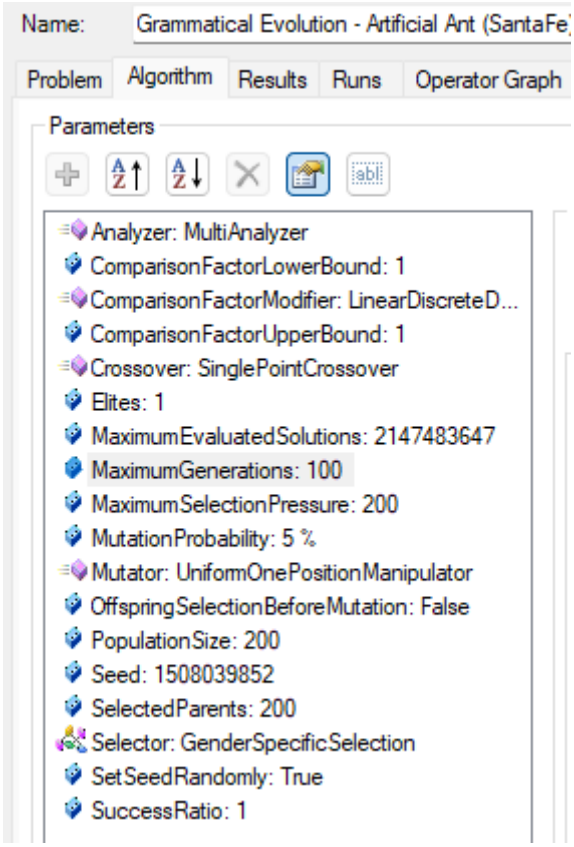


Gráfica con valores de las cualidades de las soluciones

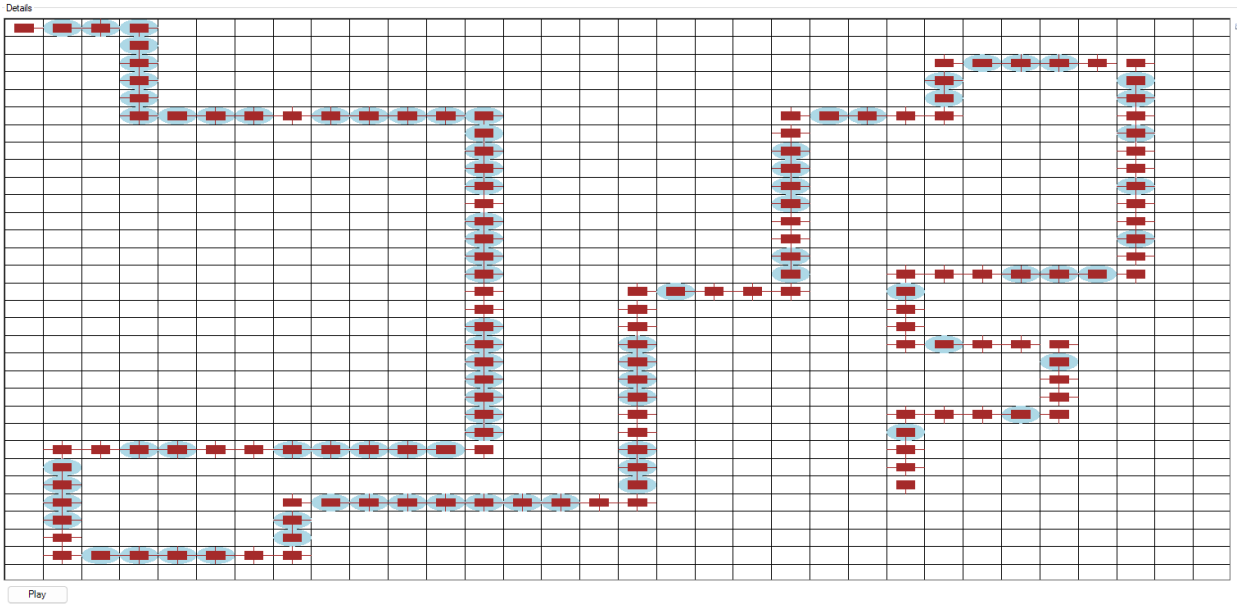


Resultados finales

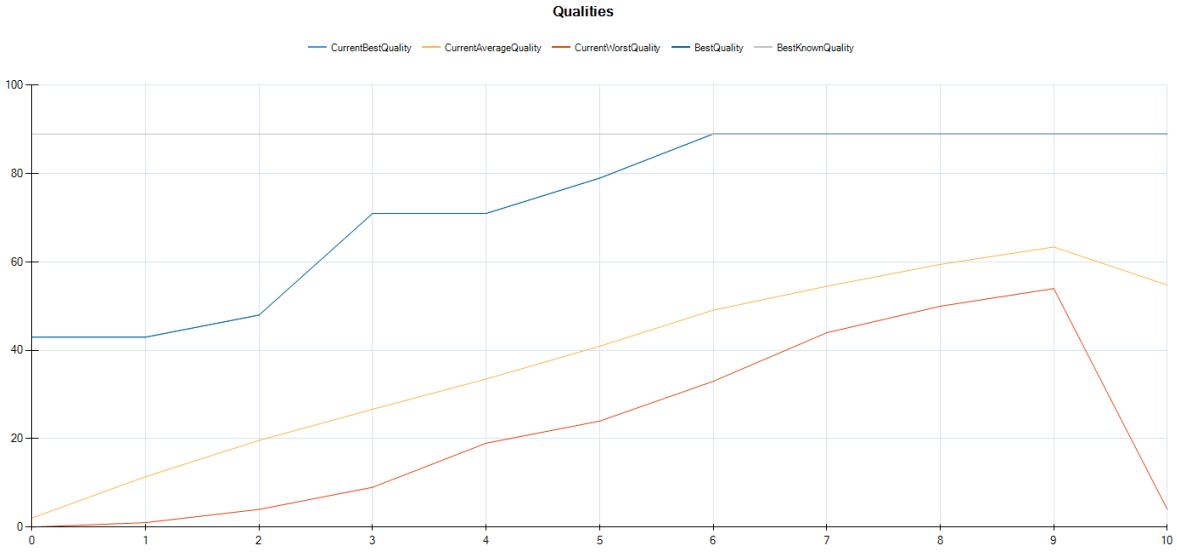
EJEMPLO GRAMÁTICA EVOLUTIVA



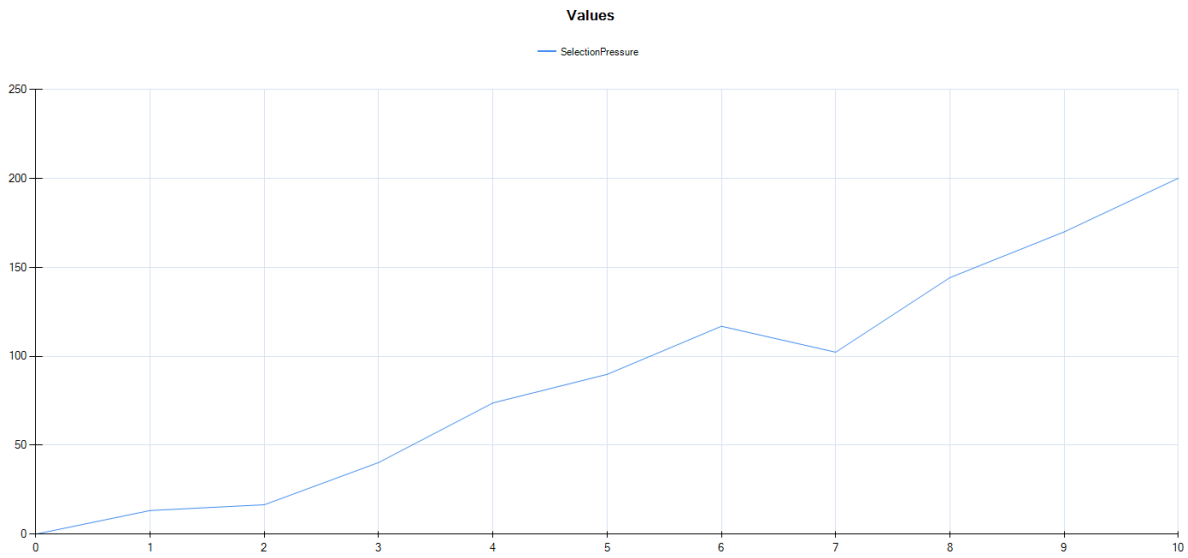
Parámetros empleados para el problema de gramática evolutiva



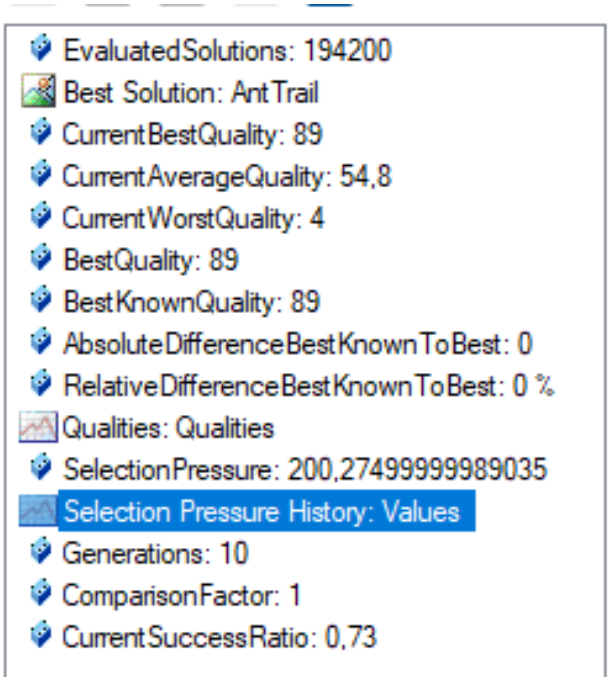
Rastro óptimo del individuo comiendo los 90 trozos de comida



Gráfica con valores de las cualidades de las soluciones



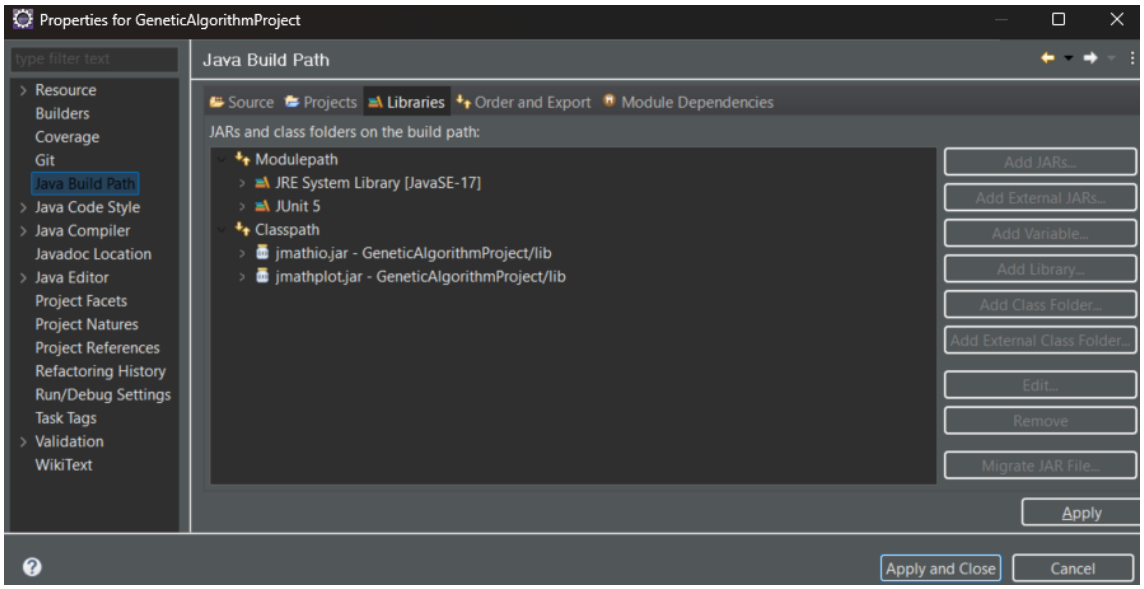
Gráfica con el valor de la presión de selección



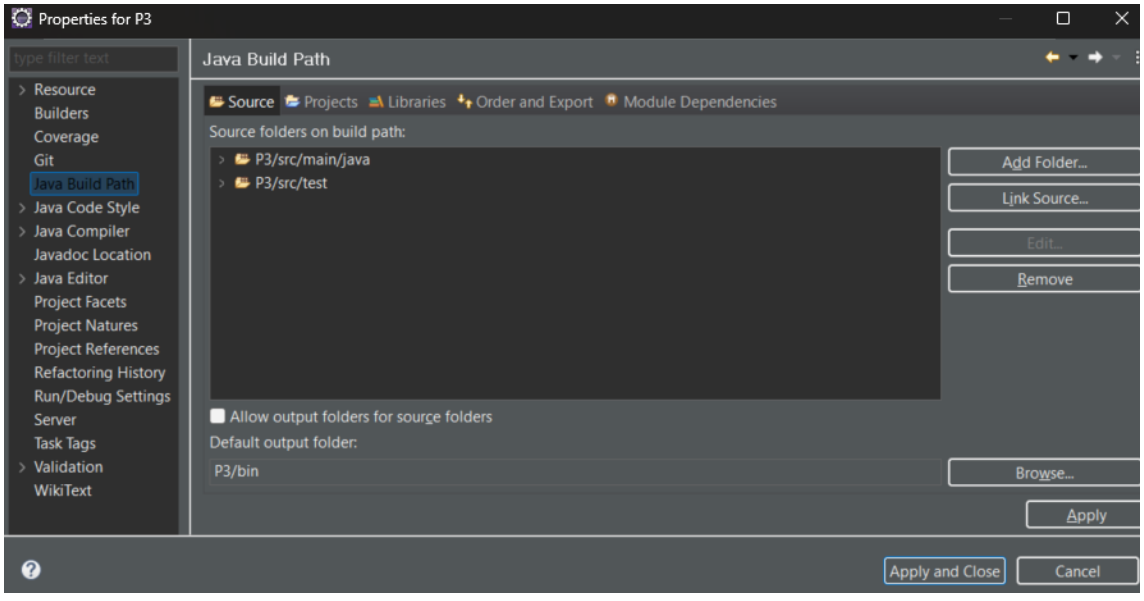
Resultados finales

5. Ejecución del proyecto

Para ejecutar el código es necesario importarlo como proyecto de Java en Eclipse. Posteriormente, antes de poder darle al *run,* es necesario asegurarse de que las propiedades del proyecto tienen la siguiente configuración aplicada:



Properties -> Java Build Path -> Libraries -> Archivos .jar



Properties -> Java Build Path -> Source -> src/main/java, src/test

Tras asegurarse de que el proyecto está bien configurado, sólo quedará elegir la configuración del run para poder ejecutarlo, el cual será el archivo Main.java.

6. Distribución del trabajo

Artem: Ha implementado la estructura para almacenar los árboles y calcular el fitness junto con el control de *bloating* y por otra parte la representación y decodificación de individuos en gramática evolutiva.

Mario: Ha implementado la representación del mapa, las mutaciones e inicializaciones para la programación evolutiva y ajustado la interfaz para que se muestre correctamente la solución en el mapa.

7. Conclusiones

La práctica permitió comparar dos estrategias en el problema de la hormiga de Santa Fe (programación genética y gramáticas evolutivas). La PG demostró mayor eficacia, evolucionando árboles de comportamiento complejos mediante cruce y mutación, aunque con riesgo de *bloating*. Las GE, aunque más simples, mostraron limitaciones en escalabilidad.

Ambos métodos lograron soluciones viables, pero la PG superó a las GE en consistencia y adaptabilidad. Los resultados destacan la importancia de ajustar operadores (profundidad, *wraps*) para evitar estancamientos. Una combinación de ambas técnicas o mejoras en la función de *fitness* podrían optimizar aún más el rendimiento.

En conclusión, la PG es más adecuada para este problema, pero las GE ofrecen un enfoque alternativo con menor complejidad inicial. La práctica reforzó la necesidad de elegir la representación correcta según el problema.