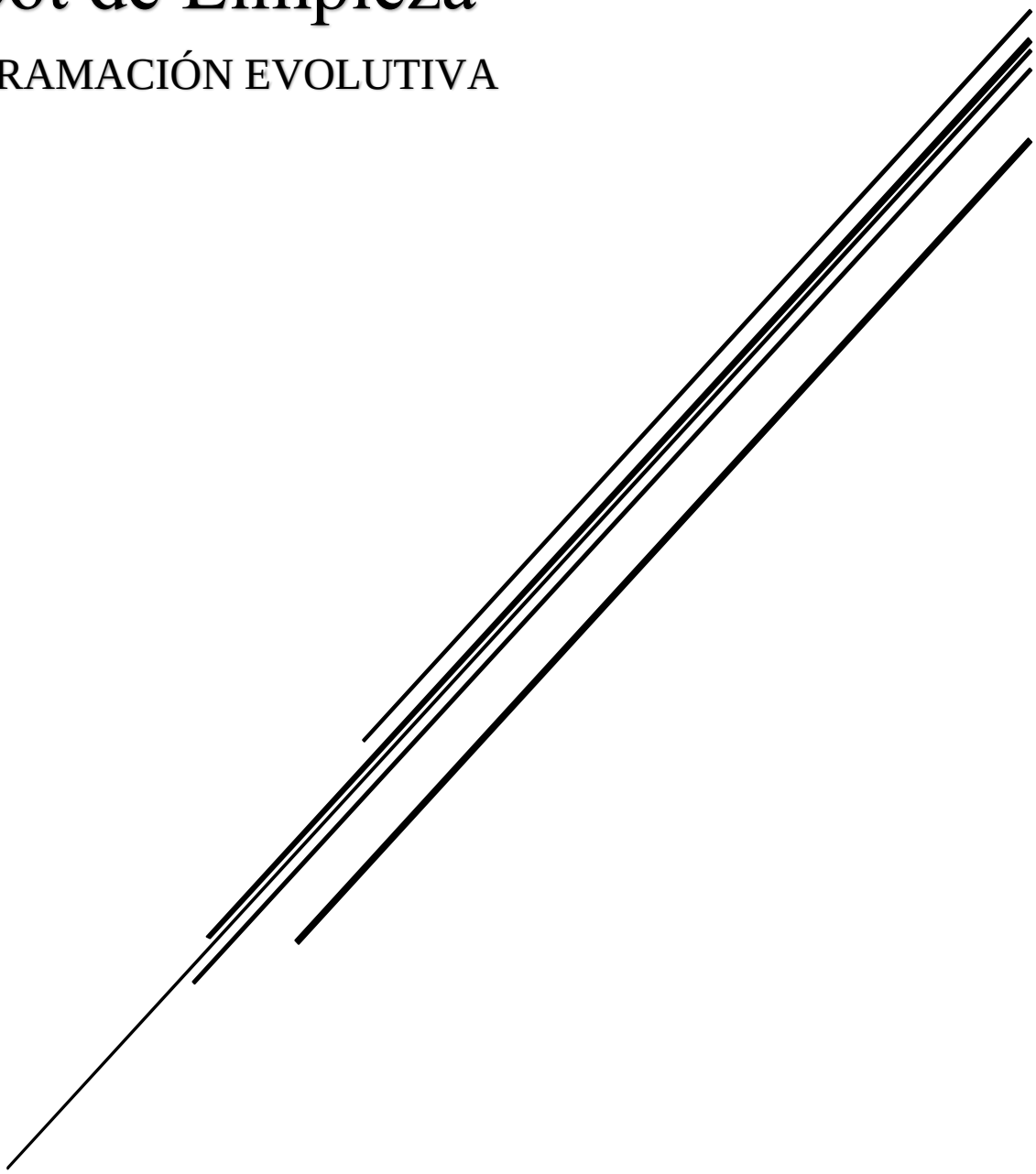
MEMORIA PRÁCTICA II: “Optimización de la Ruta de un Robot de Limpieza”

PROGRAMACIÓN EVOLUTIVA



Grado en Ingeniería de Datos e Inteligencia Artificial

Artem Vartanov y Mario López Díaz

ÍNDICE

1. Capturas de resultados

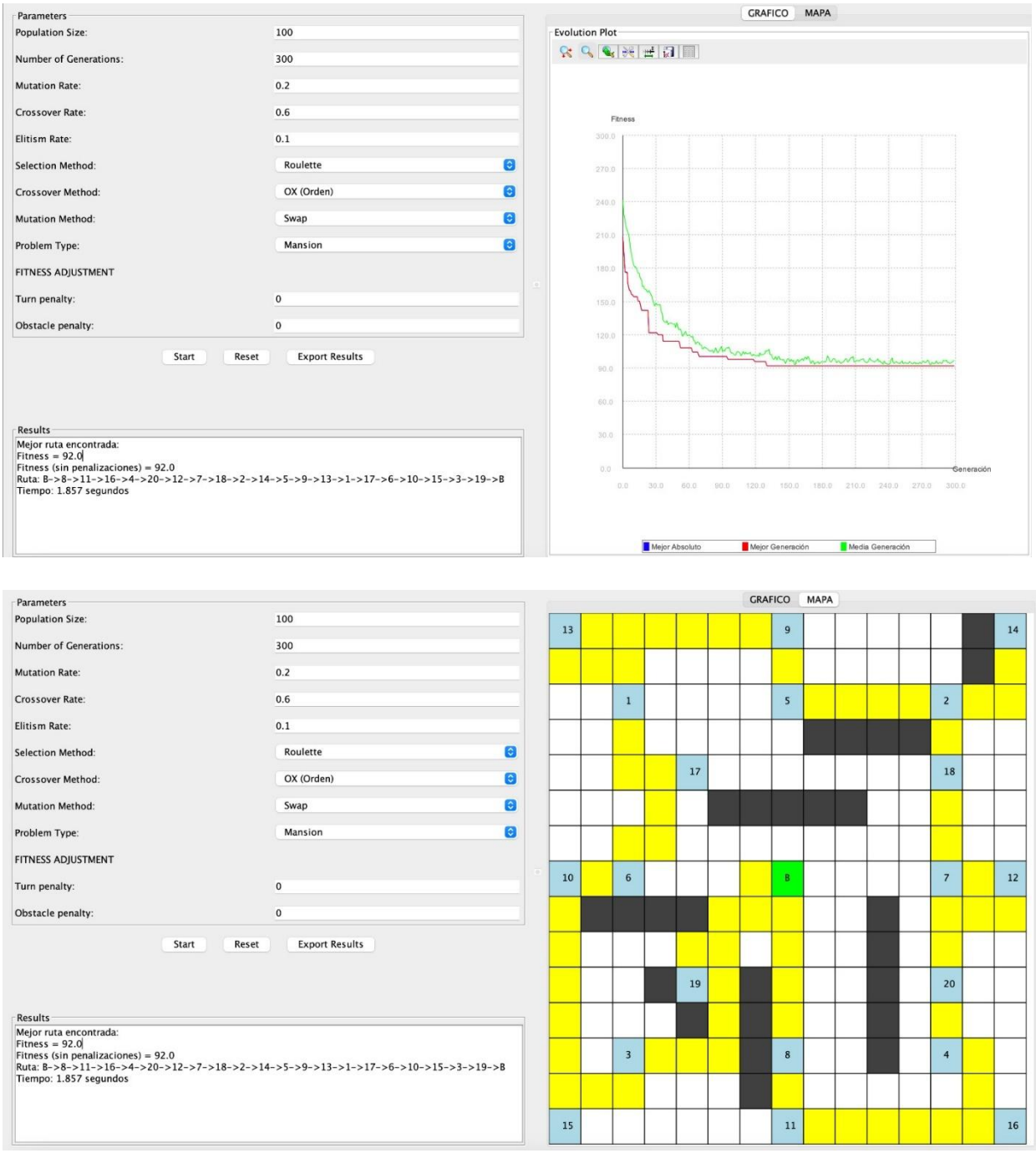
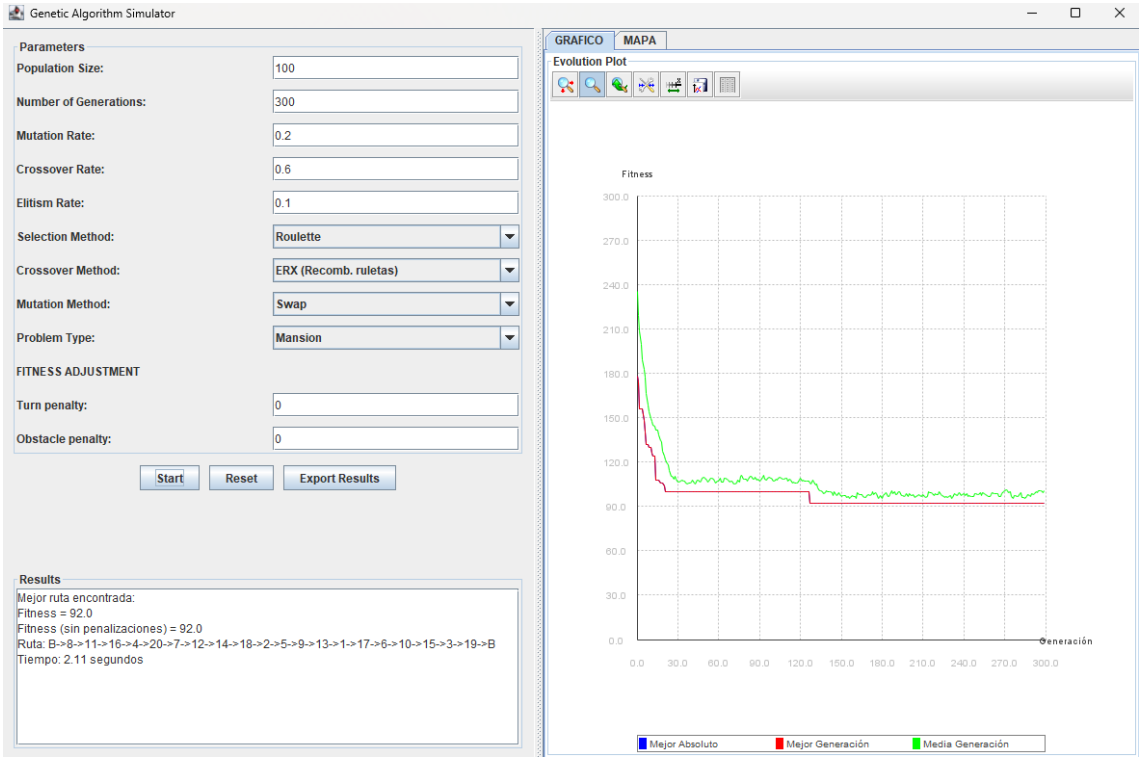


Figura 1.1 Ejecución con elitismo (10%), 300 generaciones, selección por ruleta, cruce OX y mutación por intercambio.



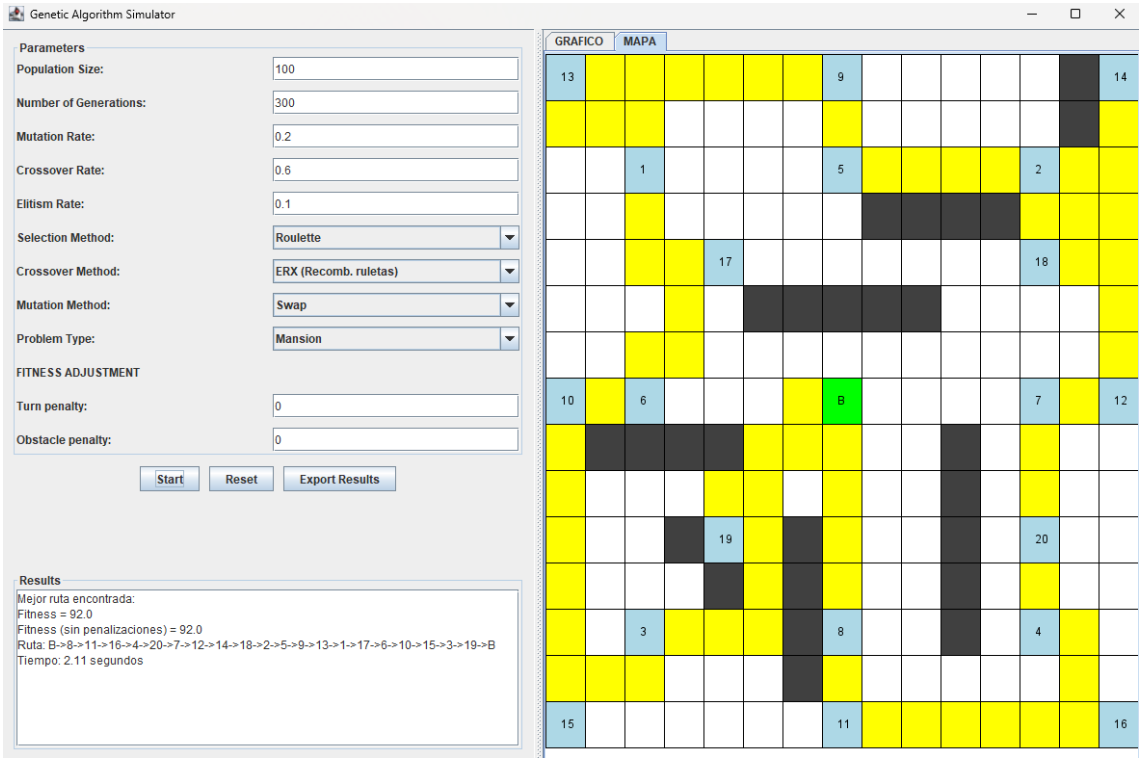


Figura 1.2 Ejecución con elitismo (10%), 300 generaciones, selección por ruleta, cruce ERX y mutación por intercambio.

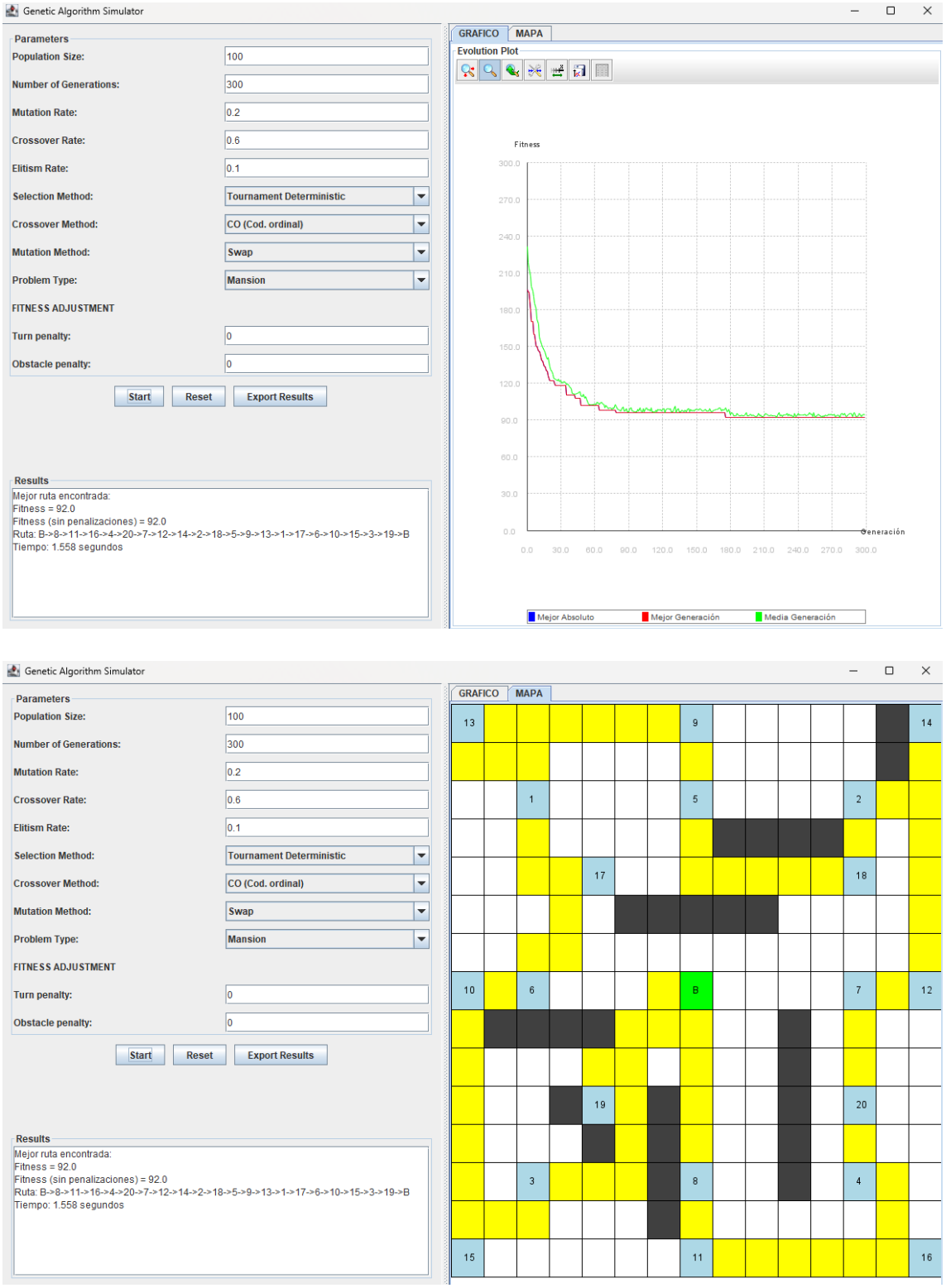


Figura 1.3 Ejecución con elitismo (10%), 300 generaciones, selección por torneo determinista, cruce CO y mutación por intercambio.

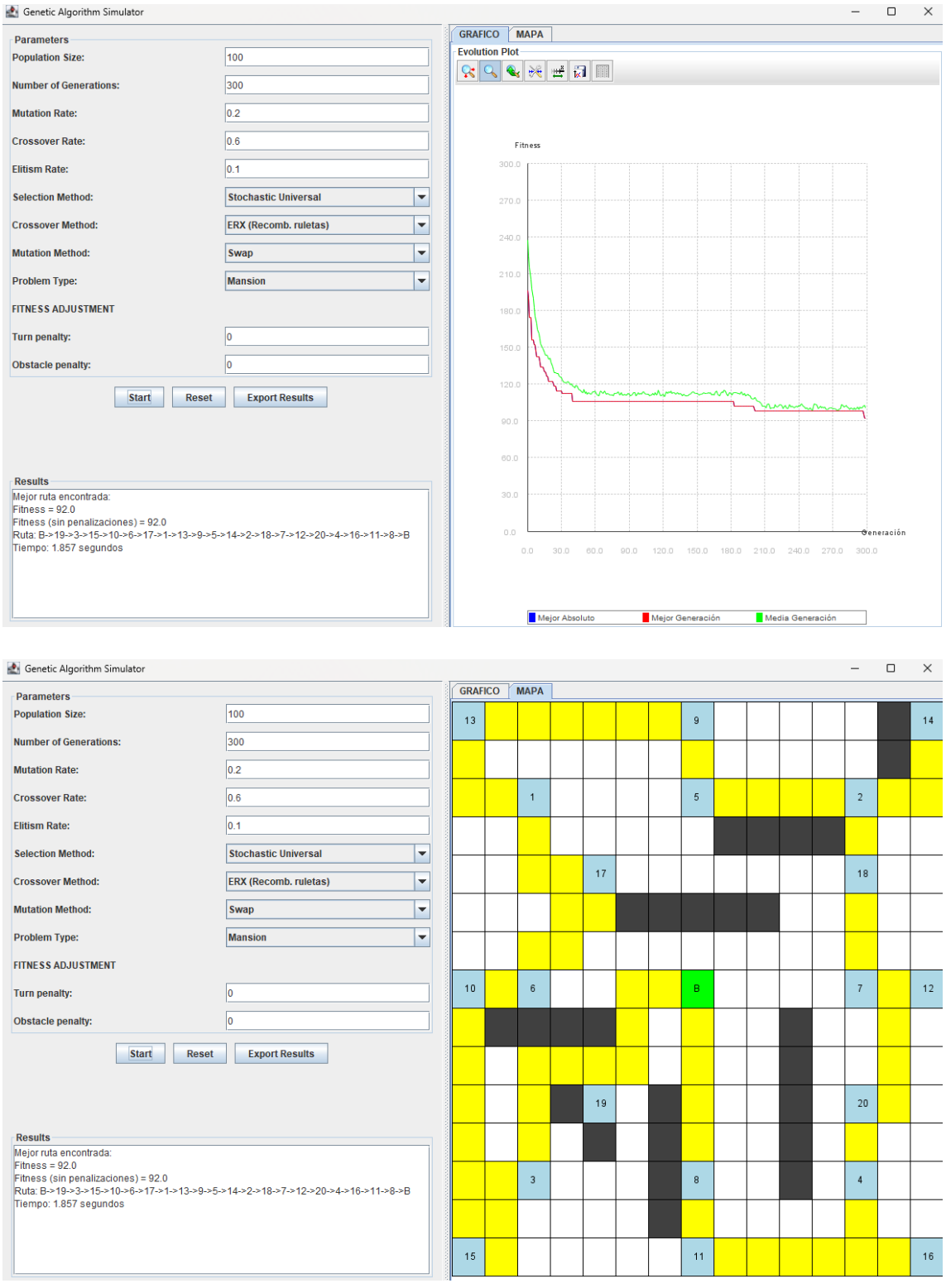


Figura 1.4 Ejecución con elitismo (10%), 300 generaciones, selección estocástica universal, cruce ERX y mutación por intercambio.

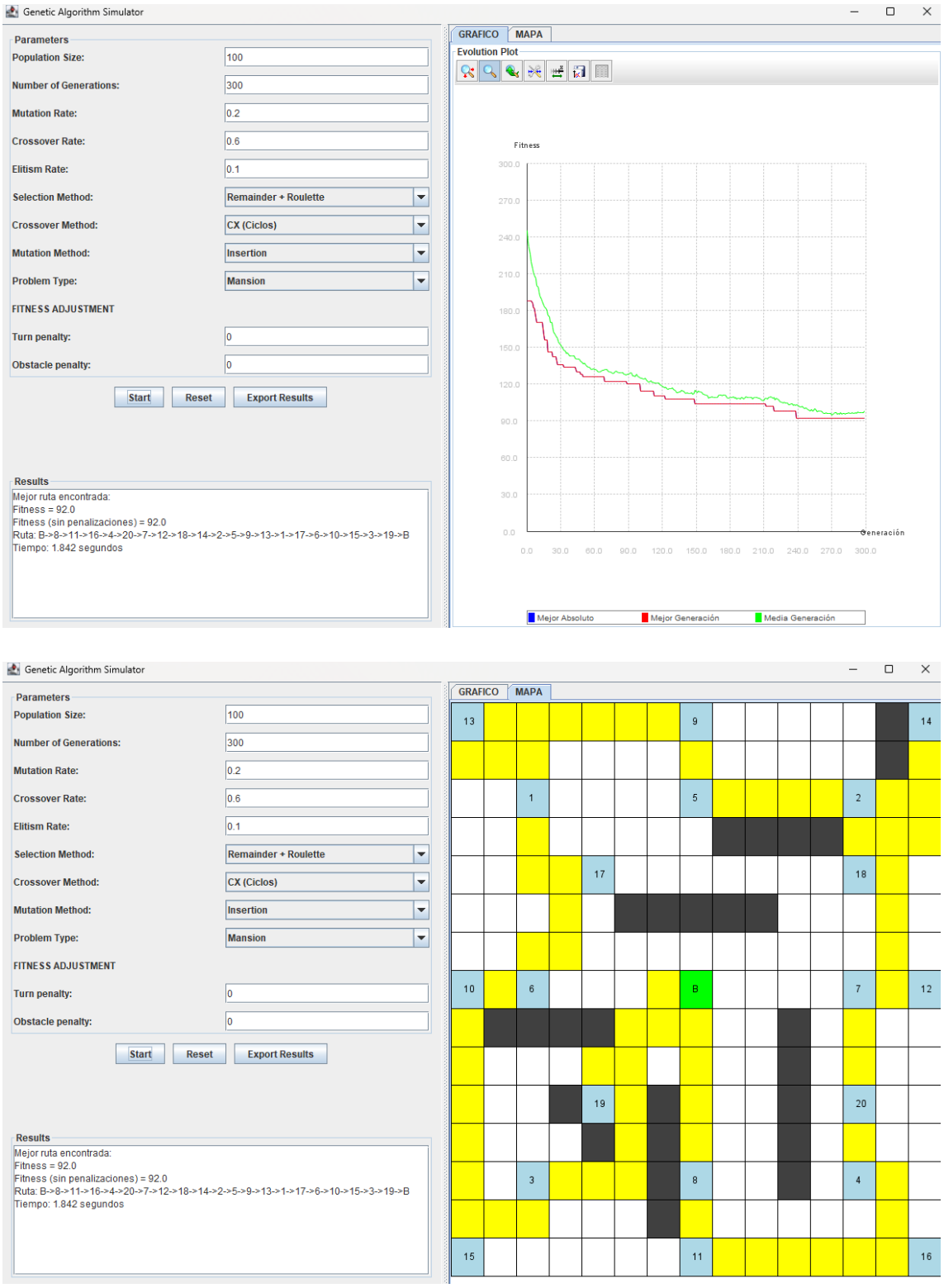
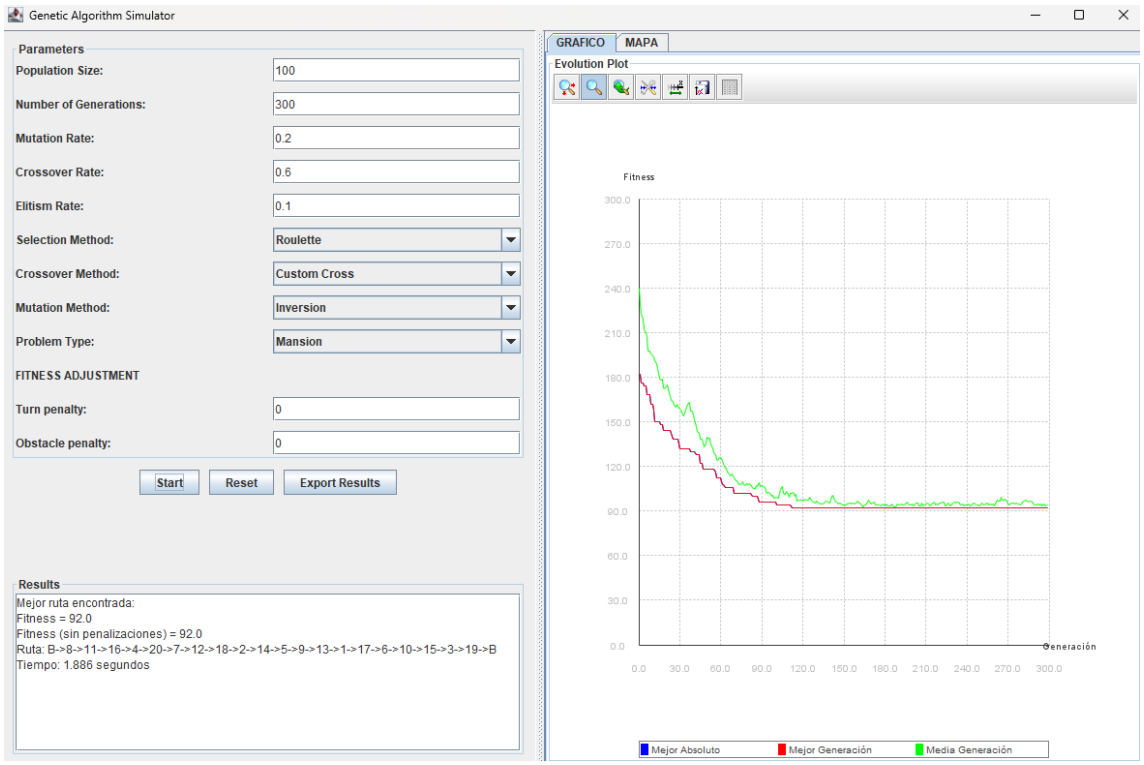


Figura 1.5 Ejecución con elitismo (10%), 300 generaciones, selección por restos y ruleta, cruce CX y mutación por inserción.

EJEMPLOS CON MÉTODOS PROPIOS



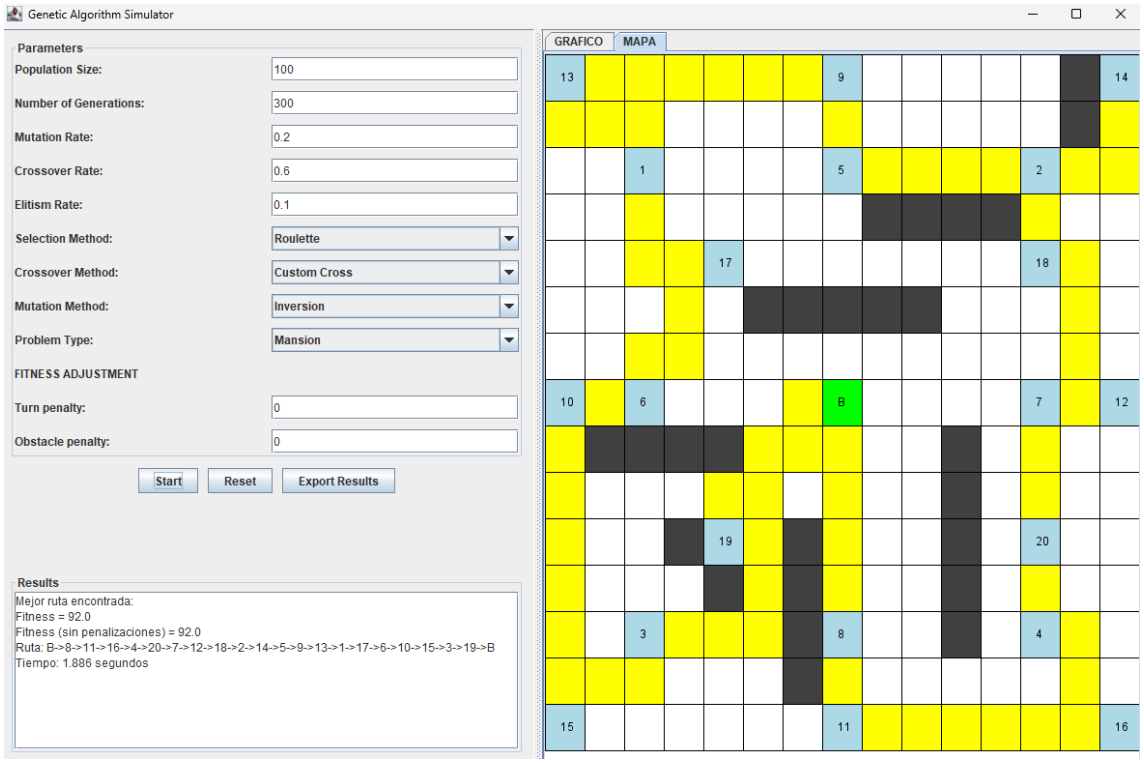


Figura 1.6 Ejecución con elitismo (10%), 300 generaciones, selección por ruleta, cruce personalizado y mutación por inversión.

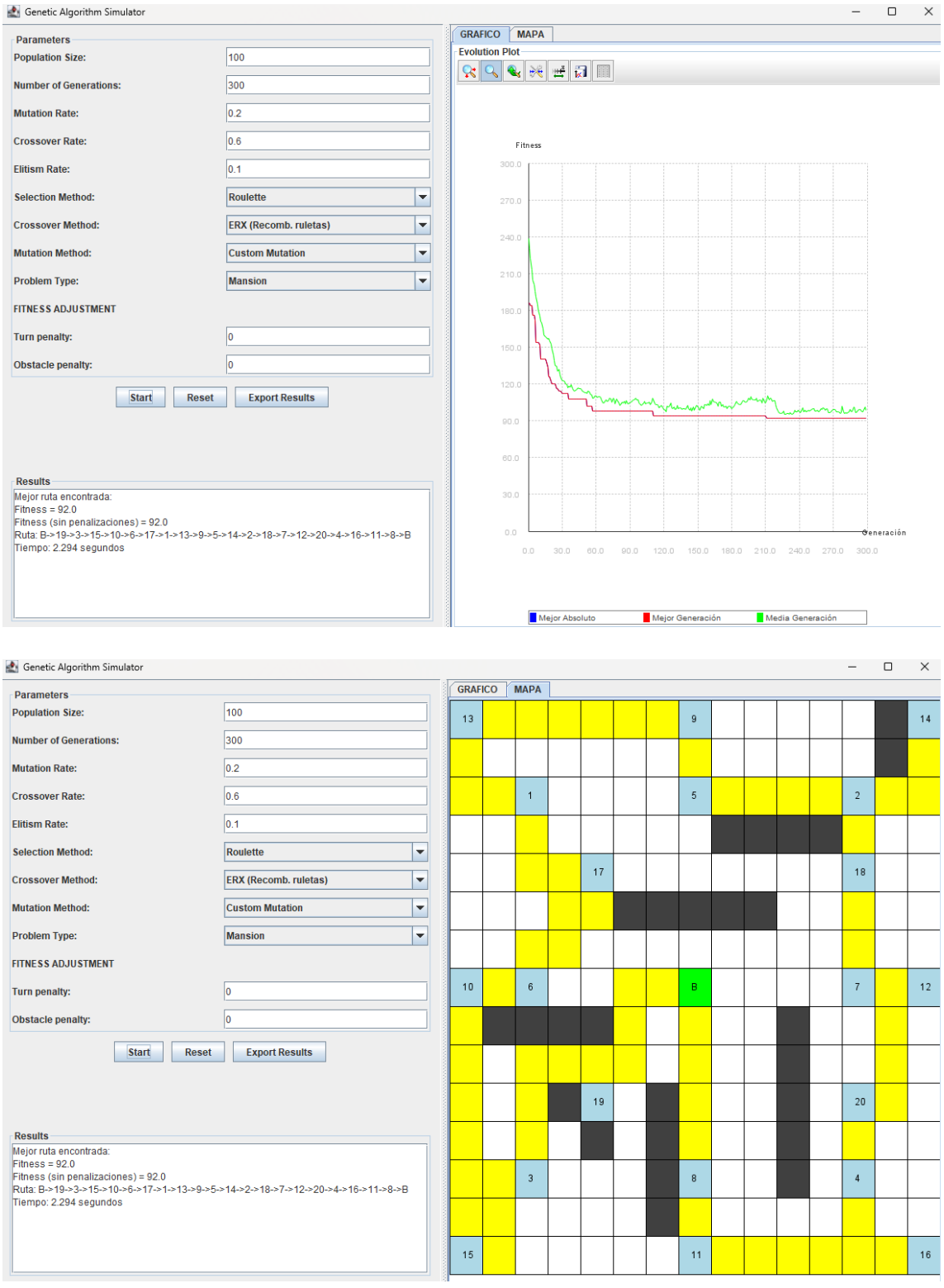


Figura 1.7 Ejecución con elitismo (10%), 300 generaciones, selección por ruleta, cruce ERX y mutación personalizada.

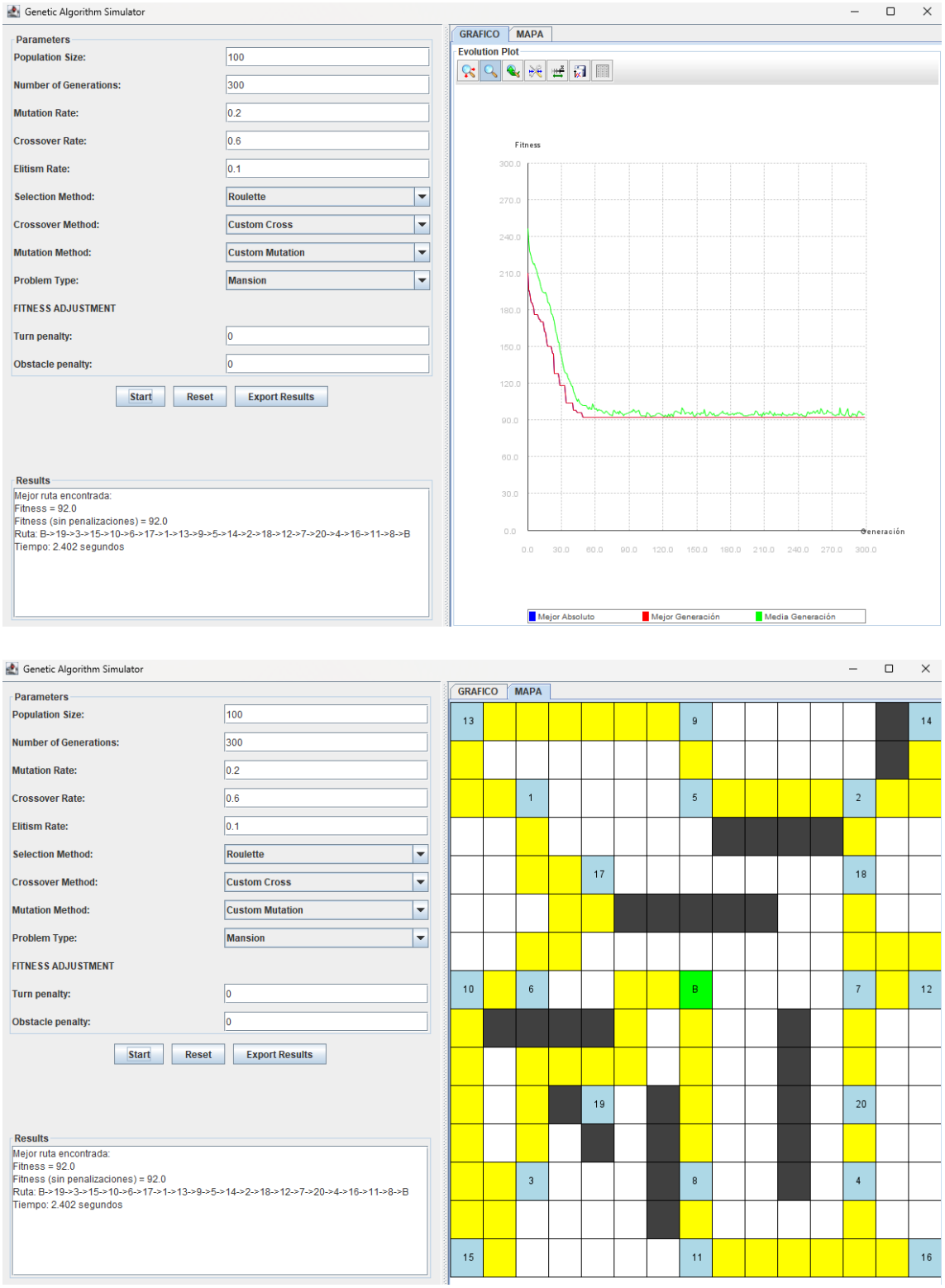
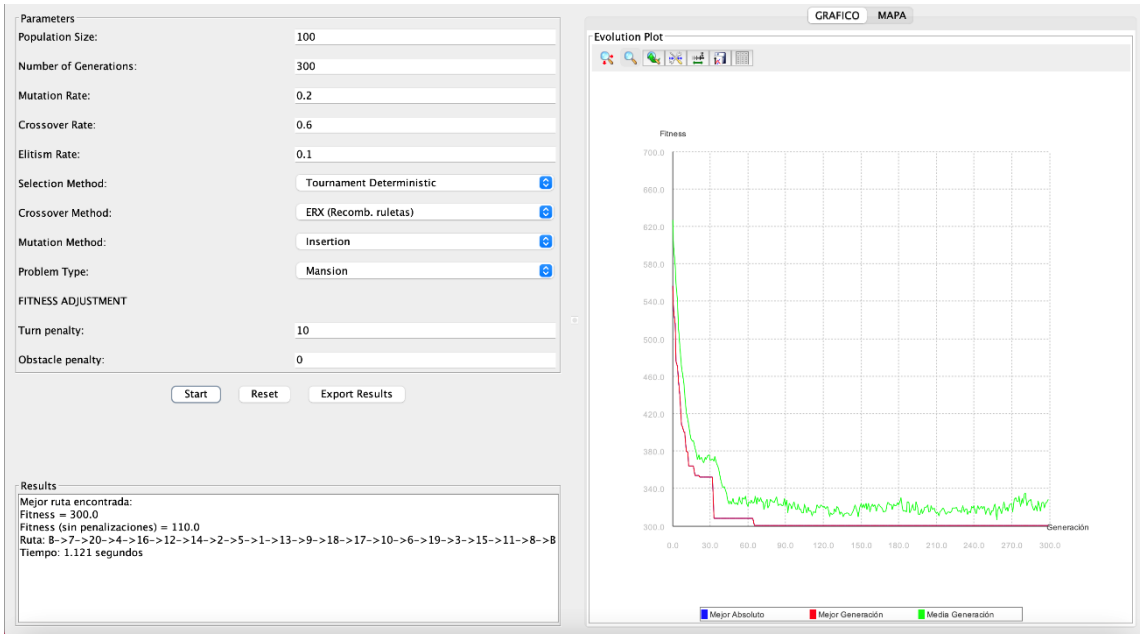


Figura 1.8 Ejecución con elitismo (10%), 300 generaciones, selección por ruleta, cruce y mutación personalizados.

EJEMPLOS CON PENALIZACIÓN



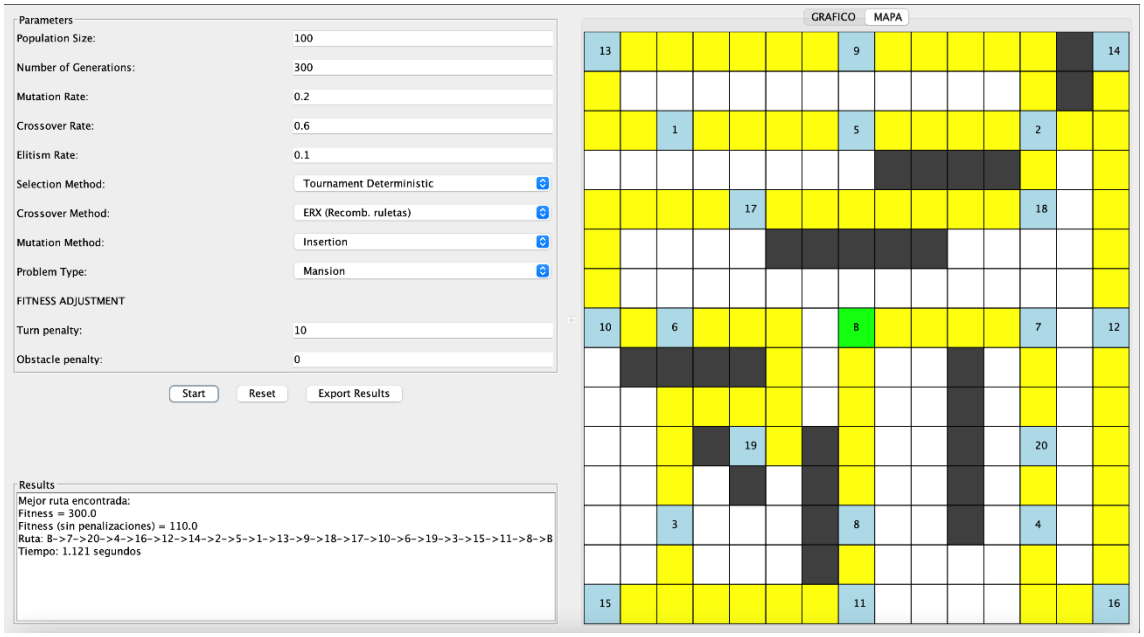


Figura 1.9 Ejecución con elitismo (10%), 300 generaciones, selección por torneo determinista, cruce ERX y mutación por inserción con una penalización de coeficiente 10 por giros.

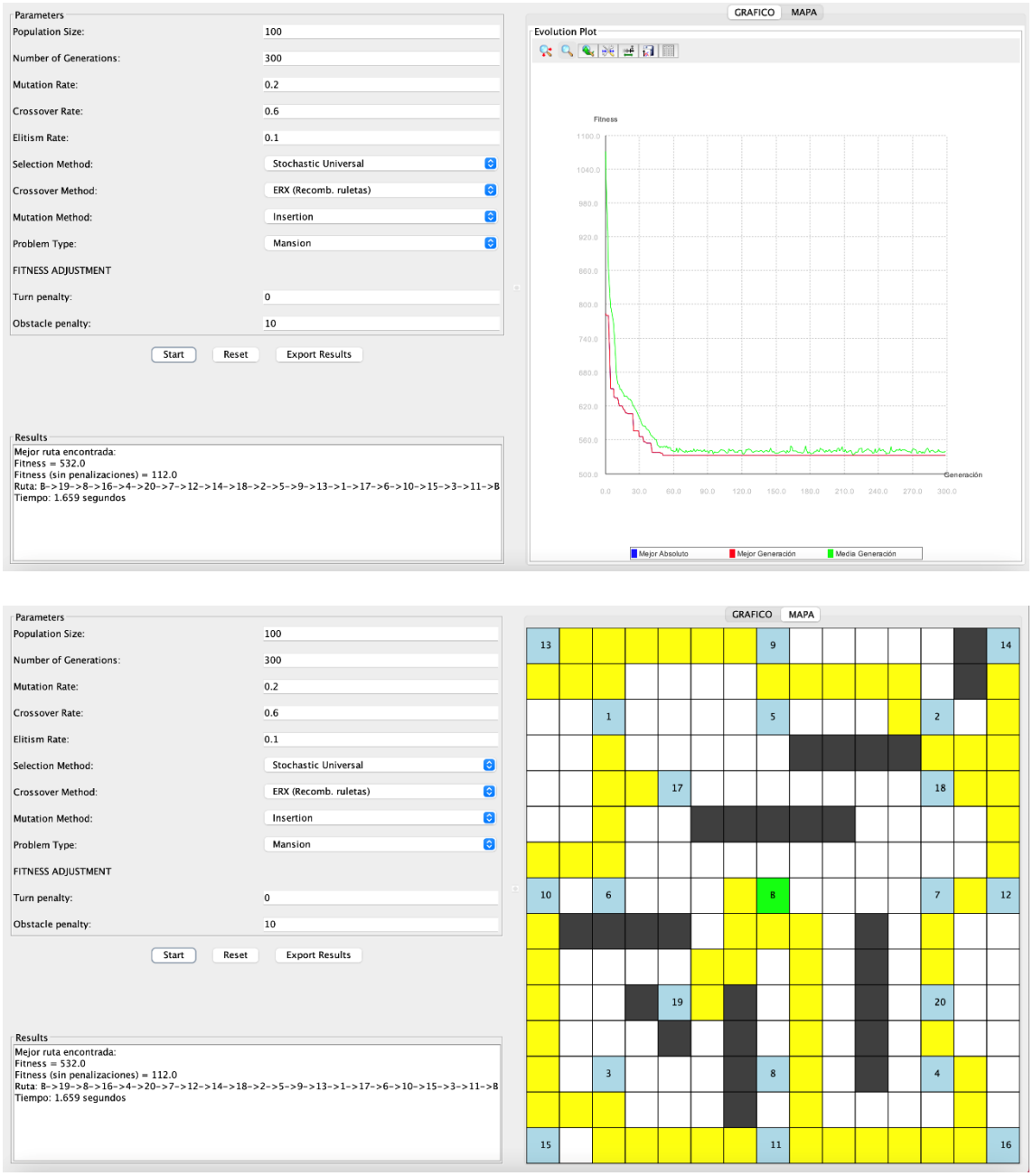


Figura 1.10 Ejecución con elitismo (10%), 300 generaciones, selección estocástica universal, cruce ERX y mutación por inserción con una penalización de coeficiente 10 por cada obstáculo cerca del camino.

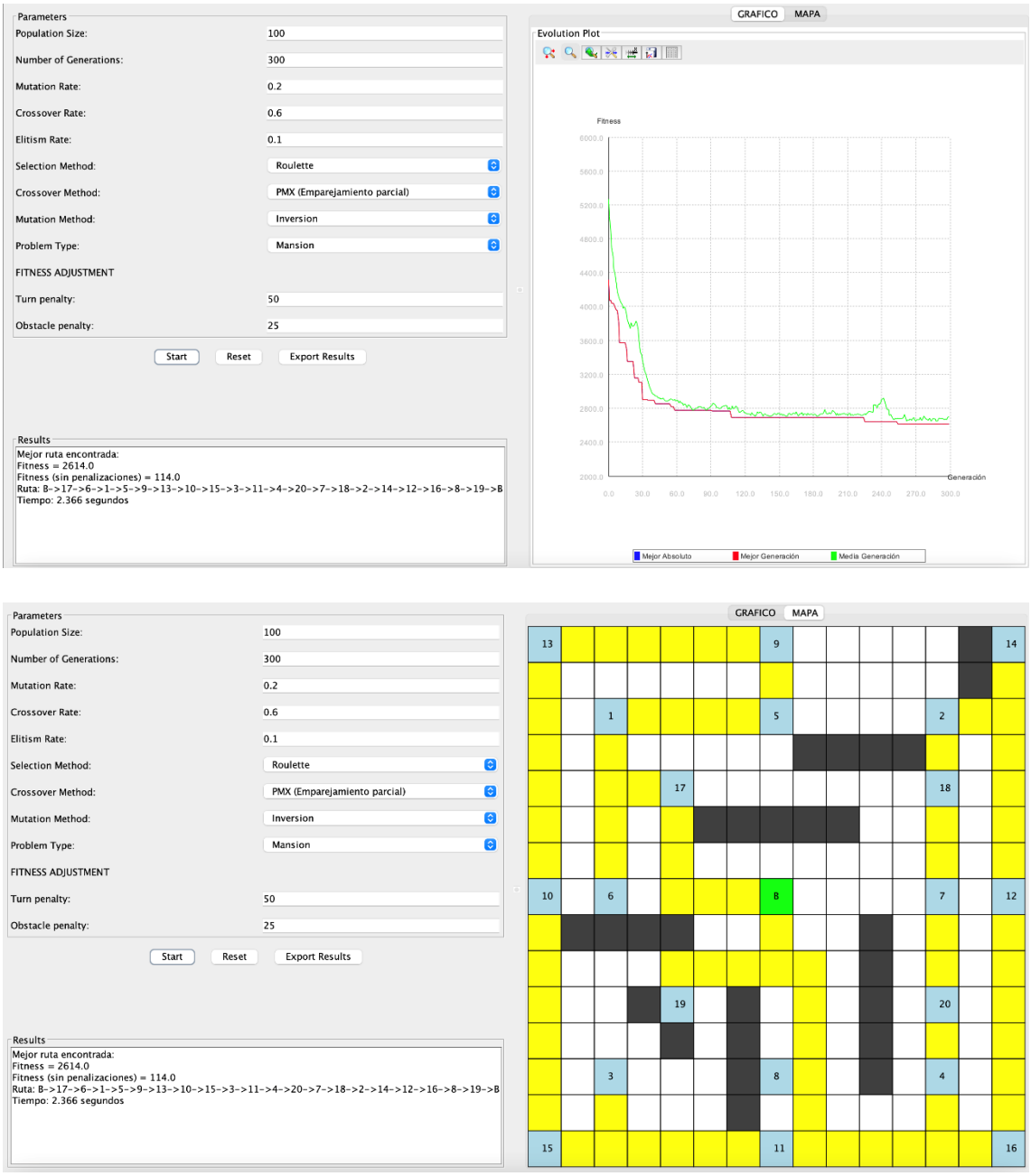


Figura 1.11 Ejecución con elitismo (10%), 300 generaciones, selección por ruleta, cruce ERX y mutación por inversión con una penalización de coeficiente 25 por cada obstáculo cerca del camino y 50 por cada giro.

2. Observaciones de la práctica

Para este problema hemos tenido que aumentar el número de generaciones a 300 y la probabilidad de mutación a 20%. Hemos dejado el elitismo en el 10%, si aumentar demasiado, como hemos hecho en la anterior práctica.

Analizando las capturas, vemos que alcanzan el óptimo del fitness en 92. La mayoría de las ejecuciones converge en la generación 140-190, excepto algunas como en la Figura 1.4, que converge casi en la generación 300.

Otra cosa curiosa, es en ejecuciones sin penalización, el segundo nodo (después de la base) o el penúltimo nodo pueden ser o bien 19 y 8 o bien 8 y 19. Esto es porque el recorrido es cíclico y nos da igual si empezar por un lado o por el otro. Los caminos en general son muy parecidos, pero diferentes.

Respecto las ejecuciones con penalización, en la penalización por giros (figura 1.9) vemos como la ruta tiene menos giros, aunque no conseguimos llegar al fitness ideal (92). En la penalización por obstáculos (figura 1.10) prefiere no caminar cerca de los obstáculos (por ejemplo, ir de la habitación 19 a la 3). Por último, en la optimización multiobjetivo (figura 1.11), vemos que efectivamente combina ambos objetivos.

La dificultad que hemos tenido era mejorar el rendimiento del algoritmo, ya que el A* es muy costoso. Para extinguir este problema hemos tirado de usar caches para no calcular el fitness de la misma permutación 2 veces. Como vemos en las capturas, los tiempos de ejecución sin penalización cambian un poco respecto de los tiempos de ejecución con penalización. Aun así, hemos conseguido bastante buen rendimiento, teniendo en cuenta que se optimizan correctamente los problemas.

Por último, los métodos propios que hemos implementado parecen conseguir buen rendimiento también.

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.

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 MansionMapPanel: Representa un panel gráfico que dibuja el mapa de la mansión, incluyendo habitaciones, obstáculos y rutas.

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 IndividuoAspiradoraFactory: Implementa fábrica específica para el tipo de individuo (función a optimizar).

3. Paquete es.ucm.individuos

Este paquete contiene las clases que representan a los individuos de la población. (en este caso, el problema tiene asignado el IndividuoAspiradora)

Clase Individuo: Representa a un individuo de la población, recibe el mapa como parámetro.

Tiene una caché para el fitness para mejorar el rendimiento.

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.

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.

Clases COCross, CXCross, CustomCross etc.: Implementan métodos de cruce específicos para el problema del mapa (encontrar la mejor permutación).

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 HeuristicMutate, InsertionMutate, etc.: Implementan métodos de mutación específicos para el problema del mapa (encontrar la mejor permutación).

8. Paquete es.ucm.mansion

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

Clase AbstractMansionMap: Representa un mapa abstracto de la mansión, con habitaciones, obstáculos y métodos para calcular rutas y fitness (con o sin penalizaciones).

Clase MansionMap: Representación del mapa de la mansión del problema propuesto, definiendo la ubicación de habitaciones y obstáculos.

Clase MiniMansionMap: Otro mapa más pequeño para probar de forma más rápida el funcionamiento del algoritmo.

9. Paquete es.ucm.mansion.busqueda

Este paquete contiene las clases que se utilizan para encontrar la mejor ruta entre 2 puntos (incluyendo penalizaciones y evitando obstáculos).

Clase AEstrellaBusquedaCamino: Implementa el algoritmo A* para encontrar la ruta de menor coste entre dos puntos en el mapa. (incluyendo las penalizaciones en ese coste). Esta clase es la que más tiempo y recursos consume.

Enumeración MovementEnum: Define los movimientos posibles en el mapa (arriba, abajo, izquierda, derecha).

Clase NodoCamino: Representa un nodo en la búsqueda de rutas con el algoritmo A*. Se utiliza para sacar la ruta de forma recursiva al encontrar la solución, ir llamando al nodo anterior.

10. Paquete es.ucm.mansion.objects

Este paquete contiene las clases que representan los objetos del mapa (habitaciones u obstáculos).

Clase AbstractMansionObject: Clase base abstracta para representar objetos en la mansión (habitaciones u obstáculos). Sirve principalmente para detectar si es hay un obstáculo en alguna celda.

Clase Obstacle: Representa un obstáculo en la mansión.

Clase Room: Representa una habitación en la mansión.

11. Paquete es.ucm.utils

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

4. Métodos propios

Clase ScrambleMutate: La clase *ScrambleMutate* es un operador de mutación diseñado para alterar la estructura de un individuo en un algoritmo genético.

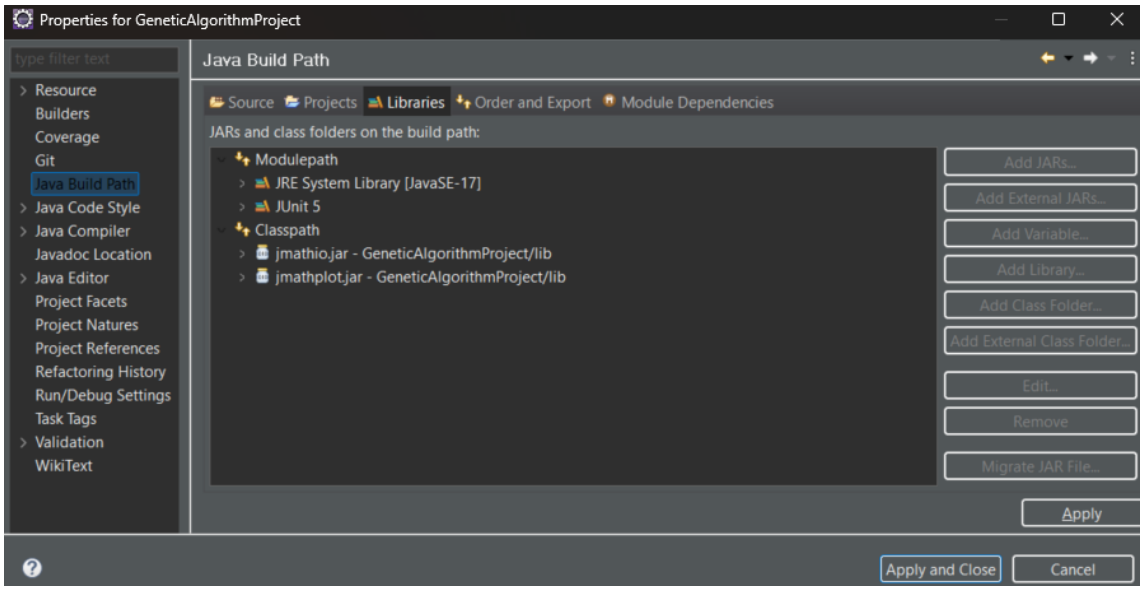
El proceso comienza seleccionando aleatoriamente un segmento de la permutación. Para ello, se eligen dos índices aleatorios que definen el inicio y el fin del segmento. Una vez delimitado este segmento, los elementos que lo componen se mezclan de manera aleatoria, como si se barajara un mazo de cartas. Esta mezcla altera el orden original, creando una nueva variante de la permutación. Finalmente, la permutación modificada se asigna de nuevo a los genes del individuo, generando así una nueva versión que podrá ser evaluada y seleccionada en futuras generaciones.

Clase CustomCross: La clase *CustomCross* implementa un operador de cruce personalizado.

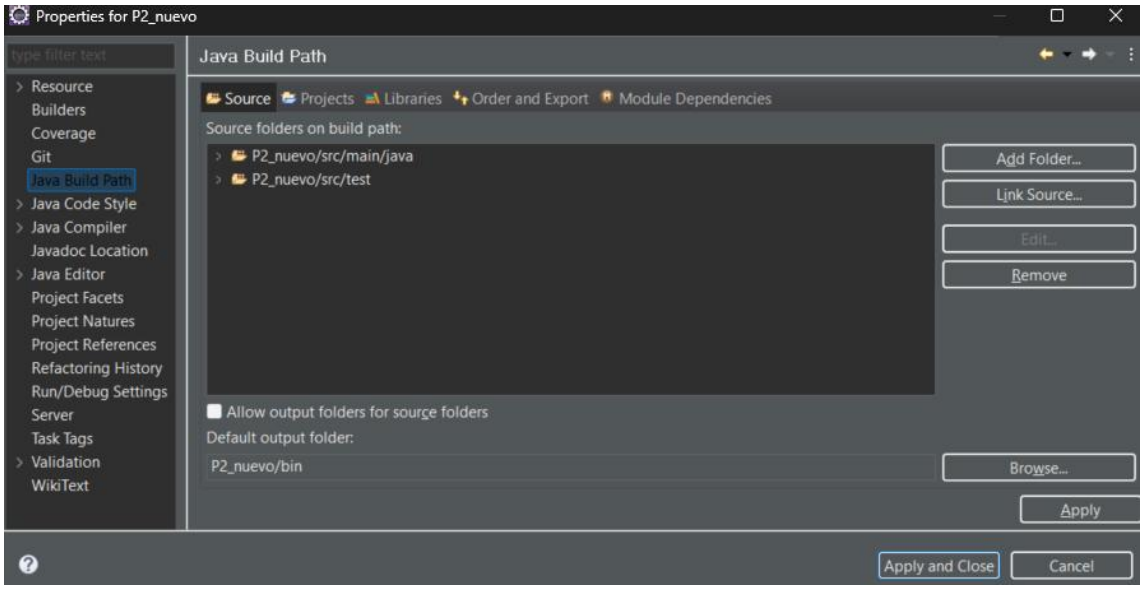
El proceso comienza seleccionando un conjunto de posiciones aleatorias en la permutación del primer padre. Estas posiciones definen los elementos que serán intercambiados entre los padres. A continuación, se identifican los valores correspondientes a estas posiciones en el segundo padre, así como sus ubicaciones. Una vez identificados los valores y sus posiciones, se procede a intercambiarlos entre los padres. Los valores seleccionados del primer padre se reordenan según el orden en que aparecen en el segundo padre, y viceversa. Finalmente, se crean dos nuevos individuos (hijos) a partir de las permutaciones modificadas.

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:

Se ha encargado de ajustar la estructura de la práctica 1 para que funcione con individuos enteros, creando las clases para la mansión, gen de enteros. Ha diseñado e implementado el cruce propio (CustomCross). Por otra parte, ha implementado el algoritmo de A* y la penalización por obstáculos.

Mario:

Ha mejorado la interfaz para poder visualizar la mansión. A partir de la estructura adaptada, ha añadido los nuevos métodos de cruce y mutaciones. Ha diseñado e implementado la mutación propia (ScrambleMutate). Por otra parte, ha implementado la penalización por turnos.

7. Conclusiones

En conclusión, con esta práctica se ha demostrado que los algoritmos genéticos se pueden extender en áreas de logística, ofreciendo una herramienta poderosa para resolver problemas complejos de optimización, como la planificación de rutas en entornos con múltiples restricciones. Aunque no ha sido sencillo optimizar el rendimiento y encontrar las configuraciones apropiadas, el resultado ha valido la pena, ya que se han obtenido soluciones eficientes y cercanas al óptimo en un tiempo razonable.