UAX

UNIVERSIDAD ALFONSO X EL SABIO

FEEDBACK UNIDADES 1 Y 2

Inteligencia Artificial

Álvaro Valera López.
avarelop@myuax.com

Índice general

Índice de figuras

CAPÍTULO 1

Ejercicio 1.

1.1. Marco Teórico.

Antes de profundizar en el ejercicio se introducirán los conceptos teóricos necesarios para resolver el problema.

1.1.1. Algoritmo A*.

Definición $\longrightarrow$ Algoritmo de búsqueda de caminos que utiliza una combinación de búsqueda de costo uniforme y heurística para guiar la exploración de manera más eficiente hacia la solución óptima.

Su objetivo principal es encontrar el camino más corto entre un nodo inicial y un nodo objetivo en un grafo o espacio de búsqueda.

En qué consiste.

El algoritmo A* utiliza listas de nodos abiertos y cerrados. Comienza con el nodo inicial y selecciona el nodo de menor valor en cada iteración.

Se expande el nodo seleccionado, se calcula y actualiza el costo g(n) para los nodos vecinos. Si un nodo vecino no está en la lista de nodos abiertos, se agrega a ella y se registra como el nodo padre. Si ya está en la lista de nodos abiertos, se actualiza el costo g(n) si es menor.

El algoritmo continúa expandiendo nodos y actualizando los costos hasta alcanzar el objetivo o vaciar la lista de nodos abiertos. En ese punto, si se encontró un camino al objetivo, se puede reconstruir el camino óptimo siguiendo los nodos padres desde el objetivo hasta el nodo inicial.

Función de Evaluación.

Función empleada para determinar la prioridad de los nodos.
Se utiliza para seleccionar el siguiente nodo a explorar.

El algoritmo A* evalúa cada nodo en base a la combinación del costo acumulado desde el nodo inicial hasta el nodo actual y una estimación heurística del costo restante hasta el nodo objetivo.

La **función de evaluación** es de la forma:
$$\mathbf{f(n)} = g(n) + h(n)$$

El valor $f(n)$ determina la prioridad de un nodo en la lista de nodos abiertos. Cuanto menor sea el valor de $f(n)$, mayor será la prioridad del nodo para ser seleccionado.

Costo Acumulado *g(n)*.

Representa la suma total de los costos asociados con cada arista o transición recorrida desde el nodo inicial hasta el nodo actual n a lo largo del camino que se ha explorado hasta el momento.

Cada vez que se expande un nuevo nodo en el algoritmo A*, se calcula su costo acumulado actualizando el valor de $g(n)$.

El costo acumulado $g(n)$ en el algoritmo A* se actualiza al explorar los nodos adyacentes. Si se encuentra un camino más corto desde el nodo inicial hasta un nodo vecino, se actualiza el valor de $g(n)$ para ese nodo.

A* busca el camino más óptimo y eficiente hacia el objetivo, actualizando el costo acumulado mientras avanza a través de los nodos y elige las aristas que minimizan el costo total.

Heurística *h(n)*.

Valor de la heurística del estado n, es decir, una estimación del coste de llegar desde el estado n al estado solución.

La heurística debe ser admisible para que el algoritmo A* funcione correctamente.

Una heurística admisible es aquella que nunca sobreestima el costo real para llegar al objetivo. En otras palabras, la estimación proporcionada por la heurística debe ser igual o menor que el costo real.

Si una heurística sobreestima el costo, puede conducir a una solución subóptima o incluso a una solución incorrecta.

1.2. Caracterización del problema mediante estados.

1.2.1. Definición de estado.

1.2.2. Definición de operadores.

El robot, solo podrá efectuar desplazamientos verticales y horizontales que le mantengan dentro del espacio, pero nunca diagonales.

En este ejercicio ninguna casilla dispone de todos los movimientos, pues existen obstáculos que impiden el desplazamiento.

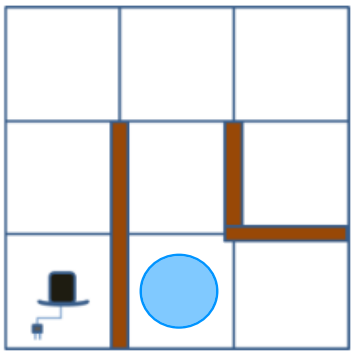

Figura 1.1: Estado Inicial

Este es el estado inicial. En esta posición el robot únicamente podrá efectuar un desplazamiento hacia arriba o hacia la derecha. Puesto que el desplazamiento hacia abajo le sacaría del espacio disponible y el desplazamiento hacia la izquierda le haría chocarse contra la pared.

Deberá existir una función 'movimientoValido' que indique al robot que desplazamientos puede realizar en cada casilla.

1.3. Elementos de la búsqueda A*.

1.3.1. Función de Coste.

Para todos los hijos del estado inicial el valor de $g(n)$ es de 1, ya que mide el coste real de llegar a esta solución, que para este problema es el numero casillas desplazadas por el robot, y solo se habrá realizado un movimiento.

1.3.2. Función Heurística.

Tal y como se ha mencionado en la sección 1.2.2 los operadores disponibles para nuestro robot de limpieza son movimientos verticales y horizontales, pero nunca diagonales.

Cuando se trabaja con este tipo de movimientos, existe un concepto llamado *Distancia Manhattan*, que servirá como función Heurística para nuestro algoritmo A*.

Distancia Manhattan.

Definición $\longrightarrow$ La distancia de Manhattan entre dos puntos en un espacio euclidiano (como un plano o una cuadrícula) mide la distancia horizontal y vertical para alcanzar el punto de destino, sin tomar en cuenta los caminos diagonales.

Se calcula sumando las diferencias absolutas entre las coordenadas de los dos puntos en cada dimensión. Expresado de manera formal, si se tienen dos puntos A y B con coordenadas (x1, y1) y (x2, y2) respectivamente, la distancia de Manhattan (M) se calcula:

$$\mathbf{M} = |x2 - x1| + |y2 - y1|$$

Demostración para la distancia Manhattan.

Si recordamos, en el apartado 1.1.1 se definió a una función heurística como admisible si subestimaba el costo real para llegar al objetivo.

En un espacio de búsqueda donde solo se permiten movimientos verticales y horizontales, se puede demostrar de manera matemática que la distancia de Manhattan siempre subestima el costo real para llegar al objetivo. Esto se puede hacer utilizando la desigualdad triangular.

La desigualdad triangular establece que, para cualquier tres puntos A, B y C en un espacio, la distancia directa de A a C siempre será menor o igual a la suma de las distancias directas de A a B y de B a C.

Considere tres puntos en el espacio de búsqueda:

1. El nodo inicial N.
2. Un nodo intermedio M.
3. El nodo objetivo O.

Supongamos que la distancia de Manhattan desde N hasta O es $D(N, O)$, y la distancia de Manhattan desde N hasta M es $D(N, M)$, mientras que la distancia de Manhattan desde M hasta O es $D(M, O)$.

Aplicando la desigualdad triangular, tenemos:

$D(N, O) \leq D(N, M) + D(M, O)$

Dado que solo se permiten movimientos verticales y horizontales, $D(N, M)$ y $D(M, O)$ son las diferencias absolutas entre las coordenadas de los puntos en cada dimensión.

La distancia de Manhattan $D(N, O)$ se calcula como la suma de las diferencias absolutas entre las coordenadas de los puntos en cada dimensión.

Entonces, podemos reescribir la desigualdad triangular de la siguiente manera:

$$D(N,O) \leq |x_N - x_M| + |y_N - y_M| + |x_M - x_O| + |y_M - y_O|$$

Dado que los movimientos son verticales y horizontales, los términos $|x_N - x_M|$ y $|y_N - y_M|$ corresponden a las distancias directas entre los puntos N y M en cada dimensión.

Por lo tanto, podemos simplificar la desigualdad triangular a:

$$D(N,O) \leq D(N,M) + D(M,O)$$

Esto demuestra que la distancia de Manhattan $D(N,O)$ es menor o igual a la suma de las distancias de Manhattan $D(N,M)$ y $D(M,O)$.

La distancia de Manhattan siempre subestima el costo real para llegar al objetivo en un espacio de búsqueda donde solo se permiten movimientos verticales y horizontales.

1.3.3. Función de Evaluación.

La función de evaluación es el resultado de sumar la función heurística con la de coste:

$$f(n) = h(n) + g(n)$$

1.4. Evolución del Proceso de Búsqueda.

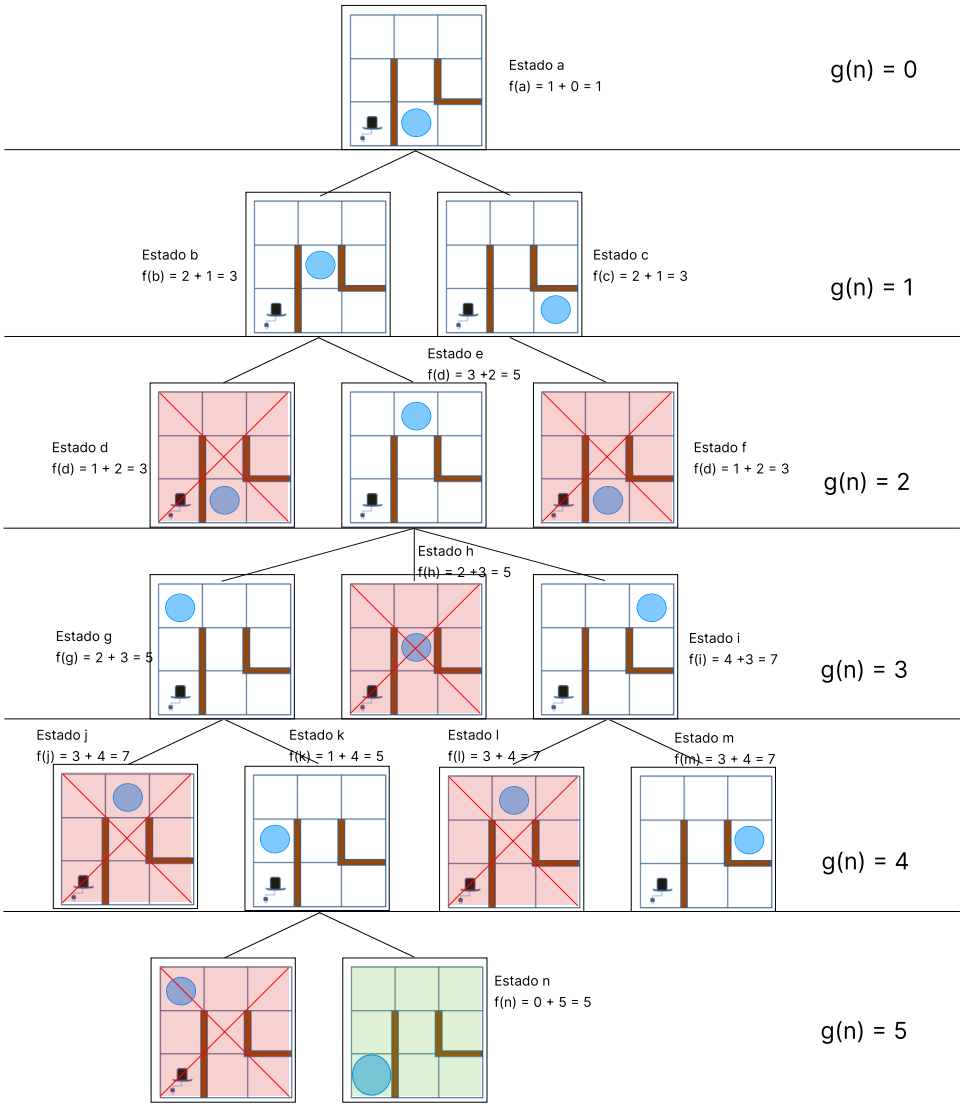

Figura 1.2: Diagrama de Estados de la búsqueda.

CAPÍTULO 2

Ejercicio 2.

Programa capaz de encontrar mediante un algoritmo genético un código numérico prefijado.

2.1. Introducción.

La resolución de este segundo ejercicio se dividirá en 3 secciones claramente diferenciadas:

1. **Diseño del Programa.** $\longrightarrow$ En este apartado se analizarán los requerimientos del enunciado y se explicará el código entregado. La sección esta compuesta por los siguientes sub-apartados:
   a) Diagrama UML de Clase.
   b) Diagrama UML de Secuencia.
   c) Explicación del código entregado.
2. **Diseño de la Experimentación.** $\longrightarrow$ En esta sección se expondrán los diferentes casos de prueba que se han llevado a cabo para explicar con detalle el funcionamiento del algoritmo genético. El apartado esta compuesto por:
   a) Definición de los casos de prueba.
   b) Ejecución de los casos de prueba.
3. **Conclusiones.**

2.2. Diseño del Programa.

2.2.1. Diagrama UML de clase.

Tras analizar el enunciado del problema se ha decidido que la estructura del programa que mejor se adapta al ejercicio es la siguiente:

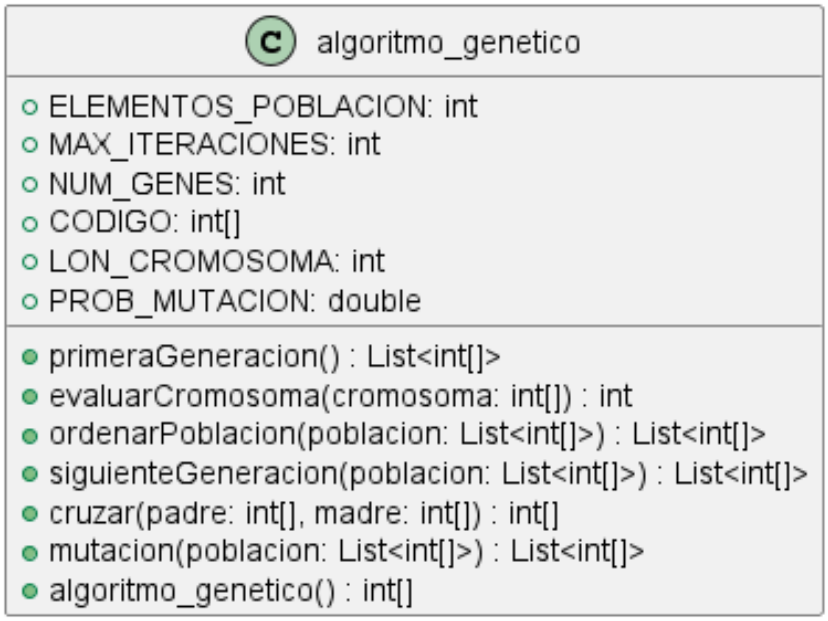

Figura 2.1: Diagrama de Clases para el Algoritmo Genético.

En el diagrama se diferenciarán las constantes que forman parte del *cuadro de mando* de los métodos que componen al algoritmo genético.

2.2.2. Explicación del código programado.

Librerias importadas.

```
import random
import numpy as np
```

Figura 2.2: Librerías importadas en el programa.

La biblioteca NumPy se utiliza en este problema por varias razones:

1. **Cálculo vectorizado**: NumPy es fundamentalmente útil para realizar cálculos numéricos de manera eficiente. Ofrece soporte para operaciones vectorizadas, lo que significa que puede realizar operaciones en matrices enteras de manera simultánea, en lugar de tener que utilizar bucles para iterar sobre cada elemento individualmente. En este problema, por ejemplo, se utiliza esta característica para generar los cromosomas de la población inicial y para calcular el fitness de cada cromosoma.

2. **Creación de arrays**: NumPy proporciona funciones para generar arrays de números aleatorios de manera muy sencilla, como numpy.random.randint(), que se usa en este problema para generar los genes de cada cromosoma.

3. **Manejo de arrays**: Las operaciones de arrays, como la concatenación y la indexación, son muy sencillas y eficientes con NumPy. En este problema, por ejemplo, se utiliza la función numpy.concatenate() para realizar el cruce entre dos cromosomas.

Parámetros de ejecución.

Con el objetivo de poder modificar los parámetros del algoritmo de manera sencilla se han creado las siguientes constantes globales que permitirán llevar a cabo experimentos ágilmente.

```
# Parámetros del algoritmo
ELEMENTOS_POBLACION = 100  # Tamaño de la población
MAX_ITERACIONES = 1000  # Número máximo de iteraciones
NUM_GENES = 30
CODIGO = np.random.randint(1, 100, NUM_GENES)
LON_CROMOSOMA = len(CODIGO)
PROB_MUTACION = 0.1  # Probabilidad de mutación
```

Figura 2.3: Parámetros de control del algoritmo.

Función primeraGeneracion().

Esta función crea la población inicial para el algoritmo genético. Los cromosomas se generan aleatoriamente, y su longitud es igual a la del código que se está buscando.
El método no recibe ningún parámetro y devuelve una lista de cromosomas, cada uno de los cuales es un array de NumPy que contiene números aleatorios entre 1 y 100.

```
def primeraGeneracion():
    """
    Función para generar la población inicial de cromosomas. Cada cromosoma es un vector de enteros
    generados aleatoriamente.

    :return: lista de cromosomas (numpy arrays)
    """
    return [np.random.randint(1, 100, LON_CROMOSOMA) for _ in range(ELEMENTOS_POBLACION)]
```

Figura 2.4: Fragmento de código de la función primeraGeneración.

Función cruzar(padre, madre).

Este método toma dos cromosomas (el 'padre' y la 'madre') y produce un nuevo cromosoma que es una combinación de ambos. El punto de cruce (el lugar donde se dividen los dos cromosomas para combinarlos) se elige al azar.
El método recibe dos cromosomas y devuelve un nuevo cromosoma.

```python
def cruzar(padre, madre):
    """
    Función para cruzar dos cromosomas. La descendencia se genera tomando la primera parte del padre
    y la segunda parte de la madre.

    :param padre: Un array de numpy que representa un cromosoma.
    :param madre: Un array de numpy que representa un cromosoma.
    :return: Un nuevo cromosoma generado a partir de los cromosomas del padre y la madre.
    """
    punto_cruce = random.randint(0, LON_CROMOSOMA)
    hijo = np.concatenate((padre[:punto_cruce], madre[punto_cruce:]))
    return hijo
```

Figura 2.5: Fragmento de código de la función cruzar.

Función ordenarPoblacion(poblacion).

Esta función ordena la población de cromosomas en función de su fitness, de menor a mayor. Esto significa que los cromosomas que están más cerca del código objetivo estarán al principio de la lista.
El método recibe la población (una lista de cromosomas) y devuelve la misma lista, pero ordenada por fitness.

```python
def ordenarPoblacion(poblacion):
    """
    Función para ordenar la población de cromosomas en orden ascendente de aptitud.

    :param poblacion: Lista de cromosomas.
    :return: La población ordenada.
    """
    poblacion.sort(key=evaluarCromosoma)
    return poblacion
```

Figura 2.6: Fragmento de código de la función ordenarPoblacion.

Función evaluarCromosoma(cromosoma).

Esta función calcula el fitness de un cromosoma. El fitness es una medida de cuánto difiere el cromosoma del código objetivo. En este caso, el fitness se calcula como la cantidad de genes (números) en el cromosoma que no coinciden con los correspondientes en el código.
El método recibe un cromosoma (un array de NumPy de números) y devuelve un número entero que representa el fitness del cromosoma.

```
def evaluarCromosoma(cromosoma):
    """
    Función para evaluar la aptitud de un cromosoma. La aptitud se calcula como la suma de las diferencias
    entre los genes del cromosoma y los del código objetivo.

    :param cromosoma: Un array de numpy que representa un cromosoma.
    :return: un entero que representa la aptitud del cromosoma.
    """
    return np.sum(cromosoma != CODIGO)
```

Figura 2.7: Fragmento de código de la función evaluarCromosoma.

Función comprobarMejorIndividuo(mejor_individuo).

Esta función comprueba si el mejor individuo del que se dispone en este momento coincide con el estado objetivo.
El método recibe un cromosoma (un array de NumPy de números) y devuelve un número un valor booleano que indica si el fitness del cromosoma es 0 con respecto al del mejor individuo.

```
def comprobarMejorIndividuo(mejor_individuo):
    return evaluarCromosoma(mejor_individuo) == 0
```

Figura 2.8: Fragmento de código de la función comprobarMejorIndividuo.

Función **mutacion(poblacion).**

Este método aplica una mutación aleatoria a los cromosomas de la población. Para cada cromosoma, hay una probabilidad (definida por PROB_MUTACION) de que uno de sus genes se cambie por un número aleatorio entre 1 y 100.
El método recibe la población (una lista de cromosomas) y devuelve la misma lista, pero con posibles mutaciones.

```python
def mutacion(poblacion):
    """
    Función para aplicar una mutación aleatoria a los cromosomas de la población.

    :param poblacion: Lista de cromosomas.
    :return: La población con cromosomas posiblemente mutados.
    """
    for i in range(len(poblacion)):
        if random.random() < PROB_MUTACION:
            punto_mutacion = random.randint(0, LON_CROMOSOMA - 1)
            poblacion[i][punto_mutacion] = random.randint(1, 100)
    return poblacion
```

Figura 2.9: Fragmento de código de la función mutación.

Función siguienteGeneracion(poblacion).

Este método crea la siguiente generación de cromosomas a partir de la actual. Primero, selecciona la élite (el 10 % superior de la población, en términos de fitness) y los añade directamente a la nueva generación. Luego, selecciona pares de cromosomas de la población (siempre incluyendo al menos uno de la élite) y los cruza para generar nuevos cromosomas, que se añaden a la nueva generación hasta que alcanza el tamaño de la población original.
El método recibe la población actual (una lista de cromosomas) y devuelve la nueva generación (otra lista de cromosomas).

```python
def siguienteGeneracion(poblacion):
    """
    Función para generar la próxima generación de cromosomas a partir de la población actual.

    :param poblacion: Lista de cromosomas que representan la población actual.
    :return: Nueva generación de cromosomas.
    """
    nueva_generacion = []  # Lista para la nueva generación
    elite = poblacion[:ELEMENTOS_POBLACION // 10]  # Selección del 10% de los mejores individuos
    nueva_generacion.extend(elite)  # La élite pasa directamente a la siguiente generación

    # Completamos la población con descendencia de individuos elegidos de la élite y la población
    while len(nueva_generacion) < ELEMENTOS_POBLACION:
        padre = random.choice(elite)
        madre = random.choice(poblacion)
        hijo = cruzar(padre, madre)
        nueva_generacion.append(hijo)

    return nueva_generacion
```

Figura 2.10: Fragmento de código de la función siguienteGeneracion.

Método **algoritmoGenetico().**

Este es el método principal, que controla el flujo del algoritmo genético. Primero, genera una población inicial y la ordena por fitness. Luego, en cada iteración, crea una nueva generación a partir de la actual, aplica posibles mutaciones, y ordena la nueva generación por fitness. Si en cualquier momento el cromosoma con el mejor fitness coincide con el código objetivo, el algoritmo se detiene y devuelve ese cromosoma. Si no se encuentra el código objetivo después del número máximo de iteraciones, el algoritmo se detiene y devuelve None.
Este método no recibe ningún parámetro y devuelve el cromosoma que coincide con el código objetivo, o None si no se encuentra tal cromosoma.

```python
def algoritmo_genetico():
    """
    Función principal que controla el flujo del algoritmo genético.

    :return: El cromosoma que coincide con el código objetivo, o None si no se
     encuentra tal cromosoma después del número máximo de iteraciones.
    """
    poblacion = primeraGeneracion()
    poblacion = ordenarPoblacion(poblacion)
    # Si el mejor individuo es igual al código, se ha encontrado la solución
    if comprobarMejorIndividuo(poblacion[0]): return poblacion[0]

    for _ in range(MAX_ITERACIONES):
        poblacion = siguienteGeneracion(poblacion)
        poblacion = mutacion(poblacion)
        poblacion = ordenarPoblacion(poblacion)

        # Si el mejor individuo es igual al código, se ha encontrado la solución
        if comprobarMejorIndividuo(poblacion[0]): return True, poblacion[0]

    # Si se llega al número máximo de iteraciones y no se ha encontrado la solución,
    #  se informa y se termina
    return False, None
```

Figura 2.11: Fragmento de código del método algoritmoGenetico.

2.3. Diseño de las Pruebas.

2.3.1. Guía de desarrollo para las pruebas.

Introducción

El algoritmo genético es un método de optimización inspirado en la teoría de la evolución y la genética natural. Este tipo de algoritmos son especialmente útiles cuando se abordan problemas de optimización complejos, en los que la exploración del espacio de búsqueda completo no es factible debido a su gran tamaño o a la complejidad de las relaciones entre las variables del problema.

Uno de los aspectos más críticos en el diseño de un algoritmo genético es la elección de los parámetros que controlan su comportamiento. Estos parámetros incluyen:

- El tamaño de la población.
- El número máximo de iteraciones.
- La longitud del cromosoma.
- Las probabilidades de cruce y mutación.

.

La elección de los valores de estos parámetros puede tener un impacto significativo en la eficiencia y la efectividad del algoritmo. Por lo tanto, es crucial llevar a cabo una serie de pruebas para evaluar cómo la variación de estos parámetros afecta al rendimiento del algoritmo.

Motivación

La principal motivación detrás de estas pruebas es obtener una comprensión más profunda de cómo cada uno de los parámetros afecta el comportamiento del algoritmo y la calidad de las soluciones que genera. Al hacerlo, esperamos poder identificar una combinación de parámetros que optimice el rendimiento del algoritmo en términos de la eficacia de la búsqueda y la eficiencia computacional.

Es importante destacar que, aunque nuestro objetivo final es optimizar el rendimiento del algoritmo, no buscamos simplemente encontrar la combinación de parámetros que produzca las mejores soluciones en un conjunto específico de problemas. En su lugar, buscamos entender los principios subyacentes que rigen el comportamiento del algoritmo y cómo estos se relacionan con la elección de los parámetros.

Diseño de las Pruebas

Las pruebas se llevarán a cabo variando sistemáticamente los valores de los parámetros del algoritmo y observando su impacto en el rendimiento del algoritmo. Para cuantificar este rendimiento, se utilizará un índice compuesto que tiene en cuenta tanto la calidad de las soluciones generadas como el tiempo de ejecución y el número de iteraciones realizadas.

El índice se calcula de acuerdo a una fórmula específica (ver sección sobre índice de evaluación) y los pesos asignados a cada factor en el índice pueden ajustarse para reflejar la importancia relativa de cada uno en el contexto del problema que se está abordando.

A lo largo de las pruebas, se registrará el valor de este índice para cada conjunto de parámetros, junto con los valores de los parámetros mismos, en un DataFrame de Pandas para facilitar el análisis de los resultados.

2.3.2. Rúbrica de evaluación

En la evaluación del algoritmo genético, se utilizará un índice compuesto que pondera tres factores principales:

- Tiempo de ejecución.
- Corrección de la respuesta.
- Número de iteraciones realizadas.

Cada uno de estos factores tiene un impacto distinto en la eficiencia general del algoritmo, y se incorporan en el índice mediante un sistema de pesos.

El índice se calcula de la siguiente manera:

Índice = a * (1 / Tiempo de Ejecución) + b * (Corrección de la Respuesta) - c * (Número de Iteraciones)

Donde:

- **Tiempo de Ejecución**: Tiempo total que el algoritmo tarda en completar su ejecución para un conjunto de parámetros determinado.
- **Corrección de la Respuesta**: Factor binario, es decir, toma el valor de 1 si el algoritmo encuentra una solución (la solución es correcta), y 0 en caso contrario (la solución no es correcta).
- **Número de Iteraciones**: Total de iteraciones realizadas por el algoritmo para un conjunto de parámetros determinado.
- **a, b y c**: Pesos[1] asignados a cada uno de estos factores para ajustar su relevancia en el índice.

[1]Estos pesos pueden variar dependiendo de las prioridades del evaluador.
Por ejemplo, si se valora más la rapidez del algoritmo, 'a' tendría un valor más alto. Si se prioriza la corrección de la respuesta, 'b' sería mayor. Si se quiere penalizar fuertemente el uso excesivo de iteraciones, 'c' sería mayor.

La elección de estos pesos puede depender de los objetivos del problema y de la importancia relativa de la eficiencia, la corrección de la respuesta y la economía de iteraciones. Se recomienda ajustar estos pesos en función de los resultados obtenidos durante las pruebas y la evaluación del algoritmo.

La utilización de este índice permitirá evaluar de forma más completa y precisa el desempeño del algoritmo, tomando en cuenta no solo si el algoritmo es capaz de encontrar una solución, sino también cuánto tiempo tarda y cuántas iteraciones necesita para hacerlo. Con esto, se puede obtener una visión más global de la eficiencia del algoritmo y hacer ajustes más informados para mejorar su rendimiento.

2.3.3. Explicación de los parámetros del algoritmo.

Cabe señalar que estos parámetros no operan de forma aislada, sino que interactúan entre sí. La variación de un solo parámetro puede tener efectos en el comportamiento global del algoritmo, y el balance óptimo puede depender del problema específico que se esté intentando resolver. Por eso, a menudo es útil realizar una serie de pruebas para determinar la combinación más eficaz de parámetros para el problema en cuestión.

1. **ELEMENTOS_POBLACION**: Esta variable controla el tamaño de la población en cada generación del algoritmo genético. En general, una población más grande permite una mayor diversidad de individuos, lo que puede ayudar a evitar el estancamiento en óptimos locales. Sin embargo, también aumenta el coste computacional de cada generación.

   Si se incrementa el valor de ELEMENTOS_POBLACION, se podría esperar una mayor capacidad de exploración del espacio de soluciones, pero también un mayor tiempo de ejecución del algoritmo. En cambio, si se disminuye este valor, el tiempo de ejecución será menor, pero también podría disminuir la capacidad de encontrar la solución óptima.

2. **MAX_ITERACIONES**: Este parámetro controla el número máximo de generaciones que el algoritmo genético realizará. Aumentar este valor permite más oportunidades para la evolución y la convergencia hacia la solución óptima, pero también aumenta el tiempo de ejecución.

   Al incrementar MAX_ITERACIONES, el algoritmo tendría más tiempo para buscar la solución, pero a costa de un tiempo de ejecución mayor. Reducir este valor podría acelerar la ejecución, pero puede que el algoritmo no tenga suficiente tiempo para encontrar la mejor solución.

3. **NUM_GENES y CODIGO**: NUM_GENES determina el número de genes en el cromosoma, es decir, la longitud del código. El cromosoma es una representación de una posible solución. CODIGO es el cromosoma objetivo que el algoritmo está intentando encontrar.

   Incrementar NUM_GENES haría que el espacio de búsqueda sea más grande, lo que podría dificultar la búsqueda de la solución, pero podría ser necesario si la solución requiere un cromosoma de mayor longitud.

   Si se reduce este valor, el espacio de búsqueda sería más pequeño, lo que podría facilitar la búsqueda de la solución, pero podría limitar la representación de soluciones.

---

4. **LON_CROMOSOMA**: Esta variable simplemente representa la longitud del cromosoma objetivo (CODIGO), que es igual a NUM_GENES.

   No debería ser modificada independientemente, ya que debe coincidir con la longitud de CODIGO.

---

5. **PROB_MUTACION**: Este parámetro controla la probabilidad de que ocurra una mutación en un individuo durante la generación de la nueva población. La mutación introduce diversidad en la población y ayuda a evitar el estancamiento en óptimos locales.

   Si se incrementa PROB_MUTACION, se introduciría más diversidad en la población, lo que podría ayudar a explorar mejor el espacio de soluciones, pero también podría dificultar la convergencia del algoritmo.

   Si se disminuye esta probabilidad, la población podría converger más rápidamente, pero existe el riesgo de quedarse atascado en un óptimo local.

2.3.4. Preparación del entorno para las pruebas.

El código que se ha presentado hasta el momento es válido y correcto. Pero, para facilitar la ejecución de las pruebas se llevaron a cabo modificaciones: Se realizaron los siguientes cambios en el código desde la versión inicial:

- Se creó una clase llamada AlgoritmoGenetico para encapsular la lógica del algoritmo genético y organizar el código de manera más estructurada y reutilizable.
- El código que estaba originalmente en funciones se movió dentro de métodos de la clase AlgoritmoGenetico. Esto permite una mejor encapsulación de la funcionalidad y facilita la reutilización del código en diferentes partes del algoritmo.
- Se modificó el constructor de la clase AlgoritmoGenetico para recibir parámetros y configurar los atributos del algoritmo. Esto permite flexibilidad al crear instancias del algoritmo con diferentes configuraciones, lo que facilita la experimentación y adaptación a diferentes problemas.
- Se añadieron docstrings a los métodos y argumentos de la clase AlgoritmoGenetico para documentar su funcionamiento y facilitar la comprensión del código.
- Se implementaron métodos como primera_generacion, evaluar_cromosoma, ordenar_poblacion, siguiente_generacion, cruzar, mutacion, y comprobar_mejor_individuo para encapsular las diferentes etapas del algoritmo genético.
- Se creó un método ejecutar que ejecuta el algoritmo genético completo, generando la población inicial, iterando a través de las generaciones, aplicando cruces y mutaciones, y verificando si se ha encontrado el mejor individuo o se ha alcanzado el número máximo de iteraciones.
- Se utilizaron métodos de la clase random en lugar de np.random para generar números aleatorios, ya que los arrays de NumPy tienen una sintaxis ligeramente diferente.

La estructura de clases se adoptó porque ofrece una serie de ventajas:

1. **Encapsulación**: Permite agrupar el código y los datos relacionados en una sola entidad, facilitando la organización y comprensión del código. Los métodos y atributos se mantienen dentro de la clase, evitando la dispersión de la lógica en diferentes partes del programa.

2. **Reutilización**: Al encapsular la funcionalidad en métodos de la clase, se pueden utilizar esos métodos en diferentes partes del programa sin necesidad de duplicar el código. Esto promueve la reutilización y reduce la cantidad de código repetitivo.

3. **Modularidad**: Al dividir el problema en clases y métodos, se puede abordar cada aspecto del algoritmo genético de forma modular e independiente. Esto facilita la comprensión del código y permite realizar modificaciones o mejoras en partes específicas sin afectar al resto del programa.

4. **Flexibilidad**: Al utilizar una estructura de clases, se pueden crear múltiples instancias de la clase AlgoritmoGenetico con diferentes configuraciones y ejecutarlos de forma independiente. Esto permite probar diferentes parámetros y ajustes para adaptarse a problemas específicos o realizar experimentos.

2.4. Desarrollo de las Pruebas.

El objetivo de estas pruebas fue evaluar el desempeño del algoritmo en la tarea de encontrar el código objetivo, así como analizar cómo diferentes configuraciones de variables afectan a su rendimiento.

En este estudio, se realizaron pruebas utilizando diferentes configuraciones de variables clave del algoritmo genético. Estas variables incluyen el tamaño de la población, el número máximo de iteraciones, el número de genes, y la probabilidad de mutación. Para cada configuración de variables, se llevaron a cabo pruebas con distintos valores de n, que representa la cantidad de pruebas realizadas.

Se realizaron tres conjuntos de pruebas con valores de "nïgual a 100, 500 y 1000, respectivamente. Estos conjuntos de pruebas nos permiten analizar cómo el tamaño del conjunto de pruebas afecta al rendimiento del algoritmo genético. Además, se generaron diferentes métricas a partir de los resultados de las pruebas, como la tasa de acierto, el fitness promedio y el número promedio de iteraciones.

2.4.1. Explicación de los gráficos empleados.

1. **Gráfico de Aciertos**:
   El gráfico de aciertos muestra el número de aciertos del algoritmo genético en la búsqueda del código genético objetivo en función del número de pruebas realizadas. Cada acierto representa una prueba en la que el algoritmo encontró el código objetivo. Este gráfico proporciona información sobre la consistencia del algoritmo y su capacidad para encontrar soluciones en diferentes conjuntos de pruebas.

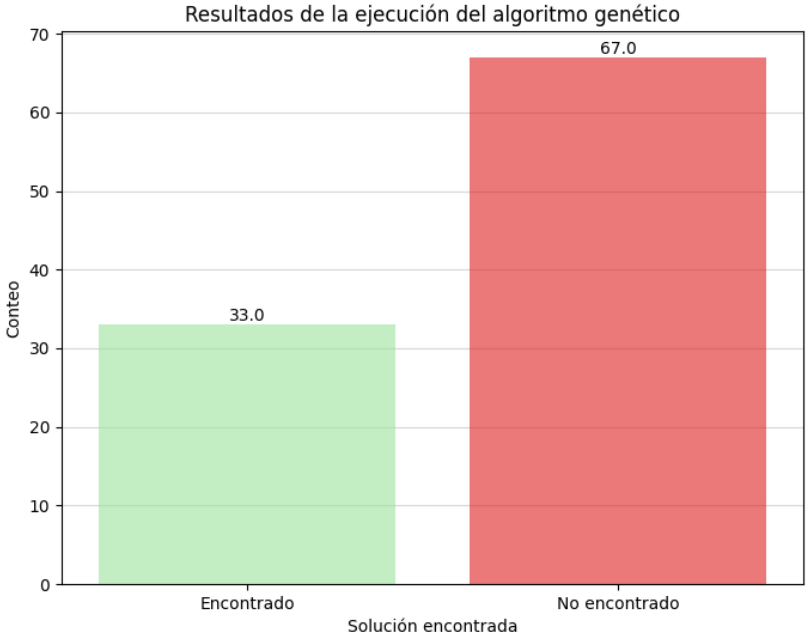

Figura 2.12: Comparativa de aciertos y errores del algoritmo.

2. **Gráfico de Éxito**:
   El gráfico de éxito muestra la tasa de éxito del algoritmo genético en la búsqueda del código genético objetivo. Representa el porcentaje de pruebas en las que el algoritmo logró encontrar el código objetivo dentro del número máximo de iteraciones. Este gráfico es importante para evaluar la eficacia del algoritmo en la tarea de búsqueda y determinar su capacidad para encontrar soluciones óptimas.

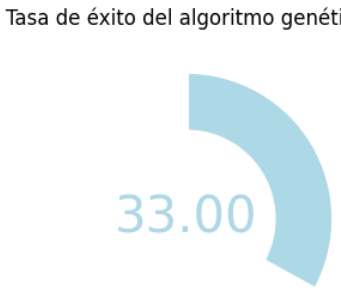

Figura 2.13: Tasa de éxito del algoritmo.

3. **Gráfico de Fitness**:
   El gráfico de fitness evalúa, únicamente, aquellas soluciones que no han conseguido reproducir la respuesta esperada. Esto representa la cantidad de dígitos que nuestro mejor individuo difiere de la solución esperada. Este gráfico es importante para evaluar la efectividad del algoritmo en la optimización y su capacidad para encontrar soluciones de alta aptitud.

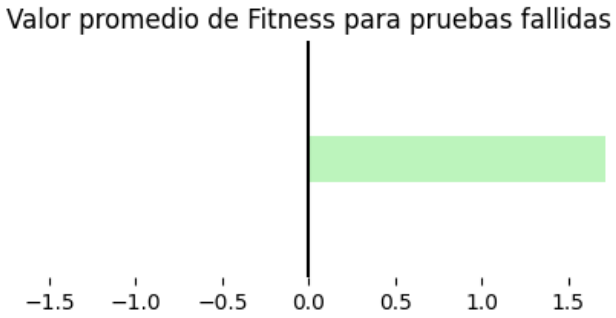

Figura 2.14: Número de elementos no coincidentes con la respuesta esperada.

4. **Gráfico de Resultados**:
   El gráfico de resultados muestra cuanto tiempo ha tardado cada una de las iteraciones, de modo que se pueda definir un rango del tiempo de ejecución del algoritmo.

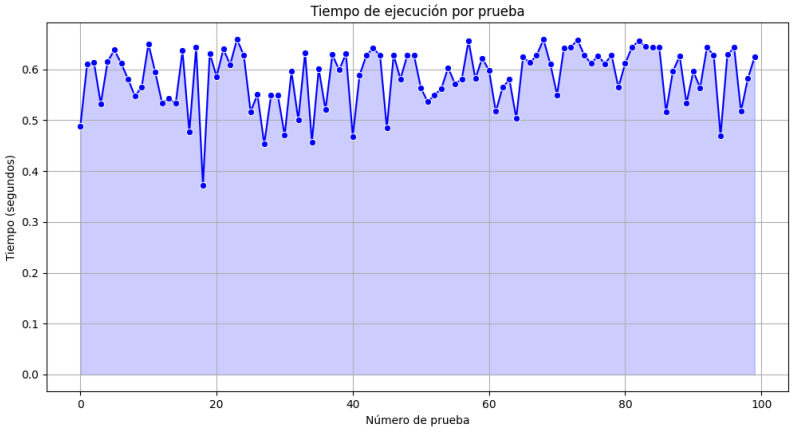

Figura 2.15: Tiempo de ejecución del algoritmo.

5. **Gráfico de Iteraciones**:
   El gráfico de iteraciones muestra el número promedio de iteraciones necesarias para encontrar el código objetivo en cada conjunto de pruebas.
   Proporciona información sobre la eficiencia del algoritmo y su capacidad para encontrar soluciones en un tiempo razonable. Este gráfico es útil para comparar el rendimiento del algoritmo en diferentes configuraciones de variables y tamaños de prueba.

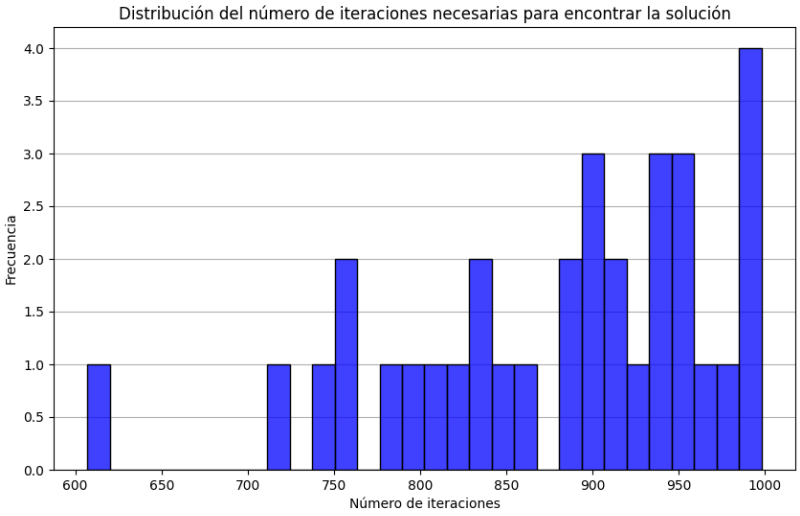

Figura 2.16: Tasa de acierto del algoritmo.

2.4.2. Informes de las pruebas.

Para poder ahorrar espacio y concentrar la información el resto de gráficos se mostrarán en modo de informe.

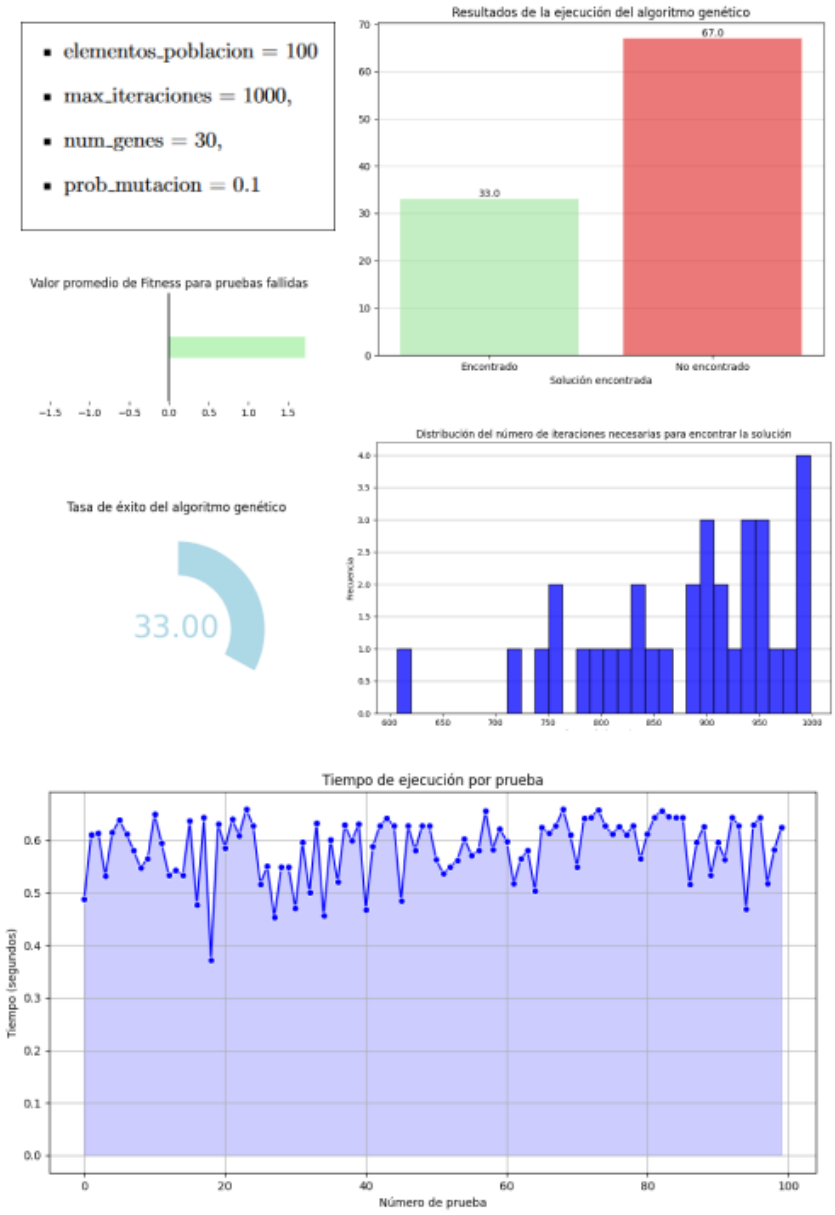

Figura 2.17: Informe configuración 0 y n = 100.

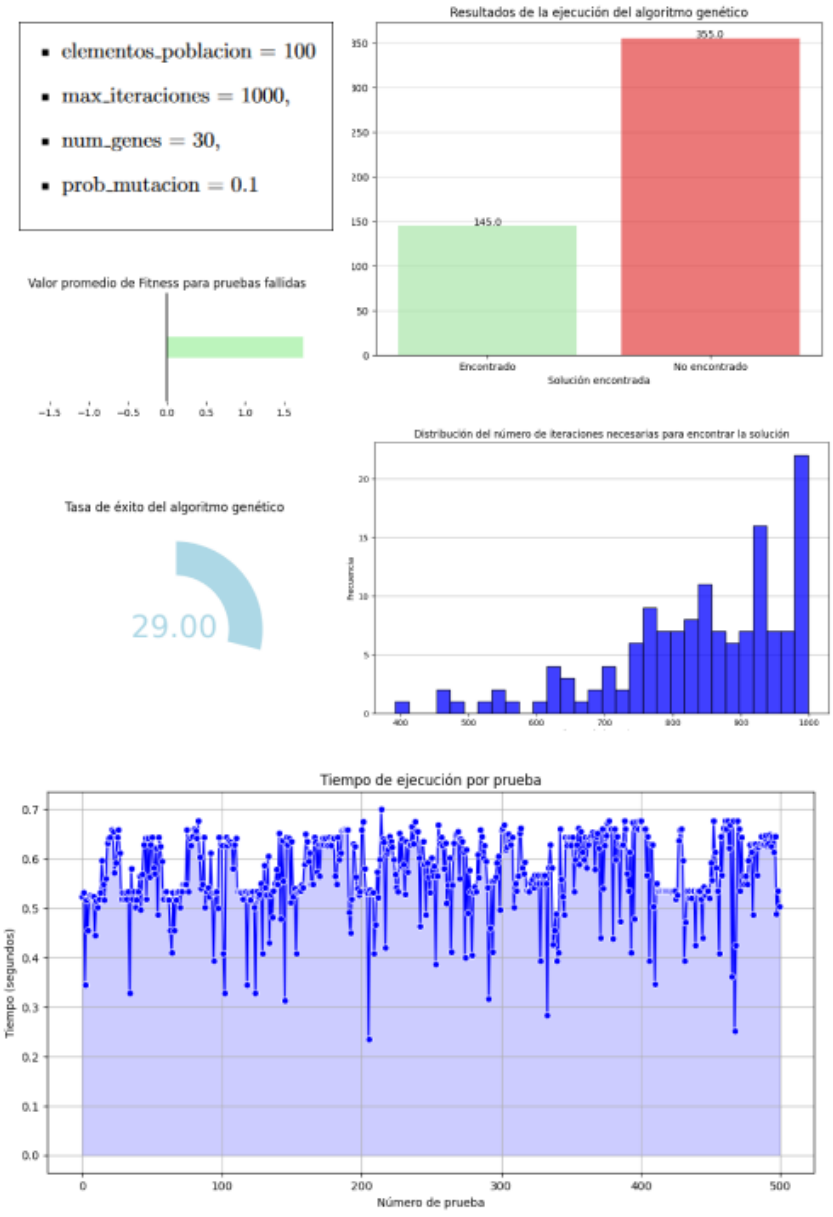

Figura 2.18: Informe configuración 0 y n = 500.

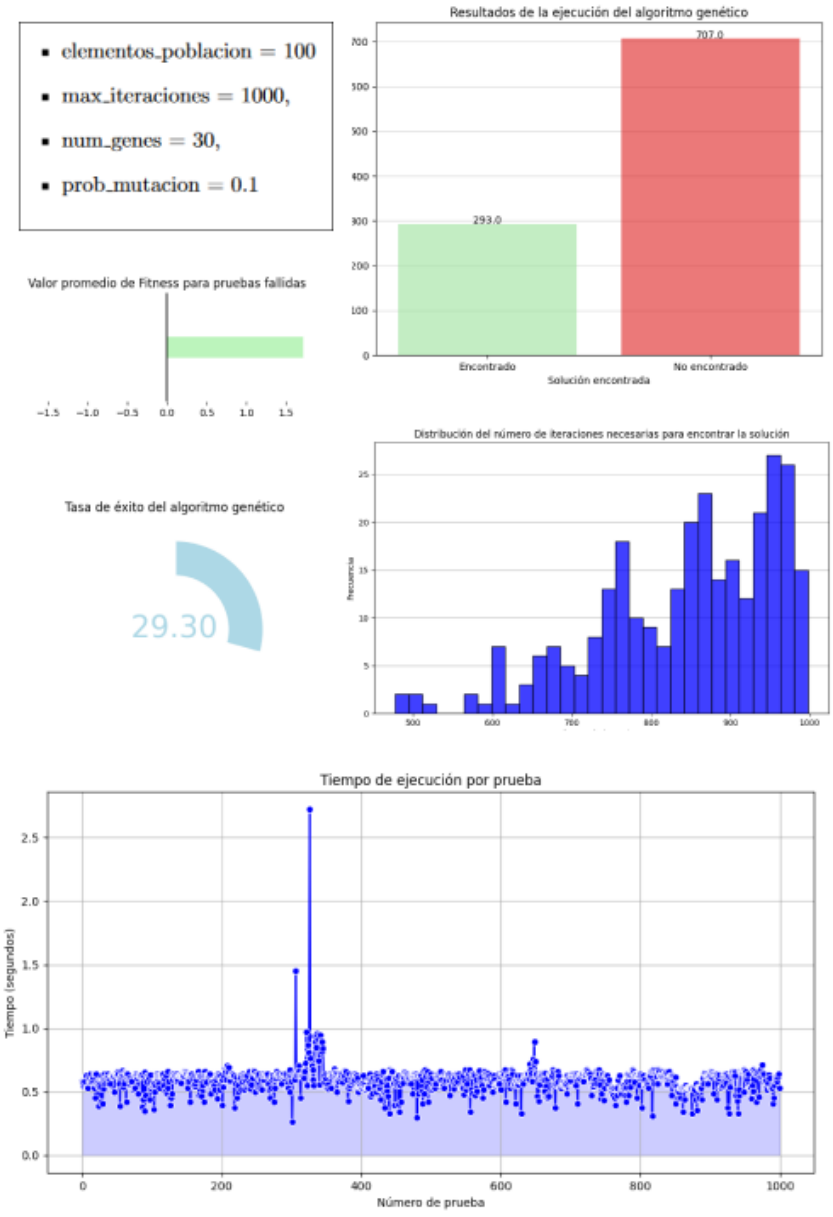

Figura 2.19: Informe configuración 0 y n = 1000.

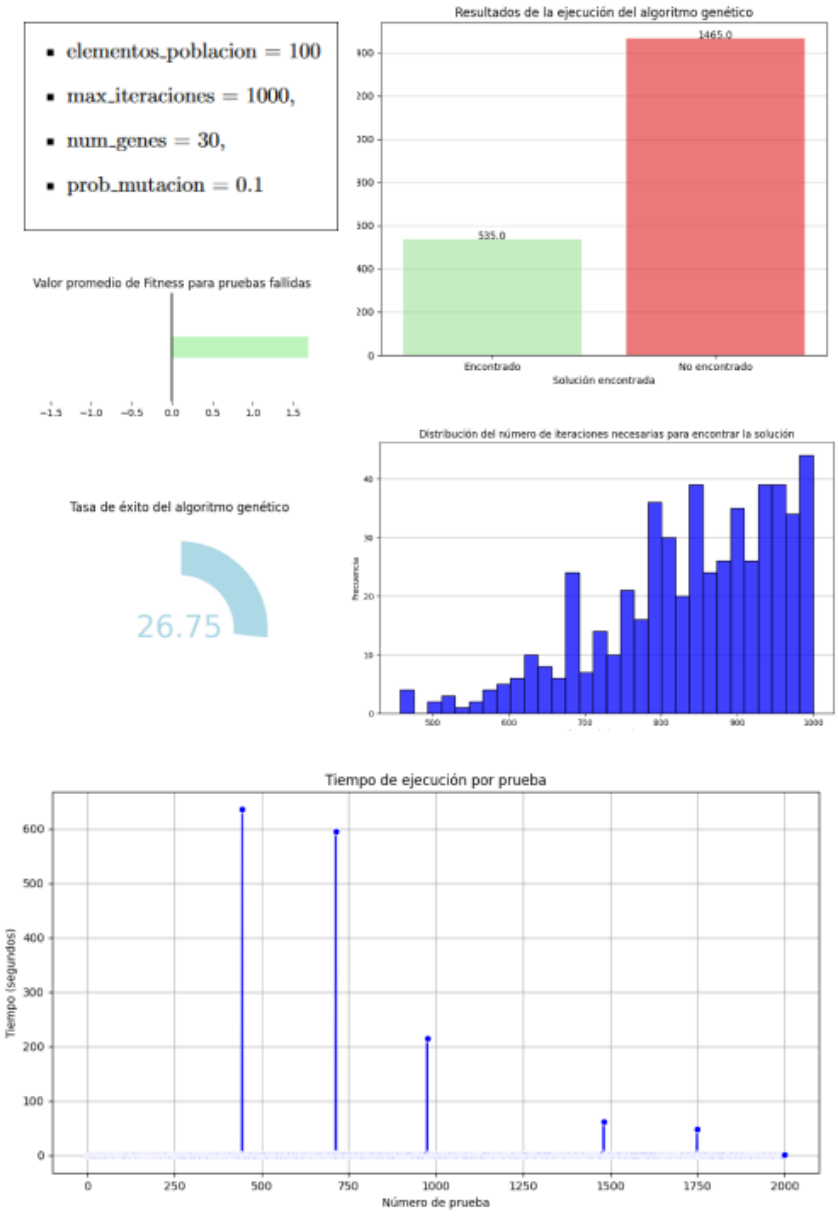

Figura 2.20: Informe configuración 0 y n = 2000.

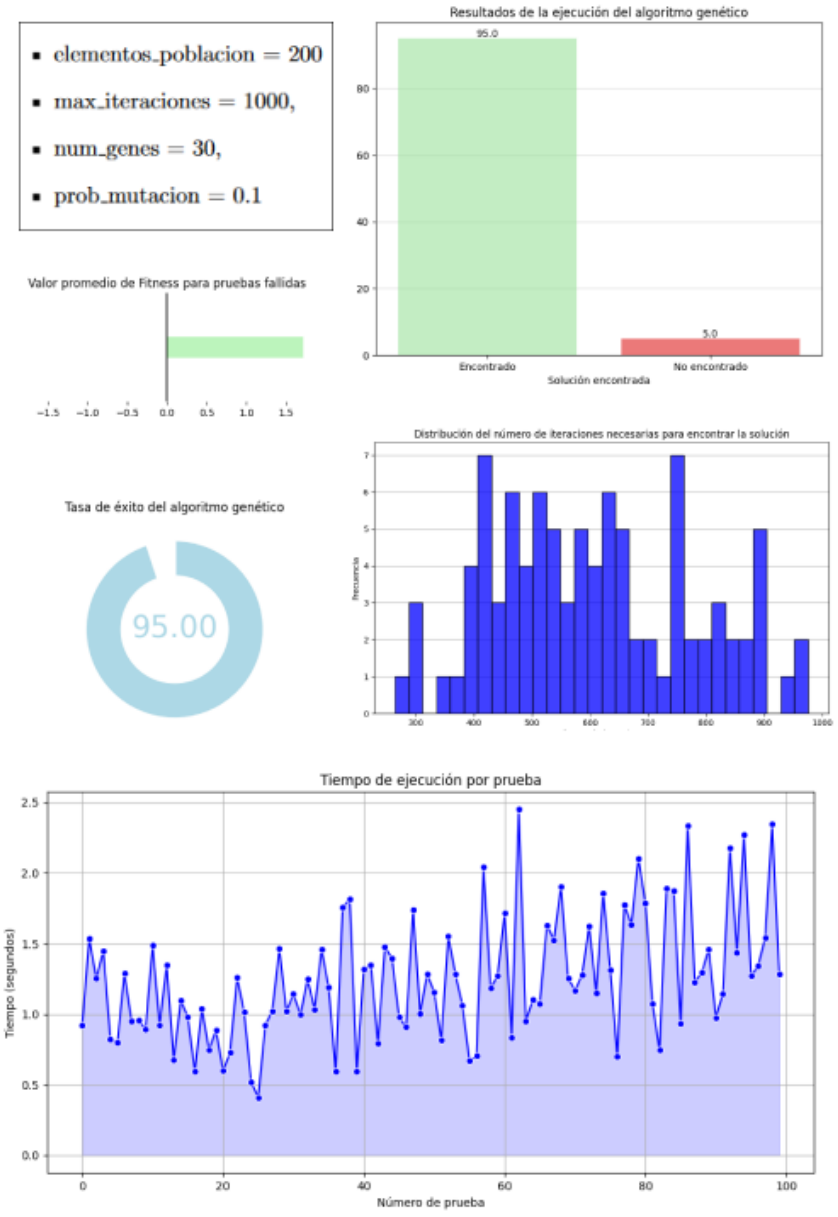

Figura 2.21: Informe configuración 1 y n = 100.

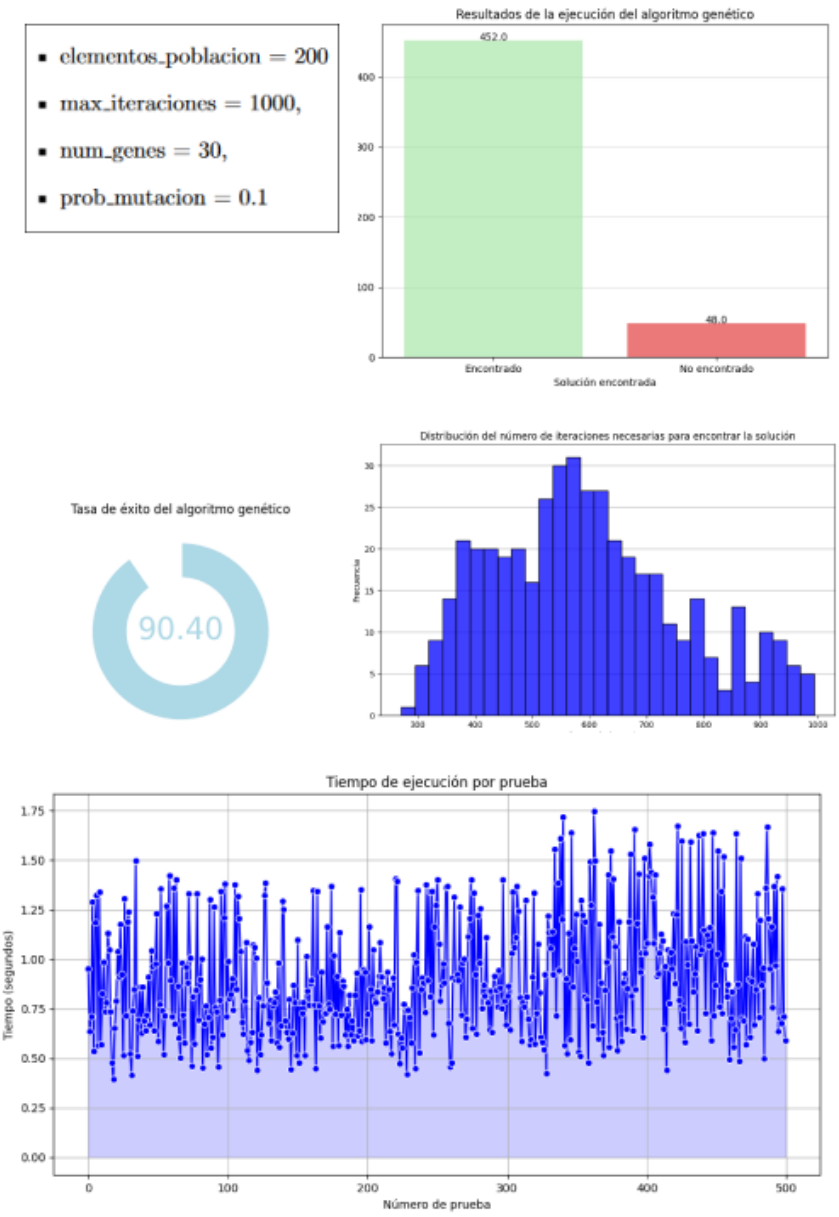

Figura 2.22: Informe configuración 1 y n = 500.

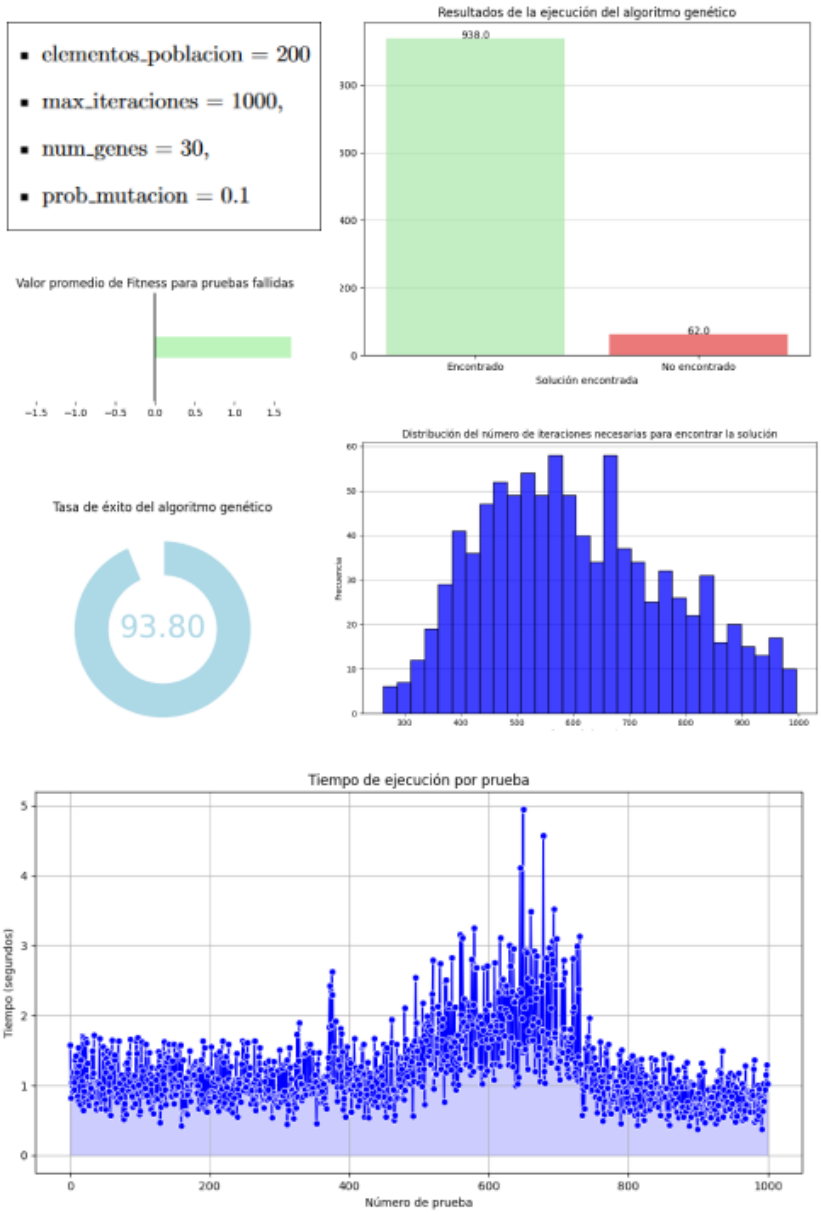

Figura 2.23: Informe configuración 1 y n = 1000.

Figura 2.24: Informe configuración 2 y n = 100.

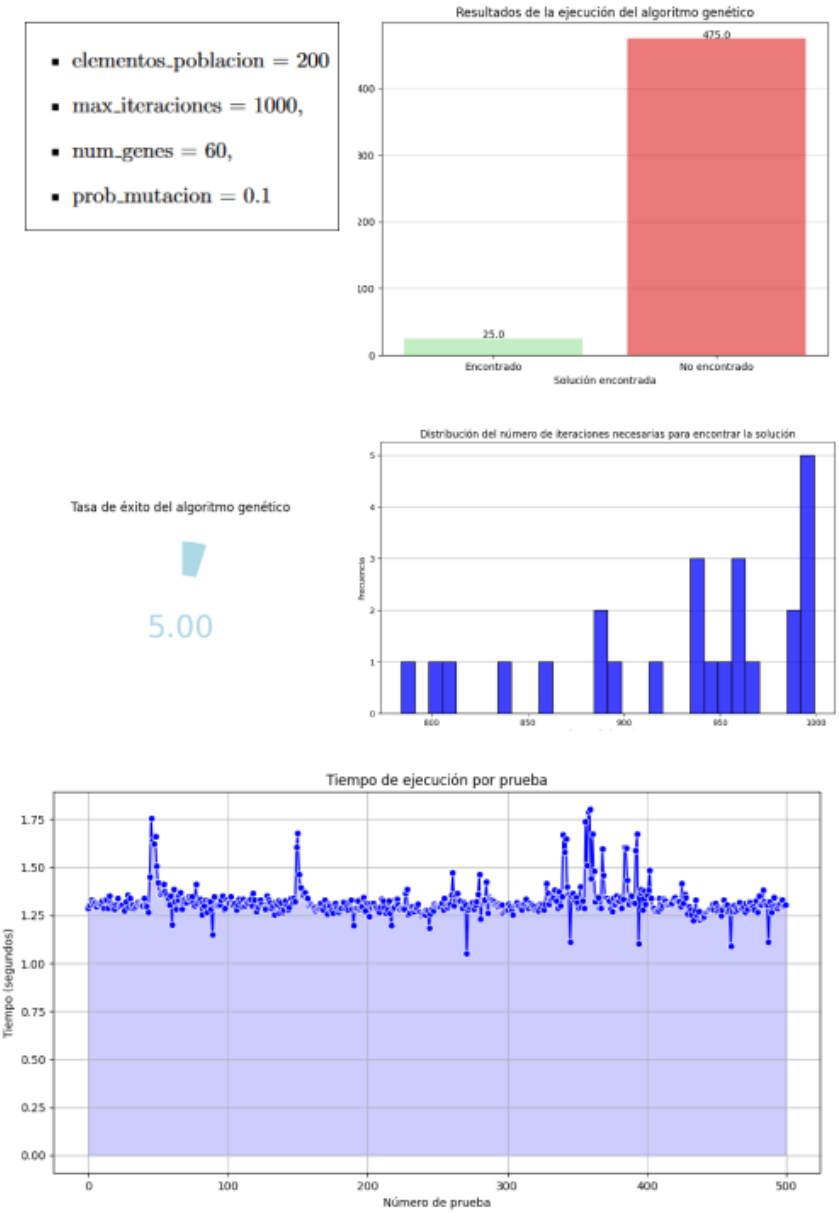

Figura 2.25: Informe configuración 2 y n = 500.

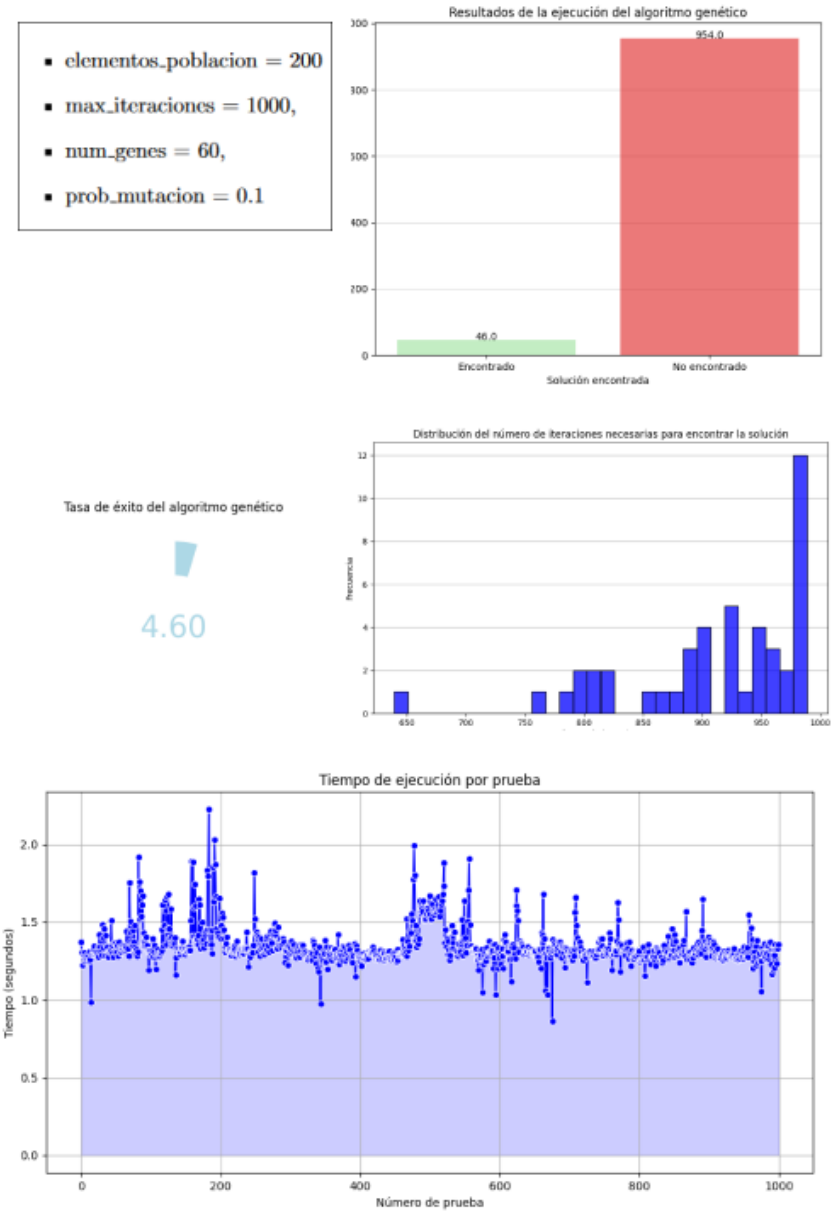

Figura 2.26: Informe configuración 2 y n = 1000.

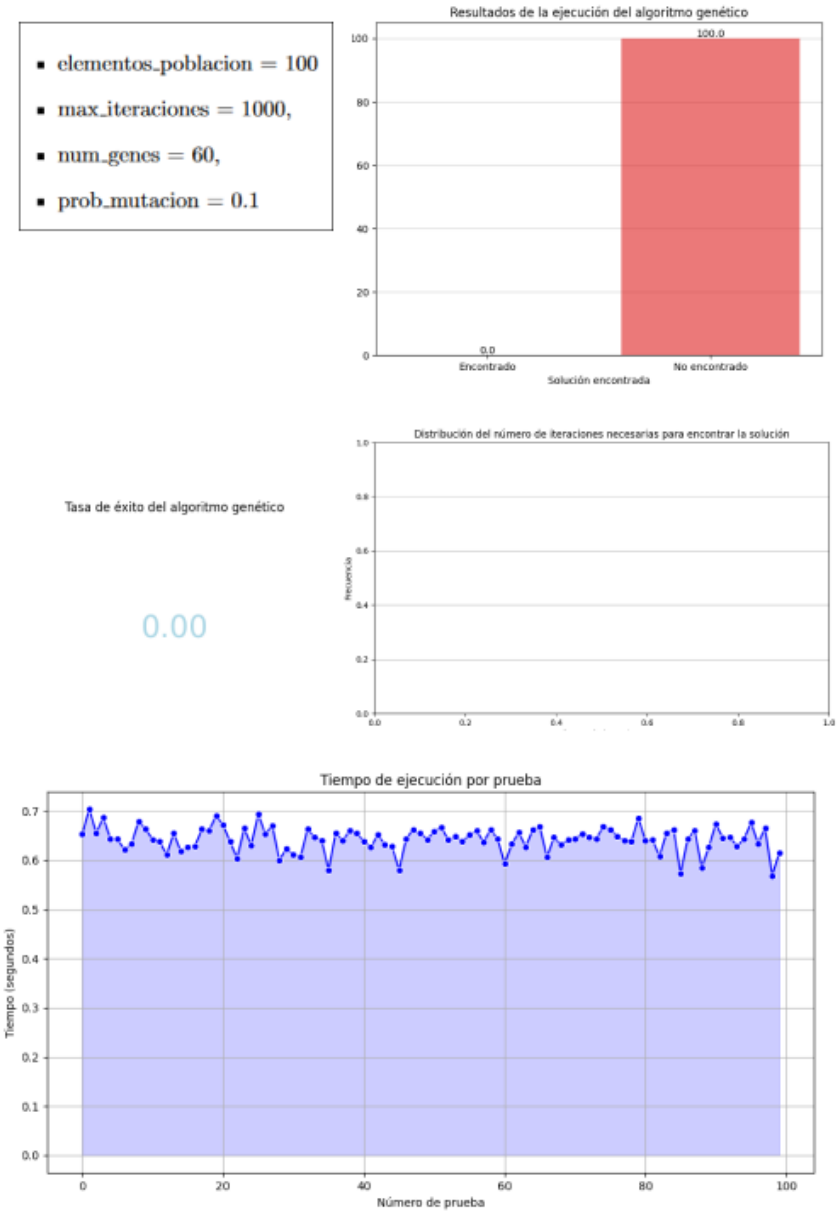

Figura 2.27: Informe configuración 3 y n = 100.

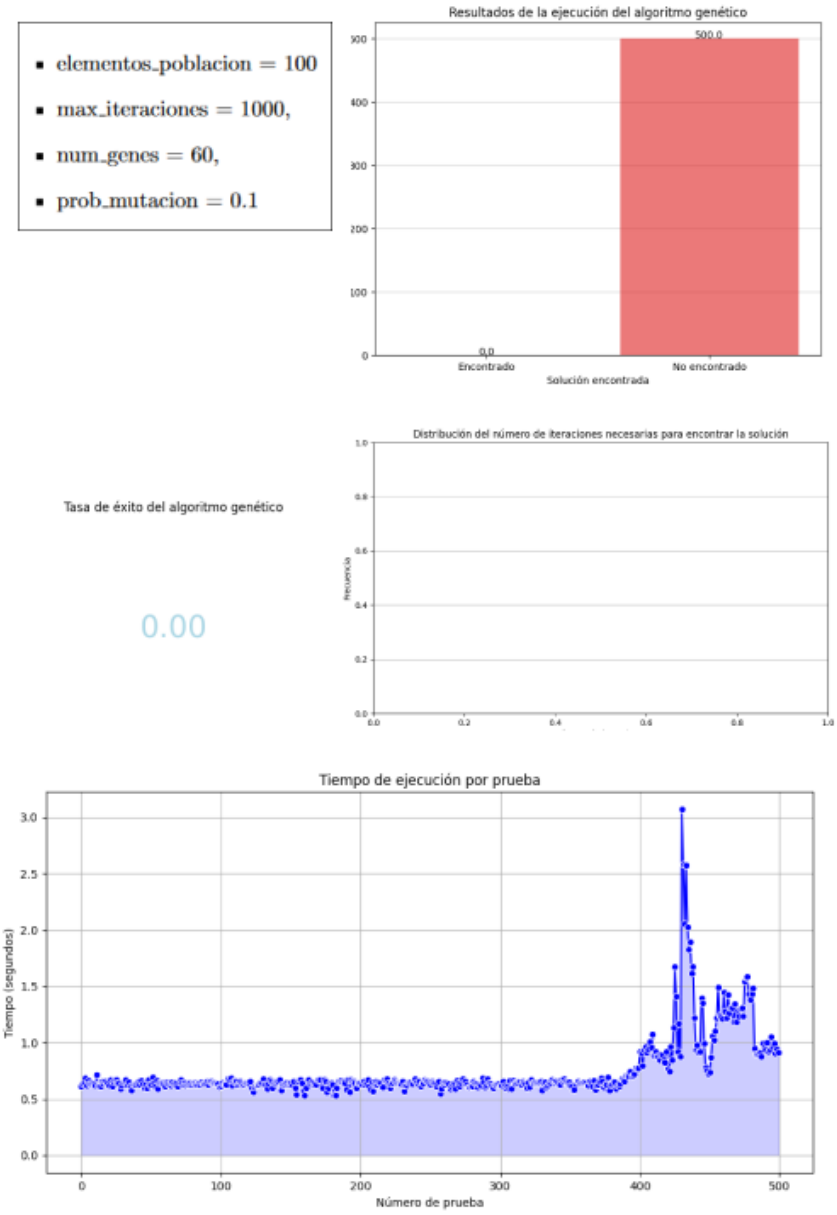

Figura 2.28: Informe configuración 3 y n = 500.

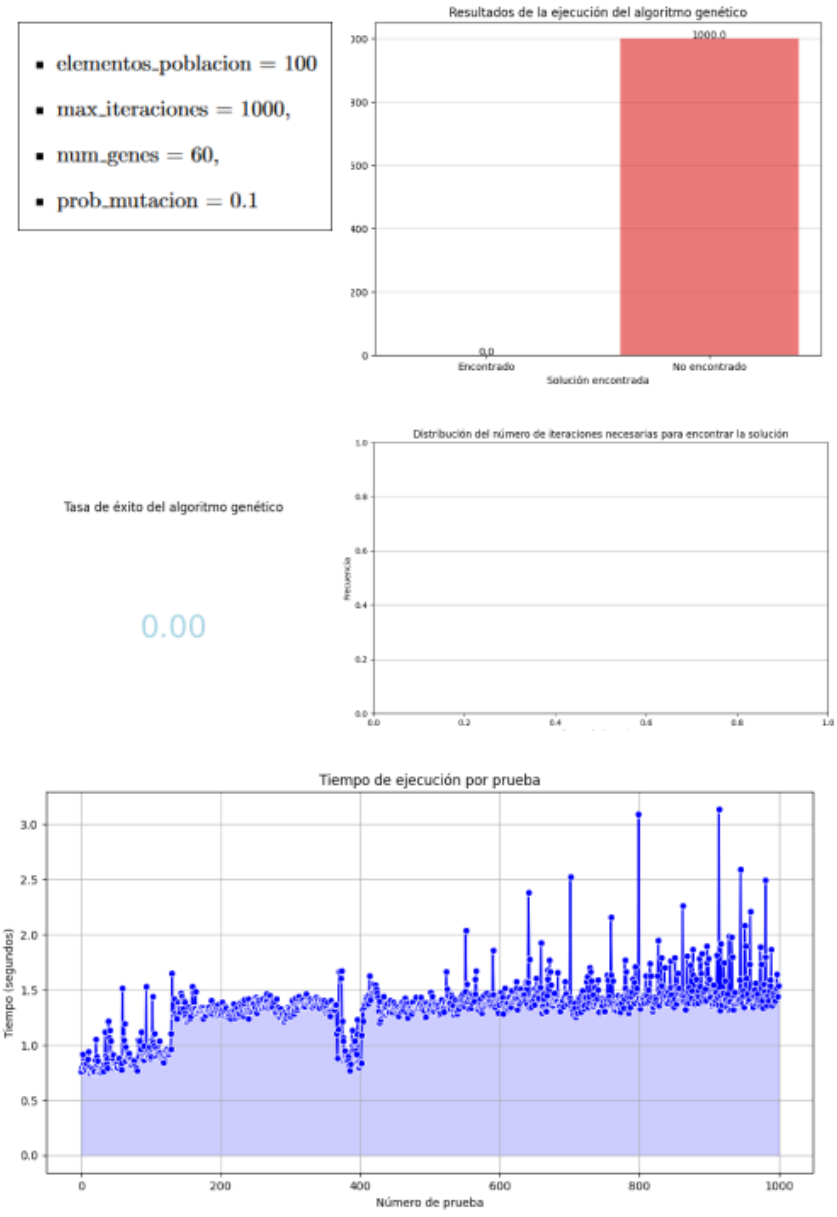

Figura 2.29: Informe configuración 3 y n = 1000.

Figura 2.30: Informe configuración 4 y n = 100.

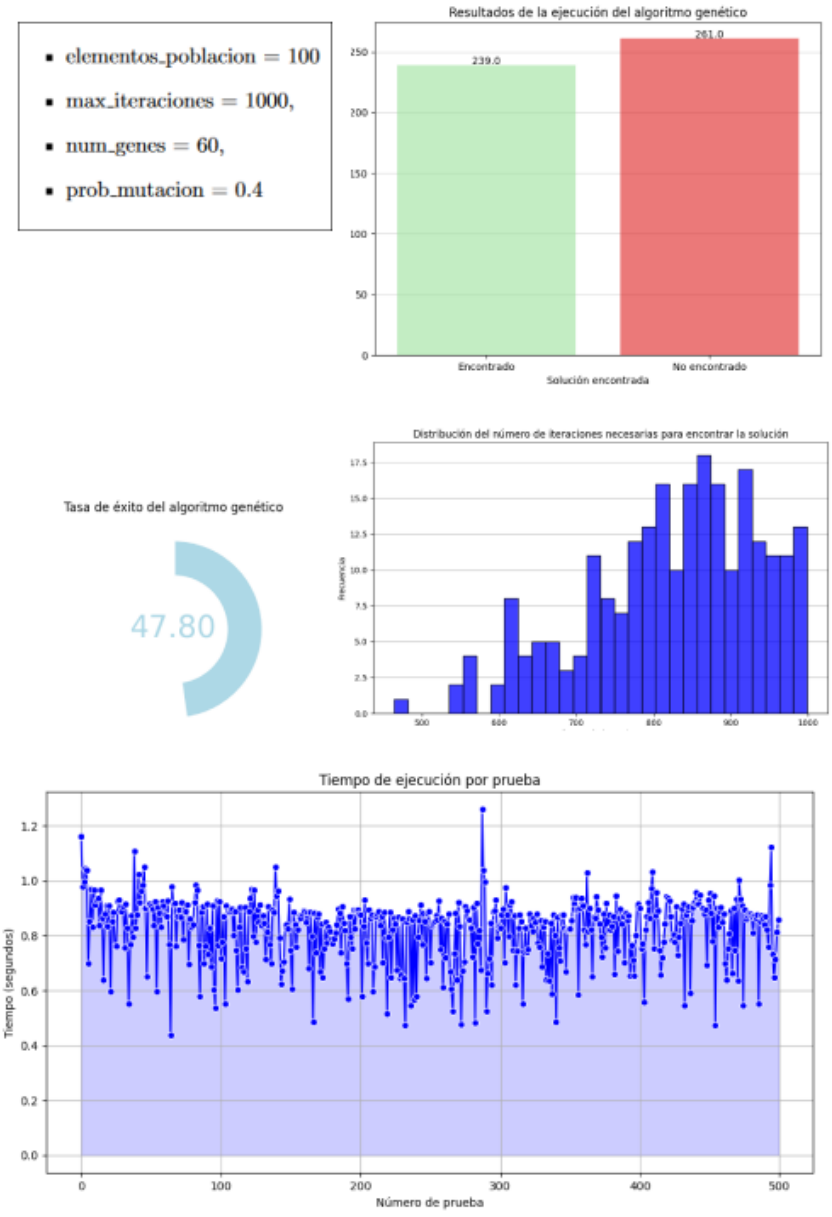

Figura 2.31: Informe configuración 4 y n = 500.

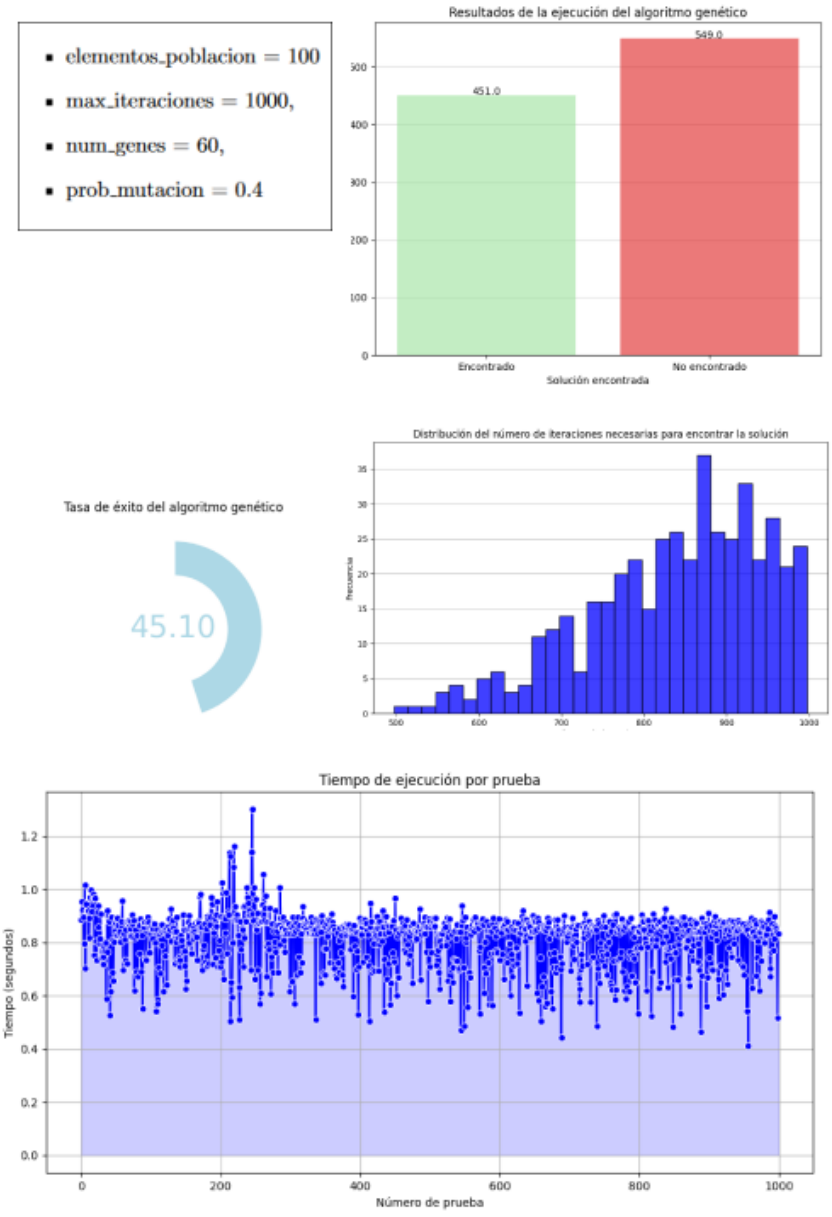

Figura 2.32: Informe configuración 4 y n = 1000.

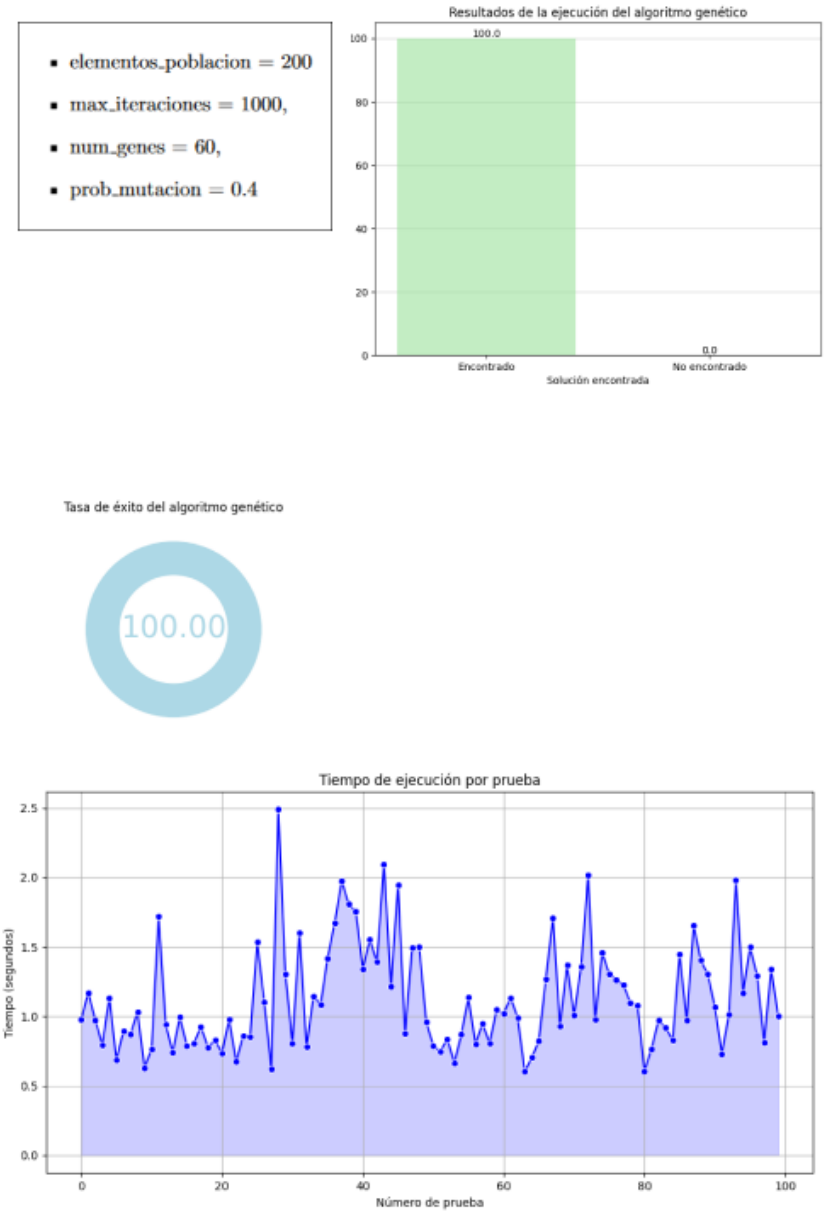

Figura 2.33: Informe configuración 5 y n = 100.

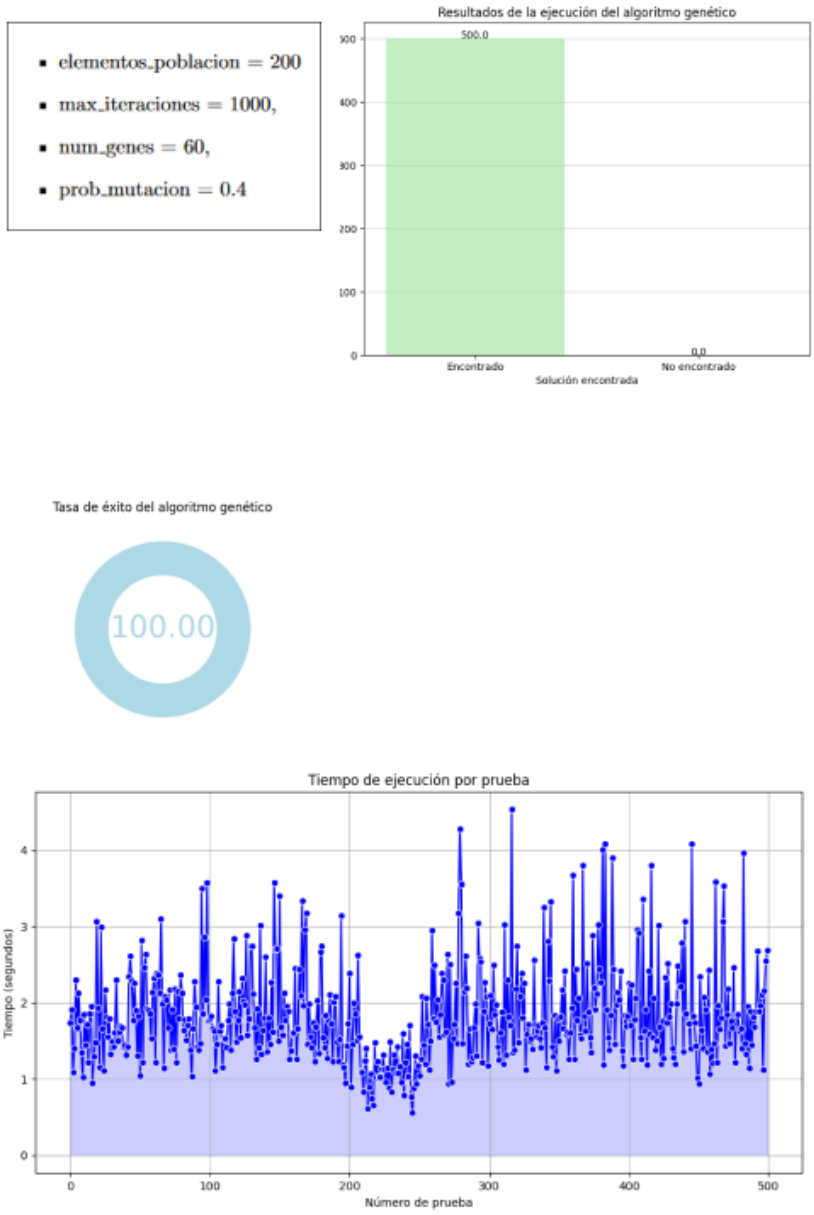

Figura 2.34: Informe configuración 5 y n = 500.

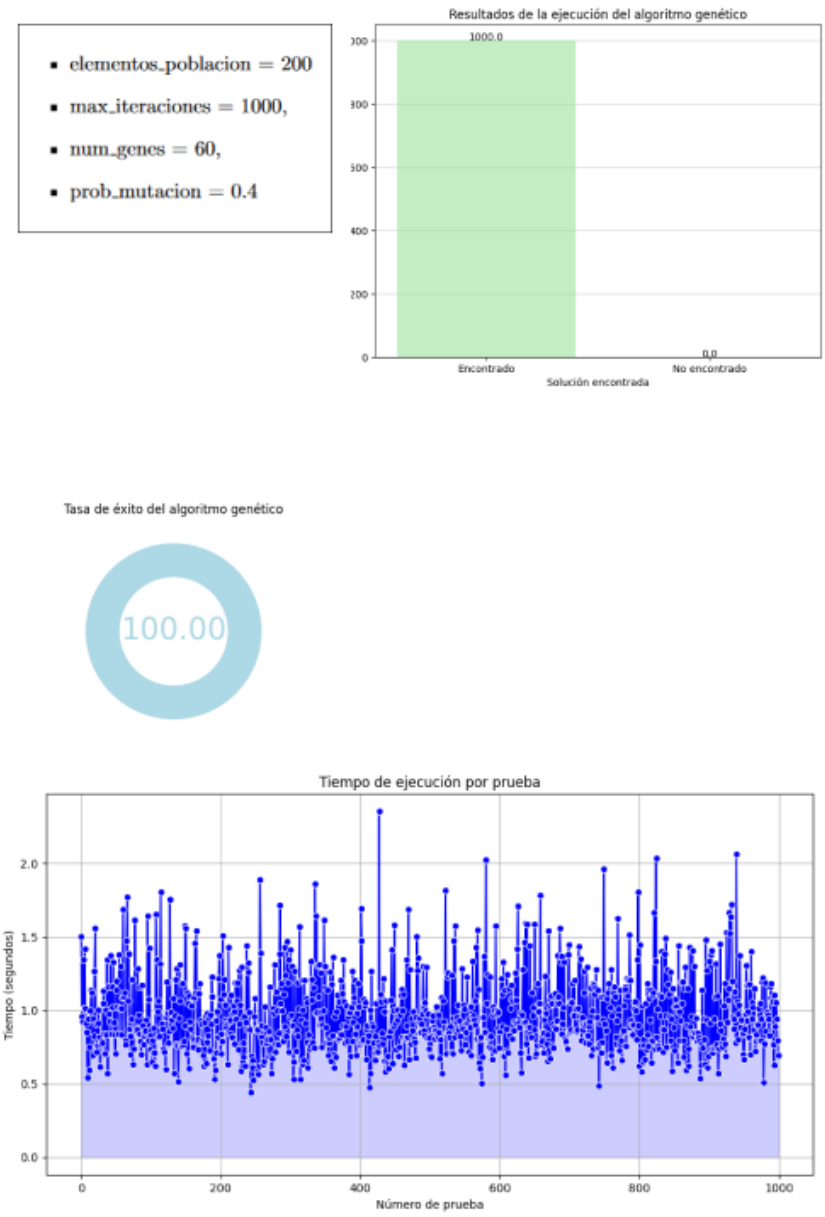

Figura 2.35: Informe configuración 5 y n = 1000.