Introducción a la Computación Evolutiva
Tarea No. 3

Instructor: Dr. Carlos Artemio Coello Coello

Alumno: Ángel Alonso Galarza Chávez

11 de junio de 2025

Introducción

En este trabajo se implementó un algoritmo genético con el objetivo de evaluar el impacto que tienen distintas representaciones de los individuos en el desempeño del algoritmo. Para ello, se utilizaron dos codificaciones: una representación binaria basada en códigos de Gray, y una representación entera donde los genes corresponden a dígitos decimales entre 0 y 9. Ambas implementaciones fueron evaluadas utilizando la función F5 de De Jong como función objetivo, dicha funcion se muestra en la 1.

$$\frac{1}{\frac{1}{K} + \sum_{j=1}^{25} f_j^{-1}(x_1, x_2)}, \quad \text{donde} \quad f_j(x_1, x_2) = c_j + \sum_{i=1}^{2} (x_i - a_{ij})^6, \tag{1}$$

donde: $-65{,}536 \leq x_i \leq 65{,}536$, $K = 500$, $c_j = j$, y:

$$[a_{ij}] = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \cdots & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \cdots & 32 & 32 \end{bmatrix} \tag{3}$$

Como método de selección se utilizó la **selección universal estocástica**, una técnica que tiene como objetivo minimizar la mala distribución de los individuos, siendo una técnica mas justa en selección con respecto a técnicas de selección proporcional como la ruleta, permite una distribución proporcional de los individuos seleccionados de acuerdo con su valor esperado.

El objetivo principal de este tarea fue comparar el rendimiento de ambas representaciones, analizando métricas como la mejor solución alcanzada, la media y desviación estándar de las aptitudes obtenidas, así como el comportamiento de convergencia a lo largo de múltiples corridas independientes con distintas semillas aleatorias.

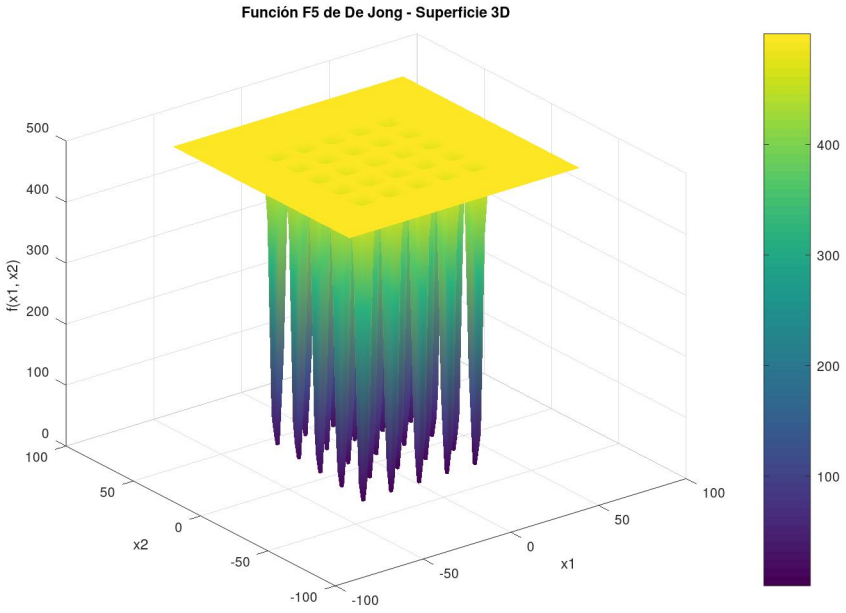

Figura 1: Gráfica de función

Código

En esta sección se presenta el código fuente correspondiente a la implementación de un algoritmo genético. Cada componente del algoritmo desde la representación de los individuos hasta las funciones como decodificación, evaluación (función objetivo), selección, cruza y mutación está descrito mediante funciones en lenguaje C.
En esta sección se presenta el código fuente desarrollado en el lengauje C++ de los modulos esenciales para la representación y selección en el algoritmo genético.

Individuo

setSeed

Esta función permite establecer una semilla específica para el generador de números aleatorios global, lo que asegura que los resultados sean reproducibles en distintas ejecuciones del algoritmo.

Individual(int length, Encoding enc)

Este constructor genera un nuevo individuo inicializado aleatoriamente, dependiendo del tipo de codificación especificada. Para codificación binaria o Gray, crea una cadena de bits aleatorios.
En caso de usar codificación entera, genera una secuencia de dígitos del 0 al 9. Si se usa Gray, realiza la conversión desde binario justo después de la generación. Finalmente, se decodifican los valores para obtener las variables reales asociadas.

Individual(const std::string& cromo, Encoding enc)

Este constructor permite crear un individuo a partir de una cadena de cromosoma ya definida, útil para réplicas exactas o pruebas específicas. Después de asignar el cromosoma, se decodifica para obtener los valores numéricos asociados.

Individual(const std::vectorint & digits, Encoding enc)

Este constructor se utiliza cuando se desea crear un individuo con una representación entera, proporcionando directamente los dígitos del cromosoma. Luego de almacenar los valores, también se ejecuta la decodificación correspondiente.

getChromosome, getDigitChromosome, getFitness, getVariables, getEncoding, getParent1, getParent2, isMutated

Estas funciones proporcionan acceso a las diferentes propiedades del individuo, como su cromosoma en forma de cadena o dígitos, su valor de aptitud, las variables reales que representa, su tipo de codificación, los padres de los que fue generado, y si fue mutado o no.

setChromosome, setDigitChromosome, setFitness, setVariables, setParents, setMutated

Estas funciones permiten modificar directamente los atributos del individuo, como su cromosoma, valores de aptitud, variables reales y relaciones de parentesco. También es posible marcar si ha sido sometido a mutación.

decode

Esta función es fundamental, ya que convierte la representación interna del cromosoma del individuo (ya sea en bits o dígitos) a valores reales que pueden ser evaluados por la función objetivo. En el caso de codificación binaria o Gray, convierte primero a binario (si aplica), luego a decimal y finalmente a un valor real en el rango definido.
En la codificación entera, concatena los dígitos por variable, los normaliza y los transforma también a su correspondiente valor real.

setDecoding

Esta función estática define los parámetros de decodificación para todos los individuos: los valores mínimo y máximo del rango, el número de variables, el número de bits por variable y el tipo de codificación global. Esto establece las bases para interpretar correctamente cualquier cromosoma.

BinToGray

Esta función toma una cadena binaria y la convierte en su equivalente en código Gray. Esto se logra copiando el primer bit tal cual, y luego aplicando una regla simple de comparación entre bits consecutivos. Es útil para mejorar el rendimiento de los algoritmos genéticos en ciertos problemas.

GrayToBin

Esta función convierte una cadena en código Gray a su forma binaria tradicional. Comienza copiando el primer bit, y luego genera los siguientes bits mediante comparaciones sucesivas con el resultado anterior. Es necesaria para poder interpretar y decodificar individuos que fueron codificados en Gray.

size

Devuelve el tamaño del cromosoma del individuo. Este tamaño depende del tipo de codificación que esté utilizando el individuo: puede ser la longitud de la cadena binaria o la cantidad de dígitos en la codificación entera.

printIndividual

Esta función imprime la información más relevante del individuo: su cromosoma (ya sea en bits o dígitos), el valor de aptitud y las variables decodificadas. Es útil para fines de depuración o visualización de resultados.

```cpp
#include "individuo.hpp"

#include <cmath>
#include <iostream>
#include <random>
#include <sstream>
#include <stdexcept>

// INICIALIZACION DE VARIABLES ESTATICAS
double Individual::min_val = 0.0;
double Individual::max_val = 0.0;
int Individual::num_vars = 0;
int Individual::bits_var = 0;
Individual::Encoding Individual::global_encoding = Individual::Encoding::
    BINARY;
```

```cpp
std::mt19937 Individual::_globalGen = std::mt19937(std::random_device{}());

// INICIALIZAR SEMILLA
void Individual::setSeed(unsigned int seed) { _globalGen.seed(seed); }

// CONSTRUCTORES
Individual::Individual(int length, Encoding enc) : _encoding(enc) {
    switch (_encoding) {
        // CASO BINARIO
        case Encoding::BINARY:
        case Encoding::GRAY:
            _cromosoma = std::string(length, '0');
            for (int i = 0; i < length; ++i) {
                _cromosoma[i] = (std::uniform_int_distribution<>(0, 1)(_globalGen)) ? '1' : '0';
            }
            if (_encoding == Encoding::GRAY) {
                _cromosoma = BinToGray(_cromosoma);
            }
            break;

        // CASO ENTERO REAL
        case Encoding::INTEGER_DECIMAL:
            _digitChromosome.resize(length);
            for (int& digit : _digitChromosome) {
                digit = std::uniform_int_distribution<>(0, 9)(_globalGen);
            }
            break;
    }
    decode();
}

Individual::Individual(const std::string& cromo, Encoding enc) : _cromosoma(cromo), _encoding(enc) {
    decode();
}

Individual::Individual(const std::vector<int>& digits, Encoding enc)
    : _digitChromosome(digits), _encoding(enc) {
    decode();
}

// OBTENCION
const std::string& Individual::getChromosome() const { return _cromosoma; }
const std::vector<int>& Individual::getDigitChromosome() const { return _digitChromosome; }
double Individual::getFitness() const { return _fitness; }
const std::vector<double>& Individual::getVariables() const { return _variables; }
Individual::Encoding Individual::getEncoding() const { return _encoding; }
```

```
61
62 std::shared_ptr<Individual> Individual::getParent1() const { return
       _padre1; }
63 std::shared_ptr<Individual> Individual::getParent2() const { return
       _padre2; }
64 bool Individual::isMutated() const { return _mutated; }
65
66 // ACTUALIZACION
67 void Individual::setChromosome(const std::string& chromo) { _cromosoma =
       chromo; }
68 void Individual::setDigitChromosome(const std::vector<int>& digits) {
       _digitChromosome = digits; }
69 void Individual::setFitness(double fit) { _fitness = fit; }
70 void Individual::setVariables(const std::vector<double>& vars) {
       _variables = vars; }
71 void Individual::setParents(std::shared_ptr<Individual> p1, std::
       shared_ptr<Individual> p2) {
72     _padre1 = p1;
73     _padre2 = p2;
74 }
75 void Individual::setMutated(bool mut) { _mutated = mut; }
76
77 // DECODIFICACION
78 void Individual::decode() {
79     _variables.clear();
80     _variables.reserve(num_vars);
81
82     switch (_encoding) {
83         case Encoding::BINARY:
84         case Encoding::GRAY: {
85             std::string cromosoma = _cromosoma;
86             // CONVIERTE DE GRAY A DECIMAL
87             if (_encoding == Encoding::GRAY) {
88                 cromosoma = GrayToBin(_cromosoma);
89             }
90             // CONVIERTE DE BINARIO A REAL
91             for (int i = 0; i < num_vars; ++i) {
92                 int start = i * bits_var;
93                 std::string segment = cromosoma.substr(start, bits_var);
94                 unsigned long intValue = std::stoul(segment, nullptr, 2);
95                 double value =
96                     min_val + (max_val - min_val) * intValue / (std::pow
       (2, bits_var) - 1);
97                 _variables.push_back(value);
98             }
99             break;
100        }
101
102        case Encoding::INTEGER_DECIMAL: {
103            int digits_per_var = _digitChromosome.size() / num_vars;
104            // CONCATENACION DE LOS ENTEROS
105            for (int i = 0; i < num_vars; ++i) {
```

```
106                 std::ostringstream oss;
107                 for (int j = 0; j < digits_per_var; ++j) {
108                     oss << _digitChromosome[i * digits_per_var + j];
109                 }
110                 // CONVIERTE DE ENTERO A REAL
111                 std::string segment = oss.str();
112                 long long intValue = std::stoll(segment);
113                 double normalized = static_cast<double>(intValue) / std::
    pow(10, digits_per_var);
114                 double value = min_val + (max_val - min_val) * normalized;
115                 _variables.push_back(value);
116             }
117             break;
118         }
119     }
120 }
121
122 // REPRESENTACION
123 void Individual::setDecoding(double min, double max, int vars, int bits,
    Encoding encoding) {
124     min_val = min;
125     max_val = max;
126     num_vars = vars;
127     bits_var = bits;
128     global_encoding = encoding;
129 }
130
131 // CODIGO GRAY
132 std::string Individual::BinToGray(const std::string& bin) {
133     std::string gray;
134     gray += bin[0];
135     for (size_t i = 1; i < bin.size(); ++i) {
136         gray += (bin[i - 1] != bin[i]) ? '1' : '0';
137     }
138     return gray;
139 }
140
141 std::string Individual::GrayToBin(const std::string& gray) {
142     std::string bin;
143     bin += gray[0];
144     for (size_t i = 1; i < gray.size(); ++i) {
145         bin += (gray[i] != bin[i - 1]) ? '1' : '0';
146     }
147     return bin;
148 }
149
150 // METODOS AUXILIARES
151 int Individual::size() const {
152     switch (_encoding) {
153         case Encoding::BINARY:
154         case Encoding::GRAY:
155             return static_cast<int>(_cromosoma.size());
```

```
156         case Encoding::INTEGER_DECIMAL:
157             return static_cast<int>(_digitChromosome.size());
158     }
159     return 0;
160 }
161 
162 void Individual::printIndividual() const {
163     std::cout << "Cromosoma: ";
164     if (_encoding == Encoding::INTEGER_DECIMAL) {
165         for (int digit : _digitChromosome) {
166             std::cout << digit;
167         }
168     } else {
169         std::cout << _cromosoma;
170     }
171     std::cout << "\nFitness: " << _fitness << "\nVariables: ";
172     for (double var : _variables) {
173         std::cout << var << " ";
174     }
175     std::cout << std::endl;
176 }
```

Selección

Universal_Estocastica::selectMany

Este método implementa la estrategia de selección universal estocástica, una variante de la ruleta que garantiza una distribución más uniforme y menos aleatoria. Primero se calcula el valor esperado de cada individuo, que representa cuántas veces debería ser seleccionado según su aptitud relativa.
Luego, se utiliza un puntero aleatorio inicial (ptr) que avanza en pasos constantes para seleccionar a los individuos.
Esto permite una cobertura más equitativa del espacio de selección, mejorando la diversidad genética y reduciendo el riesgo de seleccionar siempre a los mismos individuos. El método retorna un vector con los individuos seleccionados.

SelectionFactory::create

Esta función de fábrica permite crear dinámicamente una estrategia de selección según el tipo especificado: ruleta o torneo. En el caso de torneo, también se requiere el tamaño del torneo. Si el tipo no es reconocido, se lanza una excepción para alertar del error.

```
1 #include "seleccion.hpp"
2 
3 #include <algorithm>
4 #include <numeric>
5 #include <stdexcept>
6 
```

```cpp
// IMPLEMENTACION UNIVERSAL ESTOCASTICA
std::vector<std::shared_ptr<Individual>> Universal_Estocastica::selectMany(
    const Population& population, int numToSelect) const {
    if (population.size() == 0 || numToSelect <= 0) {
        throw std::runtime_error("ERROR DE TAMANIO DE LA POBLACION");
    }

    double avgFitness = population.getAverageFitness();

    // CALCULAR VALORES ESPERADOS
    std::vector<double> expectedValues;
    for (const auto& ind : population.getIndividuals()) {
        expectedValues.push_back(ind->getFitness() / avgFitness);
    }

    std::vector<std::shared_ptr<Individual>> selected;

    double ptr = std::uniform_real_distribution<>(0.0, 1.0)(_gen);
    double cumulativeSum = 0.0;
    size_t i = 0;

    // SELECCIONAR N INDIVIDUOS
    while (selected.size() < static_cast<size_t>(numToSelect)) {
        if (i >= expectedValues.size())
            break;
        cumulativeSum += expectedValues[i];
        // SI PTR ES MENOR Y LA SELECCION DE INDIVIDUOS ES MENOR A LA CANTIDAD NECESARIA
        // SELECCIONAR(I)
        while (ptr < cumulativeSum && selected.size() < static_cast<size_t>(numToSelect)) {
            selected.push_back(population.getIndividual(i));
            ptr += 1.0;
        }
        ++i;
    }

    // RETORNAR VECTORES CON LOS INDIVIDUOS DE LOS SELECCIONADOS
    return selected;
}

// SELECCION DE ESTRATEGIA
std::unique_ptr<ISelectionStrategy> SelectionFactory::create(Type type, int tournamentSize) {
    switch (type) {
        case Type::RULETA:
            return std::make_unique<Ruleta>();
        case Type::TORNEO:
            return std::make_unique<Torneo>(tournamentSize);
        default:
            throw std::invalid_argument("TIPO DE SELECCION DESCONOCIDA");
```

```
    }
}
```

Cruza

UniformCrossover::setSeed

Permite establecer una semilla para el generador de números aleatorios utilizado durante la cruza.

UniformCrossover::UniformCrossover

Este constructor inicializa el operador de cruza uniforme con dos parámetros: la probabilidad general de cruza entre dos padres y la probabilidad de intercambio genético por cada posición del cromosoma. Valida que ambas probabilidades se encuentren dentro del rango [0, 1].

UniformCrossover::crossover

Esta función implementa la cruza uniforme, una técnica de recombinación donde cada posición del cromosoma tiene una probabilidad independiente de intercambiar información entre los padres. Si la probabilidad de cruza general no se cumple, los hijos son réplicas exactas de los padres. De lo contrario, se evalúa posición por posición si se intercambian los genes o no. El algoritmo admite tanto codificación binaria y Gray como codificación entera, realizando cruza de bits o de dígitos según corresponda. Al finalizar, se crean dos nuevos individuos (hijos), se les asignan sus respectivos padres, y se retornan como resultado.

CrossoverFactory::create

Esta función de fábrica permite construir dinámicamente un operador de cruza según el tipo especificado. Actualmente, soporta la cruza uniforme. Si se pasa un tipo de cruza no reconocido, lanza una excepción.

```
#include "cruza.hpp"
#include <memory>
#include <stdexcept>
#include "individuo.hpp"

std::mt19937 UniformCrossover::_globalGen = std::mt19937(std::
    random_device{}());

// INICIALIZACION DE LA SEMILLA
void UniformCrossover::setSeed(unsigned int seed) { _globalGen.seed(seed);
     }

// CONSTRUCTOR DE LA CRUZA UNIFORME
UniformCrossover::UniformCrossover(double crossoverProb, double
    crossoverGenProb)
    : _crossoverProb(crossoverProb), _crossoverGenProb(crossoverGenProb) {
```

```
    if (crossoverProb < 0.0 || crossoverProb > 1.0 || crossoverGenProb < 0.0 ||
        crossoverGenProb > 1.0) {
        throw std::invalid_argument("PARAMETROS DE PROBABILIDADES FUERA DEL RANGO [0, 1]");
    }
}

// FUNCION CRUZA UNIFORME
std::pair<std::shared_ptr<Individual>, std::shared_ptr<Individual>> UniformCrossover::crossover(
    const std::shared_ptr<Individual>& parent1, const std::shared_ptr<Individual>& parent2) const {
    if (parent1->size() != parent2->size()) {
        throw std::invalid_argument("NO COINCIDEN LA LONGUITUDES DE LOS CROMOSOMAS DE LOS PADRES");
    }

    std::uniform_real_distribution<double> dist(0.0, 1.0);
    Individual::Encoding enc = parent1->getEncoding();

    // PASAR LOS PADRES INTACTOS A LOS HIJOS
    if (dist(_globalGen) >= _crossoverProb) {
        auto child1 = std::make_shared<Individual>(*parent1);
        auto child2 = std::make_shared<Individual>(*parent2);
        child1->setParents(parent1, parent2);
        child2->setParents(parent1, parent2);
        return {child1, child2};
    }

    // CRUZA PARA REPRESENTACION BINARIA
    if (enc == Individual::Encoding::BINARY || enc == Individual::Encoding::GRAY) {
        std::string p1 = parent1->getChromosome();
        std::string p2 = parent2->getChromosome();

        // if (enc == Individual::Encoding::GRAY) {
        //     p1 = Individual::GrayToBin(p1);
        //     p2 = Individual::GrayToBin(p2);
        // }

        std::string child1, child2;
        child1.reserve(p1.size());
        child2.reserve(p1.size());

        for (size_t i = 0; i < p1.size(); ++i) {
            if (dist(_globalGen) < _crossoverGenProb) {
                child1 += p1[i];
                child2 += p2[i];
            } else {
                child1 += p2[i];
                child2 += p1[i];
```

```
            }
        }

        // if (enc == Individual::Encoding::GRAY) {
        //     child1 = Individual::BinToGray(child1);
        //     child2 = Individual::BinToGray(child2);
        // }

        auto offspring1 = std::make_shared<Individual>(child1, enc);
        auto offspring2 = std::make_shared<Individual>(child2, enc);
        offspring1->setParents(parent1, parent2);
        offspring2->setParents(parent1, parent2);
        return {offspring1, offspring2};
    }

    // CRUZA PARA REPRESENTACION ENTERA
    if (enc == Individual::Encoding::INTEGER_DECIMAL) {
        const auto& p1 = parent1->getDigitChromosome();
        const auto& p2 = parent2->getDigitChromosome();

        std::vector<int> child1(p1.size()), child2(p2.size());
        for (size_t i = 0; i < p1.size(); ++i) {
            if (dist(_globalGen) < _crossoverGenProb) {
                child1[i] = p1[i];
                child2[i] = p2[i];
            } else {
                child1[i] = p2[i];
                child2[i] = p1[i];
            }
        }

        // CREA A LOS HIJOS Y LOS RETORNA
        auto offspring1 = std::make_shared<Individual>(child1, enc);
        auto offspring2 = std::make_shared<Individual>(child2, enc);
        offspring1->setParents(parent1, parent2);
        offspring2->setParents(parent1, parent2);
        return {offspring1, offspring2};
    }

    throw std::runtime_error("SE DESCONOCE LA CODIFICACION");
}

// PATRON FACTORY DE LA CRUZA
std::unique_ptr<ICrossoverOperator> CrossoverFactory::create(Type type,
    double crossoverProb,
                                                             double
    crossoverGenProb) {
    switch (type) {
        case Type::UNIFORM:
            return std::make_unique<UniformCrossover>(crossoverProb,
    crossoverGenProb);
        default:
```

```
109             throw std::invalid_argument("TIPO DE CRUZA DESCONOCIDO");
110     }
111 }
```

Mutación

UniformMutation::setSeed

Permite establecer una semilla para el generador de números aleatorios utilizado en la mutación.

UniformMutation::UniformMutation

Este constructor inicializa el operador de mutación uniforme con una tasa de mutación especificada por el usuario. La tasa debe estar en el rango [0, 1]; de lo contrario, se lanza una excepción para evitar errores de configuración.

UniformMutation::mutate

Esta función aplica el operador de mutación uniforme sobre un individuo, modificando su información genética con una probabilidad determinada por la tasa de mutación. El comportamiento varía según el tipo de codificación del individuo:

- **Codificación binaria o Gray:** Se recorre cada bit del cromosoma y, con cierta probabilidad, se invierte su valor (de '0' a '1' o de '1' a '0').
- **Codificación entera:** Se recorre cada dígito del cromosoma y, con la misma probabilidad, se reemplaza por un valor aleatorio entre 0 y 9.

Al finalizar cualquier tipo de mutación, se decodifica nuevamente el cromosoma para actualizar las variables reales del individuo, asegurando coherencia con la representación interna.

MutationFactory::create

Esta función de fábrica crea un operador de mutación según el tipo especificado. Actualmente soporta la mutación uniforme. Si se indica un tipo desconocido, se lanza una excepción para alertar del error.

```
1 #include "mutacion.hpp"
2
3 #include <algorithm>
4 #include <stdexcept>
5
6 std::mt19937 UniformMutation::_globalGen = std::mt19937(std::random_device
    {}());
7 // ASIGNACION DE SEMILLA
8 void UniformMutation::setSeed(unsigned int seed) { _globalGen.seed(seed);
    }
```

```
9
10 // CONSTRUCTOR
11 UniformMutation::UniformMutation(double mutationRate) : _mutationRate(
       mutationRate) {
12     if (_mutationRate < 0.0 || _mutationRate > 1.0) {
13         throw std::invalid_argument("LA TASA DE MUTACION DEBE DE ESTAR EN
       EL RANGO [0, 1]");
14     }
15 }
16
17 // OPERADOR DE MUTACION
18 void UniformMutation::mutate(std::shared_ptr<Individual> individual) const
       {
19     std::uniform_real_distribution<double> probDist(0.0, 1.0);
20     auto encoding = individual->getEncoding();
21
22     switch (encoding) {
23         case Individual::Encoding::BINARY:
24         case Individual::Encoding::GRAY: {
25             std::string chromo = individual->getChromosome();
26             // HACE LA INVERSION DE UN BIT (0 => 1) O (1 => 0)
27             for (char& bit : chromo) {
28                 if (probDist(_globalGen) < _mutationRate) {
29                     bit = (bit == '0') ? '1' : '0';
30                 }
31             }
32             individual->setChromosome(chromo);
33             break;
34         }
35
36         // MUTACION PARA REPRESENTACION ENTERA
37         case Individual::Encoding::INTEGER_DECIMAL: {
38             auto digits = individual->getDigitChromosome();
39             std::uniform_int_distribution<int> digitDist(0, 9);
40             // GENERA UN DIGITO ALEATORIO EN EL RANGO [0, 9]
41             for (int& digit : digits) {
42                 if (probDist(_globalGen) < _mutationRate) {
43                     digit = digitDist(_globalGen);
44                 }
45             }
46             individual->setDigitChromosome(digits);
47             break;
48         }
49     }
50
51     individual->decode();
52 }
53
54 // FACTORY
55 std::unique_ptr<IMutationOperator> MutationFactory::create(Type type,
       double mutationRate) {
56     switch (type) {
```

```
        case Type::UNIFORME:
            return std::make_unique<UniformMutation>(mutationRate);
        default:
            throw std::invalid_argument("TIPO DE MUTACION DESCONOCIDO");
    }
}
```

Función Objetivo

DeJong::F5::evaluate

Esta función implementa la conocida **función F5 de De Jong**. Su propósito es evaluar la aptitud de una solución representada por un vector de dos variables reales. La función calcula una sumatoria que combina distancias elevadas a la sexta potencia entre las coordenadas de entrada y una matriz fija de 25 puntos (`Aij`), que representan los centros de los valles.

La función devuelve el inverso de esta sumatoria, de manera que los puntos más cercanos a los centros de los valles tienen valores de evaluación más altos.

DeJong::F5::printFxWithVariables

Esta función permite imprimir en consola el valor de la función F5 evaluada en un punto específico, junto con los valores de las variables que componen ese punto.

```
#ifndef F5DEJONG_HPP
#define F5DEJONG_HPP

#include <cmath>
#include <iostream>
#include <vector>

namespace DeJong {

class F5 {
   public:
    static double evaluate(const std::vector<double>& x) {
        // MATRIZ Aij
        static const int A[25][2] = {{-32, -32}, {-16, -32}, {0, -32},
   {16, -32}, {32, -32},
                                     {-32, -16}, {-16, -16}, {0, -16},
   {16, -16}, {32, -16},
                                     {-32, 0},   {-16, 0},   {0, 0},
   {16, 0},   {32, 0},
                                     {-32, 16},  {-16, 16},  {0, 16},
   {16, 16},  {32, 16},
                                     {-32, 32},  {-16, 32},  {0, 32},
   {16, 32},  {32, 32}};

        // OBTENER LA SUMATORIA
        double sum = 0.0;
```

```
        for (int j = 0; j < 25; ++j) {
            double denom = j + 1;
            // (x1 - Aij) ^ 6 + (x2 - Aij) ^ 6
            denom += std::pow(x[0] - A[j][0], 6);
            denom += std::pow(x[1] - A[j][1], 6);
            // f^-1
            sum += 1.0 / denom;
        }
        // 1 / (1/5 + SUMA)
        return 1.0 / (0.002 + sum);
    }

    // IMPRESION DE LA FUNCION CON SUS VARIABLES
    static void printFxWithVariables(const std::vector<double>& x) {
        double fx = evaluate(x);
        std::cout << "f(x) = " << fx << "\tVariables: ";
        for (double v : x) {
            std::cout << v << " ";
        }
        std::cout << "\n";
    }
};

}  // namespace DeJong

#endif
```

Main

Este programa implementa un algoritmo genético que optimiza la función F5 de De Jong, conocida por su paisaje complejo con múltiples mínimos locales. La ejecución comienza solicitando al usuario parámetros clave como el tamaño de la población, tasas de cruza y mutación, número de generaciones, semilla aleatoria y nombre del archivo de salida.

El algoritmo inicializa una población de individuos representados en código Gray, los evalúa mediante la función objetivo F5, y luego entra en un ciclo evolutivo que se repite por un número definido de generaciones. En cada generación se aplica selección universal estocástica, cruza uniforme y mutación uniforme para generar una nueva población. Se implementa elitismo para preservar al mejor individuo de cada generación.

El rendimiento evolutivo se registra en archivos de salida que contienen estadísticas y los valores del mejor individuo. Al final, el mejor individuo global encontrado durante toda la evolución se muestra junto con su evaluación bajo la función F5.

```
#include <fstream>
#include <iostream>
#include <string>

#include "F5DeJong.hpp"
```

```cpp
#include "cruza.hpp"
#include "individuo.hpp"
#include "mutacion.hpp"
#include "poblacion.hpp"
#include "seleccion.hpp"

int main() {
    // PARAMETROS
    int POP_SIZE, MAX_GENERATIONS, SEED;
    double CROSS_PROB, MUT_RATE;
    std::string outputFilename;

    std::cout << "Ingrese el tama o de la poblaci n: ";
    std::cin >> POP_SIZE;
    std::cout << "Ingrese el porcentaje de cruza (0-1): ";
    std::cin >> CROSS_PROB;
    std::cout << "Ingrese el porcentaje de mutaci n (0-1): ";
    std::cin >> MUT_RATE;
    std::cout << "Ingrese el n mero m ximo de generaciones: ";
    std::cin >> MAX_GENERATIONS;
    std::cout << "Ingrese un numero entero para la semilla: ";
    std::cin >> SEED;
    std::cout << "Ingrese el nombre del archivo de salida: ";
    std::cin >> outputFilename;

    std::ofstream logFile(outputFilename);
    std::ofstream eliteFile("elite_log.csv");

    // INICIALIZACION DE LOS OPERADORES
    const int DIM = 2;
    const int BITS_PER_VAR = 17;

    Individual::setSeed(SEED);
    UniformCrossover::setSeed(SEED);
    UniformMutation::setSeed(SEED);
    Individual::setDecoding(-65.536, 65.536, DIM, BITS_PER_VAR, Individual
    ::Encoding::GRAY);

    auto crossover = CrossoverFactory::create(CrossoverFactory::Type::
    UNIFORM, CROSS_PROB, 0.5);
    auto mutator = MutationFactory::create(MutationFactory::Type::UNIFORME
    , MUT_RATE);
    Universal_Estocastica susSelector;

    // CREA LA POBLACION CON EL TIPO DE REPRESENTACION
    Population population(POP_SIZE, DIM * BITS_PER_VAR, Individual::
    Encoding::GRAY);

    for (int i = 0; i < population.size(); ++i) {
        auto ind = population.getIndividual(i);
        ind->decode();
        double fx = DeJong::F5::evaluate(ind->getVariables());
```

```
54          ind->setFitness(1.0 / (1.0 + fx));
55      }
56
57      Individual bestGlobal = *population.getFittest();
58
59      logFile << "Generacion,Media,Mejor,Peor,Cruzas,Mutaciones,Mejor_FX,
    Variables\n";
60      eliteFile << "Generacion,Fx,Var1,Var2\n";
61
62      // CICLO PRINCIPAL PARA LAS GENERACIONES
63      for (int gen = 1; gen <= MAX_GENERATIONS; ++gen) {
64          // MANTIENE AL ELITE
65          auto elite = population.getFittest();
66          Population newPopulation;
67
68          // SELECCIONA N INDIVIDUOS CON SELECCION UNIVERSAL ESTOCASTICA
69          auto selected = susSelector.selectMany(population, POP_SIZE);
70
71          int crosses = 0;
72          int mutations = 0;
73
74          // CICLO PARA REALIZAR LAS CRUZAS Y MUTACIONES
75          for (size_t i = 0; i + 1 < selected.size(); i += 2) {
76              // CRUZA UNIFORME
77              auto [h1, h2] = crossover->crossover(selected[i], selected[i +
    1]);
78              ++crosses;
79
80              // MUTACION  UNIFORME
81              mutator->mutate(h1);
82              mutations++;
83              mutator->mutate(h2);
84              mutations++;
85
86              // EVALUACION DE LOS HIJOS
87              h1->decode();
88              double fx1 = DeJong::F5::evaluate(h1->getVariables());
89              h1->setFitness(1.0 / (1.0 + fx1));
90
91              h2->decode();
92              double fx2 = DeJong::F5::evaluate(h2->getVariables());
93              h2->setFitness(1.0 / (1.0 + fx2));
94
95              // AGREGA A LOS HIJOS A LA NUEVA POBLACION
96              newPopulation.addIndividual(h1);
97              if (newPopulation.size() < POP_SIZE)
98                  newPopulation.addIndividual(h2);
99          }
100
101         // AGREGA AL ELITE EN LA POSICION DEL PEOR INDIVIDUO
102         newPopulation.Replace(elite);
103         population = newPopulation;
```

```
104
105         // ESTADISTICAS
106         double avg = population.getAverageFitness();
107         double min = population.getWorstFitness();
108         double max = population.getFittest()->getFitness();
109
110         auto bestGen = population.getFittest();
111         double fxBest = DeJong::F5::evaluate(bestGen->getVariables());
112         if (bestGen->getFitness() > bestGlobal.getFitness()) {
113             bestGlobal = *bestGen;
114         }
115
116         logFile << gen << "," << avg << "," << max << "," << min << "," <<
    crosses << ","
117                 << mutations << "," << fxBest << ",";
118         for (double v : bestGen->getVariables())
119             logFile << v << " ";
120         logFile << "\n";
121
122         eliteFile << gen << "," << fxBest << "," << bestGen->getVariables
    ()[0] << ","
123                   << bestGen->getVariables()[1] << "\n";
124     }
125
126     std::cout << "\n=== MEJOR INDIVIDUO GLOBAL ===\n";
127     bestGlobal.printIndividual();
128     std::cout << "EVALUACION F5 DEL MEJOR INDIVIDUO GLOBAL: ";
129     DeJong::F5::printFxWithVariables(bestGlobal.getVariables());
130
131     logFile.close();
132     eliteFile.close();
133     return 0;
134 }
```

Entorno de compilación y ejecución

Las pruebas del algoritmo genético fueron realizadas en una computadora portátil Apple MacBook Air con chip Apple M4 (arquitectura ARM de 64 bits), ejecutando macOS.

La compilación del programa se realizó utilizando el compilador de C++ de GNU (g++) con soporte para el estándar C++17, junto con las opciones `-Wall` y `-Wextra` para activar advertencias útiles durante el desarrollo como se muestra en las figuras 2 y 3. La línea de compilación utilizada fue:

```
1 g++ -std=c++17 -Wall -Wextra individuo.cpp poblacion.cpp seleccion.cpp
    cruza.cpp mutacion.cpp main_gray.cpp -o algoritmo
```

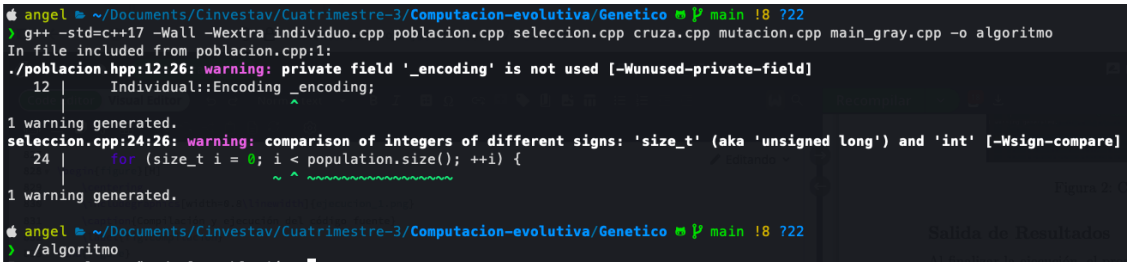

Figura 2: Compilación y ejecución del código fuente

Y para representación entera:

```
g++ -std=c++17 -Wall -Wextra individuo.cpp poblacion.cpp seleccion.cpp
    cruza.cpp mutacion.cpp main_entero.cpp -o algoritmo
```

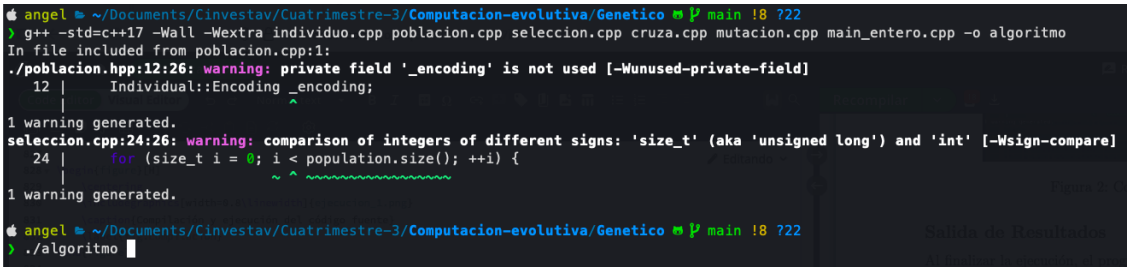

Figura 3: Compilación y ejecución del código fuente

Y la ejecución del programa se realizó mediante el comando:

`./algoritmo`

Y por ultimo ejemplos de corridas de las dos representaciones.

Representación Gray

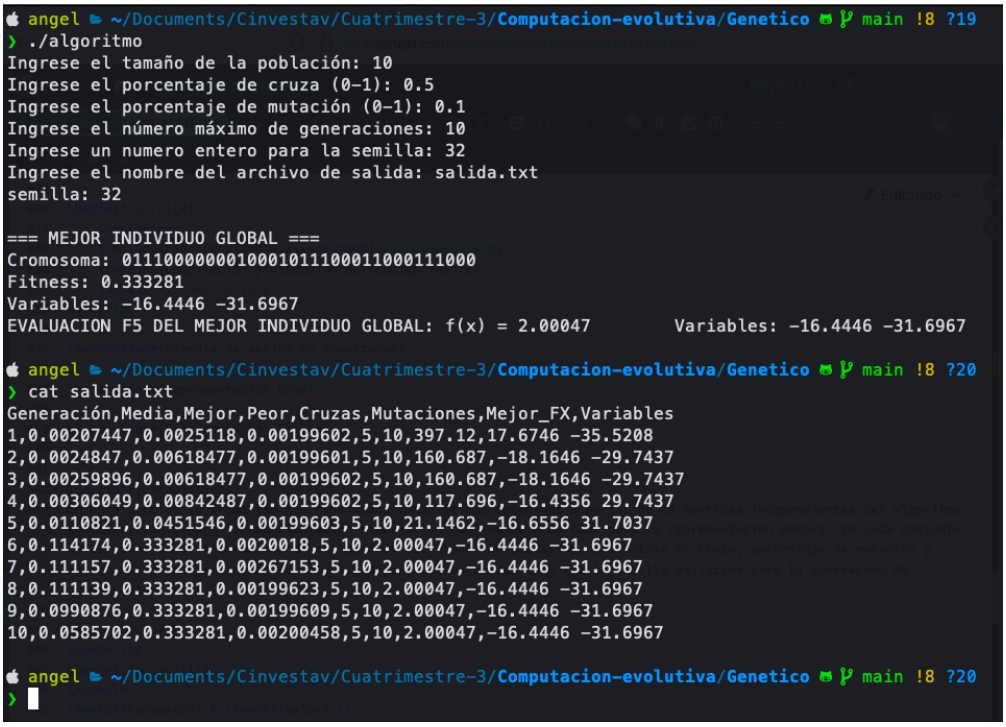

Figura 4: Salida con representación Gray

Representación Entera

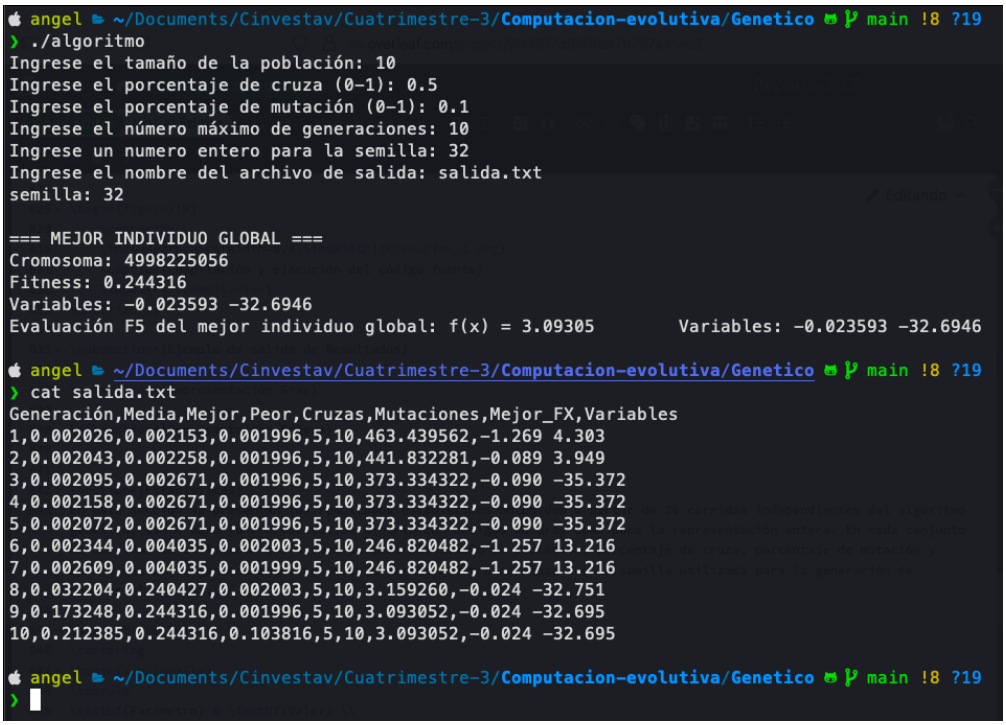

Figura 5: Salida con representación Entera

Estadísticas

En esta sección se presentan los resultados estadísticos obtenidos a partir de 20 corridas independientes del algoritmo genético, tanto para la representación binaria con códigos de Gray como para la representación entera. En cada conjunto de corridas se utilizaron los mismos parámetros: tamaño de población, porcentaje de cruza, porcentaje de mutación y número máximo de generaciones. La única diferencia entre corridas fue la semilla utilizada para la generación de números aleatorios.

Parámetro	**Valor**
Total population size	10
Chromosome length	17
Maximum number of generations	100
Crossover probability	0.500000
Mutation probability	0.10000

A continuación, se analizan las estadísticas obtenidas para ambas representaciones, permitiendo comparar el desempeño global del algoritmo en cada caso.

Representación Gray

Los resultados estadísticos obtenidos con la representación Gray muestran un comportamiento altamente consistente. En las 20 corridas realizadas, el algoritmo genético encontró en todos los casos la solución óptima, obteniendo un valor de $f(x) = 0{,}998004$ en cada corrida. Esto se refleja en una media de aptitud de 0.500497, acompañada de una varianza y desviación estándar prácticamente nulas.

Estos valores indican una excelente estabilidad del algoritmo bajo esta representación, logrando converger rápidamente hacia el óptimo global en todas las ejecuciones. Esta regularidad también sugiere que los operadores genéticos utilizados (cruza uniforme, mutación bit a bit) son muy adecuados para trabajar con esta codificación.

Cuadro 1: Resultados de las 20 corridas del algoritmo genético con variables, fitness y evaluación $f(x)$

Corrida	**Semilla**	x_1	x_2	**Fitness**	$f(x)$
1	648	-31.9917	-31.9807	0.5005	0.998004
2	915	-31.9597	-31.9357	0.5005	0.998004
3	18	-31.9667	-31.9707	0.5005	0.998004
4	935	-31.9697	-32.0417	0.5005	0.998004
5	596	-31.9937	-31.9737	0.5005	0.998004
6	224	-32.0217	-32.0017	0.5005	0.998004
7	773	-31.9687	-31.9787	0.5005	0.998004
8	178	-31.9997	-31.9797	0.5005	0.998004
9	796	-31.9647	-32.0067	0.5005	0.998004
10	543	-32.1937	-31.7667	0.500446	0.998217
11	419	-32.0407	-31.9867	0.5005	0.998004
12	891	-31.9747	-31.9787	0.5005	0.998004
13	159	-31.9807	-32.0337	0.5005	0.998004
14	369	-32.0657	-31.9337	0.500499	0.998004
15	518	-31.9807	-31.9767	0.5005	0.998004
16	989	-31.9617	-31.9787	0.5005	0.998004
17	230	-31.9827	-31.9847	0.5005	0.998004
18	345	-31.9917	-31.9817	0.5005	0.998004
19	325	-32.0117	-32.0067	0.5005	0.998004
20	149	-31.9677	-31.9807	0.5005	0.998004

Estadísticas de los valores de Fitness y $f(x)$:
Media Fitness (μ): 0.500497
Varianza Fitness (σ^2): $1{,}46 \times 10^{-10}$
Desviación estándar Fitness (σ): $1{,}21 \times 10^{-5}$
Media $f(x)$: 0.998015
Varianza $f(x)$: $2{,}27 \times 10^{-9}$
Desviación estándar $f(x)$: $4{,}76 \times 10^{-5}$
Mejor valor $f(x)$: 0.998004 Peor valor $f(x)$: 0.998217

Representación Entera

En contraste, la representación entera mostró una mayor variabilidad en los resultados. Aunque algunas corridas lograron alcanzar la solución óptima, en otros casos el valor de $f(x)$ fue significativamente mayor (peor), con algunos resultados por encima de 10 o incluso cercanos a 15. La media de aptitud fue considerablemente menor que en la representación Gray, con un valor de 0.282329.

Además, tanto la varianza (0.027328) como la desviación estándar (0.165278) fueron notablemente más altas, lo que evidencia una menor consistencia y mayor dispersión en los

resultados. Esto puede atribuirse a la naturaleza de la mutación en esta representación, donde se reemplaza un dígito entero aleatoriamente entre 0 y 9, lo cual puede causar cambios bruscos en la codificación y dificultar la exploración fina del espacio de soluciones, especialmente cerca del óptimo.

Cuadro 2: Resultados de las 20 corridas con representación real

Corrida	**Semilla**	x_1	x_2	**Fitness**	$f(x)$
1	467	-16.3997	-31.9724	0.333770	1.996080
2	134	-15.9921	-31.9724	0.334221	1.992030
3	580	-32.2044	15.8322	0.060592	15.503900
4	895	-1.1954	-15.5805	0.085547	10.689500
5	262	-31.9527	-31.9645	0.500500	0.998004
6	626	-31.9685	-31.9842	0.500500	0.998004
7	418	-16.0078	-15.9751	0.126529	6.903340
8	81	-15.9869	-31.9842	0.334221	1.992030
9	534	-31.4573	-15.9921	0.143806	5.953790
10	200	-31.9147	-32.2581	0.500426	0.998299
11	801	0.0079	-31.9868	0.251123	2.982110
12	613	-31.9685	-31.9855	0.500500	0.998004
13	70	15.7247	-32.0720	0.201261	3.968680
14	409	15.9252	-31.8885	0.201278	3.968250
15	597	-32.2070	-31.9881	0.500480	0.998082
16	782	-31.9881	-31.9475	0.500500	0.998004
17	716	-31.9540	-15.9646	0.144324	5.928850
18	826	-31.9789	-31.9422	0.500500	0.998004
19	197	15.9633	0.0262	0.068406	13.618600
20	547	-31.9960	-32.1153	0.500499	0.998006

Estadísticas de los valores de Fitness y $f(x)$:
Media Fitness (μ): 0.314449
Varianza Fitness (σ^2): 0.028921
Desviación estándar Fitness (σ): 0.170062
Media $f(x)$: 4.174078
Varianza $f(x)$: 18.508946
Desviación estándar $f(x)$: 4.302202
Mejor valor $f(x)$: 0.998004
Peor valor $f(x)$: 15.503900

Comparación entre representaciones

Una comparación de resultados entre las dos implementaciones muestra diferencias claras en el desempeño del algoritmo genético según la representación utilizada. Basándonos en los resultados promediados de las 20 corridas efectuadas para cada representación, se observa

que la codificación con códigos de Gray presenta un rendimiento superior. En todas las corridas realizadas con esta representación, el algoritmo logra alcanzar el óptimo, lo que se refleja en una media de aptitud de 0.500497 y en valores de varianza y desviación estándar prácticamente nulos.

Por otro lado, la representación entera muestra un desempeño más irregular. Aunque en algunas corridas también se alcanza el óptimo, en varias otras el algoritmo converge a soluciones subóptimas, con valores de $f(x)$ superiores a 15.0. Las estadísticas generales de este grupo muestran una media de aptitud inferior, y una varianza y desviación estándar significativamente mayores en comparación con la codificación Gray, lo cual evidencia una menor estabilidad en la calidad de las soluciones obtenidas.

Estas diferencias pueden explicarse en gran medida por la forma en que opera la mutación en cada representación. En la codificación con códigos de Gray, una mutación cambia un único bit del cromosoma, lo que suele implicar un cambio relativamente pequeño en los valores decodificados del individuo. Esto favorece una exploración más suave del espacio de soluciones, permitiendo al algoritmo ajustarse finamente hacia el óptimo. En contraste, en la representación entera, la mutación reemplaza un dígito del cromosoma con un entero aleatorio entre 0 y 9. Esta operación puede generar cambios bruscos en los valores decodificados, lo que en ocasiones aleja al individuo de una buena solución y dificulta la convergencia estable hacia el óptimo.

Gráficas de convergencia

A continuación, se presentan las gráficas de convergencia correspondientes a los resultados ubicados en la mediana de ambas representaciones.

La figura 6 muestra la convergencia hacia el óptimo utilizando la representación Gray. Se puede observar que el algoritmo alcanza la aptitud óptima aproximadamente en la generación 10. A partir de ese punto, la aptitud se mantiene constante, lo cual indica que el algoritmo ha convergido a una solución óptima.

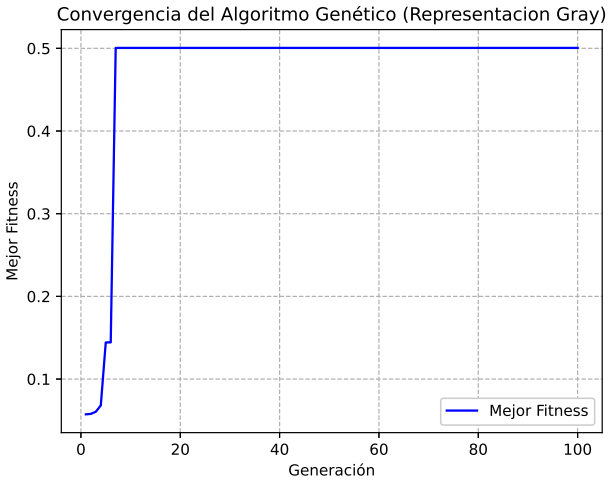

Figura 6: Gráfica de convergencia con representación Gray

En contraste, la figura 7, correspondiente a la representación entera, muestra dos fases claras de incremento en la aptitud. La primera ocurre casi de inmediato, alrededor de la generación 2, donde se observa un aumento significativo. Posteriormente, se presentan pequeñas variaciones en la aptitud, hasta que en la generación 41 se produce un segundo incremento, tras el cual la aptitud se estabiliza en las generaciones siguientes.

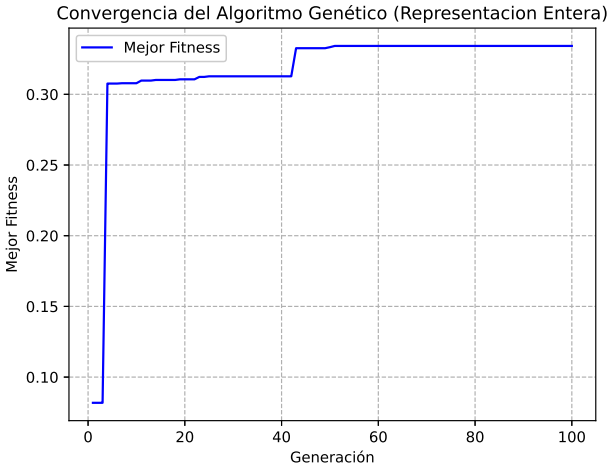

Figura 7: Gráfica de convergencia con representación Entera

Como se observa en ambas figuras, la representación con códigos de Gray alcanza una convergencia más rápida y una aptitud más alta (0.5005), mientras que la representación entera logra una aptitud máxima de 0.334221 a lo largo de las 100 generaciones.

Lectura

Cuando los autores se refieren a la evidencia que informa un cambio, ¿cuál es la diferencia entre evidencia absoluta y evidencia relativa? Explique con sus propias palabras. La evidencia absoluta refiere a monitorear un valor y si supera un umbral, que accione un evento para aplicar una regla a un valor de la estrategia de parámetros con el objetivo de ser alterada. El monitoreo sobre un valor para verificar si supera un umbral se le conoce como evidencia.

La evidencia relativa se basa en compara la aptitud de la nueva población, donde se utiliza varios valores para determinar el cambio de un valor o varios valores y pueda determinar si esos valores se siguen utilizando en las siguientes generaciones.

¿Cuál es la diferencia entre ajuste de parámetros y control de parámetros? Explique con sus propias palabras. El ajuste de parámetros es la técnica que se utiliza normalmente en los algoritmos genéticos, el cual es la búsqueda de parámetros elegibles para el algoritmo genético y dichos parámetros no cambian en la ejecución del algoritmo genético.

En cambio con control de parámetros, es una técnica alternativa en la que se parte de unos parámetros definidos al inicio y conforme avanza la ejecución del algoritmo genético, los parámetros cambian de valor.

¿Qué desventajas tiene el ajuste de parámetros? Explique con sus propias palabras. Las principales desventajas del ajuste de parámetros es debido al espacio de búsqueda que se tiene, explorar ese espacio de búsqueda para encontrar una combinación optima que permita mejorar el desempeño de un algoritmo genético no es una tarea sencilla, haciendo una tarea que consuma tiempo y recursos computacionales para la exploración en el espacio de búsqueda.

Según los autores, ¿qué cambio de actitud se ha dado en la comunidad de computación evolutiva respecto al papel que juegan los parámetros de un algoritmo evolutivo en su desempeño en un problema dado? Explique. Los autores comentan que la comunidad de computación evolutiva considera el ajuste de los parámetros de un algoritmo evolutivo para un problema es necesario en menor o mayor medida [1].

Referencias

[1] A. E. Eiben et al. “Parameter Control in Evolutionary Algorithms”. En: *Parameter Setting in Evolutionary Algorithms*. Ed. por Fernando G. Lobo, Cláudio F. Lima y Zbigniew Michalewicz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, págs. 19-46. ISBN: 978-3-540-69432-8. DOI: 10.1007/978-3-540-69432-8_2. URL: https://doi.org/10.1007/978-3-540-69432-8_2.