Introducción a la Computación Evolutiva
Tarea No. 4

Instructor: Dr. Carlos Artemio Coello Coello

Alumno: Ángel Alonso Galarza Chávez

1 de julio de 2025

Introducción

En este trabajo se implementó un algoritmo genético para resolver instancias de la *Asignación Cuadrática*, un problema de optimización combinatoria. La estructura del algoritmo incluye operadores de cruza, mutación e inversión adaptados a representaciones enteras basadas en permutaciones.

Como método de selección se utilizó el torneo binario determinístico, que garantiza la supervivencia de individuos con mejor aptitud en cada generación. Además, se implementó el operador de cruza Order Based Crossover y como operador de mutación la implementación de Mutación por Inversión.
La función objetivo se calcula mediante la siguiente ecuación:

$$\text{costo} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} dist[i][j] \times flujo[x[i]][x[j]]$$

donde $dist$ representa la matriz de distancias, $flujo$ la matriz de flujos y x la permutación que describe la asignación de instalaciones a ubicaciones.

Código

En esta sección se presenta el código fuente correspondiente a la implementación de los metodos utilizados en el algoritmo genético. Cada componente del algoritmo desde la repre-

sentación de los individuos, evaluación (función objetivo), selección, cruza y mutación está descrito mediante funciones en lenguaje C++.

Individuo

Cuando se utiliza el caso `Encoding::INTEGER_DECIMAL` dentro del constructor de la clase `Individual`, se inicializa el cromosoma como una secuencia de enteros que representa una permutación. Para ello, primero se ajusta el tamaño del vector interno `_digitChromosome` al valor de `length` dependiendo del problema.

A continuación, se llenan las posiciones del cromosoma con los valores consecutivos que van desde cero hasta $n - 1$, formando inicialmente una secuencia ordenada. Este paso garantiza que se tengan todos los elementos necesarios para una representación de permutación completa.

Finalmente, la secuencia entera se revuelve aleatoriamente utilizando `std::shuffle` junto con el generador de números global `_globalGen`. Este barajado inicial asegura que cada individuo creado tenga una permutación válida y diferente, respetando la unicidad de los elementos sin repeticiones.

```cpp
// 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
        case Encoding::INTEGER_DECIMAL: {
            _digitChromosome.resize(length);
            // INSETAR VALORES AL CROMOSOMA [0, ..., n - 1]
            for (int i = 0; i < length; ++i) {
                _digitChromosome[i] = i;
            }
            // REVOLVER CROMOSOMA
            std::shuffle(_digitChromosome.begin(), _digitChromosome.end(),
    _globalGen);
            break;
        }
```

```
    }

    decode();
}
```

Selección

Este método implementa la **selección por torneo binario determinístico**. Para cada individuo que se desea seleccionar, el algoritmo organiza competencias entre pares de individuos de la población y selecciona siempre al mejor de cada par según su valor de aptitud (*fitness*).

El procedimiento comienza verificando que la población tenga al menos dos individuos; de lo contrario, lanza una excepción para evitar torneos inválidos. Luego, inicializa una lista vacía que almacenará los individuos ganadores del torneo hasta alcanzar la cantidad deseada de seleccionados.

Se copia la población original en un vector auxiliar y se revuelve aleatoriamente mediante `std::shuffle` utilizando el generador de números `_gen`.

Dentro de cada ronda, se recorre la población mezclada de dos en dos. Para cada par de individuos, se comparan sus valores de aptitud y se selecciona determinísticamente el mejor. Este individuo ganador se agrega a la lista de seleccionados. El ciclo se repite hasta completar el número requerido de individuos.

Finalmente, la función retorna la lista de individuos elegidos como resultado del torneo binario determinístico.

```
// TORNEO DETERMINISTICO
std::vector<std::shared_ptr<Individual>> Torneo_Binario_Deterministico::
    selectMany(
    const Population& population, int numToSelect) const {
    if (population.size() < 2) {
        throw std::runtime_error("ERROR DE TAMANIO DE POBLACION.");
    }

    // LISTA DE LOS INDIVIDUOS A SELECCIONAR
    std::vector<std::shared_ptr<Individual>> selected;

    // CICLO DE SELECCIONAR N INDIVIDUOS
    while (static_cast<int>(selected.size()) < numToSelect) {
        // OBTENER LA POBLACION Y REVOLVERLO CON SHUFFLE
        std::vector<std::shared_ptr<Individual>> shuffled = population.
    getIndividuals();
        std::shuffle(shuffled.begin(), shuffled.end(), _gen);

```

```
        // BUCLE PARA SELECCIONAR EN PARES A LOS INDIVIDUOS
        for (size_t i = 0;
             i + 1 < shuffled.size() && static_cast<int>(selected.size())
    < numToSelect; i += 2) {
            // OBTENER DOS INDIVIDUOS
            auto& ind1 = shuffled[i];
            auto& ind2 = shuffled[i + 1];

            // COMPARAR INDIVIDUOS
            auto winner = (ind1->getFitness() >= ind2->getFitness()) ?
    ind1 : ind2;
            // AGREGAR GANADOR
            selected.push_back(winner);
        }
    }

    // RETORNAR LISTA DE SELECCIONADOS
    return selected;
}
```

Order-based Crossover

El método implementa el operador **Order-Based Crossover** para cromosomas representados como permutaciones enteras. Primero, se verifica que ambos padres tengan cromosomas de igual longitud; si no es así, se lanza una excepción. También se asegura que la codificación sea de tipo `INTEGER_DECIMAL`, ya que este operador solo aplica a representaciones enteras.

Se obtienen los cromosomas de ambos padres, `P1` y `P2`, y se determinan las estructuras necesarias para generar los hijos: vectores para los cromosomas de salida inicializados con `-1`, y conjuntos auxiliares para registrar los genes seleccionados de cada padre.

A continuación, para cada hijo se seleccionan genes del primer padre con una probabilidad definida por `_crossoverGenProb`. Estos valores se almacenan en un vector y en un conjunto para evitar duplicados durante el armado del cromosoma final.

Luego, se recorre el cromosoma del segundo padre para cada posición. Si un gen no ha sido seleccionado previamente, se inserta directamente en la misma posición del hijo. De esta forma, se garantiza que el hijo mantenga parte del orden del segundo padre.

Una vez colocados los genes, los valores seleccionados faltantes se insertan de izquierda a derecha en los huecos disponibles. Esto completa el cromosoma del hijo asegurando que cada gen aparezca exactamente una vez, respetando la naturaleza de la permutación.

El mismo procedimiento se repite de forma inversa para el segundo hijo: se seleccionan genes del segundo padre con probabilidad y se complementa con el primer padre siguiendo la misma lógica de inserción y preservación del orden.

Finalmente, se construyen los nuevos individuos usando los cromosomas generados y retorna ambos hijos generados por el operador `OrderBasedCrossover`.

```
// ORDER BASED CROSSOVER
std::pair<std::shared_ptr<Individual>, std::shared_ptr<Individual>>
    OrderBasedCrossover::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 LAS LONGITUDES DE LOS
    CROMOSOMAS DE LOS PADRES");
     }

     Individual::Encoding enc = parent1->getEncoding();
     if (enc != Individual::Encoding::INTEGER_DECIMAL) {
         throw std::runtime_error("OrderBasedCrossover SOLO FUNCIONA CON
    REPRESENTACION ENTERA");
     }

     // OBTENCION DE LOS CROMOSOMAS DE LOS PADRES
     const auto& P1 = parent1->getDigitChromosome();
     const auto& P2 = parent2->getDigitChromosome();
     size_t n = P1.size();

     std::uniform_real_distribution<double> distReal(0.0, 1.0);

     // CREA EL VECTOR PARA LOS HIJOS
     std::vector<int> child1(n, -1);
     std::vector<int> child2(n, -1);

     // ESTRUCTURAS PARA SELECCIONAR LOS VALORES
     std::vector<int> seleccionados1;
     std::unordered_set<int> sel1;

     std::vector<int> seleccionados2;
     std::unordered_set<int> sel2;

     // SELECCIONA LOS VALORES CON PROBABILIDAD
     for (size_t i = 0; i < n; ++i) {
         if (distReal(_globalGen) < _crossoverGenProb) {
             seleccionados1.push_back(P1[i]);
             sel1.insert(P1[i]);
         }
     }

     // SI EL VALOR DE LA POSICION i EN EL CROMOSOMA DE P2 NO ESTA EN LOS
    VALORES SELECCIONADOS
     // AGREGAR VALOR AL HIJO
     for (size_t i = 0; i < n; ++i) {
         if (sel1.find(P2[i]) == sel1.end()) {
             child1[i] = P2[i];
         }
```

```
    }

    // INSERTAR LOS VALORES FALTANTES AL CROMOSOMA DE IZQUIERDA A DERECHA
    size_t k1 = 0;
    for (size_t i = 0; i < n && k1 < seleccionados1.size(); ++i) {
        if (child1[i] == -1) {
            child1[i] = seleccionados1[k1++];
        }
    }

    // SELECCIONA LOS VALORES CON PROBABILIDAD
    for (size_t i = 0; i < n; ++i) {
        if (distReal(_globalGen) < _crossoverGenProb) {
            seleccionados2.push_back(P2[i]);
            sel2.insert(P2[i]);
        }
    }
    // SI EL VALOR DE LA POSICION i EN EL CROMOSOMA DE P1 NO ESTA EN LOS
   VALORES SELECCIONADOS
    // AGREGAR VALOR AL HIJO
    for (size_t i = 0; i < n; ++i) {
        if (sel2.find(P1[i]) == sel2.end()) {
            child2[i] = P1[i];
        }
    }

    // INSERTAR LOS VALORES FALTANTES AL CROMOSOMA DE IZQUIERDA A DERECHA
    size_t k2 = 0;
    for (size_t i = 0; i < n && k2 < seleccionados2.size(); ++i) {
        if (child2[i] == -1) {
            child2[i] = seleccionados2[k2++];
        }
    }

    // CREA LOS HIJOS CON LOS CROMOSOMAS OBTENIDOS
    auto offspring1 = std::make_shared<Individual>(child1, enc);
    auto offspring2 = std::make_shared<Individual>(child2, enc);

    // EMPAREJA A LOS PADRES CON HIJOS
    offspring1->setParents(parent1, parent2);
    offspring2->setParents(parent1, parent2);

    // DEVUELVE A LOS DOS HIJOS
    return {offspring1, offspring2};
}
```

Mutación por Inserción

El método implementa la **mutación por inserción** para individuos con codificación de tipo permutación entera. Para determinar si se aplica la mutación, primero se genera un número aleatorio en el rango [0, 1] y se compara con la tasa de mutación `_mutationRate`. Si el valor

aleatorio es mayor o igual, no se realiza ninguna alteración en el cromosoma.

A continuación, se verifica que la codificación del individuo sea de tipo `INTEGER_DECIMAL`, ya que este tipo de mutación solo tiene sentido cuando se trabaja con cromosomas representados como secuencias de enteros sin repetición. Si la condición no se cumple, se lanza una excepción para detener el proceso.

Se obtiene el cromosoma entero del individuo y se evalúa su longitud. Si la longitud es menor a dos, no se ejecuta la mutación ya que no tiene sentido insertar un elemento en otra posición cuando solo existe un único gen.

Para realizar la mutación, se generan dos posiciones aleatorias dentro de los límites del cromosoma: `pos_valor` indica la posición del valor que se desea mover, mientras que `pos_insercion` indica la nueva posición de inserción. Si ambas posiciones resultan iguales, se regenera `pos_insercion` hasta que difiera de la posición original.

Se obtiene el valor ubicado en `pos_valor`, se elimina de su posición original y se inserta en la nueva posición `pos_insercion`. Este desplazamiento reorganiza el cromosoma manteniendo la validez de la permutación.

Finalmente, se actualiza el cromosoma del individuo con la nueva permutación mediante `setDigitChromosome`.

```
1  // MUTACION POR INSERCION
2  void InsertionMutation::mutate(std::shared_ptr<Individual> individual)
       const {
3      std::uniform_real_distribution<double> probDist(0.0, 1.0);
4
5      // SI MUTATIONRATE < PROBDIST -> NO MUTAR
6      if (probDist(_globalGen) >= _mutationRate) {
7          return;
8      }
9
10     // OBTENER CODIFICACION DEL INDIVIDUO
11     auto encoding = individual->getEncoding();
12     // VERIFICAR SI LA CODIFICACION ES ENTERA
13     if (encoding != Individual::Encoding::INTEGER_DECIMAL) {
14         throw std::runtime_error("InsertionMutation SOLO ES VALIDA PARA
       REPRESENTACION ENTERA");
15     }
16
17     // OBTENER CROMOSOMA DEL INDIVIDUO
18     auto chromo = individual->getDigitChromosome();
19     // LONGUITUD DEL CROMOSOMA
20     int n = static_cast<int>(chromo.size());
21
22     // SI SOLO HAY 1 CROMOSOMA, NO HACER NADA
23     if (n < 2)
```

```
24          return;
25
26      std::uniform_int_distribution<int> distPos(0, n - 1);
27
28      // GENERAR POSICION ALEATORIA
29      int pos_valor = distPos(_globalGen);
30      int pos_insercion = distPos(_globalGen);
31
32      // SI LA POSICION DE INSERCION ES LA MISMA QUE POSICION DEL VALOR,
    VOLVER A GENERAR UN NUMERO
33      // ALEATORIO
34      while (pos_insercion == pos_valor) {
35          pos_insercion = distPos(_globalGen);
36      }
37
38      // OBTENER EL VALOR DE LA POSICION ALEATORIA
39      int val = chromo[pos_valor];
40
41      // ELIMINA E INSERTA LA POSICION EL VALOR SELECCIONADO
42      // DESPLAZA LOS VALORES POSTERIORES DE LA POSICION DE INSERCION
43      chromo.erase(chromo.begin() + pos_valor);
44      chromo.insert(chromo.begin() + pos_insercion, val);
45
46      individual->setDigitChromosome(chromo);
47      individual->decode();
48  }
```

Utils

chromosomeToString Esta función convierte un cromosoma representado como un vector de enteros en una cadena de caracteres. Para ello, recorre cada valor del cromosoma y lo concatena con un guion como separador. El resultado es una representación única y legible que puede usarse como clave para verificar duplicados en la población.

removeDuplicates Esta función recorre toda la población y se asegura de que no existan cromosomas duplicados. Para cada individuo, convierte su cromosoma a cadena usando `chromosomeToString` y verifica si ya existe en un conjunto de representaciones vistas. Si encuentra duplicados, aplica el operador de mutación pasado como parámetro hasta obtener un cromosoma único, garantizando así diversidad dentro de la población.

invertSubsequence Esta función invierte una subsecuencia de genes dentro de un cromosoma entero delimitada por dos valores dados. Primero, localiza las posiciones de ambos valores dentro del vector. Si los delimitadores están invertidos en el orden, los intercambia para asegurar la correcta inversión. Luego, aplica la función estándar `std::reverse` para invertir todos los elementos entre ambos valores.

applyInversion Esta función aplica una mutación de tipo inversión a un individuo con una probabilidad determinada. Si la probabilidad se cumple, invierte una subsecuencia del cromosoma usando `invertSubsequence`. Dependiendo de la política de inversión seleccionada (`ALWAYS_REPLACE`, `ONLY_IF_BETTER` o `RANDOM_CHOICE`), decide si reemplazar el cromosoma original por la versión invertida.

```
#include "utils.hpp"

#include <sstream>
#include <string>
#include <unordered_set>
#include <vector>

#include "mutacion.hpp"

namespace Utils {

// CROMOSOMA A CADENA DE CARACTERES
std::string chromosomeToString(const std::vector<int>& chromo) {
    std::ostringstream oss;
    for (int val : chromo) {
        oss << val << "-";
    }
    return oss.str();
}

// REMOVER DUPLICADOS
void removeDuplicates(std::vector<std::shared_ptr<Individual>>& population
    ,
                      IMutationOperator& mutator) {
    std::unordered_set<std::string> seen;

    // RECORRER LA POBLACION POR INDIVIDUO
    for (auto& individual : population) {
        // OBTENER CROMOSOMA
        std::string repr = chromosomeToString(individual->
    getDigitChromosome());

        // APLICAR MUTACION SI ENCUENTRA UN INDIVIDUO CON LA MISMA
    PERMUTACION
        while (seen.find(repr) != seen.end()) {
            mutator.mutate(individual);
            repr = chromosomeToString(individual->getDigitChromosome());
        }

        // INSERTA EL NUEVO INDIVIDUO
        seen.insert(repr);
    }
}

// INVERTIR SUBSECUENCIA
```

```cpp
void invertSubsequence(std::vector<int>& chromosome, int value1, int value2) {
    // SE INSTANCIA LOS VALORES PARA DELIMITAR LA SUBSECUENCIA
    auto it1 = std::find(chromosome.begin(), chromosome.end(), value1);
    auto it2 = std::find(chromosome.begin(), chromosome.end(), value2);

    if (it1 == chromosome.end() || it2 == chromosome.end()) {
        throw std::invalid_argument("ERROR DE VALORES");
    }

    // INTERCAMBIA LOS VALORES DE LOS DELIMITADORES SI T1 > T2
    if (it1 > it2)
        std::swap(it1, it2);

    // INVIERTA LA SUBSECUENCIA
    std::reverse(it1, it2 + 1);
}

// INVERSION
void applyInversion(std::shared_ptr<Individual> individual, double inversionRate, int value1,
                    int value2, InversionPolicy policy) {
    static std::mt19937 gen(std::random_device{}());
    std::uniform_real_distribution<double> probDist(0.0, 1.0);
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    // SI LA PROBABILIDAD ES MAYOR A LA INVERSION, NO HACE NADA
    if (probDist(gen) >= inversionRate) {
        return;
    }

    // GUARDA EL CROMOSOMA ORIGINAL
    auto originalChromo = individual->getDigitChromosome();

    // APLICA INVERSION
    auto newChromo = originalChromo;
    invertSubsequence(newChromo, value1, value2);

    // PARA LA POLITICA DE ONLY_IF_BETTER SE REQUIERE EVALUAR AL INDIVIDUO Y AL INDIVIDUO CON
    // INVERSION
    Individual temp(newChromo, individual->getEncoding());
    temp.decode();
    double originalFitness = individual->getFitness();
    double newFitness = 0.0;

    // SWITCH DE LAS POLITICAS DE INVERSION
    switch (policy) {
        // POLITICA: SIEMPRE REMPLAZO
        case InversionPolicy::ALWAYS_REPLACE:
            individual->setDigitChromosome(newChromo);
            individual->setFitness(newFitness);
```

```
92              break;
93
94              // POLITICA: REMPLAZAR SI ES MEJOR
95          case InversionPolicy::ONLY_IF_BETTER:
96              if (newFitness > originalFitness) {
97                  individual->setDigitChromosome(newChromo);
98                  individual->setFitness(newFitness);
99              }
100             break;
101             // POLITICA: ALEATORIA
102         case InversionPolicy::RANDOM_CHOICE:
103             if (dist(gen) < 0.5) {
104                 individual->setDigitChromosome(newChromo);
105                 individual->setFitness(newFitness);
106             }
107             break;
108     }
109 }
110 }
```

Main

Este programa implementa un algoritmo genético para resolver instancias de la *Asignación Cuadrática*, utilizando matrices de distancia y flujo leídas desde un archivo de entrada. La ejecución inicia solicitando al usuario parámetros clave como el nombre del archivo de datos, el tamaño de la población, tasas de cruza, mutación e inversión, el número máximo de generaciones, una semilla aleatoria para la generación de números pseudoaleatorios y el nombre del archivo de salida.

El algoritmo lee las matrices de distancia y flujo, que definen el problema de asignación cuadrática, y calcula la función objetivo como el costo total de una permutación dada. Las permutaciones de los individuos se leen de izquierda a derecha que representa las soluciones.

Durante la inicialización, se genera una población de soluciones aleatorias y se evalúa su aptitud utilizando la función de evaluación. Se eliminan individuos duplicados mediante un proceso de verificación y mutación para asegurar diversidad. Se implementa un operador de cruza de tipo `Order-Based Crossover`, una mutación por inserción y una operación de inversión de subsecuencias, todos configurados con tasas definidas por el usuario.

El ciclo evolutivo se repite por el número máximo de generaciones definido. En cada generación, se selecciona la nueva población mediante un torneo binario determinístico, se cruzan pares de padres y se generan hijos que se someten a mutación e inversión. Los hijos son evaluados y la población se actualiza reemplazando al peor individuo por el mejor de la generación anterior para aplicar elitismo. Se registran estadísticas de cada generación, como la media, el mejor y peor valor de aptitud, número de cruzas, mutaciones e inversiones, así como la permutación y su costo.

Los resultados del algoritmo genético se almacenan en archivos de salida: uno general con estadísticas de la evolución y otro exclusivo para el registro de los individuos élite. Al finalizar la evolución, se muestra en pantalla el mejor individuo global encontrado junto con su costo asociado.

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

#include "cruza.hpp"
#include "individuo.hpp"
#include "mutacion.hpp"
#include "poblacion.hpp"
#include "seleccion.hpp"
#include "utils.hpp"

// LEER ARCHIVO
void readFile(const std::string& filename, int& n, std::vector<std::vector
    <int>>& dist,
              std::vector<std::vector<int>>& flujo) {
    std::ifstream file(filename);
    if (!file)
        throw std::runtime_error("NO SE PUDO ABRIR EL ARCHIVO: " +
    filename);

    file >> n;
    dist.assign(n, std::vector<int>(n));
    flujo.assign(n, std::vector<int>(n));

    // MATRIZ DE DISTANCIA
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            file >> dist[i][j];

    // MATRIZ DE FLUJO
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            file >> flujo[i][j];
}

// FUNCION DE EVALUACION
double evaluate(const std::vector<int>& perm, const std::vector<std::
    vector<int>>& dist,
                const std::vector<std::vector<int>>& flujo) {
    int n = perm.size();
    double cost = 0.0;

    // SUMATORIA
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
```

```
            cost += dist[i][j] * flujo[perm[i]][perm[j]];
    return cost;
}

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

    std::cout << "INGRESE EL NOMBRE DEL ARCHIVO (ejemplo.dat): ";
    std::cin >> filename;
    std::cout << "INGRESE EL TAMANIO DE LA POBLACION: ";
    std::cin >> POP_SIZE;
    std::cout << "INGRESE EL PORCENTAJE DE CRUZA (0-1): ";
    std::cin >> CROSS_PROB;
    std::cout << "INGRESE EL PORCENTAJE DE MUTACION (0-1): ";
    std::cin >> MUT_RATE;
    std::cout << "INGRESE EL PORCENTAJE DE INVERSION (0-1): ";
    std::cin >> INVERSION_RATE;
    std::cout << "INGRESE EL NUMERO MAXIMO DE GENERACIONES: ";
    std::cin >> MAX_GENERATIONS;
    std::cout << "INGRESE UN NUMERO ENTERO PARA LA SEMILLA: ";
    std::cin >> SEED;
    std::cout << "INGRESE NOMBRE DEL ARCHIVO DE SALIDA: ";
    std::cin >> outputFilename;

    std::cout << "SEMILLA: " << SEED << "\n";

    int n;
    std::vector<std::vector<int>> dist, flujo;

    // ARCHIVO PARA GUARDAR EL PROGRESO
    readFile(filename, n, dist, flujo);

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

    // INICIALIZACION DEL ALGORITMO GENETICO
    Individual::setSeed(SEED);
    OrderBasedCrossover::setSeed(SEED);
    InsertionMutation::setSeed(SEED);

    OrderBasedCrossover obx(CROSS_PROB, 0.5);
    auto mutator = MutationFactory::create(MutationFactory::Type::
   INSERCION, MUT_RATE);
    Torneo_Binario_Deterministico selector;

    // INICIALIZACION DE LA POBLACION
    Population population(POP_SIZE, n, Individual::Encoding::
   INTEGER_DECIMAL);
    for (int i = 0; i < population.size(); ++i) {
```

```
93          auto ind = population.getIndividual(i);
94          double fx = evaluate(ind->getDigitChromosome(), dist, flujo);
95          ind->setFitness(1.0 / (1.0 + fx));
96      }
97
98      // REMOVER INDIVIDUOS DUPLICADOS
99      Utils::removeDuplicates(population.getIndividuals(), *mutator);
100
101     // MEJOR INDIVIDUO
102     Individual bestGlobal = *population.getFittest();
103
104     // COLUMNAS DE LOS ARCHIVOS
105     logFile << "Generaci n,Media,Mejor,Peor,Cruzas,Mutaciones,Inversiones
    ,Costo,Permutaci n\n";
106     eliteFile << "Generaci n,Costo,Permutaci n\n";
107
108     // CICLO
109     int gen = 1;
110     while (gen < MAX_GENERATIONS) {
111         std::cout << "\n=== GENERACION " << gen << " ===" << std::endl;
112
113         // OBTENER EL MEJOR INDIVIDUO DE LA GENERACION
114         auto elite = population.getFittest();
115
116         // CREAR NUEVA POBLACION
117         Population newPop;
118
119         // SELECCIONAR N INDIVIDUOS
120         auto selected = selector.selectMany(population, POP_SIZE);
121
122         int crosses = 0, mutations = 0, inversions = 0;
123
124         // CRUZAR, MUTAR, INVERTIR Y EVALUAR LOS HIJOS
125         for (size_t i = 0; i + 1 < selected.size(); i += 2) {
126             // OBTENER PADRES
127             auto p1 = selected[i];
128             auto p2 = selected[i + 1];
129
130             // GENERAR HIJOS
131             auto [h1, h2] = obx.crossover(p1, p2);
132             ++crosses;
133
134             // MUTAR LOS HIJOS
135             mutator->mutate(h1);
136             mutator->mutate(h2);
137             mutations += 2;
138
139             // INVERSIO CON POLITICA DE REMPLAZAR SIEMPRE
140             Utils::applyInversion(h1, INVERSION_RATE, p1->
    getDigitChromosome()[0],
141                                   p1->getDigitChromosome()[n - 1],
142                                   Utils::InversionPolicy::ALWAYS_REPLACE);
```

```
143             Utils::applyInversion(h2, INVERSION_RATE, p2->
    getDigitChromosome()[0],
144                                   p2->getDigitChromosome()[n - 1],
145                                   Utils::InversionPolicy::ALWAYS_REPLACE);
146             inversions += 2;
147 
148             // EVALUACION DE LOS HIJOS
149             double fx1 = evaluate(h1->getDigitChromosome(), dist, flujo);
150             double fx2 = evaluate(h2->getDigitChromosome(), dist, flujo);
151             h1->setFitness(1.0 / (1.0 + fx1));
152             h2->setFitness(1.0 / (1.0 + fx2));
153 
154             // AGREGAR INDIVIDUOS A LA POBLACION
155             newPop.addIndividual(h1);
156             if (newPop.size() < POP_SIZE)
157                 newPop.addIndividual(h2);
158         }
159 
160         // REMOVER DUPLICADOS
161         Utils::removeDuplicates(newPop.getIndividuals(), *mutator);
162 
163         // REMPLAZAR AL PEOR INDIVIDUO POR EL MEJOR DE LA GENERACION
    ANTERIOR
164         newPop.Replace(elite);
165 
166         // ESTADISTICAS
167         double avg = newPop.getAverageFitness();
168         double min = newPop.getWorstFitness();
169         double max = newPop.getFittest()->getFitness();
170 
171         // OBTENER AL MEJOR INDIVIDUO
172         auto bestGen = newPop.getFittest();
173         double bestCost = evaluate(bestGen->getDigitChromosome(), dist,
    flujo);
174         bestGen->setFitness(1.0 / (1.0 + bestCost));  // RE-SINCRONIZA
    fitness
175 
176         if (bestGen->getFitness() > bestGlobal.getFitness()) {
177             bestGlobal = *bestGen;
178             bestGlobal.setFitness(1.0 / (1.0 + bestCost));
179         }
180 
181         // COLOCAR ESTADISTICAS EN LOS ARCHIVOS
182         logFile << gen << "," << avg << "," << max << "," << min << "," <<
     crosses << ","
183                 << mutations << "," << inversions << "," << bestCost << ",
    ";
184         for (int val : bestGen->getDigitChromosome()) {
185             logFile << val << " ";
186         }
187         logFile << "\n";
188 
```

```
        eliteFile << gen << "," << bestCost << ",";
        for (int val : bestGen->getDigitChromosome()) {
            eliteFile << val << " ";
        }
        eliteFile << "\n";

        // IMPRIMIR ESTADISTICAS EN PANTALLA
        std::cout << "Media aptitud: " << avg << "\n";
        std::cout << "Maxima aptitud: " << max << "\n";
        std::cout << "Minima aptitud: " << min << "\n";
        std::cout << "Cruzas: " << crosses << ", Mutaciones: " <<
    mutations
                  << ", Inversiones: " << inversions << "\n";

        // IMPRESION DEL MEJOR INDIVIDUO DE LA GENERACION
        std::cout << "|--- MEJOR INDIVIDUO DE LA GENERACION " << gen << "
    ---|" << std::endl;
        bestGen->printIndividual();
        std::cout << "COSTO : " << bestCost << std::endl;

        // REMPLAZO DE LA GENERACION
        population = newPop;
        gen++;
    }

    // IMPRIMIR ESTADISTICAS FINALES
    std::cout << "\n|--- MEJOR INDIVIDUO GLOBAL ---|" << std::endl;
    bestGlobal.printIndividual();
    std::cout << "COSTO : " << std::fixed << evaluate(bestGlobal.
    getDigitChromosome(), dist, flujo)
              << "\n";
    // CERRAR ARCHIVOS
    logFile.close();
    eliteFile.close();

    return 0;
}
```

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` para activar advertencias útiles durante el desarrollo como se muestra en las figuras 1. La línea de compilación utilizada es:

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

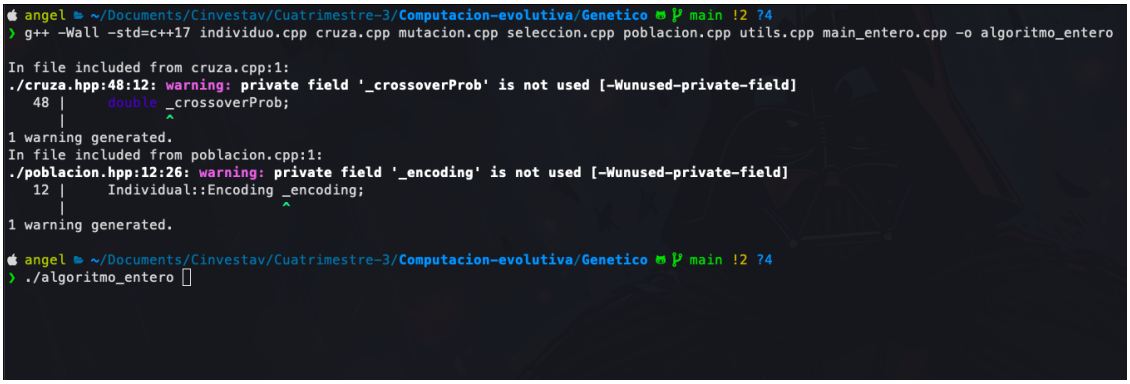

Figura 1: Compilación y ejecución del código fuente

Y la ejecución del programa se realizó mediante el comando como se muestra en la figura 2:

```
./algoritmo_entero
```

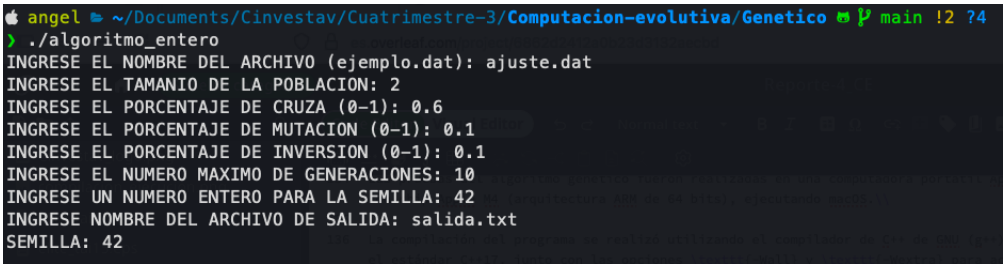

Figura 2: Ejecución del algoritmo genético

A continuación, en la figura 3 se muestra un ejemplo de una corrida para el archivo ajuste.dat que muestra la salida del código.

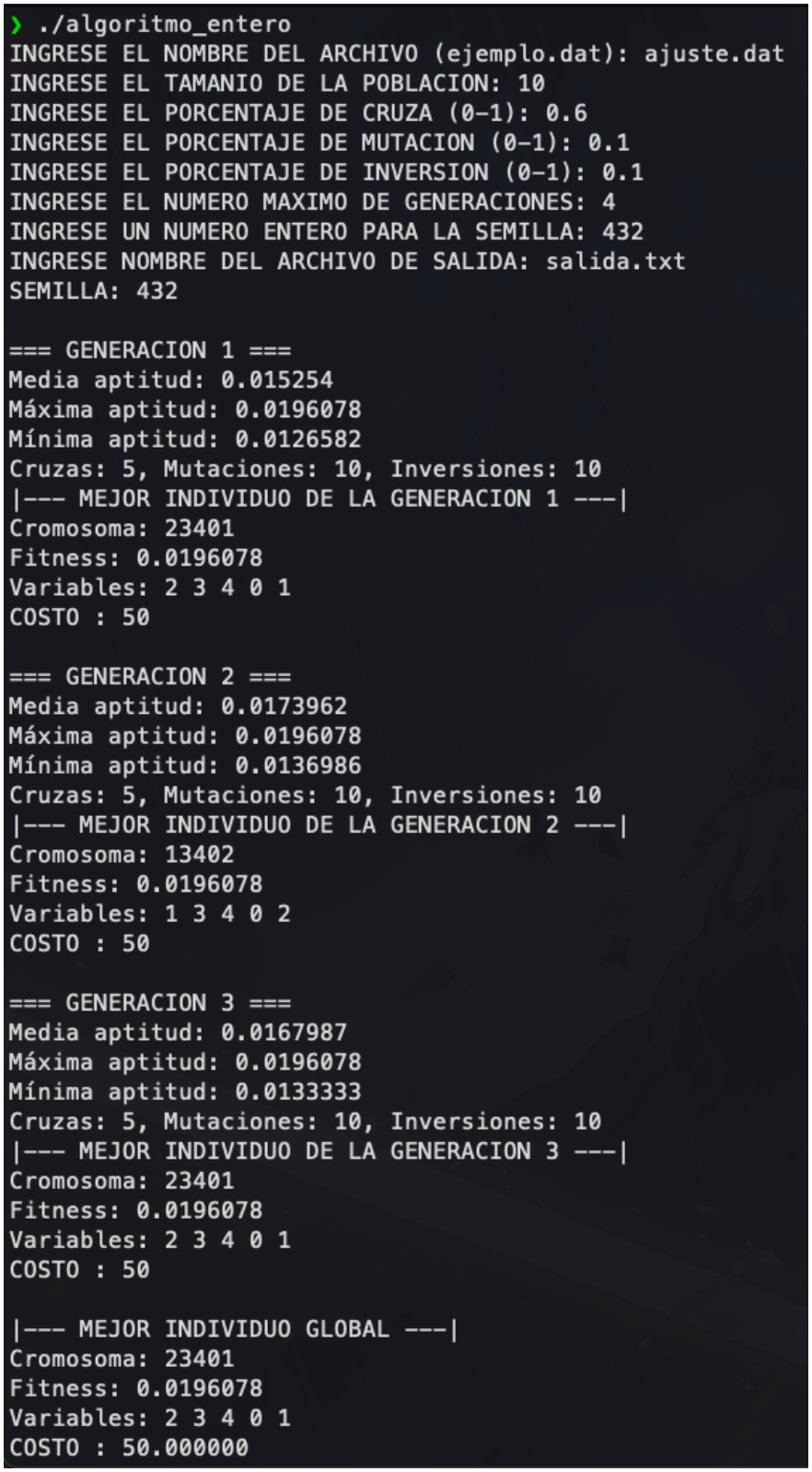

Figura 3: Ejemplo de corrida del algoritmo genético

Estadísticas

En esta sección se presentan los resultados estadísticos obtenidos a partir de 20 corridas independientes del algoritmo genético, Las soluciones que representan las permutaciones se leen de izquierda a derecha.

Parámetro	**Valor**
Tamaño de población	100
Máximo numero de generaciones	1000
Probabilidad de cruza	0.600000
Probabilidad de mutación	0.10000
Probabilidad de inversión	0.10000

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

Archivo TAI12

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 permutación, fitness y costo

Corrida	**Semilla**	**Permutación**	**Fitness**	**Costo**
1	146	10 4 3 0 6 9 5 7 8 11 2 1	4.30728e-06	232164
2	94	1 11 4 0 8 3 10 9 5 7 2 6	4.20764e-06	237662
3	274	1 3 6 11 7 2 9 0 8 5 4 10	4.45599e-06	224416
4	398	6 9 11 3 0 4 7 5 2 1 8 10	4.16981e-06	239818
5	846	10 11 3 0 1 9 4 8 7 5 2 6	4.33454e-06	230704
6	477	10 11 3 0 1 9 4 8 7 5 2 6	4.33454e-06	230704
7	59	1 3 6 11 7 2 9 0 8 5 4 10	4.45599e-06	224416
8	903	6 2 10 11 5 4 1 3 8 7 0 9	4.19877e-06	238164
9	638	10 2 6 0 3 5 8 4 1 7 11 9	4.18132e-06	239158
10	249	9 11 5 0 10 1 4 8 3 7 2 6	4.24529e-06	235554
11	703	10 11 3 0 1 9 4 8 7 5 2 6	4.33454e-06	230704
12	327	2 11 9 3 5 8 10 0 1 7 4 6	4.09851e-06	243990
13	43	1 3 6 11 7 2 9 0 8 5 4 10	4.45599e-06	224416
14	17	10 11 3 0 1 9 4 8 7 5 2 6	4.33454e-06	230704
15	876	9 6 2 4 0 11 7 3 10 8 5 1	4.13741e-06	241696
16	68	1 3 6 11 7 2 9 0 8 5 4 10	4.45599e-06	224416
17	918	1 3 6 11 7 2 9 0 8 5 4 10	4.45599e-06	224416
18	511	6 9 11 3 0 4 7 5 2 1 8 10	4.16981e-06	239818
19	722	2 10 9 11 3 7 1 0 8 5 4 6	4.16929e-06	239848
20	292	4 11 8 0 1 9 5 7 10 3 2 6	4.24259e-06	235704

Estadísticas:
Media Fitness: 4.296e-06
Varianza Fitness: 1.23e-14
Desviación estándar Fitness: 1.11e-07
Media Costo: 236,504
Peor Costo: 243,990
Mejor Costo: 224,416
Mejor Permutación: 1 3 6 11 7 2 9 0 8 5 4 10

Archivo TAI15

Cuadro 2: Resultados de las 20 corridas del algoritmo genético con permutación, fitness y costo

Corrida	Semilla	Permutación	Fitness	Costo
1	888	10 11 3 8 14 6 2 12 13 1 7 5 4 9 0	2.39856e-06	416916
2	664	5 11 14 7 1 3 10 6 2 0 9 8 4 13 12	2.50426e-06	399318
3	802	10 2 11 0 14 4 1 8 7 5 3 13 9 6 12	2.57077e-06	388988
4	325	6 5 4 10 9 8 0 13 1 7 11 14 2 12 3	2.4355e-06	410592
5	973	10 8 12 3 9 4 6 1 13 14 7 5 2 0 11	2.45738e-06	406936
6	192	0 7 14 8 1 3 10 6 2 5 9 13 4 11 12	2.48655e-06	402162
7	703	1 0 4 11 12 2 6 14 9 8 7 5 13 10 3	2.48881e-06	401798
8	604	14 10 9 13 0 11 1 6 12 4 5 8 2 3 7	2.43054e-06	411430
9	155	9 3 2 8 7 0 6 4 11 13 14 1 5 10 12	2.46958e-06	404926
10	22	8 5 7 0 13 4 1 6 3 11 10 2 14 12 9	2.47276e-06	404406
11	318	14 6 10 7 9 11 3 1 0 4 12 8 13 2 5	2.43038e-06	411458
12	269	1 2 4 12 11 0 10 8 9 14 5 3 7 6 13	2.48855e-06	401840
13	487	5 4 12 7 1 2 6 10 0 9 8 14 11 13 3	2.48601e-06	402250
14	945	1 2 13 5 11 3 10 4 14 9 8 6 0 7 12	2.47347e-06	404290
15	95	2 1 14 3 5 11 10 6 13 12 9 8 4 7 0	2.42424e-06	412500
16	205	14 10 13 11 12 3 6 9 8 0 1 5 4 7 2	2.4482e-06	408462
17	93	1 11 13 0 7 4 6 9 2 3 14 5 8 10 12	2.42612e-06	412180
18	846	9 11 2 4 7 14 6 5 1 10 0 3 8 13 12	2.44368e-06	409218
19	121	7 1 11 3 2 4 6 10 14 0 8 9 5 13 12	2.42424e-06	412500
20	586	9 1 13 0 11 10 3 12 5 7 2 6 14 4 8	2.4541e-06	407480

Estadísticas:
Media Fitness: 2.460e-06
Varianza Fitness: 2.2e-15
Desviación estándar Fitness: 4.7e-08
Media Costo: 406,883
Peor Costo: 416,916
Mejor Costo: 388,988
Mejor Permutación: 10 2 11 0 14 4 1 8 7 5 3 13 9 6 12

Archivo TAI30

Cuadro 3: Resultados de las 20 corridas del algoritmo genético para TAI 30

Corrida	Semilla	Permutación	Fitness	Costo
1	247	25 7 16 5 4 26 9 1 10 18 29 19 6 3 21 28 8 15 27 0 12 22 14 11 2 20 17 13 23 24	4.93038e-07	2028240
2	147	9 10 19 3 25 4 29 22 13 18 21 1 17 11 7 2 16 27 28 12 24 8 15 26 14 23 5 0 6 20	4.93329e-07	2027044
3	788	20 23 21 6 27 24 12 1 17 29 2 9 8 7 11 13 22 10 5 3 0 28 19 18 25 16 14 15 4 26	4.95965e-07	2016272
4	426	26 8 18 27 11 23 19 0 9 20 1 7 6 14 2 4 24 22 28 17 12 29 15 10 21 3 25 13 16 5	4.94648e-07	2021640
5	966	10 22 1 5 20 2 16 7 6 29 15 27 8 26 19 24 25 13 28 17 18 3 14 23 4 9 21 0 11 12	5.03089e-07	1987718
6	411	1 13 16 15 5 3 26 17 27 25 28 14 24 23 10 29 12 7 20 22 2 21 8 9 6 0 18 11 19 4	4.91006e-07	2036632
7	798	1 15 2 3 7 9 22 12 17 13 4 5 10 14 18 16 28 0 27 6 19 8 21 24 26 11 29 20 23 25	4.92433e-07	2030734
8	71	16 8 10 0 9 12 27 28 5 3 22 25 26 18 15 13 1 24 29 19 17 14 23 6 2 11 20 4 7 21	4.9178e-07	2033430
9	944	17 2 3 21 18 20 29 23 19 27 10 22 13 25 1 5 6 12 15 14 11 8 28 24 16 26 7 0 9 4	4.9638e-07	2014586
10	345	27 29 15 0 2 13 9 18 4 16 3 1 11 19 21 23 22 25 5 17 14 10 20 28 7 12 8 26 24 6	4.89719e-07	2041986
11	778	12 29 13 11 26 4 15 18 9 7 20 3 19 2 17 21 14 22 1 24 25 23 0 8 5 27 6 16 10 28	4.98503e-07	2006006
12	177	26 0 17 23 8 20 12 14 19 15 27 22 16 4 29 13 1 10 2 28 7 9 18 21 25 6 11 24 3 5	4.95365e-07	2018712
13	645	16 24 1 28 7 18 14 21 27 23 22 6 15 25 20 26 4 29 10 2 3 11 9 17 5 12 13 0 19 8	4.92999e-07	2028400
14	650	1 8 22 4 0 11 5 7 18 19 15 28 3 29 9 17 16 10 12 24 20 25 2 14 13 6 23 27 21 26	4.89757e-07	2041826
15	921	2 3 8 29 28 10 16 4 11 13 17 18 14 15 22 24 6 20 12 27 25 0 7 1 9 21 23 26 19 5	4.93741e-07	2025352
16	570	17 15 14 4 2 19 6 16 5 11 7 10 12 29 1 18 13 21 3 23 20 9 26 22 28 25 27 8 24 0	4.92087e-07	2032158
17	638	1 6 11 26 25 9 14 16 12 13 8 5 21 7 18 17 10 22 27 3 23 24 0 20 28 19 2 29 15 4	4.96414e-07	2014446
18	298	20 27 10 24 9 2 4 5 6 25 14 23 22 18 16 7 8 13 12 1 15 26 11 21 29 19 3 0 17 28	4.93737e-07	2025370
19	123	16 20 28 9 21 3 12 8 10 23 13 19 4 0 7 11 26 6 22 27 29 17 1 14 2 25 18 5 15 24	4.88156e-07	2048526
20	306	6 19 29 3 2 7 27 20 17 28 5 16 24 18 22 15 0 13 23 25 9 11 21 10 14 26 12 8 1 4	4.93814e-07	2025052

Cuadro 4: Estadísticas finales de las 20 corridas TAI 30

Estadística	**Valor**
Media Fitness	4.940e-07
Varianza Fitness	3.2e-17
Desviación estándar Fitness	5.7e-09
Media Costo	2 026 805
Peor Costo	2 048 526
Mejor Costo	1 987 718
Mejor Permutación	10 22 1 5 20 2 16 7 6 29 15 27 8 26 19 24 25 13 28 17 18 3 14 23 4 9 21 0 11 12

Gráficas de convergencia

A continuación, se presentan las gráficas de convergencia correspondientes a los resultados ubicados en la mediana de sus respectivos problemas. Cada gráfica muestra la evolución del individuo élite en términos de su valor de aptitud a lo largo de las generaciones.

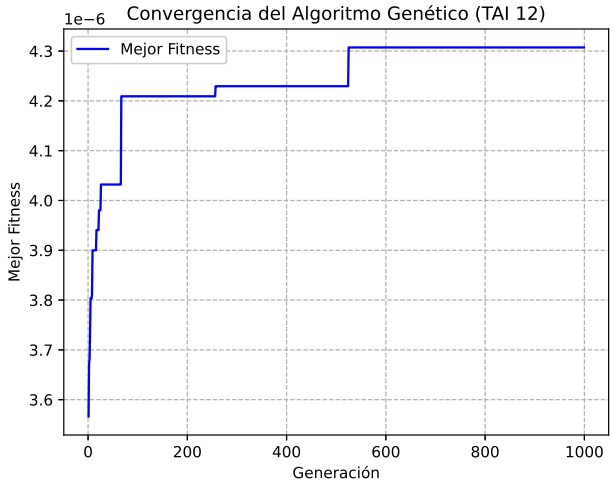

Figura 4: Gráfica de convergencia TAI12

La Figura 4 ilustra cómo el individuo élite mejora su aptitud generación tras generación para la instancia TAI12, mostrando una tendencia ascendente constante hasta estabilizarse cerca del óptimo encontrado.

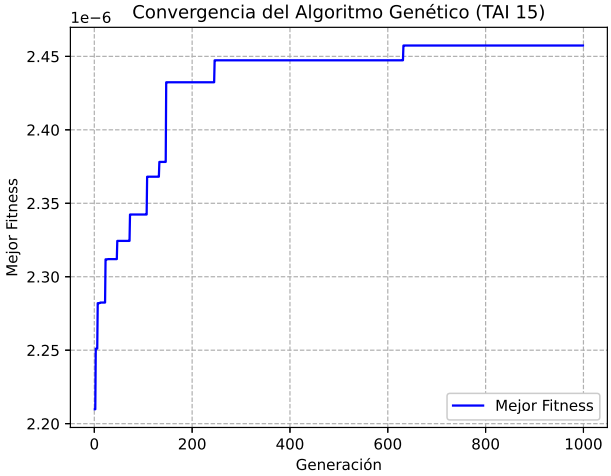

Figura 5: Gráfica de convergencia TAI15

En contraste, la Figura 5 muestra la evolución para TAI15, donde se observa un patrón similar de mejora gradual, aunque con posibles oscilaciones debido a la complejidad del paisaje de búsqueda.

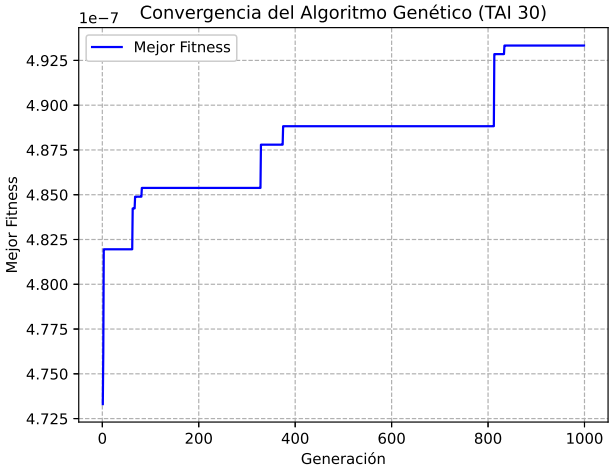

Figura 6: Gráfica de convergencia TAI30

Por último, la Figura 6 corresponde a la instancia TAI30, evidenciando un progreso sostenido en la calidad de la solución élite a lo largo de todas las generaciones.

Como se observa en las figuras, el individuo élite mejora su valor de aptitud de forma consistente, alcanzando una convergencia progresiva que valida el buen desempeño del algoritmo evolutivo implementado.