UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Departamento de Sistemas Informáticos y Computación

Problema del Cartero Chino
Técnicas de Inteligencia Artificial

Luis Cardoza Bird.

Octubre de 2023

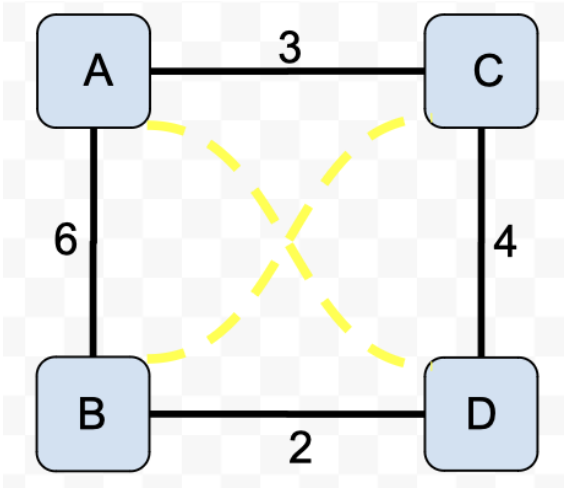
I. Introducción

El **problema del cartero chino**, a menudo denominado "**problema de la ruta del cartero**", es un reto emblemático en la teoría de grafos y la optimización combinatoria.

A diferencia de otros problemas clásicos, como el del Viajante de Comercio, el carácter distintivo del **Problema del Cartero Chino** radica en la permisividad de recorrer ciertas rutas más de una vez con el propósito de **minimizar** un **costo total**, en lugar de visitar cada punto exactamente una vez.

II. Problema

Imaginemos una ciudad cuyas calles están representadas por un grafo. Cada intersección es un vértice y cada calle es una arista. El desafío para el cartero es recorrer todas las calles, minimizando la distancia total y evitando repetir tramos innecesarios.



NODOS	A	B	C	D
A	-	AB	AC	AD
B	BA	-	BC	BD
C	CB	CB	-	CD
D	DB	DB	DC	-

III. Objetivos

A. General

- Desarrollar la solución más óptima utilizando técnicas de inteligencia artificial.

B. Específico

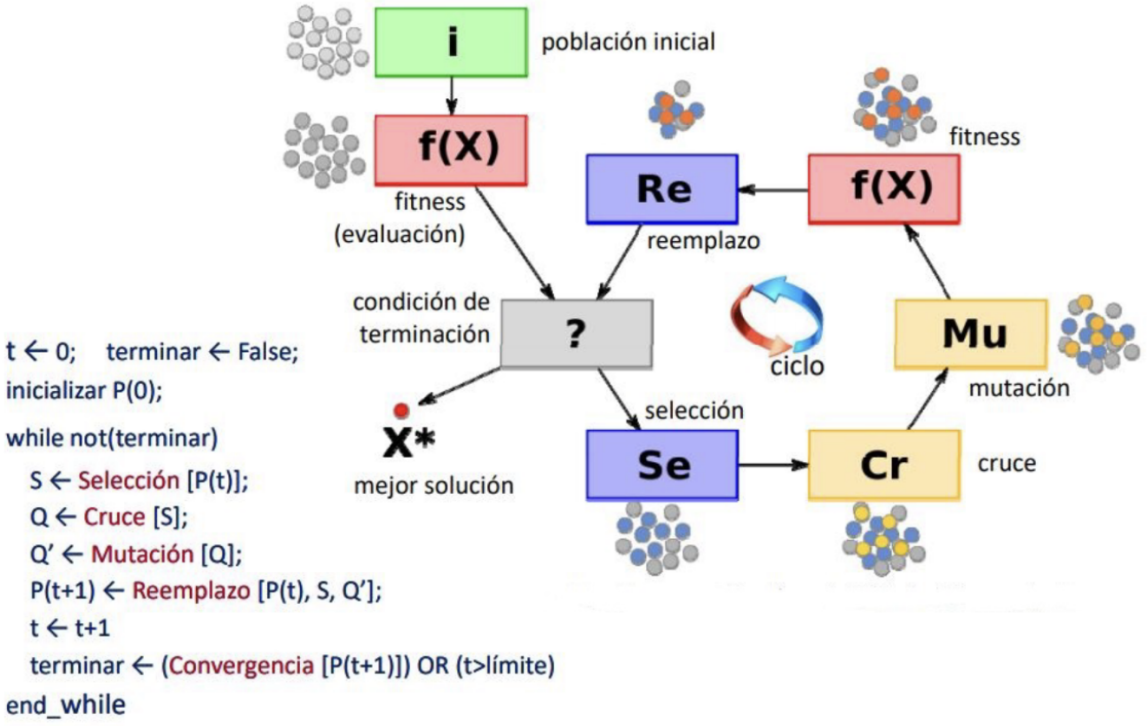
- Generación de solución con **Algoritmo Genético**.
- Generación de solución con **Algoritmo de enfriamiento simulado**.

IV. Algoritmos

1. Algoritmo Genético

a. Definición

Un algoritmo genético (AG) es un algoritmo de búsqueda heurística inspirado en el proceso de selección natural que pertenece a la clase más grande de algoritmos evolutivos. Los AGs son usados para encontrar soluciones aproximadas a problemas de optimización y búsqueda, especialmente cuando el espacio de búsqueda es grande.



Se podría definir que los algoritmos genéticos se inspiran en la naturaleza y han demostrado ser herramientas poderosas para resolver problemas difíciles en diversas áreas, desde la optimización hasta la modelización y el diseño.

b. Elementos

- **Población:** Conjunto inicial de posibles soluciones (individuos). Estas soluciones pueden codificarse de muchas formas, aunque la representación binaria es la más común.
- **Selección**: Mecanismo por el cual se eligen individuos de la población para ser padres. La selección suele estar basada en la calidad o adaptación de los individuos.
- **Cruzamiento (Reproducción):** Proceso mediante el cual dos individuos (padres) producen descendencia combinando sus características. Esto se hace a través de operadores de cruzamiento.
- **Mutación:** Pequeños cambios aleatorios en los individuos para mantener la diversidad en la población y evitar la convergencia prematura.
- **Función de Aptitud:** Evalúa qué tan "buena" es una solución dada. Es esencial para guiar la búsqueda.
- **Criterio de Terminación:** Define cuándo debe detenerse el algoritmo. Esto podría basarse en un número fijo de generaciones, un tiempo de ejecución o una mejora mínima en la función de aptitud.

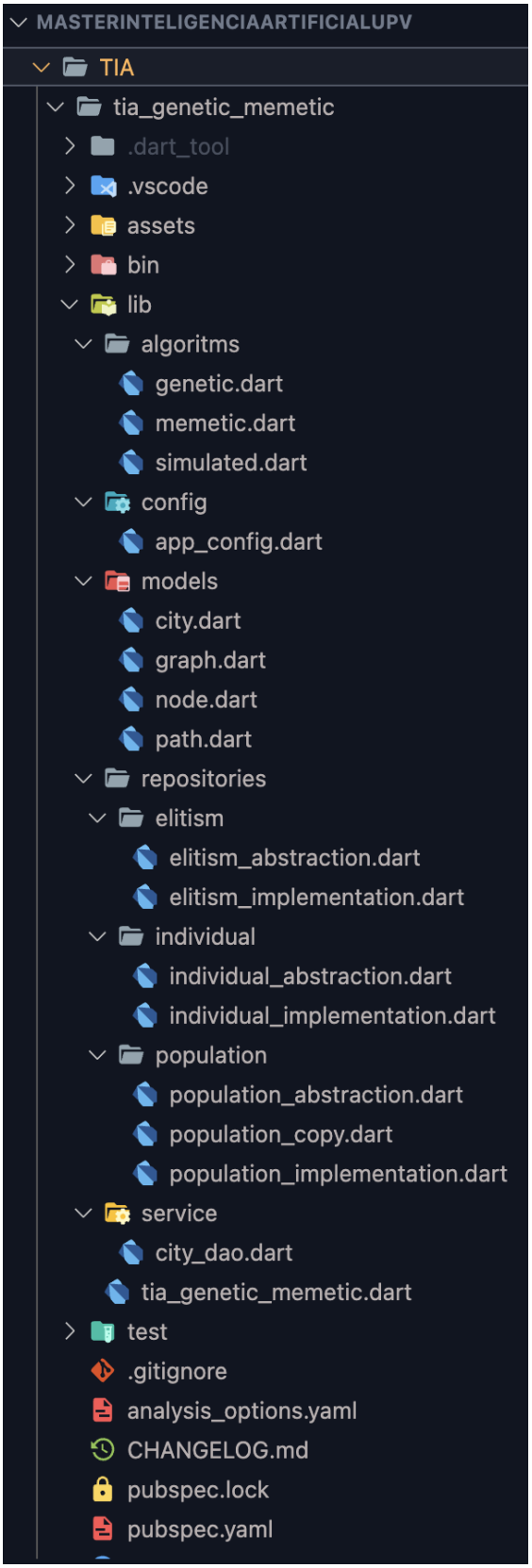
c. Ventajas / Desventajas

VENTAJAS	DESVENTAJAS
Adaptabilidad	No garantizado
Generalidad	Rendimiento
Soluciones Aproximadas	Configuración
Paralelismo	Convergencia Prematura

V. Stack

- 

VI. Estructura del Proyecto



genetic.dart: Implementación de algoritmo genético.

memetic.dart: Implementación de un algoritmo memético, que combina aspectos de algoritmos genéticos y búsqueda local.

simulated.dart: Implementación de enfriamiento simulado, una técnica probabilística para aproximar el óptimo global de una función dada.

app_config.dart: Variables de configuración globales o ajustes utilizados en todo el proyecto.

city.dart: Define la estructura de una entidad "Ciudad", utilizada en simulaciones de búsqueda de rutas o enrutamiento.

graph.dart: Define una estructura de grafo, potencialmente para representar rutas o redes en simulaciones.

node.dart: Define la estructura de un "Nodo" dentro de un grafo, que podría ser un punto o ubicación dentro de una red.

path.dart: Define la estructura de un "Camino", que podría representar una ruta o secuencia de nodos/ciudades.

Repositorios:

El subdirectorio "repositories" contiene clases abstractas y sus implementaciones.

Servicio: Directorio donde se realizan las operaciones relacionadas con el modelo “**City**”.

VII. Diseño

El diseño general del algoritmo se ha realizado con el lenguaje **DART**, el cual posee los siguientes criterios de entrada:

```dart
static const int maxIteration = 2000; // maximum number of iterations
static const int populationSize = 10; // population size
static const int mainCity = 1; // starting city id
static const int memePopulationRate = 50; // population rate
static const double temperature = 100.0; // starting temperature
static const double coolingRate = 0.9; // cooling rate
static const double mutateRate = 0.3; // mutation rate
static const double replaceRate = 0.3; // replacement rate
static const int compactRate = 5; // compaction rate
```

1. Diseño del individuo. Codificación.

En el problema del cartero chino, podemos considerar que un posible individuo sería la combinación de rutas donde el cartero debe pasar.

Se agregó la restricción de que el cartero no puede pasar por nodos que ya ha recorrido, adicionalmente se ha agregado la restricción que las ciudades solo pueden ser recorridas cuando su población es superior a 10.

Haciendo uso de las técnicas proporcionadas por el lenguaje *Dart*, se ha decidido emplear una lectura desde un archivo .CSV

```
Luis Cardoza Bird, 2 weeks ago | 1 author (Luis C
node1,node2,trail,distance,color
rs_end_north,v_rs,rs,0.3,red,0
v_rs,b_rs,rs,0.21,red,0
b_rs,g_rs,rs,0.11,red,0
g_rs,w_rs,rs,0.18,red,0
w_rs,o_rs,rs,0.21,red,0
o_rs,y_rs,rs,0.12,red,0
y_rs,rs_end_south,rs,0.39,red,0
rc_end_north,v_rc,rc,0.7,red,0
v_rc,b_rc,rc,0.04,red,0
b_rc,g_rc,rc,0.15,red,0
g_rc,o_rc,rc,0.13,red,0
o_rc,w_rc,rc,0.23,red,1      Lui
w_rc,y_rc,rc,0.14,red,0
y_rc,rc_end_south,rc,0.36,red,0
rt_end_north,v_rt,rt,0.3,red,0
v_rt,tt_rt,rt,0.16,red,0
tt_rt,b_rt,rt,0.04,red,1
b_rt,g_rt,rt,0.09,red,0
g_rt,o_rt,rt,0.05,red,1
o_rt,w_rt,rt,0.06,red,0
w_rt,y_rt,rt,0.19,red,0
y_rt,rt_end_south,rt,0.18,red,0
rh_end_north,v_rh,rh,0.24,red,0
```

2. Funcion de evaluacion

La función que hemos determinado es una *fitness* **f(x)**, donde se trata de minimizar la distancia total recorrida a través de una serie de puntos o nodos.

$$f(x) = dist(yn - 1, y0) + \Sigma i = 1\, dist(xi, xi - 1)$$

3. Generación de la población inicial

Para crear la población inicial, el algoritmo toma la variable **mainCity.**

Esta población inicial puede reemplazarse por otra ciudad, para generar una comparación más óptima.

4. Selección, mutación, cruce y reemplazo

Antes de la selección de individuos que serán sometidos a cruce, primero ordenamos toda la población actual en función de la aptitud de cada uno, de mejor a peor. Posteriormente, seleccionamos, con un porcentaje de ***replacementRate***, los mejores individuos. Por ejemplo, si hay 5701 individuos y ***replacementRate*** es 0.2, se selecciona una sublista equivalente al 20% de los individuos.

Una vez están seleccionados, el cruce que se ha realizado consiste en obtener dos soluciones usando un valor aleatorio dentro de la sublista. Debido a que es altamente probable que se repitan ciudades, se recorre las nuevas listas eliminando repetidos. De igual forma, si un número no está en ninguna lista, se añadirá a una de ellas. Esta eliminación y adición se hace de forma proporcional, de manera que no siempre se suprimen los repetidos.

Se descartan los peores individuos de la población para añadir a la nueva generación (tanto los que han mutado como los que no). Por tanto, el número de descartados es el número de nuevos individuos.

Ejecución del Algoritmo Genético:

```
P execute(int iters) {
  P p = population.copy() as P;
  for (int i = 0; i < iters; i++) {
    p.evolve(AppConfig.compactRate % (1 + i).abs() == 0);
    reportProgress(p, i, iters);
  }
  log('LAST POPULATION: ${p.retrieveBestIndividual()}');
  return p;
}
```

Mutación del individuo:

```
@override
 void mutate() {
  Random random = Random();
  int mutations = random.nextInt((genotype.length /
AppConfig.populationSize)
          .clamp(1, genotype.length)
          .toInt()) +
      1;
  for (int k = 0; k < mutations; k++) {
    int i, j;
    do {
      i = random.nextInt(genotype.length - 1) + 1;
      j = random.nextInt(genotype.length - 1) + 1;
    } while (i == j);

    int aux = genotype[i];
    genotype[i] = genotype[j];
    genotype[j] = aux;
  }
  fitness = -1.0;
}
```

SELECCION DE MEJORES INDIVIDUOS:

La selección implementada se basa en el método elitista, en el que aquellos individuos con mayor *fitness* son elegidos para el posterior cruce. El esquema algorítmico se puede apreciar a continuación:

```
List<IndividualImplementation> elitism(
    List<IndividualImplementation> population, bool compact) {
  List<IndividualImplementation> nextGeneration = List.from(population)
    ..sort();
  int eliteCount = (population.length * 0.2).round();
  nextGeneration = nextGeneration.sublist(0, eliteCount);
  int startingCity = population[0].getGenotype()[0];
  addNewCppIndividuals(nextGeneration, startingCity, eliteCount);
  while (nextGeneration.length < population.length) {
    IndividualImplementation parent1 = selectRandomIndividual(population);
    IndividualImplementation parent2 = selectRandomIndividual(population);
    nextGeneration
        .add(parent1.getChildren(parent2)[0] as IndividualImplementation);
  }
  if (compact) {
    Set<IndividualImplementation> compactedPop = Set.from(nextGeneration);
    nextGeneration.clear();
    nextGeneration.addAll(compactedPop);
    addNewCppIndividuals(nextGeneration, startingCity,
        population.length - nextGeneration.length);
  }
  nextGeneration.sort();
  return nextGeneration;
}
```

FITNESS:

```
@override
 double getFitness() {
   if (fitness < 0.0) {
     AppConfig.fitnessCalls++;
     double cost = 0.0;
```

```
    for (int i = 0; i < genotype.length - 1; i++) {
      cost += distances[genotype[i]][genotype[i + 1]];
    }
    cost += distances[genotype[genotype.length - 1]][genotype[0]];

    int? visitedEdges = originalGraph?.getVisitedEdges(genotype);
    fitness = cost /
        (visitedEdges == AppConfig.maxNumEdges
            ? (visitedEdges ?? 0) * 1000
            : 1);
  }
  return fitness;
}
```

Convergencia

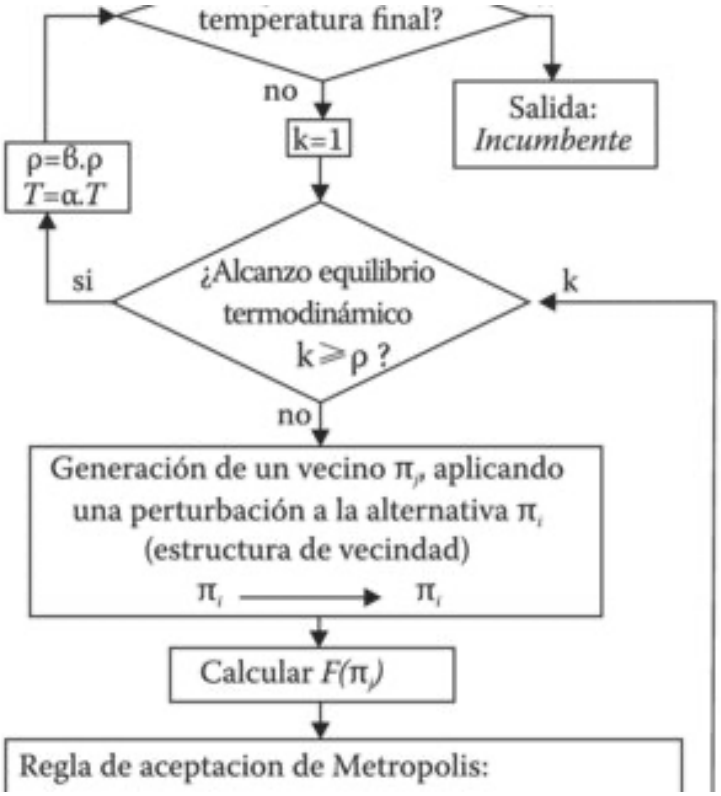
El objetivo de este método es el de comprobar si un individuo es o no óptimo, de manera que el algoritmo genético finalizará en caso afirmativo. Su esquema algorítmico es:

```
@override
I retrieveBestIndividual() {
  sort();
  return population[0].copy() as I;
}
```

2. Algoritmo búsqueda local (Enfriamiento simulado)

a. Definición

El Enfriamiento Simulado (Simulated Annealing, SA) es una técnica de optimización basada en el proceso de enfriamiento y fortalecimiento de metales. A diferencia de la búsqueda local estándar que solo se mueve a soluciones vecinas que mejoran la función objetivo, Se permite movimientos a soluciones peores con una probabilidad que decrece con el tiempo. Esta característica le permite escapar de los óptimos locales.



b. Elementos

- **Solución Inicial:** Punto de partida en el espacio de búsqueda.
- Función de Vecindad: Define qué soluciones son "vecinas" de una solución dada.
- **Función Objetivo o de Costo:** Evalúa la calidad de una solución.
- Temperatura: Parámetro que controla la probabilidad de aceptar soluciones peores. Comienza alto y disminuye con el tiempo.
- **Esquema de Enfriamiento:** Define cómo se reduce la temperatura a lo largo del tiempo, por ejemplo, multiplicar la temperatura actual por un factor constante.
- **Probabilidad de Aceptación:** Dada una solución peor que la actual, define la probabilidad con la que se acepta. Es una función de la diferencia de costo entre las soluciones y la temperatura actual.
- **Criterio de Terminación:** Establece cuándo detener el algoritmo, como un número máximo de iteraciones o una temperatura mínima.

c. Ventajas / Desventajas

VENTAJAS	DESVENTAJAS
Capacidad de evitar óptimos locales	Requiere afinación
Aplicable a una amplia gama de problemas	Convergencia
Soluciones Aproximadas	Tiempo de ejecución

Esquema general

El esquema algorítmico para el enfriamiento simulado puede resumirse de esta forma: En el caso de obtener una solución que mejore el *fitness*, se acepta. En caso de que la solución empeore la aptitud, se acepta con una determinada probabilidad, que depende de una cierta temperatura. Inicialmente para valores altos de la temperatura la mayoría de sucesores

serán aceptados. A medida que la temperatura vaya decreciendo, disminuirá la probabilidad de aceptar un sucesor.

El esquema algorítmico general de nuestro enfriamiento simulado es:

```
I start() {
  while (temperature > 1.0) {
    var neighbors = currentIndividual.getNeighborhood().cast<I>();
    neighbors.sort((a, b) => a.getFitness().compareTo(b.getFitness()));

    I bestNeighbor = neighbors[0];
    double fitnessDifference =
        bestNeighbor.getFitness() - currentIndividual.getFitness();

    if (fitnessDifference < 0.0 || _shouldAccept(fitnessDifference)) {
      currentIndividual = bestNeighbor;
      if (currentIndividual.getFitness() < bestIndividual.getFitness()) {
        bestIndividual = currentIndividual;
      }
    }

    temperature *= coolingRate;
  }
  return bestIndividual;
}

bool _shouldAccept(double fitnessDifference) {
  if (fitnessDifference < 0) return true;
  return Random().nextDouble() < exp(-fitnessDifference / temperature);
}
```

Selección de Sucesores:

```
@override
List<IndividualInterface<int>> getChildren(IndividualInterface<int> partner) {
  List<IndividualInterface<int>> children = [];
  int blockSize = Random().nextInt(7) + 1;
  List<int> p1 = List.from(genotype);
  List<int> p2 = List.from((partner as IndividualImplementation).genotype);
  IndividualImplementation child = IndividualImplementation._empty();
  bool useParent1 = true;
  for (int i = 0; i < child.genotype.length; i++) {
    if (i % blockSize == 0) useParent1 = !useParent1;
    int gene = useParent1 ? p1.removeAt(0) : p2.removeAt(0);
```

```
    p1.remove(gene);
    p2.remove(gene);
    child.genotype[i] = gene;
  }
  children.add(child);
  return children;
}
```

Parámetros de evaluación

Por otra parte, en la evaluación del enfriamiento simulado se ha realizado una com- parativa con los resultados que nos proporcionan distintos valores de los parámetros del algoritmo. Estos valores son:

Para temperatura: 100. Para coolingRate: 0.9

A. Resumen del proceso de optimización

- Resultados Iniciales
  - **Numero de Iteracion:** 0/2000
- Resultados de Algoritmo Genético
  - **Seed:** `1633124807218`
  - **Fitness:** `595.14`
  - **Número de ciudades:** `77`
  - **Secuencia de Genotipo:**

```
1, 20, 3, 2, 17, 21, 28, 38, 40, 42, 45, 51, 12, 53, 58, 14, 15, 64, 50,
26, 74, 27, 34, 62, 16, 18, 46, 43, 44, 70, 22, 25, 8, 11, 9, 76, 60, 29,
32, 49, 59, 35, 68, 65, 48, 57, 36, 71, 61, 66, 10, 19, 13, 24, 4, 23, 56,
47, 69, 73, 67, 54, 72, 63, 55, 75, 41, 52, 5, 39, 6, 30, 0, 31, 37, 33, 7
```

- Resultados de Enfriamiento Simulado
  - **Seed:** `1633124807218`
  - **Fitness:** `595.14`
  - **Número de ciudades:** `77`
  - **Secuencia de Genotipo:**

```
1, 20, 3, 2, 17, 21, 28, 38, 40, 42, 45, 51, 12, 53, 58, 14, 15, 64, 50,
26, 74, 27, 34, 62, 16, 18, 46, 43, 44, 70, 22, 25, 8, 11, 9, 76, 60, 29,
32, 49, 59, 35, 68, 65, 48, 57, 36, 71, 61, 66, 10, 19, 13, 24, 4, 23, 56,
47, 69, 73, 67, 54, 72, 63, 55, 75, 41, 52, 5, 39, 6, 30, 0, 31, 37, 33, 7
```

B. Iteración 1999/2000

- Resultados de Algoritmo Genético
  - **Seed:** `1633124807218`
  - **Fitness:** `559.38`
  - **Número de ciudades:** `77`
  - **Secuencia de Genotipo:**

```
1, 31, 0, 20, 18, 7, 3, 5, 39, 2, 37, 17, 30, 16, 23, 21, 33, 28, 68, 38,
52, 40, 76, 36, 72, 55, 57, 26, 75, 42, 65, 45, 59, 34, 51, 12, 53, 58,
```

64, 43, 60, 14, 50, 63, 15, 74, 27, 62, 29, 54, 46, 66, 44, 70, 22, 25, 8, 11, 9, 32, 49, 35, 48, 71, 61, 10, 19, 13, 24, 4, 56, 47, 41, 69, 73, 67, 6

- Resultados de Enfriamiento Simulado
  - **Seed:** `1633124807218`
  - **Fitness:** `553.42`
  - **Número de ciudades:** `77`
  - **Secuencia de Genotipo:**

1, 20, 0, 4, 3, 6, 5, 2, 39, 14, 23, 17, 31, 7, 18, 33, 27, 63, 16, 30, 21, 37, 28, 75, 38, 68, 26, 52, 34, 62, 40, 44, 41, 42, 45, 51, 29, 12, 57, 53, 71, 58, 74, 64, 60, 15, 50, 46, 43, 72, 65, 48, 76, 70, 22, 25, 8, 11, 9, 32, 69, 49, 59, 35, 36, 61, 66, 10, 19, 13, 24, 56, 55, 47, 73, 67, 54

- **Resultados Finales**
  - **Seed:** `1633124807218`
  - **Fitness:** `559.38`
  - **Número de ciudades:** `77`
  - **Secuencia de Genotipo:**

1, 31, 0, 20, 18, 7, 3, 5, 39, 2, 37, 17, 30, 16, 23, 21, 33, 28, 68, 38, 52, 40, 76, 36, 72, 55, 57, 26, 75, 42, 65, 45, 59, 34, 51, 12, 53, 58, 64, 43, 60, 14, 50, 63, 15, 74, 27, 62, 29, 54, 46, 66, 44, 70, 22, 25, 8, 11, 9, 32, 49, 35, 48, 71, 61, 10, 19, 13, 24, 4, 56, 47, 41, 69, 73, 67, 6

C. Resultados Óptimos

- **Fitness Calls:** `18249701`
- **Best Fitness (Enfriamiento Simulado):** `559.38`
- **Best Fitness (Genético):** `553.42`

VIII. Conclusiones

Como se ha podido observar, el algoritmo genético es el que aporta la mejor solución a igualdad de condiciones. Sin embargo, el coste computacional y el número de llamadas a la función fitness que realiza, lo hace demasiado lento en comparación.

Adicional, los resultados en esta iteración, al volver a realizar el proceso de evaluación se ha encontrado que en otras seeds, el mejor modelo se invierte y apunta al Enfriamiento Simulado.