

Resolución del problema de las N Reinas mediante Algoritmos Genéticos con JGAP

José Antonio Pozo Núñez

Métodos Computacionales en Vida Artificial

Máster Universitario en Lógica, Computación e Inteligencia Artificial

1. Introducción

Este proyecto viene motivado por dos razones principales, primeramente por el propósito de profundizar en el ámbito de los algoritmos evolutivos, más concretamente, sobre los algoritmos genéticos. Como segunda finalidad, la de aprender, probar y desarrollar con el framework para algoritmos genéticos, JGAP.

1.1 Algoritmos Genéticos

Los algoritmos genéticos son mecanismos de búsqueda basados en las leyes de la selección natural y de la genética. Combinan la supervivencia de los individuos mejor adaptados junto con operadores de búsqueda genéticos como la mutación y el cruce, de ahí que sean comparables a una búsqueda biológica. Fueron desarrollados por John Holland y Rechemberg que crearon algoritmos de optimización imitando los principios básicos de la naturaleza. Estos algoritmos se utilizan con éxito para gran variedad de problemas que no permiten una solución eficiente a través de la aplicación de técnicas convencionales.

Tiene asimismo aplicaciones variadas en ingeniería, como es el diseño de piezas (turbinas, rotores, etc.), diseño y control de redes, planificación de tareas, síntesis de mecanismos, diseño de tolerancias, etc. y aplicaciones en otros gremios como en sistemas de computación paralelos (paralelización automática de programas secuenciales), química (optimización de procesos de producción, diseño tecnológico y de instalaciones), negocios y comercio (modelización de sistemas económicos complejos, predicción de mercados), medicina (análisis de datos en medicina, diagnóstico automático) o gestión (análisis de datos en gestión, asistentes de gestión, sistemas automáticos de decisión).

Basados en modelos computacionales de la evolución biológica natural, los algoritmos genéticos pertenecen a la clase de los algoritmos evolutivos, junto con la programación evolutiva, la evolución de estrategias y la programación genética.

Para la ingeniería, los algoritmos genéticos y la programación evolutiva en general presentan oportunidades de plantearse problemas que no permiten una solución eficiente a través de la aplicación de técnicas convencionales.

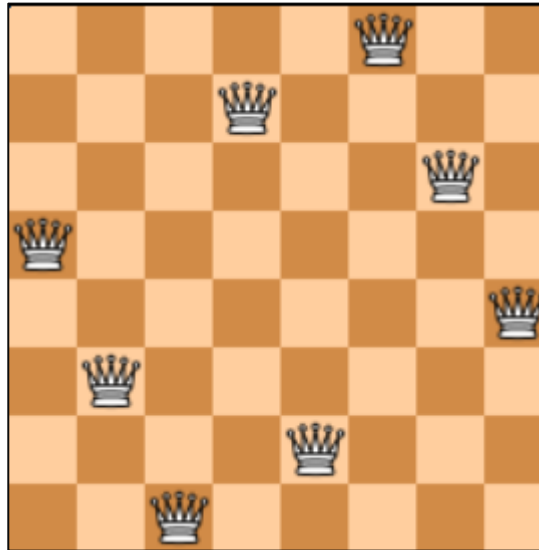
1.2. Problema de las N Reinas

El problema de las N Reinas es un problema que consiste en poner N Reinas en un tablero de ajedrez sin que se amenacen. Fue propuesto por el ajedrecista alemán Max Bezzel en 1848. En el juego del ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal. El juego de las

N reinas consiste en poner sobre un tablero de ajedrez N reinas sin que estas se amenacen entre ellas.

Aunque este problema no es un típico problema de algoritmos genéticos, como los problemas de optimización, ya que no existen soluciones buenas (o aceptablemente buenas), es decir, todas las soluciones son iguales de buenas. Sin embargo, es un problema donde se ve fácilmente la aplicación de los algoritmos genéticos sobre él.

Una posible solución a un problema de 8 Reinas, podría ser el siguiente:



2. Java Genetic Algorithms Package (JGAP)

Para la implementación de algoritmos genéticos, disponemos de un framework para Java llamado JGAP (Java Genetic Algorithms Package).

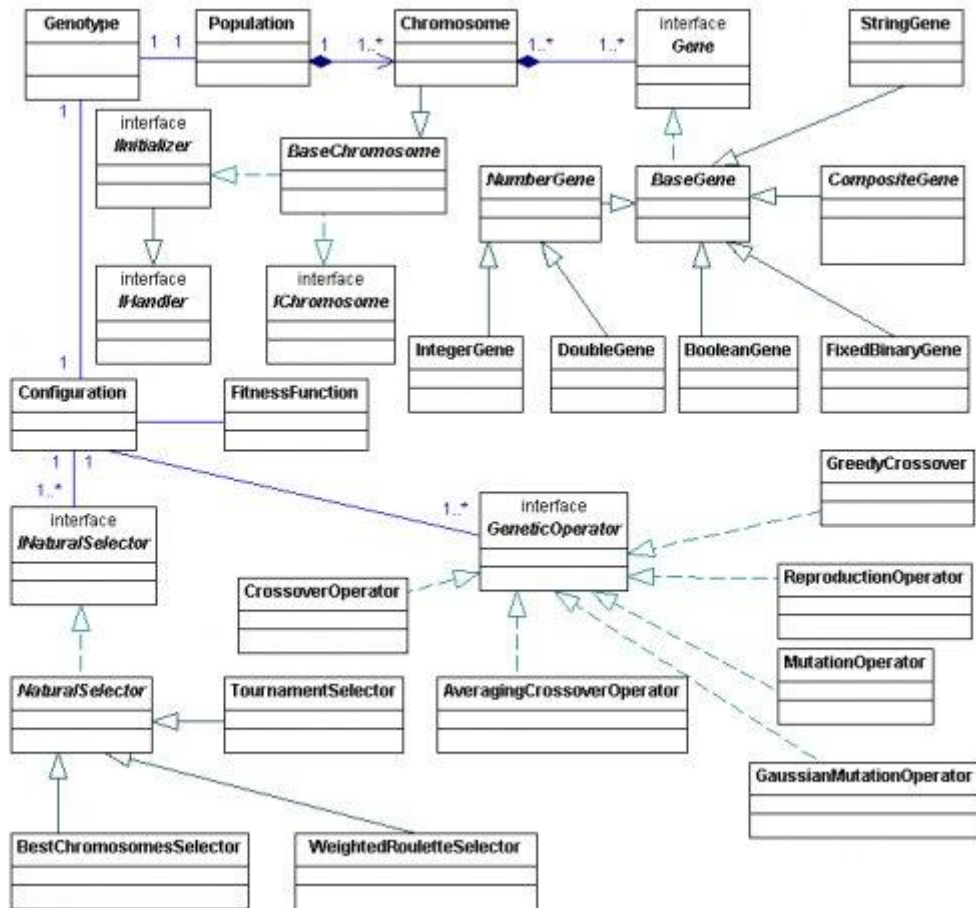
JGAP

JGAP proporciona los métodos y mecanismos básicos para la implementación de estos. Fue diseñado para que fuera bastante fácil de usar, siendo altamente modular y configurable, llegando a ser muy sencillo crear incluso nuevos y personalizados operadores genéticos.

Básicamente, para implementar un problema con algoritmos genéticos con JGAP, como mínimo, es necesario desarrollar la función de aptitud, y especificar la representación del cromosoma (o solución).

Se distribuye bajo licencia GPL. En su página web se puede encontrar gran variedad de ejemplos de uso y documentación.

El diagrama de clases de JGAP es el siguiente:



Se puede comprobar en el diagrama como ya están implementados la mayoría de operadores genéticos (cruce, mutación, selección y reproducción) y los tipos de genes más comunes. De todas formas, se puede crear los operadores y tipos de gen que se necesite para cada problema particular.

El framework JGAP se encarga de evolucionar o ejecutar los operadores en cada generación, devolviendo el individuo con mayor función de aptitud.

3. Implementación

Como hemos comentado más arriba, JGAP proporciona un framework altamente adaptable y configurable, para cada problema concreto que se quiera resolver. Para el caso que nos ocupa hemos desarrollado la función de aptitud, dos operadores de cruce y otros dos de mutación que veremos más adelante.

3.1. Representación

La representación de los cromosomas por la que se ha optado es la de un array de n posiciones (tantos como el número de reinas), en el que cada posición puede contener un número natural en el intervalo de $[0, n)$. Por ejemplo, para el caso de 5 Reinas, la representación de un cromosoma quedaría de la siguiente forma:

4	2	1	3	0
---	---	---	---	---

3.2. Función de Aptitud

Para el caso de la función de aptitud, hemos decidido vincular su valor al del número total de conflictos del cromosoma (o solución), de tal manera, que cuanto menor sea el valor, mejor aptitud tendrá el cromosoma. De hecho, sabemos que cuanto el valor de aptitud de un cromosoma sea 0, habremos encontrado una solución al problema y podremos detener la búsqueda y ejecución. Por tanto, esa será nuestra condición de parada (además, se ha acotado el número máximo de generaciones a 25.000).

```
@Override
protected double evaluate(ICHromosome cromosoma) {
    Gene[] genes = cromosoma.getGenes();
    int numeroConflictos = obtenerNumeroDeConflictos(genes);
    return numeroConflictos;
}

private int obtenerNumeroDeConflictos(Gene[] genes) {
    int numeroConflictos = 0;
    for (int i = 0; i < this.numeroReinas - 1; i++) {
        for (int j = i + 1; j < this.numeroReinas; j++) {
            if (conflictosDiagonal(genes[i], genes[j], i, j) || conflictosFila(genes[i], genes[j]))
                numeroConflictos++;
        }
    }
    return numeroConflictos;
}
```

```

private boolean conflictosDiagonal(Gene reina1, Gene reina2, int columnaReina1, int columnaReina2) {
    boolean existeConflictoDiagonal = false;
    if (columnaReina1 != columnaReina2) {
        IntegerGene gen1 = (IntegerGene) reina1;
        IntegerGene gen2 = (IntegerGene) reina2;
        int filaReina1 = gen1.intValue();
        int filaReina2 = gen2.intValue();
        existeConflictoDiagonal = (Math.abs(filaReina1 - filaReina2) == Math.abs(columnaReina1 - columnaReina2));
    }
    return existeConflictoDiagonal;
}

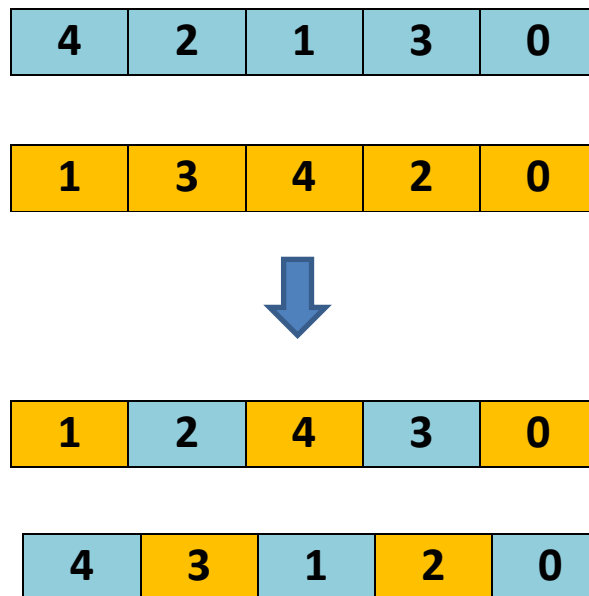
private boolean conflictosFila(Gene reina1, Gene reina2) {
    boolean existeConflictoFila = reina1.equals(reina2);
    return existeConflictoFila;
}

```

3.3. Operadores de Cruce

Se ha desarrollado dos operadores de cruce, que exponemos a continuación:

- **Parlmpar:** Como su propio nombre indica, este operador cruza dos cromosomas, obteniendo dos cromosomas hijos, cambiando los genes pares o impares de los dos padres aleatoriamente.



```

public IChromosome operate(final IChromosome primerCromosoma, final IChromosome segundoCromosoma,
    final boolean operadorPar) throws InvalidConfigurationException {
    IChromosome cromosomaResultante = (IChromosome) primerCromosoma.clone();
    Gene[] genesSegundoCromosoma = segundoCromosoma.getGenes();
    int tamanyoCromosoma = primerCromosoma.size();
    Gene[] genesResultantes = cromosomaResultante.getGenes();
    for (int i = 0; i < tamanyoCromosoma; i++) {
        if (operadorPar && i % 2 == 0) {
            genesResultantes[i] = genesSegundoCromosoma[i];
        }
        if (!operadorPar && i % 2 != 0) {
            genesResultantes[i] = genesSegundoCromosoma[i];
        }
    }
    cromosomaResultante.setGenes(genesResultantes);
    return cromosomaResultante;
}

```

- **Uniforme:** Este operador cruza dos cromosomas, obteniendo dos cromosomas hijos, obteniendo cada gen de cada cromosoma hijo del gen del padre o de la madre aleatoriamente.

4	2	1	3	0
---	---	---	---	---

1	3	4	2	0
---	---	---	---	---



4	3	4	3	0
---	---	---	---	---

4	3	1	2	0
---	---	---	---	---

```

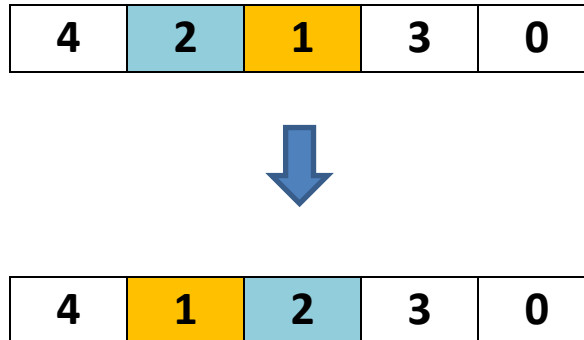
public IChromosome operate(final IChromosome primerCromosoma, final IChromosome segundoCromosoma)
    throws InvalidConfigurationException {
    IChromosome cromosomaResultante = (IChromosome) primerCromosoma.clone();
    Gene[] genesSegundoCromosoma = segundoCromosoma.getGenes();
    int tamanyoCromosoma = primerCromosoma.size();
    Gene[] genesResultantes = cromosomaResultante.getGenes();
    for (int i = 0; i < tamanyoCromosoma; i++) {
        boolean genSegundoCromosoma = getConfiguration().getRandomGenerator().nextBoolean();
        if (genSegundoCromosoma) {
            genesResultantes[i] = genesSegundoCromosoma[i];
        }
    }
    cromosomaResultante.setGenes(genesResultantes);
    return cromosomaResultante;
}

```


3.4. Operadores de Mutación

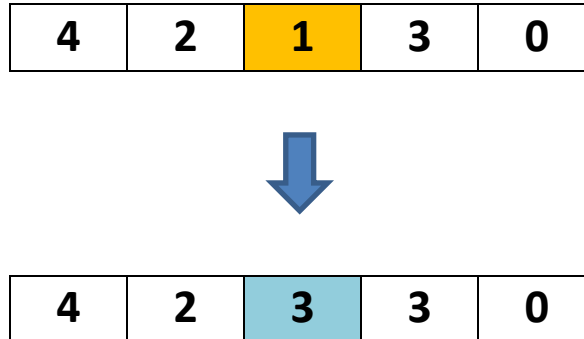
Se ha desarrollado dos operadores de mutación, que exponemos a continuación:

- **Permutación:** Este operador intercambia un gen por el gen de su posición anterior. Si se da el caso de que el primer gen muta, se intercambia con el último.



```
public IChromosome operate(final IChromosome cromosomaOriginal) throws InvalidConfigurationException {
    IChromosome cromosomaResultante = (IChromosome) cromosomaOriginal.clone();
    int tamanyoCromosoma = cromosomaResultante.size();
    Gene[] genesResultantes = cromosomaResultante.getGenes();
    boolean haMutado = false;
    int indiceInicio = getConfiguration().getRandomGenerator().nextInt(2);
    for (int i = indiceInicio; i < tamanyoCromosoma; i = i + 2) {
        boolean mutar = getConfiguration().getRandomGenerator().nextInt(porcentajeMutacion)==0;
        if (mutar) {
            int indiceGenAnterior = 0;
            if (i == 0) {
                indiceGenAnterior = tamanyoCromosoma - 1;
            } else {
                indiceGenAnterior = i - 1;
            }
            Gene genAnterior = genesResultantes[indiceGenAnterior];
            genesResultantes[indiceGenAnterior] = genesResultantes[i];
            genesResultantes[i] = genAnterior;
            haMutado = true;
        }
    }
    if (!haMutado) {
        cromosomaResultante = null;
    } else {
        cromosomaResultante.setGenes(genesResultantes);
    }
    return cromosomaResultante;
}
```

- **Uniforme:** Este operador intercambia un gen por otro obtenido aleatoriamente entre los posibles valores.



```

public IChromosome operate(final IChromosome cromosomaOriginal) throws InvalidConfigurationException {
    IChromosome cromosomaResultante = (IChromosome) cromosomaOriginal.clone();
    int tamanyoCromosoma = cromosomaResultante.size();
    Gene[] genesResultantes = cromosomaResultante.getGenes();
    boolean haMutado = false;
    for (int i = 0; i < tamanyoCromosoma; i++) {
        boolean mutar = getConfiguration().getRandomGenerator().nextInt(porcentajeMutacion)==0;
        if (mutar) {
            genesResultantes[i].setAllele(getConfiguration().getRandomGenerator().nextInt(tamanyoCromosoma));
            haMutado = true;
        }
    }
    if (!haMutado) {
        cromosomaResultante = null;
    } else {
        cromosomaResultante.setGenes(genesResultantes);
    }
    return cromosomaResultante;
}

```

Además de estas implementaciones, JGAP dispone de operadores ya implementados por defecto, en este proyecto utilizaremos los siguientes:

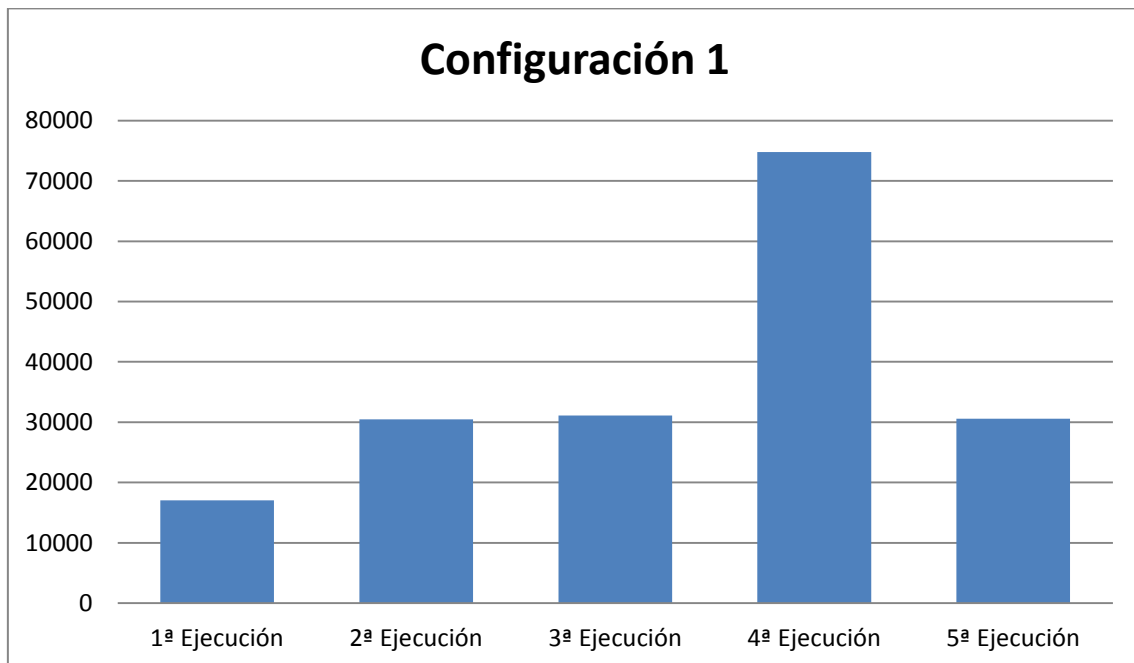
- **Average:** Operador de cruce. No es más que un operador de cruce por puntos.
- **Gaussian:** Operador de mutación, Se le aplica una distribución normal.
- **Tournament:** Operador de selección. Se realizan torneos entre algunos cromosomas escogidos al azar. El ganador de cada torneo es seleccionado para el cruzamiento.
- **Best Chromosomes:**
 - Operador de selección. Reproducción elitista, un porcentaje de los cromosomas más aptos son seleccionados para ser cruzados, el resto se selecciona al azar.
 - Operador de reproducción. Selección elitista, un porcentaje de los cromosomas más aptos pasan directamente a la siguiente generación, el resto se selecciona al azar. Es el operador de reproducción en este proyecto.

4. Estudio comparativo de distintas configuraciones

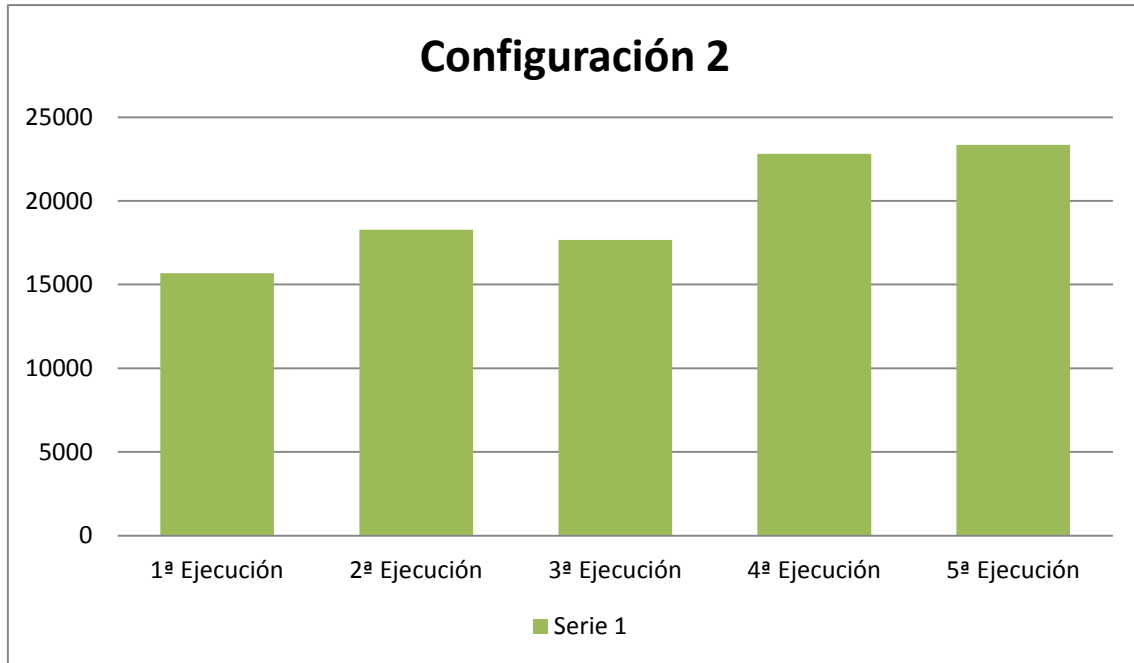
En este apartado hemos realizado un estudio comparativo del número de generaciones hasta encontrar una solución, con distintas configuraciones. En total, hemos realizado 5 ejecuciones para encontrar una solución con 50 Reinas y se ha aumentado el límite de generaciones a 100.000.

Hemos seleccionado varias configuraciones, que por sus características quizás son más adecuadas entre ellas:

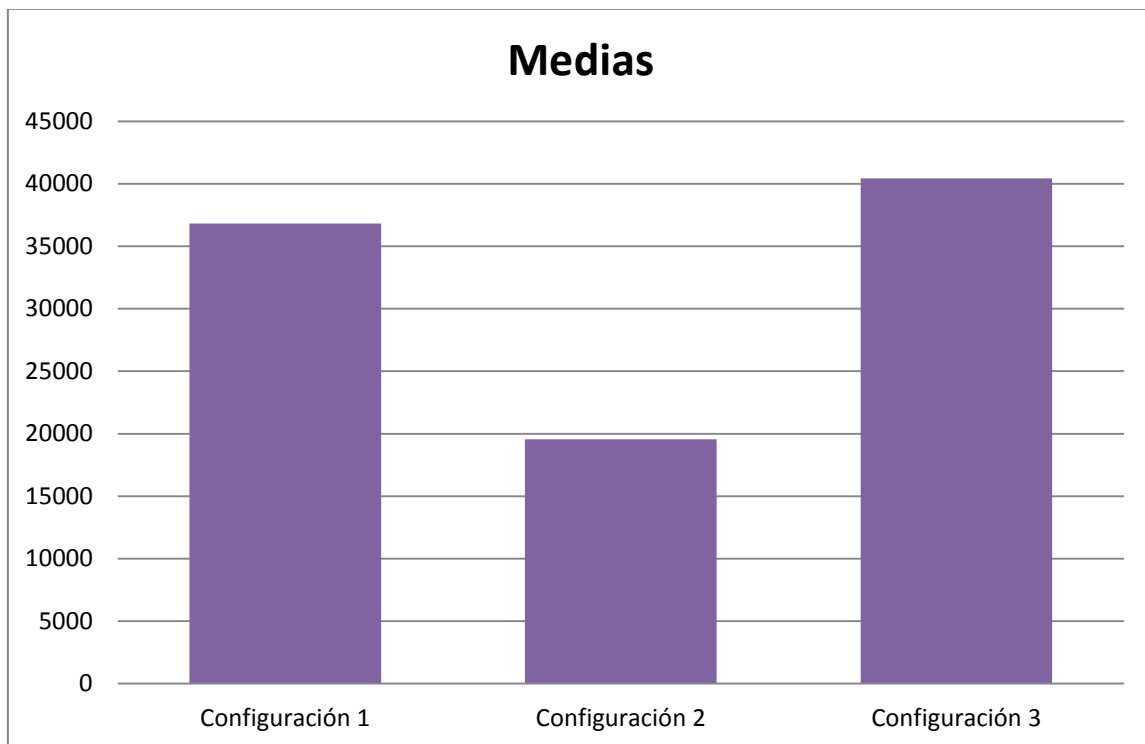
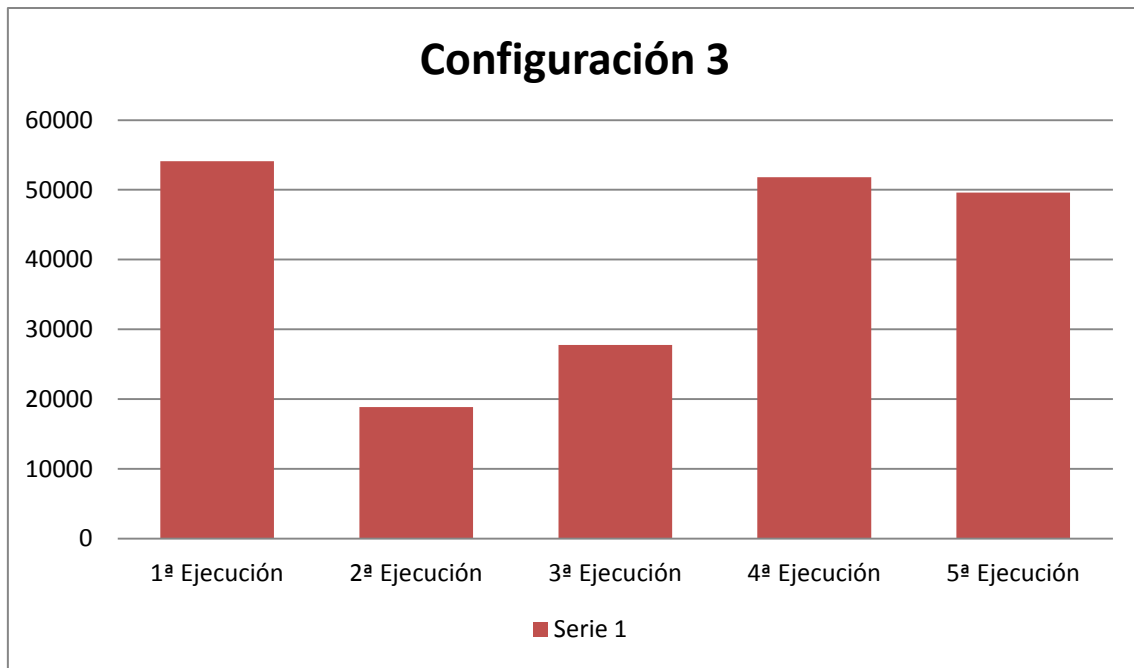
- Configuración 1 (Por defecto):
 - Operador Cruce: Average
 - Operador Mutación: Gaussian
 - Operador Selección: Tournament
 - Operador Reproducción: Best Chromosomes
 - Probabilidad Cruce: 0.8
 - Probabilidad Mutación: 0.2
 - Número Individuos: 50

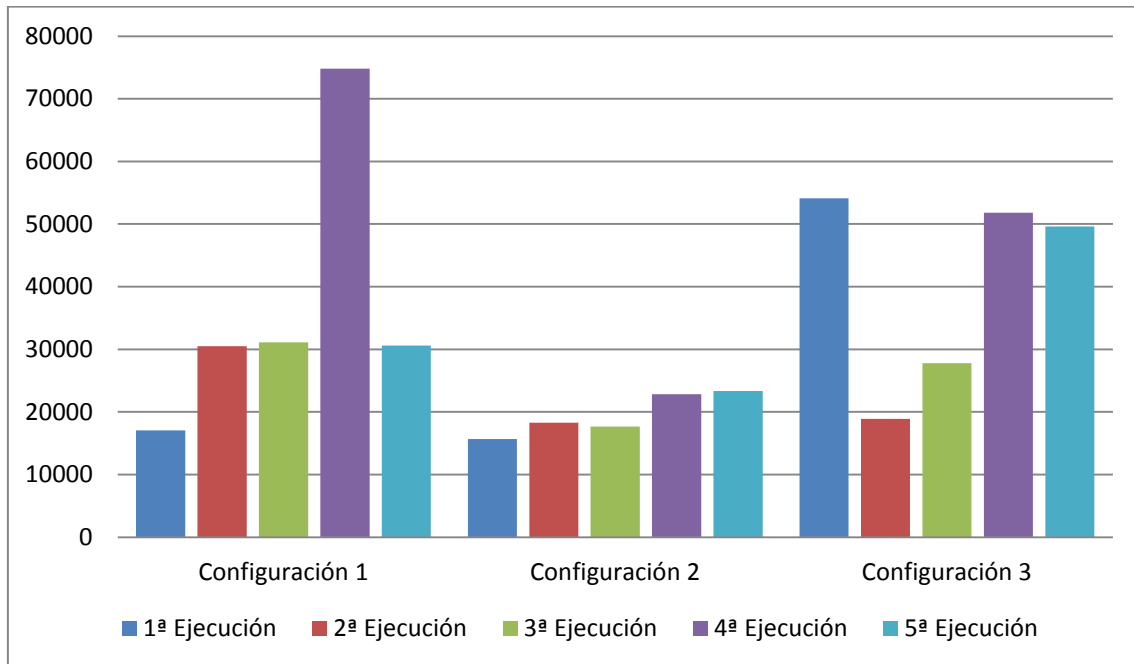


- Configuración 2 (Orientada a las permutaciones):
 - Operador Cruce: ParImpar
 - Operador Mutación: Permutación
 - Operador Selección: Tournament
 - Operador Reproducción: Best Chromosomes
 - Probabilidad Cruce: 0.8
 - Probabilidad Mutación: 0.2
 - Número Individuos: 50



- Configuración 3 (Orientada al azar):
 - Operador Cruce: Uniforme
 - Operador Mutación: Uniforme
 - Operador Selección: Tournament
 - Operador Reproducción: Best Chromosomes
 - Probabilidad Cruce: 0.8
 - Probabilidad Mutación: 0.2
 - Número Individuos: 50





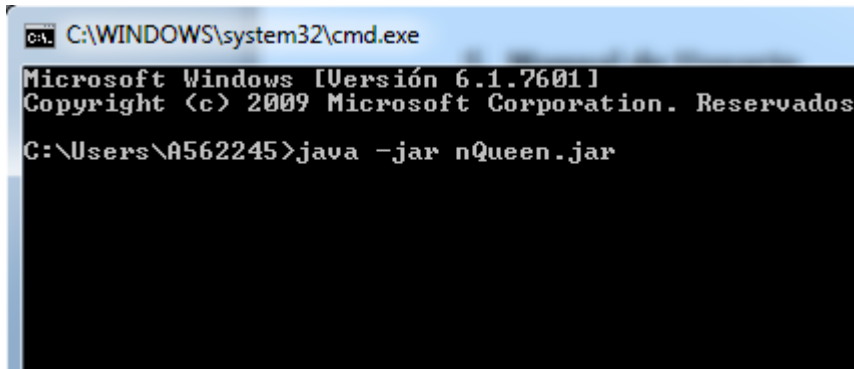
Como muestra las gráficas, la configuración 2 es la que menos generaciones de media ha necesitado para encontrar una solución.

La interpretación más directa a la que llegamos, es que las soluciones a los problemas de N Reinas no son más que permutaciones y esta configuración es la que más se acerca a realizar permutaciones. En el caso de la configuración 3, la que peores datos obtiene, se comporta de forma bastante aleatoria, por lo que es más probable que genere cromosomas con valores repetidos, que como sabemos, no son solución.

5. Manual de Usuario

- Para arrancar la aplicación, ejecutar por consola de comandos el siguiente comando:

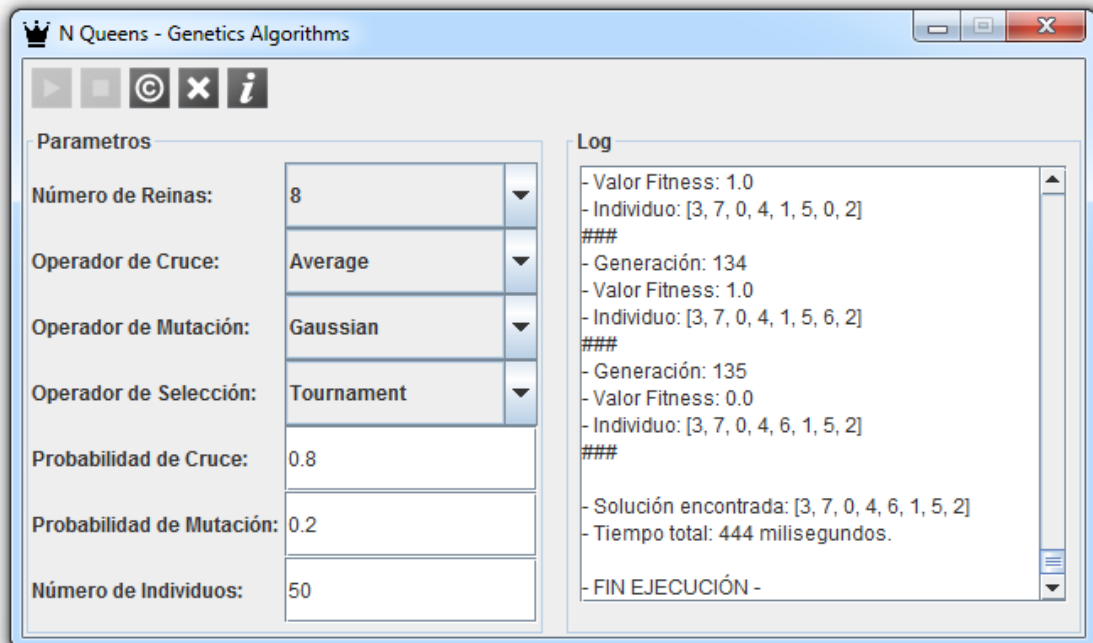
java -jar nQueen.jar



```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados
C:\Users\A562245>java -jar nQueen.jar
  
```

- Para realizar una ejecución:
 - 5.1. Seleccionar los parámetros.
 - 5.2. Pulsar sobre el botón Setup para cargar la configuración seleccionada.
 - 5.3. Pulsar sobre el botón Run para comenzar la ejecución.



Bibliografía

- Tutorial sobre JGAP: <https://www.adictosaltrabajo.com/tutoriales/jgap>
- Problema 8 Reinas: https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas
- Algoritmos Genéticos: https://es.wikipedia.org/wiki/Algoritmo_genetico
- Página oficial JGAP: <http://jgap.sourceforge.net/doc/tutorial.html>