UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Algoritmo genético para la generación automática de equipos de trabajo en entornos educativos

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Miguel Ángel Chacón Martínez

Tutor: Julian Inglada, Vicente Javier
Elena del Val
Juan M Alberola

Curso 2017-2018

Resumen

Es algo importante y necesario el trabajo en equipo en la sociedad moderna. Sin embargo, la creación de un equipo de trabajo no es tarea fácil para los humanos por la gran cantidad de posibilidades que se nos aparece. Este proyecto intenta solucionar el problema de crear equipos balanceados mediante el diseño, desarrollo e implantación de un algoritmo genético. Durante el mismo se llevarán a cabo simulaciones en el que se demuestra la viabilidad para conseguir resultados buenos para el mundo real en un tiempo muy reducido.

Palabras clave: algoritmos genéticos, formación de equipos, educación, inteligencia artificial

Abstract

Teamworking is a special and important in this times. Unfortunally, the creation of a good team is not an easy task for humans, because the huge quantity of possibilities. In this project, we are going to fix this problem using genetic algorithms. During the same simulations will be carried out in which the viability is demonstrated to obtain good results for the real world in a very reduced time.

Key words: genetic algorithms, team formation, artificial intelligence

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

En el mundo que nos encontramos, es necesario trabajar con otras personas en el día a día con el fin de lograr nuestros proyectos y metas. Es por ello que se considera el trabajo en equipo como una disciplina muy importante en la educación superior [2].

En este trabajo se trata la problemática de crear equipos balanceados de trabajo en el ámbito educativo. Los equipos más eficientes están compuestos por equipos multidisciplinarios, donde cada miembro ejerce un rol distinto en este [5]. No obstante, formar equipos balanceados no es tarea sencilla por su elevado número de posibilidades.

$$\frac{(n \cdot m)!}{(n!)^m \cdot m!}$$

n = num alumnos por equipo
m = num de equipos

Supongamos que queremos generar 10 equipos de 3 personas,
existen 1.208.883.745.669.600.000 configuraciones distintas. Una cantidad totalmente intratable, resolver el problema de forma óptima se hace pues imposible.

Durante este trabajo, se usarán algoritmos genéticos para buscar una solución aceptable al problema en un tiempo razonable. Los algoritmos genéticos se aprovechan del mecanismo natural de la evolución para generar soluciones a partir de otras existentes.

1.1 Objetivos

El objetivo es la investigación, diseño e implementación de una solución en una aplicación que genere equipos heterogéneos. Dicha aplicación recibirá una entrada que constará del número de equipos a formar y la composición de la clase según el rol.

Una vez correctamente implementado, se analizará y se valorará la viabilidad de la solución así como su eficiencia.

Como objetivo segundario, se busca conseguir tiempos muy reducidos incluso para clases muy grandes (800 personas). La calidad de la solución obtenida debe ser muy buena, teniendo poca diferencia con la solución óptima.

1.2 Metodología

Para la realización de este proyecto se usará una metodología tradicional en cascada. Se compone de varias fases:

- **Análisis:** Se enumeran y se valoran todas las posibilidades de diseño.
- **Diseño:** Se valora y se decide que posibilidad usaremos, en los casos que dudemos se usará la experimentación empírica para valorar la viabilidad.
- **Implementación:** Se implementará la solución, evitando reescribir la rueda siempre que sea posible.
- **Valoración:** Se valoran los resultados obtenidos, un juicio negativo de estos lleva a retroceder una etapa en la metodología

Se empezará primero con el más bajo nivel, y se seguirá iterando hacia arriba hasta terminar la aplicación.

1.3 Estructura de la memoria

En este apartado se describe brevemente el contenido de los diferentes apartados de la memoria:

Capítulo I: Introducción
En este apartado se describe los objetivos y características principales del proyecto, así como la distribución de los capítulos.

Capítulo II: Antecedentes
En este apartado se presentarán los antecedentes teóricos del trabajo y se explican brevemente que son los algoritmos genéticos.

Capítulo III: Diseño e implementación del algoritmo genético
En este apartado se tratará del diseño e implementación de un algoritmo genético adecuado que resuelva el problema de generación de grupos de forma óptima.

Capítulo IV: Implementación
Implementación del trabajo. Se describirá la estructura software que se ha seguido para desarrollar la aplicación.

Capítulo V: Evaluación del sistema
En este capítulo se evalúa el rendimiento de la aplicación final y la calidad de sus soluciones.

Capítulo VI: Conclusiones En este apartado se hacen valoraciones sobre el resultado del proyecto.

Capítulo VII: Bibliografía
En este apartado se mencionan las referencias usadas para el desarrollo del proyecto.

CAPÍTULO 2
Antecedentes

A continuación, se describirá que soluciones han dado otros autores al problema en cuestión y como funcionan la técnica de los algoritmos genéticos.

2.1 Estado del Arte

En [1] se plantea y se usa un algoritmo genético para la creación de equipos de trabajo en entornos educativos. Sin embargo, las conclusiones sobre la mejor configuración del algoritmo son distintas a las obtenidas aquí. Esto se debe a que el autor ha probado soluciones demasiado perfectas, con tamaños de prueba pequeños (24-48) y una solución óptima conocida. En este trabajo se estudiarán problemas más grandes (hasta 400 alumnos) con una configuración generada aleatoriamente donde a menudo no existirá solución perfecta, además intentaremos razonar si cambiar el algoritmo de selección tiene impactos en su rendimiento.

En [12] se busca formar grupos de personas con determinadas características, tales como la edad o la disciplina universitaria. Para ello forma una estructura de grafo, donde cada persona está asociada a un rol determinado (ej: rango de edad), la distancia de grafo se usará pues para establecer la independencia que existen entre dos personas ([12, pag 34])

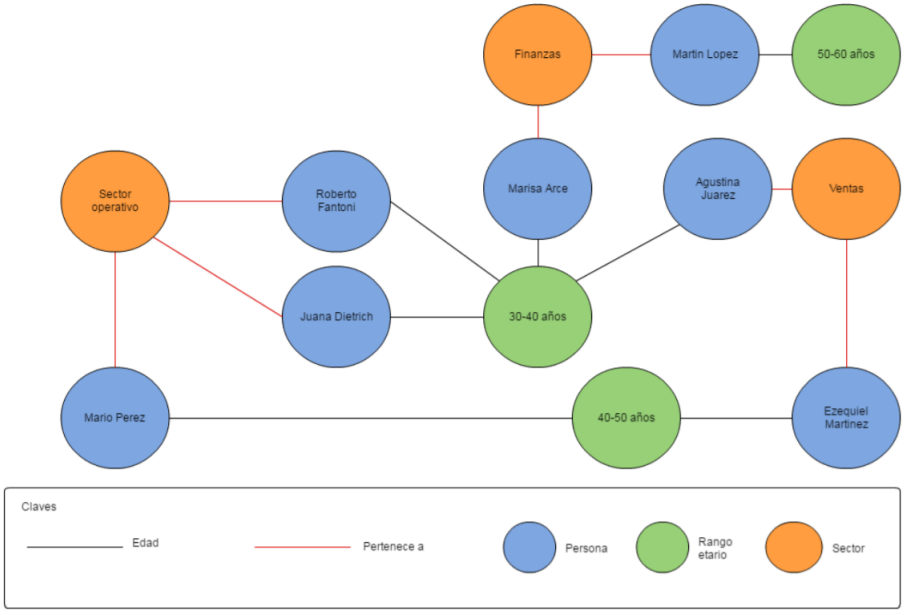

Figura 2.1: Arquitectura de grafo [12, pag 34]

El objetivo de ese trabajo es formar equipos que cumplan unas características dadas. Lo cual provoca que el trabajo sea generalista, en este trabajo nos vamos a centrar principalmente en formar equipos balanceados.

En [7] , los autores proponen crear una aplicación web donde los profesores puedan generar grupos de trabajo de manera automática. Para ello usan un algoritmo de programación lineal (CPLEX) que encuentra la solución óptima, pero a expensas de un coste muy elevado. En este trabajo, usaremos un algoritmo genético para disminuir el coste temporal, aunque no garantizaremos que la solución sea óptima.

2.2 Algoritmos genéticos

Todo problema puede ser resuelto usando la fuerza bruta (probar todas las combinaciones) pero la gran cantidad de posibilidades puede hacer que tardemos horas, días, meses o incluso años. Los algoritmos genéticos fueron desarrollados para resolver problemas complejos de una forma automática y rápida.

Estos algoritmos se basan en el mecanismo natural de la evolución, de modo que los individuos más aptos sobreviven y pasan sus genes a la descendencia. Conforme van pasando generaciones, se van creando organismos más aptos y más capaces de hacer frente al problema.

Esta peculiaridad les da a los algoritmos genéticos una eficiencia muy buena, permitiendo resolver problemas que con una búsqueda exhaustiva sería imposible, por su alto coste computacional.

Un ejemplo curioso de su uso, [6] es en el diseño de coches deportivos, pues permiten obtener soluciones nuevas que a un humano no se les hubiera ocurrido.

2.2.1. Funcionamiento

Su funcionamiento se basa en el siguiente bucle:

1. **Codificación:** Se codifica la solución al problema en un vector. En el caso de este tfg, cada posición del vector podría corresponder a que equipo pertenece ese alumno.
2. **Población inicial:** Se genera aleatoriamente una población de N individuos, donde cada individuo es una solución al problema. El tamaño de la población depende del problema, aunque algunos autores sugieren un tamaño de m^2 , Donde m es la longitud del vector.
3. **Evaluación:** A cada individuo de la población, se le pasa una función de evaluación (fitness) que puntúa lo buena que es dicha solución. Se ordena a los individuos según ese puntaje.
4. **Selección:** Se seleccionan a los individuos aptos para reproducirse, existen muchos criterios y algoritmos de selección distintos. Los buenos tienen muy en cuenta el fitness para elegir pero también pueden coger individuos con menor puntaje para dar mayor variabilidad a la población.
5. **Cruce:** Se cruzan los individuos seleccionados, eso significa que se crean nuevos individuos a partir de una combinación de dos progenitores. Existen varios criterios/algoritmos sobre como hacer la mezcla, siendo el más sencillo coger la mitad izquierda de uno y la mitad derecha de otro.
6. **Mutación:** Se le da una probabilidad (baja) de que un individuo sufra una mutación, tal y como ocurre en la evolución natural. Esto significa que unas de las posiciones del vector, se verá alterada aleatoriamente.
7. **Criterio de parada:** Al final de la generación, se evalúa si es necesario parar. Si se detiene, se coge al mejor individuo y se ofrece como solución, sino se detiene, se vuelve al paso 2. Los criterios de paradas pueden ser varios, como número de iteraciones, tiempo, valor de fitness, convergencia del fitness, etc...

2.2.2. Ventajas e inconvenientes

Los algoritmos genéticos tienen varias ventajas:

- No requieren de ninguna información derivada del problema para funcionar. Esto permite que puedan evolucionar solos, llegando a encontrar soluciones contra-intuitivas que un humano no habría pensado.
- Ofrecen un buen rendimiento en tiempo, superando con diferencia a las búsquedas exhaustivas.
- Son any-time, esto significa que puedes pararlos en cualquier momento y siempre tendrás una solución al problema.
- Son paralelizables.
- Ofrecen una lista de soluciones al problema y no una única solución.

Pero también tienen algunas desventajas:

- Son muy propensos a quedarse atascados en máximos locales. Pudiendo en algunos casos dar soluciones muy lejanas al óptimo:

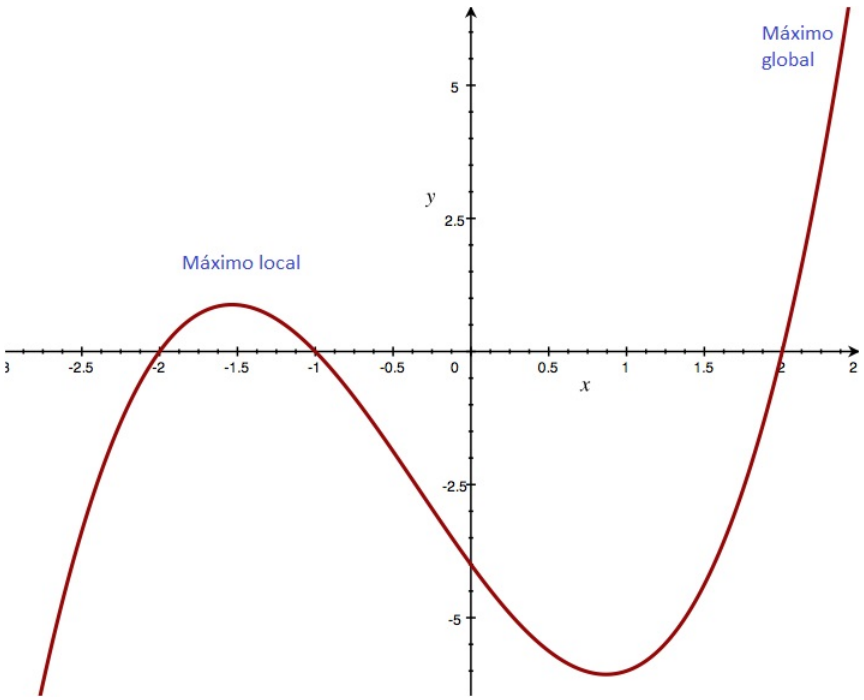

Figura 2.2: Máximo local y máximo global

  Los algoritmos genéticos han intentado paliar este problema trayendo más variabilidad a la población, usando técnicas como las mutaciones, aleatoriedad en la selección o simplemente con poblaciones más grandes.

- Requiere de un estudio para averiguar la configuración idónea del algoritmo para cada problema (codificación, índice de mutación, etc...). Mecanismos que funcionan bien en un problema no tienen porque funcionar bien en otros. Algunos autores [3] han intentado paliar este problema introduciendo la configuración del algoritmo genético dentro de la misma solución.

CAPÍTULO 3
Diseño del algoritmo genético

A continuación, se hará un estudio y se razonará como se implantará el algoritmo genético.

3.1 Entrada

La entrada del algoritmo corresponde a:

- Número de alumnos.
- Número de roles.
- Que rol posee cada alumno.
- Número de integrantes por equipo.

Se ha optado que la entrada sea definido por un fichero, con la siguiente estructura:

```
# ROL <name> <number of parameters> [<parameter name> <parameter value>]
# AGENT <name> <rol>
# CONFIGURATION <number of agents> <utility value> [<rol name>]
ROL r0 1 p0 0
ROL r1 1 p0 0
ROL r2 1 p0 0
ROL r3 1 p0 0
ROL r4 1 p0 0
ROL r5 1 p0 0
AGENT a0 r0
AGENT a1 r5
AGENT a2 r5
AGENT a3 r1
AGENT a4 r4
AGENT a5 r0
AGENT a6 r2
AGENT a7 r5
AGENT a8 r0
AGENT a9 r5
```

Al principio, se definen los roles, a los que se le asocia un identificador (r0, r1, r2...). A cada alumno se le asocia un identificador (a0, a1, a2...) y el rol que le corresponde.

3.2 Codificación

Es necesario codificar el problema en cuestión en un genoma. De forma que se pueda representar una solución en un vector.

Para este caso de estudio, se ha optado por usar un vector V de N enteros. Siendo N el número de alumnos, cada posicion del vector corresponde a que grupo pertenece, lo que también que

cada V[i] estará en el rango [0, M), siendo M el número de equipos.

Una solución es un vector de alumnos donde se indica a que equipo pertenece.

3.3 Función fitness

La función de fitness debe valorar la calidad de la solución, en este caso debe tener en cuenta las siguientes condiciones:

- El número de equipos creado debe corresponder con el número de equipos óptimo.
- El número de miembros por equipo debe ser balanceado y corresponder con el valor de la entrada.
- El número de roles distintos que contiene el equipo.

Esto nos lleva a la siguiente función de evaluación:

$$Fitness = \frac{gruposFormados}{gruposEsperados} + \frac{alumnosPorGrupo}{AlumnosEsperados} + \frac{rolesDistintos}{rolesEsperados}$$

Podemos acotar la función de fitness entre 0 y 1. De forma que el 1 sea la solución óptima y el 0 sea la peor solución.

Para ello, acotaremos las tres partes de la función fitness entre 0 e 1 y lo juntaremos con un producto:

$$Fitness = \frac{gruposFormados}{gruposEsperados} * \frac{alumnosPorGrupo}{AlumnosEsperados} * \frac{rolesDistintos}{rolesEsperados}$$

Esta será nuestra función fitness. Definiremos estas partes a continuación.

3.3.1. Balance de grupos

$$\frac{gruposFormados}{gruposEsperados}$$

Aquí se busca que el número de grupos formados coincida con el número de equipos esperados.

Por ejemplo, si son 60 alumnos y formamos equipos de 5. Una simple división dice que deberíamos formar 12 grupos.

Para valorar esto, usaremos la siguiente fracción [1, pag 41]

$$Score = 1 - \frac{|gruposFormados - gruposEsperados|}{numALumnos - gruposEsperados}$$

En el caso más óptimo, grupos formados equivaldrá al de esperados. Por lo que la fracción dará 0 y por tanto la puntuación será de 1.

Cuando el número de gruposFormados sea el máximo posible, significará que gruposFormados es equivalente al número de alumnos (grupos de 1). Por tanto la frac nos queda:

$$Score = 1 - \frac{numAlumnos - gruposEsperados}{numAlumnos - gruposEsperados}$$

Esta fracción se puede simplificar con un 1, lo que significa que la puntuación final será de 0.

Cualquier otro caso intermedio, tendrá una puntuación entre 0 y 1, dependiendo de como se aleje del número de gruposEsperados.

3.3.2. Integrantes por grupo

Este problema implica que el número de integrantes por equipo debe ser lo más similar al esperado, si estamos haciendo equipos de 4, penalizar los equipos de 2 y 5 miembros.

Este problema es similar al del apartado anterior, y por tanto se puede resolver de la misma forma:

$$Score = 1 - \frac{|estudiantesPorGrupo - estudiantesEsperados|}{numALumnos - estudiantesEsperados}$$

La única diferencia, esque aquí va a existir N resultados (uno por equipo). Que debemos juntar con una multiplicación.

3.3.3. Balance de roles

En esta sección, buscamos que el número de roles distintos en el equipo sea equivalente con lo que esperamos. Si tenemos 4 miembros por equipos y 4 roles, buscaremos que sean todos de un rol distinto. Si en cambio tenemos 4 miembros por equipo y 3 roles, buscaremos que estén al menos los 3 roles representados.

Se resuelve con la misma metodología que antes:

$$Score = 1 - \frac{|rolesPorGrupo - rolesEsperados|}{numALumnos - rolesEsperados}$$

Donde rolesEsperados = min(estudiantesPorGrupo, estudiantesPorGrupo / numRoles)

3.4 Población

El tamaño de la población es un factor importante. Si el número es muy escaso, la población tendrá poca variabilidad y no podrá evolucionar de forma óptima.

Si el número es muy elevado, el coste computacional por generación crece rápidamente, así como su consumo de memoria y el número de generaciones necesarias para que converja [8, p. 4]. Durante las pruebas, se probó crear 1000 equipos en una clase de 3000 alumnos y una población de 500.000, el consumo de memoria superó los 6 GB, lo que obligó a cancelar la operación.

Averiguar el tamaño justo no es una operación sencilla, algunos autores [4] sugieren usar una población $genoma^2$. Donde genoma es el número de genes que contiene el genoma.

Más adelante se determinará el proceso que usaremos para determinar cual es el tamaño de población más idóneo.

3.5 Selección

La selección es el algoritmo que se usará para elegir los individuos aptos para reproducirse. Se valoran los siguientes algoritmos:

- **Tournament:** [9] Agrupa los individuos de la población en grupos de k individuos. Los miembros de cada grupo compiten entre sí y el ganador es elegido para reproducirse. Para determinar el ganador, se ordenan los miembros del grupo de mayor a menor fitness. El primero tiene la mayor probabilidad de reproducirse, el segundo menos probabilidad, el tercero aún menos, etc.... De elegir una selección por torneo, debemos decidir también cual debe ser el tamaño del grupo.

- **Truncation:** Se ordenan a los individuos por su valor de fitness y se cogen a los K primeros para reproducirse. El valor de k podría corresponder incluso a la población entera. Implicando que no hay selección alguna y todos se reproducen.
- **RouletteWheelSelector:** [10] Se consideran a todos los individuos comom segmentos y se colocan a todos en línea. La longitud de cada segmento va acorde a su valor de fitness, cuanto más grande más largo. Se coge un número aleatorio, el individuo que ocupe esa posición en el segmento es elegido para reproducirse. El proceso se repite hasta que se ha cogido a todos los padres que se quieren.
- **LinearRankSelector:** [10] Se cogen a todos los individuos como segmentos y se colocan todos en línea. Pero a diferencia del algoritmo anterior, el tamaño de cada segmento no va acorde a su valor de fitness sino a su posición en el ranking. El individuo que tiene el mayor fitness ocupa el primer puesto y por tanto su tamaño es el más grande, el individuo con el segundo mayor fitness tiene el segundo tamaño más grande, y así sigue. El reparto de los tamaños es lineal.
- **ExponentialRankSelector:** Una variante del mecanismo anterior, de forma que la distribución de los tamaños es exponencial conforme a su rango.
- **BoltzmannSelector:** Es un selector probabilístico, que determina la probabilidad de cada individuo (i) acorde con:

$$P(i) = \frac{e^{bf(i)}}{\sum_{j=1}^{n} e^{f(j)}}$$

  Donde b es una constante que controla intensidad de la selección. F(i) corresponde con el valor de fitness del individuo i.
- **StochasticUniversalSelector:** [11] Es una variante del RouteletteWheelSelector con el fin de ser más "justoçon individuos con un fitness más abajo. Se selecciona un número aleatorio (generalmente pequeño), a partir de ahí se va avanzando secuencialmente en el segmento y seleccionando a los elementos.
- **Elitism:** En este algoritmo, se seleccionan los k mejores miembros de la población y se copian para la siguiente generación. La población se regenera por completo duplicando estos individuos o volviendola a generar aleatoriamente.

En adelante, determinaremos cual es el mejor algoritmo de selección que podemos usar en este problema.

3.6 Cruce

Existen varios algoritmos de cruce utilizados:

- **SinglePointCrossover:** En este algoritmo, se parte el genoma de cada padre en 2 trozos, cada hijo recibirá un trozo de un padre.

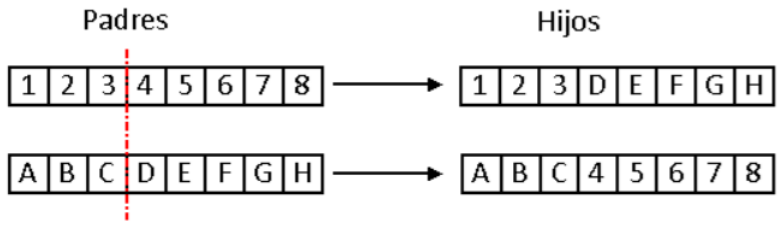

Figura 3.1: Cruces normal entre 2 padres

- **MultiPointCrossover:** En este algoritmo, partimos el genoma en varias partes. De forma que cada hijo está compuesto de varios trozos:

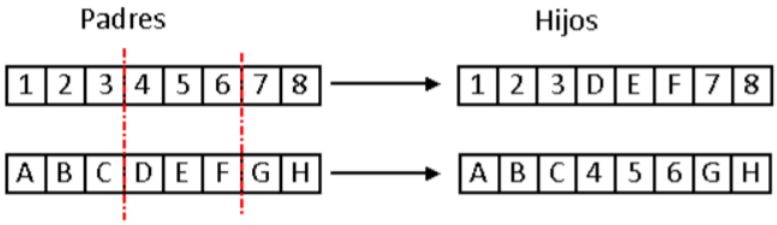

Figura 3.2: Cruce multiparte entre dos padres

- **LinearCrossover:** [13] En este algoritmo, se cogen los dos vectores y crean el vector hijo mediante una combinación lineal de estos.
- **UniformCrossover:** Los genes se cogen arbitrariamente de cada padre conforme a una probabilidad dada p. Supongamos que los padres A y B tienen una probabilidad 0,5. Por cada gen que haya que cruzar, se lanza üna moneda al aire", dependiendo del resultado se coge el gen un padre u otro.

Nuevamente, decidiremos más adelante que algoritmo de cruce usaremos.

3.7 Mutación

Se entiende como mutación, la probabilidad que un gen cambie aleatoriamente. En el diseño de un algoritmo genético, hay que determinar la probabilidad p que hará que un gen mute.

Es necesario elegir un valor adecuado. Valores muy bajos (o incluso cero) convergen más rápido, pero es muy probable que se quede atascado en máximos locales. Valores de mutación muy altos requieren de más tiempo para que el algoritmo converja, [14] cuando el valor es excesivamente alto, el algoritmo genético acaba convirtiéndose en una simple búsqueda aleatoria.

El valor óptimo depende del problema en cuestión, decidiremos más adelante que valor usaremos.

3.8 Condición de parada

Existen varios criterios de parada que se pueden usar para detener el algoritmo:

- Un número máximo de K generaciones.
- Un tiempo máximo de ejecución.
- Cuando pasen K generaciones sin que el fitness mejore.
- Cuando el fitness alcance un determinado valor.

Se ha decidido usar el primero criterio, con una K = 300. Este número sera personalizable desde la interfaz.

3.9 Determinación de los parámetros adecuados

Disponemos de varios parámetros que debemos determinar:

- Algoritmo de selección, se han barajado hasta 8 algoritmos distintos.
- Parámetro para configurar el algoritmo de selección, como por ejemplo la probabilidad, el tamaño K de un torneo, etc...
- Algoritmo de cruce utilizado. Hasta 4 algoritmos distintos.
- Parámetro para configurar el algoritmo de cruce, por ejemplo en un cruce multipunto sería el número de cortes realizado.
- Índice de mutación.
- La proporción de población necesaria para conseguir los mejores resultados.

Para determinar estos parámetros, ejecutaremos un algoritmo genético sencillo. Su genoma está compuesto de un vector de 28 bits que determina la configuración del algoritmo genético, de los cuales:

- Los tres primeros bits determinan cual de los 8 selectores se usará
- Los tres siguientes bits determinan el parámetro del selector usado. Cada selector tiene una lista de 8 argumentos distintos para probar, por ejemplo en el linearrankselector están las probabilidades entre 0 y 1 (1.0, 0.8, 0.6, 0.4, 0.2...).
- Los dos siguientes bits determinan el algoritmo de cruce a utilizar.
- Los tres siguientes bits determinan los parámetros del algoritmo de cruce.
- Los tres siguientes bits determinan la mutación, hasta 8 valores distintos (0.15 , 0.1, 0.05, 0.025, 0.01, 0.005, 0.0025..)
- Los 12 siguientes bits determinan el multiplicador de la población, el valor máximo es 4096. Este número se divide entre 100 y nos da un multiplicador de población de 40,96 (40 veces el tamaño del genoma), es posible también obtener valores menores a 1 con este método.

La función de fitness consiste en ejecutar esta configuración con una batería de 6 casos de prueba, cada caso se ejecuta 2 veces lo que da un total de 12 ejecuciones. Devolveremos la media de los fitness obtenidos como su valor:

```
File directory = new File("examples");

// Para cada caso de prueba
for (File file : directory.listFiles()) {
    Genetic genetic = new Genetic(new FileInputStream(file));

    // Selecciona la poblacion
    genetic.setPopulation((int)Math.round(numAgents()*multiplicator));

    // Selecciona el selector
    genetic.setSelector(selector, selectorParam);

    // Selecciona el algoritmo de cruce
    genetic.setCrossover(cross, crossParam);

    // Selecciona el algoritmo de mutacion
    genetic.setMutation(mutation);

    // Condicion de parada: 1 segundo de ejecucion
    genetic.setTimeLimit(1000);

    double miniFitness = 0.0;
    for (int i = 0; i < 2; i++) {
        double value = genetic.run();
        miniFitness += value;
        cnt++;
    }
    fitness += miniFitness;
}

return fitness/cnt; // Devolver la media
```

Se procede pues, a lanzar este algoritmo durante un largo período de tiempo. Estos son los mejores resultados obtenidos:

Tabla 3.1: Ranking de resultados usando diversos metodos

Selector	Selector Arg	Cross	Cross Arg	Mutacion	Mult pobl	Fitness
Truncation	50	Uniform	0.35	0.025	9.12	0.95284
ExponentialRank	0.3	Uniform	0.5	0.025	22.26	0.94655
Truncation	10	Uniform	0.5	0.1	32.81	0.94143
Tournament	9	Uniform	0.35	0.025	18.79	0.93828
ExponentialRank	0.9	Uniform	0.35	0.05	31.65	0.93762
ExponentialRank	0.8	Uniform	0.5	0.01	25.55	0.93735
Truncation	35	Uniform	0.5	0.1	40.41	0.9340
ExponentialRank	0.4	Uniform	0.5	0.005	12.32	0.93337
Tournament	6	Uniform	0.35	0.005	39.53	0.93297
Tournament	7	Uniform	0.5	0.1	21.44	0.930846
ExponentialRank	0.5	MultiPoint	0.25	0.05	4.47	0.93069
Tournament	9	Uniform	0.25	0.05	7.6	0.93065
Truncation	300	Uniform	0.15	0.025	2.61	0.90354
Truncation	25	Uniform	0.2	0.0025	35.42	0.93028
Elite	7	Uniform	0.35	0.0025	18.6	0.929856
ExponentialRank	0.6	Uniform	0.35	0.0025	7.65	0.92638
ExponentialRank	0.8	Uniform	0.35	0.001	11.73	0.9258
Tournament	6	MultiPoint	0.25	0.01	11.37	0.92565
Boltzmann	8	Uniform	0.25	0.0025	12.2	0.925202
LinearRank	0.25	Uniform	0.15	0.0025	14.62	0.92498
Blotzmann	6	Uniform	0.5	0.001	7.44	0.92455
Tournament	6	Uniform	0.25	0.001	19.87	0.92391
Truncation	35	Uniform	0.2	0.1	9.28	0.923109
Tournament	6	Uniform	0.2	0.0025	33.35	0.92281
Truncation	75	Uniform	0.2	0.001	14.61	0.92277
Truncation	25	Uniform	0.25	0.0025	36.39	0.92245
Elite	6	Uniform	0.2	0.05	12.74	0.9220934
Elite	4	Uniform	0.5	0.005	25.3	0.921444
ExponentialRank	0.9	MultiPoint	0.15	0.1	17.74	0.92065
Truncation	10	Uniform	0.25	0.1	23.76	0.91825
Tournament	9	Uniform	0.25	0.0025	9.0	0.91799
Boltzmann	10	Uniform	0.5	0.0025	15.67	0.91472
ExponentialRank	0.4	Uniform	0.5	0.0025	15.67	0.91472
ExponentialRank	0.7	MultiPoint	0.1	0.15	18.05	0.91119
Boltzmann	0.5	Uniform	0.15	0.001	4.44	0.91308
Truncation	75	Uniform	0.1	0.01	16.75	0.91091
ExponentialRank	0.9	MultiPoint	0.25	0.0025	26.67	0.90906
ExponentialRank	0.3	Uniform	0.15	0.005	17.34	0.90904
Truncation	35	MultiPoint	0.25	0.15	39.97	0.9085655
Boltzmann	8	Uniform	0.2	0.0025	16.15	0.90704
Truncation	25	Uniform	0.05	0.0025	28.66	0.90505
Elite	6	MultiPoint	0.25	0.05	21.21	0.90482611
Elite	2	SinglePoint	0.25	0.025	22.68	0.9046
Tournament	2	Uniform	0.05	0.0025	31.79	0.904387
Tournament	8	Uniform	0.1	0.001	12.18	0.90327
Tournament	8	SinglePoint	0.25	0.05	18.96	0.901705

El significado de las columnas es el siguiente:

- **Selector:** Hace referencia al algoritmo de selección utilizado.
- **Selector Arg:** Hace referencia al parámetro que usa para alimentar el algoritmo de seleccion. Por ejemplo, en el algoritmo de selección por torneo, este corresponde con el número de participantes en cada torneo.
- **Cross:** Hace referencia al algoritmo de cruce utilizado.
- **Cross arg:** Hace referencia al argumento utilizado para construir el algoritmo de cruce.
- **Mutacion:** La tasa de mutación, entre 0 y 1.
- **Mult pobl:** El multiplicador de población, un 9.12 significa que el tamaño de la población es de 9,12 veces el tamaño de la clase.
- **Fitness:** El valor de fitness conseguido con esta configuración.

Analizando esta tabla, se sacan las siguientes conclusiones:

- Uniform es sin duda la que mejores resultados ofrece, esto puede ser debido a que el cruce uniforme hace menos probable que se formen equipos inconsistentes (equipos con personas de más o de menos). Si en cambio, hacemos un corte y juntamos, es muy posible que esto ocurra.
- En cuanto al parámetro del Uniform, aunque no se saca conclusiones claras parece tener a valores altos (0,3-0,5).
- El método de selección se inclina por tres métodos: Truncation, ExponentialRank, Tournament
- EL parametro de la selección no parece tener tanta importancia.
- Los índices de mutación son relativamente altos (1-10 %)
- No existen conclusiones sobre la población a priori. Si bien, se descanta por proporciones grandes (10-30 veces).

3.10 Solución propuesta

Teniendo en cuenta los resultados obtenidos, se ha optado por la siguiente solución:

- Método de selección: Tournament, pero se permitirá elegir al usuario.
- Método de cruce uniforme (0.5).
- Índice de mutación por defecto del 2,5 %
- Población de al menos 10 veces el tamaño de la muestra.

3.10.1. Valoración del esfuerzo

Para implementar la solución propuesta se requieren de al menos 15 horas de trabajo. Donde la mayor parte iría en la creación y prueba de la interfaz, debido a que la parte lógica del algoritmo genético ya está definida e implementada por la librería.

Con anterioridad, se han invertido 25 horas para el análisis, diseño y manejo de la librería de algoritmos genéticos.

3.10.2. Presupuesto

El coste total del proyecto se dividen entre costes humanos y materiales:

- **Materiales:** Un portátil con 8 GB de ram y cuádruple núcleo. Su precio ronda los 400 euros.
- **Humanos:** En torno a 40 horas de trabajo de un programador. Suponiendo un precio bruto de 15 euros/hora, esto implica un coste de 600 euros.

Por tanto el coste del proyecto es en torno a 1000 euros, pero tiene la ventaja de que no requiere de un mantenimiento exhaustivo. Por lo que son costes fijos que pueden rentabilizarse con el tiempo.

CAPÍTULO 4
Implementación

Para la implementación, se ha realizado una interfaz que reciba como entrada la configuración de la clase y responda con la solución idónea de mayor valor de fitness.

La tecnología usada será Java con la librería JavaFX para la interfaz y la librería Jenetics para la implementación del algoritmo genético. Como sistema de control de versiones se ha usado GIT.

La arquitectura de la aplicación se compone de dos módulos:

- **Capa interfaz:** Corresponde con todo lo que el usuario ve e interactúa con él. Incluye botones, gráficas, campos de texto, etc... Todo el código relacionado con la visualización vendrá aquí.
- **Capa lógica:** Corresponde con toda la lógica de negocio, la capa de interfaz interactúa con esta para hacerle peticiones y mandarle tareas. El código relacionado con el algoritmo genético vendrá aquí.

4.1 Capa interfaz

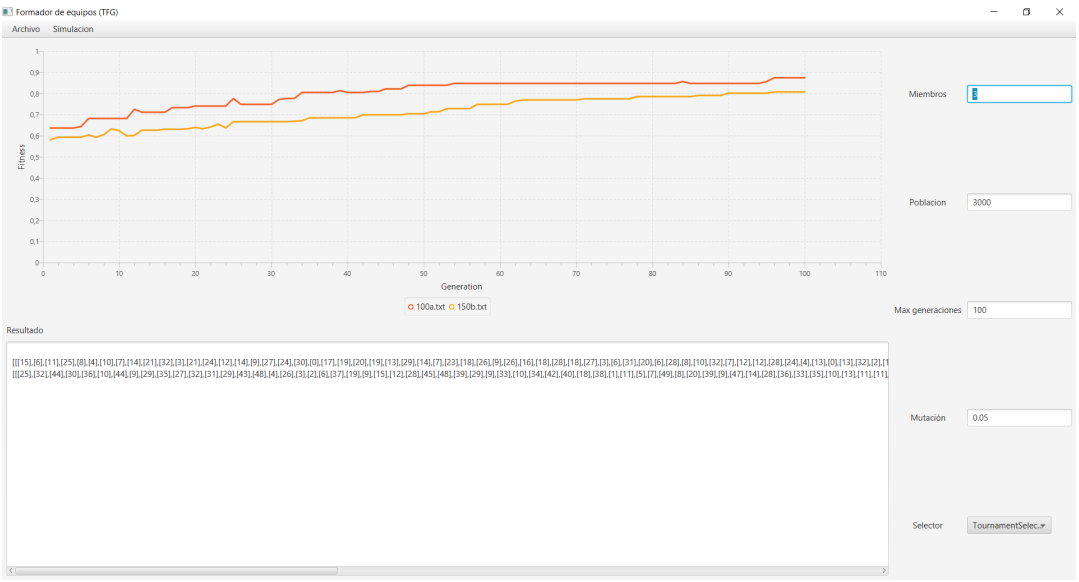

Figura 4.1: La aplicación para la generación de equipos

La interfaz de la aplicación se compone de una gráfica, que muestra la evolución del algoritmo genético, unos selectores de configuración, el resultado y un menú para las opciones. Es posible lanzar varias tareasr (ver figura 4.1).

Se sigue la metodología MVC (Modelo, Vista, Controlador) y se busca que la interfaz sea responsive. Es decir, que se pueda adaptar a cualquier pantalla.

4.1.1. Vista

La vista se compone de un fichero FXML, que representa la composición de la ventana de forma descriptiva. Este fichero utiliza el lenguaje XML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.chart.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>

<GridPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-
    Infinity" prefHeight="400.0" prefWidth="1000.0" xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1" fx:controller="chacon.miguel.tfg.gui.Controller"
    >
  <columnConstraints>

<!--- Mas codigo... --->

   </children>
</GridPane>
```

Funcionalmente, la interfaz se compone de varias rejillas (ver figura 4.2).

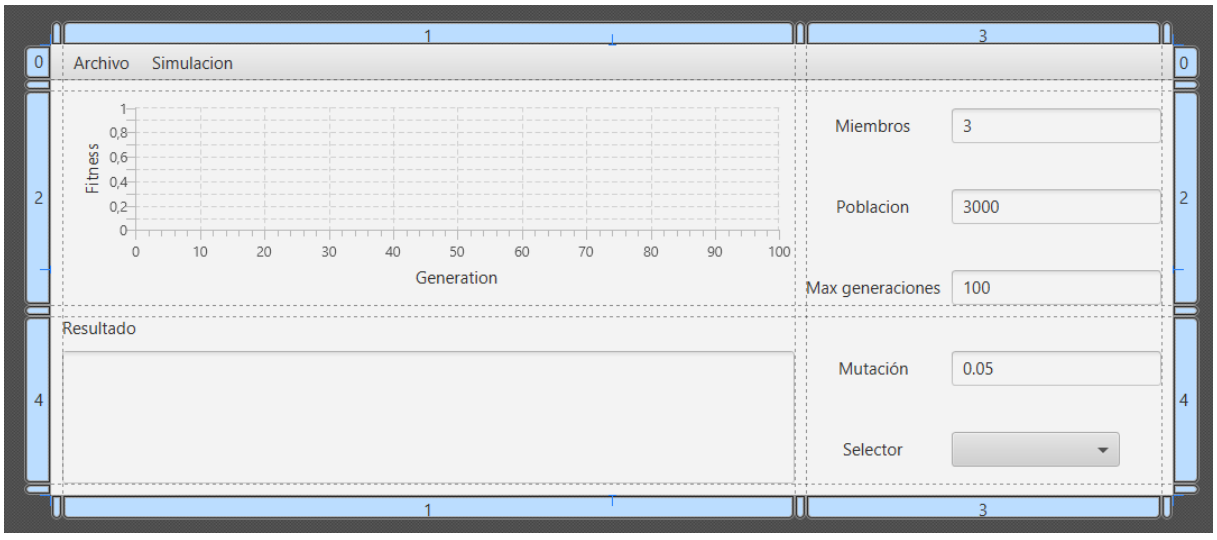

Figura 4.2: Composición de la interfaz por rejillas

En la figura se puede ver la rejilla padre. Arriba se encuentra la barra de menús. La gráfica ocupa un cuadrado de la interfaz así como el campo de resultado. Los controles de configuración se componen de otra rejilla adicional

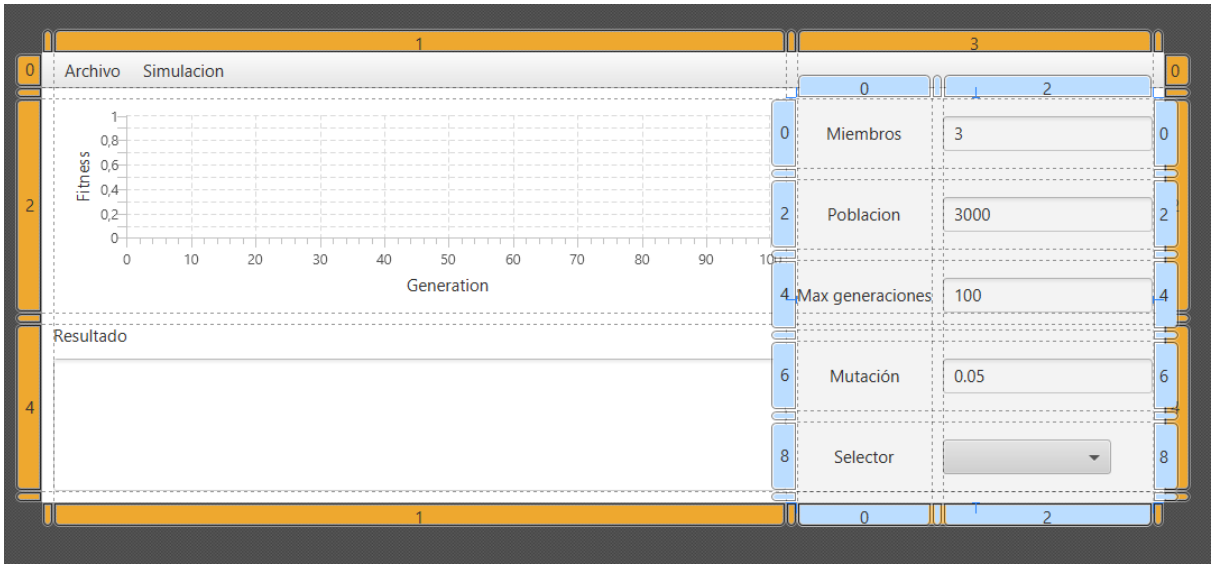

Figura 4.3: Composición del panel de la derecha

Mediante este sistema de rejillas, podemos conseguir una interfaz responsive, de forma que se pueda adaptar a cualquier tamaño de pantalla (ver figura 4.3).

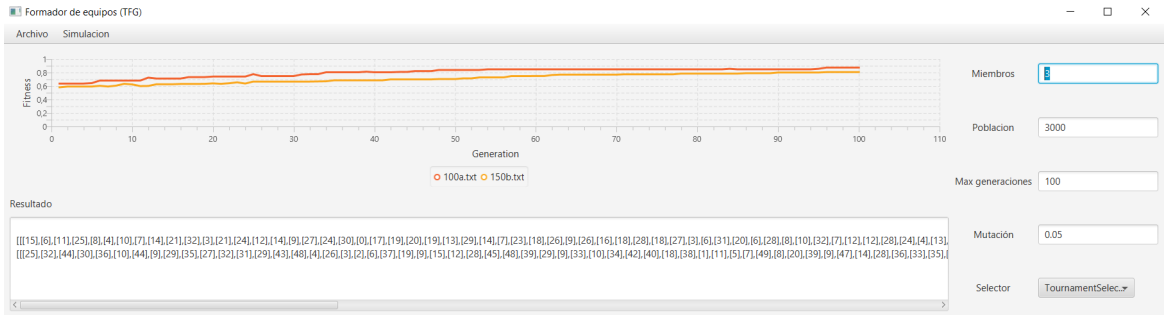

Figura 4.4: Vista en pantalla panoramica

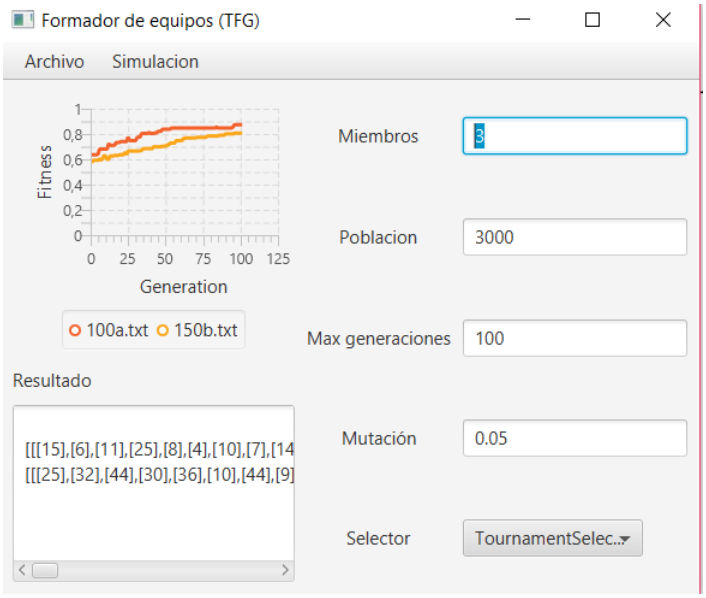

Figura 4.5: Vista en pantalla cuadrada

4.1.2. Controlador

Esta sección se compone de código java que interaccionará con la interfaz. Existen dos clases:

- **TFG:** Se encuentra el código de la función main. Aquí se inicia el programa, se lee el archivo de la vista y se lanza su controlador.
- **Controller:** Como su nombre indica, es el archivo que tiene el código del controlador de la vista.

La principal función es recibir los datos que se introducen en los campos y llamar a la capa lógica cuando corresponda.

Cuando el usuario añade un nuevo caso de prueba, se crea un nuevo hilo que llama a la capa de lógica para resolver el problema.

```
Thread thread = new Thread(()-> {
    try {
        Genetic genetic = new Genetic(new FileInputStream(file));
        genetic.setNumMembersTeam(numMembersTeam);
        genetic.setPopulation(population);
        genetic.setGenerations(generations);
        genetic.setMutation(mutation);
        genetic.setSelector(selector);
        genetic.setLiveStream((generation, fitness, bestPhenotype) -> notifyGeneration(
            series, generation,fitness, bestPhenotype));
        String result = genetic.run();
        notifyResult(result);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
});
thread.setDaemon(true);
thread.start();
```

Para actualizar la gráfica en tiempo real, se le pasa un manejador a la capa lógica. Cada vez que termina una generación, la capa lógica llama al manejador con el resultado que ha tenido (número de generación, valor de fitness..) y se actualiza la gráfica en consecuencia.

Por último, mencionar el generador de casos de prueba.

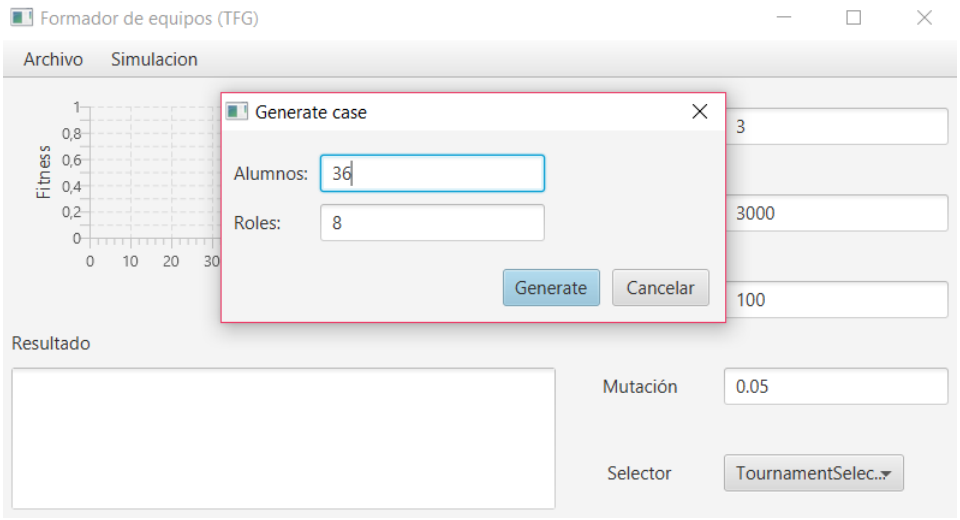

Figura 4.6: Dialógo de la interfaz de casos de prueba

Se compone de un dialogo, tras introducir los datos (ver figura 4.6) salta un dialogo de guardar archivo para que elijas donde quieres guardar el caso de prueba. Finalmente se hace una llamada a la capa lógica para que genere ese caso aleatorio.

4.2 Capa lógica

La capa lógica usa la librería Jenetics para realizar el algoritmo genético. Se compone de 5 clases:

- **CaseGenerator:** Generador de casos de prueba, recibe como parametros el número de alumnos, el número de roles y el stream de destino. Genera aleatoriamente y lo guarda en el stream de destino.
- **Fitness:** Clase que gestiona el fitness del algoritmo genético. Se construye pasándole el archivo que define la configuración. Su función más importante es eval(), que devuelve el valor de la función fitness para una genoma dado.
- **Genetic:** Clase principal del algoritmo genético. Construye la clase fitness, inicializa la librería y la arranca. Devuelve el resultado como un string.
- **LiveStream:** Una interfaz que contiene el método generationPassed, es usado por la capa lógica para notificar de los resultados de cada generación (por cual generación vamos, cual es el mayor valor de fitness, etc...).
- **LiveStreamProxy:** Clase auxiliar que, transforma el livestream propio que usa la librería jenetics a la interfaz que he definido previamente.

4.2.1. CaseGenerator

La clase CaseGenerator es breve y concisa, únicamente construye un fichero a través de datos aleatorios.

```
1  package chacon.miguel.tfg.logic;
2
3  import java.io.IOException;
4  import java.io.OutputStream;
5  import java.io.PrintWriter;
6  import java.util.Random;
7
8  public class CaseGenerator {
9      private int students;
10     private int roles;
11     private OutputStream out;
12
13     public CaseGenerator(int students, int roles, OutputStream out)
14     {
15         this.students = students;
16         this.roles = roles;
17         this.out = out;
18     }
19
20     public void save() throws IOException {
21         Random random = new Random();
22         PrintWriter writter = new PrintWriter(out);
23         for (int i = 0;i < roles;i++) {
24             writter.println("ROL r"+i+ " 1 p0 0");
25         }
26
27         for (int i = 0;i < students;i++) {
28             writter.println("AGENT a" + i + " r" + random.nextInt(roles));
29         }
30
31         writter.close();
32     }
33 }
```

4.2.2. Fitness

La clase de fitness se construye a partir de un fichero. Esto quiere decir que dispone un HashMap que relaciona cada agente con su rol. Este hashmap es usado luego en la función para valorar

el balance de cada equipo. A este hashmap se le ha llamado *agentsToRole*.

La función de eval() se ha implementado de la siguiente manera:

```
public Double eval(Genotype<IntegerGene> gt) {
    double score = 0.0;
    int numPerTeam = getSize() / numTeam;

    IntegerChromosome g = gt.getChromosome().as(IntegerChromosome.class);
    Iterator it = g.iterator();

    //Obtener numero de equipos
    HashMap<Integer,List<Integer>> teams = new HashMap<>();
    int cnt = 0;
    while (it.hasNext()) {
        int team = ((IntegerGene)it.next()).intValue();

        List<Integer> list = teams.get(team);
        if (list == null) {
            list = new ArrayList<>();
            teams.put(team,list);
        }
        list.add(cnt);
        cnt++;
    }

    // Valorar el numero de equipos con su valor correcto
    score = 1- Math.abs(teams.size() - numTeam) / (double)(getSize() - numTeam);

    // Valorar numero de roles con su valor correcto
    int rolePerGroup = numPerTeam < numRoles ? numPerTeam : (int) Math.ceil(numPerTeam /
        numRoles);
    int studentsPerGrup = 0;
    for (int team : teams.keySet()) {
        List<Integer> list = teams.get(team);

        // Cuantos estudiantes hay en este equipo? Puntuar
        studentsPerGrup = list.size();
        score *= (1-Math.abs(studentsPerGrup - numPerTeam ) / (double)(getSize() -
            numPerTeam));

        HashSet<Integer> roles = new HashSet<Integer>();
        for (Integer value : list) {
            roles.add(agentsToRole.get(value));
        }

        // CUantos roles hay en este equipo? Puntuar
        score *= (1- Math.abs(roles.size()-rolePerGroup)/(double)(getSize() -
            rolePerGroup));
    }

    return score;
}
```

4.2.3. Genetic

La interfaz creará una instancia de esta clase y la configurará. Esta es la clase que lanza la ejecución del algoritmo genético y recoge el resultado. Se compone mayormente de una función run() que inicia la ejecución:

```

public String run()
{
    Fitness fitness = new Fitness(new Scanner(in));
    int agents = fitness.getAgents();
    int numTeams = agents/numMembersTeam;

    fitness.setNumTeams(numTeams);

    Engine<IntegerGene, Double> engine = Engine
            .builder(fitness::eval, IntegerChromosome.of(0,numTeams-1,agents))
```

```
12              .populationSize(population)
13              .maximizing()
14              .selector(selector)
15              .alterers(
16                      new Mutator<>(mutation),
17                      new UniformCrossover<>(0.5))
18              .build();
19
20      final Phenotype<IntegerGene,Double> result = engine.stream()
21              .limit(Limits.byFixedGeneration(generations))
22              .peek(statistics) // LiveStreamProxy
23              .collect(EvolutionResult.toBestPhenotype());
24
25      return "" + result;
26  }
```

4.2.4. LiveStream

Únicamente contiene un método: generationPassed(generation, fitness, result).

```
1  public interface LiveStream {
2    void generationPassed(long generation, double fitness, List<Integer> result);
3  }
```

Indica la generación actual, el mejor valor de fitness y el genoma del mejor individuo hasta el momento.

4.2.5. LiveStreamProxy

Esta clase recibe un objeto EvolutionResult al final de cada generación, su función es transformar dicho objeto a una llamada a la interfaz LiveStream.

CAPÍTULO 5
Evaluación del sistema

5.1 Prueba de validez

Ejecutando un caso de prueba con un tamaño de clase más normal (60 alumnos), la resolución es muy rápida (5 segundos):

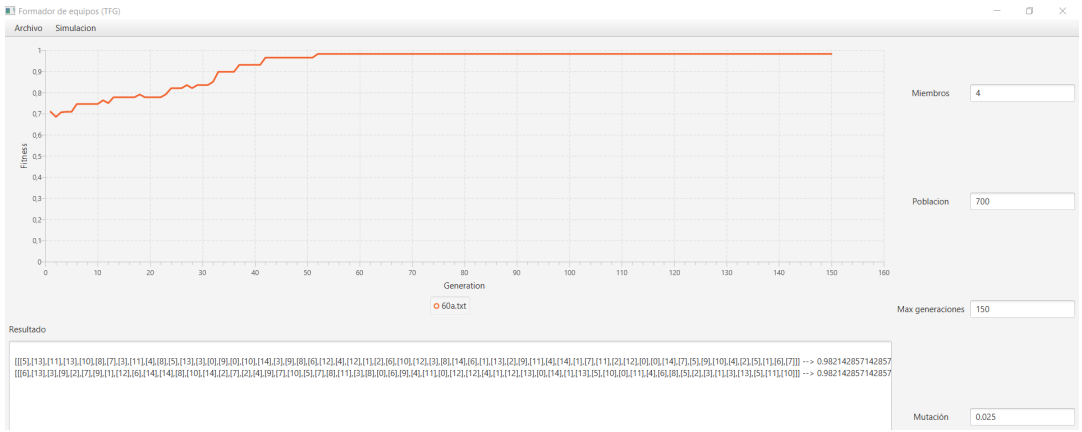

Figura 5.1: Resultado de ejecución en un caso de prueba normal

El mejor valor de fitness obtenido es de 0,98. Lo cual es una solución perfectamente válida para el problema en cuestión.

5.2 Prueba de estrés

Se ha generado un caso de una clase de 900 alumnos. Tal cantidad es totalmente impracticable para el algoritmo de fuerza bruta.

En la configuración del sistema se va a usar equipos de 4, población de 10.000 (10 veces el tamaño de la clase) y 400 generaciones. Este es el resultado de la ejecución

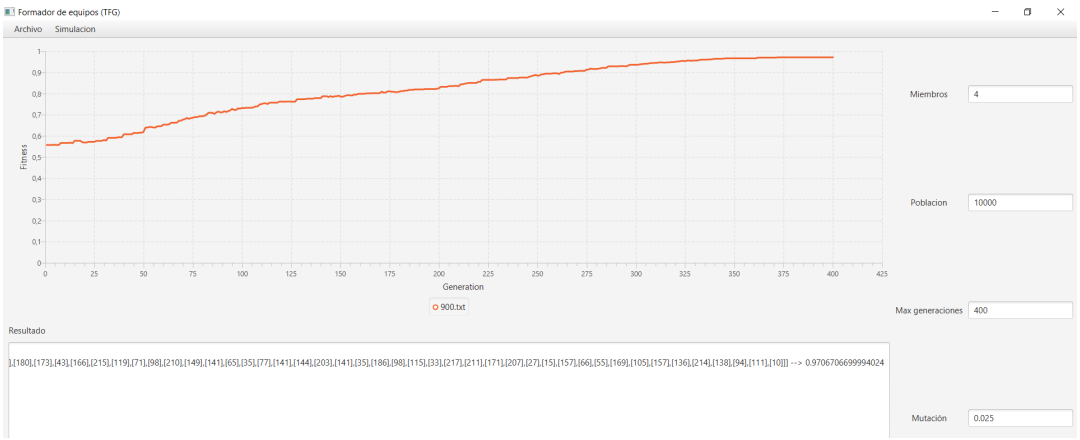

Figura 5.2: Ejecución en un caso de prueba grande

En aproximadamente 3 minutos, ha obtenido una solución con un valor de fitness de 0,97. Lo cual es una muy buena solución al problema.

5.3 Interfaz

La interfaz permite las siguientes operaciones:

- Iniciar la resolución de un caso de prueba con seleccionar el archivo desde el menú.
- Configurar los parámetros del algoritmo.
- Visualizar su evolución en tiempo real.
- Visualizar la evolución de varios casos de prueba de forma simultanea (gráficas). Permite así comparar distintas configuraciones frente al mismo problema.
- Relanzar casos de prueba.

La interfaz cumple pues, con todos los requisitos básicos que se requieren.

CAPÍTULO 6
Conclusiones

6.1 Valoraciones

En la realización de este proyecto, se han conseguido los objetivos propuestos. El principal objetivo era construir un sistema que resolviera el problema de crear equipos balanceados. Los tiempos y resultados obtenidos son bastantes buenos, lo cual satisface las necesidades del problema.

Se ha analizado las distintas posibilidades y configuraciones de algoritmos genético, se ha realizado experimentos y se ha elegido la mejor configuración posible en base a lo aprendido.

Se ha diseñado e implementado una pequeña aplicación gráfica para facilitar la entrada de información y la visualización de los resultados.

Como se ha podido comprobar, los resultados y tiempos obtenidos han superado claramente a las de un algoritmo clásico de fuerza bruta.

6.2 Relación con los estudios cursados

Existe una relación entre el trabajo y los estudios cursados:

- El uso de algoritmos genéticos se encuadra dentro de la asignatura de Técnicas, entornos y aplicaciones de inteligencia artificial (TIA).
- La arquitectura software entra dentro de la asignatura Ingeniería Software (I).
- El diseño y manejo de interfaces entra dentro de la asignatura Interfaces, persona y computador (IPC).
- La aplicación y pensamiento práctico de una solución a un problema propuesto forma parte de las competencias del grado. Así como el diseño de un proyecto

Bibliografía

[1] Algoritmo genético para la generación automática de equipos de trabajo. Pau Aguilar González *Proyecto final de Máster*, 2017.

[2] Competencias grado ingeniería informática. https://www.upv.es/titulaciones/GII/menu_1013005c.html

[3] Respuesta en stackoverflow sobre la configuración de un algoritmo genético:
https://stackoverflow.com/a/1075650/2489715
Graphics Noob. 2, Jul 2009

[4] Respuesta en stackoverflow sobre el tammaño de una población:
https://stackoverflow.com/a/7609715/2489715
NWS 30, Sep 2011

[5] Sobre la construcción de equipos balanceados http://innovationexcellence.com/blog/2010/11/12/building-balanced-teams/

[6] Uso de algoritmos genéticos para el diseño de vehículos https://ieeexplore.ieee.org/document/1703182/

[7] A team formation tool for educational environments
Val, E., Alberola, J.M Sanches-Anguix, V., Palomare, A., y Teruel, M. D. (2014).
Trends in Practical Applications of Heterogeneous Multi-Agent Systems. The PAAMS Collection. Advances in Intelligent Systems and COmputing.

[8] Optimal Population Size and the Genetic Algorithm
Stanley Gotshall, University of Portland
Bart Rylander, University of Portland
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.2431&rep=rep1&type=pdf

[9] GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with Matlab
Chapter 3
http://www.geatbx.com/docu/algindex-02.html

[10] Respuesta en stackoverflow sobre el roulettewheelselector y el rank selector:
https://stackoverflow.com/a/35014885/2489715

[11] Entrada en wikipedia sobre la selección stochastica.
https://en.wikipedia.org/wiki/Stochastic_universal_sampling

[12] Implementación de un algoritmo genético para la conformación de grupos mediante Análisis de redes sociales
Eduardo Grave
Leo Andrés Rubino
http://www.ridaa.unicen.edu.ar/xmlui/bitstream/handle/123456789/1473/Tesis%20Grave-Rubino.PDF?sequence=1&isAllowed=y

[13] Documentación del linecrossover en la librería jenetics.
Jenetic Library
http://jenetics.io/javadoc/org.jenetics/org/jenetics/LineCrossover.html

[14] Discursión sobre los indices de mutación en un algoritmo genético
https://www.researchgate.net/post/Why_is_the_mutation_rate_in_genetic_algorithms_very_small