

# Actividad 1

Pablo Hidalgo

## Contents

<b>1</b>	<b>Introducción</b>	<b>1</b>
<b>2</b>	<b>Descripción del problema del sudoku</b>	<b>1</b>
2.1	¿En qué consiste? . . . . .	2
2.2	Formalmente . . . . .	3
<b>3</b>	<b>Implementación</b>	<b>4</b>
3.1	Estructura de códigos en R . . . . .	4
3.2	Representación de cada individuo . . . . .	4
3.3	Función de adaptación ( <i>fitness</i> ) . . . . .	5
3.4	Inicialización de la población . . . . .	6
3.5	Selección de padres . . . . .	7
3.6	Cruce . . . . .	7
3.7	Mutación . . . . .	8
3.8	Selección de supervivientes . . . . .	8
3.9	El algoritmo . . . . .	8
<b>4</b>	<b>Evaluación experimental</b>	<b>9</b>
4.1	Metodología . . . . .	10
4.2	Aproximación 1 . . . . .	10
4.3	Aproximación 2 . . . . .	15
<b>5</b>	<b>Conclusiones</b>	<b>18</b>
<b>Bibliografía</b>		<b>19</b>

## 1 Introducción

Los algoritmos propios de la computación evolutiva pueden aplicarse en campos muy diversos. **En esta actividad se implementará un algoritmo evolutivo en R y se utilizará para la resolución del pasatiempo conocido como Sudoku.**

Comenzaremos describiendo en qué consiste el problema del sudoku y definiéndolo de una manera formal. Después, se comentará cómo se ha llevado a cabo la implementación en R para pasar a hacer una evaluación experimental que nos permita discutir su comportamiento.

## 2 Descripción del problema del sudoku

El juego del Sudoku comenzó a ser popular en Japón en 1986 y consiguió popularidad internacional en 2005 (Lynce and Ouaknine 2006). Se trata de un *pasatiempo* cuyas reglas son simples de entender pero su resolución puede ser todo un reto. Además, un sudoku bien construido tiene una única solución y puede ser resuelto mediante el razonamiento sin tener que recurrir a procedimientos de *ensayo y error*.

Los sudokus suelen tener asociado un *nivel de dificultad* que suele depender del número de casillas rellenas en el inicio. En (McGuire, Tugemann, and Civario 2014) se muestra que el número mínimo de casillas que deben de estar rellenas para que un sudoku tenga solución (*única*) es 17.

## 2.1 ¿En qué consiste?

El juego del Sudoku tradicional se representa por una matriz de dimensión  $9 \times 9$  en la que se disponen números enteros del 1 al 9. En un inicio solo algunas de las casillas están ocupadas con números siendo el objetivo del juego llenar las casillas vacías.

Los elementos principales del sudoku serán **las filas, las columnas y las subcuadrículas** como se muestra

COLUMNS									
	1	2	3	4	5	6	7	8	9
1									
2		1			2			3	
3									
R O W S	4				BLOCKS				
5		4			5			6	
6									
7									
8		7			8			9	
9									

a continuación.

Para que el **sudoku** se considere **resuelto**, se ha de verificar que:

1. cada **fila** contiene los números enteros del 1 al 9 apareciendo cada uno de ellos una y solo una vez,
2. cada **columna** contiene los números enteros del 1 al 9 apareciendo cada uno de ellos una y solo una vez,
3. cada **subcuadrícula** debe contener los números enteros del 1 al 9 apareciendo cada uno de ellos una y solo una vez.

Estas reglas de resolución explican porqué Sudoku significa “un solo número” en japonés (Lynce and Ouaknine 2006).

## 2.2 Formalmente

El problema puede ser expresado matemáticamente de múltiples formas. El desarrollo elegido tiene la motivación de que la aplicación del algoritmo genético sea más natural.

Una solución del sudoku se expresa como un vector  $\mathbf{s}$ , en términos de los algoritmos genéticos, un genotipo  $\mathbf{s} = (s_1, s_2, \dots, s_{81})$  donde  $s_i \in \{1, 2, \dots, 9\}$  para  $i = 1, \dots, 81$ . Para saber si el pasatiempo ha sido resuelto correctamente, es necesario definir una función de error que, en el caso de los algoritmo genéticos, se suele denominar función de adaptación o **fitness**. Para ello, comenzamos describiendo de manera formal qué elementos constituyen las **filas**, **columnas** y **subcuadrículas**, es decir, dada una posición del vector (genotipo) qué otras posiciones comparten la misma fila, columna o cuadrícula.

$$\begin{aligned} column(a) &= \{i \in \{1, \dots, 81\} \mid mod(i, 9) = mod(a, 9)\} \\ fila(a) &= \{i \in \{1, \dots, 81\} \mid \lfloor j/9 \rfloor = \lfloor a/9 \rfloor\} \end{aligned}$$

$$subcuadrícula(a) = \{i \in \{1, \dots, 81\} \mid (j-1) \cdot 3 < i \leq j \cdot 3 \cup mod(i, 3) + 1 = a\}$$

donde  $\lfloor a \rfloor$  se refiere a la parte entera de  $a$ .

Por otro lado, definimos las siguientes funciones que nos sirven para, dada una posición del genotipo, cuántos genes que comparten fila -columna o subcuadrícula- contienen el mismo número.

$$\begin{aligned} c(\mathbf{s}, i) &= \sum_{\{j \in column(i) \mid j \neq i\}} I(s_i = s_j) \\ f(\mathbf{s}, i) &= \sum_{\{j \in fila(i) \mid j \neq i\}} I(s_i = s_j) \\ s(\mathbf{s}, i) &= \sum_{\{j \in subcuadrícula(i) \mid j \neq i\}} I(s_i = s_j) \end{aligned}$$

siendo

$$I(s_i = s_j) = \begin{cases} 1, & s_i = s_j \\ 0, & s_i \neq s_j \end{cases}$$

El **sudoku** se considerará resuelto si y solo si  $\sum_i f(\mathbf{s}, i) = 0$ ,  $\sum_i c(\mathbf{s}, i) = 0$  y  $\sum_i s(\mathbf{s}, i) = 0$  a la vez y, ya que en  $f(\mathbf{s}, i), c(\mathbf{s}, i), s(\mathbf{s}, i) \geq 0$  se puede resumir como

$$fitness(\mathbf{s}) = \sum_{i=1}^{81} [f(\mathbf{s}, i) + c(\mathbf{s}, i) + s(\mathbf{s}, i)] = 0$$

Por tanto, la resolución del sudoku se puede expresar como el **problema de optimización**:

$$\arg \min_s fitness(s)$$

Pero el juego consiste en resolver el sudoku **sin poder modificar el valor de algunas casillas ya determinadas**. Sean  $s^*$  el valor de las casillas rellenas e  $i^*$  las posiciones que estas ocupan en el vector (*genotipo*), el problema de optimización debería incluir esta restricción:

$$\begin{aligned} \arg \min_s & \quad fitness(s) \\ s.a. & \quad s(i^*) = s^* \end{aligned}$$

### 3 Implementación

El problema de optimización descrito en la sección 2.2 se va a abordar mediante un **algoritmo genético**. Hay distintas estrategias de algoritmos genéticos que se podrían seguir para la implementación del problema. En este caso, la implementación se ha realizado conforme a las características que se detallan en las siguientes secciones.

La implementación computacional del algoritmo se ha desarrollado en **R**. R es un lenguaje de programación *Open Source* muy utilizado para computación de carácter estadístico. Tiene la ventaja de ser un lenguaje con una sintaxis sencilla que permite el cálculo vectorial. Existe una comunidad de usuarios muy activa.

#### 3.1 Estructura de códigos en R

Los *scripts* implementados se encuentran en la carpeta \R:

- `func_algoritmo_genetico.R` contiene la implementación de las distintas funciones necesarias para algoritmo genético.
- `algoritmo_genetico.R` contiene las funciones principales que lanzan el algoritmo genético.
- `fitness.R` contiene la función de fitness para el sudoku.

Para que los resultados de las siguientes secciones sea reproducibles, se fija la semilla para la generación de valores aleatorios

```
set.seed(123)
```

Todas las funciones se han implementado de tal forma que puedan utilizarse para aplicar un algoritmo genético en general y no se han particularizado para el problema del Sudoku **a excepción de la función de fitness**.

#### 3.2 Representación de cada individuo

Cada celda  $s_i$  del sudoku estará asociada a un gen representado por números enteros del 1 al 9, es decir,  $s_i \in \{1, 2, \dots, 9\}$ . En las ejecuciones el genotipo estará representado por un vector de enteros. Por ejemplo, en el caso de que el genotipo estuviese compuesto de 4 genes, una representación podría ser

```
c(1, 8, 6, 9)
```

```
## [1] 1 8 6 9
```

Como ya se ha dicho, el sudoku tendrá una serie de casillas rellenas al inicio y que no pueden ser cambiadas. Por lo tanto, la representación de los individuos que maneje el algoritmo, estará limitada a aquellas casillas sin un número asociado.

Un ejemplo de configuración inicial del sudoku se podría representar con el siguiente vector:

```
sudoku_facil <- c(NA, NA, NA, NA, 3, NA, NA, NA, 4,
                  NA, 9, NA, 4, NA, 6, NA, 7, NA,
                  NA, 5, NA, NA, NA, NA, 3, 8, NA,
                  NA, NA, NA, NA, 7, 8, NA, NA, 3,
                  3, NA, NA, NA, NA, NA, 6, 9, NA,
                  5, 4, NA, 6, NA, NA, NA, 2, NA,
                  7, NA, 5, NA, 2, 4, NA, NA, NA,
                  9, 8, 4, NA, 6, 5, 2, NA, NA,
                  NA, 2, 6, NA, 8, NA, NA, NA, 9)
```

El genotipo de los individuos tendrá una longitud igual al número de casillas vacías en la solución inicial, en este caso 48.

### 3.3 Función de adaptación (*fitness*)

La función de *fitness* que el algoritmo tratará de minimizar será la que aparece en la sección 2.2. Como se dijo anteriormente, un valor de  $fitness(s) = 0$  asegura que el juego ha sido resuelto. El valor máximo de esta función se alcanzará cuando **todas las casillas (genes) contengan el mismo valor** y tendrá un valor de  $fitness(s) = 972$ .

La función `fitness_sudoku()` calcula el valor de la función de *fitness* para un individuo dado cuyos argumentos son

- `x`: vector numérico que representa el genotipo del individuo en aquellas casillas vacías en la configuración inicial.
- `ind_cuadricula`: vector indicando qué posiciones equivalen a la representación de las subcuadrículas.
- `solucion_inicial`: vector representando la configuración inicial del sudoku que contendrá NA en aquellas posiciones que originalmente apareciesen vacías.

En nuestro caso, el `ind_cuadricula` se representa como

```
ind_cuadricula <- c(
  rep(rep(x = 1:3, each = 3), 3),
  rep(rep(x = 4:6, each = 3), 3),
  rep(rep(x = 7:9, each = 3), 3))
ind_cuadricula

## [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 4 4 4 4 5 5 5 6 6
## [36] 6 4 4 4 5 5 5 6 6 6 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9 7 7 7 8 8 8 9
## [71] 9 9 7 7 7 8 8 8 9 9 9
```

Viendo este vector como una matriz, se pone de relieve cómo están representadas las subcuadrículas

```
matrix(ind_cuadricula, nrow = 9, ncol = 9)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    1    1    4    4    4    7    7    7
## [2,]    1    1    1    4    4    4    7    7    7
## [3,]    1    1    1    4    4    4    7    7    7
## [4,]    2    2    2    5    5    5    8    8    8
## [5,]    2    2    2    5    5    5    8    8    8
## [6,]    2    2    2    5    5    5    8    8    8
## [7,]    3    3    3    6    6    6    9    9    9
## [8,]    3    3    3    6    6    6    9    9    9
## [9,]    3    3    3    6    6    6    9    9    9
```

Un ejemplo de solución inicial del sudoku se podría representar así:

```
sudoku_facil <- c(NA, NA, NA, NA, 3, NA, NA, NA, 4,
                    NA, 9, NA, 4, NA, 6, NA, 7, NA,
                    NA, 5, NA, NA, NA, NA, 3, 8, NA,
                    NA, NA, NA, NA, 7, 8, NA, NA, 3,
                    3, NA, NA, NA, NA, NA, 6, 9, NA,
                    5, 4, NA, 6, NA, NA, NA, 2, NA,
                    7, NA, 5, NA, 2, 4, NA, NA, NA,
                    9, 8, 4, NA, 6, 5, 2, NA, NA,
                    NA, 2, 6, NA, 8, NA, NA, NA, 9)
```

El genotipo de los individuos estará representado por el número de casillas vacías en la solución inicial, es decir,

```
tam_individuos <- sum(is.na(sudoku_facil))  
tam_individuos
```

```
## [1] 48
```

Simulamos un individuo con 48 genes (en la siguiente sección se describe la función `random_integer_representation()`, en este caso,

```
individuo <- random_integer_representation(valores = 1:9, tam = tam_individuos)
```

Ya podemos calcular el valor de la función de fitness para este individuo.

```
fitness_sudoku(x = individuo,  
                 ind_cuadricula = ind_cuadricula,  
                 solucion_inicial = sudoku_facil)
```

```
## [1] 87
```

### 3.4 Inicialización de la población

La población inicial del algoritmo se realizará de forma aleatoria, es decir, cada individuo estará representado por los genes que le corresponda y cada gen se inicializará como un número entero aleatorio comprendido entre 1 y 9.

La función `random_integer_representation()` es la que permite obtener este tipo de representación. Necesita los argumentos:

- `valores`: vector con los valores posibles que puede tomar cada gen.
- `tam`: número entero con el tamaño del genotipo (número de genes).

Como ejemplo:

```
random_integer_representation(valores = 1:9, tam = 9)
```

```
## [1] 3 8 1 4 8 2 6 2 2
```

La generación de la población se realiza mediante la función `generacion_poblacion()` cuyos argumentos son:

- `valores_posibles`: vector con los valores posibles que puede tomar cada gen.
- `num_genes`: número entero con el tamaño del genotipo (número de genes).
- `tam_poblacion`: entero con el número de individuos de los que estará formada la población.

El resultado es una `lista` compuesta por los distintos individuos. Por ejemplo,

```
poblacion <- generacion_poblacion(valores_posibles = 1:9,  
                                    num_genes = tam_individuos,  
                                    tam_poblacion = 3)
```

```
poblacion
```

```
## [[1]]  
## [1] 7 9 4 6 1 4 3 8 5 8 8 8 4 7 6 7 1 5 2 4 6 4 2 3 7 4 8 1 4 9 9 8 2 2 6  
## [36] 4 6 3 2 8 1 5 5 6 3 5 9 5  
##  
## [[2]]  
## [1] 9 9 6 4 2 9 3 1 9 7 2 5 9 6 4 6 3 3 2 4 9 2 1 2 7 6 9 7 7 5 6 8 8 9 4  
## [36] 3 4 1 2 8 3 3 1 3 7 8 5 4
```

```

## 
## [[3]]
## [1] 3 1 4 6 2 5 2 5 4 6 4 4 5 7 2 4 3 6 2 8 7 7 6 4 5 8 6 8 3 7 3 6 5 3 6
## [36] 9 9 3 3 9 6 9 5 4 6 2 6 3

```

### 3.5 Selección de padres

Para poder realizar el cruce entre individuos que se describe en la siguiente sección, es necesario elegir los padres, es decir, aquellos individuos entre los que realizar el cruce. En este caso se realizará mediante la **selección por torneo**.

La función `seleccion_padres()` tiene como argumentos:

- `num_padres`: número de padres.
- `poblacion`: lista con los individuos que componen la población.
- `fitness_poblacion`: vector con el valor de fitness para cada individuo de la población.
- `k`: entero con el número de competidores del torneo.

El resultado es una lista con aquellos padres elegidos en el torneo.

```

padres <- seleccion_padres(num_padres = 2,
                            poblacion = poblacion,
                            fitness_poblacion = sapply(poblacion,
                                                        fitness_sudoku,
                                                        ind_cuadricula = ind_cuadricula,
                                                        solucion_inicial = sudoku_facil),
                            k = 2)

padres

## [[1]]
## [1] 9 9 6 4 2 9 3 1 9 7 2 5 9 6 4 6 3 3 2 4 9 2 1 2 7 6 9 7 7 5 6 8 8 9 4
## [36] 3 4 1 2 8 3 3 1 3 7 8 5 4
##
## [[2]]
## [1] 7 9 4 6 1 4 3 8 5 8 8 8 4 7 6 7 1 5 2 4 6 4 2 3 7 4 8 1 4 9 9 8 2 2 6
## [36] 4 6 3 2 8 1 5 5 6 3 5 9 5

```

### 3.6 Cruce

El cruce entre los padres se realizará mediante **cruce por un punto**.

Función `one_point_crossover()` tiene como argumentos:

- `padres`: lista con los padres seleccionados para el cruce.
- `prob_cruce`: `numeric` en  $[0, 1]$  representando la probabilidad de que los padres se crucen y tengan descendencia o, por el contrario, se mantengan intactos.

```
resultado_cruce <- one_point_crossover(padres = padres, prob_cruce = 0.75)
```

```
resultado_cruce
```

```

## [[1]]
## [1] 9 9 6 4 2 9 3 1 9 7 2 5 9 6 4 6 3 3 2 4 9 2 1 2 7 6 9 7 7 5 6 8 8 9 4
## [36] 3 4 1 2 8 3 3 1 6 3 5 9 5
##
## [[2]]

```

```
## [1] 7 9 4 6 1 4 3 8 5 8 8 8 4 7 6 7 1 5 2 4 6 4 2 3 7 4 8 1 4 9 9 8 2 2 6
## [36] 4 6 3 2 8 1 5 5 3 7 8 5 4
```

### 3.7 Mutación

Una vez producido el cruce entre padres y obtenida su descendencia, se aplicará mutación de tipo *random resetting*, es decir, cada uno de los genes de cada individuo podrá sufrir una mutación con una probabilidad dada. En el caso de que se produzca una mutación en un gen, esta podrá tomar un valor entero aleatorio entre 1 y 9.

La función `random_resetting()` tiene parámetros:

- `x`: individuo sobre el que provocar mutaciones.
- `prob`: probabilidad de que se aplique mutación en cada gen.
- `valores_posibles`: vector con valores posibles para la mutación de cada gen.

```
random_resetting(individuo, prob = 0.1, valores_posibles = 1:9)
```

```
## [1] 3 8 4 8 9 1 5 8 5 5 9 5 7 6 1 9 3 2 3 9 9 7 6 9 6 7 4 6 3 2 9 9 7 8 1
## [36] 5 7 2 3 3 2 4 4 4 2 2 3 5
```

### 3.8 Selección de supervivientes

Se utilizará un modelo de tipo generacional, reemplazando la población actual enteramente por una nueva aplicando elitismo intercambiando el peor individuo de la siguiente generación por el mejor de la generación actual. No hay una función específica, sino que esta funcionalidad está directamente implementada en la función del algoritmo genético de la siguiente sección.

### 3.9 El algoritmo

Una ejecución del algoritmo se realizará gracias a la función `algoritmo_genetico()` con parámetros

- `poblacion_inicial`: lista con los individuos que compone la población inicial.
- `funcion_fitness`: función que calcula el fitness de cada individuo de la población.
- `genes_fijos`: vector con la configuración inicial de parámetros.
- `valores_mutacion`: vector con los valores posibles para la mutación.
- `num_padres`: entero con el número de padres.
- `prob_cruce`: probabilidad de que se produzca cruce entre padres.
- `tam_torneo`: tamaño del torneo en la selección de padres por torneo.
- `prob_mutacion`: probabilidad de que se produzca una mutación en cada gen del individuo.
- `tam_poblacion`: tamaño de la población.
- `max_iter`: iteraciones máximas que realizará el algoritmo
- `print_each`: imprimir por pantalla información del algoritmo cada `print_each` iteraciones.

El algoritmo se ejecutará hasta que el valor de fitness para algún individuo sea 0 o hasta que se llegue al número máximo de iteraciones especificado en el argumento `max_iter`.

El resultado será una lista con los siguientes elementos:

- `fitness`: lista donde habrá un elemento para cada iteración con los diferentes valores de fitness de cada individuo.
- `tiempo`: tiempo en segundos empleado en cada iteración.
- `poblacion_final`: lista con cada uno de los individuos de la población de la última iteración realizada.
- `parametros`: data.frame con los parámetros empleados en el algoritmo.

Para el estudio del comportamiento del algoritmo, se suelen realizar distintas ejecuciones para una misma combinación de parámetros. La función `pruebas_ga()` permite realizar diversas ejecuciones de una misma configuración de parámetros utilizando paralelización. Los argumentos de la función son los mismos que en la función `algoritmo_genetico()` añadiendo el argumento:

- `num_pruebas`: número de pruebas a realizar de una misma configuración de parámetros.

El resultado será una lista con las listas obtenidas en cada ejecución de `algoritmo_genetico()`.

Para ejecutar distintas combinaciones de parámetros lo podemos hacer definiendo un grid con las distintas configuraciones y con las siguientes líneas de código:

```
grid <- data.frame(tam_poblacion = 100,
                     prob_cruce = 0.9,
                     tam_torneo = 5,
                     prob_mutacion = c(0.01, 0.05, 0.1, 0.15, 0.25))

# Ejecución de pruebas ----

library(parallel)
set.seed(1234)
ini <- Sys.time()
resultados_dificil <- mcmapply(pruebas_ga,
                                 num_pruebas      = 30,
                                 num_padres       = 2,
                                 tam_poblacion    = grid$tam_poblacion,
                                 prob_cruce       = grid$prob_cruce,
                                 tam_torneo        = grid$tam_torneo,
                                 prob_mutacion     = grid$prob_mutacion,
                                 max_iter          = 5000,
                                 print_each        = 250,
                                 MoreArgs = list(generacion_poblacion_ini = generacion_poblacion,
                                                 genes_fijos           = sudoku_dificil,
                                                 funcion_fitness        = fitness_sudoku,
                                                 valores_posibles       = 1:9,
                                                 valores_mutacion        = 1:9
                                                 )
                                 ,
                                 SIMPLIFY = FALSE
                                )
fin <- Sys.time()
fin-ini
```

## 4 Evaluación experimental

Los algoritmos genéticos llevan asociados una serie de parámetros que rigen su comportamiento y pueden impactar en los resultados obtenidos. Por lo tanto, es fundamental conocer cómo afectan los distintos parámetros al algoritmo para poder tener un conocimiento sobre los mismos que nos permita ser conscientes

de los cambios. Por ejemplo, en función del problema, nos puede interesar favorecer una combinación de parámetros que necesite un menor tiempo computacional (un menor número de iteraciones) para obtener una solución cuya función de fitness sea ligeramente peor (en media) que la de otra combinación de parámetros pero que necesite un alto número de iteraciones.

## 4.1 Metodología

Para contrastar el comportamiento del algoritmo genético aplicado al problema de resolver un Sudoku, se han elegido dos configuraciones iniciales, una fácil con 33 casillas iniciales rellenas y otra difícil, con 30 casillas rellenas. Como se dijo en la sección 2, se considera que la dificultad del problema del Sudoku es inversamente proporcional al número de celdas rellenas al principio.

En la implementación descrita del algoritmo genético, los parámetros del algoritmo serían:

- **Tamaño de la población.**
- En la selección de padres por torneo, el **tamaño del torneo**.
- **Probabilidad de cruce**
- **Probabilidad de mutación.**

En nuestro caso, una de las preguntas fundamentales que queremos responder es **cómo impactan los diferentes valores de la probabilidad de mutación en el algoritmo genético**.

A continuación se describen las dos metodologías seguidas.

### 4.1.1 Aproximación 1

Se han fijado todos los parámetros del algoritmo a excepción de la probabilidad de mutación. Se ha ejecutado cada combinación de parámetros un total de 50 veces.

### 4.1.2 Aproximación 2

Para tratar de obtener un resultado del impacto global de los parámetros, se han generado 50 configuraciones aleatorias de los parámetros con la siguiente lógica:

- El tamaño de la población se fija en 10 para la instancia sencilla y 100 para la instancia compleja.
- Probabilidad de cruce se supone una variable aleatoria uniforme  $U(0.5, 1)$ .
- **Tamaño del torneo:** valor entero aleatorio entre 2 y 10, ambos incluidos.
- **Probabilidad de cruce:** valor con distribución uniforme  $U(0.5, 1)$ .
- **Probabilidad de mutación:** permutación aleatoria de 50 números igualmente espaciados entre 0 y 0.25.
- 10 ejecuciones para cada combinación de parámetros

## 4.2 Aproximación 1

Tanto la instancia compleja como la instancia sencilla se recogen el script `\sudoku_prueba.R` en los objetos `sudoku_facil` y `sudoku_dificil` respectivamente, tal y como se muestra a continuación.

```
sudoku_facil <- c(NA, NA, NA, NA, 3, NA, NA, NA, 4,
                  NA, 9, NA, 4, NA, 6, NA, 7, NA,
                  NA, 5, NA, NA, NA, NA, 3, 8, NA,
                  NA, NA, NA, NA, 7, 8, NA, NA, 3,
                  3, NA, NA, NA, NA, NA, 6, 9, NA,
                  5, 4, NA, 6, NA, NA, NA, 2, NA,
                  7, NA, 5, NA, 2, 4, NA, NA, NA,
```

```

9, 8, 4, NA, 6, 5, 2, NA, NA,
NA, 2, 6, NA, 8, NA, NA, NA, 9)

sudoku_dificil <- c(NA, 3, 8, NA, NA, NA, 6, NA,
5, 4, NA, NA, NA, NA, 2, NA, NA,
NA, NA, 1, NA, 9, NA, 3, NA, NA,
8, 6, NA, NA, 4, NA, NA, 7, NA,
4, NA, NA, NA, 8, 3, 5, NA, NA,
NA, NA, 7, NA, NA, NA, NA, 4, NA,
3, 8, 6, NA, NA, 9, NA, NA, 4,
NA, 2, NA, 4, 6, NA, NA, 3, NA,
NA, NA, NA, NA, NA, 7, NA, 8, NA)

```

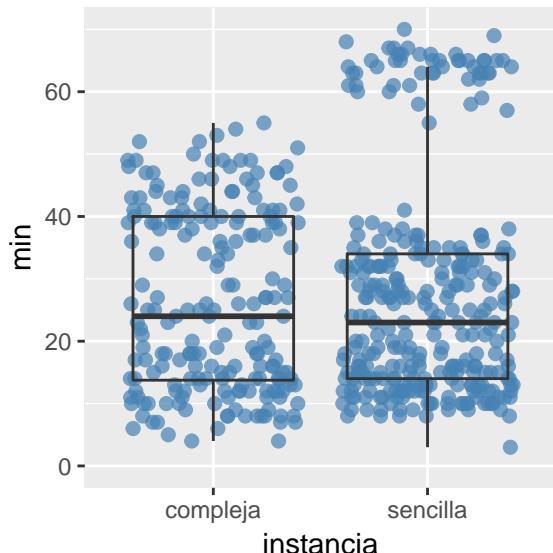
Para esta primera aproximación, se han fijado todos los parámetros del algoritmo a excepción de la probabilidad de mutación con los valores:

- **Tamaño de la población** 10 individuos para la instancia sencilla y 100 individuos para la compleja.
- **Tamaño del torneo:** 2 individuos para la instancia sencilla y 5 para la compleja.
- **Probabilidad de cruce:** 0.9 para ambas instancias.

La **probabilidad de mutación** se ha variado con los valores: 0, 0.01, 0.05, 0.1, 0.15 y 0.25.

Cada combinación de parámetros se ha lanzado un total de 50 veces con un máximo de 5000 iteraciones.

En el siguiente gráfico se representa la distribución de las ejecuciones para las dos instancias. Se ha utilizado un diagrama de cajas.



Como puede verse, **ninguna de las ejecuciones ha conseguido obtener un valor de la función fitness de 0**, es decir, no se ha conseguido resolver el sudoku. Ambas distribuciones son muy similares.

Para la **instancia sencilla**, obtenemos los siguientes estadísticos:

```

##      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
##      3.00 14.00 23.00 28.06 34.00 70.00

```

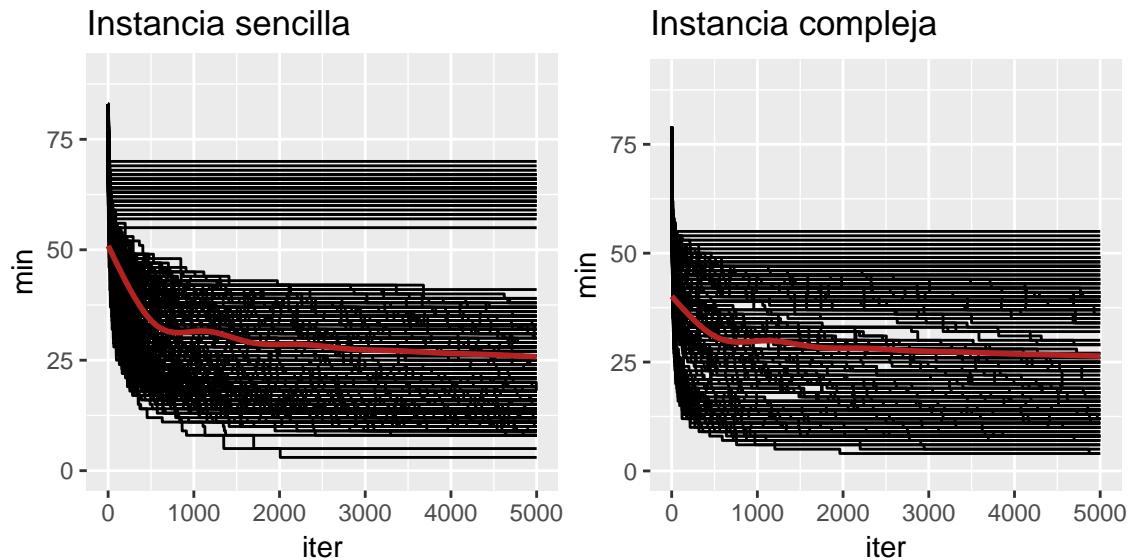
Análogamente, calculamos los mismos valores para la **instancia compleja**:

```

##      Min. 1st Qu. Median   Mean 3rd Qu.   Max.
##      4.00 13.75 24.00 26.42 40.00 55.00

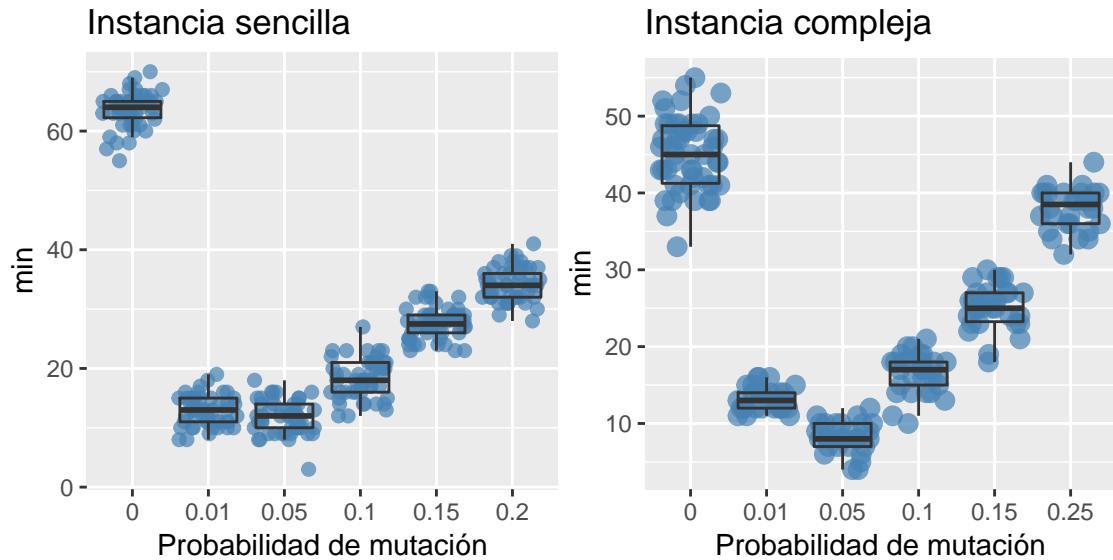
```

A continuación se representa la evolución seguida por cada iteración del algoritmo genético.



La curva roja representa un ajuste global mediante *GAM* que permite ver la tendencia seguida por todas las ejecuciones en media. Como suele ser habitual en los algoritmos genéticos, en las primeras iteraciones el algoritmo presenta una fuerte disminución de la función de fitness para después empezar a estabilizarse. Al haberse aplicado elitismo en el algoritmo, se aprecian escalones en las líneas. Cabe destacar las líneas superiores que se mantienen casi constantes en la instancia sencilla; se corresponden con las ejecuciones realizadas con el valor de **probabilidad de mutación igual a 0** con unos valores de fitness elevados y constantes que pone de manifiesto la incapacidad del algoritmo de mejorar sin poder usar mutaciones en los genes.

Uno de los objetivos de la discusión es el de conocer el comportamiento del algoritmo frente a diversos valores de la probabilidad de mutación. En los siguientes gráficos se muestra la distribución de las ejecuciones hechas variando esta probabilidad



En ambas instancias se aprecia una relación -no lineal- entre la probabilidad de mutación y el valor mínimo de fitness, alcanzándose el mínimo valor en una probabilidad de 0.05. Hay que destacar que en el caso de la instancia sencilla, los resultados obtenidos para las probabilidades de 0.01 y 0.05 son indistinguibles, pero que el valor de fitness para la probabilidad de 0.01 es ligeramente peor que para 0.05 en la instancia compleja.

Las correlaciones lineales de pearson para la relación entre la probabilidad de mutación y el valor de fitness

son -0.0991411 y 0.0781608 para las instancias sencilla y compleja respectivamente siendo las correlaciones bastante altas.

Si aproximamos la tendencia de los puntos con una **regresión lineal** obtenemos para la **instancia sencilla**

```
##
## Call:
## lm(formula = min ~ I(prob_mutacion * 100), data = min_iter_facil2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -11.5379  -2.5379   0.2947   2.2947   9.3960
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)             8.37047   0.41271  20.28 <2e-16 ***
## I(prob_mutacion * 100)  1.23348   0.03368  36.63 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.618 on 248 degrees of freedom
## Multiple R-squared:  0.844, Adjusted R-squared:  0.8434
## F-statistic: 1342 on 1 and 248 DF, p-value: < 2.2e-16
```

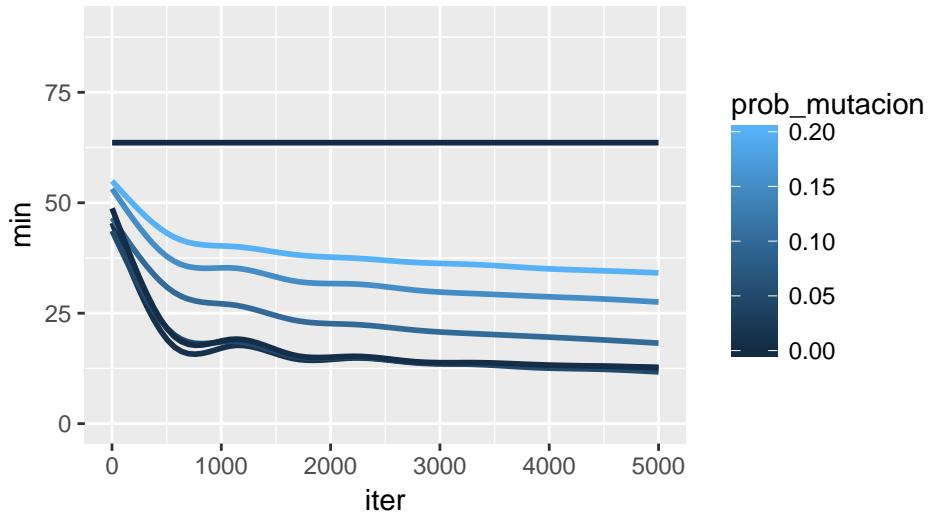
Y para la **instancia compleja**

```
##
## Call:
## lm(formula = min ~ I(prob_mutacion * 100), data = min_iter_dificil2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -8.8470 -2.8470  0.2232  3.4336  7.8969
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)             6.91713   0.55685 12.42 <2e-16 ***
## I(prob_mutacion * 100)  1.18597   0.03986 29.76 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.077 on 148 degrees of freedom
## Multiple R-squared:  0.8568, Adjusted R-squared:  0.8558
## F-statistic: 885.4 on 1 and 148 DF, p-value: < 2.2e-16
```

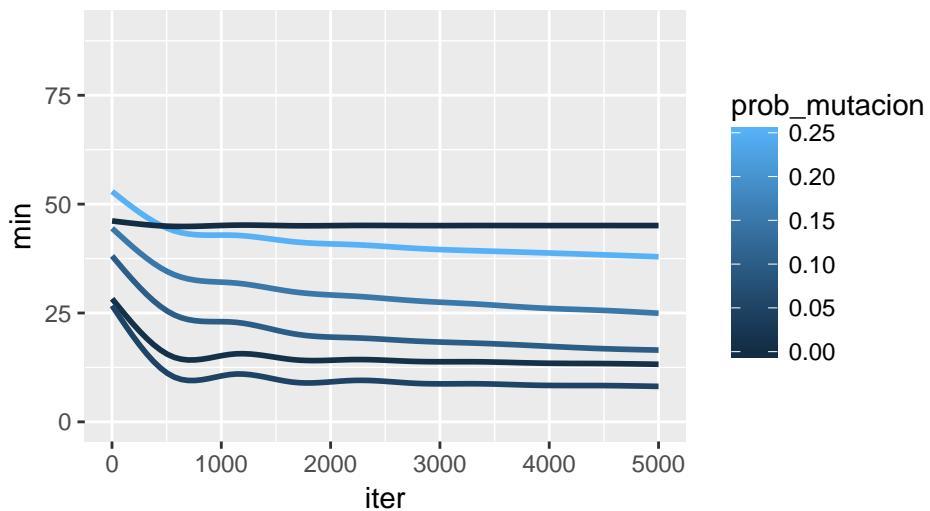
Somos conscientes de que, a la vista del gráfico anterior, existe una cierta tendencia no lineal y, por lo tanto, una regresión lineal no es el modelo más adecuado. No obstante, sí que nos sirve para representar la tendencia global y descartaremos el caso de que la probabilidad de mutación es 0 ya que parece que no se comporta de forma coherente con el resto de valores. Los parámetros de la probabilidad de mutación en ambas regresiones lineales se muestran con p-valores <2e-16, indicando que no hay evidencia estadística significativa para suponer que no hay una influencia de la probabilidad de mutación en el valor de la función de fitness.

El comportamiento también se puede intuir si representamos el progreso del algoritmo aproximando cada una de las ejecuciones mediante un ajuste por GAM's para cada valor de la probabilidad de mutación.

### Instancia sencilla

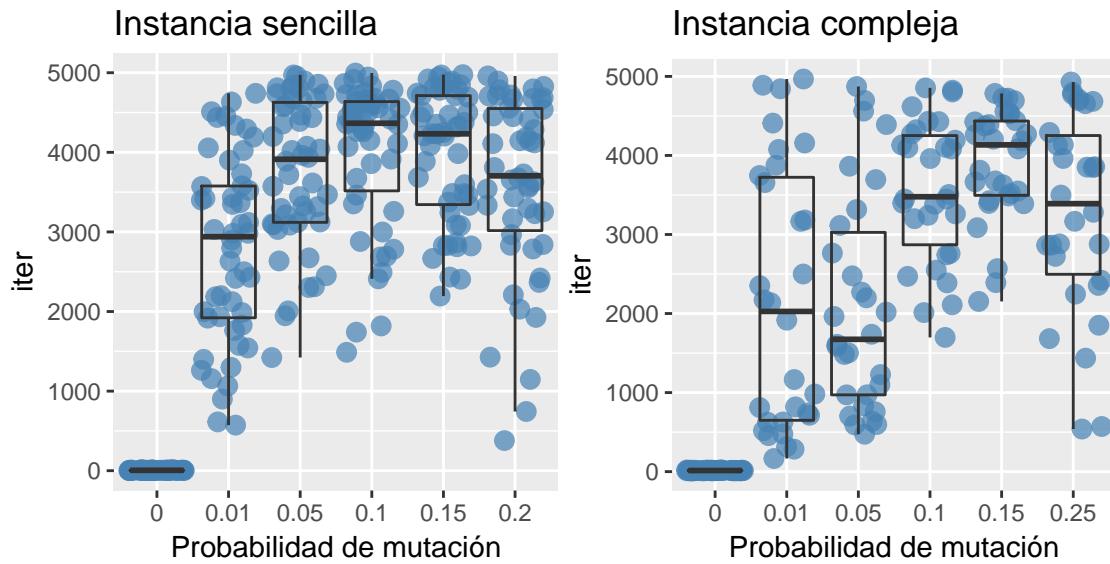


### Instancia compleja



La tendencia en ambas instancias es similar: cuanto menor es la probabilidad de mutación, llega a valores de fitness mejores en menos iteraciones. El límite se ha impuesto en 5000 iteraciones, pero si se hubiese dejado un mayor recorrido, puede que alguna de las configuraciones con una mayor probabilidad de mutación pudiese llegar a ser competitiva. Sin embargo, el tiempo computacional requerido hubiese sido también muy elevado.

Para conocer también cómo afecta la probabilidad de mutación a la eficiencia computacional del algoritmo, se ha obtenido para cada una de las ejecuciones la primera iteración en la que se alcanza el mínimo valor de fitness, obteniendo



En este caso, la relación no es tan clara. En la línea del gráfico anterior, cuanto mayor es la probabilidad de mutación se necesitan un mayor número de iteraciones. En este caso, la probabilidad de mutación del orden de 0.01, favorece el menor número de iteraciones en la instancia sencilla mientras que en la compleja aunque los resultados en media, son parecidos para 0.01 y 0.05, la probabilidad de 0.05 presenta una menor variabilidad.

### 4.3 Aproximación 2

A continuación se muestran las configuraciones de parámetros simuladas.

Para la **instancia sencilla**:

```
min_iter_facil_aprox2 %>%
  ungroup() %>%
  select(tam_poblacion,
         prob_mutacion,
         prob_cruce,
         tam_torneo) %>%
  distinct()

## # A tibble: 50 x 4
##   tam_poblacion prob_mutacion prob_cruce tam_torneo
##       <dbl>          <dbl>        <dbl>      <int>
## 1           10  0.005102041  0.5568517      2
## 2           10  0.137755102  0.8111497      4
## 3           10  0.066326531  0.8046374      8
## 4           10  0.045918367  0.8116897      6
## 5           10  0.030612245  0.9304577      3
## 6           10  0.071428571  0.8201553      6
## 7           10  0.229591837  0.5047479      6
## 8           10  0.025510204  0.6162753      8
## 9           10  0.091836735  0.8330419      3
## 10          10  0.250000000  0.7571256      9
## # ... with 40 more rows
```

Para la **instancia compleja**:

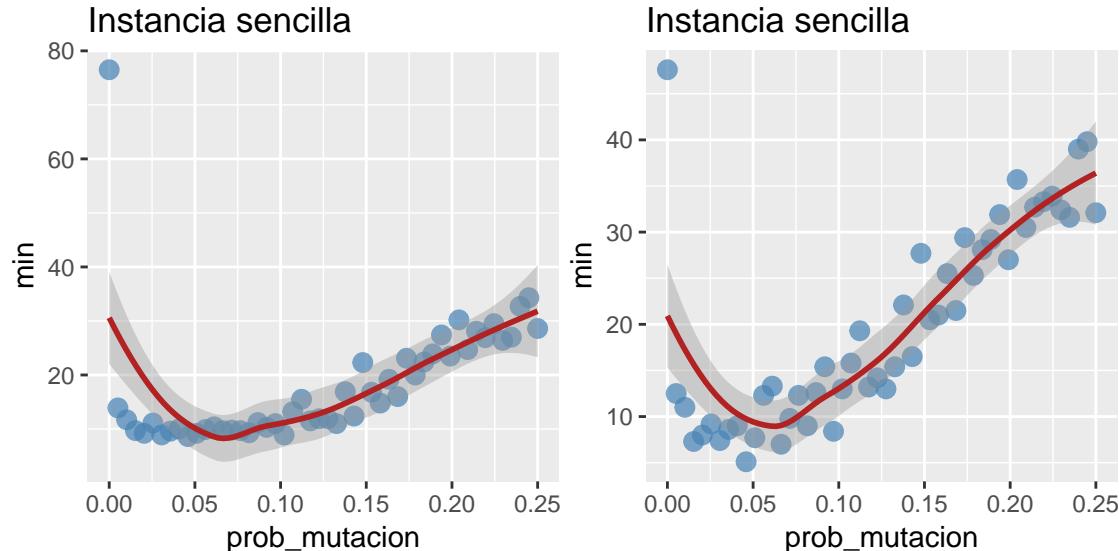
```

min_iter_dificil_aprox2 %>%
  ungroup() %>%
  select(tam_poblacion,
         prob_mutacion,
         prob_cruce,
         tam_torneo) %>%
  distinct()

## # A tibble: 50 x 4
##   tam_poblacion prob_mutacion prob_cruce tam_torneo
##       <dbl>          <dbl>        <dbl>      <int>
## 1 100 0.005102041 0.5568517 2
## 2 100 0.137755102 0.8111497 4
## 3 100 0.066326531 0.8046374 8
## 4 100 0.045918367 0.8116897 6
## 5 100 0.030612245 0.9304577 3
## 6 100 0.071428571 0.8201553 6
## 7 100 0.229591837 0.5047479 6
## 8 100 0.025510204 0.6162753 8
## 9 100 0.091836735 0.8330419 3
## 10 100 0.250000000 0.7571256 9
## # ... with 40 more rows

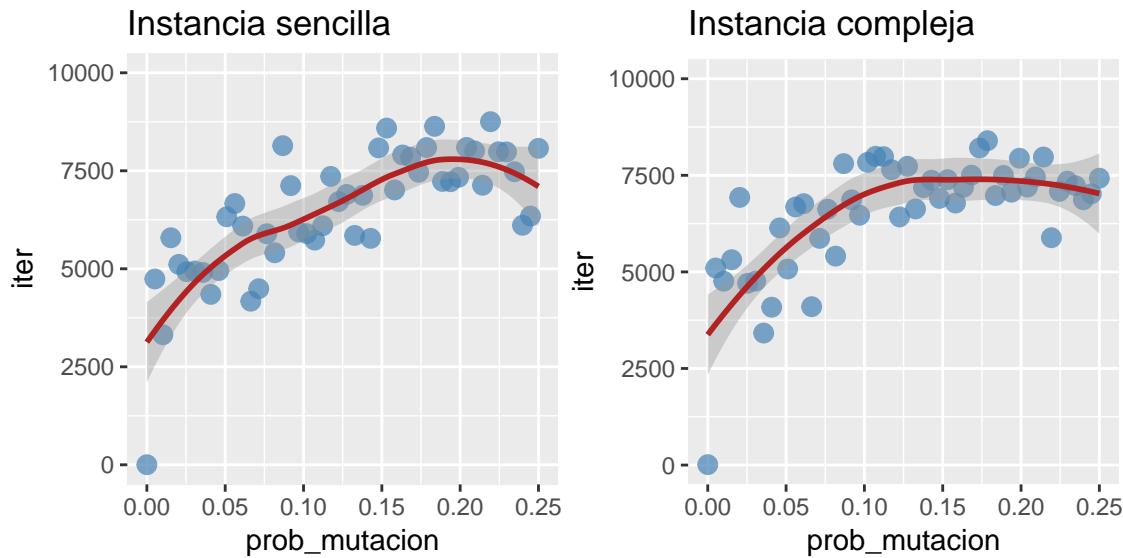
```

Vamos a repetir el análisis anterior pero esta vez sobre las ejecuciones realizadas con el espacio de parámetros definido en la metodología.

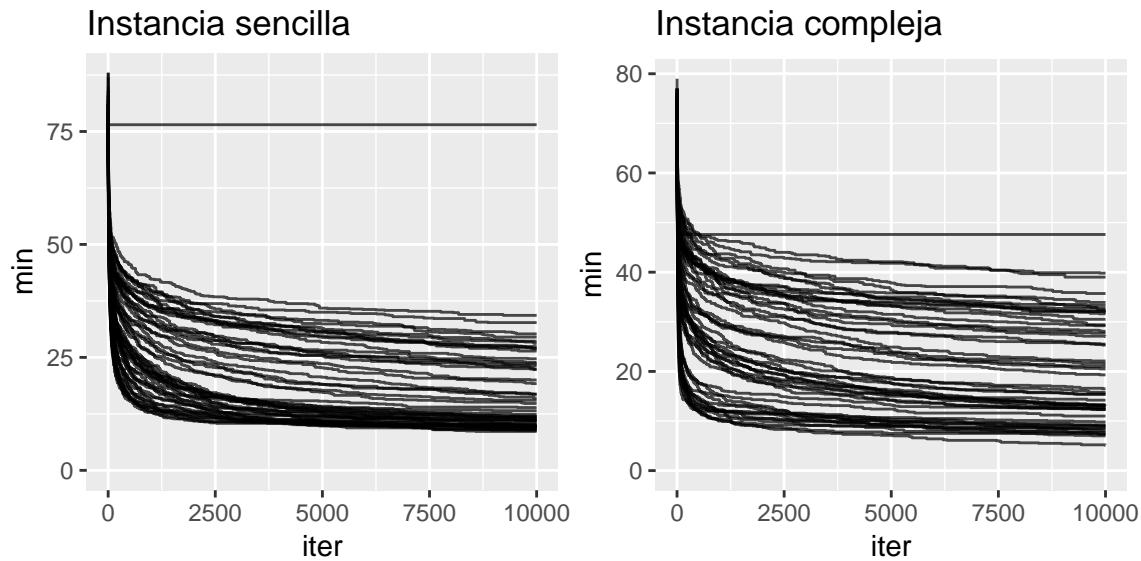


La idea expuesta en la sección anterior se ve reforzada en este caso en el que se han simulado diversas configuraciones de parámetros. Se sigue observando una tendencia no lineal, favoreciendo valores menores de fitness para probabilidades de mutación bajas. De nuevo, aunque en este caso se ha aumentado el número límite de iteraciones a 10000, tampoco se ha llegado a fitness de 0.

En los siguientes gráficos, en los que enfrentamos la probabilidad de mutación contra el número de iteraciones (en media) para alcanzar el mínimo valor de fitness en cada iteración, se obtiene una tendencia no lineal. Además, este número de iteraciones mínimas no se alcanza en el límite de 10.000 y una menor probabilidad de mutación favorece un menor número de iteraciones. Como hemos visto en el resto del análisis, las ejecuciones con probabilidad de mutación 0 conllevan un comportamiento anómalo y el algoritmo no consigue mejorar el menor fitness conseguido en la población inicial.



En el caso de la tendencia que sigue las iteraciones, se ha resumido la información trazando una línea para cada combinación de parámetros que se corresponde con los valores medios de la función de fitness en cada iteración. De nuevo, las conclusiones son las mismas que se obtuvieron anteriormente.



Por último, los parámetros de las regresiones lineales asociadas, vuelven a ser significativamente distintos de 0 indicando una relación entre la probabilidad de mutación y los valores de fitness obtenidos.

#### Instancia sencilla:

```
##
## Call:
## lm(formula = min ~ I(prob_mutacion * 100), data = min_iter_facil_aprox2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -13.132  -4.948  -1.804   2.218  71.814 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept)  10.0000    0.0000  10.000 0.0000000 ***
## I(prob_mutacion * 100) 0.0000    0.0000  0.000  0.9999999
```

```

## (Intercept) 10.18588 0.94134 10.821 <2e-16 ***
## I(prob_mutacion * 100) 0.62625 0.06489 9.651 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.68 on 498 degrees of freedom
## Multiple R-squared: 0.1576, Adjusted R-squared: 0.1559
## F-statistic: 93.15 on 1 and 498 DF, p-value: < 2.2e-16

Instancia compleja:

## 
## Call:
## lm(formula = min ~ I(prob_mutacion * 100), data = min_iter_dificil_aprox2)
## 
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -13.007  -4.124  -0.857   2.701  52.615 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 6.38471  0.66083  9.662 <2e-16 ***
## I(prob_mutacion * 100) 1.09578  0.04555 24.056 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 7.499 on 498 degrees of freedom
## Multiple R-squared: 0.5375, Adjusted R-squared: 0.5365
## F-statistic: 578.7 on 1 and 498 DF, p-value: < 2.2e-16

```

## 5 Conclusiones

El objetivo marcado era el de implementar un algoritmo capaz de resolver el problema del sudoku. En las pruebas realizadas, siguiendo dos instancias del problema de diferente dificultad, el algoritmo no ha sido capaz de resolverlo en ningn caso.

Una parte que se podría tratar de estudiar es la de si la representación de los individuos mediante un gen en cada celda con un nmero comprendido entre 1 y 9 es la mss adecuada. Se propone como trabajo futuro el estudio de otras respresentaciones, como la representación con permutaciones. Es decir, cada individuo estaría representado por una permutación del siguiente genotipo

$$s = (1, 2, \dots, 9, \\ 1, 2, \dots, 9, \\ 1, 2, \dots, 9, \\ \dots, \\ 1, 2, \dots, 9)$$

Esto supondría elegir un tipo de cruce adecuado a este tipo de representaciones.

Además, podría ser también tenido en cuenta para un estudio posterior otra forma de mutación dirigida, por ejemplo, no dejando que cada gen mute aleatoriamente hacia cualquier entero, sino que solo se permitan mutaciones a otros nmeros que no aparezcan ya ni en su misma fila, ni columna, ni subcuadrícula.

En cuanto a la probabilidad de mutación, se ha observado una influencia clara en el algoritmo. Por un lado, los mejores resultados se han obtenido con valores entorno a 0.05 tanto para los valores de la función de fitness

como para el número de iteraciones necesarias. A la vista de estos resultados, la recomendación sería lanzar las ejecuciones con probabilidades de mutación bajas, sin ser 0, que favorecerían la eficiencia computacional y los valores de fitness.

El sudoku ha sido probado con dos instancias que, a priori, se consideraban de diversa dificultad. En las pruebas los resultados obtenidos para ambas instancias han sido similares.

## Bibliografía

- Lynce, I, and J Ouaknine. 2006. “Sudoku as a SAT Problem.” *Symposium A Quarterly Journal in Modern Foreign Literatures*, 1–9. <http://citeseervx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.6274&rep=rep1&type=pdf>.
- McGuire, Gary, Bastian Tugemann, and Gilles Civario. 2014. “There Is No 16-Clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration.” *Experimental Mathematics* 23 (2):190–217. <https://doi.org/10.1080/10586458.2013.870056>.