

# Computación evolutiva: primera práctica

Andrés Mañas Mañas

December 11, 2016

## **Abstract**

Implementación desde cero un algoritmo genético para la resolución del Problema de la Mochila Binario y evaluación de los resultados con herramientas gráficas de monitoreo.

La implementación cumple con estas características: representación binaria de los individuos, función fitness basada en ordenamiento por ratio de los objetos y de forma que trate de forma adecuada aquellos individuos que representan soluciones no factibles al problema de la mochila binario, población inicial aleatoria, selección de padres por torneo, cruce por punto con probabilidad dada, mutación de todos los genes con una probabilidad dada, selección de supervivientes atendiendo a un modelo generacional con elitismo.

En la implementación, todos los metaparámetros del algoritmo son configurables al lanzarse.

# 1 Contenidos

1. Contenidos
2. Introduccion
  1. MUY IMPORTANTE !!!
3. Descripción del problema y requisitos
4. Implementación
  1. Representación del conocimiento
  2. Funciones más relevantes
5. Formas de ejecutar el algoritmo
  1. Ejecución desde consola
  2. Ejecución programática del algoritmo
  3. Ejecución desde fichero del algoritmo
6. Evaluación Experimental
  1. Análisis de resultados
  2. Experimento simple
    1. Curvas de evolución
    2. Conclusiones del experimento simple
  3. Experimento complejo
    1. Curvas de evolución
    2. Conclusiones del experimento complejo

# 2 Introduccion

El presente documento contiene el informe con el trabajo realizado para la primera práctica de Computación Evolutiva por el alumno Andrés Mañas Mañas.

El lenguaje de programación elegido para la realización de la práctica es **Clojure** (un dialecto de Lisp). He elegido este lenguaje porque hace muy sencilla la representación del conocimiento como estructuras de datos muy fácilmente manipulables. Es además mi lenguaje favorito.

En el material entregado para esta práctica se incluye el archivo **README.pdf**. Es del todo recomendable leer tal archivo en primer lugar para:

- documentarse sobre cómo disponer de un entorno en el que ejecutar el código fuente de la práctica
- visualizar rápidamente la estructura de árbol con todos los documentos que se incluyen en la práctica

Parte de los entregables de la práctica consisten en un sistema que permita ver la evolución del algoritmo y los resultados alcanzados. No son tantas las librerías que permiten visualizar gráficos en Clojure. Si acaso la más destacada sea <http://incanter.org/>. Sin embargo es una librería pesadísima, y he preferido no agregarla como dependencia del proyecto. Además de lo desagradable que es que al ejecutar los algoritmos se vayan abriendo popups en background.

**Por eso me he tomado la libertad de desplegar en un pequeño servidor que tengo en**

**AWS un web service en php y un panel html que permite ver en tiempo real la evolución de una ejecución del algoritmo de aprendizaje.**

Cuando se lanza una ejecución de un experimento, de forma transparente se envían periódicamente estadísticas del avance del algoritmo a <http://amanas.ml/ce/service.php>

Por otro lado, si simultáneamente tenemos abierta en un navegador la página <http://amanas.ml/ce/status.html>, podremos ir viendo en tiempo real y de forma gráfica la evolución del aprendizaje del experimento que estemos ejecutando en cada momento.

## 2.1 MUY IMPORTANTE !!!

Hay que disponer de conexión a internet y abrir <http://amanas.ml/ce/status.html> cuando ustedes vayan a probar los algoritmos que les entrego con estos materiales.

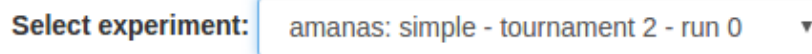


Figura 1. Selector de experimentos

Cada vez que se inicia un experimento, uno de los parámetros que se indican en la configuración del mismo es el nombre del experimento. Después, en la página <http://amanas.ml/ce/status.html> basta seleccionar del combo box el nombre que se ha indicado en la configuración y así se puede ver el resultado de la ejecución del algoritmo.

Estos informes gráficos tienen configurado un tiempo de vida en el servidor que debiera ser suficiente para que siguieran estando disponibles cuando ustedes decidan visualizarlos. En el caso de todos los realizados por mí, el nombre siempre lo empiezo por “amanas: ...”. De este modo, si ustedes realizan experimentos de los mismo ejemplos que yo he realizado, pueden comparar las distintas realizaciones.

## 3 Descripción del problema y requisitos

En la primera práctica se nos pide que implementemos desde cero un algoritmo genético para la resolución del **Problema de la Mochila Binario**.

El **Problema de la Mochila Binario** se describe así:

*Dada una mochila con cierta “capacidad” y varios objetos con cierto “volumen” y “valor”, se trata de determinar qué objetos hay que introducir en la mochila para maximizar el valor total de los objetos contenidos en la misma. Obviamente, se debe cumplir que la suma de los volúmenes de los objetos introducidos en la mochila no exceda la capacidad de ésta.*

Los requisitos pedidos son:

- representación binaria de los individuos

- función fitness basada en ordenamiento por ratio de los objetos y de forma que trate de forma adecuada aquellos individuos que representan soluciones no factibles al problema de la mochila binario
- población inicial aleatoria
- selección de padres por torneo
- cruce por punto con probabilidad dada
- mutación de todos los genes con una probabilidad dada
- selección de supervivientes atendiendo a un modelo generacional con elitismo

## 4 Implementación

Como dije más arriba, el lenguaje que he seleccionado es **Clojure**, un dialecto de Lisp que se engloba por lo tanto en el paradigma de programación funcional.

A mi juicio, las ventajas de utilizar un lenguaje como Clojure son:

- facilidad en la representación del conocimiento, parte clave del problema
- sencillez en la programación de las distintas funciones clave en el algoritmo de aprendizaje. Es un lenguaje de alto nivel que permite realizar operaciones de mapeo o filtrado con una simple función, por lo que se ahorra mucho código y por lo tanto se evita complejidad en la solución desde la base
- evitar los problemas relacionados con la concurrencia y la mutación del estado de los objetos en el paradigma de la orientación a objetos
- y por qué no, diversión garantizada pues Clojure es el lenguaje más divertido que conozco, y puestos a trabajar, mejor hacerlo con alegría

La única desventaja que le veo a la utilización de clojure es que no es el lenguaje que mejor rendimiento ofrece. Corre sobre la máquina virtual java (aunque también hay intérpretes para .Net o javascript). Y, aunque dependiendo del caso puede ofrecer rendimientos muy superiores a, por ejemplo, java, cuando la carga de computación es elevada suele flaquear un poco. En este sentido, lenguajes como C seguro que brindarían rendimientos mejores.

La implementación del algoritmo que satisface las especificaciones indicadas anteriormente pueden ustedes encontrarlas en el archivo **src/ce/p1.clj**

Vamos a comentar un poco las partes claves del código que entrego.

### 4.1 Representación del conocimiento

Tenemos que representar como mínimo dos conceptos clave: los objetos (genes) y los individuos.

**Cada objeto lo represento como un mapa** (estructura clave-valor) con tres claves: nombre del objeto, valor del objeto y volumen del objeto.

```
{:nam "Objeto 0" :val 1 :vol 3}
```

Como parte del preprocesado inicial de los objetos de entrada del problema, agrego automáticamente una nueva propiedad a cada objeto que será el **ratio (valor/volumen)**, de modo

que en los distintos accesos que haga al objeto no tenga que recalcular este valor y así ahorro en computación. Es decir, quedaría:

```
{:nam "Objeto 0" :val 1 :vol 3 :ratio 1/3}
```

Ésta sería la representación de cada objeto individualmente.

Pero además, en las primeras fases del algoritmo, lo que hago es **ordenar todos los objetos en un array por orden de ratio**. Es decir, los objetos con mayor valor y menor volumen los primeros y los de menor valor y mayor volumen los últimos. La razón de esta ordenación es mejorar el rendimiento de los cálculos que se realizarán para determinar el valor de fitness de cada individuo a lo largo de la evolución.

De este modo, **el conjunto de objetos que tenemos** para meter en la mochila **acabaría representado** de un modo similar al siguiente:

```
[{:nam "Objeto 9" :val 10 :vol 3 :ratio 10/3}
{:nam "Objeto 5" :val 7 :vol 3 :ratio 7/3}
{:nam "Objeto 2" :val 8 :vol 4 :ratio 2}
...]
```

Tenemos que determinar además cómo representar a los individuos. Siguiendo las indicaciones decido **representar cada individuo como un vector de genes binarios**. Es decir, si el gen  $n$ -ésimo de un individuo es `True`, con  $0 \leq n < \text{número de objetos}$ , entonces ese individuo incluiría en la mochila el objeto  $n$ -ésimo. Y no lo incluiría si el gen hubiera tenido el valor `False`. Nótese que en lugar de  $[0, 1]$  utilizo  $[\text{True}, \text{False}]$ , sencillamente porque se me antoja más claro.

Por ejemplo, el individuo `[True False True...]` representa meter en la mochila el primer y tercer objetos (una vez reordenados por ratio, según se comentó anteriormente).

## 4.2 Funciones más relevantes

```
;; Genera un objeto a partir de sus propiedades.
;; - nam: nombre del objeto
;; - val: valor del objeto
;; - vol: volumen del objeto
(defn new-object [nam val vol]

;; Genera un objeto aleatoriamente con valor entre 1 y max-val y
;; volumen entre 1 y max-vol.
;; - i: número de objeto
;; - max-val: valor máximo
;; - max-vol: volumen máximo
(defn rand-object [i max-val max-vol]

;; Necesitamos una función que genere una estructura con los objetos iniciales
;; del problema optimizada para accesos y para economía de computación.
;; Para ello:
;; 1. enriquezco los objetos iniciales con el valor del ratio
;; (val/vol) para no tener que recalcularlo en cada acceso.
```

```
;; 2. ordeno en un array el conjunto resultante por ratio de mayor a menor
;; 3. genero un mapa con los índices de los objetos en el array (como claves)
;;     que apuntan a los propios objetos (como valores)
;; Parámetros:
;; - objects: los objetos iniciales definidos en el problema
(defn arrange-objects [objects]

;; Función que devuelve el valor de aptitud o conveniencia de un individuo y
;; el volumen de la mochila que rellena un individuo antes de desbordarla.
;; Procedimiento:
;; - se queda con los objetos que tiene en la mochila (gen true)
;;   ordenados por ratio descendente
;; - acumula los valores de los objetos, mientras quepan en la mochila
;; - devuelve el valor acumulado y el volumen acumulado
;; Parámetros:
;; - individual: el individuo
(defn fitness-and-volume [individual]

;; Función que devuelve el valor de aptitud o conveniencia de un individuo
;; Parámetros:
;; - individual: el individuo
(defn fitness [individual]

;; Función que devuelve el volumen de mochila que rellena un individuo
;; antes de desbordarla.
;; Parámetros:
;; - individual: el individuo
(defn volume [individual]

;; Devuelve la representación de un individuo como el conjunto de objetos
;; que introduce en la mochila.
(defn decode [individual]

;; Genera aleatoriamente un individuo.
(defn rand-individual []

;; Genera aleatoriamente una población.
(defn rand-population []

;; Selecciona el primer elemento de una lista con probabilidad stochastic-prob.
;; De no ser seleccionado, selecciona el primero del resto con probabilidad
;; stochastic-prob también. Y así hasta agotar la lista.
(defn first-stochastic [col]

;; Selecciona por torneo estocástico.
;; - population: población de la que se selecciona
;; - size: número de individuos a seleccionar
(defn tournament-stochastic [population size]
```

```
;; Cruza dos padres por un punto o devuelve los padres tal cual, dependiendo
;; de la probabilidad de cruce.
;; - parent1: primer padre
;; - parent2: segundo padre
(defn crossover-one-point [parent1 parent2]

;; Muta los genes de un individuo atendiendo a una probabilidad de mutación
;; data por 1/número de objetos del individuo.
;; - individual: el individuo a mutar
(defn mutate [individual]

;; Determina si se ha alcanzado el máximo de generaciones del problema.
;; - generation: la generación en curso
(defn too-much-generations? [generation]

;; Determina si se llevan demasiadas generaciones sin que aparezcan individuos
;; con fitness mejorado.
;; - generation: la generación en curso
;; - best: el mejor individuo de la generación
(defn too-much-idle? [generation best]

;; Determina si la evolución ha llegado a su fin, bien por haberse alcanzado
;; demasiadas generaciones, bien por llevar demasiadas generaciones sin que se
;; incremente el fitness del mejor individuo.
;; - generation: la generación en curso
;; - best: el mejor individuo de la generación
(defn done? [generation best]

;; Construye la nueva generación a partir de la generación actual
;; aplicando el modelo generacional.
;; - population: la población actual
(defn build-offspring [population]

;; Inicializa y lleva a cabo la evolución. Reporta el resultado.
;; Devuelve el mejor individuo encontrado.
;; - objects: los objetos a introducir en la mochila.
;; - config: mapa con la configuración del experimento.
;; La configuración adopta esta forma:
;; {:pack-size 500
;;  :rand-gen-prob 1/2
;;  :population-size 10
;;  :stochastic-prob 9/10
;;  :tournament-size 2
;;  :replacement true
;;  :crossover-prob 3/4
;;  :max-generations 100
;;  :idle-generations 5
```

```
;; :report-delta 1
;; :name "Nombre del experimento"}
;; Los objetos son un array con esta forma:
;; [{"nombre 1" valor-1 volumen-1}
;;  {"nombre 2" valor-2 volumen-2}
;;  ...
;; ]
(defn go-live [conf objs]

  ;; Inicializa y lleva a cabo la evolución. Reporta el resultado.
  ;; Devuelve el mejor individuo encontrado.
  ;; - path: ruta al fichero con objetos y configuración a utilizar
  ;; - config-override: parámetros de la configuración indicada en el fichero
  ;;                      que se desean sobrescribir en esta ejecución.
  ;;                      Tiene el mismo formato que config.
  ;; El fichero tiene que tener un formato como el siguiente:
  ;; {:config {:pack-size 500
  ;;           :rand-gen-prob 1/2
  ;;           :population-size 10
  ;;           :stochastic-prob 9/10
  ;;           :tournament-size 2
  ;;           :replacement true
  ;;           :crossover-prob 3/4
  ;;           :max-generations 100
  ;;           :idle-generations 5
  ;;           :report-delta 1
  ;;           :name "amanas: Todo desde fichero 1"}}
  ;; :objects [{"objeto 1" 150 9}
  ;;           [{"objeto 2" 120 8}
  ;;            ...]}
  (defn go-live-from-file [path & [config-override]]

    ;; Función que permite ejecutar el algoritmo invocando el jar ejecutable
    ;; desde una consola.
    ;; El comando para llamar al algoritmo es:
    ;; java -jar ejecutable.jar simple/complex tournament-size
    ;; Por ejemplo, se puede llamar con:
    ;; java -jar ejecutable.jar simple 5
    ;; Esta llamada ejecutará el experimento, cuya evolución puede verse en:
    ;; http://amanas.ml/ce/status.html
    ;; seleccionando en el combobox el experimento con nombre:
    ;; profe: simple - 5
    (defn -main [& [type tour :as args]]
```



## 5 Formas de ejecutar el algoritmo

Son tres, que describo a continuación.

### 5.1 Ejecución desde consola

En las indicaciones de la práctica se pide además proporcionar un ejecutable que se pueda lanzar sin necesidad de compilar el código o utilizar un IDE.

En mi caso, todo el proyecto lo compilo en un ejecutable java, **ejecutable.jar**, que adjunto.

La parametrización que permito en este tipo de ejecución del algoritmo es:

- **primer parámetro: el conjunto de datos utilizado, 'simple' o 'complex'**
- **segundo parámetro: el tamaño en la selección por torneo**

Por lo tanto, en un sistema en el que tengamos una máquina virtual java, podremos lanzar un ejecución del algoritmo que utilizase el conjunto de datos complex con tamaño de torneo, por ejemplo 7, haciendo esta simple llamada:

```
java -jar ejecutable.jar complex 7
```

Después, o mientras la ejecución ocurre, podemos visitar <http://amanas.ml/ce/status.html> y seleccionar en el combobox el experimento con nombre **"profe: complex - 7"**. De este modo se puede monitorizar la evolución del aprendizaje llamando al algoritmo desde consola.

### 5.2 Ejecución programática del algoritmo

Por último, es necesario indicar el modo de representación de los metaparámetros del algoritmo y la forma de iniciar la ejecución del mismo.

En la implementación que entrego **todos los metaparámetros del algoritmo son configurables** en el momento de su lanzamiento.

De este modo, la función que lleve a cabo la ejecución de un experimento es:

```
(defn go-live [conf objs]
  ...
```

a la que se le pasa un mapa con los parámetros de configuración y el conjunto de objetos.

Los parámetros de configuración son los siguientes:

```
{;; El tamaño de la mochila
:pack-size 500 ;; [100,10.000] [10.000,1.000.000]
;; Probabilidad de activación de los genes cuando se genera un individuo
:rand-gen-prob 1/2
;; Tamaño inicial de la población
:population-size 10 ;; [10,100]
;; Probabilidad utilizada para selección estocástica en el torneo
```

```

:stochastic-prob 9/10
;; Número de individuos seleccionados por ronda en selección por torneo
:tournament-size 2 ;; Máxima explotación
;; Torneo con o sin reemplazamiento
:replacement true
;; Probabilidad de mezcla en crossover
:crossover-prob 3/4 ;; [0.6,0.9]
;; Número de generaciones tope para el experimento
:max-generations 100
;; Número de generaciones sin mejora del fitness que se permiten
;; antes de dar por acabado el exp.
:idle-generations 5
;; Cada cuantas generaciones se reporta el estado al web-service
;; en la nube para su visualización.
:report-delta 1}

```

Por ejemplo, una ejecución del algoritmo de aprendizaje podría hacerse así:

```

;; Un conjunto de objetos de prueba
(def some-objects (map #(rand-object % 20 20) (range 200)))

(decode (go-live {:name "amanas: experimento 1"
                  :pack-size 500
                  :rand-gen-prob 1/2
                  :population-size 10
                  :stochastic-prob 9/10
                  :tournament-size 2
                  :replacement true
                  :crossover-prob 3/4
                  :max-generations 100
                  :idle-generations 5
                  :report-delta 1}
                  some-objects))

```

### 5.3 Ejecución desde fichero del algoritmo

Sin embargo, en las indicaciones sobre como realizar la práctica se hace saber que **sería recomendable que se pudiera proceder a la ejecución de un experimento leyendo tantos los datos como la configuración desde un fichero**. Por lo tanto, también se permite proceder de este modo haciendo una llamada como la siguiente:

```

;; Inicializa y lleva a cabo la evolución. Reporta el resultado.
;; Devuelve el mejor individuo encontrado.
;; - path: ruta al fichero con objetos y configuración a utilizar
;; - config-override: parámetros de la configuración indicada en el fichero
;;                   que se desean sobrescribir en esta ejecución.
;;                   Tiene el mismo formato que config.
;; El fichero tiene que tener un formato como el siguiente:

```

```

;; {:config {:pack-size 500
;;           :rand-gen-prob 1/2
;;           :population-size 10
;;           :stochastic-prob 9/10
;;           :tournament-size 2
;;           :replacement true
;;           :crossover-prob 3/4
;;           :max-generations 100
;;           :idle-generations 5
;;           :report-delta 1
;;           :name "amanas: Todo desde fichero 1"}
;; :objects [{"objeto 1" 150 9]
;;           ["objeto 2" 120 8]
;;           ...]}
(defn go-live-from-file [path & [config-override]]

```

Obsérvese que en este caso, para poder utilizar un mismo fichero con distintas configuraciones, se ofrece la posibilidad de utilizar el parámetro `config-override`, en el que se indicarían los parámetros y valores que queremos sobrescribir sobre aquellos indicados en la configuración del fichero.

Un ejemplo de fichero que cumpliría con los requisitos anteriores sería:

```

{:pack-size 400
 :population-size 5
 :tournament-size 5
 :replacement true
 :rand-gen-prob 5/10
 :stochastic-prob 8/10
 :crossover-prob 5/10
 :max-generations 200
 :idle-generations 20
 :report-delta 1
 :name "amanas: Rosseta Code desde fichero"
 :objects [{"map" 150 9]
           ["compass" 35 13]
           ["water" 200 153]
           ["sandwich" 160 50]
           ["glucose" 60 15]
           ...]}

```

Y un ejemplo de ejecución del algoritmo desde fichero sobrescribiendo parte de la configuración sería:

```

(go-live-from-file "resources/data/simple.edn"
  {:name "amanas: simple - tournament 5 - run 0"
   :tournament-size 5})

```

Y con estas aclaraciones, creo que ya podemos pasar a comentar las experimentaciones que he realizado siguiendo el guión de la práctica.

## 6 Evaluación Experimental

Tal como se indica en la práctica, he generado dos conjuntos de objetos, uno simple y otro complejo. He ido variando el número de individuos seleccionados en cada ronda de la fase de selección por torneo para así poder contrastar las implicaciones de tal parámetro en el rendimiento de la búsqueda de la solución.

Para cada tamaño de la selección por torneo, he ejecutado el algoritmo varias veces, de modo que pueda obtener medias de los rendimientos y poder así realizar un análisis más consistente.

Todas las ejecuciones que he realizado han quedado guardadas en el servicio que tengo publicado en la nube y que permite revisar tranquilamente tales ejecuciones. De todos modos, para evitar problemas de eliminado de caches, también he guardado los resultados en la carpeta “results” que se adjunta con este documento.

Por ejemplo, la Figura 2 muestra el resultado de la primera ejecución del algoritmo para el conjunto simple con 2 individuos en la selección por torneo.

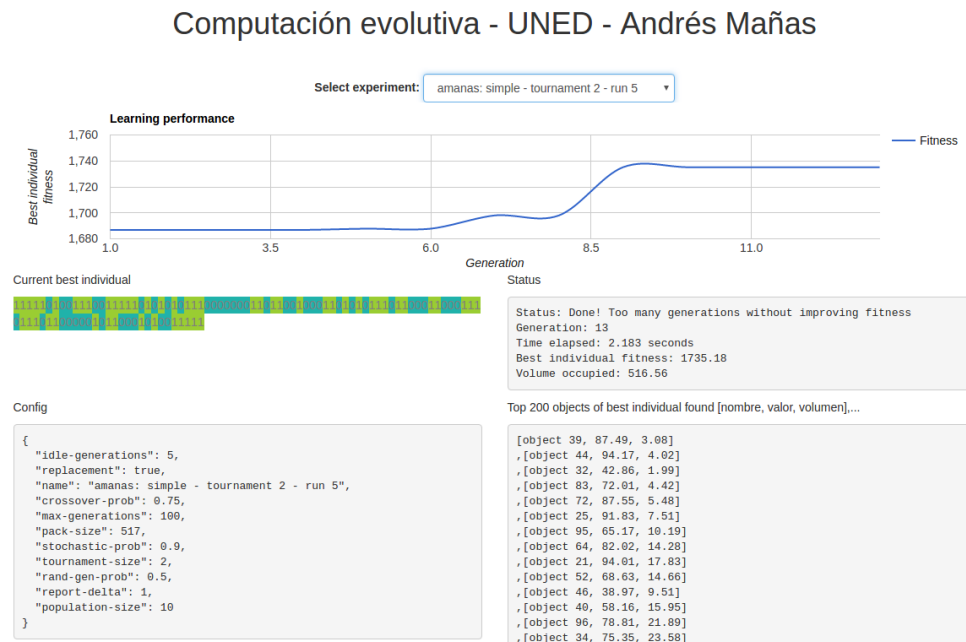


Figura 2. Caso simple con tamaño de torneo 2

### 6.1 Análisis de resultados

Con el siguiente script podemos agregar todos los resultados y así poder tomar medias. La forma más sencilla es descargarlos desde el servidor y procesarlos aquí. Para prevenir posibles accidentes (que se borre la caché del servidor, guardo una copia de los resultados en el proyecto).

```
In [2]: import os.path
import json
import requests
```

```

import pandas as pd

def get_exp_name(type,tournament,run):
    template='amanas: %s - tournament %s - run %s'
    return template % (type,tournament,run)

def get_exp_url(type,tournament,run):
    template='http://amanas.ml/ce/service.php?action=get&name=%s'
    return template % get_exp_name(type,tournament,run)

def get_exp_path(type,tournament,run):
    template='/home/ubuntu/ai/ce/pl/results/%s.json'
    return template % get_exp_name(type,tournament,run)

def get_exp_data(type,tournament,run):
    path=get_exp_path(type,tournament,run)
    if os.path.isfile(path):
        with open(path) as data:
            return json.load(data)
    url=get_exp_url(type,tournament,run)
    data=requests.get(url).json()
    with open(path, 'w') as fp:
        json.dump(data, fp)
    return data

def to_DataFrame(type,tournaments,runs):
    data = []
    for tour in tournaments:
        for run in runs:
            d = get_exp_data(type,tour,run)
            data += [[type,tour,run,d['best-volume'],
                        d['best-fitness'],d['generation'],
                        (d['current-time']-d['start-time'])/1000]]
    df=pd.DataFrame(data)
    df.columns=['type','tournament','run','best-volume',
                'best-fitness','generation','seconds']
    return df

```

## 6.2 Experimento simple

En el caso del experimento simple, he generado un fichero de datos que se puede consultar en “resources/data/simple.edn”. El experimento se caracteriza por:

- 10 individuos
- 100 objetos
- con valor y volumen aleatoriamente entre [1, 100]
- y con capacidad de la mochila un valor aleatorio en el intervalo real [100, 10.000]

Para cada ejecución del algoritmo, he utilizado estos mismos datos pero he ido cambiando dos propiedades de la configuración (el nombre del experimento y el tamaño de la ronda por torneo). Se han utilizado torneos con 2, 3, 4, 5, 6 y 10 individuos.

Así, finalmente he conseguido las siguientes ejecuciones, que se pueden consultar en <http://amanas.ml/ce/status.html>

Los resultados obtenidos son los siguientes:

```
In [29]: simpleDF=to_DataFrame('simple', [2,3,4,5,6,10], range(10))
        print(simpleDF)
```

	type	tournament	run	best-volume	best-fitness	generation	seconds
0	simple	2	0	514.36	1730.41	6	1.529
1	simple	2	1	511.48	1944.15	27	4.558
2	simple	2	2	495.85	1803.01	6	1.049
3	simple	2	3	505.16	1642.68	6	1.086
4	simple	2	4	513.63	1795.95	20	3.205
5	simple	2	5	516.56	1735.18	13	2.183
6	simple	2	6	514.69	1724.98	11	1.853
7	simple	2	7	509.69	1704.55	6	1.085
8	simple	2	8	514.03	1894.94	18	3.053
9	simple	2	9	508.41	1692.50	7	1.190
10	simple	3	0	513.95	1790.91	14	2.359
11	simple	3	1	514.26	1899.42	13	2.188
12	simple	3	2	501.17	1762.21	11	1.921
13	simple	3	3	510.52	1956.27	25	4.028
14	simple	3	4	516.74	1795.97	12	2.045
15	simple	3	5	509.72	1877.08	15	2.467
16	simple	3	6	511.43	1871.65	15	2.548
17	simple	3	7	516.72	1864.20	19	3.389
18	simple	3	8	505.94	1928.38	20	3.769
19	simple	3	9	512.89	1955.25	25	4.216
20	simple	4	0	516.82	1787.13	10	1.670
21	simple	4	1	511.05	1945.55	17	2.931
22	simple	4	2	510.03	1644.48	9	1.572
23	simple	4	3	505.66	1838.41	10	1.740
24	simple	4	4	510.63	1797.50	7	1.232
25	simple	4	5	498.33	1953.89	25	4.152
26	simple	4	6	515.46	1876.11	20	3.359
27	simple	4	7	511.04	1919.47	17	2.807
28	simple	4	8	488.69	1814.38	8	1.465
29	simple	4	9	510.22	1831.20	11	1.931
30	simple	5	0	506.22	1892.06	22	3.719
31	simple	5	1	504.32	2033.13	10	1.799
32	simple	5	2	513.53	1826.98	14	2.397
33	simple	5	3	510.57	1796.28	15	2.570
34	simple	5	4	513.27	1909.05	21	3.467
35	simple	5	5	514.99	1971.36	15	2.596
36	simple	5	6	499.70	1706.18	8	1.361

37	simple	5	7	513.23	1855.45	23	3.800
38	simple	5	8	507.73	1764.69	12	2.009
39	simple	5	9	502.25	1999.27	28	5.021
40	simple	6	0	512.14	1853.22	8	1.426
41	simple	6	1	515.56	1953.98	11	1.960
42	simple	6	2	515.62	1969.77	23	3.707
43	simple	6	3	514.40	1609.02	10	1.767
44	simple	6	4	509.69	1773.11	13	2.260
45	simple	6	5	509.93	1900.35	13	2.230
46	simple	6	6	513.30	1977.44	24	3.935
47	simple	6	7	514.98	1971.05	25	5.189
48	simple	6	8	490.83	1966.07	13	2.249
49	simple	6	9	514.17	1702.12	10	1.783
50	simple	10	0	507.03	1822.71	10	1.931
51	simple	10	1	513.47	1851.79	15	2.572
52	simple	10	2	505.85	1664.67	6	1.094
53	simple	10	3	516.29	1657.51	7	1.234
54	simple	10	4	513.45	1663.33	6	1.129
55	simple	10	5	515.88	1745.00	10	1.804
56	simple	10	6	516.68	1930.65	22	3.714
57	simple	10	7	516.02	1857.87	6	1.081
58	simple	10	8	515.78	1858.70	17	2.738
59	simple	10	9	499.57	1939.24	27	4.520

De dónde, agrupando por el tamaño del torneo:

```
In [30]: grouped=simpleDF.groupby('tournament')
```

podemos observar los valores medios agrupados por tamaño del torneo:

```
In [31]: print(grouped.mean())
```

	run	best-volume	best-fitness	generation	seconds
tournament					
2	4.5	510.386	1766.835	12.0	2.0791
3	4.5	511.334	1870.134	16.9	2.8930
4	4.5	507.793	1840.812	13.4	2.2859
5	4.5	508.581	1875.445	16.8	2.8739
6	4.5	511.062	1867.613	15.0	2.6506
10	4.5	512.002	1799.147	12.6	2.1817

y las desviaciones medias agrupadas por tamaño del torneo:

```
In [32]: print(grouped.std())
```

	run	best-volume	best-fitness	generation	seconds
tournament					
2	3.02765	6.150247	93.635257	7.423686	1.180946
3	3.02765	4.850416	68.657361	5.108816	0.869132
4	3.02765	8.428697	91.463693	5.985167	0.967190

5	3.02765	5.338875	105.796080	6.408328	1.111706
6	3.02765	7.422401	131.156494	6.429101	1.211516
10	3.02765	5.841436	108.915766	7.426679	1.196469

### 6.2.1 Curvas de evolución

En las figuras 3 y 4 vemos un par de ejecuciones (torneo tamaño 2 y torneo tamaño 10).

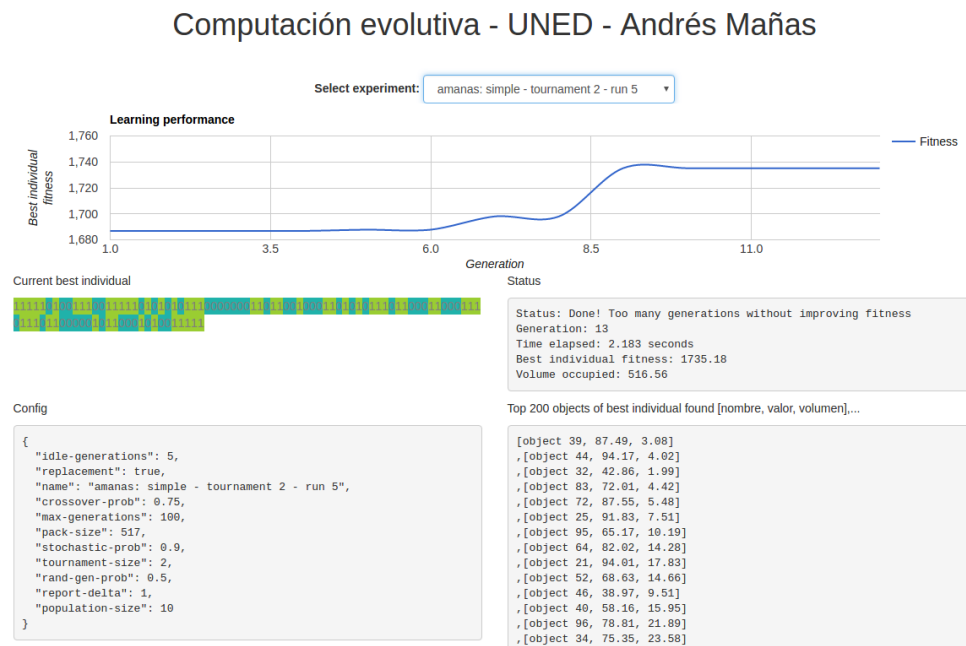


Figura 3. Caso simple con tamaño de torneo 2

### 6.2.2 Conclusiones del experimento simple

En el caso simple observo que **el tamaño del torneo parece condicionar el número de generaciones necesarias y la calidad de la solución encontrada.**

El tiempo que tarda cada ejecución parece mantenerse estable en torno a los 2.5 o 3 segundos y la desviación en torno al segundo.

**El número de generaciones necesarias parece situarse en torno a las 12 o 16.**

Nótese que hay que restar 5 generaciones de las que se muestran necesarias, porque es el umbral que tengo configurado para acabar la ejecución si no se mejora el fitness durante esas generaciones.

Curiosamente con tamaños de torneo 3, 4, 5 y 6 parece que se necesitan más generaciones para dar con una solución, que por otro lado es de mayor calidad (el mejor individuo tiene mejor fitness).



## Computación evolutiva - UNED - Andrés Mañas

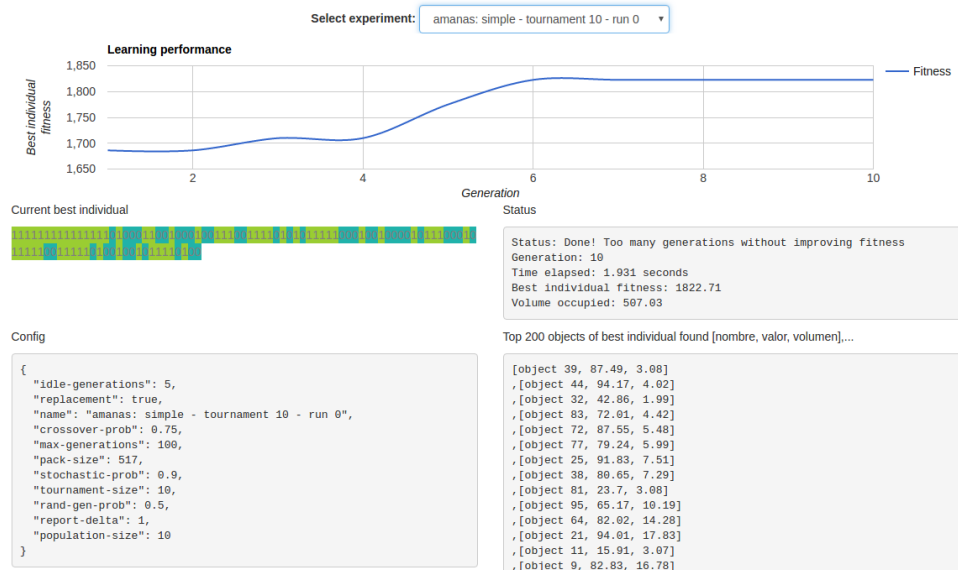


Figura 4. Caso simple con tamaño de torneo 10

Observo además que el mejor fitness parece encontrarse en los torneos que se disputan entre 3 y 6 individuos. Luego parece que el tamaño de torneo óptimo para este experimento estaría en el intervalo 3 a 6..

Por lo tanto, al amparo de los datos experimentales arrojados por mis pruebas, me aventuro a concluir que en experimentos pequeños (o al menos en este experimento), **el tamaño del torneo condiciona la calidad de la solución y no compromete el tiempo de ejecución**. Los torneos realizados con tamaño 3 y 5 brindan los mejores resultados.

**Maximizar el grado de exploración (torneo tamaño 2) genera resultados malos**, o al menos inferiores a otros que vemos en los resultados.

Igualmente, **abusar de la explotación (torneos con tamaños grandes) tampoco parece conducir a los mejores resultados**.

### 6.3 Experimento complejo

En el caso del experimento complejo, he generado un fichero de datos que se puede consultar en “resources/data/complex.edn”. El experimento se caracteriza por:

- 100 individuos
- 1.000 objetos
- con valor y volumen aleatoriamente entre [1, 100]
- y con capacidad de la mochila un valor aleatorio en el intervalo real [10.000, 1.000.000]

Para cada ejecución del algoritmo, he utilizado estos mismos datos pero he ido cambiando dos propiedades de la configuración (el nombre del experimento y el tamaño de la ronda por torneo). Se han utilizado torneos con 2, 3, 4, 5, 6, 10 y 25 individuos.

Así, finalmente he conseguido las siguientes ejecuciones, que se pueden consultar en <http://amanas.ml/ce/status.html>

Desgraciadamente, Clojure no es un lenguaje que destaque en rendimiento de computación, por lo que he tenido que disminuir un poco la dimensión del experimento complejo propuesto en la práctica. Por eso no puedo utilizar 10.000 objetos sino 1.000 y el tamaño de la mochila lo disminuye apropiadamente también. En otro caso, con el ordenador que tengo no creo que hubiera podido hacer los experimentos que presento a continuación.

Los resultados obtenidos son los siguientes:

```
In [38]: complexDF=to_DataFrame('complex',[2,3,4,5,6,10,25],range(5))
        print(complexDF)
```

	type	tournament	run	best-volume	best-fitness	generation	seconds
0	complex	2	0	15230.20	26157.12	47	83.605
1	complex	2	1	15221.31	27696.88	100	180.433
2	complex	2	2	15234.33	27883.82	100	183.731
3	complex	2	3	15218.90	25616.34	37	65.821
4	complex	2	4	15216.79	27786.92	100	181.075
5	complex	3	0	15224.27	28443.31	100	191.920
6	complex	3	1	15171.85	28812.17	100	198.455
7	complex	3	2	15193.73	28897.77	100	196.336
8	complex	3	3	15235.61	28485.35	92	174.801
9	complex	3	4	15223.07	28860.88	100	192.739
10	complex	4	0	15236.62	29266.92	100	208.551
11	complex	4	1	15235.68	29094.89	100	213.516
12	complex	4	2	15199.93	29057.25	100	217.622
13	complex	4	3	15201.55	28851.26	100	208.260
14	complex	4	4	15217.06	29353.02	100	218.401
15	complex	5	0	15233.59	29325.59	100	220.515
16	complex	5	1	15229.24	29155.91	100	225.437
17	complex	5	2	15224.58	29177.34	100	219.938
18	complex	5	3	15236.36	29387.19	100	214.369
19	complex	5	4	15212.81	29298.05	100	213.949
20	complex	6	0	15221.73	29458.21	100	216.820
21	complex	6	1	15210.05	29377.48	100	217.151
22	complex	6	2	15236.54	29073.40	100	214.551
23	complex	6	3	15192.91	29380.80	100	222.033
24	complex	6	4	15231.70	29328.41	100	218.795
25	complex	10	0	15232.78	29642.31	100	245.710
26	complex	10	1	15235.37	30042.45	100	246.510
27	complex	10	2	15234.64	29747.34	100	242.208
28	complex	10	3	15236.05	29646.31	100	246.072
29	complex	10	4	15221.48	29620.03	100	243.652
30	complex	25	0	15234.78	29987.07	100	339.744
31	complex	25	1	15225.75	30305.63	100	343.110
32	complex	25	2	15233.81	30282.58	100	344.320
33	complex	25	3	15224.89	30062.10	100	339.096
34	complex	25	4	15188.34	30308.28	100	339.794

De dónde, agrupando por el tamaño del torneo:

```
In [39]: grouped=complexDF.groupby('tournament')
```

podemos observar los valores medios agrupados por tamaño del torneo:

```
In [40]: print(grouped.mean())
```

	run	best-volume	best-fitness	generation	seconds
tournament					
2	2.0	15224.306	27028.216	76.8	138.9330
3	2.0	15209.706	28699.896	98.4	190.8502
4	2.0	15218.168	29124.668	100.0	213.2700
5	2.0	15227.316	29268.816	100.0	218.8416
6	2.0	15218.586	29323.660	100.0	217.8700
10	2.0	15232.064	29739.688	100.0	244.8304
25	2.0	15221.514	30189.132	100.0	341.2128

y las desviaciones medias agrupadas por tamaño del torneo:

```
In [41]: print(grouped.std())
```

	run	best-volume	best-fitness	generation	seconds
tournament					
2	1.581139	7.581407	1061.485232	31.964042	58.973746
3	1.581139	26.220379	217.681982	3.577709	9.357027
4	1.581139	17.728203	195.297285	0.000000	4.813955
5	1.581139	9.257004	99.002451	0.000000	4.781301
6	1.581139	17.580729	147.409213	0.000000	2.775852
10	1.581139	6.041261	176.240847	0.000000	1.830432
25	1.581139	19.085556	152.861653	0.000000	2.340153

### 6.3.1 Curvas de evolución

En las figuras 5 y 6 vemos un par de ejecuciones (torneo tamaño 2 y torneo tamaño 10).

### 6.3.2 Conclusiones del experimento complejo

En el caso complejo observo que **el tamaño del torneo parece que influye notablemente en la calidad de la solución pero incide considerablemente en el tiempo empleado para encontrarla**. A mayor tamaño de torneo, mayor fitness del mejor individuo encontrado.

**El tiempo que tarda cada ejecución crece considerablemente según se aumenta el tamaño del torneo** (más individuos de mayor tamaño a los que calcularles el fitness cuando participan en el torneo).

## Computación evolutiva - UNED - Andrés Mañas

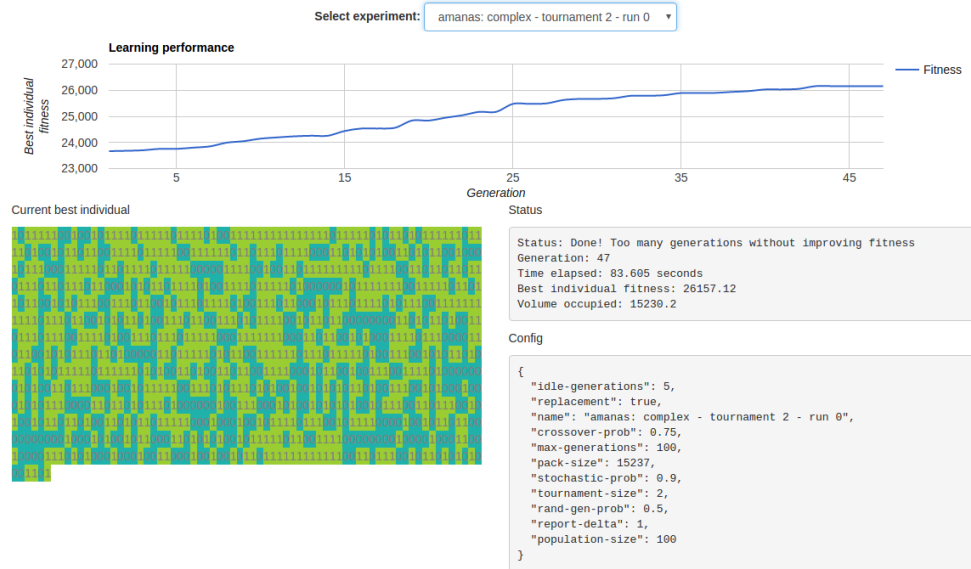


Figura 5. Caso complejo con tamaño de torneo 2

## Computación evolutiva - UNED - Andrés Mañas

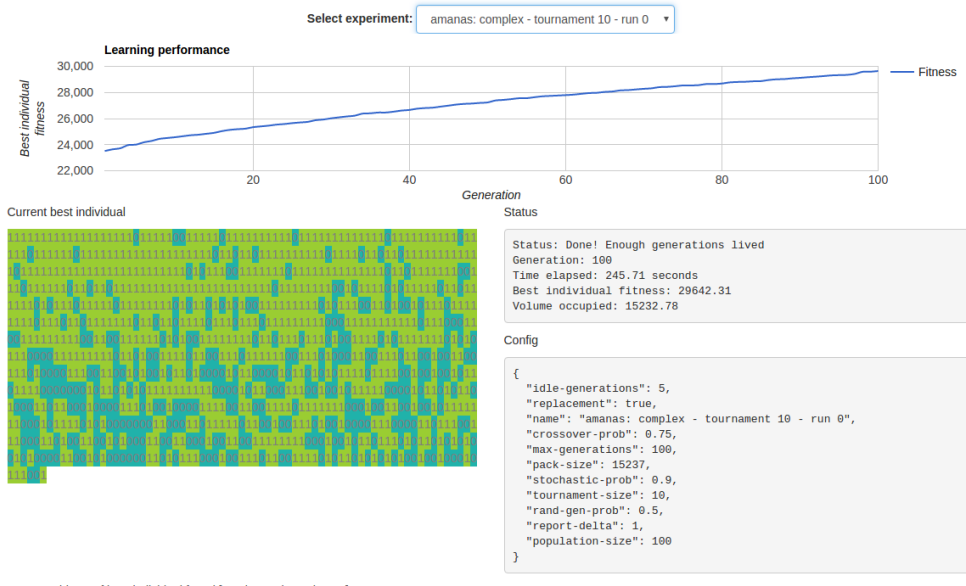


Figura 6. Caso complejo con tamaño de torneo 10

**El número de generaciones necesarias aumenta según el torneo se hace con mayor número de individuos.** Es decir, cuando el tamaño del torneo es mayor, el fitness sigue creciendo durante más tiempo y se llega a valores superiores a los obtenidos con torneos pequeños. Supongo que se traduce en la explotación de un máximo local.

Al amparo de los datos experimentales arrojados por mis pruebas, concluyo que en experimentos complejos la **explotación** (tamaño de torneo mayor) parece conducir a resultados mejores (individuos con mayor fitness). Sin embargo, el coste computacional es muy elevado.

**Es decir, evolucionar explotando la calidad genética de los mejores individuos en problemas complejos parece conducir a mejores soluciones, o al menos encuentra mejores soluciones (quizá no óptimas).**

Además, en nuestro ejemplo, si atendemos a la función decodificadora y al modo con se definen genéticamente los individuos (los genes a la izquierda se corresponde con objetos de mayor ratio), **parece intuitivo pensar que las mejores soluciones serán aquellas con mayor concentración de genes activos a la izquierda.** Curiosamente, en el caso del experimento complejo vemos que **cuando el tamaño del torneo es mayor, este fenómeno se presenta con bastante claridad.** Obsérvense que no ocurre esto en la figura 7 (torneo de tamaño 2) y sí ocurre en las figuras 8 y 9 (tamaños 10 y 25 respectivamente).

*Nota: las figuras 7, 8 y 9 representan el genoma de tres individuos. La secuencia genética debe interpretarse como una fila que aparece truncada en varias por motivos de representación en este documento. O dicho de otro modo, el gen de la fila 3 y columna 7 se corresponde con el gen  $2 \cdot 72 + 7 = 151$  del objeto.\**

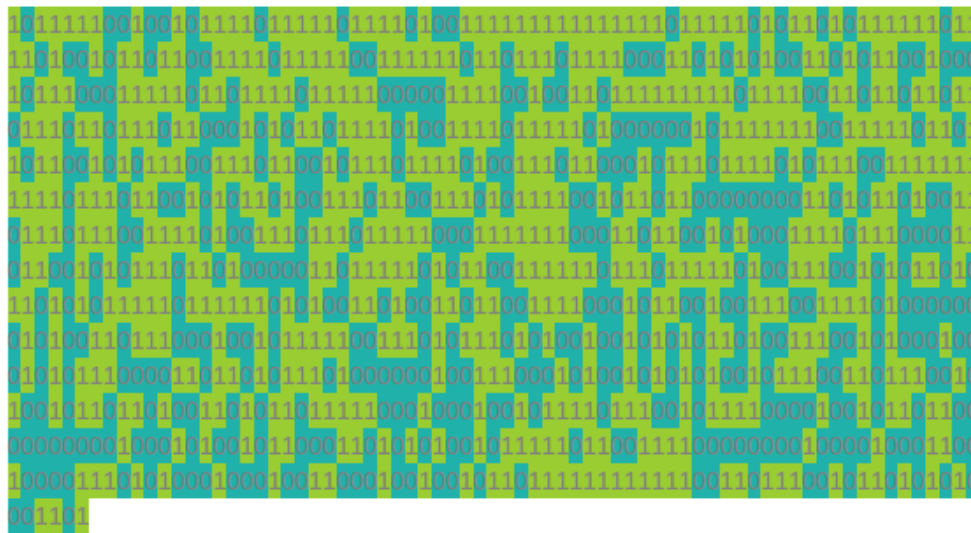


Figura 7. Caso complejo - genoma del mejor individuo con tamaño de torneo 2

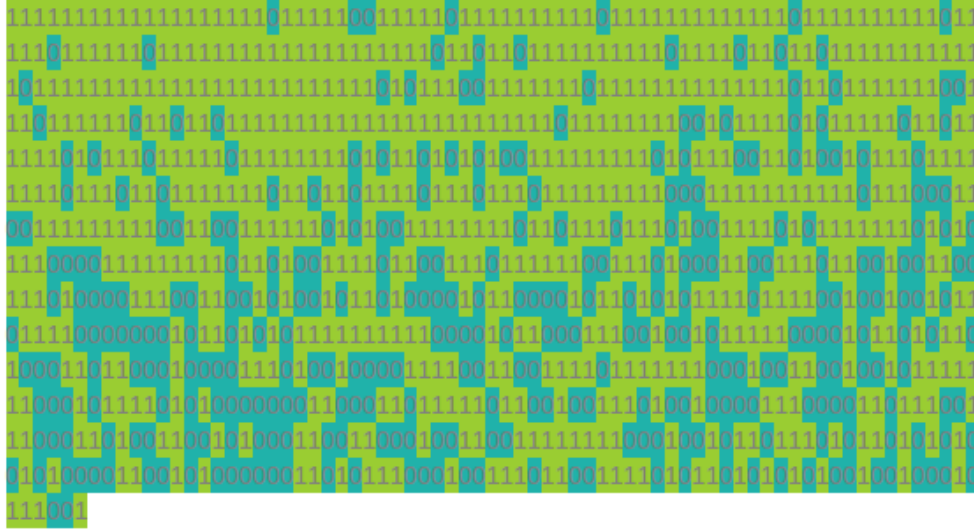


Figura 8. Caso complejo - genoma del mejor individuo con tamaño de torneo 10

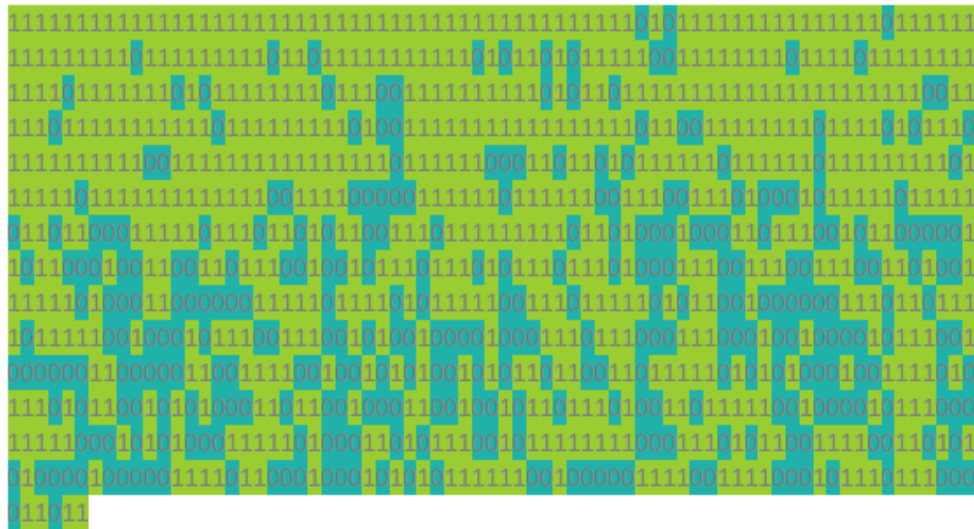


Figura 9. Caso complejo - genoma del mejor individuo con tamaño de torneo 25