Sistemas Inteligentes 1

Práctica 2
Algoritmos Genéticos. Operadores

Héctor Garbisu Arocha
Curso 2015-2016
Grado en Ingeniería Informática
Universidad de las Palmas de Gran Canaria

Índice

1. Introducción

Esta práctica hace énfasis en comparar varios operadores de selección, mutación y cruce. A diferencia de la anterior, aquí se valora la capacidad de implementar múltiples operadores y de conocer sus diferencias para optimizar el algoritmo.

El problema presentado consiste en optimizar la secuencia de activación de ocho semáforos situados dos a dos en cuatro intersecciones, de forma que los dos semáforos situados en la misma intersección tienen valores opuestos (verde/rojo). Las cuatro intersecciones son el resultado de cruzar perpendicularmente dos parejas de carreteras paralelas, formando una especie de '#'.
Este entramado es modelado por un autómata celular.

El esquema del programa en general es más o menos como en la práctica 1, con la salvedad de que no se muestran las soluciones 'decodificadas', ya que esto implicaría mostrar el progreso de las intersecciones a lo largo de muchos ciclos.

2. Codificación de cromosomas

Como los semáforos son alternativos dos a dos, sólo hace falta codificar la mitad de ellos, y poner en los otros cuatro los valores opuestos.
Un cromosoma será una secuencia de tamaño fijo de estos conjuntos de cuatro valores. Asumiendo, como dice el guión de la práctica, que necesitamos 12 cuartetos para simular 2 minutos de ejecución y que esos 2 minutos pueden repetirse durante toda la duración de la simulación, un cromosoma puede quedar bien codificado por una matriz de 4x12 valores binarios.

Como Matlab permite usar arrays multidimensionales muy fácilmente, una población será un array(4,12,tampop) donde tampop es una característica ajustable del algoritmo.

Cromosoma aleatorio:

```
ans =

     0     0     1     1     1     0     0     1     1     1     0     0
     0     1     0     1     0     1     1     1     1     0     0     1
     0     0     0     1     1     1     0     1     1     1     1     1
     1     0     1     0     0     0     0     1     1     0     0     0
```

3. Función de fitness

Para una población de 'tampop' individuos la función de fitness devuelve un vector a lo largo de la tercera dimensión, coincidiendo con la dimensión en la que se distribuyen los diferentes individuos dentro de la población.
El valor de fitness está medido en proporción a la cantidad de vehículos que salen del autómata en cada iteración. Está ajustado para que sea cercano a 1 en el mejor caso y 0 en el peor caso.
Por la geometría del problema, es imposible que salgan simultáneamente coches de más de dos calles diferentes, por eso el fitness se escala x2. Además es imposible que salgan dos coches en iteraciones consecutivas por la misma calle, de modo que el fitness se escala otra vez x2.

```
fitn = output./(10*numpasos*ciclos*4);
fitn = 4*fitn;
```

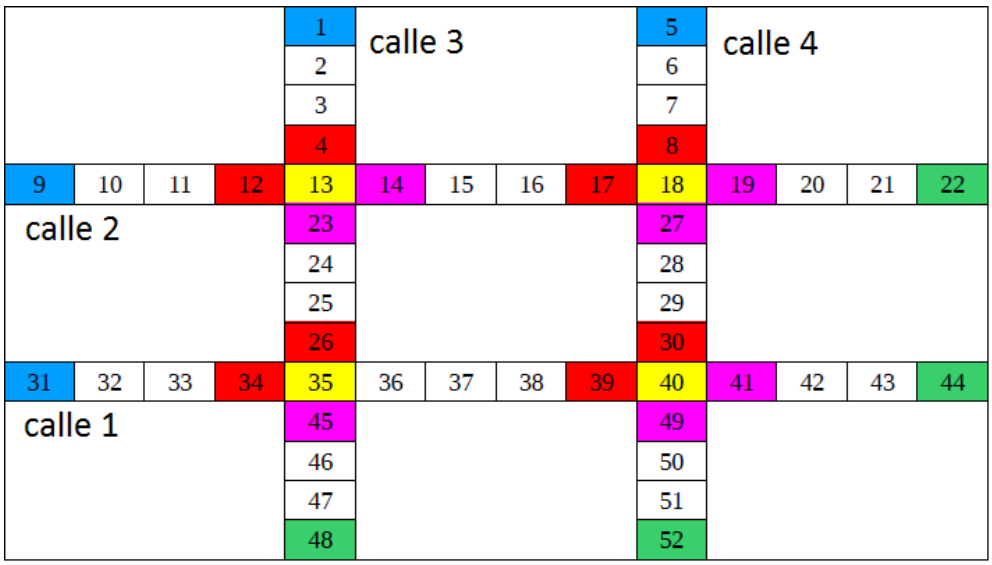
Donde:
output es un vector de tampop elementos, cada uno de ellos siendo la suma de todos los cohces que han salido en toda la simulación.
Numpasos es el numero de pasos por cada ciclo (12 para 2 minutos)
ciclo es el numero de veces que se repite una secuencia de pasos (60 para 2h)

Para calcular el número de coches que salen en toda la simulación (output) se genera un mapa del problema al que he llamado 'terreno', acompañado por una matriz escasa de semáforos a la que he llamado 'puede_pasar' y una matriz 'cola' que indica cuántos coches están esperando para entrar por las entradas.

3.1. Simulación/Decodificación de cromosomas

La matriz terreno (en realidad es un tensor de profundidad tampop) se actualiza en cada iteración siguiendo las reglas de las celdas.

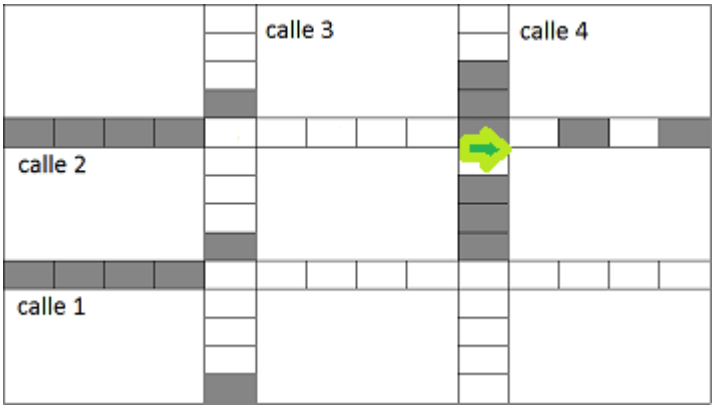
En terreno, cada fila codifica una carretera, de modo que las intersecciones están representadas en las columnas 5 y 10 y los semáforos, en las columnas 4 y 9.
La forma en la que se indica que un coche dentro de una intersección está apuntando en una dirección y no en otra, es simplemente que hay un uno en la calle que corresponde a la dirección en que apunta.
El criterio que he seguido para numerar las calles es muy simple. Mirando las entradas, de abajo a la izquierda de 1 a 4 en sentido horario. Lo ilustraré con un ejemplo.



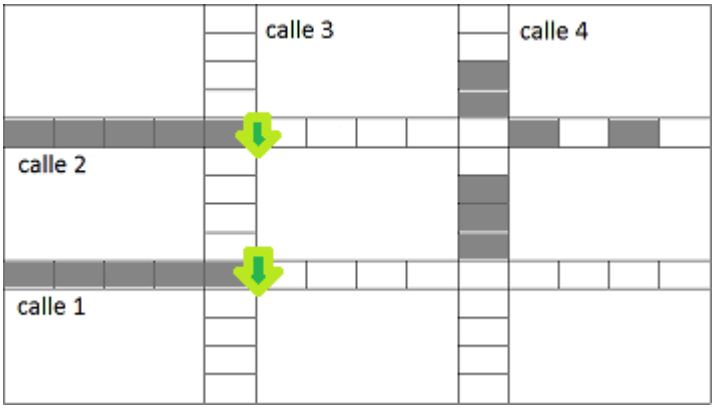
Durante una simulación uno de los cromosomas produce esta secuencia de terrenos, intersecciones en amarillo.

```
ans =

     1     1     1     1     0     0     0     0     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     1     0     1     0     1
     0     0     0     1     0     0     0     0     1     0     0     0     0     1
     0     0     1     1     0     0     1     1     1     0     0     0     0     0


ans =

     1     1     1     1     0     0     0     0     0     0     0     0     0     0
     1     1     1     1     0     0     0     0     0     0     1     0     1     0
     0     0     0     0     1     0     0     0     0     1     0     0     0     0
     0     0     1     1     0     0     1     1     1     0     0     0     0     0
```

Si dibujamos cómo serían las carreteras según esa matriz, sería así.



Se sabe que el coche de la intersección (2, 4) va a la derecha porque su bit está en la fila 2.



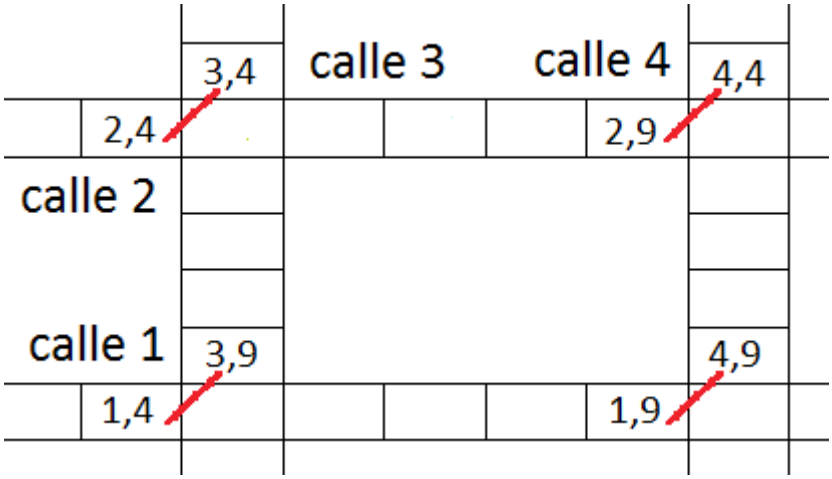
En el siguiente paso se puede ver cómo los coches del final de la calle 2 han avanzado cada uno una casilla, uno de ellos abandonando el terreno. También podemos adivinar qué semáforos estaban activados viendo qué coches se han movido y cuáles no.

3.2. Actualización de estados

Semáforos

La función de fitness simula la actividad del terreno durante 2 horas. Por defecto, se hace un total de 7200 iteraciones.
Cada 5 iteraciones se suma 1 a las colas
Cada 10 segundos se actualiza el valor de los semáforos
Cada segundo se actualizan todas las celdas.

Los semáforos están ligados entre ellos por parejas de una forma específica. Por ejemplo, el primer semáforo de la primera calle, está vinculado con el segundo de la tercera calle. Las relaciones, marcadas en rojo, son las siguientes:



Por eso, los valores del cromosoma se le asignan a cuatro semáforos que no estén vinculados entre ellos. En concreto {(1,4),(3,4),(2,9),(4,9)}.

```
puede_pasar([1 3],4,:) = pop([1 2],semstep,:);
puede_pasar([2 4],9,:) = pop([3 4],semstep,:);
```

Semstep aumenta cada 10 segundos para actualizar los semaforos.

Y luego se actualizan sus complementarios:

```
puede_pasar(2,4,:) = ~puede_pasar(3,4,:);
puede_pasar(4,4,:) = ~puede_pasar(2,9,:);
puede_pasar(1,9,:) = ~puede_pasar(4,9,:);
puede_pasar(3,9,:) = ~puede_pasar(1,4,:);
```

Colas

Las colas son mucho más sencillas. Simplemente, por cada cromosoma hay un vector de cuatro enteros, simbolizando el número de coches que quiere entrar en la simulación desde que la primera celda de su calle esté vacía.
En el bucle principal del cálculo de fitness se suma 1 a todas las colas cada 5 iteraciones y cada iteración se resta 1 a la calle que tenga un coche en su última celda.

Celdas

Las celdas se actualizan siguiendo una serie de fórmulas lógicas (la presencia de un coche es booleana) en función del estado de las celdas circundantes y la propia.
Por ejemplo, las celdas de entrada de cada calle dependen de su estado, del siguiente y de la cola, así que se actualizan con el siguiente código:

```
terreno(:,1,:) = ((cola(:,1,:)>0).*(~M(:,1,:)|M(:,2,:))) | M(:,1,:).*M(:,2,:);
```

Ahí se puede leer (con algo de imaginación) que una celda de entrada sólo tiene un coche ante dos situaciones; o bien hay un coche nuevo disponible para entrar:

```
((cola(:,1,:)>0).*(~M(:,1,:)|M(:,2,:)))
```

O bien ya hay un coche en esta celda pero no puede avanzar porque hay otro justo delante:

```
M(:,1,:).*M(:,2,:);
```

La fórmula para las celdas en las que hay semáforos es la más complicada porque se ha de tener en cuenta el estado actual, el de la celda anterior, la celda siguiente, la celda siguiente del semáforo conjugado, y el estado del semáforo actual.

Este es el código de actualización de la celda en (1,4)

```
terreno(1,4,:) = M(1,4,:).*(M(1,5,:)|M(3,10,:)|~puede_pasar(1,4,:));
terreno(1,4,:) = terreno(1,4,:)|(M(1,3,:).*(~M(1,4,:)|M(1,4,:).*(M(1,5,:)|
M(3,10,:))));
```

4. Métodos de selección

Para permitir cambiar fácilmente de operador durante las pruebas, las funciones de selección, cruce y mutación seleccionan mediante una variable qué algoritmo utilizar. El esquema para la selección es el siguiente:

```
function pop = sele(pop,fitn,repro,elite,metodo,generacion,maxgen)
if (nargin>4)
    pop = esca(pop,generacion,maxgen);
end
if(nargin<4)
    metodo='truncamiento'
end
if(nargin<3)
    elite = 0
end
if(nargin<2)
    repro = size(pop,3)
end
switch metodo
    case 'truncamiento'
        pop = sele_truncamiento(pop,repro,elite);
    case 'rwsr'
        pop = sele_rwsr(pop,fitn,repro,elite);
    case 'sus'
        pop = sele_sus(pop,fitn,repro,elite);
    case 'torneo'
        pop = sele_torneo(pop,fitn,repro,elite);
    otherwise
        metodo
end
end
```

A todas las subfunciones se les pasa la población, la cantidad de individuos no truncados y la cantidad de elitismo, por si se quiere implementar alguna de esas dos cosas.

Truncamiento

Este mecanismo es el más simple.

```
pop(:,:,repro+1:end) = pop_init(size(pop,1),size(pop,2),size(pop,3)-repro);
```

Y ya está. Sencillamente se sustituye a los truncados por unos nuevos aleatorios.

Roulette Wheel Selection with Replacement

Se prepara una matriz de pesos acumulados, de forma que el último elemento tiene un valor igual a la suma de todos los fitness. Se lanza un dado de 0 a el peso total y se escoge al primer elemento que sea mayor que el valor dado. Se repite el proceso tantas veces como individuos haga falta seleccionar, excluyendo a los élite, que tendrán su lugar asegurado.

```
function [pop]  = sele_rwsr(pop,fitn,repro,elite)
cumfitn = cumsum(fitn(:,:,elite+1:repro));
result = zeros(size(pop));
for i=1:1:repro-elite
    roll = rand().*cumfitn(:,:,end);
    k = find(cumfitn>=roll,1);
    result(:,:,i) = pop(:,:,k+elite);
```

```
end
pop(:,:,elite+1:repro) = result(:,:,1:repro-elite);
pop(:,:,repro+1:end) = pop_init(size(pop,1),size(pop,2),size(pop,3)-repro);
end
```

Stochastic Universal Selection

Conceptualmente, este método parecía más complicado que el anterior pero en realidad no lo es. Al necesitar sólo una tirada de número aleatorio, podría ahorrarse el bucle for, pero por consistencia y simplicidad lo he mantenido.
Se hace una tirada entre 0 y el tamaño de cada segmento. Cada segmento tiene un tamaño de 1/cantidad_candidatos. Se calculan n punteros, cada uno en un segmento, guardando una posición relativa a su propio segmento de exactamente el valor aleatorio anterior. Por cada puntero se calcula el primer candidato con un fitness mayor que el valor de dicho puntero.

```
function [pop]  = sele_sus(pop,fitn,repro,elite)
cumfitn = cumsum(fitn(:,:,elite+1:repro));
total = cumfitn(:,:,end);
result = zeros(size(pop));
tam_segmento = total/size(cumfitn,3);
roll = rand().*tam_segmento;
for i=1:1:repro-elite
    puntero = (i-1)*tam_segmento+roll;
    k = find(cumfitn>=puntero,1);
    result(:,:,i) = pop(:,:,k+elite);
end
pop(:,:,elite+1:repro) = result(:,:,1:repro-elite);
pop(:,:,repro+1:end) = pop_init(size(pop,1),size(pop,2),size(pop,3)-repro);
end
```

Los elementos menores que 'elite' se mantienen igual, los mayores se sustituyen por los recién extraídos, y los truncados se reemplazan por aleatorios.

Torneo

Primero se genera una lista aleatoriamente distribuida de índices que van de 1 hasta la cantidad de candidatos. Los candidatos ordenados aleatoriamente se emparejan de dos en dos y se selecciona el de mejor fitness. Como consecuencia, sólo se selecciona una cantidad de individuos que es la mitad de los candidatos.

```
function [pop]  = sele_torneo(pop,fitn,repro,elite)
k = randperm(repro-elite);
ksize = size(k);
if(mod(ksize,2)~=0)
   ksize = ksize-1;
end
result = zeros(size(pop));
for i=2:2:ksize
    oponente1f = fitn(k(i)+elite);
    oponente2f = fitn(k(i-1)+elite);
    if(oponente1f<oponente2f)
        result(:,:,i/2) = pop(:,:,k(i)+elite);
    else
        result(:,:,i/2) = pop(:,:,k(i-1)+elite);
    end
end
result = result(:,:,1:end/2);
pop(:,:,elite+1:elite+size(result,3)) = result(:,:,:);
```

```
pop(:,:,elite+size(result,3)+1:end) = pop_init(4,12,size(pop,3)-
(size(result,3)+elite));
end
```

5. Métodos de cruce

El esquema del cruce es semejante al de los métodos de selección.

OnePoint Crossover

El punto de cruce es un lugar aleatorio en una matriz cromosoma. La parte superior izquierda de ese punto proviene de un padre y el resto del cromosoma, del otro padre. Esta implementación hace que de media se herede más del segundo padre que del primero, aunque como los padres empiezan desordenados, el impacto de esta asimetría no es tanto.

```
function [hijos] = cruza_onepoint(padres)
hijos = a_desordena_padres(padres);
numpadres = size(hijos,3);
if mod(numpadres,2)~= 0 %Paridad forzada
    numpadres = numpadres-1;
    hijos(:,:,end) = pop_init(4,12,1);
end
for i = 2:2:numpadres
    dim1 = round(rand().*3+1);
    dim2 = round(rand().*11+1);
    hijos(1:dim1,1:dim2,i) = hijos(1:dim1,1:dim2,i-1);
    hijos(1:dim1,1:dim2,i-1) = hijos(1:dim1,1:dim2,i);
end
end
```

Uniforme

Se itera por los padres aleatorios de dos en dos y en cada par se decide qué gen proviene de cada padre.

```
function [hijos] = cruza_uniforme(pop)
padres = a_desordena_padres(pop);
numpadres = size(padres,3);
hijos = padres;
if mod(numpadres,2)~= 0 %Paridad forzada
    numpadres = numpadres-1;
    hijos(:,:,end) = pop_init(4,12,1);
end
for i = 2:2:numpadres
    for gen = 1:1:12
        if(rand()<0.5)
            hijos(:,gen,i-1) =  hijos(:,gen,i);
        end
        if(rand()<0.5)
            hijos(:,gen,i) = hijos(:,gen,i-1);
        end
    end
end
end
```

TwoPoint Crossover

Esta implementación es diferente a la anterior porque los cromosomas se dividen linealmente. En consecuencia, es más probable recibir la misma cantidad de información genética de un padre y de otro.

```
function [hijos] = cruza_twopoint(padres)
hijos = a_desordena_padres(padres);
numpadres = size(hijos,3);
if mod(numpadres,2)~= 0 %Paridad forzada
    numpadres = numpadres-1;
    hijos(:,:,end) = pop_init(4,12,1);
end
for i = 2:2:numpadres
    hijo1 = hijos(:,:,i-1);
    hijo2 = hijos(:,:,i);
    pto1 = floor(rand().*4.*12)+1;
    aux = floor(rand().*4.*12)+1;
    pto2=max(pto1,aux);
    pto1=min(pto1,aux);
    aux = hijo1;
    aux(pto1:pto2) = hijo2(pto1:pto2);
    hijos(:,:,i-1) = aux;
    aux = hijo2;
    aux(pto1:pto2) = hijo1(pto1:pto2);
    hijos(:,:,i) = aux;
end
end
```

6. Métodos de mutación

BitString

Si un valor aleatorio es menor que la probabilidad de mutación se muta un cromosoma, oponiendo el valor de una posición aleatoria.

```
function [pop] = muta_bitstring(pop,pmut)
for i = 1:size(pop,3)
    if(rand()<=pmut)
        dim1 = round(rand().*3+1);
        dim2 = round(rand().*11+1);
        pop(dim1,dim2,i) = ~pop(dim1,dim2,i);
    end
end
end
```

Boundary

Si un cromosoma se decide elegido para mutar, se intercambia un gen y el siguiente (un gen sería una columna de semáforos).

```
function [pop] = muta_boundary(pop,pmut)
for i = 1:size(pop,3)
    if(rand()<=pmut)
        dim2 = round(rand().*10+1);
        pop(:,[dim2 dim2+1],i) = pop(:,[dim2+1 dim2],i);
    end
end
end
```

Hipermutación

Se hace un BitString con probabilidad de mutación inversamente proporcional a la generación actual. La probabilidad de mutación en la primera generación es la que se pasa por parámetro, por eso en este operador de mutación se espera que el valor pasado sea mayor que en otros casos.

```
function [pop] = muta_hipermut(pop,pmut,age,ngen)
    pmut = pmut*(1-age/ngen);
    pop = muta_bitstring(pop,pmut);
end
```

7. Pruebas y conclusión

Para terminar, probamos todas las combinaciones posibles de los operadores diferentes implementados. Los otros operadores tienen valores fijos:

- Criterio de parada: 10 generaciones
- Tamaño de la población: 60
- Truncamiento: Se truncan 18 individuos (42 reproductores)
- Elitismo: 1
- Probabilidad de mutación: Entre 0.6 y 0.06 para hipermutación y 0.1 para las otras

Para hacer las pruebas se ejecuta un script que rota por las tres categorías de variables y se apunta el mejor fitness que se ha podido obtener con cada combinación de operadores (36 combinaciones).

```
mostrar = false;
ngeneraciones = 10;
tampop = 60;
repro = 42;
elite = 1;
pmut = 0.1;
sel = {'truncamiento','rwsr','sus','torneo'};
cruce = {'onepoint' 'twopoint' 'uniforme'};
puntuacion = [];
contador = 1;
for s=sel
    selo=s{1};
for c=cruce
    cruceo=c{1};
    pmut=0.1;
    mutao = 'boundary';
    SGA2
    puntuacion = [puntuacion;contador, s, c, mutao, best(end), avrg(end)]
    contador = contador+1;
    mutao = 'bitstring';
    SGA2
    puntuacion = [puntuacion;contador, s, c, mutao, best(end), avrg(end)]
    contador = contador+1;
    pmut=0.6;
    mutao = 'hipermut';
    SGA2
    puntuacion = [puntuacion;contador, s, c, mutao, best(end), avrg(end)]
    contador = contador+1;
end
end
```

El objeto puntuación tiene 36 filas con los resultados

```
puntuacion =
   [ 1]    'truncamiento'    'onepoint'    'boundary'     [0.8954]    [0.7380]
                                                                ...
   [14]    'rwsr'            'twopoint'    'bitstring'    [0.8975]    [0.7515]
                                                                ...
   [36]    'torneo'          'uniforme'    'hipermut'     [0.9140]    [0.3888]
```

Para analizar rápidamente (muy, muy por encima) los resultados, podemos agrupar las puntuaciones por cada categoría haciendo la media, lo que da las siguientes cantidades:

Tipo de operador	Operador	Suma de fitness
Selección	Truncamiento	0.8886
	RWSR	0.8894
	SUS	0.8840
	Torneo	0.6516
Cruce	One Point	0.8867
	Two Point	0.8744
	Uniforme	0.8871
Mutación	Btistring	0.8755
	Boundary	0.8853
	Hipermutación	0.8874

A simple vista se puede apreciar que todos los valores son muy parecidos. Esto se debe a que es muy poco probable obtener una configuración que dificulte la salida de coches del laberinto. Por ejemplo, haría falta una combinación de todo 0s, todo 1s o algunas pocas más para poder tener un fitness de exactamente 0; ésto es menos probable que uno entre un billón.

Lo único que llama la atención es la selección por torneo, pero es una implementación que seguramente está muy afectada por el hecho de que en realidad se está duplicando el truncamiento y ésto puede estar debilitando el algoritmo más que el propio método de selección.

En conclusión:
Personalmente me parece que el diseño del cromosoma y la función de fitness han sido mucho más relevantes para la obtención de individuos que los otros operadores.

Me gustaría haber podido implementar varios escalados, cruce semiuniforme y aplicar reemplazo en la selección por torneo y RWSR.

FICHEROS ADJUNTOS:

- Se adjuntan todos los ficheros Matlab con el código original.
- Un gráfico en png con el progreso del fitness para una ejecución con `'sus'`,`'onepoint'` y `'bitstring'`.
- Ficheros en txt con los resultados de la prueba.