Metodología de la Programación Paralela

Silvia Perez Ruiz
E-mail: silvia.perezr@um.es, DNI: 49215974E
Raúl Hernández Martínez
E-mail: raul.hernandezm@um.es, DNI: 24423648V
Eduardo Gallego Nicolás
E-mail: eduardo.gallegon@um.es, DNI: 48844607J

Práctica 0. Implementación de un Algoritmo Genético

Grado en Ingeniería Informática
Metodología de la Programación Paralela
Convocatoria de
Profesor: Jose Matias Cutillas Lozano (josematias.cutillas@um.es)
Entrega: 06/10/2023

UNIVERSIDAD DE
MURCIA

Silvia Pérez Ruiz
Raúl Hernández Martínez
Eduardo Gallego Nicolás

Índice

1 Ejercicio 1.

Enunciado: Detectar los errores de memoria presentes en el código mediante el uso de la herramienta Valgrind (solo la usaremos en esta Práctica 0). Mostrar el informe devuelto por Valgrind e indicar cómo se han solucionado los errores.

Para poder comprobar el código con Valgrind primero hemos hecho un make sec y a partir del ejecutable obtenido hemos realizado el siguiente comando:

```
> valgrind --tool=memcheck --leak-check=yes ./sec 3 10 2
```

Como la salida de este comando es muy larga, vamos a ir separando los errores de uno en uno. El primero que nos da es este:

```
==7387== Memcheck, a memory error detector
==7387== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7387== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==7387== Command: ./sec 3 10 2
==7387== Parent PID: 6913
==7387==
==7387== Invalid write of size 8
==7387==    at 0x109570: generar_matriz_distancias (io.c:20)
==7387==    by 0x109296: main (main.c:30)
==7387==  Address 0x4a95060 is 32 bytes inside a block of size 36 alloc'd
==7387==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so
    )
==7387==    by 0x1094D3: generar_matriz_distancias (io.c:14)
==7387==    by 0x109296: main (main.c:30)
==7387==
==7387== Invalid write of size 8
==7387==    at 0x10953A: generar_matriz_distancias (io.c:20)
==7387==    by 0x109296: main (main.c:30)
==7387==  Address 0x4a95068 is 4 bytes after a block of size 36 alloc'd
==7387==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so
    )
==7387==    by 0x1094D3: generar_matriz_distancias (io.c:14)
==7387==    by 0x109296: main (main.c:30)
```

Listing 1: Salida de Valgrind con errores.

Como podemos ver en el informe, Valgrind nos indica que la función `generar_matriz_distancias` da error porque se está intentando acceder a una zona de memoria que no está reservada y que está ocurriendo 4 bytes después de la que tenemos reservada.

Si nos vamos a la función vemos que este es su código:

```
double *generar_matriz_distancias(int n)
{
  double f;
  double *d = (double *) malloc(n * n *sizeof(int));

  srand(time(NULL) + getpid());
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      f = (double) rand() / ((double) RAND_MAX + 1);
      *(d + (i*n + j)) = (j != i) ? (min + f*(max - min)) : 0.0;
    }
  }
  return d;
}
```

Listing 2: Código función generar$_{m}atriz_{d}istancias$.

Si ahora nos fijamos en la línea 4 vemos que está haciendo una reserva de memoria para un entero, pero lo está guardando en una variable de tipo `double`. Este es el error que estaba saltando ya que un `double` ocupa el doble que un entero es por eso que da el error que se está accediendo a una zona no reservada.
Para solucionarlo simplemente tenemos que modificar esa línea para que ponga `double` en vez de `int`.

```
1   double *d = (double *) malloc(n * n *sizeof(double));
```

Listing 3: Linea corregida.

El siguiente error que nos da es este:

```
1  ==9646== Invalid read of size 4
2  ==9646==    at 0x109690: printf (stdio2.h:112)
3  ==9646==    by 0x109690: print_solution (io.c:45)
4  ==9646==    by 0x10935E: main (main.c:55)
5  ==9646==  Address 0x4a9a0b0 is 0 bytes inside a block of size 12 free'd
6  ==9646==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
7  ==9646==    by 0x10934A: main (main.c:52)
8  ==9646==  Block was alloc'd at
9  ==9646==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so
       )
10 ==9646==    by 0x1092AA: main (main.c:37)
```

Listing 4: Salida de Valgrind con errores.

Como podemos ver en el informe, nos indica que se esta intentado leer cuatro bytes de memoria que no deberían de ser leídos y nos dice que esto se esta produciendo en la función `print_solution()`. Este error podemos ver que sucede al mirar el código:

```
1  void print_solution(int n, const int *solucion, double valor)
2  {
3      printf("\nSolution: ");
4      for(int i = 0; i <= n; i++) {
5          printf("%d ", solucion[i]);
6      }
7      printf("\nDistance: %.0lf\n", valor);
8  }
```

Listing 5: Código función $\text{print}_s olution$.

Como podemos ver cuando se recorre el bucle `for` vemos que la condición de parada es cuando `i` sea menor o igual que `n` esto lo que hace es que se recorra una vez más el bucle provocando ese error en el valgrind. El código corregido se quedaría así:

```
1  void print_solution(int n, const int *solucion, double valor)
2  {
3      printf("\nSolution: ");
4      for(int i = 0; i < n; i++) {
5          printf("%d ", solucion[i]);
6      }
7      printf("\nDistance: %.0lf\n", valor);
8  }
```

Listing 6: Código función $\text{print}_s olution solucionado$.

Si miramos unas líneas más abajo del informe podemos ver que nos dice que se esta intentando acceder a una dirección de memoria que ya había sido liberada.

Ahora tenemos que buscar donde se llama a esta función y ver cuál de las variables se esta liberando antes de tiempo.
Si nos vamos al `main.c` podemos ver que se hace un `free(sol)` y justo después se llama a la función

`print_solution()` con la variable `sol`. Esto se puede ver en el código que se muestra a continuación:

```
int main(int argc, char **argv)
{
....
   // Free Allocated Memory
    free(sol);

    #ifdef DEBUG
        print_solution(n, sol, value);
    #endif
 ....
}
```

Listing 7: Código del main.

Para poder solucionar este error simplemente tenemos que mover el `free(sol)` justo después de la llamada a la función `print_solution()`. El código solucionado sería así:

```
int main(int argc, char **argv)
{
....
    #ifdef DEBUG
  print_solution(n, sol, value);
    #endif

    // Free Allocated Memory
    free(sol);
 ....
}
```

Listing 8: Código del main arreglado.

Los últimos errores a solucionar son los relacionados con la liberación de la memoria, Valgrind nos muestra esta salida:

```
==9099== HEAP SUMMARY:
==9099==     in use at exit: 88 bytes in 7 blocks
==9099==   total heap usage: 10 allocs, 3 frees, 1,144 bytes allocated
==9099==
==9099== 16 bytes in 1 blocks are definitely lost in loss record 1 of 3
==9099==    at 0x4848899: malloc (vg_replace_malloc.c:381)
==9099==    by 0x1094D3: generar_matriz_distancias (io.c:14)
==9099==    by 0x109296: main (main.c:30)
==9099==
==9099== 72 (48 direct, 24 indirect) bytes in 3 blocks are definitely lost in loss record 3
    of 3
==9099==    at 0x4848899: malloc (vg_replace_malloc.c:381)
==9099==    by 0x109879: aplicar_ga (ga.c:76)
==9099==    by 0x1092F0: main (main.c:44)
==9099==
==9099== LEAK SUMMARY:
==9099==    definitely lost: 64 bytes in 4 blocks
==9099==    indirectly lost: 24 bytes in 3 blocks
==9099==      possibly lost: 0 bytes in 0 blocks
==9099==    still reachable: 0 bytes in 0 blocks
==9099==         suppressed: 0 bytes in 0 blocks
```

Listing 9: Salida de Valgrind con errores.

Esta salida nos indica que al hacer esta llamada `double *d = generar_matriz_distancias(n);`, la variable `d` se reserva con un `malloc` es por eso que al final del código `main.c` habría que liberarlo tal que así: `free(d);`.

El siguiente error que nos da es que no se está liberando bien en la función `aplicar_ga()` una variable. Si nos fijamos en el código vemos que la variable población se declara como una matriz pero se está liberando como un array normal tal que así `free(poblacion);`. Para solucionar

este error hay que cambiar ese código por este:

```
double aplicar_ga(const double *d, int n, int n_gen, int tam_pob, int *sol, double m_rate)
{
....
    // se libera la memoria reservada
  for(i = 0; i < tam_pob; i++) {
    free(poblacion[i]->array_int);
    free(poblacion[i]);
  }

  free(poblacion);
 ....
}
```

Listing 10: Código de aplicar$_g aarreglado$.

El último error es el que nos daba por no poner en su sitio correcto la liberación de la variable `solucion` y como ya lo hemos solucionado antes no hace falta volverlo hacer. Comentar también que aparte de esos errores, dan otros de cosas sin inicializar que eso se arreglaría con los siguientes ejercicios al completar las funciones y demás.

Con todo esto hecho, si volvemos hacer el mismo comando que antes, el informe que nos sale es este:

```
==3818== Memcheck, a memory error detector
==3818== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3818== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3818== Command: ./sec 3 10 2 0.15
==3818== Parent PID: 2765
==3818==
==3818==
==3818== HEAP SUMMARY:
==3818==     in use at exit: 0 bytes in 0 blocks
==3818==   total heap usage: 8 allocs, 8 frees, 1,180 bytes allocated
==3818==
==3818== All heap blocks were freed -- no leaks are possible
==3818==
==3818== For lists of detected and suppressed errors, rerun with: -s
==3818== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Listing 11: Salida de Valgrind sin errores.

2 Ejercicio 2.

Enunciado: Implementar las funciones cruzar(), mutar() y fitness() contenidas en el fichero ga.c. Los cambios realizados en el código deben ir acompañados de comentarios. En la memoria se explicará cómo se ha implementado cada una de las funciones, justificando las decisiones tomadas. El código mostrado en la memoria debe coincidir con el que se entregue.

2.1 Función cruzar.

```
1  void cruzar(Individuo *padre1, Individuo *padre2, Individuo *hijo1, Individuo *hijo2, int n)
2  {
3      // Elegir un punto (o puntos) de corte aleatorio a partir del que se realiza el
       intercambio de los genes.
4      int puntoCorteAleatorio = aleatorio(n-1);
5
6      // En este for se copia el array de cada padre en el array del hijo, es decir, se cruzan
       .
7      for(int i = 0 ; i<puntoCorteAleatorio ; i++){
8          hijo1->array_int[i] = padre1->array_int[i];
9          hijo2->array_int[i] = padre2->array_int[i];
10     }
11
12     // Para seguir cruzandolo ahora se coge de un punto aleatorio y se vuelve a copiar lo
       del padre en lo del hijo.
13     for(int i = puntoCorteAleatorio ; i<n ; i++){
14         hijo1->array_int[i] = padre2->array_int[i];
15         hijo2->array_int[i] = padre1->array_int[i];
16     }
17
18 }
```

Listing 12: Código cruzar.

Para poder entender mejor lo que hace, lo hemos ilustrado con una imagen. En el primer `for` dejaría así los dos arrays. Copiaría en el hijo lo del padre por el punto aleatorio, como podemos ver en la imagen 1.

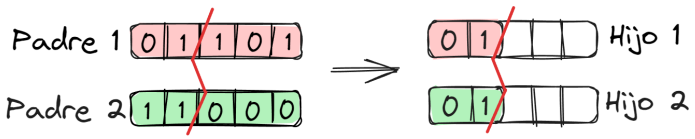

Figura 1: Primer cruce

En el segundo bucle se rellena la segunda parte del hijo con lo que haya en la otra parte del otro padre. Y se quedaría como se muestra en la imagen 2

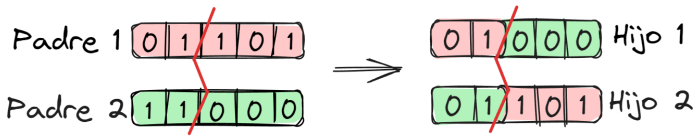

Figura 2: Segundo cruce

2.2 Función mutar.

```c
void mutar(Individuo *actual, int n) {
    double m_rate = 0.15;
    int i, j;

    srand(time(NULL));

    // Seleccionar dos posiciones aleatorias (i, j) tal que i < j
    i = rand() % (n - 1) + 1;
    j = rand() % (n - i) + i + 1;

    // Realizar la mutaci n: invertir el orden de los elementos entre i y j
    while (i < j) {

        // Determinar si mutar o no basado en la tasa de mutaci n
        double random_value = ((double)rand() / RAND_MAX);

        // Si el valor aleatorio sacado es mayor que el predeterminado entonces sucede la mutaci n
        if (random_value > m_rate) {
            // Una vez que ya estamos aqui dentro tenemos que intercambiar i con j, para ello guardamos i en una variable temporal para ponerla luego en el lugar de j.

            int temp = actual->array_int[i];
            actual->array_int[i] = actual->array_int[j];
            actual->array_int[j] = temp;
        }
        // Avanzamos para seguir mutando.
        i++;
        j--;
    }
}
```

Listing 13: Código mutar.

Para poder entender mejor el código vamos a ilustrarlo también con imágenes. En la imagen 8 podemos ver como de un array se eligen los valores i y j para poder hacer la mutación.

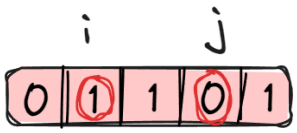

Figura 3: Elección de puntos de mutación

A continuación, ocurria lo de la tasa de probabilidad, para poder ilustrarlo mejor vamos a suponer que siempre se cumple la mutación. A continuación, se intercambiarían i y j, i avanzaría una posición del array y j retrocedería una posición. Esto lo muestra la figura 4.

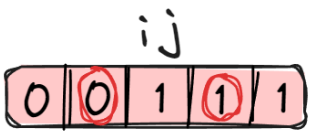

Figura 4: Mutación hecha

Como ahora i y j están en la misma posición no se cumpliría que `i < j` y se saldría del bucle.

2.3 Función fitness.

```
1  void fitness(const double *d, Individuo *individuo, int n)
2  {
3      // Determina la calidad del individuo calculando la suma de la distancia entre cada par
       de ciudades consecutivas en el array
4      individuo->fitness=0;
5
6      for(int i=0 ; i < n ; i++){
7          // Se calcula la diferencia absoluta entre el valor d y el valor que haya en el
       array del individuo
8          individuo->fitness += abs((d - individuo->array_int[i]));
9      }
10 }
```

Listing 14: Código fitness.

En esta función lo único interesante para resaltar es que al final del bucle `for` vamos a tener en el valor fitness del individuo la suma total de todas las diferencias absolutas.

3 Ejercicio 3.

Enunciado: Introduce un nuevo parámetro que represente la tasa de mutación, m_rate, que será leído del fichero de entrada in.txt, y que sustituirá la misma variable definida en la función mutar del algoritmo, facilitándose así la experimentación con este parámetro. Habrá que modificar también los ficheros run.sh y Makefile así como los argumentos de algunas funciones del programa. Entonces, en el fichero de entrada habrá cuatro columnas: n, n_gen, tam_pob, m_rate codificando, por ejemplo, los valores 100 150 100 0.15.

Hemos modificado los ficheros run.sh y Makefile. Lo que hemos hecho es añadir una cuarta variable, aquí están los códigos modificados:

Añadimos la cuarta variable **rate** y la añadimos la imprimimos por pantalla

```
#!/bin/bash

make sec

in_file="../input/in.txt"
out_file="../output/out.txt"

while read line ; do

  if [[ $line != \#* ]]
  then
    n=$(echo $line | tr -s ' ' | cut -f1 -d ' ')
    gen=$(echo $line | tr -s ' ' | cut -f2 -d ' ')
    tam=$(echo $line | tr -s ' ' | cut -f3 -d ' ')
    rate=$(echo $line | tr -s ' ' | cut -f4 -d ' ')

    echo -e >> $out_file
    echo -n "Executing with: " >> $out_file
    echo -e "N = "$n" N_GEN = "$gen" TAM_POB = "$tam M_RATE = "$rate" >> $out_file
    make test_sec N=$n N_GEN=$gen T_POB=$tam M_RATE=$rate
  fi
done < $in_file
```

Listing 15: Código run.sh.

Creamos la variable dentro del Makefile.

```
CC = gcc
CFLAGS = -O3 -Wall -std=c99 -g

N =
N_GEN =
T_POB =
M_RATE =

EXEC = sec
OUTFILE = ../output/out.txt
C_FILES = main.c io.c ga.c

sec: $(C_FILES)
  $(CC) $(CFLAGS) $(C_FILES) -o $(EXEC) -DTIME -DDEBUG

test_sec:
  ./$(EXEC) $(N) $(N_GEN) $(T_POB) $(M_RATE) >> $(OUTFILE)

clean:
  rm -f $(EXEC)
```

Listing 16: Código Makefile.

Ampliamos el número de argumentos del main y lo añadimos a la entrada.

```
int main(int argc, char **argv)
{
```

```
3  //  Check Number of Input Args
4    if(argc < 4) {
5      fprintf(stderr,"Ayuda:\n");
6      fprintf(stderr,"  ./programa n nGen tamPob mRate\n");
7      return(EXIT_FAILURE);
8    }
9
10   int n = atoi(argv[1]);
11   int n_gen = atoi(argv[2]);
12   int tam_pob = atoi(argv[3]);
13   double m_rate = atof(argv[4]);
```

Listing 17: Código Main.

4 Ejercicio 4.

Enunciado: Cambiar el criterio de convergencia del algoritmo evolutivo de forma que se realice en función del valor de fitness. El algoritmo finalizará cuando el valor de fitness no mejore el obtenido en la iteración anterior en un valor porcentual establecido. Prueba con valores como 2 %, 5 %, 10 % u otros que estimes oportunos. Los resultados obtenidos con cada uno deben reflejar tanto el tiempo empleado en obtener la solución como valor final de fitness alcanzado.

Obtenemos la mejora porcentual calculando la mejora entre el valor del fitness de la generación anterior y la generación actual de la siguiente manera:

```
double mejora_porcentual = ((fitness_anterior - poblacion[0]->fitness) / (fitness_anterior))
     * 100.0;
```

Listing 18: Mejora porcentual.

Se verifica si la mejora porcentual es menor que "m_rate". Si es así, el bucle en el que se encuentra este código se romperá, lo que significa que el programa dejará de ejecutarse. Esto indica que la mejora en el fitness ha alcanzado un nivel por debajo del umbral establecido en m_rate.

```
if (mejora_porcentual < m_rate) {
    break;
}
```

Listing 19: Comprobación mejora porcentual.

```
raul@raul-NBLK-WAX9X:~/Escritorio/P0/src$ ./sec 100 150 100 0.02
Generacion 0 - Fitness = 4601
Diferencia con Fitness Anterior = 7.05e+00%
Generacion 1 - Fitness = 4541
Diferencia con Fitness Anterior = 1.30e+00%
Generacion 2 - Fitness = 4442
Diferencia con Fitness Anterior = 2.18e+00%
Generacion 3 - Fitness = 4278
Diferencia con Fitness Anterior = 3.69e+00%
Generacion 4 - Fitness = 4154
Diferencia con Fitness Anterior = 2.90e+00%
Generacion 5 - Fitness = 4082
Diferencia con Fitness Anterior = 1.73e+00%
Generacion 6 - Fitness = 3980
Diferencia con Fitness Anterior = 2.50e+00%
Generacion 7 - Fitness = 3878
Diferencia con Fitness Anterior = 2.56e+00%
Generacion 8 - Fitness = 3878
Diferencia con Fitness Anterior = 0.00e+00%
Execution Time: 0.01 sec

Solution: 0 66 86 17 1 43 44 9 94 33 73 27 5 21 20 40 16 37 38 14 18 45 44 6 85 14 24 11 75
 12 9 5 2 73 17 35 96 37 28 7 55 4 20 23 46 13 92 93 7 58 19 41 55 17 86 82 31 94 24 57 51
43 22 42 18 16 9 14 12 49 4 50 47 97 91 63 37 79 70 71 34 13 39 8 54 14 77 1 8 66 56 5 6 84
 71 70 1 58 24 60
Distance: 3878
```

Figura 5: Prueba con valor 2 %

Es evidente que el tiempo de ejecución apenas muestra variaciones para esos porcentajes específicos. Esto se explica por el hecho de que, en todos los casos, el programa ejecuta más o menos el mismo número de generaciones. Esto ocurre porque el porcentaje establecido es alto, y podemos observar que cuanto mayor es el porcentaje, menos generaciones hace. Para aumentar el número de generaciones, sería necesario reducir significativamente el valor porcentual. Cuando este valor es 0, el programa se ejecuta en su totalidad.

```
raul@raul-NBLK-WAX9X:~/Escritorio/P0/src$ ./sec 100 150 100 0.05
Generacion 0 - Fitness = 4652
Diferencia con Fitness Anterior = 6.02e+00%
Generacion 1 - Fitness = 4494
Diferencia con Fitness Anterior = 3.40e+00%
Generacion 2 - Fitness = 4417
Diferencia con Fitness Anterior = 1.71e+00%
Generacion 3 - Fitness = 4236
Diferencia con Fitness Anterior = 4.10e+00%
Generacion 4 - Fitness = 4111
Diferencia con Fitness Anterior = 2.95e+00%
Generacion 5 - Fitness = 3748
Diferencia con Fitness Anterior = 8.83e+00%
Generacion 6 - Fitness = 3748
Diferencia con Fitness Anterior = 0.00e+00%
Execution Time: 0.01 sec

Solution: 0 93 16 37 61 19 39 9 59 47 11 66 38 24 6 21 28 75 16 65 3 2 63 95 37 31 39 35 5
68 13 53 93 19 26 45 59 22 66 76 29 23 14 35 25 46 24 30 77 21 54 11 18 41 64 15 82 5 52 20
 65 6 29 18 45 38 67 68 10 13 21 75 15 3 38 18 86 7 73 36 59 18 26 21 96 32 28 3 20 5 31 22
 53 70 58 77 16 23 14 79
Distance: 3748
```

Figura 6: Prueba con valor 5 %

```
raul@raul-NBLK-WAX9X:~/Escritorio/P0/src$ ./sec 100 150 100 0.1
Generacion 0 - Fitness = 4476
Diferencia con Fitness Anterior = 9.58e+00%
Generacion 1 - Fitness = 4350
Diferencia con Fitness Anterior = 2.82e+00%
Generacion 2 - Fitness = 4164
Diferencia con Fitness Anterior = 4.28e+00%
Generacion 3 - Fitness = 4164
Diferencia con Fitness Anterior = 0.00e+00%
Execution Time: 0.00 sec

Solution: 0 46 26 6 96 37 14 27 7 23 37 58 76 85 93 33 27 19 25 11 35 36 66 54 64 23 16 7 1
4 15 62 10 79 86 96 89 48 9 67 53 30 40 87 6 29 43 76 28 4 57 84 71 1 10 29 39 83 13 10 79
16 48 91 34 65 14 19 38 95 72 96 43 90 80 66 12 59 52 11 35 20 15 5 77 1 50 30 17 42 3 21 3
2 14 76 69 18 19 78 39 8
Distance: 4164
```

Figura 7: Prueba con valor 10 %

```
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 138 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 139 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 140 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 141 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 142 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 143 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 144 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 145 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 146 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 147 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 148 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Generacion 149 - Fitness = 3223
Diferencia con Fitness Anterior = 0.00e+00%
Execution Time: 0.04 sec

Solution: 0 50 9 56 20 5 65 1 18 15 6 86 44 71 15 21 3 75 27 58 8 29 41 77 9 59 3 29 19 13 36 88 4 1 87 96 68 8 3 52 43 47 10
6 35 34 32 14 42 15 10 29 20 36 25 94 18 27 2 20 20 31 82 11 35 49 4 65 65 4 49 35 11 82 31 20 20 2 27 37 43 16 54 67 3 5 99 3
2 2 8 1 21 10 49 15 45 26 31 47 35
Distance: 3223
```

Figura 8: Prueba con valor 0 %