

# Computación Neuronal y Evolutiva

(Práctica 05)



# UNIVERSIDAD DE BURGOS

Autores:

Mario Ubierna San Mamés

Jorge Navarro González

## Índice de Contenido

Descripción de la carga y almacenamientos de los datos .....	4
Descripción de cómo se ha adaptado la computación evolutiva a la solución del problema: .....	5
¿Cómo se representan los individuos? .....	5
¿Cómo se calcula la adaptación? .....	6
Restricciones.....	6
MultiObjetivo.....	7
Ventajas e inconvenientes del genotipo y el fitness .....	9
Fenotipo que nos devuelve el algoritmo .....	10
Restricciones.....	10
MultiObjetivo.....	12
Experimentos.....	14
Restricciones .....	15
Caso 1: .....	15
Caso 2: .....	15
Caso 3: .....	15
Caso 4: .....	16
Caso 5: .....	16
Caso 6: .....	16
Caso 7: .....	16
Caso 8: .....	17

Cambiando el número de generaciones .....	18
Cambiando el número de población inicial .....	20
Cambiando la probabilidad de cruce .....	22
Cambiando la probabilidad de mutación .....	24
MultiObjetivo .....	26
Caso 1: .....	26
Caso 2: .....	26
Caso 3: .....	26
Caso 4: .....	27
Caso 5: .....	27
Caso 6: .....	27
Caso 7: .....	27
Caso 8: .....	28
Cambiando el Número de Generaciones .....	29
Cambiando la Población Inicial .....	31
Cambiando la Probabilidad de Cruce .....	33
Cambiando la Probabilidad de Mutación .....	35

## Descripción de la carga y almacenamientos de los datos

Para realizar la carga de datos hemos usado un fichero al igual que la practica anterior llamado LecturaDatos.py, que está contenido en el zip.

El cómo guardamos los datos, estarán contenidos una vez leídos en variables que van a ser las siguientes:

- **Tiempo:** que es una variable de tipo int en la que guardaremos el tiempo que sacamos del fichero, que tendremos para recorrer el mayor número de ciudades.
- **Punto\_inicio:** que va a ser una lista, en la cual vamos a tener 4 datos, el primero va a ser el id de la ciudad, el segundo y tercero van a ser la latitud y la longitud y por último guardaremos en el cuarto la puntuación que vamos a tener.
- **Punto\_fin:** que es básicamente lo mismo que el punto de inicio, pero tendremos en este caso la información correspondiente al último punto de todos.
- **Punto\_visita:** que va a ser una lista de listas en la cual vamos a tener la lista grande y dentro sublistas de tamaño 4, en las cuales las posiciones serán las mismas que las de las dos variables anteriores.

Estas serían todas las variables en las que guardamos los datos necesarios que leemos desde cualquiera de los ficheros que tenemos con el formato correspondiente.

## Descripción de cómo se ha adaptado la computación evolutiva a la solución del problema:

### ¿Cómo se representan los individuos?

El genotipo que hemos pensado nosotros para dicho problema es un vector de números enteros, cuyo tamaño va a ser variable y va a contener números comprendidos 1 (el primero de todos que sería el de inicio no le contamos) y el tamaño del array de punto\_visita (el último que siempre va a ser el mismo al igual que el inicio no le contamos).

Si tuviéramos el punto de inicio y fin y luego tuviéramos por ejemplo 5 puntos a los que poder visitar, una población de individuos sería la siguiente:

- Cada punto al que poder visitar tiene unas id que irán des de 1 a 5.

Los posibles genotipos que tendríamos serían del siguiente tipo:

Genotipo1:

2	3	5	4	1
---	---	---	---	---

Genotipo2:

3	1	4	2
---	---	---	---

En los cuales como podemos ver que tendríamos una lista de enteros que puede ser de distinto tamaño con los puntos a los que vamos a visitar, al igual que hemos dicho en el genotipo no están incluidos el punto de inicio y el punto de fin.

En cuanto al significado, sería el siguiente:

- El genotipo1 va a pasar por las ciudades de id 2, 3, 5, 4, 1.
- El genotipo2 va a pasar por las ciudades de id 3, 1, 4, 2.

## ¿Cómo se calcula la adaptación?

### Restricciones

Para nuestro ejercicio realizado con restricciones tendremos nuestra función de fitness que es la que va cogiendo las puntuaciones de todas las ciudades por las que va pasando, obviamente hay que penalizar a los individuos que no vayan a ser válidos en nuestro problema, por lo tanto, lo que hacemos es emplear el deltaPenalty con el fin de dar un valor de fitness a esos individuos que no son válidos.

Nuestro fitness, como vamos a poder ver a continuación en una imagen, le pasamos como parámetro un genotipo, y lo que hace con ese genotipo es recorrerlo e ir quedándose con la puntuación que tiene ese punto al que visitamos, el fitness, lo que nos va a devolver será la puntuación total que ha conseguido ese camino.

```
def fitness(genotipo):
    ind = []
    for i in genotipo:
        ind = creator.Individual(list(dict.fromkeys(genotipo).keys()))

    puntuacion = 0
    for visitado in ind:
        for i in dv.punto_visita:
            if i[0] == visitado:
                puntuacion+=i[3]
    #puntuacion/=penalizar_genotipo(genotipo)
    return puntuacion,
```

Como vemos esta es nuestra función de fitness, al principio sí que teníamos una función que nos servía para penalizar, pero finalmente vamos a penalizar con el deltaPenalty como vamos a ver a continuación.

```
toolbox.decorate("evaluate", tools.DeltaPenalty(Evaluacion.esValido, 0, Evaluacion.distancia))
```

Con esto es con lo que penalizamos, para ello creamos dos funciones, una que sirve para sacar la distancia y otra para saber si es válido o no, las cuales se lo vamos a pasar a la función de penalizar que vemos arriba con el fin de penalizar todos los individuos que no van a ser correctos.

```
def esValido(genotipo):
    return tiempoGenotipo(genotipo) <= dv.tiempo

def distancia(genotipo):
    return tiempoGenotipo(genotipo) - dv.tiempo
```

La función tiempo genotipo es la que nos va a decir cuánto tiempo tardamos en recorrer todos los puntos por tanto para saber si es válido miramos a ver si es menor que el tiempo total que nos da y para calcular la distancia pues lo que hacemos es restar ese tiempo entre el que nos dan en el fichero.

Esta sería nuestra función de tiempoGenotipo con la que calculamos todo el tiempo que tardamos:

```
def tiempoGenotipo(genotipo):
    primero = genotipo[0]
    ultimo = genotipo[-1]
    pre = None
    km = 0
    for visitado in genotipo:
        for i in dv.punto_visita:
            if i[0] == visitado:
                temp = i
                break

        if temp[0] == primero:
            km += math.sqrt(abs(temp[1]-dv.punto_inicio[1])**2+abs(temp[2]-dv.punto_inicio[2])**2)
            pre = temp
        elif temp[0] == ultimo:
            km += math.sqrt(abs(dv.punto_fin[1]-temp[1])**2+abs(dv.punto_fin[2]-temp[2])**2)
            pre = temp
        else:
            km += math.sqrt(abs(temp[1]-pre[1])**2+abs(temp[2]-pre[2])**2)
            pre = temp
    return km/5
```

Como vemos hacemos la distancia Euclídea que nos da entre dos puntos, por ello nos quedamos primero con el primer punto y con el último que son los que nunca van a cambiar, además también nos quedamos con el punto anterior para sacar la distancia entre el siguiente que toca y ese anterior que hemos sacado.

## MultiObjetivo

En este apartado hacemos prácticamente lo mismo que hacíamos en el apartado anterior, con la diferencia de que ahora el fitness va a devolver dos valores, la puntuación y la resta que sería en nuestro caso cuánto falta o cuánto sobra de tiempo, nuestro atributo resta hace, valga la redundancia, la resta entre el tiempo que tarda nuestro genotipo y el tiempo que nos dan por fichero, por tanto lo que busca nuestro algoritmo va a ser maximizar la puntuación y minimizar lo máximo posible la resta, de tal manera que intentemos conseguir la máxima puntuación en el mínimo tiempo, como nuestra resta, si da negativo es porque sobra tiempo, multiplicamos en la función de penalizar por -1, con el fin de que al final podamos maximizar ese valor.

```

def fitness(genotipo):
    ind = []
    for i in genotipo:
        ind = creator.Individual(list(dict.fromkeys(genotipo).keys()))

    puntuacion = 0
    for visitado in ind:
        for i in dv.punto_visita:
            if i[0] == visitado:
                puntuacion+=i[3]
    suma_pen, resta = penalizar_genotipo(ind)
    #puntuacion/= suma_pen
    return puntuacion, resta

def penalizar_genotipo(genotipo):
    suma_pen = 1

    tiempo_genotipo = tiempoGenotipo(genotipo)
    resta = tiempo_genotipo - dv.tiempo

    if resta > 0:
        suma_pen+=resta/dv.tiempo
    if suma_pen != 1:
        suma_pen*=1
    return suma_pen, resta*-1

```

No penaliza realmente, pero lo hemos dejado como lo teníamos en la anterior práctica. Esta función llama a tiempoGenotipo que devuelve el tiempo que tarda en hacer todos los km.

```

def tiempoGenotipo(genotipo):
    ind = []
    for i in genotipo:
        ind = creator.Individual(list(dict.fromkeys(genotipo).keys()))

    primero = ind[0]
    ultimo = ind[-1]
    pre = None
    km = 0
    for visitado in ind:
        for i in dv.punto_visita:
            if i[0] == visitado:
                temp = i
                break

        if temp[0] == primero:
            km += math.sqrt(abs(temp[1]-dv.punto_inicio[1])**2+abs(temp[2]-dv.punto_inicio[2])**2)
            pre = temp
        elif temp[0] == ultimo:
            km += math.sqrt(abs(dv.punto_fin[1]-temp[1])**2+abs(dv.punto_fin[2]-temp[2])**2)
            pre = temp
        else:
            km += math.sqrt(abs(temp[1]-pre[1])**2+abs(temp[2]-pre[2])**2)
            pre = temp
    return km/5

```



## Ventajas e inconvenientes del genotipo y el fitness

Algunas de las ventajas que podemos comentar de usar nuestro genotipo es que al introducir valores aleatorios entre el rango de puntos que se visitan, no es nada difícil de implementar y realizar.

Esto anterior, aunque puede ser una ventaja también va a ser un inconveniente ya que aunque sea más fácil de realizar, vamos a ver que nos van a quedar en ocasiones algunos valores que están repetidos, por tanto vamos a tener que hacer una función que nos elimine esos valores repetidos, ya que no se puede pasar dos veces por el mismo punto.

Aun así, tenemos otra ventaja de nuestro genotipo que se hace visible cuando observamos una población, que es que los individuos van a ser de tamaño variable lo cual dota a la población de una mayor diversidad.

Algo negativo que decir a lo anterior es que puede que, al hacer cruces entre genotipos más pequeños, siempre empleamos el de menor tamaño por lo que podría ocurrir que al ser cada vez más pequeños los genotipos acabáramos teniendo un vector final que sea de un tamaño más pequeño del que debería.

En cuanto a nuestro fitness, es muy sencillo de implementar ya que solo tenemos que ir observando los puntos y cogiendo sus puntuaciones, por tanto, es algo sencillo, por el contra, si tenemos vectores muy grandes es verdad que tendría que recorrer todo el vector, cogiendo cada una de las puntuaciones y aumentándola por lo que podría tener una penalización de tiempo en el algoritmo, pero no supimos qué otro método emplear.

## Fenotipo que nos devuelve el algoritmo

### Restricciones

El fenotipo que nos va a devolver nuestro algoritmo va a ser un fichero en el cual tendremos todos los puntos por los que va pasando nuestro algoritmo y a la izquierda el orden en el que salen.

Fenotipo:

```
100 69.77768197660951|
0 0
1 2
2 13
3 40
4 64
5 34
6 8
7 16
8 53
9 41
10 20
11 24
12 18
13 27
14 11
15 37
16 63
17 22
18 56
19 51
20 60
21 3
22 31
23 62
24 32
25 49
26 58
27 7
28 12
29 59
30 33
31 52
32 50
33 1
34 17
35 4
36 5
37 57
38 35
39 6
40 23
41 14
42 9
43 65
```

Para hacer esto lo que hacemos es la siguiente función en la que imprimimos primero el tiempo que tenemos disponible y lo que tarda nuestro camino y también imprimimos el camino que tenemos.

```

def fenotipo(genotipo):
    ind = []
    for i in genotipo:
        ind = creator.Individual(list(dict.fromkeys(genotipo).keys()))

    fenotipo = []
    for i in range(len(ind)):
        fenotipo.append((i, ind[i]))

    with open("fenotipo.out", "w") as out:
        out.write(str(dv.tiempo)+"\t"+str(tiempoGenotipo(ind))+"\n")
        out.write("\t"+str(dv.punto_inicio[0])+"\n")
        for i in fenotipo:
            out.write(str(i[0])+"\t"+str(i[1])+"\n")
        out.write("\t"+str(dv.punto_fin[0])+"\n")

```

Para saber la solución válida, lo que hacemos es imprimir la penalización que hace al mejor de todos, si esa penalización es igual a 1 es que la solución es válida ya que no modifica para nada ese resultado.

## MultiObjetivo

En cuanto a la multiobjetivo ya que vamos a tener varios resultados que van a ser buenos, lo que vamos a hacer es imprimir el frente de Pareto, es decir, vamos a imprimir los individuos que tiene le frente de Pareto.

```
100 59.568490982733046
0
0 8
1 14
2 44
3 35
4 62
5 42
6 50
7 15
8 56
9 43
10 37
11 22
12 9
13 10
14 5
15 36
16 4
17 32
18 48
19 51
20 59
21 20
22 16
23 2
24 13
25 61
26 23
27 49
28 46
29 19
30 41
31 39
32 47
33 17
34 33
35 12
36 3
65
-----
100 54.93153909096641
0
0 29
1 10
2 9
3 24
4 15
5 3
6 55
7 45
```

Como podemos ver imprime todos los individuos por los que va a pasar, en todos los individuos del frente de Pareto, además de pones el tiempo disponible que tenemos y el tiempo que tarda nuestro algoritmo.

```

def fenotipo(pareto):
    ind = []
    for i in pareto:
        ind.append(creator.Individual(list(dict.fromkeys(i).keys())))

    fenotipo = []
    with open("fenotipo.out", "w") as out:
        for i in ind:
            out.write(str(dv.tiempo)+"\t"+str(tiempoGenotipo(i))+"\n")
            out.write("\t"+str(dv.punto_inicio[0])+"\n")
            for j in range(len(i)):
                #fenotipo.append((j,i[j]))
                out.write(str(j)+"\t"+str(i[j])+"\n")
            out.write("\t"+str(dv.punto_fin[0])+"\n")
            out.write("-----"+"\n\n\n")

```

Este es el código que tenemos para poner el fenotipo en el cual además del tiempo como ya hemos visto, imprimimos el primero de todos que siempre pasa por ahí, así como el último y además todos los nodos que va visitando entre medias.

Para saber si es válido cuando imprimimos la penalización del genotipo mejor, vemos dos datos, el primero es la penalización y el segundo es el valor que da el tiempo, si el primer valor es 1 entonces el individuo es válido.

# Experimentos

Para la realización de los experimentos hemos seguido las pautas que nos indicaron en ubuvirtual, las cuales son las siguientes:

- 1- Seleccionar una configuración base. Basta con que aporte una solución medianamente buena. Simplemente es para tener un punto de partida.
- 2- Seleccionar un aspecto a comprobar (Ej: Tipo de Selección, Número de Individuos, Probabilidad de Cruce, etc).
- 3- Elegir el rango disponible para probar. El resto de los parámetros quedan fijos según la configuración base elegida en (1):
  - a. Si es un número, se debe comprobar un rango de valores. 4 o 5 son suficientes (Ej: num individuos = [50, 100, 150, 200, 250]).
  - b. Si es una selección, simplemente se eligen entre los disponibles (Ej: Selección = [Torneo, Ruleta, Restos])
- 4- Seleccionar 2 o 3 configuraciones del problema / ficheros.
- 5- Repetir la ejecución del algoritmo una vez por cada opción en el punto (3) y cada fichero en (4).
- 6- Agrupar los datos de cada experimento en tablas o gráficos para presentarlos. Por cuestiones de resumen de información, incluso se puede comprobar solamente el valor óptimo encontrado en cada ejecución, en lugar de la gráfica completa de la búsqueda.

Cabe mencionar que hicimos 4 experimentos, y en cada experimento hicimos 9 casos, hemos ejecutado esos 9 casos para los 4 ficheros de cada experimento.

## Restricciones

Las pruebas realizadas son sobre los siguientes ficheros:

- Set\_64\_1\_70
- Set\_66\_1\_070
- Set\_66\_1\_100

Para cada uno de los ficheros hemos realizado los siguientes casos:

### Caso 1:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

### Caso 2:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	600
init_pop	toolbox.population(n=100)

### Caso 3:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=200)

#### Caso 4:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=500)

#### Caso 5:

alg_param['cxpb']	0.5
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

#### Caso 6:

alg_param['cxpb']	0.9
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

#### Caso 7:

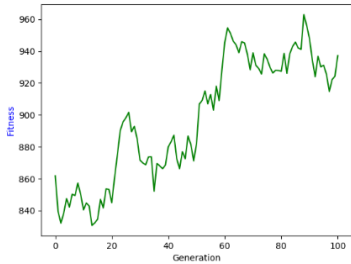
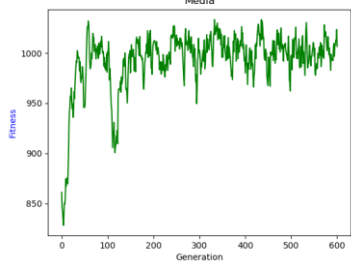
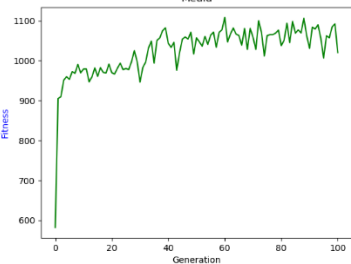
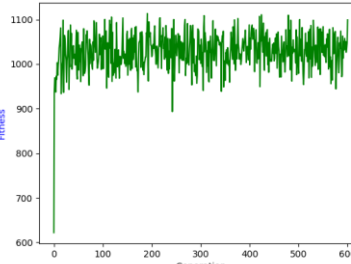
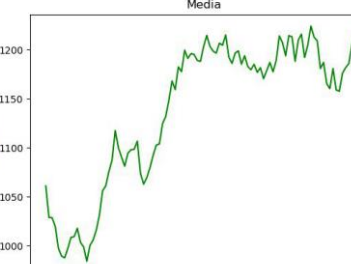
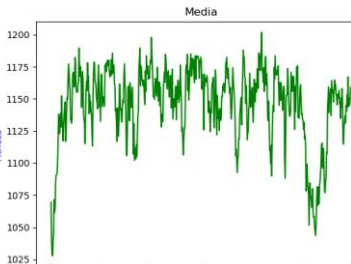
alg_param['cxpb']	0.75
alg_param['mutpb']	0.1
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)



Caso 8:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.5
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

## Cambiando el número de generaciones

	CASO 1	CASO 2
Set_64_1_70	<p><b>Fitness Mejor Genotipo(1205.0,)</b></p> 	<p><b>Fitness Mejor Genotipo(1044.0,)</b></p> 
Set_66_1_07 0	<p><b>Fitness Mejor Genotipo(1150.0,)</b></p> 	<p><b>Fitness Mejor Genotipo(1170.0,)</b></p> 
Set_66_1_10 0	<p><b>Fitness Mejor Genotipo(1240.0,)</b></p> 	<p><b>Fitness Mejor Genotipo(1185.0,)</b></p> 

Como podemos observar en la anterior tabla, el valor fitness depende de cada problema, ya que por ejemplo el primer fichero nos devuelve un mejor individuo que en el segundo fichero.

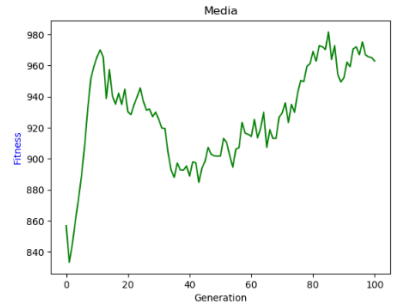
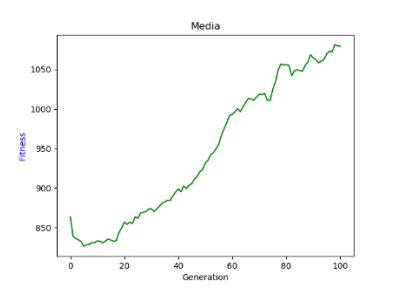
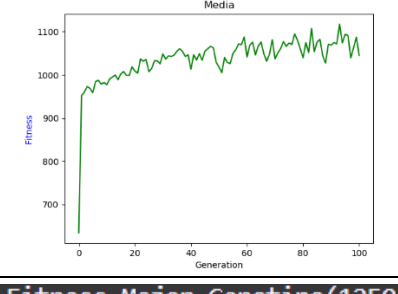
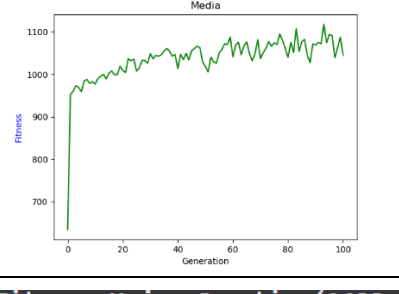
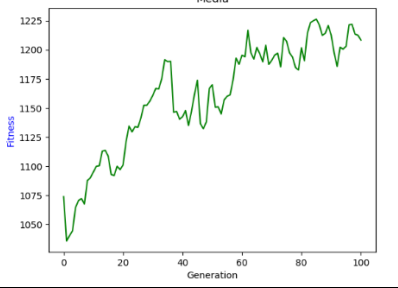
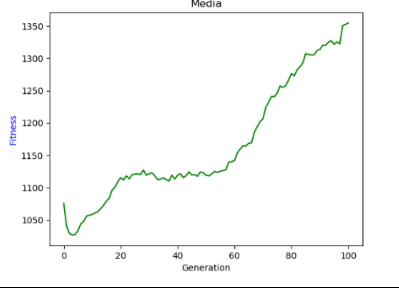
Por otro lado, podemos apreciar que, por norma general al aumentar el número de generaciones, aumenta el valor fitness, es decir, nos devuelve un mejor individuo, recalco que por norma general, ya que en el segundo fichero nos devuelve diferentes individuos con el mismo valor fitness.

Respecto al primer fichero podemos observar que al inicio crece de una forma exponencial, sin embargo, al llegar a la generación 40 crece y decrece todo el rato la función, esto se debe a que el algoritmo genético está probando todas las soluciones muy diversas, lo cual permite que luego cuando hemos aumentado el número de generaciones aumente el valor fitness.

Algo peculiar que podemos observar en el segundo fichero, es que en las primeras generaciones la función crece de forma muy exponencial (es casi una línea recta), sin embargo, este fichero es el que proporciona un menor valor fitness de los tres.

Respecto al tercer fichero podemos ver, que la función al principio crece de forma exponencial, y que todo el rato los individuos se mueven en un valor fitness de 1150 y 1300, es decir, que no se desvía como en el primer fichero que se mueve entre 1000 y 1300. Esto es algo bueno, ya que como podemos ver en la gráfica crecemos poco a poco, pero siempre va creciendo, sin embargo, esto no es del todo idóneo, ya que no genera individuos muy diversos entre unos y otros, lo cual a la larga provoca que aunque aumentemos el número de generaciones, tampoco haya mucha diferencia entre pocas y muchas generaciones.

Cambiando el número de población inicial

	CASO 3	CASO 4
Set_64_1_70	<div>Fitness Mejor Genotipo(1002.0,)</div> 	<div>Fitness Mejor Genotipo(1110.0,)</div> 
Set_66_1_07 0	<div>Fitness Mejor Genotipo(1220.0,)</div> 	<div>Fitness Mejor Genotipo(1320.0,)</div> 
Set_66_1_10 0	<div>Fitness Mejor Genotipo(1250.0,)</div> 	<div>Fitness Mejor Genotipo(1400.0,)</div> 

Como podemos observar en la anterior tabla, los fitness dependen de cada problema. Cuando aumentamos la población inicial, por norma general aumenta también el valor fitness, esto se debe a que hay más individuos con los que poder realizar el cruce y la mutación.

Respecto al primer fichero vemos que sí que ha aumentado el fitness, algo más que cuando aumentamos el número de generaciones. Sin embargo, parece ser que para este problema, no afecta mucho si tenemos una población inicial de 200 o de 500, ya que nos devuelve un valor fitness parecido.

Como bien hemos mencionado antes, el valor fitness depende de cada problema, y por ejemplo en el fichero dos sí que afecta el aumentar la población inicial, ya que obtenemos un mejor valor fitness. Al igual que sucedía en el primer caso, el rango por el que se mueve el valor fitness es bastante alto entre 1200 y 1400, aun así es menor que si solo aumentamos el número de generaciones.

Finalmente, respecto al fichero tres parece algo similar a lo que sucedía en el caso anterior, ya que al aumenta el número de individuos iniciales mejora, y al igual que sucedía en el primer caso vemos que se estabiliza muy rápido, es decir, desde la generación 20 el valor fitness se mueve entre 1600 y 1700.

En este tercer fichero vamos a ver como el fitness mejora bastante al aumentar la población inicial, es decir hay una diferencia más bien considerable entre los dos valores fitness, en el caso de menor población inicial vamos a ver cómo crece más rápido, pero el segundo vamos a ver cómo crece más lento pero llega a tener una mayor fitness.

Por último, cabe mencionar que lo del párrafo anterior es por norma general, ya que todos los algoritmos genéticos dependen del problema que estén intentando resolver.

Cambiando la probabilidad de cruce

	CASO 5	CASO 6
Set_64_1_70	<div>Fitness Mejor Genotipo(1014.0,)</div>	<div>Fitness Mejor Genotipo(1020.0,)</div>
Set_66_1_07 0	<div>Fitness Mejor Genotipo(1185.0,)</div>	<div>Fitness Mejor Genotipo(1215.0,)</div>
Set_66_1_10 0	<div>Fitness Mejor Genotipo(1235.0,)</div>	<div>Fitness Mejor Genotipo(1245.0,)</div>

En la anterior tabla mostramos las diferentes gráficas para cada uno de los ficheros anterior, en el caso 5 usamos una probabilidad baja de cruce, mientras que en el caso 6 usamos una probabilidad alta.

Cuando uno aumenta la probabilidad de cruce lo que provoca es que haya menos individuos elites en la siguiente generación, ya que se cruzan más. Esto tiene la ventaja de que va a generar mucha diversidad, pero no siempre generar una alta diversidad es bueno ya que esto provoca que el problema no converja de la forma correcta.

Respecto al primer fichero vemos que nos proporciona el mismo fitness, por lo que no tiene muchos problemas el algoritmo entre bajar y subir la probabilidad para este problema, sin embargo, podemos observar que en la segunda gráfica hay mucha más diversidad por lo comentado anteriormente.

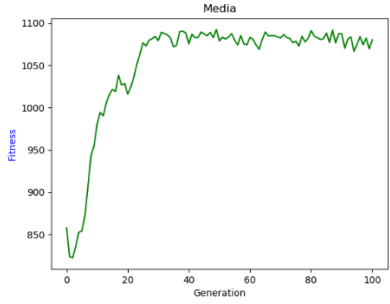
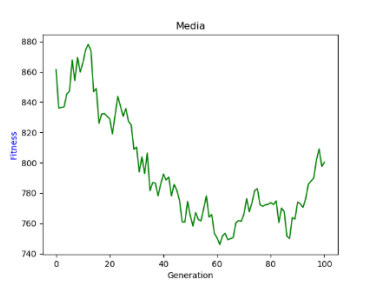
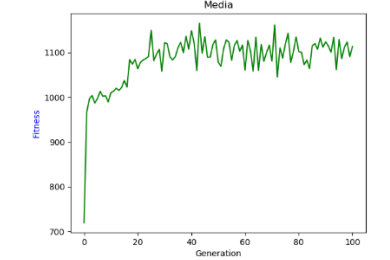
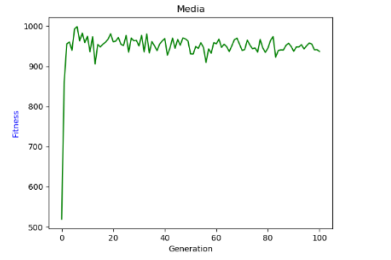
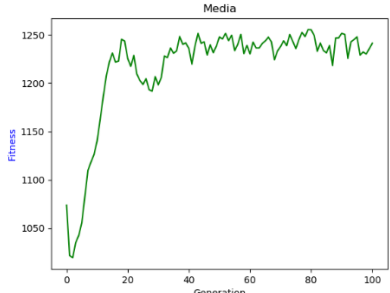
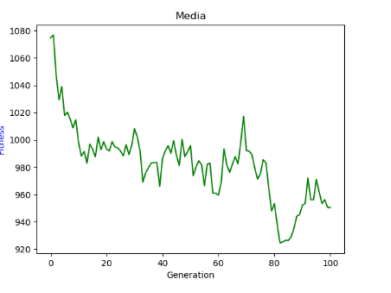
Respecto al segundo fichero se parece mucho al anterior, es decir los dos casos generan un fitness bastante parecida por lo cual no se nota demasiada la diferencia entre una probabilidad y otra.

Finalmente, en el tercer fichero también se parece mucho al anterior, es decir los dos casos generan un fitness bastante parecida por lo cual no se nota demasiada la diferencia entre una probabilidad y otra.

Aun así, podemos ver que poner una mejor probabilidad de cruce sí que nos mejora el fitness respecto a una probabilidad más baja, esto respecto a nuestras ejecuciones.

Cabe mencionar que no por poner una probabilidad muy baja en todos los problemas vamos a obtener una mejor solución, cada problema es diferente y hay que tratarlo como tal, por ello es ideal realizar diferentes configuraciones para cada fichero para saber cuáles son los mejores valores para cada uno de los parámetros de ese problema.

Cambiando la probabilidad de mutación

	CASO 7	CASO 8
Set_64_1_70	<div><div>Fitness Mejor Genotipo(1098.0,)</div></div>	<div><div>Fitness Mejor Genotipo(912.0,)</div></div>
Set_66_1_07 0	<div><div>Fitness Mejor Genotipo(1170.0,)</div></div>	<div><div>Fitness Mejor Genotipo(1105.0,)</div></div>
Set_66_1_10 0	<div><div>Fitness Mejor Genotipo(1260.0,)</div></div>	<div><div>Fitness Mejor Genotipo(1075.0,)</div></div>



Para realizar este apartado hemos modificado la probabilidad de mutación, en el caso 7 tenemos una baja probabilidad de mutación, mientras que en el caso 8 tenemos una alta probabilidad de mutación.

Cuando cambiamos la probabilidad de mutación a la alta, lo que generamos es que haya mayor diversidad, por norma general esto significa que el fitness que obtenemos es menor que si usamos una probabilidad "normal". Por otro lado, si cambiamos la probabilidad de mutación a la baja, lo que generamos es que haya menos diversidad, esto hace que haya menos cantidad de individuos.

Respecto al primer fichero, vemos que utilizar una probabilidad menor nos genera una mejor solución, esto se debe a que el problema resolviéndose de una forma corriente genera muy buenas soluciones, sin embargo, si mutan mucho esas generaciones nos genera individuos los cuales son peores o no tan buenos que los anteriores. Por otro lado, podemos ver que cuando usamos una baja probabilidad la gráfica está más controlada (no tiene tantos picos como sucede con una probabilidad muy alta).

Respecto al segundo fichero, vemos que el problema sigue sin cambiar, es decir, una menor probabilidad de mutación sigue provocando que de una mejor solución.

Finalmente, en el tercer fichero sigue pasando lo mismo que los dos anteriores.

En este caso podríamos decir que en nuestros experimentos, bajar la probabilidad de mutación resulta beneficioso para la solución final.

Para resumir, no porque bajemos o aumentemos la probabilidad de mutación vamos a obtener mejores individuos, esto al igual que sucedía con los casos anteriores depende del problema.

## MultiObjetivo

Las pruebas realizadas son sobre los siguientes ficheros:

- Set\_64\_1\_70
- Set\_66\_1\_070
- Set\_66\_1\_100

Para cada uno de los ficheros hemos realizado los siguientes casos:

### Caso 1:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

### Caso 2:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	600
init_pop	toolbox.population(n=100)

### Caso 3:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=200)

#### Caso 4:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=500)

#### Caso 5:

alg_param['cxpb']	0.5
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

#### Caso 6:

alg_param['cxpb']	0.9
alg_param['mutpb']	0.2
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

#### Caso 7:

alg_param['cxpb']	0.75
alg_param['mutpb']	0.1
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

Caso 8:

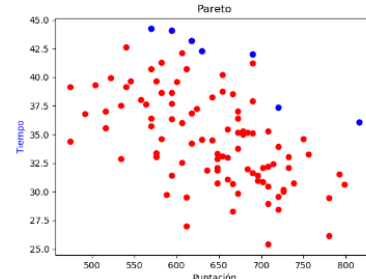
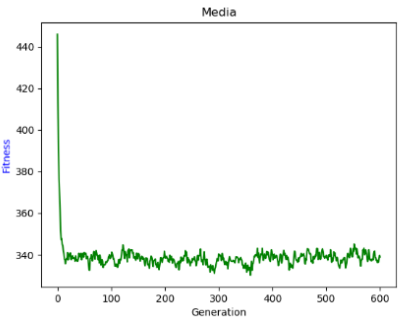
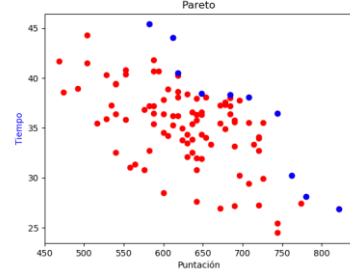
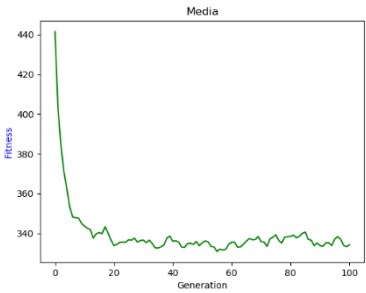
alg_param['cxpb']	0.75
alg_param['mutpb']	0.5
alg_param['pop_size']	25
alg_param['ngen']	100
init_pop	toolbox.population(n=100)

Cambiando el Número de Generaciones

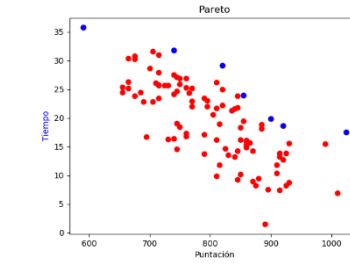
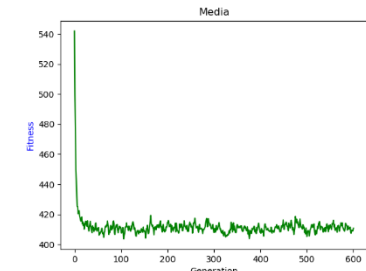
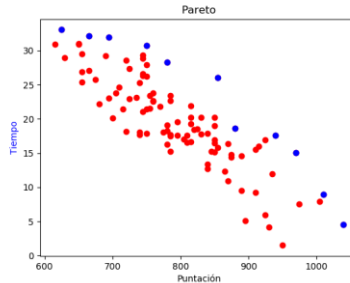
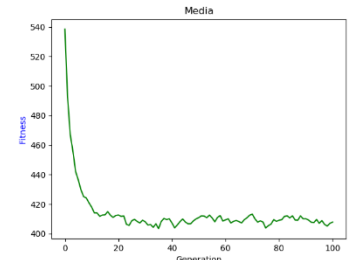
CASO 1

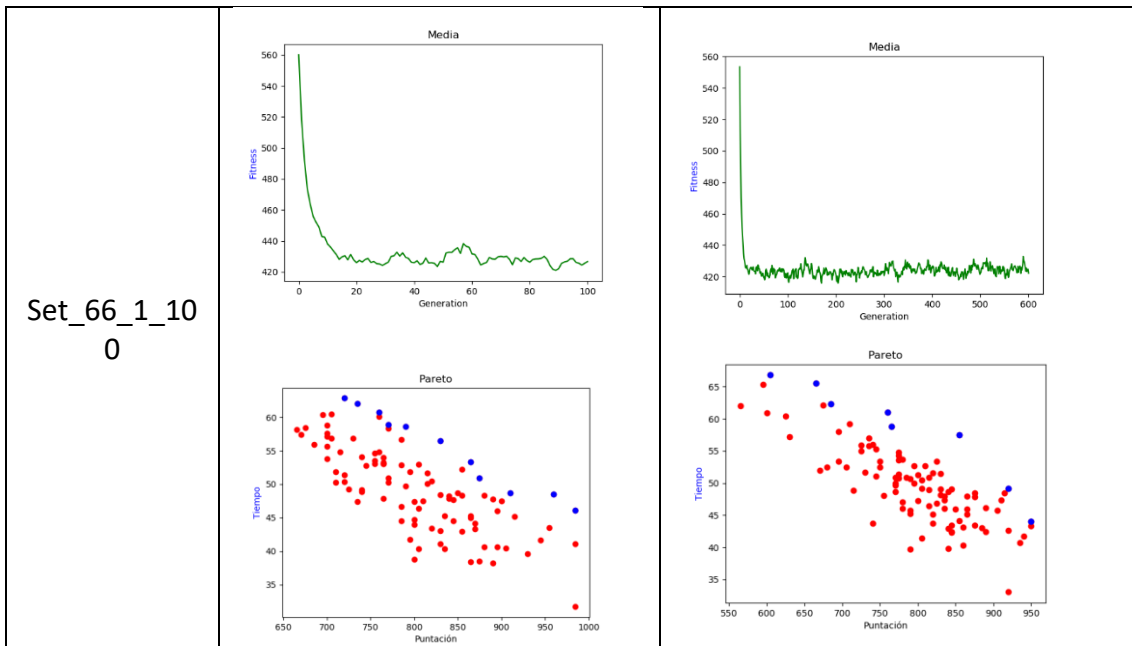
CASO 2

Set\_64\_1\_70



Set\_66\_1\_07  
0





En este primer caso de experimento lo que vamos a hacer es modificar el número de generaciones, como vamos a ver lo hemos dividido en tres ficheros.

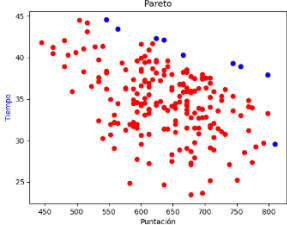
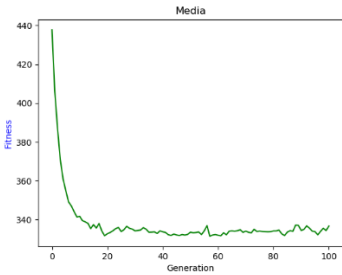
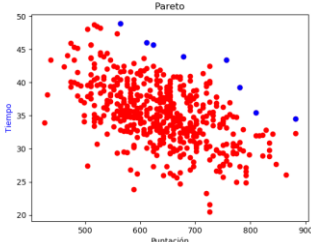
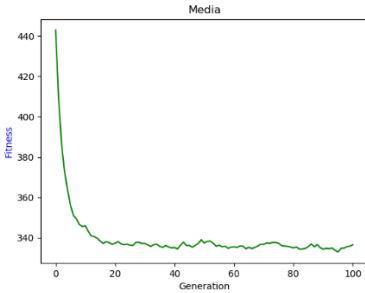
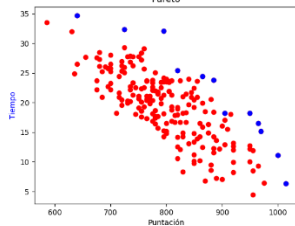
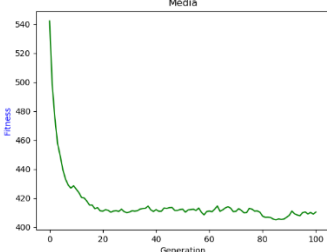
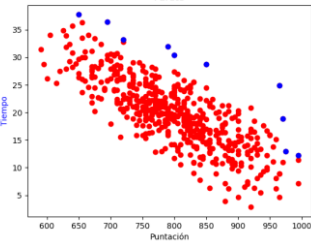
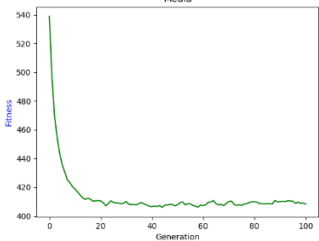
En el primer fichero podemos ver cómo, aunque aumentemos el número de generaciones, los resultados siguen siendo muy parecidos a simple vista, por tanto, podemos concluir en nuestro ejemplo que da igual en ese fichero si aumentamos o no.

En el segundo fichero podemos ver como el resultado da bastante parecido también por lo que las conclusiones son muy parecidas al anterior.

En el tercer fichero lo que podemos observar es más de lo mismo, resultados muy parecidos.

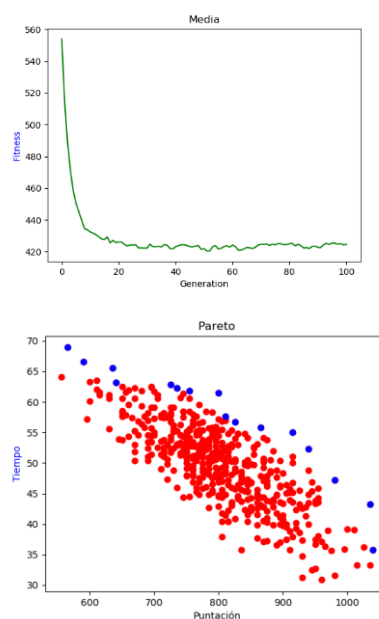
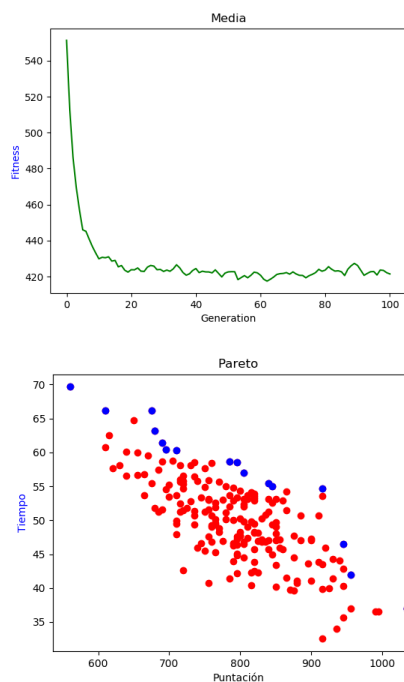
Podríamos concluir que en las pruebas que hemos hecho nosotros no afecta demasiado al resultado final el tener un mayor número de generaciones, lo que sí cambia algo es que en el frente de Pareto si aumentamos las generaciones, aparecen menos puntos.

Pensamos que las gráficas decrecen debido a que nosotros intentamos maximizar la puntuación y también maximizar el tiempo sobrante que tenemos, por tanto al intentar que sobre el mayor tiempo posible, puede ser que consiga puntuaciones peores pero que aun así sean óptimas.

Cambiando la Población Inicial		
	CASO 3	CASO 4
Set_64_1_70	<div></div>	<div></div>
Set_66_1_070	<div></div>	<div></div>



Set\_66\_1\_10  
0



En este caso vamos a probar a cambiar la población inicial lo cual en nuestro experimento va a hacer que cambie en cada uno de los ficheros, como podemos ver en el primer fichero vamos a ver como tarda más tiempo en realizar el camino pero conseguimos una puntuación más alta al aumentar la población inicial.

En el segundo fichero vamos a ver como aumentar la población hace que tarde más tiempo en realizar la ruta pero no consigue una mayor puntuación por lo cual deberíamos pensar que es peor opción.

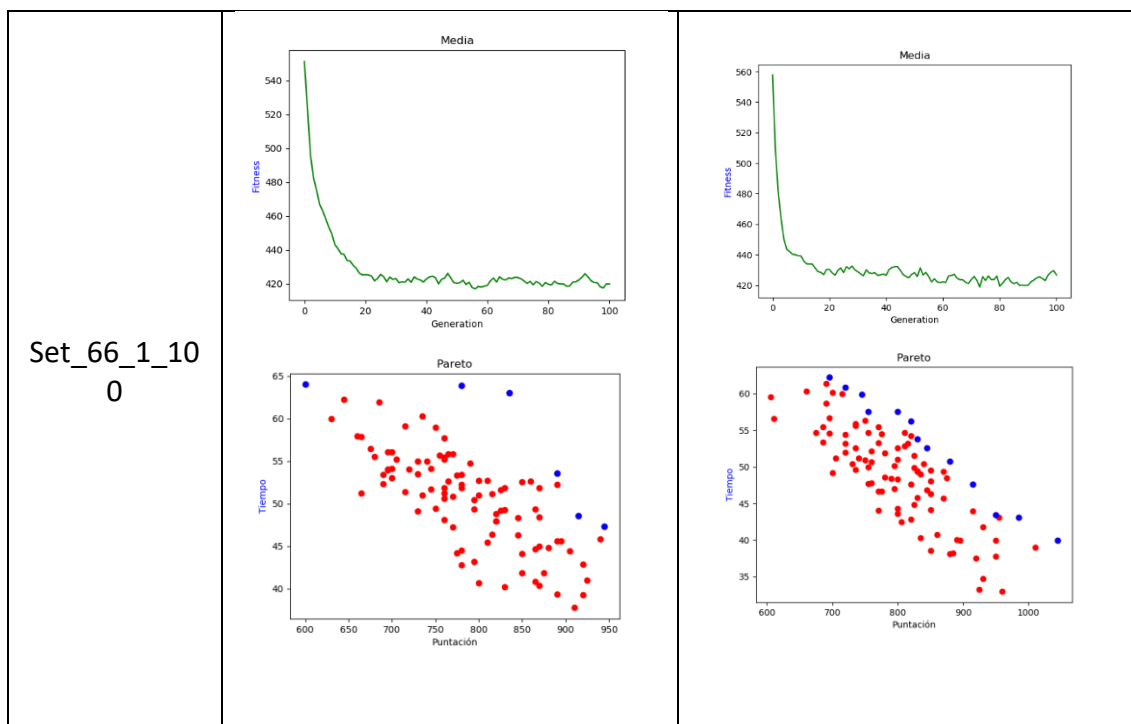
En el último de los ficheros vemos como aumentar o disminuir la población inicial no afecta en nuestro ejemplo de ningún modo apenas, ya que da individuos de una puntuación parecida con un tiempo bastante parecido.

Pensamos que las gráficas decrecen debido a que nosotros intentamos maximizar la puntuación y también maximizar el tiempo sobrante que tenemos, por tanto al intentar que sobre el mayor tiempo posible, puede ser que consiga puntuaciones peores pero que aun así sean óptimas.





Cambiando la Probabilidad de Cruce		
	CASO 5	CASO 6
Set_64_1_70		
Set_66_1_07 0		

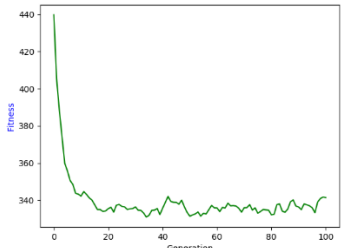
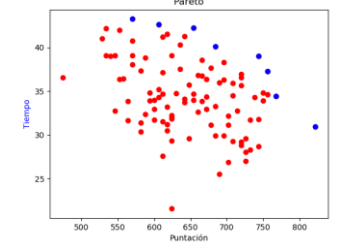
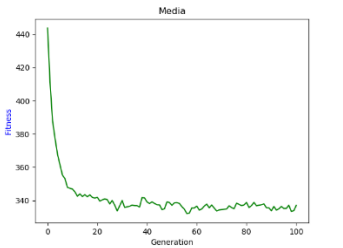
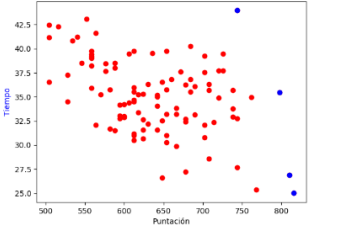
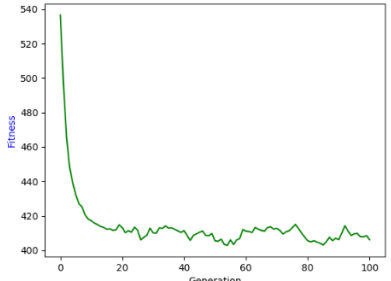
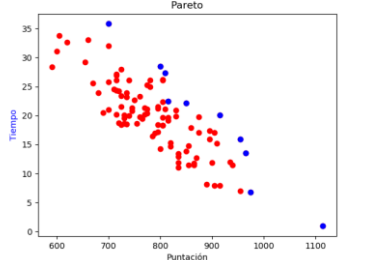
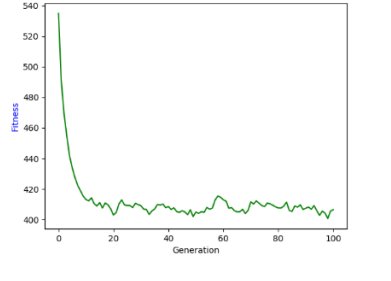
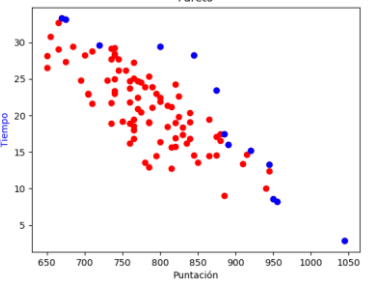


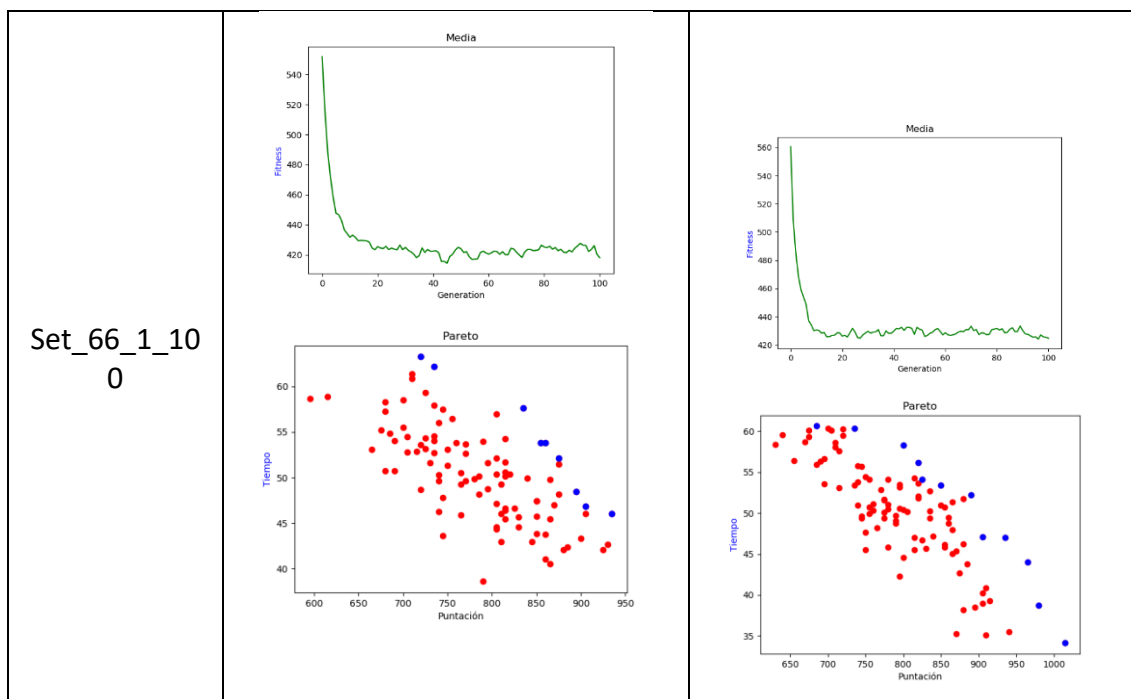
Así como hemos tenido mayor variedad con todos los anteriores casos, cambiar la probabilidad de cruce consigue algo bastante parecido en todos los ficheros, ya que en los casos en los que aumentamos la probabilidad de cruce vemos como tarda un poco menos de tiempo en recorrer todos los puntos.

A su vez, también podemos ver que tanto en el fichero uno, como en los ficheros dos y tres, la puntuación conseguida es bastante mejor que cuando tenemos menor probabilidad de cruce, esto puede deberse a que cuando cruzamos más individuos, generamos una mayor diversidad lo que hace que puedan surgir individuos mejores con mayor facilidad.

Pensamos que las gráficas decrecen debido a que nosotros intentamos maximizar la puntuación y también maximizar el tiempo sobrante que tenemos, por tanto al intentar que sobre el mayor tiempo posible, puede ser que consiga puntuaciones peores pero que aun así sean óptimas.



Cambiando la Probabilidad de Mutación		
	CASO 7	CASO 8
Set_64_1_70	<div><p>Media</p><p>Pareto</p></div>	<div><p>Media</p><p>Pareto</p></div>
Set_66_1_07 0	<div><p>Media</p><p>Pareto</p></div>	<div><p>Media</p><p>Pareto</p></div>



En este caso lo que vamos a hacer será aumentar la probabilidad de mutación, esto debería hacer que haya mayor diversidad lo cual puede ser positivo porque genera muchas nuevas soluciones que podrían ser buenas, pero en parte que haya demasiada es malo, ya que puede que no converjas en un punto.

Como podemos ver en todos los ficheros se nota una cierta mejoría cuando mutamos más, no quizá en tiempo donde solo disminuye ligeramente en algunos casos si no en la puntuación, que la mayoría de los individuos en los que se ve aumentada la probabilidad de cruce tienen por norma general mejor puntuación.

Pensamos que las gráficas decrecen debido a que nosotros intentamos maximizar la puntuación y también maximizar el tiempo sobrante que tenemos, por tanto al intentar que sobre el mayor tiempo posible, puede ser que consiga puntuaciones peores pero que aun así sean óptimas.