PRÁCTICA 1

ALGORITMOS GENÉTICOS Y EVOLUTIVOS

Luis Buceta Ojeda -- 100303573 | Adrián Rodríguez Grillo -- 100316457
GRUPO 83

Contenido

INTRODUCCIÓN

En esta práctica nuestro objetivo es la realización de un algoritmo genético que estudie el problema propuesto, que aborda el siguiente objetivo: la colocación de un conjunto de sensores medioambientales en un número de estaciones de forma que el coste de instalación sea el mínimo con el mínimo sacrificio en precisión. Habrá un total de 24 estaciones con una posible instalación de hasta 16 sensores por cada una.

PRIMERA PARTE

Codificación

En esta primera parte de la práctica tenemos la situación en la que el Ayuntamiento solo permite la instalación total o no instalación de los sensores medioambientales. Para ello, utilizamos una codificación binaria, que representa como '0' la no instalación de un sensor concreto y como 'F' la instalación del mismo.

Cada estación estará compuesta por 16 caracteres o genes que representan a cada uno de los tipos de sensores (definidos por las moléculas que son capaces de detectar). Por tanto, al disponer de 24 estaciones trabajaremos sobre cromosomas de 384 genes.

Resolución

Para la obtención del resultado óptimo utilizaremos un algoritmo genético, siguiendo los procedimientos para el correcto funcionamiento de este, explicados en clase. Estos son la creación de una población inicial, que será un conjunto de soluciones aleatorias; el estudio de esta población inicial, utilizando la función de evaluación aportada en el enunciado; el método de selección de individuos para la siguiente generación y funciones de cruce y mutación para evolucionar el conjunto de soluciones y poder así encontrar la solución óptima.

Inicialización

El primer paso del algoritmo genético es la creación de la población inicial, que en nuestro caso se compondrá de 100 individuos (cada uno con 384 genes) generados aleatoriamente según la codificación descrita. Esta población contiene a 100 posibles soluciones que, según avancen las iteraciones, irán adaptándose generación a generación a la función de evaluación (o *fitness*).

NOTA: Aunque el número de individuos de la población es de 100 para facilitar la explicación de la práctica, hemos desarrollado la posibilidad de pasar como argumento al programa la cantidad de población a generar, lo que nos permitirá probar el algoritmo con poblaciones de distintos tamaños.

Evaluación

Para evaluar a la población creada, disponemos de una función de *fitness* dada en el enunciado, por lo que no conocemos el criterio de evaluación que sigue para determinar la eficiencia de la solución. En cada iteración del algoritmo, en esta fase, obtendremos al mejor individuo de la población estudiada y será el estandarte en el que nos fijaremos para comprobar el rendimiento del algoritmo. Para obtener el mejor cromosoma de cada población, pasamos a la web de la función de *fitness* cada cromosoma, uno a uno, y con la función 'evaluator' vamos eligiendo al individuo con un *fitness* menor. Al final de la evaluación, obtendremos el mejor cromosoma junto con su *fitness*.

Selección

Para la selección de cromosomas para siguientes generaciones, debemos utilizar algunos de los métodos canónicos que hemos estudiado en clase (como la ruleta, los torneos, etc.). En concreto, hemos decidido utilizar los torneos, puesto que su funcionamiento otorga muchas posibilidades a los mejores cromosomas para pasar a la siguiente generación sin dejar de lado el posible sobreajuste que pueda provocar la elección directa de los mejores cromosomas para futuras generaciones. Nuestros torneos funcionan con cuatro participantes, elegidos aleatoriamente de la población generada, y con nuestra función de evaluación escogeremos al mejor de estos cuatro. Queremos mantener el mismo tamaño de población durante toda la ejecución del algoritmo (es decir, tendremos 100 individuos en la siguiente generación).

Ya que por temas del azar es posible que los mejores cromosomas no sean escogidos para participar en los torneos (aunque es una posibilidad bastante baja), hemos decidido añadir elitismo a nuestro proceso de selección. En concreto, introducimos a la siguiente generación directamente (sin pasar por torneos) a los dos mejores individuos de nuestra población. Con esto, nos aseguramos que, como mínimo, la población no empeore su rendimiento según pasen las iteraciones del algoritmo. Al introducir siempre a los dos mejores de la generación anterior, solo debemos realizar un número de torneos equivalente a la longitud de la población menos dos.

Al final de esta fase, tendremos a una población optimizada respecto a la generación anterior, pero la falta de un cambio real entre los cromosomas entre una generación y otro solo conseguirá que el algoritmo se estanque. Para la formación de una nueva generación real, debemos cruzar a los individuos de la población para crear a "hijos" que conserven la información de los "padres" y a la vez contemple nuevas configuraciones para evaluar. Además, añadiremos un elemento de mutación para, de nuevo, dar variedad genética en la población.

NOTA: al igual que la población, en la práctica nuestro algoritmo permite elegir al número de participantes para que compitan en los torneos, útil para probar distintas configuraciones del algoritmo. Sin embargo, mantenemos el número de cromosomas afectados por el elitismo (dos), para no sobre ajustar el resultado del algoritmo.

Cruce

La función de cruce es la encargada de generar la nueva población real que será utilizada en la siguiente iteración del algoritmo. El proceso de selección anterior escoge a los mejores padres posibles que, cruzándolos, generarán hijos que mantendrán la información ya evaluada de los padres. Además, este cruce generará individuos con configuraciones genéticas distintas a lo obtenido únicamente por los torneos, y es la clave para hacer progresar el funcionamiento del algoritmo. En caso de no cruzar a los cromosomas padre, el algoritmo solo podría determinar que el mejor *fitness* fuese el mejor *fitness* encontrado en la población inicial (que, generada aleatoriamente, y con 384 bits, hace altamente improbable que pueda ser una solución realmente óptima del problema).

Nuestra función de cruce se basa en el cruce uniforme estudiado en clase. Este es un tipo de cruce donde se escoge a dos padres (nosotros lo cogemos en orden, ya que la población ya fue seleccionada aleatoriamente), y gen a gen determinamos, aleatoriamente, de cuál de los dos padres obtiene el valor del gen el hijo que se está formando. Los hijos, sin embargo, se forman a pares, por lo que lo que no se lleva un hijo se lo lleva el otro (por ejemplo, si el hijo 1 obtiene el gen6 del padre 1, el hijo 2 obtendrá el gen 6 del padre 2). Así generaremos a toda la población de la siguiente iteración, que mantendrá a los dos cromosomas que cogimos por elitismo en la fase anterior (a estos no los cruzamos, los mantenemos igual).

Ya tenemos a la siguiente generación, pero antes de volver a iterar sobre ella, hemos decidido que pueda haber una pequeña mutación en algunos de los genes de los individuos de la población, que nos dará un poco más de diversidad en la población y nos ayuda a prevenir estancamientos.

Mutación

La mutación de la población se basa en el recorrido de toda la población, y por cada gen de cada cromosoma, calcular una pequeña probabilidad aleatoria que determinará si el gen debe cambiar o no. Esta probabilidad aleatoria es de un 0.01 (un 1%), y en el caso de que se dé, el gen cambiará su valor a otro dentro de la codificación permitida (en el caso de la práctica, o ‘0’ o ‘F’).

Tras terminar esta fase, tendremos una población totalmente distinta a la anterior, pero optimizando los cromosomas para que cada vez estén más cerca del *fitness* óptimo. Sin embargo, todo lo descrito es solo una iteración del algoritmo. En realidad, el algoritmo se ejecuta durante muchas iteraciones, que van usando las poblaciones generadas por la iteración anterior para ir optimizando las posibles soluciones. El paso final para completar el funcionamiento del algoritmo es establecer el bucle de ejecución que realiza cada iteración, y determinar cuándo debe parar el bucle.

NOTA: al igual que con la generación de la población inicial y los participantes de los torneos, hemos programado la posibilidad de ejecutar el algoritmo con una probabilidad de mutación elegida por el usuario. Nos servirá para la realización de pruebas con distintas configuraciones del algoritmo.

Función principal

Nuestra función principal de la función es la encargada de llevar el bucle de ejecución, que realiza las fases anteriores y se encarga también de llevar un registro de cada iteración, con el tiempo de ejecución que tarda en realizarse y el *fitness* obtenido por su mejor cromosoma. Únicamente en la primera iteración, generaremos la población inicial, mientras que en esta y en el resto de iteraciones realizaremos la evaluación, selección, cruce y mutación de la población. Para facilitar la creación de pruebas y la posibilidad de cambiar los parámetros del funcionamiento del algoritmo, nuestra función principal acepta como argumentos el tamaño de la población deseada, el número de iteraciones o generaciones a ejecutar, la tasa de mutación (1/valor establecido), la cantidad de participantes de los torneos y un flag para activar el modo de parada por estancamiento o no.

El algoritmo tiene dos criterios de parada, que son complementarios, y nos permiten acotar la ejecución del algoritmo a los parámetros que veamos necesarios. El primer criterio de parada es sencillo: damos al algoritmo un número máximo de generaciones a ejecutar. En cuanto comprobamos que hemos ejecutado tantas iteraciones como decidimos por parámetro, el algoritmo parará. Este criterio funciona junto con otro algo más complejo, que estudia el estancamiento de una población durante la ejecución del algoritmo.

Este segundo criterio de parada ha sido generado para evitar el estancamiento del algoritmo, teniendo en cuenta el ruido que existirá en la versión final. Para ello toma los resultados de la función de evaluación en un marco de ejecución de 40 iteraciones, tomando los 10 más actuales en un conjunto y los 30 más antiguos en otro, una vez formados los conjuntos, lo que se hará es tomar el mínimo valor de *fitness* de cada uno y se compararán. En el caso que el valor del conjunto antiguo sea menor o igual al conjunto nuevo durante 5 iteraciones el algoritmo interrumpirá su ejecución al considerar que se ha producido un estancamiento.

Finalmente, somos capaces de recoger los resultados de la ejecución total del programa en un archivo de texto, que contendrá el número de iteración realizada, el tiempo de ejecución y el mejor fitness de dicha iteración.

Aplicación a Python

Nuestra resolución anteriormente descrita ha sido implementada en el lenguaje de programación Python. El funcionamiento principal del programa está en el archivo ***main.py***, que contiene el bucle principal del algoritmo, y es el encargado de llamar a todas las funciones del programa. La función main, tal y como se dijo en el apartado anterior, toma como argumentos a varios parámetros que hemos decidido hacer variables para permitir configurar el programa con distintas variaciones de parámetros como el tamaño de población o la tasa de mutación (véase Función principal). El resto de funciones del programa están en distintos archivos, cada uno contenedor de distintas funcionalidades prácticamente equivalentes a las fases del algoritmo:

- ***Generator.py:*** encargado de crear una población aleatoria, con un tamaño determinado por parámetro, de individuos de 384 genes (con valores ‘0’ o ‘F’). Devuelve dicha población para futuro uso. Esta función solo se activa en la primera iteración del algoritmo, para crear la población inicial.

- ***Tester.py:*** contenedor de la función de evaluación del algoritmo. Recorre una población pasada por parámetro, y envía a cada uno de los cromosomas a la página web dada en el enunciado, y ejecuta la función de fitness para cada uno de estos. Nosotros leemos el contenido de la página, y devolvemos finalmente el cromosoma con mejor fitness (también devolvemos el fitness).
- ***Tournament.py:*** función ejecutora de los torneos. Recibe a una población y el número de participantes a estudiar por cada torneo creado, y mete en una nueva población al ganador de cada uno de los torneos. El número de torneos es equivalente a la longitud de la población menos dos. Estos dos cromosomas sueltos serán los elegidos por elitismo, que pasan directos a la nueva población. Devolvemos la nueva población seleccionada.
- ***CrossMutation.py:*** este archivo contiene dos funciones claves en el funcionamiento del algoritmo:
    - **Cruce:** recibimos una población y, mediante cruce uniformado, vamos escogiendo a un par de padres de dicha población, para generar a dos hijos que, aleatoriamente, recibirán los genes de un padre u otro. Esto lo hacemos para toda la población menos para los cromosomas elegidos por elitismo, que entran en la nueva población sin cruzarse.
    - **Mutación:** con una población como parámetro, recorremos cada uno de sus elementos gen a gen y, utilizando la tasa de mutación también recibida por parámetro, calculamos la probabilidad de cambiar cada gen a su alelo contrario (de '0' a 'F', o de 'F' a '0'). Finalmente, devolvemos a la población mutada.
- ***dataChecker.py****:* este archivo contiene 3 funciones que no se equivalen con ninguna fase del funcionamiento del algoritmo, sino que son funciones auxiliares creadas para ayudarnos en la realización de la práctica:
    - **saveData:** con esta función, creamos un archivo de texto que exporta toda la información necesaria para conocer el funcionamiento del algoritmo. En concreto, guarda todos los parámetros con los que se ha ejecutado el algoritmo, y el resultado de cada una de las iteraciones de este, para un futuro estudio más cómodo del algoritmo y su funcionamiento.
    - **paradaAlgoritmo:** tal y como hemos explicado en la resolución del problema, nuestro algoritmo tiene dos criterios de parada. Esta función se encarga de hacer funcionar el segundo criterio de parada (basado en el estancamiento de una iteración a otra). Esta función realiza la comprobación entre los últimos resultados de la función de evaluación, explicada anteriormente, y devuelve **True** para interrumpir la ejecución del programa.
    - **estadoEjecución:** esta función muestra en la terminal una barra de estado del funcionamiento del programa, con un porcentaje de progreso completado a su lado (clave para conocer el estado de ejecución del programa).
- ***Exec.py:*** archivo contenedor de todas las configuraciones del programa utilizadas para la prueba. Llama a main con distintos parámetros (por lo que ejecuta el algoritmo con distintos parámetros).

Actualmente, para ejecutar el programa, debemos introducir en *exec.py* la llamada a main con los parámetros que decidamos utilizar para la ejecución del algoritmo (ejemplificado con una llamada ya creada y una explicación comentada en el propio archivo).

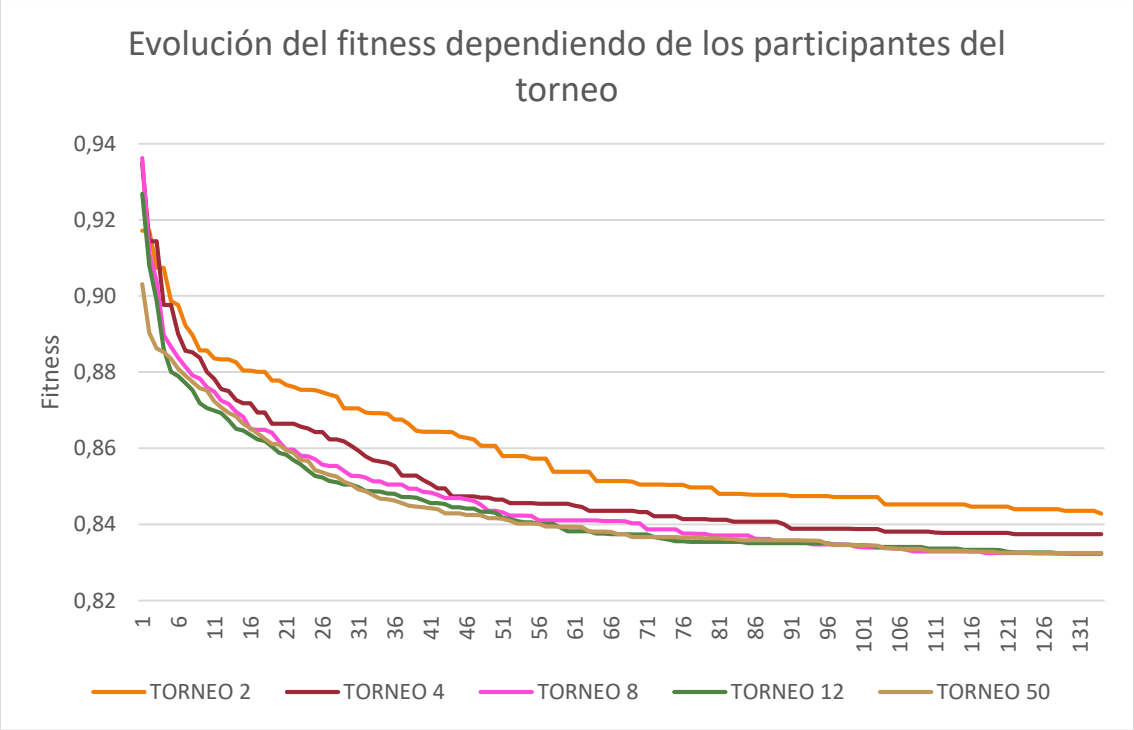
Por supuesto, el desarrollo de gran parte de este algoritmo se hizo con el modelo simplificado planteado en el enunciado (cadenas de 64 bits, '0' y '1', con una función de fitness alternativa), con los que conseguimos ajustar el algoritmo para llegar al cromosoma perfecto (con fitness = 0) en unas pocas iteraciones. Con unos pocos cambios en esa primera versión simplificada del algoritmo, hemos sido capaces de convertir la resolución teórica expuesta en el apartado Resolución a una solución práctica y funcional, cuyos resultados estudiaremos en el siguiente apartado.

Análisis

En este apartado, queremos estudiar los resultados obtenidos por el algoritmo con distintas configuraciones, y ver cuál es la que mejor resultado da en el menor tiempo posible. Para la realización de las pruebas, hemos rellenado *exec.py* con numerosas llamadas al algoritmo, con variaciones en cada uno de los parámetros variables que tenemos disponibles.

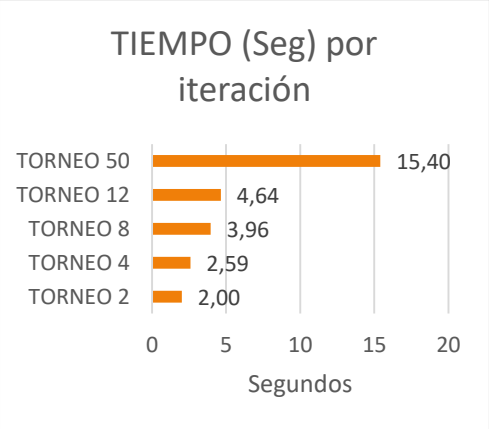
Participantes en los torneos

El primer parámetro de configuración que modificamos para ver su efecto en el sistema fue la cantidad de participantes en el torneo, este aspecto es muy importante para la selección de individuos, ya que si el conjunto es muy pequeño puede que los mejores individuos no sean seleccionados, al seleccionarse los participantes al azar, y, si es muy grande, se producirá un elitismo muy acusado, que será muy positivo al principio pero que posteriormente te puede llevar a un estancamiento en un mínimo local. Para ello probamos tamaños de torneos de 2, 4, 8, 12 y 50 participantes, obteniendo los siguientes resultados de fitness con respecto al tiempo:



Como podemos observar, y como era de esperar, con torneos de pocos participantes, especialmente el de 2, la mejora de los individuos es más lenta, obteniendo peores resultados en el mismo tiempo. Sin embargo, en el caso del torneo de 50 (mitad de la población) este se desempeña correctamente y junto con los torneos de 12 y 8 participantes obtienen los mismos resultados. Observado esta gráfica, las opciones estaban entre estas tres opciones de tamaño, por lo que para tomar una decisión calculamos el tiempo medio por iteración de cada una.

Como podemos observar en la tabla de tiempos de la derecha, cada iteración con torneos de 50 participantes necesita más de 15 segundos de media, convirtiendo a esta opción en poco eficiente debido a que se necesita el triple de tiempo necesario para llegar a resultados muy similares a los de 12 y 8 participantes.

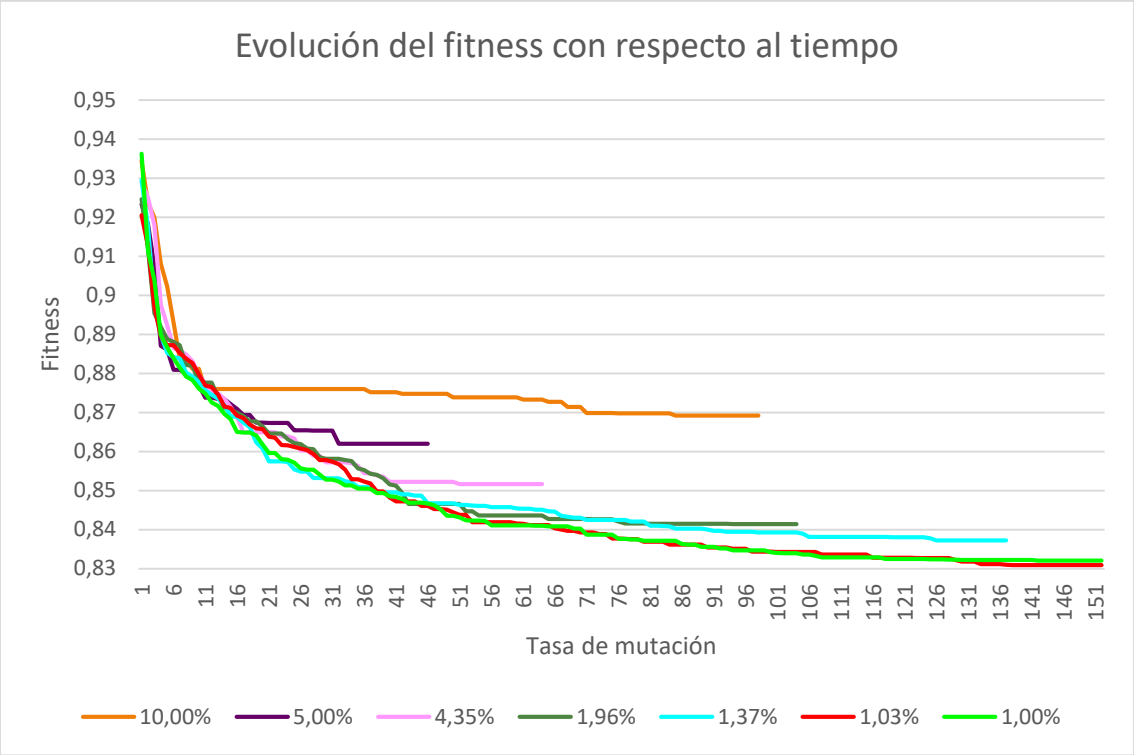


Por tanto, tras examinar estos datos, decidimos que 8 individuos sería la selección más adecuada de las utilizadas, ya que, aunque al principio fuese más lenta que la de 12 participantes, acaba con los mismos resultados y es cerca de un segundo más rápido por iteración.

Tasa de mutación

El segundo atributo que comprobamos fue la tasa de mutación, que evita la perdida de alelos en fases de la ejecución avanzadas, donde la población es menos variada debido a la reproducción de los mejores individuos durante sucesivos ciclos. Por tanto, esta tasa de mutación tendría que ser mayor a 0 para evitar esta pérdida, pero, por otra parte, tampoco debería ser muy alta ya que anularía todo el proceso de selección por torneos y reproducción al variar a los descendientes en gran medida.

Para la experimentación probamos a que se produzca una mutación por cada 10, 20, 23, 51, 73, 97 y 100 casos, poniendo énfasis en la utilización de números primos para este proceso. En la siguiente tabla se mostrará la evolución de las diferentes poblaciones dependiendo de las tasas de mutación, el máximo son 151 iteraciones, pero hay algunas que se quedan estancadas antes, por lo que su línea no llega al final.



Vista la tabla podemos obtener las siguientes conclusiones, cuanto más alta es la tasa de mutación antes se estanca la población. Esto es debido a que la búsqueda se hace de forma ciega, ya que los individuos varían muchos entre generaciones, perdiéndose los mejores en el proceso de mutación. En nuestro caso, al haber elitismo, siempre se quedan los dos mejores de la población anterior por lo que los valores se estancan en el mejor individuo que nunca es eliminado hasta que el sistema se considera estancado.

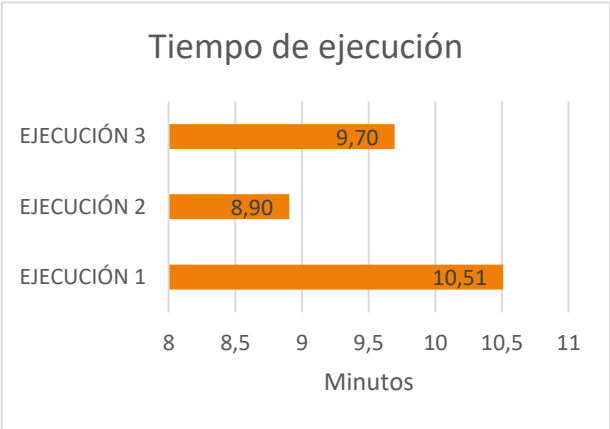
También observamos como las mejores poblaciones son las que tienen una tasa de mutación del 1.03% (1/97) y 1% (1/100), pudiendo observar mejores resultados en el primero, por lo que decidimos tomar este valor.

Conclusiones del análisis

Tras analizar el número de individuos en los torneos y las tasas de mutación del sistema, como hemos explicado anteriormente, nos quedamos con los siguientes atributos:

- Población de 100 individuos.
- Torneos de 8 participantes.
- Tasa de mutación del 1.03%, es decir, una mutación por cada 97 casos.
- Parada por estancamiento activada.

En la siguiente tabla, se realiza una comparación de tres ejecuciones con estos parámetros:



Como podemos observar el comportamiento es muy similar entre las tres ejecuciones, dependiendo su diferente desempeño del azar, primero con la creación de la población inicial, donde observamos que la tercera ejecución parte con ventaja y, posteriormente, con la selección en los torneos y la mutación. Con respecto a los tiempos de ejecución son también muy similares, siendo más cortos lo que se han estancado antes. Por tanto, observando estas tablas lo que podemos concluir es que este algoritmo llega a estancarse cuando es levemente superior a un fitness de **0.83** y suele de media suele tardar en llegar a este valor unas 140 generaciones, momento en el que empieza a estancarse y no mejora sus resultados.

En este apartado el mejor resultado ha dado un *fitness* de **0,831720815**.

SEGUNDA PARTE (AMPLIACIÓN)

La segunda parte de la práctica (una ampliación sobre la primera parte) nos plantea un problema algo más complejo que el original. En el enunciado se nos comenta que la instalación de sensores medioambientales se ha complicado, y ahora tenemos la posibilidad de instalar sensores que se activen de manera intermitente para ahorrar energía, a costa de no estar midiendo el nivel del aire activamente (y por ello, quizá perdiendo información en el proceso).

Nuestro trabajo es estudiar si es posible lograr una solución mejor con este sistema o con el anterior. Para ello, debemos hacer ciertas modificaciones a nuestro algoritmo anteriormente creado para aceptar las nuevas condiciones.

Codificación

Tal y como se especifica en el enunciado, esta parte de la práctica es una ampliación, por lo que usaremos como base lo creado en la primera parte. La codificación será muy similar a la de la primera parte, utilizando cromosomas de 384 genes, donde cada gen representa a cada una de las posibles posiciones de los sensores de 24 estaciones medioambientales (16 sensores * 24 estaciones).

La gran diferencia reside en los posibles alelos de los genes. Mientras que en la primera parte solo podíamos tener ‘0’ (sin colocar) o ‘F’ (colocado completamente), ahora a estos les añadimos el alelo ‘H’ (colocado intermitentemente). Por lo tanto, ahora tenemos una codificación ternaria en lugar de la binaria original. Esto provoca que ciertos elementos de la resolución del problema puedan cambiar.

Resolución

Gran parte de la resolución de la primera parte es totalmente aplicable a esta ampliación, con ciertos detalles que especificaremos a continuación.

Inicialización

La inicialización, gracias al cambio en la codificación del problema, ha sufrido cambios en su planteamiento. Ahora generará una población aleatoria, de tamaño determinado por parámetro, pero con genes que pueden tomar un valor de ‘0’, ‘F’, y ‘H’, en lugar de solo los dos primeros como en la primera parte.

Evaluación

La fase de evaluación se mantendrá exactamente igual a la primera parte, ya que no hay un cambio significativo en el funcionamiento de esta (y, además, usamos la misma URL para obtener el fitness que en la anterior parte).

Selección

De nuevo, el proceso de selección por torneos se mantendrá exactamente igual que en la primera parte, ya que principalmente hace uso de la intacta función de evaluación. De nuevo, mantenemos a un par de cromosomas para realizar el elitismo y no perjudicar el rendimiento del algoritmo de iteración a iteración.

Cruce

La función de cruce, al igual que la evaluación y la selección, seguirá intacta, de nuevo utilizando cruce uniforme para la creación de los descendientes de la siguiente generación.

Mutación

La mutación ha sufrido cambios, de nuevo por el cambio en la codificación del problema. Al tener una codificación ternaria, la mutación debe determinar no solo que gen va a mutar, si no a cuál de los otros dos alelos cambiará. Esta elección será totalmente aleatoria, utilizando el mismo proceso de elección de gen a mutar que en la primera parte.

Función principal

La función principal del programa funcionará igual que en la primera parte, dedicándose simplemente a lanzar las fases anteriores en orden y de manera iterativa.

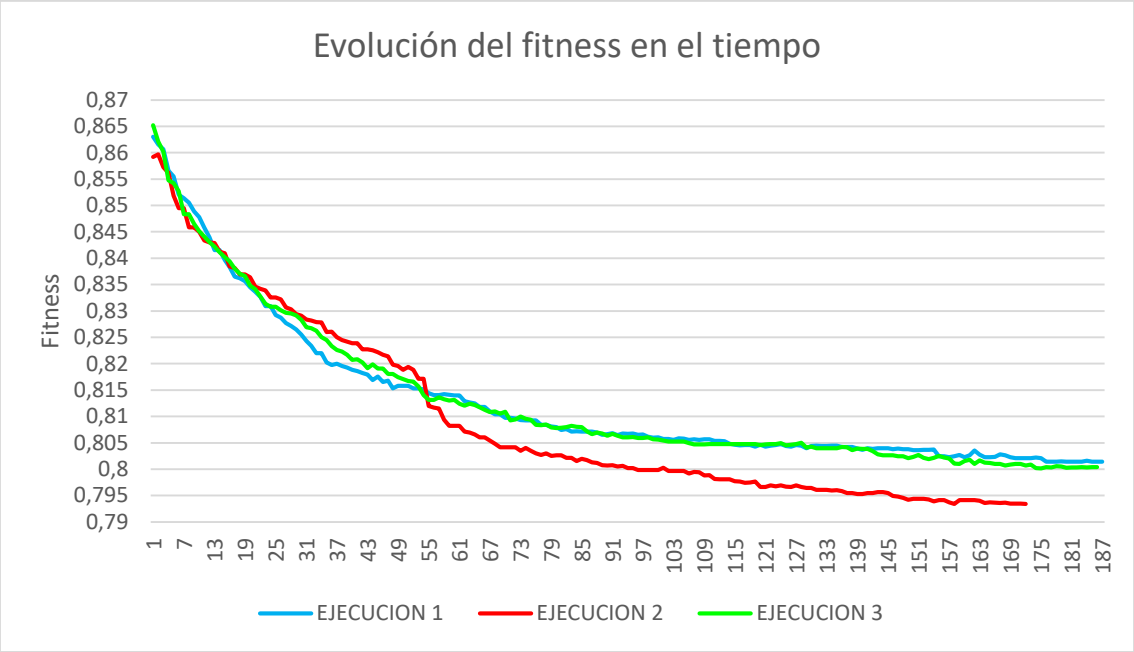
La traducción de esta resolución a Python es prácticamente idéntica que en la primera parte (véase Aplicación a Python). La mayor diferencia entre ambas partes es, de nuevo, el cambio de codificación. La función de generador de poblaciones (***generator.py***) utilizará los tres posibles valores de cada gen para formar su población inicial, y la mutación también elegirá aleatoriamente entre los dos alelos inversos al elegido para mutar en ***crossMutation.py.mutacion***. El resto de los archivos son idénticos a la primera parte, al igual que el funcionamiento del algoritmo.

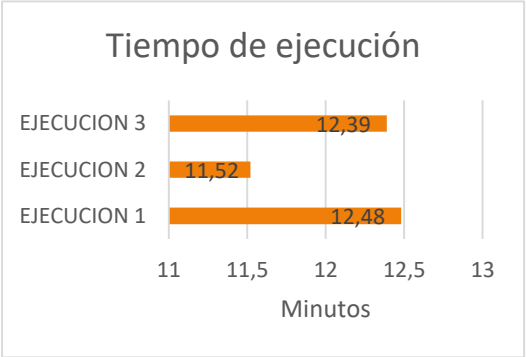
Parametrización

Este problema es muy similar al anterior con la diferencia de que se añade una nueva letra al diccionario de los individuos de la población, por lo que tras pocas pruebas concluimos que los parámetros a utilizar serían los mismos que en el caso anterior, es decir, la configuración sería la siguiente:

- Población de 100 individuos.
- Torneos de 8 participantes.
- Tasa de mutación del 1.03%, es decir, una mutación por cada 97 casos.
- Parada por estancamiento activada.

Con estos parámetros vamos a comparar diferentes ejecuciones para analizar los resultados.



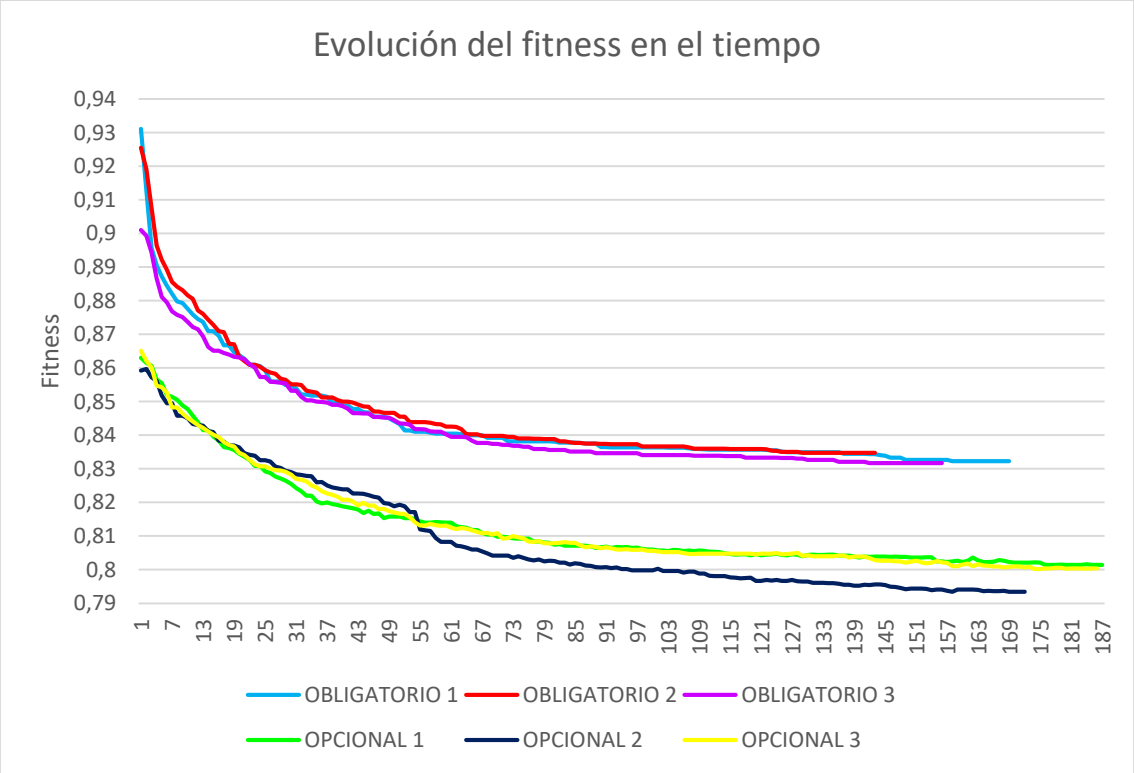


Como podemos observar el comportamiento es muy similar entre las tres ejecuciones, como en el caso del ejercicio obligatorio. Aunque en esta tabla podemos observar que la segunda ejecución consigue un mejor resultado. Con respecto a los tiempos de ejecución son también muy similares, siendo más corta la segunda ejecución que se ha estancado antes. Por tanto, observando estas tablas lo que podemos concluir es que este algoritmo llega a estancarse cuando su fitness llega a un valor de **0.80** y suele de media suele tardar en llegar a este valor unas 155 generaciones, momento en el que empieza a estancarse y no mejora sus resultados.

En este apartado el mejor resultado ha dado un *fitness* de **0,79343591**.

Comparación

El elemento más interesante de esta parte es la comparación de ambas codificaciones, es decir, ver si de verdad merece la pena introducir este nuevo tipo de sensor en comparación con el problema original. Para la comparación entre ambos problemas, utilizaremos la configuración del algoritmo que mejor resultado nos ha dado en el análisis de la primera parte, ya que consideramos justo que ambos se comparen en los mismos términos. Para ello vamos a comparar tres ejecuciones de cada parte y las compararemos.



Como podemos observar claramente la inclusión de los sensores de bajo consumo hace que los individuos creados por el segundo apartado sean mejores en todo momento. Por tanto, podemos afirmar que el segundo apartado genera mejores soluciones, aunque con un coste temporal mayor, es decir, el tiempo medio de ejecución por iteración es ligeramente mayor. Esto es debido a que, incluyendo una nueva letra en el diccionario, se aumentan el número de comprobaciones para la actuación del programa, como en el caso de la mutación, además también es mayor debido a que la parte opcional tarda más generaciones en estancarse y salir del programa.

CONCLUSIONES Y COMENTARIOS

En conclusión, creemos que hemos aprendido a realizar un algoritmo genético capaz de generar soluciones más que adecuadas al problema, y con muchos mecanismos para fortalecer su rendimiento (elitismo que evita conseguir fitness mayores o criterios de parada para distintas situaciones, por ejemplo, que dan resultados muy positivos de iteración a iteración).

Sin embargo, algo que debemos reseñar es lo sucedido en la evaluación en clase del algoritmo. Mientras que las pruebas del algoritmo en casa (utilizando una conexión SSH a Guernika para ejecutar el algoritmo como si estuviésemos en el laboratorio) dan unos tiempos bastante razonables para cada iteración del problema, el funcionamiento del algoritmo con una carga del servidor de fitness grande (como la que nos encontramos en la evaluación en clase) se ralentiza mucho, de 5 a 10 veces más de lo probado durante la resolución de la práctica. La razón de esto es que nuestro algoritmo evalúa a cada "minipoblación" de participantes de los torneos llamando a la web de fitness, en vez de utilizar algún tipo de caché que guarde los fitness de toda la generación en la fase de evaluación. Por este motivo, al utilizar el algoritmo con una carga grande en el servidor de fitness, nuestros tiempos de iteración en iteración pasaron de unos 9 segundos a más de un minuto con el fitness alfa. Con estos tiempos, una correcta ejecución del algoritmo resultó ser imposible de realizar (en especial, con funciones de fitness más complejas como *Madriz*, que llegaba a tiempos por generación cercanos a los 5 minutos). Ahora entendemos que la solución habría sido utilizar la caché de fitness como decíamos anteriormente, pero es algo que, en nuestras casas, y debido al rápido desarrollo de cada iteración, no contemplamos como algo necesario.

ANEXO: EJECUCIÓN DEL PROGRAMA

El algoritmo está programado en Python 2.7, necesario para la ejecución del programa. Además, hace uso de librerías que, en principio, están integradas en Python (incluido en la versión de Python instalada en los laboratorios de la universidad).

Para ejecutar el algoritmo, solo hay que introducir en la consola (dentro de la carpeta contenedora del código fuente de la práctica):

python exec.py

En caso de querer cambiar los parámetros con los que queramos ejecutar el algoritmo, es necesario abrir el archivo **exec.py** y cambiar la siguiente línea:

main(tamanoPoblacion=100, numGeneraciones=300, tasaMutacion=97, partipantesTorneo=8, parada=True)

Podemos cambiar cualquiera de los parámetros que llaman al main por el valor deseado. Aunque creemos que el nombre de los parámetros es auto descriptivo, el archivo contiene un comentario que explica a cada uno.