A1: Problema del Viajante

Computación Evolutiva

Cosmin Madalin Marina

15 de diciembre de 2022

1. Introducción

El Problema del Viajante (TSP por sus siglas en inglés), así como su generalización en el Problema de Enrutamiento de Vehículos (VRP) y sus variantes (VRPPD) son problemas típicos de optimización en la literatura.

Pero su alcance no se queda únicamente en la literatura, sino que en muchos casos tienen aplicabilidad directa a problemas reales, como puede ser la creación de rutas óptimas para repartos (p.e. Correos y empresas de transporte), suministros de materias primas, rutas turísticas, etc. Se trata de un problema fácil de entender, didáctico, pero complejo.

Vamos a abordar la versión más básica del TSP siguiendo esta estructura: en la sección 2 daremos una definición concreta del Problema del Viajante, en la sección 3 presentaremos la implementación práctica que se ha llevado a cabo para resolver el problema, en la sección 4 discutiremos los resultados obtenido de los experimentos llevados a cabo con nuestra implementación y, finalmente, en la sección 5 sintetizaremos las conclusiones de nuestro análisis.

2. Descripción del Problema del Viajante

El Problema del Viajante (o del Mensajero) tiene por objetivo encontrar la ruta más corta entre n puntos o *ciudades*, habiendo atravesado cada ciudad una única vez. Hay algunas particularidades que se deben mencionar y concretar en el problema. Por un lado, la simetría del mismo, esto es, si consideramos un TSP simétrico o asimétrico. En el caso simétrico, la distancia para ir de la ciudad $\mathcal{A}$ a la ciudad $\mathcal{B}$ es la misma que para ir de la ciudad $\mathcal{B}$ a la ciudad $\mathcal{A}$, mientras que en el caso asimétrico estas distancias no tienen por qué ser iguales.

El caso asimétrico es el más fiel a la realidad, ya que los caminos no son nunca totalmente iguales (véase calles de único sentido, accidentes, accidentes naturales o carreteras desiguales). Esto cobra mucho sentido en escalas pequeñas, pero en escalas grandes (p.e. distancias de km entre ciudades lejanas) que exista una pequeña diferencia entre el sentido de los caminos (p.e. del orden de los cm o m) puede considerarse indiferente.

El caso simétrico permite tratar el problema como un grafo no dirigido, donde cada nodo representa un punto o ciudad, y cada enlace representa un camino tomado. Por otro lado, debemos hablar de si el problema es cíclico o no. A lo que nos referimos con esto es si $C_0 = C_n$ o $C_0 \neq C_n$, donde C_0 es la ciudad inicial en la ruta y C_n es la ciudad final o, dicho de otra forma, si queremos acabar donde hemos empezado. En el primer caso, debemos encontrar un grafo cíclico, donde cada nodo $\{n_1, n_2, ..., n_k\} \in \mathcal{N}$, para $\mathcal{N}$ conjunto de todos los nodos, debe tener exactamente 2 y sólo 2 enlaces. En el segundo caso, deben existir 2 nodos n_i, n_j con $i \neq j$ con 1 único enlace, mientras que todos los demás $\{n_1, n_2, ..., n_{k-2}\}$ deben tener 2.

En nuestro caso, nuestra instancia del TSP será simétrica y cíclica. Cada grafo cíclico define el estado en el que nos encontramos.

Otra cuestión de relevancia es la métrica que se emplea para medir o calcular esas distancias. Existe una gran variedad de formas para resolver el problema TSP, desde lo más obvio como la fuerza bruta, hasta técnicas como la programación dinámica o las resoluciones estrictamente matemáticas. Un conjunto de técnicas que han sido muy empleadas en la resolución del problema y sus variantes (VRP, VRPPD) son son los algoritmos heurísticos y metaheurísticos. En este conjunto de técnicas se haya la computación evolutiva. En este tipo de técnicas es importante qué *heuristica* o *métrica* se emplea para la función objetivo. Esta puede ser cualquier tipo de norma y, en función de las características del problema, puede tener o no sentido. Si nos hayamos en un espacio no euclidiano, por ejemplo, las *p-normas de la métrica de Minkowski* no tienen sentido. Sin embargo, si nuestro problema se encuentra a escalas pequeñas, dentro de un espacio euclidiano como una ciudad, y queremos encontrar la ruta más rápida para ir en coche desde un lugar a otro de la misma, la norma $p = 1$ (distancia orto-normal o de Manhattan) es la más adecuada. En cambio, en una escala mayor entre ciudades donde queremos determinar la mejor ruta de reparto de un camión, una norma como la $p = 2$ (distancia euclidiana u ordinaria) tiene más sentido. Para la mayoría de estas métricas, al menos para las *p-normas de Minkowski*, es necesario que en nuestro problema se cumpla la desigualdad triangular ($\|x+y\| \leq \|x\|+\|y\|$).

Por último, queremos recordar que el Problema del Viajante es un Problema NP-complejo o NP-hard. Al igual que muchos otros, es un problema sencillo de definir, pero complicado de resolver. Esto es porque, en principio, son problemas irresolubles en tiempo polinomial. Este tipo de problemas son la base y dan sentido a las aproximaciones heurísticas al problema, que no siempre proporcionan la solución óptima, pero sí una buena aproximación de la misma. Por lo tanto, emplear un algoritmo genético (como el que se explicará en la siguiente sección 3) es justificable, siempre y cuando nadie demuestre $P = NP$ [1].

3. Implementación

En esta sección explicaremos la implementación que se ha llevado a cabo de un algoritmo genético para la resolución del Problema del Viajante definido en la sección 2. Se detallarán las decisiones tomadas y el funcionamiento del código. En este documento no se incluye el código íntegramente, sino solo algunos apartados que se hayan considerado importantes o relevantes para la explicación (si es que se ha dado el caso).

En cambio, **todo el código está disponible en el siguiente repositorio** [2]. En el mismo se puede observar el estado actual del código, la planificación que se ha seguido para su implementación (*Projects* + *Issues*), así como los distintos estados intermedios por los que ha pasado el mismo. También se puede observar de forma clara cómo la idea de la implementación ha ido cambiado, los errores y dificultades que se han ido encontrando, etc.

Lo primero a aclarar es la representación de los individuos y las estructuras que se usan para la misma. En un principio se valoró situar las ciudades en un círculo de radio dado y, que cada ciudad estuviera definida por sus coordenadas polares. Esto, sin duda, tiene numerosas ventajas en la representación y el cálculo de distancias, pero *en aras de la claridad*, finalmente nos decantamos por las habituales coordenadas cartesianas. De esta forma, cada ciudad está representada por un par (x_c, y_c) de coordenadas elegidas aleatoriamente según una distribución uniforme. Dado que en la realidad nos encontramos con pares de ciudades como Alcobendas-San Sebastián o San Fernando-Coslada, donde una calle sirve de separación entre ambos municipio o la propia frontera es difusa, y con el objetivo de ser lo más realistas posibles, no se ha incluido

[1] Evento potencialmente catastrófico para la computación evolutiva y otras ramas dedicadas a optimización.
[2] `https://github.com/cosminmarina/A1_ComputacionEvolutiva`

una tolerancia o límite entre las coordenadas de las ciudades. Si fuera el caso, una ciudad no podría encontrarse a una distancia menor que t de otra en un radio alrededor de la misma.

Partiendo de esto, cada par de coordenadas recibirá una etiqueta o identificador. Hacemos uso de un archivo de tipo *CSV*, llamado *ciudades.csv*, donde se encuentran las ciudades, como pares de coordenadas, ordenadas de forma arbitraria. Este orden arbitrario que siguen en el archivo define de forma única el identificador de la ciudad. Esto es, la ciudad que ocupa la primera posición tendrá el identificador 0 (recordemos que empezamos a contar en 0), la siguiente ciudad el identificador 1, y así sucesivamente hasta $n-1$ para n número de ciudades. Nuestro genotipo (la forma en la que representamos los individuos) será una lista que contendrá el orden de las ciudades siguiendo el identificador de la misma. Para 7 ciudades, el genotipo $\{0, 6, 2, 1, 3, 4, 5\}$ indica que nuestro camino comenzará (y finalizará) en la ciudad 0, atravesando en orden las ciudades 6, 2, 1, 3, 4 y 5. Este tipo de representación se denomina por permutaciones, ya que cada genotipo representa una serie de permutaciones o intercambios con respecto a la lista inicial de ciudades (el orden arbitrario definido en el archivo) que se deben de seguir.

Antes de desgranar cada etapa de nuestro algoritmo, vamos a definir los parámetros del *params.json*:

- *nciudades*: número entero que indica cuantas ciudades tendrá el problema.
- *radio*: número entero que expresa el radio de la circunferencia circunscrita al cuadrado en el que se pueden posicionar las ciudades. O, lo que es lo mismo, la mitad del lado de la cuadrícula en la que se inscriben las ciudades.
- *tam_poblacion*: número entero que indica el tamaño de la población de individuos del algoritmo.
- *niter*: número entero de repeticiones del algoritmo que se llevarán a cabo con los mismos parámetros.
- *ngen*: número entero de generaciones de un algoritmo.
- *pasos$_i$ntervalos*: número entero que especifica cada cuantas iteraciones se muestra el intervalo de confianza.
- *npadres*: número entero que indica de cuántos padres estará compuesto cada torneo.
- *pcruce*: número real que representa la probabilidad de cruce.
- *pmutacion*: número real que representa la probabilidad de mutación.
- *elimitacioneselitismo*: cadena de caracteres que indica el método de sustitución en el elitismo de la selección de hijos. Por ahora solo está implementado *peor* (se escogen los peores hijos para ser sustituidos por los padres elitistas) y *aleatorio* (se escogen hijos aleatorios para ser sustituidos por los padres elitistas).
- *nelitismo*: número entero de padres que pueden ser salvados por elitismo en la selección de hijos.
- $conf$: número real que indica la confianza con la que se calcularán los intervalos.
- *verbose*: booleano que especifica si se imprimirá o no cada *display_time* información sobre la generación, mejor fitness, tiempo etc. en la ejecución que se lleva a cabo.

- *display_time*: cantidad de segundos cada cual se imprimirá la infromación en caso de que $verbose = True$.
- *comp_param_inicio* y $comp\_param\_final$: números reales que expresan el intervalo en el que se moverá el análisis de robustez frente a cambios de parámetros.
- *comp_param_pasos*: número entero de valores dentro del intervalo [*comp_param_inicio*, $comp\_param\_final$] que se analizarán.
- *comp_param_name*: nombre del parámetro a analizar (cualquiera de los parámetros especificados antes de *verbose*).
- $file\_name$: nombre del archivo donde se guardará la gráfica correspondiente.

Un `Individuo` está compuesto por un genotipo (lista de permutaciones) y un fitness (valor de su fenotipo). Esta representación hace más clara y sencilla la implementación, además de asegurar que no se evaluará un mismo `Individuo` más de una única vez (a la hora de crearlo). En el mismo sentido de evitar evaluaciones, cálculos y ejecuciones repetidas, a la hora de cargar las ciudades (`cargar_ciudades`) calcularemos directamente la matriz de distancias entre las mismas. La función `distance_matrix` de `Scipy` permite calcular esta distancia para una p-norma de Minkowski concreta. Recordemos que esta métrica es la ecuación 1.

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{1}$$

, donde hemos escogido $p = 2$ para distancia euclidiana. De esta forma, la función objetivo (`fitness`) escoge uno a uno las ciudades según el orden definido por el genotipo y extrae la distancia entre ellas de la matriz de distancias. Esto convierte a la función objetivo en un simple sumatorio, evitando recalcular distancias cada vez.

La estructura que sigue el algoritmo genético está definida de forma modular y genérica en `generacion` para su reutilización posterior. El algoritmo carga la ciudades que se le han facilitado (`cargar_ciudades`, inicializa la población (`init_poblacion`) y llama a `generacion` *ngen* veces. Por su parte, `generacion` lleva a cabo 4 pasos: selección de padres (`seleccion_padres`), cruce (`cruce`), mutación (`mutacion`) y selección de hijos (`seleccion_hijos`). Esto permite crear algoritmos genético con o sin estas etapas y definir cada una de ellas con estrategias distintas. Una estrategia posible sería *no realizar* una selección de padres o un cruce.

En concreto, llevaremos a cabo una selección de padres por torneo, eligiendo para cada torneo de forma aleatoria *npadres* con posibilidad de repetición. Habrá tantos torneos como $tam\_poblacion$. En la selección de hijos se distinguen el caso sencillo de $nelitismo = 1$ del resto de valores posibles. En el caso sencillo, si el mejor padre es mejor que el mejor hijo, se selecciona la estrategia *peor* o *aleatorio* y se sustituye en la población de hijos. Tanto para el cruce como la mutación hemos creado un módulo desde el que se llamará a la estrategia concreta de cruce y mutación. Esto se podría extender a la selección de padres y a la selección de hijos, pero no se ha considerado como algo necesario. A continuación se puede ver un fragmento de código correspondiente al `cruce`. Se eligen aleatoriamente 2 padres (no se ha considerado cruce multiparental) dentro de la población de padres y, si se da la probabilidad de cruce *pcruce* se hacen las llamadas pertinentes a la estrategia o estrategias definidas. Estas estrategias de cruce deberán generar 2 hijos, `hijo_1` e `hijo_2`, que se añadirán a la población de elementos cruzados. En caso de que no se de la probabilidad, los padres pasarán directamente a la siguiente etapa. Si $tam\_poblacion$ es un número impar, el padre no seleccionado para cruce pasa directamente a la siguiente etapa.

```
    def cruce(pcruce, tam_poblacion, nciudades, padres, ...
                                                matriz_distancias):
    cruzados = []
    for i_cruce in range(tam_poblacion//2):
        indices = np.random.choice(np.arange(len(padres)),...
                                                  2, replace=False)
        padre_1 = padres[indices[0]]
        padre_2 = padres[indices[1]]
        padres = np.delete(padres, indices, 0)
        if np.random.random() <= pcruce:
            # LLAMADA A ESTRATEGIA DE CRUCE
            cruzados.append(hijo_1)
            cruzados.append(hijo_2)
        else:
            cruzados.append(padre_1)
            cruzados.append(padre_2)
    if tam_poblacion%2 == 1:
        cruzados.append(padres[0])
    return np.array(cruzados)
```

Algo más sencillo ocurre con la `mutacion` donde simplemente aplicamos una máscara de índices. Esta se genera si el número *pseudo-aleatorio*, que se asocia al individuo según su orden en la población, es menor que la probabilidad *pmutacion*. La estrategia de mutación deberá aplicarse solo a esos individuos que son guardados en la nueva población, tal y como se ve en el siguiente fragmento de código.

```
    def mutacion(pmutacion, tam_poblacion, poblacion, ...
                                    matriz_distancias, nciudades):
    mascara_mutacion = np.random.random(tam_poblacion) <= pmutacion
    poblacion[mascara_mutacion] = # LLAMADA A LA ESTRATEGIA DE MUTACION
    return np.sort(poblacion)
```

Respecto al cruce, se ha implementado la estrategia de cruce parcialmente mapeado. Esta necesita 2 genotipos y la lista de posiciones que define el segmento. Este cruce produce únicamente un hijo del primer genotipo con el segundo. Para las necesidades de nuestra función `cruce` la llamaremos 2 veces intercambiando los genotipos de lugar.

Creamos un hijo *vacío*. Representaremos los huecos libres de ese *vacío* por el valor `-1`. Copiaremos en el hijo el segmento del *genotipo_1*, guardaremos los elementos sobrantes, así como el segmento correspondiente del *genotipo_2*. Con esto podemos hallar la relación de pertenencia del sobrante con el segmento del segundo genotipo. Aquellos que no pertenezcan al segmento se marcan como `sin_coflicto`, mientras que los que sí como `con_conflicto`.

Aquellos sin conflicto se añadirán directamente en función de su posición en *genotipo_2*. En cambio, los que sí tengan conflicto pasarán por un proceso de resolución de conflictos. En este proceso se analizan los valores que ocupan las posiciones donde nos gustaría colocar a nuestros elementos con conflicto e intentaremos colocarlos en las posiciones de esos valores. De esta forma se deshace poco a poco la cadena de conflictos.

```
    def cruce_parcialmente_mapeado(genotipo_1, genotipo_2, ...
                                                        posiciones):
    # Segmentamos
    hijo = np.zeros(len(genotipo_1)) - 1
    mascara_segmento = (np.arange(len(genotipo_1)) >= posiciones[0])...
                       & (np.arange(len(genotipo_1)) <= posiciones[1])
    hijo[mascara_segmento] = genotipo_1[mascara_segmento]

    # Lo que no forma parte del segmento
    sobrante = genotipo_1[mascara_segmento==False]
```

```
    segmento_2 = genotipo_2[mascara_segmento]

    # Separamos en conjunto dentro y fuera del segmento ...
                                                del genotipo 2
    pertenencia = np.isin(sobrante, segmento_2)
    con_conflicto = sobrante[pertenencia]
    sin_conflicto = np.sort(sobrante[pertenencia==False])

    # Annadimos los elementos sin conflicto (que no ...
                  estan dentro del segmento del genotipo 2)
    indices_sin_conflicto = np.where(np.isin(genotipo_2, ...
                                             sin_conflicto))
    hijo[indices_sin_conflicto] = sin_conflicto

    # Tratamos conflicto
    for elem in con_conflicto:
        posicion = elem.copy()
        while(posicion != -1):
            genotipo_en_posicion = posicion
            posicion = hijo[np.where(genotipo_2 == ...
                                     genotipo_en_posicion)][0]
        hijo[int(np.where(genotipo_2 == genotipo_en_posicion)[0])]...
                                                          = elem
    return hijo
```

La estrategia de mutación empleada es la *mutación por intercambio*, donde se intercambian de posición 2 elementos elegido aleatoriamente dentro del genotipo. Otras funciones importantes que son complementarias al algoritmo genético son la generación de gráficas, los *wrappers* y las funciones principales.

En cuanto a la generación de gráficos disponemos de la función `generar_grafica`. Esta función por defecto genera curvas de progreso de la media del mejor individuo por generación haciendo uso de los valores de *fitness* proporcionados, sus intervalos de confianza cada cuantas generaciones se deben mostrar los mismos. Si el flag *robustez_parametro* no es `False`, este contendrá el parámetro frente al cual se le ha realizado al algoritmo un análisis de robustez. En la sección 4 se entrará en mayor detalle sobre las gráficas.

Para mejorar el desempeño en cuanto a tiempo de las ejecuciones, se han creado distintas funciones de tipo *wrapper* (envoltorios) que hacen las gestiones pertinentes para que la función `algoritmo_gentico` puede ser llamado de forma paralela utilizando hilos de la CPU. Por último, se han creado 2 funciones principales `main_progreso` y `main_robustez`. Estas se encargan de la carga de parámetros, la inicialización de las ciudades y las llamadas paralelas al algoritmo tantas veces como *niter* indique de la forma en la que se desea. Con esa información se hace uso de la generación de gráficas para curva de progreso, para la primera función, y la robustez, para la segunda función.

4. Evaluación Experimental

Llevaremos a cabo 2 experimentos para evaluar el desempeño del algoritmo en función de sus parámetros, en concreto, en función de la probabilidad de cruce *pcruce*. Para ello haremos uso de 2 tipos de gráficas para poder comparar esos desempeños en función de cada caso. Estas gráficas son: la curva de progreso y el análisis de robustez frente a cambios de parámetros.

Existen muchas formas de representar la curva de progreso, pero, en esencia, esta debe mostrar la evolución del valor de la función objetivo a lo largo de las generaciones, tiempo o

iteraciones (según la medida que se fije). Por un lado, podría usarse directamente el valor del mejor individuo a lo largo de cada generación. Si bien esto nos proporciona información suficiente en la mayoría de casos, para algunos problemas y algoritmos concretos sería interesante conocer la evolución de la población completa o, al menos, de la media de la misma. Un ejemplo podría ser el uso del algoritmo Harmony Search y sus distintas variantes. Por la forma en la que este funciona, añade 1 único individuo nuevo al cabo de cada generación. Por lo tanto, si mostráramos solamente el valor del mejor individuo, este podría mantenerse constante durante muchas generaciones. Esto nos haría pensar que el algoritmo no funciona correctamente y no está optimizando o realizando cambios sobre los individuos, cuando realmente estos cambios sí se están dando, pero lentamente. En la figura 1 podemos ver una curva de progreso de un Harmony Search que muestra el mejor y peor individuo de la población, así como el individuo que se acaba de crear en esa generación.

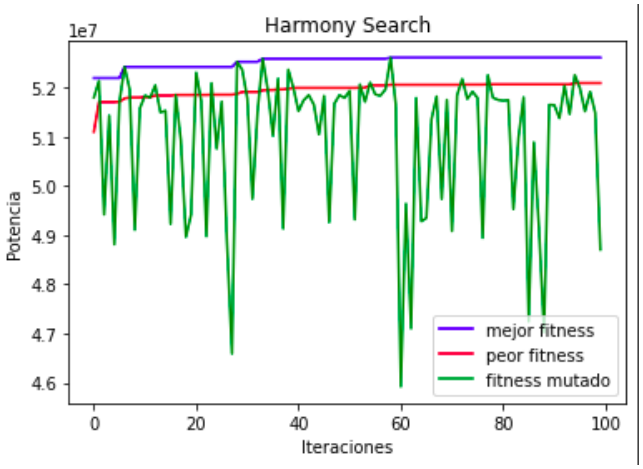


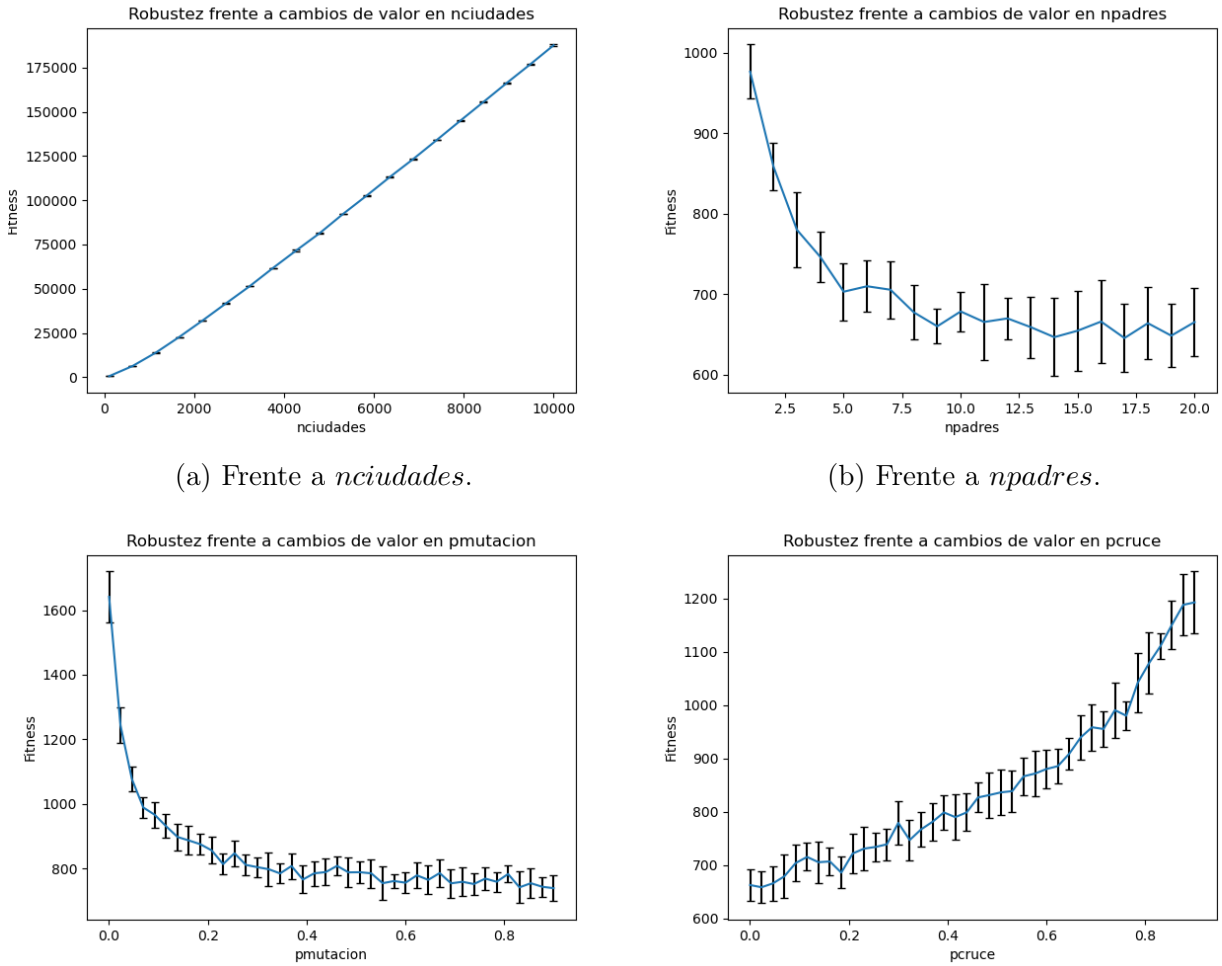
Figura 1: Curva de progreso de un Harmony Search.

Ante este tipo de casos, podemos tomar medidas como mostrar la media de la población y sus intervalos de confianza (o intervalos de varianza si se utiliza la desviación típica directamente). Por suerte no es el caso de nuestro algoritmo genético, pero este sigue teniendo el problema del carácter estocástico. Esto es un problema inherente a todos los algoritmos heurísticos, y por ese motivo es habitual realizar varias ejecuciones del mismo. Nuestra curva de progreso mostrará la media en las iteraciones del mejor individuo por generación, con su correspondiente intervalo de confianza cada *niter*.

La otra gráfica que utilizaremos es la referente al análisis de robustez del algoritmo frente al cambio de un parámetro. Este tipo de gráficas se utilizan calcular métricas, dado el rango analizado del parámetro y la variación absoluta del desempeño del algoritmo, para poder evaluar cuán resistente es a esos cambios. No es este nuestro objetivo directo. No nos interesa tanto la resistencia o robustez del algoritmo frente a estos cambios, sino una comparativa visual del desempeño y la convergencia del mismo en función de esos cambios. Utilizaremos como medida del *desempeño* del algoritmo su valor de la función objetivo al cabo de un *ngen* número de generaciones.

Llevamos a cambio un análisis inicial de la robustez del algoritmo frente a cambio en los parámetros *nciudades*, *npadres*, *pmutacion* y *pcruce* para poder elegir los valores más adecuados de estos. En la figura 2 podemos ver esta robustez para 5000 generaciones.

Respecto a *nciudades*, como era de esperar, la convergencia disminuye a medida que aumenta este parámetro (ver figura 2a). Nótese que la convergencia es inversamente proporcional al valor de la función de fitness. Como podemos ver, en los valores pequeños de *nciudades* el



(a) Frente a *nciudades*.

(b) Frente a *npadres*.

(c) Frente a *pmutacion*.

(d) Frente a *pcruce*.

Figura 2: Análisis inicial de la robustez del algoritmo.

valor del fitness aumenta ligeramente, pero a partir de 2000 se puede decir que este aumento es prácticamente linear. Nos hemos decantado por plantear 2 casos de estudio o experimentos: con 100 y con 10000 ciudades. El primero pretende ser una instancia sencilla del problema. Como hemos visto, la variación del desempeño en el rango $[1, 2000)$ no es muy grande. Por lo tanto, elegimos un valor no trivial, pero tampoco demasiado complicado. El segundo pretende ser una instancia compleja, lo más complicada posible, que ponga a prueba al algoritmo.

En relación a la robustez de *npadres* vemos en la figura 2b que para valores mayores que 7 la convergencia es muy rápida, pero sin demasiada variación. Esto significa que usar 10, 15 o 20 padres para los torneos no influirá de forma demasiado dispar en el algoritmo. En cualquiera de estos casos, *npadres* realiza una alta presión selectiva y una convergencia prematura. En el extremo contrario, elegir 1 único padre para el torneo es equivalente a una selección aleatoria de los mismos. Dentro del intervalo $[2, 7]$ no deseamos una convergencia demasiado rápida, ni demasiado lenta. Por esta razón elegimos el valor 3 para *npadres*.

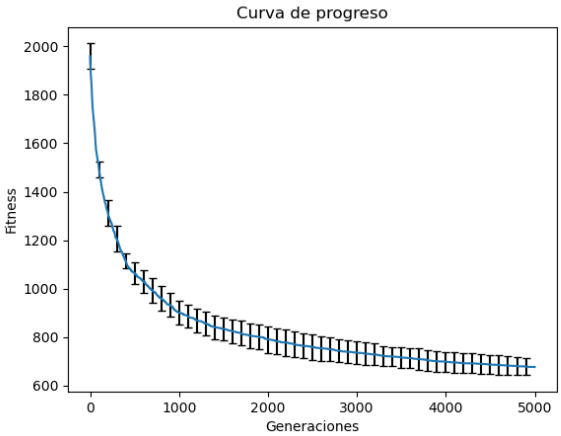
En cuanto a la probabilidad de mutación *pmutacion* (figura 2c) se dan unas circunstancias parecidas a *npadres*. Cuanto mayor es esta probabilidad, más temprana es la convergencia. Podemos fijar un umbral de aproximadamente 0,4 a partir del cual la convergencia del algoritmo no varía demasiado en función de *pmutacion*. En cambio, valores pequeños de este parámetro aseguran una convergencia lenta. Utilizaremos esta información para variar los resultados de

cada experimento a una convergencia temprana frente a otra tardía.

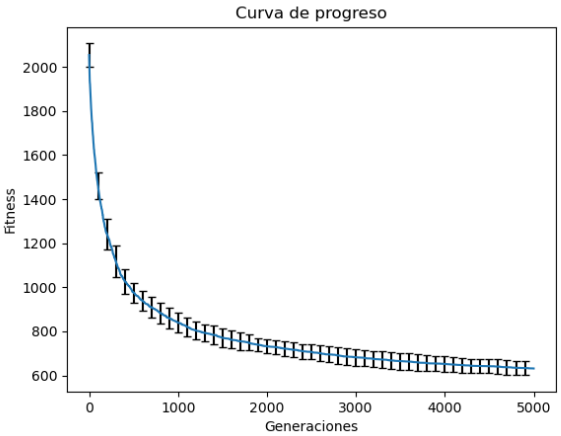
Por último y más importante, tenemos el análisis de *pcruce* en la figura 2d. Podemos definir 3 tramos en el rendimiento del algoritmo en base a este parámetro: hasta 0,2, entre 0,2 y 0,8, y desde 0,8 en adelante. En el primer tramo, la convergencia es muy rápida y, en algún caso, podría considerarse algo precipitado. En el segundo tramo se sigue un ascenso casi lineal, entardeciendo cada vez más la convergencia. A partir del tercer tramo la convergencia es extremadamente lenta.

Aclarar que la robustez de *npadres*, *pmutacion* y *pcruce* se ha analizado respecto a la instancia sencilla. Más adelante se llevará a cabo otro análisis de *pcruce* para la instancia compleja. La confianza de los intervalos que se muestran en estos análisis de robustez es del 95 % con 10 *niter* para la instancia sencilla y 3 o 5 *niter* para la compleja en función del *ngen*.

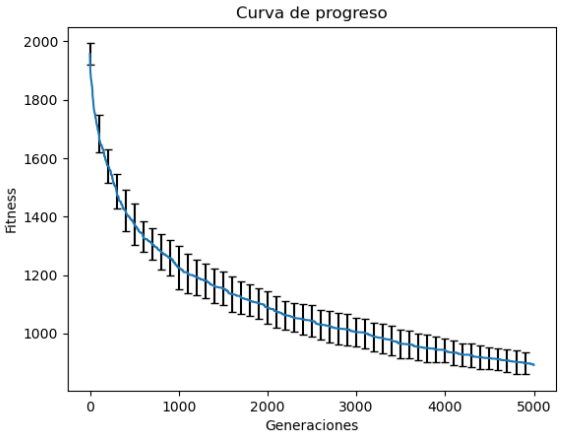
Para nuestro primer caso de estudio haremos uso de valores de *npadre* de 2 y 3, valores de *pcruce* de 0,1 y 0,6 y *pmutacion* 0,2 y 0,4. Para una instancia tan sencilla la probabilidad de mutación no influye de forma decisiva, salvo en sus valores extremos. En la figura 3 podemos ver algunas de las ejecuciones llevadas a cabo en este experimento.



(a) Parámetros: *npadres* = 2, *pcruce* = 0,1, *pmutacion* = 0,4.



(b) Parámetros: *npadres* = 3, *pcruce* = 0,1, *pmutacion* = 0,4.



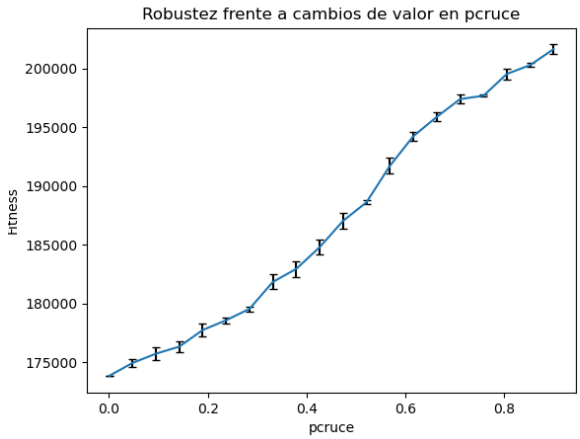
(c) Parámetros: *npadres* = 3, *pcruce* = 0,6, *pmutacion* = 0,2.

Figura 3: Curvas de progreso para el caso 1 (sencillo).

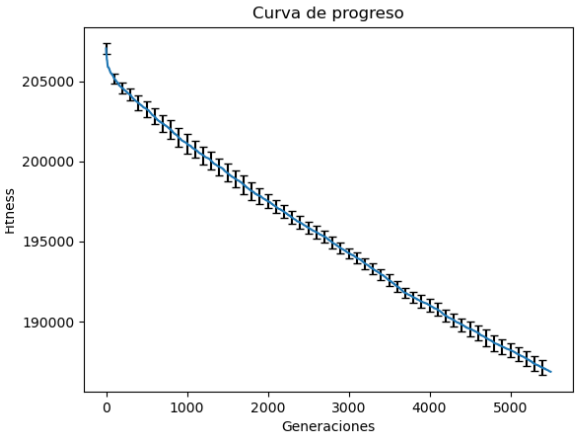
Entre la figura 3a y 3b la única diferencia en parámetros es *napdres*. Esta comparativa

refuerza nuestra idea de realizar una presión selectiva *ligeramente* [3] superior, ya que la convergencia es *ligeramente* mayor para 3 *npadres* que para 2. Esto tiene su coste en cuanto a la calidad de la solución (cuestión que también se refleja en la gráfica). Entre las figuras 3a 3b y la figura 3c la principal diferencia es la probabilidad *pcruce*. Es cierto que *pmutacion* varía con el objetivo que potenciar aún más la diferencia de convergencia entre resultados, pero el causante principal de esta diferencia es *pcruce*. Con esto se confirma el análisis de robustez realizado, ya que a mayor *pcruce* más tiempo es necesario para una convergencia.

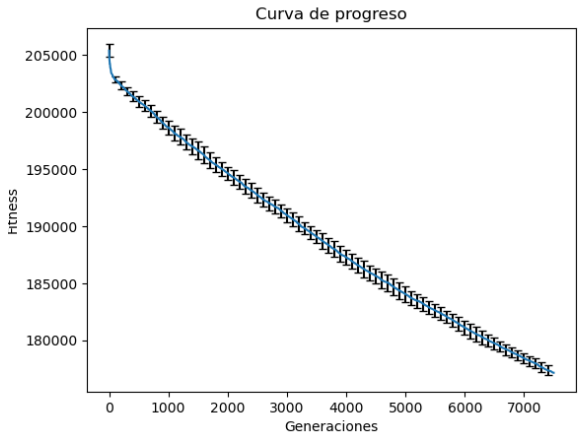
Finalmente, vamos a llevar a cabo el segundo experimento, obteniendo como resultados los reflejados en la figura 4. Por un lado, hemos repetido el análisis de robustez de *pcruce* (figura 4a) para la instancia compleja, suponiendo que los análisis de *pmutacion* y *npadres* se pueden inducir en general a cualquier caso. Este análisis, por las limitaciones computaciones y temporales, se ha llevado a cabo en 1500 *ngen* con una menor cantidad de puntos escogidos en el intervalo. Aún así, en líneas generales se mantienen las conclusiones del análisis sobre la instancia sencilla.



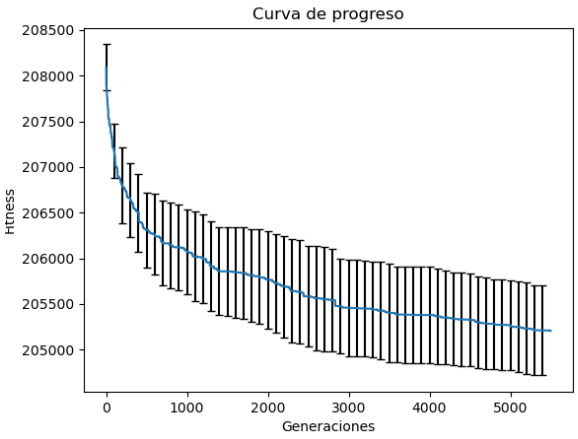
(a) Análisis de robustez de *pcruce* para el caso 2.



(b) Parámetros: *npadres* = 3, *pcruce* = 0,8, *pmutacion* = 0,95, *ngen* = 5500.



(c) Parámetros: *npadres* = 3, *pcruce* = 0,8, *pmutacion* = 0,95, *ngen* = 7500.



(d) Parámetros: *npadres* = 3, *pcruce* = 0,2, *pmutacion* = 0,9, *ngen* = 5500.

Figura 4: Gráficas para el caso 2 (complejo).

[3] Se hace hincapié en este *ligeramente*.

Podemos ver en la figura 4b que para valores altos de *pcruce*, aún con valores altos de *pmutacion* que deberían facilitar la convergencia, *pcruce* tiene un mayor peso y se impone. Extendiendo estos mismos parámetros a un mayor número de ejecuciones como en la figura 4c podemos ver que la solución aumenta en calidad y el algoritmo aún no converge. En contraste, en la figura 4d con una menor probabilidad de cruce *pcruce* sí se puede hablar de una convergencia, al menos relativa y en comparación a las otras figuras. Pero esta convergencia es a soluciones me peor calidad y con una variación muy alta en función de la ejecución concreta (para el valor de $conf = 95\,\%$).

5. Conclusiones

En la sección anterior se han presentado los resultado de los experimentos y discutido de forma objetiva y numérica las diferencias entre sí, sin ahondar en las implicaciones que estos resultados tienen. Que el valor de un parámetro llegue a mayor o menor convergencia es un hecho conciso, pero el uso que se puede dar a esto y las conclusiones que se extraen son algo distinto.

Sintetizando brevemente el apartado 4, si la probabilidad de cruce *pcruce* toma un valor bajo, convergerá más rápido y a soluciones menos eficaces; si la probabilidad de cruce toma valores altos, convergerá más lentamente y a soluciones más eficaces. Todo esto ha de tomarse en un sentido estocástico y general. A lo que nos referimos es que nada está completamente asegurado al tratarse de heurísticas aplicadas a un problema concreto con una representación concreta y unos operaciones de variación (cruce y mutación) concretos.

Para la instancia sencilla se podría hacer uso de valores *pcruce* altos con el objetivo de llegar a soluciones de mejor calidad al converger más tarde. Esto es, dado que el coste computacional para el primer experimento es bajo, la ineficiencia de esta configuración de parámetros (en términos de tiempo) no afecta demasiado. Pero esto tampoco asegura que a la larga se lleguen a soluciones suficientemente mejores para que el coste computacional sea justificable. Lo único que se asegura es que harán falta más generaciones para llegar a converger.

Ocurre lo contrario para la instancia más complicada. Dada la dificultad de la misma, puede interesar ahorrar tiempo de cómputo haciendo que el algoritmo converja antes y se obtenga una solución razonable, aunque no óptima. No obstante, al ser precisamente un problema más sofisticado, una convergencia temprana puede llevar a óptimos locales que no sean razonables porque, la propia complejidad puede impedir avanzar correctamente en el espacio de búsqueda. Siendo así, retrasar la convergencia nos permite llegar a soluciones de mayor calidad (como puede verse comparando las figuras 4b y 4d).

Por consiguiente, el desenlace de esta argumentación es que es necesario llegar a un balance entre eficacia, en términos de calidad de la solución, y eficiencia, en términos de coste computacional y temporal.

[4]*NOTA: Algunos de los archivos disponibles en el GitHub han sido sobreescritos por otras ejecuciones y no corresponden al nombre que tienen, como por ejemplo *robustez_ npadres_ 03C_ 06M_ 5000.png*. Pero esto es algo que ha ocurrido de forma ocasional, la mayoría de nombres sí corresponden a los parámetros con los que se ha ejecutado y con el nombre correcto del tipo de gráfica.