Análisis del rendimiento del operador de mutación de un algoritmo genético para la reordenación de imágenes

Enrique Vilchez Campillejo

7 de Enero, 2023

Resumen En la segunda guerra mundial, la policía secreta alemana del Stasi intentó destruir ciertos documentos para que no fuesen leídos. De forma similar, se plantea un problema de reordenación de 4 imágenes dadas, permutadas por sus filas. En concreto, se utiliza un algoritmo genético para resolver el problema mencionado. En este documento se presenta un método para la generación de individuos iniciales para el proceso de evolución, dos variantes de operadores de mutación, y el estudio de la influencia de los mismos sobre un problema de reordenación de imágenes, además del rendimiento de varias configuraciones del algoritmo genético, y su posterior análisis.

Palabras claves: Algoritmo genético, evolución, cruce, mutación, *fitness*, individuos iniciales, población, operador, reordenación, análisis

1. Introducción

1.1. Descripción del problema

Para problema que hay que resolver se consideran fotografías en blanco y negro, representadas como matrices de tamaño mxn donde cada posición i, j contiene un valor entre 0 y 255, que indica el tono de gris del píxel correspondiente. Se dispone de 4 imágenes desordenadas por filas de las que no se conoce su estado original, con que es preciso construir una solución algorítmica que con ciertos criterios permita reconstruirlas. Este problema es una variación del problema que surgió cuando investigadores encontraron documentos que la policía secreta alemana del Stasi intentó destruir [4]. Se muestra un mapa de calor de las 4 imágenes mencionadas, para representar sus tonos de grises. Para ello, se ha utilizado la herramienta seaborn [5] en lenguaje Python. Esto se utiliza también más adelante para mostrar las reordenaciones llevadas a cabo.

Como se puede observar en la figura 1, las imágenes tienen desordenaciones diferentes. La imagen número 1 es la más desordenada, mientras que la número 4 es la que menos lo está. En esta última se puede discernir la imagen de una mujer, quizás mirando a la persona que está sacando la foto, o pintando el cuadro correspondiente.

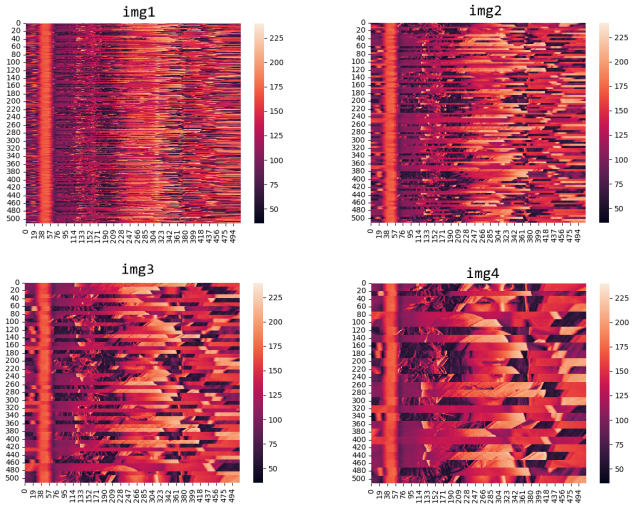

Figura 1: Imágenes a reordenar

1.2. Objeto de estudio

El objetivo principal de este trabajo es el de analizar el rendimiento de un algoritmo genético al modificar el operador de mutación, con el fin de analizar la influencia del mismo en la resolución del problema de reordenación de imágenes en blanco y negro, en concreto la reordenación de las filas desordenadas de las matrices que representan las imágenes de entrada. También se pretende presentar un método para no generar los individuos aleatoriamente de manera inicial.

2. Metodología

2.1. Algoritmo genético

Para la resolución del problema mencionado, se ha decidido utilizar un *framework* desarrollado en Python, llamado JMetalPy [1], debido a la facilidad del lenguaje, y a la sencillez que este aporta a la hora de incluir módulos preestablecidos de algoritmos de optimización y meta-heurísticas. Entre estos módulos, existen implementaciones sobre diferentes operadores de cruce, mutación, y selección, además de la implementación de algoritmos genéticos, la búsqueda local, o el recocido simulado (*simulated annealing*).

En este caso se ha utilizado un algoritmo genético, que resulta ser prácticamente igual que el concepto de algoritmo evolutivo base utilizado durante todo el curso. En concreto, este algoritmo viene predefinido en el *framework* como una clase que únicamente hay que instanciar, al igual que, como se ha comentado en el párrafo anterior, varios operadores de cruce, mutación, y selección de individuos. Además de los parámetros mencionados anteriormente, existen otros

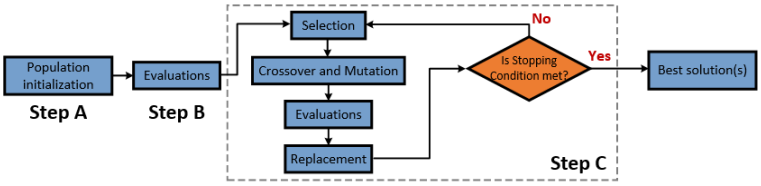

Figura 2: Esquema de Algoritmo Evolutivo en JMetalPy [2]

que son meramente numéricos, como pueden ser el tamaño de la población a evolucionar (padres), y el número de descendientes a generar (hijos)..

El algoritmo genético mencionado funciona de la forma que se puede ver esquematizada en la figura 2. Básicamente funciona como un agloritmo evolutivo básico, aunque en la implementación se detalla un poco más, pero no dista demasiado del concepto gráfico mostrado. Hablando también de la estructura general del código realizado para llevar a cabo los objetivos de este trabajo, cabe mencionar que existen principalmente 4 módulos desarrollados en Python, que incluyen respectivamente la estructura principal del programa, es decir, donde se ejecuta el algoritmo y se visualizan los resultados, el módulo que define el problema y lo permite evaluar, el módulo correspondiente con un operador de mutación modificado, y el módulo correspondiente a otro operador de mutación modificado.

El módulo del problema definido para este trabajo constra de varias partes. En primer lugar, cabe destacar que se ha construido una clase que hereda las características de otra primitiva de la librería, llamada *PermutationProblem*, que permite construir individuos en forma de permutaciones con números enteros (en este caso concreto), es decir, las estructuras utilizadas para la representación del problema son vectores de números enteros en forma de permutaciones. Algunos métodos de la clase construída han sido sobreescritos para adoptar una funcionalidad específica para el problema, como por ejemplo el método *create_solution()*, o el método *evaluate()*.

El primer método mencionado, permite generar la población inicial de una forma diferente a como se llevaría a cabo de manera aleatoria. En concreto, lo que se lleva a cabo es un bucle que itera desde el índice entero 1 hasta el número máximo de píxeles de una fila (en este caso 512 para todas las instancias del problema), para generar tantas divisiones equitativas de un individuo con una permutación ordenada de manera ascendente, como el doble del índice actual indique (se multiplica por 2 en cada iteración, para poder generar individuos variados), y posteriormente realizar una permutación aleatoria de estas divisiones. Para ejemplificar esto, si el bucle se encuentra en una iteración en la que el índice vale 16, entonces se toma el individuo generado de manera ordenada, este se divide en 16 partes equitativas, que después son permutadas aleatoriamente para generar un nuevo individuo. Esto se ha realizado de esta manera debido a que se han hecho pruebas sobre la generación aleatoria, y esta no obtiene un

mejor rendimiento que la generación planteada. Cabe destacar que no se reflejan estas pruebas en el documento debido al tiempo de ejecución que esto supone.

El segundo método permite evaluar cada individuo durante la ejecución del algoritmo genético, a través de un criterio conocido como distancia de Manhattan, que sirve para medir la distancia existente entre dos vectores numéricos. Esta métrica es una operación matemática la cual consiste en la suma de diferencias absolutas entre dos vectores, que en este caso son dos filas cualesquiera de cada imagen del problema. Estas operaciones se realizan para todas las parejas de filas posibles existentes en las imágenes, antes de ejecutar el algoritmo evolutivo, para no ocupar la memoria con operaciones repetidas durante el proceso. Cabe destacar que de un individuo (una permutación), la métrica que se ha escogido para indicar al algoritmo genético cuándo es mejor que otro, es la de la media de todas las distancias de Manhattan de este individuo de una fila con su contigua, lo que construye el *fitness*.

Además del programa principal, donde se ejecuta el algoritmo genético, y la definición del problema, también se han desarrollado dos clases, que se corresponden con una pequeña modificación aplicada a dos operadores de mutación de la librería utilizada. En primer lugar, se ha modificacdo el operador *PermutationSwapMutation*, ya que su versión original únicamente intercambia 2 elementos de un individuo, es decir 2 elementos de 512 para este caso concreto. Entonces, la nueva versión del operador permite intercambiar 20 para un sólo individuo. En segundo lugar, se realiza una modificación al operador *ScrambleMutation*, el cual permite al algoritmo mutar vectores (individuos) de 1 sola dimensión, mientras que previamente sólo permitía mutar vectores de 2 dimensiones, es decir, matrices. Esto último también se comenta como *bug* en un *issue* del propio repositorio de GitHub [3].

2.2. Procedimiento de experimentación

En este trabajo, además de la presentación de un criterio para generar soluciones iniciales, y dos modificaciones de operadores de mutación, se realiza un estudio de la influencia de 3 parámetros de mutación diferentes en el rendimiento del algoritmo genético utilizado, y en la calidad de las imágenes reconstruidas. La razón principal por la que se hace un estudio de la influencia del operador de mutación y no del de cruce es porque, tras varias pruebas iniciales y la monitorización de las soluciones evaluadas, se ha comprobado que para casi cualquier valor de cualquier parámetro, el algoritmo converge rápidamente gracias al operador de cruce, pero en el resto de iteraciones la evolución de los individuos depende casi por completo del operador de mutación.

Para realizar los experimentos, en primera instancia se ha decidido fijar ciertos parámetros. Se ha limitado el número de iteraciones a 200000, ya que por encima de este número el tiempo empleado es incogruente para este trabajo concreto, se limita el número de individuos padre a 20, el de la descendencia a 2 (todo ello para obtener individuos variados en una iteración), además de fijar el operador de cruce PMX (*Partially Mapped Crossover*), y el operador de selección de torneo binario (*Binary Tournament*).

Para la primera imagen (img1.txt), se ha decidido realizar 6 experimentos, ya que se prueban 3 operadores de mutación diferentes, que son *PermutationSwapMutation* (Sw), la variante *PermutationSwapMutationModified* (*Sw), y la variante *ScrambleMutationModified* (*Scr), con dos valores diferentes de probabilidad de mutación para cada operador. En concreto se prueban los valores 0.05 y 0.3, para tener en cuenta dos perspectivas lo más dispares posibles.

Con el fin de minimizar el número de pruebas a realizar, en primer lugar, se extrae el mejor valor del parámetro de probabilidad de mutación, y también el operador de mutación que obtiene mejores resultados para las 6 pruebas de la primera imagen, quedando sólo 1 prueba posible para cada imagen restante, con el mejor valor de probabilidad, y mejor operador obtenido. De esta manera, se llevan a cabo un total de 9 pruebas, de las que se mostrarán posteriormente varios datos estadísticos que apoyarán el análisis de resultados. Cabe destacar que se ha escogido la primera imagen para hacer la "competición de parámetros" debido a que es la menos susceptible a llegar rápidamente a óptimos locales por su grado de desordenación.

3. Resultados

Experim.	Tipo mutación	Prob. mutación	Mejor *fitness*	Media *fitness*	Mediana *fitness*	Std. *fitness*	Tiempo empleado (segundos)
a)	Sw	0.05	11254.33	13117.44	12657.24	1721.69	3405.96
b)	**Sw**	**0.3**	**8636.24**	**10197.88**	**9598.11**	1813.14	3257.90
c)	*Sw	0.05	15926.70	16476.90	16286.18	630.59	3269.86
d)	*Sw	0.3	15216.29	15886.89	15790.39	**567.44**	**2581.63**
e)	*Scr	0.05	16110.96	16714.39	16435.78	644.19	3361.94
f)	*Scr	0.3	14156.68	15036.70	14824.81	791.30	4212.50
g)	Sw	0.3	6055.30	6089.68	6055.30	77.69	3071.35
h)	Sw	0.3	4210.24	4248.71	4210.24	86.65	2920.66
i)	Sw	0.3	3313.64	3354.21	3313.64	90.15	2725.82

Cuadro 1: Resultados estadísticos de los experimentos realizados

En la tabla 1 se pueden ver los valores estadísticos obtenidos para los 9 experimentos realizados, además de otros valores numéricos. En primera instancia, cabe destacar que se tienen en cuenta, por un lado los 6 experimentos sobre la primera imagen, que van desde el a) hasta el f), y posteriormente los experimentos sobre las imágenes número 2, 3, y 4, que respectivamente corresponden con g), h), e i). Por ello, se han marcado en negrita los mejores valores de los 6 primeros experimentos, ya que los 3 últimos tienen mejores valores, pero debido a que comienzan la evolución de los individuos desde un punto más avanzado que para

la primera imagen, no se tienen en cuenta de esta manera. En primer lugar, se puede ver cómo la configuración b), con el operador *PermutationSwapMutation* (Sw), con el valor 0.3 de probabilidad de mutación, consigue el mejor *fitness*, la mejor media de *fitness*, y la mejor mediana de *fitness*. Los mejores valores de desviación estándar del *fitness*, y de tiempo empleado han sido obtenidos por la configuración d), en la que se emplea el operador *Sw, junto con el valor de probabilidad de mutación 0.3. Estos valores también se pueden ver reflejados en la gráfica b) de la figura 3, donde se puede observar la mejor convergencia de las 6 primeras configuraciones. En las gráficas a), y b), se puede observar una convergencia normal y corriente, mientras que en las gráficas c), d), e), y f) se puede observar una variabilidad mayor del *fitness*, además de la tendencia de caída del algoritmo en óptimos locales a medida que se suceden las iteraciones en el tiempo. Esto también se puede ver reflejado en los datos estadísticos mostrados en la tabla anterior, en la que se puede observar cómo la media y mediana del *fitness*, para la configuraciones a) y b) son más bajas, en comparación con las de las configuraciones c), d), e), y f). La desviación estándar no es significativa para indicar tendencias sobre estas configuraciones.

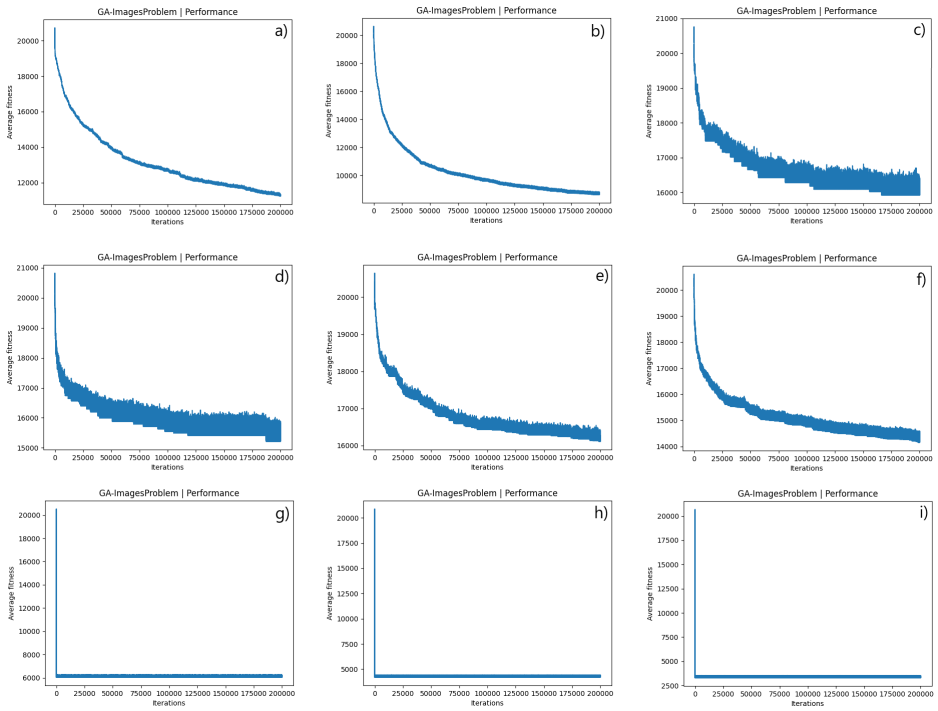

Figura 3: Rendimiento de cada configuración del algoritmo genético

Respecto a las configuraciones g), h), e i), que utilizan la configuración del algoritmo genético con el mejor operador de mutación, y valor de probabilidad de mutación (obtenidos de la "competición" de las 6 configuraciones anteriores, que son Sw, y 0.3), y que se corresponden respectivamente con las imágenes número 2, 3, y 4, se puede observar en todos los casos que el mejor valor de *fitness*, de la media, y de la mediana, prácticamente son iguales, lo que indica que la convergencia es mínima. De la misma manera, la desviación estándar del *fitness* también es mínima, por lo que no se muestra ningún tipo de tendencia en la sucesión de iteraciones en el tiempo. Esto también se puede ver reflejado en las gráficas de rendimiento g), h) e i), donde se puede observar gráficamente una convergencia completamente plana.

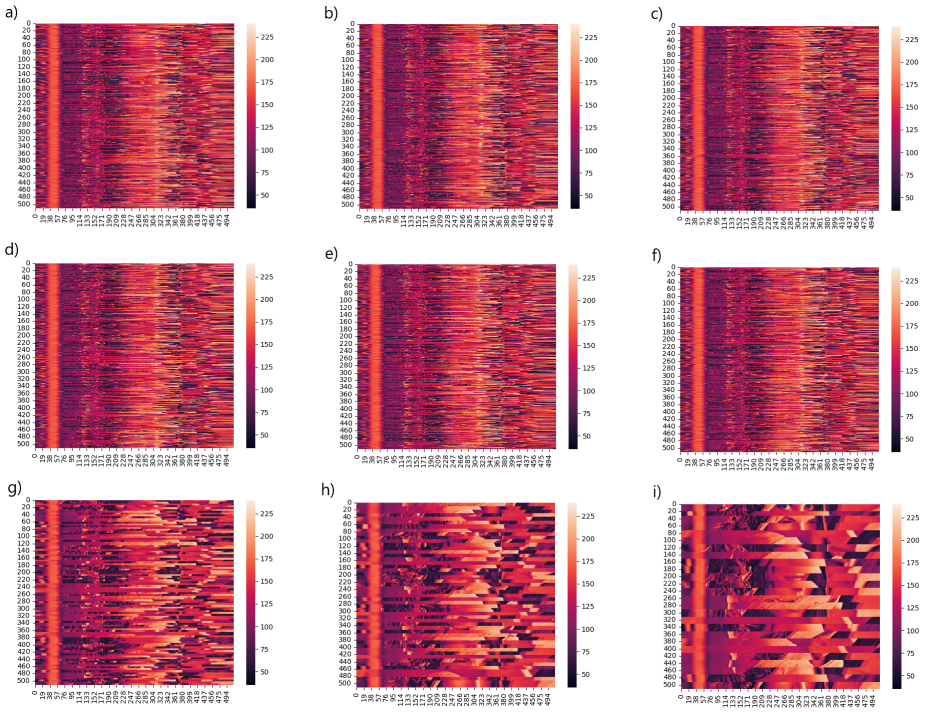

Figura 4: Reordenaciones obtenidas de cada configuración del algoritmo genético

Todo lo mencionado anteriormente se puede ver perfectamente reflejado en las reordenaciones del algoritmo genético mostradas en la figura 4. En primer lugar, sobre los 6 intentos de las 6 primeras configuraciones no se aprecia ningún patrón reconocible, pese a que incluso la configuración b) ha conseguido el mejor valor de *fitness*, incluso habiéndolo minimizado de ∼21000 a ∼8000. Con la mejor configuración de las 6 primeras, se ha intentado reordenar las imágenes número

2, 3, y 4, pero sin éxito, manteniendo prácticamente la misma estructura original en la reordenación.

4. Conclusiones

En este documento se ha presentado un método para generar los individuos iniciales de forma que se tenga en cuenta, de cierta manera, la reordenación inicial que puedan presentar las imágenes, además de dos variantes de operadores de mutación incluidos en el *framework* JMetalPy, y el estudio de la influencia de los mismos sobre las imágenes a reordenar.

Como resultado de los experimentos realizados, se ha observado que el mejor operador de los estudiados, es el de *PermutationSwapMutation* sin modificaciones, junto con el valor 0.3 del valor de probabilidad de mutación. En las gráficas mostradas, se ha podido ver reflejado lo que se comentó en el apartado de "Procedimiento de expermientación", donde se explica que, por un lado el operador de cruce PMX no es muy eficiente para este algoritmo, en este caso concreto, al igual que los operadores de mutación utilizados en los experimentos.

Para la primera imagen se puede ver que la configuración b) es la mejor, obteniendo el mejor valor de *fitness* respecto a todas las demás, aunque sin éxito en la reordenación. Para las imágenes 2, 3, y 4, la configuración óptima conseguida en pruebas anteriores, no permite converger lo más mínimo a lo largo de las iteraciones sucedidas en el tiempo. Esto último ocurre porque, aunque el parámetro de mutación se modifique, o su valor de probabilidad se cambie, el nuevo individuo no es capaz de superar a la ordenación preestablecida por la imagen, ya que todas estas configuraciones probadas son, por lo que se ha comprobado, muy susceptibles a caer en óptimos locales.

4.1. Trabajo futuro

Sería interesante probar otros algoritmos, operadores de cruce, e incluso configuraciones del operador de mutación para poder obtener mejores soluciones en el proceso de evolución, ya que en este trabajo no se ha tenido éxito en el proceso de reordenación de las imágenes.

Referencias

[1] Antonio Benítez-Hidalgo et al. "jMetalPy: A Python framework for multi-objective optimization with metaheuristics". En: *Swarm and Evolutionary Computation* 51 (dic. de 2019), pág. 100598. ISSN: 2210-6502. DOI: 10.1016/J.SWEVO.2019.100598.

[2] *Daily Blog - jMetalPy*. URL: https://gu-youngfeng.github.io/blogs/jmetalpy.html#2.-Multi-objective-Optimization.

[3] *Permutation problem operators · Issue 68 · jMetal/jMetalPy*. URL: https://github.com/jMetal/jMetalPy/issues/68.

[4] *Piecing Together the History of Stasi Spying - The New York Times*. URL: https://www.nytimes.com/2021/08/10/arts/design/stasi-archive-puzzle.html.

[5] Michael Waskom. "seaborn: statistical data visualization". En: *Journal of Open Source Software* 6 (60 abr. de 2021), pág. 3021. DOI: 10.21105/JOSS.03021.