ALGORITMO MULTIOBJETIVO BASADO EN AGREGACIÓN

DOCUMENTACIÓN

Estrella Aguilera Moreno. UVUS: estagumor

Índice

INTRODUCCIÓN

En esta práctica el objetivo era aprender a resolver un algoritmo multiobjetivo por medio de la agregación en el lenguaje de programación que prefiriésemos y ganar al que proporcionaba el profesor. Esto consiste en diferenciar cada objetivo como un subproblema y resolverlo como si fuese un algoritmo monobjetivo, para después tener como función objetivo común una “suma” de de los objetivos individuales.

En nuestro caso, el algoritmo consta de dos funciones objetivo, cuyos valores ideales se encuentran en el frente de Pareto. Utilizando un vector de referencia al que denominamos z, debíamos intentar aproximar cada algoritmo monobjetivo a su mejor valor actual; que podía ir “mejorando” en cada iteración si se encontraba un individuo/s que lo propiciase/n.

IMPLEMENTACIÓN

Para la implementación de este algoritmo nos basamos en una clase principal denominada Individual. La decisión de realizar una clase en vez de simplemente utilizar vectores ha sido que, al llegar a la utilización de los operadores evolutivos; ubicar a los vecinos y utilizarlos se me hacía muy complicado. Así que decidí que era mejor crear una clase en la que guardase todo lo que compete al individuo: un id identificativo, los pesos asociados, el fitness de f1 y f2, el gen que le corresponde y la posición de los T vecinos más cercanos conforme a la población.

Los ficheros que contiene el trabajo son:

- AuxiliaryMethods.py → Contiene todos los métodos utilizados por el algoritmo. Es el corazón del trabajo.
- MainProgram.py → Actúa como "interfaz" del algoritmo. Permite el establecimiento de las variables de entrada de los métodos e imprime resultados tanto en gráficos como en consola de los cálculos.
- Classes.py → Contiene las clases creadas para el algoritmo. La clase Individuals se encuentra en este archivo.
- Trash.py → Son versiones antiguas y experimentos de los métodos. Algunos no están en versión estable. Supuse que era interesante para ver el progreso del código.
- OutFilesZDT3 → Contiene archivos .txt con los resultados del algoritmo implementado por mí. Se utilizan para hacer comparaciones justas.
- OutFilesZDT3Ideal → Contiene archivos .txt con los resultados del NSGAII. Se utilizan para hacer comparaciones justas.
- MetricsResult → Contiene las imágenes correspondientes a la respuesta de MainProgram.py y de las métricas que proporcionó el profesor.
- Evaluation.pdf → Contiene comparaciones entre NSGAII y mi algoritmo.

Inicialización

Al **inicializar** el algoritmo se sigue el siguiente proceso:

- Primero, por medio de un método denominado "Weight vectors" al que se le proporciona el tamaño de la población N, creamos N **vectores peso** de dos elementos con los que posteriormente se crea una clase Individual (una para cada vector peso). Al principio andaba muy perdida así que creaba N vectores peso de N elementos, porque no había entendido cuantas funciones objetivo teníamos. En un archivo denominado trash.py se ubica el método, por si es de interés. Los vectores están equiespaciados para recorrer todo el frente y la suma es uno para que estén proporcionados. La inicialización de la clase por medio de los pesos está hecha así para seguir el orden del documento proporcionado de instrucciones de la práctica. Los vectores peso se crean para utilizarlos posteriormente en la "suma" de las funciones monobjetivo de la que hablamos anteriormente.
- A continuación obtenemos los **vecinos** de cada Individual, los cuáles utilizaremos en el futuro con los operadores evolutivos. Utilizo el método "Neighbors", cuyas variables de entrada son los N Individuals creados anteriormente y una variable T, que especifica cuántos vecinos tendrá cada Individual. El funcionamiento es el siguiente: para cada Individual se obtienen los índices correspondientes a los T Individuals con los vectores peso más cercanos al vector peso actual por medio de la distancia euclídea. En principio, en vez de almacenar los índices conforme al array de Individuals directamente almacenaba al Individual en cuestión. Lo tuve que cambiar porque me daba problemas en los operadores evolutivos (explicación en su apartado correspondiente).

- Posteriormente el método “Poblation” añade a cada Individual un **cromosoma** compuesto de el elementos acotados por el espacio de búsqueda. Cada elemento se obtiene de forma aleatoria. Las variables de entrada específicamente son: el, los N Individuals que hemos estado utilizando y el tamaño mínimo y máximo proporcionado por el espacio de búsqueda. El cromosoma de cada Individual se utiliza para calcular su fitness.
- Para seguir se utiliza el método “functionZDT3” que se encarga de calcular el **fitness** para cada Individual de las funciones objetivo f1 y f2 en base a ZDT3. Es el único método que no es reutilizable para CF6. Para realizarlo he utilizado la documentación de la primera práctica en la que se explica la función ZDT3.
- Por último se utiliza la función “updateZ” que crea un **vector z** que guarda el mejor valor para todos los Individuals de las funciones objetivo. Si se utiliza posteriormente, actualiza z para cada Individual de la población si procede (explicación en apartado de operadores evolutivos).

Iteraciones

Para las **iteraciones** del algoritmo se utiliza un método denominado “diferencial evolution” que realiza todo el proceso en cuestión. Está planteado así para que el fichero de mainProgram.py solo tenga las variables que se le introducen a los métodos y se vea claramente el proceso general sin entrar en detalles del algoritmo.
Los métodos están hechos de tal modo que se puedan reutilizar independientemente de que se esté produciendo o no la inicialización del algoritmo, por lo que son utilizados aquí.
El proceso que se sigue para cada individuo de la población es el siguiente:

- **Se obtienen los vecinos**. En principio los vecinos estaban guardados en el individuo, pero como podían ser sustituidos por un individuo nuevo; la sustitución de ese individuo en todos los vecinos de los individuos de los que este era vecino era prácticamente imposible (en términos de eficiencia), por lo que cambié el código. Lo siguiente que hice fue modificar el individuo al que se iba a “sustituir” (introduciéndole todos los datos el nuevo individuo) confiando en que su posición en memoria sería la misma por lo que no tendría que preocuparme de actualizar los vecinos de los demás individuos; pero no me funcionaba correctamente tampoco, así que lo tuve que volver a cambiar. La solución actual es utilizar los índices guardados en su inicialización para saber cual es la posición del vecino en la población actual (es decir, que todos los individuos tienen una posición al nacer que no cambia, lo que cambia es el individuo que le sustituye) y obtenerlo a partir de esta.
- **Se muta un nuevo cromosoma.** A continuación se escoge de forma aleatoria tres vecinos DIFERENTES a partir de los cuales por medio de la **mutación diferencial** se obtiene un nuevo cromosoma. Si algún elemento del cromosoma se sale del espacio de búsqueda, se le asigna un valor aleatorio dentro de los límites. Al principio, siguiendo esta documentación https://es.wikipedia.org/wiki/Evoluci%C3%B3n_diferencial, hice N cromosomas que además utilizaban a la población entera para elegir los tres vectores, pero al ver que solamente había que crear uno nuevo y a partir de los vecinos, lo cambié.
- **Se recombina el nuevo cromosoma.** Aleatoriamente de entre los vecinos se escoge un individuo y para cada elemento de su cromosoma se crea un valor aleatorio entre 0 y 1. Si sobrepasa a GR se coge ese elemento y se posiciona en el cromosoma que habíamos creado nuevo, si no; se pasa al siguiente elemento. Con un GR de 0,5 la recombinación prácticamente no servía para nada. El valor óptimo es 0,65.
- **Se compara el individuo nuevo con los vecinos.** En esta parte del método para cada vecino hago lo siguiente: creo un nuevo individuo que contenga sus pesos y vecinos, le asigno el cromosoma nuevo y calculo y guardo su fitness. Creo un individuo nuevo para tener la posibilidad de guardar los individuos que no mejoren en un fichero (no está hecho).

Posteriormente, por medio de la función “updateZ” que expliqué en la inicialización, compruebo si los valores de f1 y f2 del nuevo individuo mejoran a los que actualmente contiene z. Si es así, los reemplazo. A partir de aquí utilizo la función de Tchebychef tanto con el vecino actual como con el nuevo individuo, comparo los valores y si para el nuevo individuo es más pequeña, sustituyo ese vecino en la población actual por el nuevo individuo. Esto se realiza para todos los vecinos del individuo y para todos los individuos de la población.

Por alguna razón la función no me termina de funcionar correctamente. He realizado debug cambiando el código varias veces y realizando print y no encuentro el error. Mejora la función proporcionada pero la gráfica resultante me da a entender que los f1 se anclan y no se mueven.

La comparación de mi función con la función proporcionada en prácticas me ha costado porque mi manejo de C es escaso. Solo lo di un mes en primero de carrera y no lo he vuelto a utilizar… Así que me he encontrado muchas complicaciones con eso.

Para ver los cambios realizados, también está este repositorio de Github https://github.com/Wordsan/multi-Objective-Algorithm

No están los cambios desde el principio pero si desde una distancia razonable. Lo mismo es interesante mirarlo.