Metaheurísticas Poblacionales

Rosas Francisco, Gurruchaga Luciano

6 de julio de 2022

Resumen

El objetivo de este documento será el estudio, analisis y comparación de dos algoritmos metaheuristicos poblacionales, Algoritmo Genético **GA** y Evolución Diferencial **DE**. Los mismos serán sometidos a una serie de pruebas en funciones complejas conocidas *Ackley 1, Ackley 2, Ackley 3 y Ackley 4* con variacion en el tamaño de las dimensiones. Luego, se verá el desepeño de dichos algoritmos descubriendo cual es el metodo mas adecuado para la resolucion de problemas en un espacio grande de soluciones complejas de optimizar. Debido a esta problematica, se comenzará con la definicion de conceptos básicos necesarios, alguna mención del estado del arte al día de la fecha, y expondremos las implementaciones y consideraciones de las soluciones. Finalmente se procedera a la comparacion de los resolutados del objeto de estudio y por que **DE** es novedoso.

1. Introducción

X=10
$X = 10$

1.1. GA- Algoritmo Genético

Los Algoritmos Geneticos conocidos como GA por sus siglas en ingles Genetic Algorithm, son una analogía a la selección natural, es decir, que estan inspirados en ella.

Se trabaja con un conjunto de soluciones al que llamaremos **poblacion**, cada solucion particular o **individuo** es representada por medio de un vector de valores discretos llamado **cromosomas**. A los simbolos que conforman dichos vectores se les llama **genes**. Los cromosomas evolucionan a lo largo de distintas iteraciones, las llamadas **generaciones**. En cada generacion, los cromosomas son evaluados usando alguna medida de aptitud o funcion objetivo. Las siguientes generaciones, nuevos cromosomas, son generadas aplicando operadores genéticos repetidamente a la generacion actual, siendo estos los operadores de selección (selection), cruzamiento (crossover), mutación (mutation) y reemplazo.

1.2. DE- Evolución Diferencial

En la computación evolutiva (Familia de algoritmos de optimización global inspirados en la evolución biológica - rama de la Inteligencia Artificial)[2], evolución diferencial es un método que optimiza un problema intentando mejorar una solucion candidata iterativamente con respecto a una medida de calidad determinada. Estos metodos son comunmente conocidos como "Metaheuristicas", ya que no hace ninguna suposición(o muy pocas) sobre el problema que se está optimizando y puede buscar espacios muy grandes de soluciones candidatas. Sin embargo, las metaheuristicas como DE no garantizan encontrar la solucion optima.

DE es usado para funciones multidimencionales de valores Reales, sin utilizar el "gradiente"del problema a ser optimizado, en otras palabras, **DE** no requiere o no necesita que el problema a optimizar sea diferenciable, como en el clasico caso del metodo de "descenso del gradiente"[3] o "Quasi-Newton method"[4]. Por lo tanto, la evolucion diferencial también puede utilizarse en problemas de optimización que ni siquiera son continuos, son ruidosos, cambian con el tiempo, etc. La ED optimiza un problema manteniendo una población de soluciones candidatas y creando nuevas soluciones candidatas combinando las existentes según sus sencillas fórmulas, y quedándose con la solución candidata que tenga la mejor puntuación o aptitud en el problema de optimización en cuestión. De este modo, el problema de optimización se trata como una caja negra que simplemente proporciona una medida de calidad dada una solución candidata y, por tanto, el gradiente no es necesario.

La ED fue introducida por Storn y Price en la década de 1990. Se han publicado libros sobre los aspectos teóricos y prácticos del uso de la ED en la computación paralela, la optimización multiobjetivo y la optimización restringida, ymuchos otros estudios en diversas áreas de aplicación.

1.3. Funciones Akcley

Las funciones de prueba son importantes para validar y comparar el rendimiento de los algoritmos de optimización algoritmos de optimización. En la literatura se han publicado muchas funciones de prueba o de referencia; Sin embargo, no existe una lista o conjunto estándar de funciones de referencia. Lo ideal es que las funciones de prueba deben tener diversas propiedades para que puedan ser realmente útiles para probar nuevos algoritmos de una de forma imparcial.

En general, los problemas sin restricciones pueden clasificarse en dos categorías: funciones de prueba y los problemas del mundo real.

Las funciones de prueba son problemas artificiales, y pueden utilizarse para evaluar el comportamiento de un algoritmo en situaciones a veces diversas y difíciles. Los problemas artificiales pueden incluir un único mínimo global, uno o varios mínimos globales en presencia de muchos mínimos locales, valles largos y estrechos, efectos de espacio nulo y superficies planas. Por otro lado,

los problemas del mundo real se originan en diferentes campos como la física, la química ingeniería, matemáticas, etc. Estos problemas son difíciles de manipular y pueden contener expresiones algebraicas o diferenciales complicadas y pueden requerir una cantidad significativa de datos para compilarlos.

Una función con más de un óptimo local se denomina multimodal. Estas funciones se utilizan para probar la capacidad de un algoritmo para escapar de cualquier mínimo local. Si el proceso de exploración de un algoritmo está mal diseñado entonces no puede buscar en el paisaje de funciones de manera efectiva. Esto, a su vez, hace que un algoritmo se quede atascado en un mínimo local. Las funciones multimodales con muchos mínimos locales se encuentran entre la clase de problemas más difícil para muchos algoritmos. Las funciones con superficies planas suponen una dificultad para los algoritmos, ya que la planitud de la función no da al algoritmo ninguna información para dirigir el proceso de búsqueda hacia los mínimos.

Entonces, la comprobación del comportamiento, la fiabilidad, la eficiencia y la validación de los algoritmos de optimización se lleva a cabo con frecuencia utilizando un conjunto elegido de puntos de referencia estándar comunes o funciones de prueba de la literatura.[5]

2. Descripción del Problema

En este breve trabajo de estudio de algoritmos metaheuristicos, utilizaremos los dos enfoques o modelos algoritmicos descriptos en la seccion previa. Que son el algoritmo genetico y de evolución diferencial los cuales construiremos y pondremos a prueba frente a un conjunto acotado de funciones de prueba extraidas de la bibliografía[5], para, luego de extraer los resultados obtenidos, realizar un analisis comparativo de su comportamiento frente a diversos aspectos.

2.1. Conjunto de funciones Ackley a utilizar

$$f_1(x) = -20e^{-0,02}\sqrt{D^{-1}\sum_{i=1}^{D} x_i^2} - e^{D^{-1}\sum_{i=1}^{D} cos(2\pi x_i)} + 20 + e$$

$$-35 \leq x_i \leq 35, f(x^*) = 0$$

$$f_2(x) = -200e^{-0,02}\sqrt{x_1^2 + x_2^2}$$

$$-32 \leq x_i \leq 32, f(x^*) = -200$$

$$f_3(x) = 200e^{-0,02}\sqrt{x_1^2 + x_2^2} + 5e^{cos(3x_1)+sin(3x_2)}$$

$$-32 \leq x_i \leq 32, x^* = (0, \approx -0{,}4), f(x^*) \approx -219{,}1418$$

$$f_4(x) = \sum_{i=1}^{D} (e^{-0,2} \sqrt{x_i^2 + x_{i+1}^2} + 3(cos(2x_i) + sin(2x_{i+1}))$$

$$-35 \leq x_i \leq 35, x = f(\{-1{,}479252, -0{,}739807\}, \{1{,}479252, -0{,}739807\}), f(x^*) \approx -3{,}917275$$

3. Propuesta algorítmica

3.1. El algoritmo Genetico

Un algoritmo genetico es una metaheuristica poblacional inspirada en el proceso de la seleccion natural.

Contamos con una poblacion P, dicha poblacion esta formada por un conjunto de soluciones individuos I, cada individuo I esta representado por un conjunto de cromosomas, los cuales a su vez estan formados una secuencia de genes g_i de valor discreto, que representan una solucion al problema, formalmente:

$$P = \{i_0, i_1, .., i_n\}$$

$$I = \{(g_0, g_1, .., g_n)_0, (g_0, g_1, .., g_n)_1, ..., (g_0, g_1, .., g_n)_n\}$$

Se toman todos los individuos de la poblacion $I \in P$, se obtiene una medida de "desempeño"de cada solucion I, aquellos con mejor desempeño son mas propensos a reproducirse, como sucede en la seleccion natural donde los mas aptos sobreviven y tienen mas posibilidades de trasmitir sus genes.

Existen distintas formas de seleccionar que individuos se reproducen, nosotros utilizaremos la *"seleccion basada en torneos"* la cual consiste en seleccionar de forma aleatoria dos individuos distintos de la poblacion, y , aquel con mejor desempeño sera quien pueda reproducirse y por tanto sus genes seran "heredados.[en] la siguiente poblacion P. Sumado a esto existe una probabilidad de cruza y mutacion de los genes al reproducirse.

Para la primera utilizaremos el operador de cruce **PMX** o *cruce por emparejamiento parcial*. Consiste en elegir un subsegmento de los genes de uno de los progenitores y cruzarlos preservando el orden y la posición de la mayor cantidad de genes posible del otro manteniendo la coherencia. En cuanto a la mutacion utilizaremos una *permutacion simple* entre dos genes aleatorios g_i, g_j del mismo individuo.

A medida que avanzan las generaciones, en cada poblacion iran quedando aquellos individuos que cuentan con los "mejores"genes de generaciones anteriores, por tanto los de mejor desempeño. Recuperando al mejor individuo historico (de todas las generaciones) obtenemos una solucion aproximada aceptable.

3.2. El algoritmo Evolucion Diferencial

La evolución diferencial es un enfoque heurístico para la optimización global de funciones espaciales continuas no lineales y no diferenciables.

El algoritmo comienza iniciando aleatoriamente una población de vectores de decisión de valor real, también conocidos como genomas o cromosomas. Estos representan las soluciones candidatas al problema de optimización multidimensional.

En cada iteración, el algoritmo introduce mutaciones en la población para generar nuevas soluciones candidatas. El proceso de mutación añade la diferencia ponderada entre dos vectores de la población a un tercer vector, para producir un vector mutado. Los parámetros del vector mutado se mezclan de nuevo con los parámetros de otro vector predeterminado, el vector objetivo, durante un proceso conocido como cruce que pretende aumentar la diversidad de los vectores de parámetros perturbados. El vector resultante se conoce como vector de prueba. Estas mutaciones se generan de acuerdo con una estrategia de mutación, que sigue una convención de nomenclatura general de $DE/x/y/z$, donde DE significa Evolución Diferencial, mientras que x denota el vector a ser mutado, y denota el número de vectores de diferencia considerados para la mutación de x, y z es el tipo de cruce en uso. Por ejemplo, las estrategias populares

$DE/rand/1/binyDE/best/2/bin$ especifican que el vector x puede elegirse al azar (rand) de la población, o bien se selecciona el vector con el menor coste (best); que el número de vectores de diferencia considerados es 1 o 2; y que el cruce se realiza según experimentos binomiales independientes (bin).

Una última operación de selección sustituye el vector objetivo, o el padre, por el vector de prueba, su descendiente, si este último arroja un valor de función objetivo menor. De este modo, el descendiente más apto se convierte en un miembro de la nueva población generada, y posteriormente participa en la mutación de otros miembros de la población. Estas iteraciones continúan hasta que se alcanza un criterio de terminación. (como el agotamiento de las evaluaciones funcionales máximas).

El algoritmo de evolución diferencial requiere muy pocos parámetros para funcionar, a saber, el tamaño de la población, NP, un factor de escala real y constante, $F \in [0, 2]$, que pondera la variación diferencial durante el proceso de mutación, y una tasa de cruce, $CR \in [0, 1]$, que se determina experimentalmente. Esto hace que el algoritmo sea fácil y práctico de utilizar. [6]

En este caso utilizaremos un algoritmo de evolución diferencial con el esquema "$DE/rand/1/bin$" para ejecutar con las funciones ackley.

4. Resultados y estadisticas

Probamos el algoritmo en seis instancias diferente del problema, cada instancia fue ejecutada 20 veces con diferentes semillas $i = 0, 1..., 19$. A continuacion presentamos las estadisticas de las mejores soluciones sin rotacion y con rotacion:

En esta seccion mostraremos algunos resultados obtenidos de las multiples ejecuciones de diferentes instancias del problema.También, posteriormente un analisis, comparación y evaluación de estos resultados

4.1. Resultados instancia

Problema :
Algoritmo:

Podemos observar como el permitir rotaciones de rectangulos se aprovecha mejor es espacio de la tira para todas las instancias, reduciendo asi la cantidad de desperdicio de forma sustancial. Para este tipo de problemas de alta complejidad vemos que los algoritmos geneticos tienen un desempeño mas que aceptable, siendo una excelente forma de encarar este tipo de problemas.

5. Conclusiones

Dada la simplesa de los algoritmos y su desempeño a la hora de encontrar una solucion eficiente, las metaheuristicos poblacionales, en este caso Algoritmos Geneticos, son una excelente forma de atacar problemas de complejidad NP hard cuya funcion de evaluacion sea simple (minimo valor posible) como es el caso de SPP, problema que surje en distintas areas ademas de las obvias como pueden ser aprovechamiento de material (madera, vidrio, metal,etc) como en la computacion donde se modelan jobs que requieren una parte contigua de la memoria durante un período de tiempo determinado entre otros.

Por ultimo, los GA, son una prometedora herramienta para la industria.

Agradecimientos

Prof. Guillermo Leguizamon

Referencias

[1] .[Clinton Sheppard, 2018]Genetic Algorithms with Python

[2] .[Wikipedia]`https://en.wikipedia.org/wiki/Genetic_algorithm`

[3] .[Wikipedia]`https://en.wikipedia.org/wiki/Evolutionary_computation`

[4] .[Wikipedia]`https://en.wikipedia.org/wiki/Gradient_descent`

[5] .[Wikipedia]`https://en.wikipedia.org/wiki/Quasi-Newton_method`

[6] .[Cornell University - Ithaca, New York]`https://arxiv.org/abs/1308.4008v1` A Literature Survey of Benchmark Functions For Global Optimization Problems

[7] .[Machine Learning Mastery]https://machinelearningmastery.com/differential-evolution-from-scratch-in-python Differential Evolution from Scratch in Python

6. Código del Algoritmo

```
def genetiquear():
        while(not finish):
                print("codeando")
return "code"
```

Índice