Metaheurística: Práctica Alternativa al Examen
Búsqueda Ramificada con Momentos

David Cabezas Berrido

20079906D

Grupo 2: Viernes

dxabezas@correo.ugr.es

6 de junio de 2021

Índice

1. Descripción de la Metaheurística

Nuestro problema es la optimización continua de una serie de funciones. Estamos limitados por el número de evaluaciones, que en nuestro caso será 10000 multiplicado por la dimensión del espacio de soluciones.

Diseñaremos nuestra propia metaheurística y la pondremos a prueba con el benchmark CEC'2017. Por tanto, nuestro objetivo es la optimización de parámetros reales. Con nuestra propuesta esperamos alcanzar un buen compromiso entre exploración y explotación aprovechando la naturaleza continua del espacio de soluciones.

1.1. Idea general

Nuestra metaheurística, **Búsqueda Ramificada con Momentos**, pretende "lanzar" soluciones que se muevan por el espacio como lo harían bombas lanzadas desde un avión bombardeos, fragmentos que se desprenden de un cuerpo celeste o el despliegue de una sonda de una nave espacial. Es decir, se explora el espacio ramificando o dividiendo las soluciones en otras, de forma que las soluciones resultantes conserven la inercia de la solución de la que partieron.

Figura 1: Cuando se despliegan bombas desde un avión, las bombas mantienen la inercia del movimiento del avión.

Para dar sentido a este fenómeno físico que es la conservación de la inercia ideamos una estrategia de explotación basada en momentos, similar a la Búsqueda Local salvo que se pondera el momento de la solución con la dirección de un vecino mejor.

Las soluciones pueden ser truncadas bajo ciertas condiciones, así que se van lanzando nuevas soluciones desde puntos aleatorios hasta agotar el número máximo de evaluaciones.

1.2. Explicación del algoritmo

El algoritmo comienza con una solución aleatoria, un vector aleatorio de flotantes, S. Cada solución tiene asociado otro vector aleatorio que es su momento, ν, que intuitivamente se puede interpretar como la dirección en la que se está moviendo la solución actualmente; también tiene un valor escalar positivo, λ que podría interpretarse como un impulso, la solución se desplaza usando su momento y su impulso: $S \leftarrow S + \lambda\nu$.

Cada solución tiene un número determinado de evaluaciones. En cada iteración, una solución puede hacer una de las siguientes acciones:

- La solución desaparece, es truncada.
- La solución se divide en dos, que "saldrán disparadas" en direcciones opuestas.

- La solución desvía su dirección hacia un vecino del entorno con mejor fitness y luego se actualiza usando su impulso y su momento: $S \leftarrow S + \lambda\nu$. La solución pierde una cantidad de impulso determinada por el éxito de la búsqueda de un vecino mejor en el entorno.

La acción a realizar en cada momento depende de la fitness de la solución respecto a la mejor fitness encontrada hasta el momento, del número de evaluaciones restantes y del impulso que le quede a la solución. Queremos truncar soluciones que no tengan muchas posibilidades de mejorar, ya sea porque estén muy lejos de la mejor hasta el momento y les queden pocas iteraciones o porque su impulso sea prácticamente nulo. Queremos dividir soluciones que sean de las mejores, tengan bastantes evaluaciones por delante y no acaben de ser lanzadas desde otras, para evitar que haya soluciones muy cercanas entre sí.

Estos objetivos pueden ser muy difíciles de lograr debido al número de factores a tener en cuenta, a continuación presentamos nuestro intento de lograr un equilibrio.

Decisión de la acción

En nuestro caso, primero tomaremos la diferencia entre la fitness de la solución y la mejor fitness encontrada hasta el momento: $D \leftarrow fitness - best$. Como estamos minimizando, será una cantidad no negativa. En principio tenemos $D \in [0, +\infty[$, para normalizar la escala utilizamos una función creciente que identifique este intervalo con el $[0, 1[$, $\hat{D} \leftarrow \dfrac{D}{1+D}$.

Decidimos truncar la solución cuando esta diferencia sea alta y las posibilidades de mejorarla sean escasas, ya sea por que la solución tenga pocas evaluaciones restantes o porque su impulso sea bajo.

Decidimos ramificar la solución en dos cuando esta diferencia sea baja, el impulso esté en un rango determinado (no queremos dividir ni muy pronto ni muy tarde) y el número de evaluaciones restantes sea suficiente para que las soluciones resultantes puedan mejorar.

En el resto de casos, la solución se limita a avanzar.

En nuestra implementación, hemos decidido dar esta estructura al bucle principal, que combina las ideas anteriores con una componente aleatoria.

Algorithm 1: BRANCH: Búsqueda Ramificada: Bucle Principal.

Input: Una solución: vector de flotantes S
Input: Su momento: vector de flotantes μ
Input: Su impulso: escalar positivo λ
Input: Evaluaciones a realizar: entero positivo $evals$
$fitness \leftarrow \text{eval}(S)$ `// Supondremos la función de evaluación siempre disponible`
$evals \leftarrow evals - 1$
$best \leftarrow \min(best, fitness)$
while $evals > 0$ **do**
  $D \leftarrow fitness - best$ `// Suponemos accesible la mejor fitness obtenida hasta el momento`
  $\hat{D} \leftarrow \frac{D}{1+D}$ `// Diferencia normalizada en` $[0, 1[$
  $p \leftarrow U[0, 1]$ `// Flotante aleatorio en` $[0, 1]$`, elegido según la distribución uniforme`
  if $(p < P_{vanish}\frac{\hat{D}}{\lambda}$ ***and*** $evals \leq MaxEvalsTruncate)$ ***or*** $\lambda < MinImpulse$ **then**
    La solución es truncada.
  else if $p > P_{split}\frac{\hat{D}}{\lambda}$ ***and*** $MinImpulseSplit \leq \lambda \leq MaxImpulseSplit$ ***and*** $evals \geq MinEvalsSplit$ **then**
    La solución se divide en dos.
  else
    La solución avanza.

Más adelante discutiremos los parámetros que han aparecido a lo largo del algoritmo. De momento nos centraremos en detallar cada una de las acciones que aparecen en el bucle.

Truncamiento de la solución

Cuando una solución es truncada, simplemente se corta el bucle con la orden **break**. Antes, hay que acumular las evaluaciones a realizar para que no se desperdicien, las llamadas restantes se acumulan para mejorar futuras

soluciones: $spareEvals \leftarrow spareEvals + evals$.

Ramificación de la solución

Cuando una solución se ramifica, se generan dos soluciones con empujes opuestos desde ella y se acumulan los momentos. Además, se recargan en cierta proporción del impulso perdido desde el inicial, λ_0. Las evaluaciones pendientes se reparten equitativamente entre las dos soluciones generadas, como la división es entera, se rescata la evaluación sobrante en caso de que el número de evaluaciones sea impar. La solución es destruida, ya que se queda sin evaluaciones.

Algorithm 2: SPLIT: Ramificación de una solución S como la de entrada del Algoritmo 1.

$modification \leftarrow$ vector aleatorio, las componentes se generan con una $U[0,1]$
$\mu_1 \leftarrow \mu + modification$ `// Las soluciones salen en direcciones opuestas`
$\mu_2 \leftarrow \mu - modification$
$\lambda_1 \leftarrow \lambda + SplitImpulse \cdot (\lambda_0 - \lambda)$ `// Se recarga parcialmente el impulso`
$\lambda_2 \leftarrow \lambda + SplitImpulse \cdot (\lambda_0 - \lambda)$
$S_1 \leftarrow S + \lambda_1\mu_1$
$S_2 \leftarrow S + \lambda_2\mu_2$
clip(S_1) `// Si alguna componente se sale del rango, se fija en el borde`
clip(S_2)
if $evals\,\%2 = 1$ **then**
  $spareEvals \leftarrow spareEvals + 1$ `// Para evitar perder evaluaciones`
branch($S_1, \mu_1, \lambda_1, evals/2$)
branch($S_2, \mu_2, \lambda_2, evals/2$)
break `// Esta solución desaparece`

Avance de la solución

Para el avance de la solución se buscan vecinos aleatorios en un entorno de la solución, para mí siempre se tiene $\lambda \in]0,1]$ y he escogido radio $\sqrt{\lambda}$. Si se encuentra un vecino mejor en un determinado número de intentos, se considera un éxito y se modifica el momento de la solución desviándolo hacia la dirección del vecino, la medida en la que se desvía depende de la mejora que aporte; después se desplaza la solución usando el momento y el impulso y se reduce el impulso ligeramente. Si se fracasa al encontrar un vecino mejor (se agotan los intentos) interpretamos que estamos en un máximo local (o se acabaron las evaluaciones) y se reduce más severamente el impulso.

Algorithm 3: ADVANCE: Avance de la solución S (la de entrada del Algoritmo 1).

while *no se encuentre un vecino mejor* ***and*** *no se excedan ImproveLimit intentos* ***and*** $evals > 0$ **do**
  $modification \leftarrow$ vector aleatorio, las componentes se generan con una $U[-\sqrt{\lambda}, \sqrt{\lambda}]$
  $S' \leftarrow S + modification$ `// Vecino aleatorio`
  $\text{clip}(S')$
  $neighbor\_fitness \leftarrow \text{eval}(S')$
  $evals \leftarrow evals - 1$
if *el vecino* S' *mejora la fitness de* S **then**
  $best \leftarrow \min(best, neighbor\_fitness)$
  $D \leftarrow fitness - neighbor\_fitness$
  $\hat{D} \leftarrow \frac{D}{1+D}$ `// Mejora normalizada en` $[0, 1[$
  $neighbor\_weight \leftarrow BaseWeight + (1 - BaseWeight) \cdot \hat{D}$
  $\mu \leftarrow (1 - neighbor\_weight)\mu + neighbor\_weight \cdot modification$ `// Se desvía la inercia de la solución hacia el vecino mejor`
  $S \leftarrow S + \lambda\mu$ `// La solución se desplaza en su dirección`
  $\text{clip}(S)$
  $fitness \leftarrow \text{eval}(S)$
  $evals \leftarrow evals - 1$
  $best \leftarrow \min(best, fitness)$
  $\lambda \leftarrow DecreaseSuccess \cdot \lambda$ `// Éxito: se reduce ligeramente el impulso`
else
  $\lambda \leftarrow DecreaseFail \cdot \lambda$ `// Fracaso: se reduce severamente el impulso`

Invocación del algoritmo

Para aprovechar las evaluaciones de las soluciones que son truncadas se vuelven a lanzar nuevas soluciones. Cada ejecución parte de una solución aleatoria con momento aleatorio y un impulso inicial λ_0.

Algorithm 4: MAIN: Llamadas sucesivas al algoritmo de búsqueda Ramificada con Momentos hasta consumir todas las evaluaciones disponibles.

$best \leftarrow$ un valor mayor que cualquiera que vaya a aparecer al evaluar posteriormente
$spareEvals \leftarrow 10000 \cdot$ dimensión del espacio de soluciones
while $spareEvals > 0$ **do**
  $evals \leftarrow spareEvals$
  $spareEvals \leftarrow 0$
  $S \leftarrow$ vector solución aleatorio, componentes generadas de forma uniforme en el rango: $U[-100, 100]$
  $\mu \leftarrow$ vector momento aleatorio, componentes generadas con $U[-1, 1]$
  $\text{branch}(S, \mu, \lambda_0, evals)$ `//` $spareEvals$ `puede verse incrementada durante la ejecución`

2. Implementación y benchmark

2.1. Detalles de implementación

Como hemos comentado, nuestra explicación de la metaheurística es una forma de implementar las ideas que hemos ido comentando. Probablemente existan mejores alternativas para tomar las decisiones del Algoritmo 1, que se basen en las mismas ideas intuitivas que hemos comentado. Como otro ejemplo, la división de la solución podría ser en más de dos soluciones.

Además, fijada una implementación concreta (la que acabamos de describir), quedan por discutir un gran número de parámetros que han ido apareciendo a lo largo de la explicación del algoritmo. A continuación presentamos una tabla con los distintos parámetros, su papel en la metaheurística y el valor concreto que les damos en la ejecución.

Nota: Todos los parámetros toman valores positivos por naturaleza.

Parámetro	**Tipo / Rango**	**Papel**	**Valor**
λ_0	Real	Impulso inicial de las soluciones de partida	1
$MaxEvalsTruncate$	Entero menor que $10000 \cdot \dim$	Máximo de evaluaciones restantes con las que una solución puede ser truncada	$1200 \cdot \dim$
$MinImpulse$	Real menor que λ_0	Impulso por debajo del cual la solución es truncada	$0.01 \cdot \lambda_0$
$MinImpulseSplit$	Real menor que λ_0	Mínimo impulso con el que una solución se puede ramificar	$0.1 \cdot \lambda_0$
$MaxImpulseSplit$	Real en $]MinImpulseSplit, \lambda_0[$	Máximo impulso con el que una solución se puede ramificar	$0.7 \cdot \lambda_0$
$MinEvalsSplit$	Entero menor que $10000 \cdot \dim$	Mínimo de evaluaciones restantes para dividir la solución	$400 \cdot \dim$
$SplitImpulse$	Proporción (real en [0,1])	Proporción del impulso perdido que recuperan las soluciones que surgen de una división	0.5
$ImproveLimit$	Entero menor que $10000 \cdot \dim$	Máximo de intentos para encontrar un vecino mejor	$10 \cdot \dim$
$BaseWeight$	Proporción en $[0, 1[$	Mínimo peso en el momento para el vecino que mejora	0.2
$DecreaseSuccess$	Proporción en $]0, 1[$	Reducción del impulso en caso de acierto	0.99
$DecreaseFail$	Proporción en $]0, DecreaseSuccess[$	Reducción del impulso en caso de fallo	0.9

Tabla 1: Tabla con los parámetros del modelo y sus valores concretos en la ejecución.

Basamos estas elecciones únicamente en nuestra intuición y en el comportamiento del algoritmo (número de veces que hace cada acción y en qué orden, y valores de las variables, no en los resultados) sobre la función 4 del benchmark en dimensión 10. De modo que seguramente habrá mejores valores para estos parámetros.