Modelos Bioinspirados y Heurísticas de Búsqueda

AAD

“Algoritmos de Optimización Basados en Nube de Partículas”

Antonio José Morano Moriña

Índice

1. Funciones a minimizar

El objetivo es buscar el mínimo de las siguientes funciones:

Rosenbrok,

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2.$$

```
def fitnessRosenbrok(pos):
  x = pos[0]
  y = pos[1]

  valor = ((1-x)**2) + 100*((y-(x**2))**2)
  return valor
```

Rastrigin,

$$Ras(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$

```
def fitnessRastrigin(pos):
  x = pos[0]
  y = pos[1]

  valor = 20 + x**2 + y**2  - 10*(math.cos(2*math.pi*x) + math.cos(2*math.pi*y))
  return valor
```

Cada una de las funciones tiene un diferente espacio de búsqueda, y los limites en los que se van a mover las partículas son:

Rosenbrok,

Los valores en los que estaba representada la función en la imagen del enunciado.

Rastrigin,

The function is usually evaluated on the hypercube $x_i \in [-5.12, 5.12]$, for all i = 1, …, d.

```
#Limites espacios bus
#Rosenbrock
minX_f0 = -2
maxX_f0 = 2
minY_f0 = -0.5
maxY_f0 = 3
#Rastrigin
minX_f1 = -5.12
maxX_f1 = 5.12
minY_f1 = -5.12
maxY_f1 = 5.12
#Para Busqueda local
minX_BL = -10
maxX_BL = 10
minY_BL = -10
maxY_BL = 10
```

2.Funciones auxiliares.

Estas son las funciones implementadas, con la finalidad de poder seguir el pseudocódigo de los algoritmos requeridos en la práctica.

Inicialización aleatoria de partículas:

```
def generaParticula(minX,maxX,minY,maxY,minV,maxV):

  xPos = rnd.uniform(minX,maxX)
  yPos = rnd.uniform(minY,maxY)
  xV = rnd.uniform(minV,maxV)
  yV = rnd.uniform(minV,maxV)

  return (xPos,yPos) , (xV,yV)
```

Como se indica en el enunciado, generamos las partículas de forma aleatoria, para ello las inicializamos con valores aleatorios que estén en el rango del problema y devolvemos ambas variables.

Actualizar la velocidad

Ambos algoritmos PSO son modelos completos, ya que un modelo es completo si:

$$\varphi_1, \varphi_2 > 0.$$

Y en este caso tenemos ambas phis = 1.49445.

Por lo que, en las ecuaciones empleadas para determinar la velocidad de una partícula, usaremos tanto la inercia, como el componente cognitivo y el social.

$$v_{id} = \omega \cdot v_{id} + \varphi_1 \cdot rnd() \cdot (pBest_{id} - x_{id}) + \varphi_2 \cdot rnd() \cdot (g_{id} - x_{id})$$

Quedando la implementación de la formula en código de la siguiente manera:

```
def actualizarVelocidad(v,gBest,pBest,pos):

  r1 = rnd.uniform(0,1)*phis
  r2 = rnd.uniform(0,1)*phis

  inercia = (v[0]*w, v[1]*w)
  cognitivo = (r1*(pBest[0]-pos[0]), r1*(pBest[1]-pos[1]))
  social = (r2*(gBest[0]-pos[0]), r2*(gBest[1]-pos[1]))

  formulaV = (inercia[0]+cognitivo[0]+social[0], inercia[1]+cognitivo[1]+social[1])

  return formulaV
```

Limitación del espacio de búsqueda:

```
def comprobarPos(pos,v, minX, maxX, minY, maxY):
  x = pos[0]
  y = pos[1]
  vComprobada = np.copy(v)
  if(x < minX):
    x = minX
    vComprobada[0] = -vComprobada[0]
  if(x > maxX):
    x = maxX
    vComprobada[0] = -vComprobada[0]
  if(y < minY):
    y = minY
    vComprobada[1] = -vComprobada[0]
  if(y > maxY):
    y = maxY
    vComprobada[1] = -vComprobada[0]
  posNueva = (x,y)
  return posNueva,vComprobada
```

Esta función se encarga de comprobar que la partícula no se salga del espacio valido, haciendo que en caso de que llegue al límite, rebote en este con una velocidad opuesta a la que choca

Actualizar Posición:

```
def actualizarPosicion(pos, v, minX, maxX, minY, maxY):
  nuevaPos = (pos[0] + v[0] , pos[1]+v[1])
  posDevolver,vComprobada = comprobarPos(nuevaPos,v, minX, maxX, minY, maxY)

  return posDevolver, vComprobada
```

Generación de vecinos:

```
def generaVecinos(pos, nVecinos, variacion, minX,maxX,minY,maxY):
  vecinos = []

  for i in range(nVecinos):
    varX = rnd.uniform(-variacion,variacion)
    varY = rnd.uniform(-variacion,variacion)

    posVecino = (pos[0]+varX,pos[1]+varY)
    vecino, _ = comprobarPos(posVecino,(0,0),minX,maxX,minY,maxY)
    vecinos.append(vecino)

  return vecinos
```

A la hora de generar los vecinos he seleccionado vecinos que se encuentren a una distancia aleatoria comprendida entre +-0.1, así explora el espacio en diferentes direcciones que si se crearan siempre los vecinos que estén en:

(pos[] + 0.1 // pos[]-0.1 // pos[0] + 0.1 , pos[1] -0.1 // pos[0] – 0.1 , pos [1] + 0.1)

3.Algoritmos

La implementación de los algoritmos se basa en seguir los pseudocodigos disponibles en las diapositivas de teoría y para no alargar excesivamente con imágenes del código la memoria, estos mismo se encuentran comentados con los pasos seguidos en el código adjunto.

Linea 103 - PSO Local

Linea 152 - PSO Global

Linea 194 - BusquedaLocal

4.Parámetros seleccionados

Los valores de los parámetros usados en la práctica son los siguientes:

Dados en el enunciado:

```
w = 0.729
phis = 1.49445
nParticulas = 10
vecindadPSOLocal = 2
```

Seleccionados mediante experimentación:

maxIters = 10.000

He decidido dejarlo en 100.000 iteraciones debido a que a veces se estanca en encontrar el mínimo global y da igual cuantas iteraciones se ejecute que no sale del mínimo local en el que se encuentra.

5.Resultados experimentación

Para realizar la experimentación se han seleccionado las 5 mismas semillas que en la experimentación de la practica anterior: 10,11,15,20,50

Resultados para semilla 10:

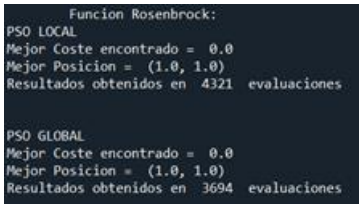
```
       Funcion Rosenbrock:
PSO LOCAL
Mejor Coste encontrado =  0.0
Mejor Posicion =  (1.0, 1.0)
Resultados obtenidos en  3768  evaluaciones


PSO GLOBAL
Mejor Coste encontrado =  0.0
Mejor Posicion =  (1.0, 1.0)
Resultados obtenidos en  3967  evaluaciones
```

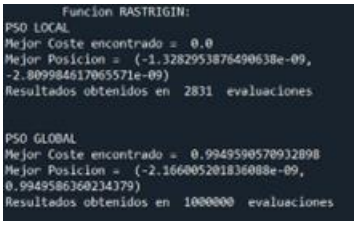
```
       Funcion RASTRIGIN:
PSO LOCAL
Mejor Coste encontrado =  0.0
Mejor Posicion =  (-2.834898193260052e-10,
-3.855718119366349e-10)
Resultados obtenidos en  2257  evaluaciones


PSO GLOBAL
Mejor Coste encontrado =  0.0
Mejor Posicion =  (-2.003674514087157e-10,
-2.0232167424268254e-09)
Resultados obtenidos en  1784  evaluaciones
```

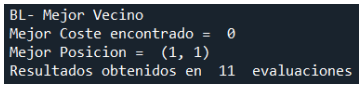
```
BL- Mejor Vecino
Mejor Coste encontrado =  0
Mejor Posicion =  (1, 1)
Resultados obtenidos en  11  evaluaciones
```

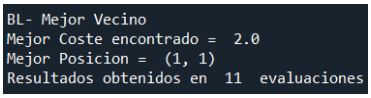
```
BL- Mejor Vecino
Mejor Coste encontrado =  1.9974692389341797
Mejor Posicion =  (0.9989688614022061, 0.9996479976059666)
Resultados obtenidos en  22  evaluaciones
```

Resultados para semilla 11:

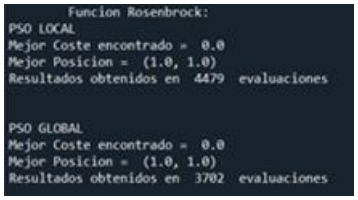


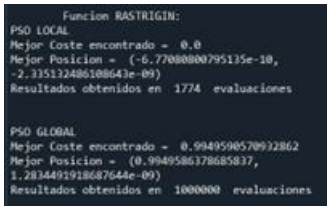


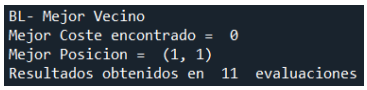


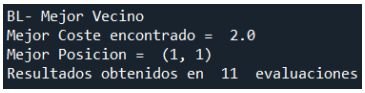


Resultados para semilla 15:

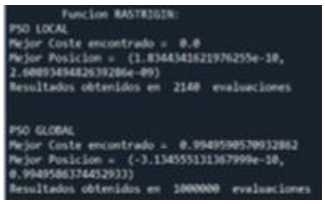


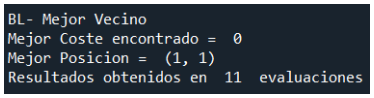


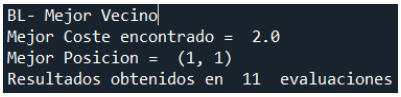




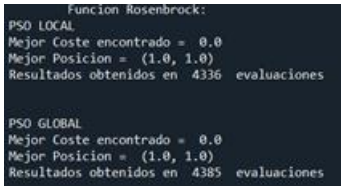
Resultados para semilla 20:

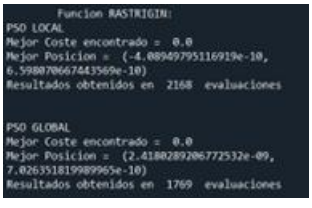


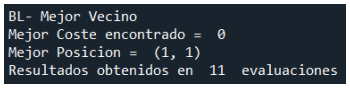




Resultados para semilla 50:







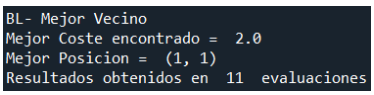


Tabla para la experimentación de la función ROSENBROCK

Algoritmo	Ev.Medias	Ev.Mejor	Ev.Desv	Mejor valor	Media valor	Desv. valor
PSO Local	4366	3768	415,251	0	0	-
PSO Global	5149	3694	2725,979	0	0,008	0,018
BL	11	11	-	0	0	-

Tabla para la experimentación de la función RASTRIGIN

Algoritmo	Ev.Medias	Ev.Mejor	Ev.Desv	Mejor valor	Media valor	Desv. valor
PSO Local	2234	1774	381,356	0	0	-
PSO Global	5065	1748	4504,656	0	0,398	0,545
BL	13,2	11	4,919	1,99	1,998	0,004

6.Análisis de resultados obtenidos

PSO Local y PSO Global

Los resultados obtenidos para ambas funciones por parte del PSO Local son bastante sólidos, El PSO Local converge un poco más rápido que el PSO Global, aunque el PSO Local debería de caer en mínimos locales con mayor facilidad que el global, pero si comparamos los resultados vemos que en esta experimentación (no sé si se debe a las semillas seleccionadas o a otro factor aleatorio del problema) no es así.

Desde el punto de vista computacional, el coste de PSO Global debería de ser menor también, pero debido a que en estas iteraciones cae varias veces en mínimos locales (3 de 10 iteraciones, si contamos el total para ambas funciones) el coste de evaluaciones realizadas se incrementa bastante, debido a que cada vez que el algoritmo se estanca en uno de estos mínimos locales, realiza 10.000 iteraciones, cuando de normal ambos algoritmos rondan las 4.000-5.000 iteraciones cuando encuentran el mínimo global de la función. Esto hace que la desviación típica de las evaluaciones del algoritmo se vea muy incrementadas, dando lugar a un algoritmo que no es muy robusto.

Búsqueda Local

En cuanto a la búsqueda local, los resultados tan buenos y el algoritmo tan sólido, son engañosos ya que el punto de inicio de estos algoritmos es el (1,1), que está muy cercano a los mínimos globales, por eso solamente con 11 iteraciones es capaz de encontrar el coste 0.

Para apoyar esta afirmación he decidido realizar 5 iteraciones más donde la posición inicial de la búsqueda local será aleatoria, y comparar resultados con las anteriores.

Semilla 10:

Rosenbrock

```
BL- Mejor Vecino
Mejor Coste encontrado =  0.5306372272083333
Mejor Posicion =  (0.28102513311075134, 0.06726514337762333)
Resultados obtenidos en  1650  evaluaciones
```

Rastrigin

```
BL- Mejor Vecino
Mejor Coste encontrado =  20.17392781101924
Mejor Posicion =  (1.9544986232661548, 3.9676812673398887)
Resultados obtenidos en  22  evaluaciones
```

Semilla 50:

Rosenbrock

```
BL- Mejor Vecino
Mejor Coste encontrado =  16.218355145182443
Mejor Posicion =  (-3.0232512241655556, 9.157881845777561)
Resultados obtenidos en  143  evaluaciones
```

Rastrigin

```
BL- Mejor Vecino
Mejor Coste encontrado =  5.579890009833843
Mejor Posicion =  (-1.0326481064495898, -1.9492381883062444)
Resultados obtenidos en  88  evaluaciones
```

Resultados búsqueda local empezando en una posición aleatoria

Algoritmo	**Ev.Medias**	**Ev.Mejor**	**Ev.Desv**	**Mejor valor**	**Media valor**	**Desv. valor**
BL_Rosenbrock	770	88	644,867	0,5	9,728	7,068
BL_Rastrigin	59,4	22	34,435	3,2	11,4	8,281

Observando los nuevos resultados de la tabla, podemos ver como si la posición es aleatoria el algoritmo de búsqueda local, encuentra soluciones en una pequeña cantidad de evaluaciones y con muy poca desviación típica, pero se podría decir que aunque sea un algoritmo robusto, encuentra soluciones de costes muy elevados para los mínimos globales de cada una de las funciones.

7.Bibliografía científica

Jorge, H. C. & Jorge Andrés, A. S. (2023). Problemas de optimización en logística: una propuesta de solución aplicando el algoritmo Particle Swarm Optimization (PSO). *Escolme, 1(1) , 2-14.*

-En este trabajo se aplica el algoritmo PSO para solucionar problemas asociados a los sistemas logísticos en las organizaciones, con un énfasis en la optimización para el abastecimiento y transporte dentro de las empresas

Henry, L. D. & Silvia Adriana, G. N. & Julián, G. & Camilo, C. (2013). Algoritmo PSO-Híbrido para solucionar el problema de ruteo de vehículos con entrega y recolección simultáneas. *Dialnet*, 35(22) , 75-90.

-Este trabajo aplica el algoritmo PSO para resolver el problema VRPSPD, el cual se basa en encontrar un conjunto de rutas para una flota de camiones, sabiendo las demandas de entregas y recolección de cada uno de los clientes por los que debe pasar la ruta, buscando minimizar el coste de ese conjunto de rutas.

Jun, Y. & Lifu, H. & Siyao, F. (2014). An improved PSO-based charging strategy of electric vehicles in electrical distribution grid. *ScienceDirect*, 128(1), 82-92.

-En este trabajo se aplica el algoritmo PSO para optimizar tanto la carga de las baterías de los coches eléctricos, como su longevidad y el coste económico de los materiales necesarios para crear estas baterías mejoradas.