Cómputo Evolutivo
Proyecto final

Adrián García Pérez
315131224

Especificación del problema

Para este proyecto se buscó realizar una implementación de la heurística *Particle Swarm Optimization* (PSO). Se trata de una heurística que busca optimizar la función de costo de un problema, esto iterando múltiples soluciones en el espacio de búsqueda hasta que se cumpla un criterio de término.

Se caracteriza por definir un enjambre (*swarm*) formado por partículas (*particles*), las cuáles tienen propiedades como *posición* y *velocidad*, las cuales cambian en cada iteración.

De esta manera, la heurística PSO nos permite encontrar buenas soluciones que hagan que la función objetivo se minimice o maximice (según dependa el problema), esto en un tiempo computacionalmente corto.

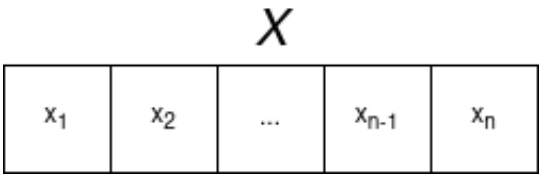
Función objetivo

Para este proyecto, se eligieron diversas funciones objetivo que cuentan con múltiples puntos mínimos, esto para observar el comportamiento de la heurística en dos dimensiones y poder entender mejor el comportamiento del enjambre al hacer cambios en los parámetros.

Por lo tanto, se considera $f \in \mathbb{R}^n \to \mathbb{R}$ como la función de costo que busquemos minimizar, la cuál tomará como argumento una solución en la forma de un vector de números reales x, y como resultado un número real cuyo valor dirá que las soluciones son buenas mientras sea más pequeño.

Representación de soluciones

De esta manera, las partículas del enjambre contarán cada una con un vector x, donde cada entrada del vector es un número real. La dimensión del tamaño del espacio búsqueda de cada función definirá la cantidad de entradas del vector, por lo que si a función está definida sobre $\mathbb{R}^n$, x será un vector de n entradas cada una con un valor real.



Implementación

En primer lugar, veamos el pseudocódigo de la heurística:

Algorithm 1: PSO

Input: Swarm: S, Double: $b_{lo}, b_{up}, \omega, \varphi_p, \varphi_g, lr$, List: $dims$, f: $\in \mathbb{R}^n \to \mathbb{R}$
Output: Particle: *best_solution*

```
1  for each p ∈ S do
2      p.x ~ U(b_lo, b_up)
3      p.v ~ U(b_lo, b_up)
4      p.b ← p.x
5      if f(p.b) < f(g) then
6          g ← p.b
7  best_solution ← None
8  while termination_criterion() == false do
9      for each p ∈ S do
10         for each d ∈ dims do
11             r_p ~ U(0, 1)
12             r_g ~ U(0, 1)
13             inertia ← ω · p.v_d
14             pFact ← φ_p · r_p · (p.b_d − p.x_d)
15             gFact ← φ_g · r_g · (g_d − p.x_d)
16             p.v_d ← inertia + pFact + gFact
17         p.x ← p.x + lr · p.v
18         if f(p.x) < f(p.b) then
19             p.b ← p.x
20             if f(p.b) < f(g) then
21                 g ← p.b
22                 best_solution ← p
23 return best_solution
```

Se considera como entrada un conjunto de partículas S, cada una con una variable x y v, siendo los vectores de posición y velocidad correspondientes. En las líneas 1 a 6, se inicializan ambas variables con valores aleatorios que siguen una distribución uniforme, como límite inferior y superior los límites de la función f respectivos. En la línea 4 se dice que la mejor posición de la partícula ($p.b$) es igual a la posición inicial de la partícula. Finalmente, si el evaluar la función en dicho vector de posición da como resultado algo menor a evaluar la función en el vector g (la mejor posición global del enjambre), entonces se hace dicha variable g igual a $p.b$

En la línea 7, se inicializa la variable *best_solution* como $None$. Las líneas 8 a 22, se ejecutan mientras el criterio de termino no sea verdadero. De esta manera, para cada partícula y cada número $d \in dims$ (con $dims$ siendo la enumeración de la dimensión de la función f, es decir, $[1, 2, ..., n]$), se obtienen dos variables con valor aleatorio en [0, 1] y se calculan los valores que modifican la velocidad de la partícula. La inercia, igual al coeficiente ω por la velocidad actual en la posición d. El factor p, igual al coeficiente φ_p por r_p por la diferencia en velocidad de la partícula actual y su mejor posición conocida. El factor g, igual al coeficiente φ_g por r_g por la diferencia en velocidad de la partícula actual y la mejor posición del enjambre conocida.

Después se recalcula la nueva posición de la partícula, siendo igual al vector de la posición actual más la nueva velocidad calculada previamente por el termino lr conocido como *learning rate*.

Posteriormente si la posición actual tiene un mejor valor (al evaluarse en la función) que la mejor posición conocida de la partícula, esta última se actualiza, y de la misma forma para la mejor posición del enjambre conocida y la mejor solución conocida, que finalmente es retornada.

Experimentación

Como vimos en la implementación, existen diversos parámetros de la heurística de los cuales sus valores modificaran el comportamiento de la misma fuertemente. Algunos valores se encuentran definidos estrictamente por la función a optimizar f, es decir, que no se consideran como parámetros con los que se pueda realizar experimentación, sino que se encuentran ya definidos. Estos parámetros son b_{lo} y b_{up}, las cotas que definen al espacio de búsqueda de la función f, es decir, donde consideramos definida a la función.

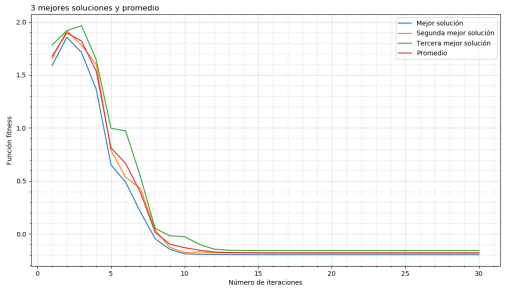
En este proyecto se decidió experimentar con los valores ω y lr, dejando el resto con valores específicos:

- Número de evaluaciones: 30
- Tamaño de la población: 50
- Dimensión: 2
- φ_p: 0.3
- φ_g: 0.7

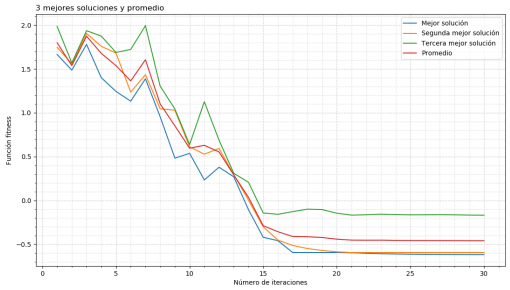
De esta manera, se realizaron 20 repeticiones de cada configuración de parámetros, dando como resultado la siguiente tabla:

Función	Learning rate	ω	Mejor valor	Valor promedio	Peor valor	Desviación std.
Sphere	0.5	0.5	0.0001	0.098	0.608	0.132
Sphere	0.5	0.7	$2.18\text{x}10^{-5}$	0.030	0.166	0.043
Sphere	0.7	0.5	0.0001	0.093	0.622	0.147
Sphere	0.7	0.7	$2.93\text{x}10^{-5}$	0.0117	0.064	0.017
Cosine	0.5	0.5	-0.999	-0.994	-0.9611	0.008
Cosine	0.5	0.7	-0.999	-0.9995	-0.998	0.0004
Cosine	0.7	0.5	-0.999	-0.993	-0.968	0.008
Cosine	0.7	0.7	-0.999	-0.9992	-0.986	0.002
A funct	0.5	0.5	-1.999	-1.177	0.608	0.782
A funct	0.5	0.7	-1.999	-1.193	0.166	0.758
A funct	0.7	0.5	-1.999	-1.179	0.622	0.781
A funct	0.7	0.7	-1.999	-1.197	0.064	0.752
B funct	0.5	0.5	-1.999	-1.177	0.608	0.782
B funct	0.5	0.7	-1.999	-1.193	0.166	0.758
B funct	0.7	0.5	-1.999	-1.179	0.622	0.781
B funct	0.7	0.7	-1.999	-1.197	0.064	0.752
C funct	0.5	0.5	-1.999	-1.177	0.608	0.782
C funct	0.5	0.7	-1.999	-1.193	0.166	0.758
C funct	0.7	0.5	-1.999	-1.179	0.622	0.781
C funct	0.7	0.7	-1.999	-1.197	0.064	0.752

De la misma manera, se hicieron gráficas para comparar las mejores tres soluciones en cada ejecución, así como el valor promedio fitness de todo el enjambre, por ejemplo:

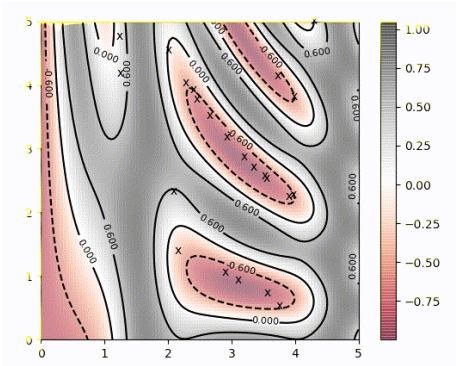


(a) LR = 0.7, $\omega = 0.\ 5$



(b) LR = 0.7, $\omega = 0.\ 7$

Figure 1: Comportamiento para C Function



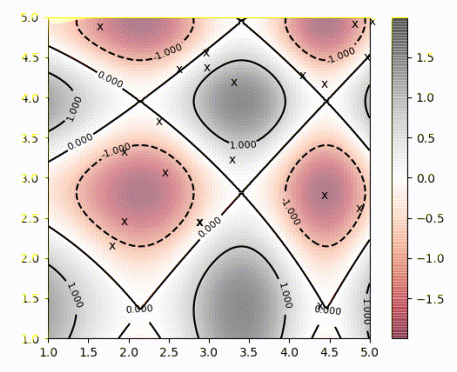


Figure 2: Ejemplos de las animaciones

Aquí es más sencillo notar el comportamiento del algoritmo al elegir diferentes valores para ω, ya que parece que el algoritmo converge más rápido cuando los valores de ω son menores, sin importar demasiado el valor del *learning rate*. Esto parece razonable debido a que un valor pequeño de inercia ω provoca que las partículas no tengan movimientos tan bruscos (lo cual puede observarse en las animaciones para valores con $\omega = 0.5$ a comparación de $\omega = 0.7$).

Al desarrollar este proyecto, tuve mayor interés por mostrar el comportamiento del algoritmo utilizando animaciones en un espacio de tres dimensiones, donde las partículas mostraran la velocidad y su posición en tiempo real. Dichas simulaciones se encuentran en la carpeta `t7/figs/` del repositorio de Github. Así mismo, en la carpeta `t7/ejecuciones/` se pueden encontrar el resto de gráficas que muestran el comportamiento de las ejecuciones.

Con los resultados de los experimentos, podemos notar que las soluciones en general no difieren de manera significativa entre ellas, y se obtienen buenos valores aún con pocas iteraciones del algoritmo. Además, vemos que los resultados se ven fuertemente guiados por los valores de la inercia, ya que al tener un valor más cercano a 1, las partículas tienden a conservar más su velocidad y eso genera que se alejen de manera más brusca a puntos en específico, lo que provoca que el algoritmo converja de forma más lenta. Esto fuera de ser necesariamente malo, puede evitar que nos quedemos atrapados en puntos mínimos locales, y encontrar mejores soluciones a lo largo del espacio de búsqueda.

Comentarios y conclusiones

Me parece que los resultados permiten entender mejor cómo se comporta la heurística aún haciendo cambios muy pequeños en los parámetros, algo que en mi opinión resulta útil didácticamente hablando. Como posible trabajo futuro, pienso que el encontrar mejores instancias para la experimentación daría resultados más interesantes, como el requerir más iteraciones para encontrar mejores soluciones o que las mejores soluciones se encuentren bastante "escondidas" entre soluciones peores.

Además, algo que podría dotar a la heurística de un comportamiento más dinámico, sería el de implementar una topología diferente para el enjambre, de manera que no todas las partículas se encuentren conectadas unas con otras. Esto podría provocar que el comportamiento tenga un mayor parecido a lo observado en la naturaleza, donde algunos seres vivos tienden a moverse por grupos pequeños.

Referencias

- Kennedy, J. (1997). "The particle swarm: social adaptation of knowledge"
- Kennedy, J.; Eberhart, R. (1995). "Particle Swarm Optimization"
- van den Bergh, F. (2001) "An Analysis of Particle Swarm Optimizers".