UNIVERSIDAD DE GRANADA

Metaheurísticas

Proyecto Final

Metaheurística:
FireFly Algorithm

Curso: 2020-2021
Autor: Javier Ramírez Pulido – 20501292N
3ºCSI – Grupo 2 (Viernes 17:30 – 19:30**)**
mailto:javierramirezp@correo.ugr.es

INDICE

PSEUDOCODIGOS

Introducción

La mayoría de los problemas de optimización en ingeniería son no lineales con muchas restricciones, por lo que encontrar soluciones óptimas a estos requiere algoritmos de optimización eficientes. Estos pueden ser deterministas (totalmente predictivos si se conocen sus entradas, conociendo la salida siempre que se den las mismas entradas) o estocásticos (funciona por azar, por lo que no siempre tiene que llegar a los mismos resultados partiendo de las mismas entradas).

Estos últimos normalmente tienen un componente determinista y otro aleatorio, y algunos son considerados metaheurísticas, como los algoritmos genéticos. Muchas de las metaheurísticas actuales basan su desarrollo en la inteligencia de enjambre de la naturaleza, lo cual ha atraído críticas en la comunidad de investigadores por enmascarar la falta de novedad con metáforas. Especialmente el algoritmo de luciérnagas recibe la crítica de carecer de una fundamentación matemática rigurosa y de diferir de la optimización del enjambre de partículas establecida.

La principal ventaja del Algoritmo de la Luciérnaga frente a otros algoritmos bio-inspirados es su capacidad de dividir la población de soluciones en forma automática y de enfrentarse a problemas multimodales.

Algoritmo de Luciérnagas. Artículo original

Este es un algoritmo bio-inspirado desarrollado por Yang en 2008 y se basa en el comportamiento idealizado de la característica parpadeante de las luciérnagas. Una forma simple de ver el funcionamiento de esta característica sería:

- Las luciérnagas son unisex, por lo que una de estas puede verse atraída por cualquier otra.
- La atracción es proporcional a su brillo, por lo que para dos luciérnagas cualquieras, la menos brillante se moverá en dirección de la más brillante. Cuanto más lejos esté una luciérnaga, menor será su atractivo y si ninguna es más brillante que una, esta se mueve aleatoriamente.
- La intensidad lumínica de cada luciérnaga se obtiene del valor de la función objetivo.

En este algoritmo tenemos dos conceptos importantes: la variación de la intensidad lumínica y la formulación del atractivo. Por simplicidad asumiremos que el atractivo de una luciérnaga viene determinado por su brillo que, a su vez, viene dado por la función objetivo, pero esta debe ser relativa dependiendo de la luciérnaga que la observe. Por ello, esta variará con la distancia $r_{i,j}$ entre la luciérnaga i y la luciérnaga j, a la vez que también lo hace según el grado de absorción del medio en el que se encuentren.

En la forma más simple, la intensidad de la luz a una distancia determinada (*I(r)*) variará monótonamente, siendo

$$I = I_0 e^{-\gamma r}$$

donde I_0 es la intensidad original y γ es el coeficiente de absorción de luz. Ahora ya podemos definir el atractivo de una luciérnaga (β) como

$$\beta = \beta_0 e^{-\gamma r^2}$$

donde β_0 representa el atractivo con distancia 0.

Una forma de ver todo lo explicado hasta ahora con un pseudocódigo general sería:

```
Begin
    1) Objective function: f(x), x = (x1, x2, …, xd);
    2) Generate an initial population of fireflies xi (i = 1, 2,
…, n);
    3) Formulate light intensity I so that it is associated with
f(x)

    4) Define absorption coefficient γ

    While (t < MaxGeneration)
        for i = 1 : n (all n fireflies)
            for j = 1 : i (n fireflies)
                if (Ij > Ii),
                    Vary attractiveness with distance r via exp(-γr);
                    move firefly i towards j;
                    Evaluate new solutions and update light intensity;
                end if
            end for j
        end for i
        Rank fireflies and find the current best;
    end while

    Post-processing the results and visualization;

end
```

La distancia entre dos luciérnagas i y j puede ser la distancia cartesiana o la norma (que es la norma de un vector definida por un vector complejo).

El movimiento de una luciérnaga i atraída por otra más brillante j viene determinado por

$$x_i = x_i + \beta_0 e^{-\gamma r(i,j)^2} (x_i - x_j) + \alpha(\varepsilon_i)$$

donde el segundo término es sobre el atractivo y el tercero es una aleatorización usando la distribución Gaussiana.

Podemos considerar $\beta_0 = 1$, $\alpha \in [0, 1]$ y $\gamma = 1$.

<u>Ventajas FFA</u>

- Puede lidiar con problemas de optimización no lineales de forma natural y eficiente.
- No usa velocidades, lo cual le evita problemas que, algoritmos como PSO tienen.
- La velocidad de la convergencia es muy alta en probabilidad de encontrar la respuesta global optimizada.
- Tiene la flexibilidad de integrarse con otras técnicas de optimización para formar herramientas híbridas.
- No requiere una buena solución inicial para iniciar su proceso de iteración.

Implementación FFA

A pesar de contar con el pseudocódigo anterior, a continuación, veremos el contenido de mi programa esquematizado:

algoritmo FFA

variables como parámetros

tipo entero dim

variables locales

tipo entero constante poblacion_inicial <- 50, cantidad_excesiva_elementos <- 100

vector tipo Luciernaga población

tipo entero evaluaciones <- cantidad_excesiva_elementos

inicio

1 poblacion <- crea_poblacion(poblacion_inicial, cantidad_excesiva_elementos, dim, generador aleat.)
2 Mientras evaluaciones sea menor que 10000 x dim
    a. Mover_luciernaga(población, dim, evaluaciones)

algoritmo mover_luciernaga

variables como parámetros

vector tipo Luciernaga &población

tipo entero dim, &evaluaciones

variables locales

tipo double Alpha <- 0.5, betamin <- 0.2, gama <- 1.0, escala <- 200

tipo booleano mejor

inicio

1. Para cada elemento el1 de la población
    a. Mejor <- Verdadero
    b. Para cada elemento el2 de la población
        i. Si el1 es igual que el2
            1. Pasamos al siguiente el2
        ii. (double) distancia <- 0
        iii. Para cada elemento el3 desde 0 hasta dim
            1. distancia <- distancia + (el1.solucion(el3) - el2.solucion(el3))^2
        iv. distancia <- raíz cuadrada(distancia)
        v. Si el2 tiene un menor fitness que el1
            1. Mejor <- Falso
            2. (double) beta <- (1-betamin)*exp(-gamma * distancia^2) + betamin
            3. Para cada elemento el3 desde 0 hasta dim
                a. (double) aleatorización <- Alpha * (distancia – 0.5) * escala
                b. el1.solucion(el3) <- el1.solucion(el3)*(1-beta) + el2.solucion(el3) * beta + aleatorización
            4. Para cada elemento el3 desde 0 hasta dim
                a. Si el1.solucion(el3) está fuera del rango[-100,100]
                    i. Le asignamos el límite por el que se ha excedido
            5. Actualizamos el fitness de el1
            6. evaluaciones <- evaluaciones + 1
    c. Si es el mejor
        i. Para cada elemento el3 desde 0 hasta dim
            1. Generamos aleatorio en [-10,10]
            2. Sumamos el valor aleatorio al que ya había en esa posicion del mejor
            3. Corregimos si esta fuera del rango de [-100, 100]

Para la implementación se genera una población como se explica en su apartado y directamente se llama a la función encargada de hacer todos los movimientos hasta superar el número de evaluaciones. Las evaluaciones se incrementan cada vez que el fitness de un elemento es calculado, por lo tanto, se hace en la generación y cada vez que se mueva este hacia otro más brillante.

El brillo de cada luciérnaga se considera directamente como el coste de su solución asociada y, como el objetivo es minimizar el coste, tendremos que acercar las luciérnagas a costes bajos. Esto trasladado al ejemplo de las luciérnagas puede sonar contradictorio, pues en realidad la mejor de estas será la que menos brillo tenga y, con esto, será la más atractiva a ojos del resto de luciérnagas.

Sobre la distancia, esta se calcula como la distancia cartesiana de los vectores de las dos luciérnagas implicadas.

$$r_{ij} = ||x_i - x_j|| = \sqrt{\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2}$$

Un ejemplo que nos ayude a entender el procedimiento seguido es el siguiente:

Supongamos una dimensión 'dim' de 10, una luciérnaga L1 con 'solucion' (25, 84, 1, -45, -65, 32, -83, 12, 44, 10) y otra luciérnaga L2 con 'solucion' (19, -3, 32, 67, 94, -35, 11, -81, 53, 9).

La distancia entre estas es:

(25, 84, 1, -45, -65, 32, -83, 12, 44, 10)

\- (19, -3, 32, 67, 94, -35, 11, -81, 53, 9)

(6, 87, -31, -112,-159, 67, -94, 93, -9, 1)

$(6^2, 87^2, (-31)^2, (-112)^2, (-159)^2, 67^2, (-94)^2, 93^2, (-9)^2, 1^2)$

36 + 7569 + 961 + 12544 + 25281 + 4489 + 8836 + 8649 + 81 + 1 = 68447

$\sqrt{68447}$ = 261.62

L1 y L2 estarían a una distancia de 261.62 unidades.

Por último, para el atractivo de una luciérnaga necesitamos la intensidad lumínica de esta ($I = I_0 e^{-\gamma r}$) y el uso de la fórmula $\beta = \beta_0 e^{-\gamma r^2}$, haciendo obligatoria la declaración de una variable

para gamma (coeficiente de absorción de luz del medio en el que se encuentren) y otra $beta_0$ (el valor de atracción de una luciérnaga a distancia 0). Ambos tienen un valor por defecto de 1, pero a raíz de una investigación entre experimentos previamente realizados he encontrado la recomendación de contar con γ = 1 y β = 0.2. Estos valores se mantendrán para el caso en el que se aplique hibridación y alguna otra mejora.

Con todo esto ya podemos calcular cómo se realizará el movimiento entre luciérnagas para nuestro programa. Su fórmula es:

$$x_i = x_i + \beta_0 e^{-\gamma r(i,j)^2} (x_i - x_j) + \alpha(\varepsilon_i)$$

para los que ya tenemos los primeros sumandos, pero nos falta definir el último. Este es una aleatorización para la que hace falta un Alpha, que por defecto podría usarse como 1. Para mí, y sacado del mismo experimento que la recomendación anterior, el valor será de 0.5.

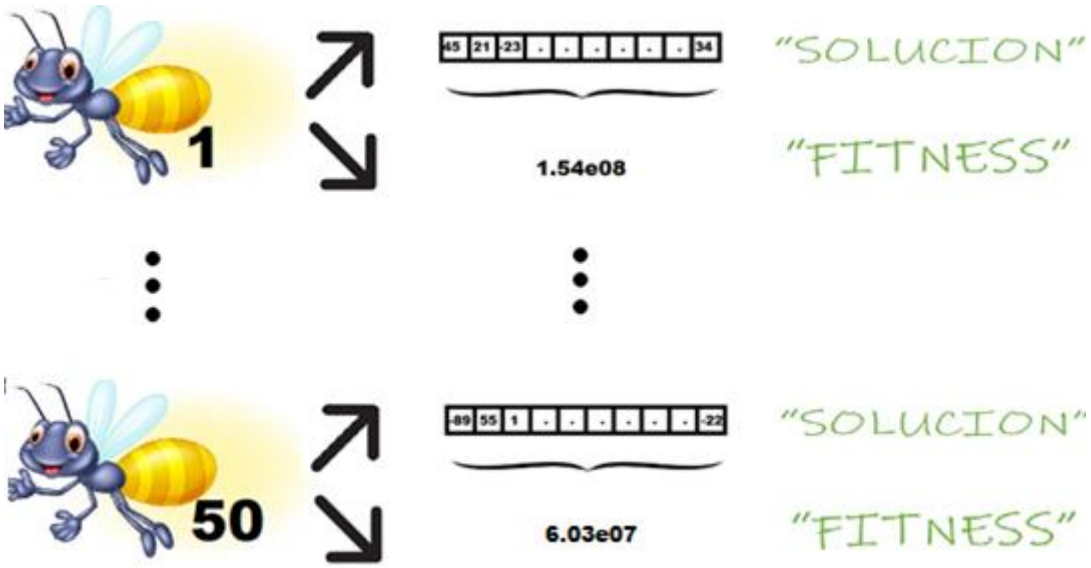
Estos parámetros han sido testeados y comparados superficialmente con los que usaríamos por defecto y hemos comprobado que para nuestro caso los resultados son ligeramente mejores, pero similares.

Esquema de representación de soluciones

Para la representación de soluciones del problema crearemos un struct llamado Luciérnaga que contendrá un vector de tipo double 'solucion' y un atributo double 'fitness'.

El primero de estos será un vector que contendrá *dim* elementos en su interior, todos ellos comprendidos en el rango [-100, 100]. Los valores de *dim* con los que trabajaremos en esta práctica son 10, 30 y 50. Esta será la configuración interna de la que dependerá el valor del segundo atributo, pues a raíz de esta, con la función de fitness de CEC2017, obtendremos el valor de la función objetivo para esa solución.

De esta forma, para el problema tendremos un vector de 50 objetos de tipo Luciérnaga (siguiendo el modelo genético de la práctica 2) generados aleatoriamente. Estos se irán modificando persiguiendo el mejor valor posible y el mejor elemento de la población en el momento en el que las evaluaciones superan el máximo será seleccionado como solución final a nuestro problema.



Generación de la solución inicial

Para la generación de una población inicial contamos con dos funciones. Una de ellas tiene el objetivo de crear una Luciérnaga con tantos valores aleatorios de la distribución normal entre [-100, 100] como determine el valor de 'dim'. En nuestro problema, para el uso de CEC2017, se harán 30 ejecuciones para cada valor de 'dim' que usemos, y estos serán 10, 30 y 50, por lo que para el primer caso cada luciérnaga tendrá 10 elementos aleatorios en su vector 'solucion', para el siguiente 30 y para el último 50.

La otra función hace uso de la mencionada anteriormente para crear cada elemento e incluirlo en una población. Aunque lo lógico sería crear tantas luciérnagas como vayamos a usar en nuestro problema (en este caso 50), tenemos la opción de seleccionar qué cantidad de elementos queremos generar y con cuántos nos queremos quedar finalmente. El motivo es que cuantos más elementos generemos, más posibilidades tendremos de generar buenas soluciones de partida, pero cada generación conlleva un gasto de evaluación, por lo que habría que buscar un equilibrio entre las que generamos y las que nos quedamos.

En mi caso, se han generado 100 elementos, estos se ordenan de mejor a peor y nos quedamos con los 50 primeros de la población.

Estas funciones mencionadas son las siguientes:

algoritmo solucion_aleatoria

variables como parámetros

tipo entero cantidad_elementos

tipo Random generador

variables locales

tipo distribución real uniforme *uniforme_real[-100,100]*

vector tipo double solucion_elemento

inicio

1. Para cada i desde 0 hasta cantidad_elementos
    a. solucion_elemento.añadir(aleatorio de *uniforme_real*)
2. Devolver solucion_elemento

algoritmo crea_poblacion

variables como parámetros

tipo entero cantidad_elementos, cantidad_excesiva_elementos, dim

tipo Random generador

variables locales

vector tipo Luciernaga poblacion

inicio

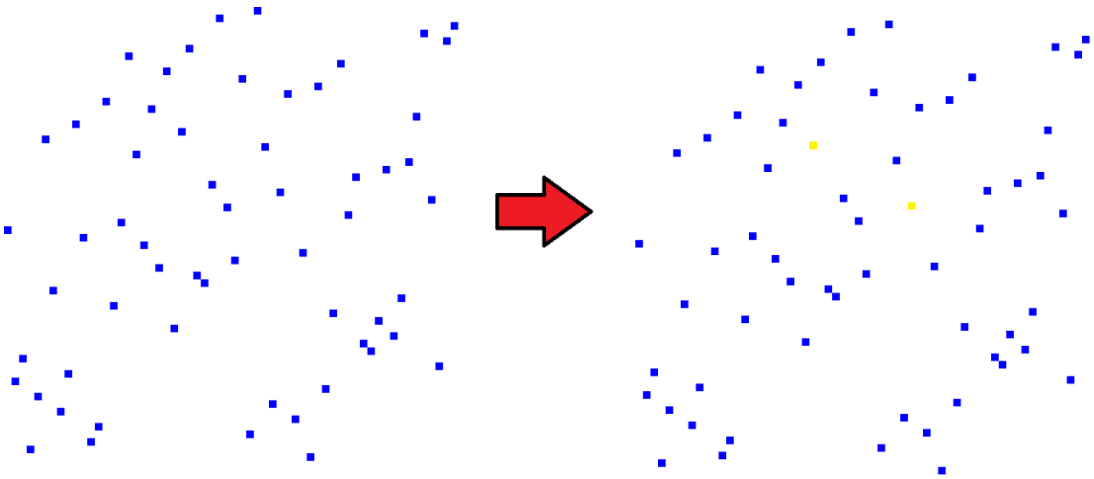
1. Para cada i desde 0 hasta cantidad_excesiva_elementos
    a. sol <- solucion_aleatoria(dim, generador)
    b. poblacion.añadir(sol)
2. ordenamos *población* de menor fitness a mayor
3. Si cantidad_excesiva_elementos es mayor que cantidad_elementos
    a. Borramos *población* desde cantidad_elementos al final
4. Si *poblacion* tiene *cantidad_elementos* elementos
5. Devolver poblacion

Hibridación (BL)

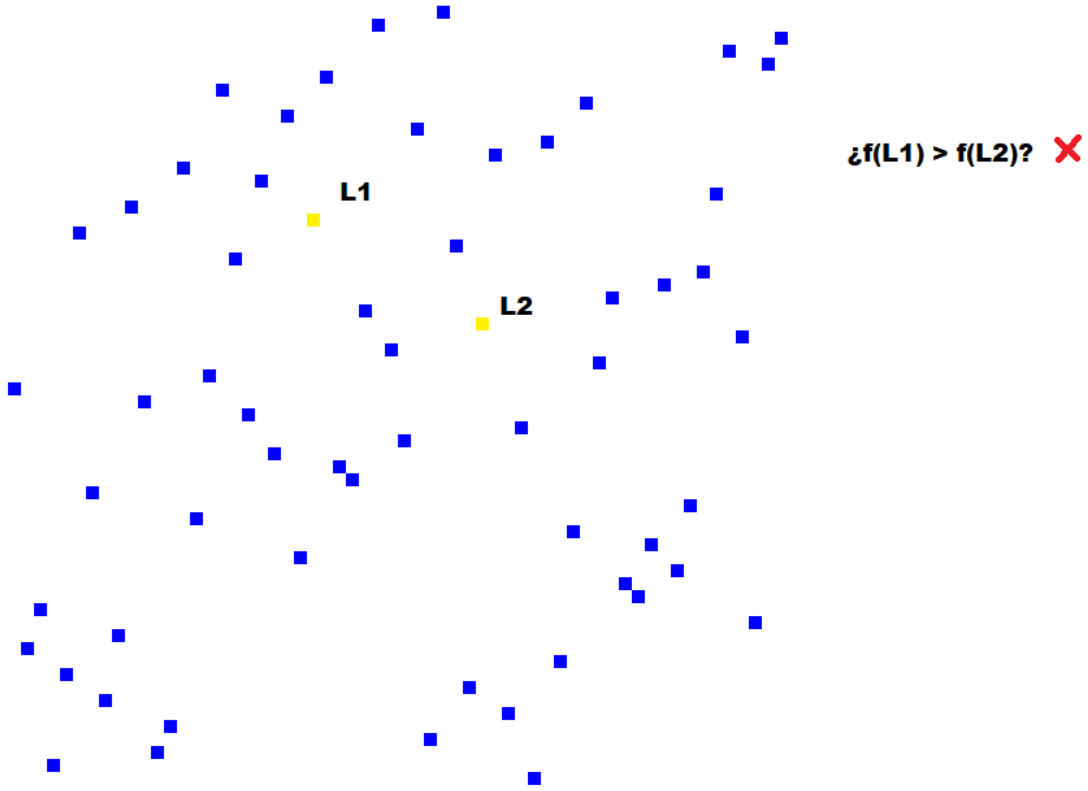
Para la hibridación se ha utilizado la búsqueda local proporcionada en el material de PRADO que fue explicada en clase. Esta se aplica a toda la población cada vez que realiza una iteración, es decir, cuando hemos pasado por la población completa acercando luciérnagas. El motivo por el que se aplica en ese momento es que con 50 luciérnagas hacemos 50 x 49 = 2450 comparaciones entre ellas, llevándose a cabo una media de 1225 movimientos. Por cada movimiento, gastamos una evaluación calculando el fitness de la luciérnaga movida en su nueva posición, por lo que al utilizar la dimensión mínima (10) haríamos una media de 100000 / 1225 = 81.63 iteraciones. Al haber limitado el número de iteraciones de BL a 100, gastaría una media de 7163 iteraciones, que es un 7.2% del total. Para los casos de dimensión 30, hay una media de 300000 / 1225 = 244.9 iteraciones, resultando en 24490 evaluaciones gastadas en la aplicación de BL (8.16%) y para los de 50, 500000 / 1225 = 408.16 * 100 (iteraciones por BL) = 40816 evaluaciones son destinadas a la hibridación (8.16%).

Una forma de ver claramente cómo se realizan una media de 1250 movimientos por iteración independientemente de la dimensión es la siguiente:

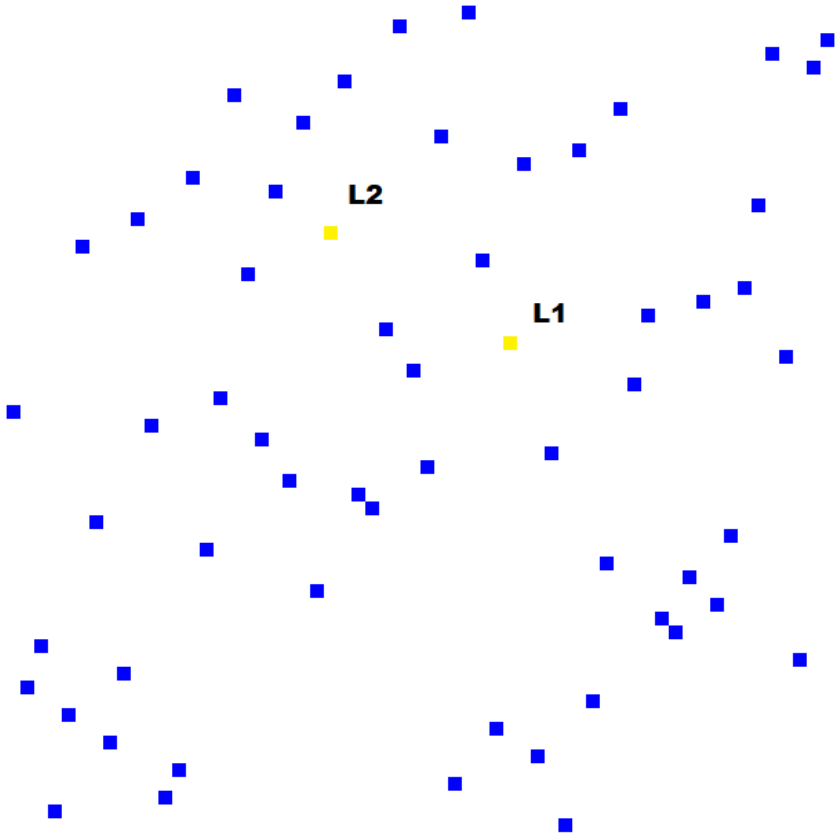
Por orden, aunque vamos a recorrer la población entera, seleccionamos las luciérnagas una a una y vamos comparándolas con todas las demás. Las dos en amarillo de la imagen derecha son las primeras seleccionadas.



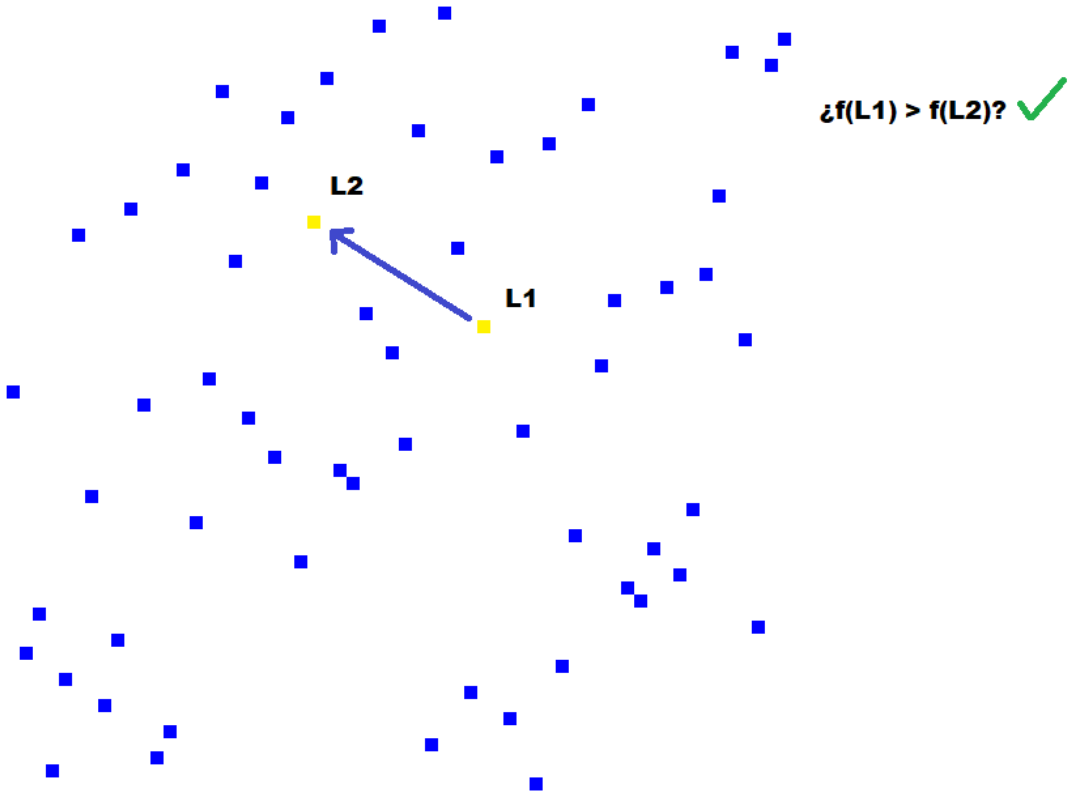
Tras la comparación veremos cuál se mueve y cuál permanece, con la posibilidad de que ninguna de las dos se mueva. Esta posibilidad existe debido a la implementación, pues esta está hecha para que solamente haya movimiento si se cumple que el primer elemento de la comparación es peor que el restante. De esta forma, supongamos que no se produciría el cambio:



Llegará un momento en la iteración en la que volvamos a comparar estas dos luciérnagas, pero L2 tendrá el papel de ser la primera y L1 será la segunda seleccionada.



Como la primera comprobación entre ellas falló, sabemos que el actual L2 no tiene mayor fitness que L1, por lo que este último tendrá más o el mismo. Como las combinaciones de soluciones son generadas aleatoriamente y el fitness viene determinado por valores reales, la probabilidad de que dos fitness sean iguales es ínfima, por lo que consideraremos que se daba que el fitness del ahora L2 es más alto. Esto hace que finalmente el movimiento sea el siguiente:



Como hemos visto, se han necesitado dos comprobaciones para que exista un movimiento, por lo que considerando los ínfimos casos en los que puede darse un empate, la cantidad de movimientos va a ser siempre la mitad de las comprobaciones que se haga, porque si el primero no es peor que el segundo, esta condición se cumplirá para la comprobación contraria.

Para este apartado, el código es el mismo que el usado hasta ahora, con la misma generación de población y luciérnagas y mismos parámetros para el movimiento. La única diferencia es el uso de la función *bl_soliswets,* quedando así nuestro main:

algoritmo FFA_hibrido

variables como parámetros

tipo entero dim

variables locales

tipo entero constante poblacion_inicial <- 50, cantidad_excesiva_elementos <- 100, evaluaciones_bl_maximas <- 100

tipo double constante delta <- 0.4

vector tipo Luciernaga población

tipo entero evaluaciones <- cantidad_excesiva_elementos

inicio

1. poblacion <- crea_poblacion(poblacion_inicial, cantidad_excesiva_elementos, dim, generador aleat.)
2. Mientras evaluaciones sea menor que 10000 x dim
    a. Para cada luciérnaga Lu de la población
        i. Aplicar BL(Lu.solucion, Lu.fitness, delta, evaluaciones_bl_maximas, -100, 100, generador)
    b. Mover_luciernaga(población, dim, evaluaciones)

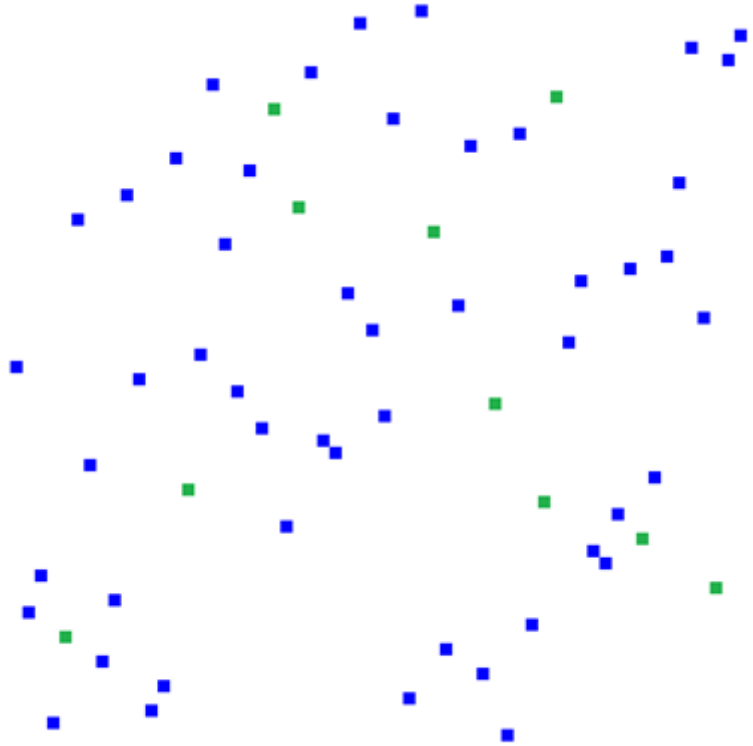
El desempeño de esta implementación será analizado en el apartado de Resultados descrito más tarde.

<u>Mejora de la metaheurística</u>

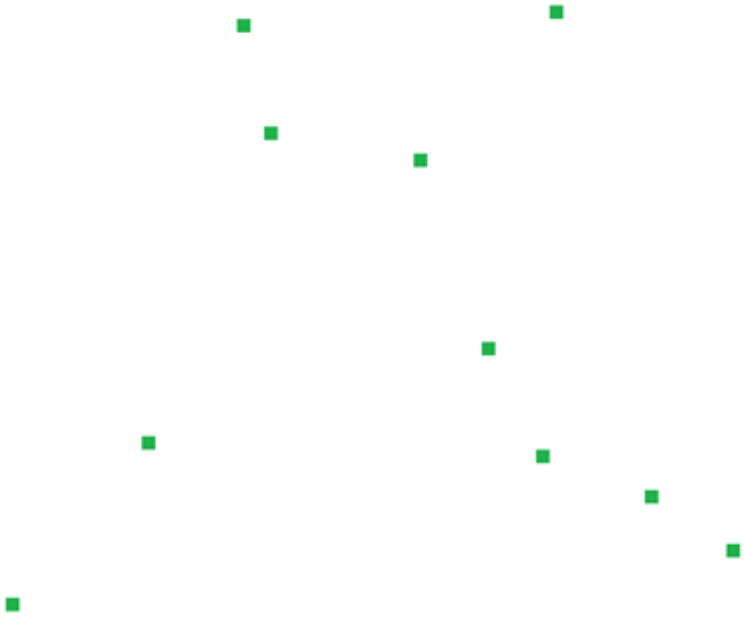
En cuanto a la mejora de la metaheurística, he pensado un modelo en el que se simula la reproducción de las mejores luciérnagas. De esta forma, no solamente tendremos algunas que se acercan a otras, sino que cada cierto tiempo se comprobará la distancia entre las 10 mejores luciérnagas y, si están lo suficientemente cerca, se genera un hijo que sustituye el peor elemento de la población.

Con más detalle, el proceso es el siguiente:

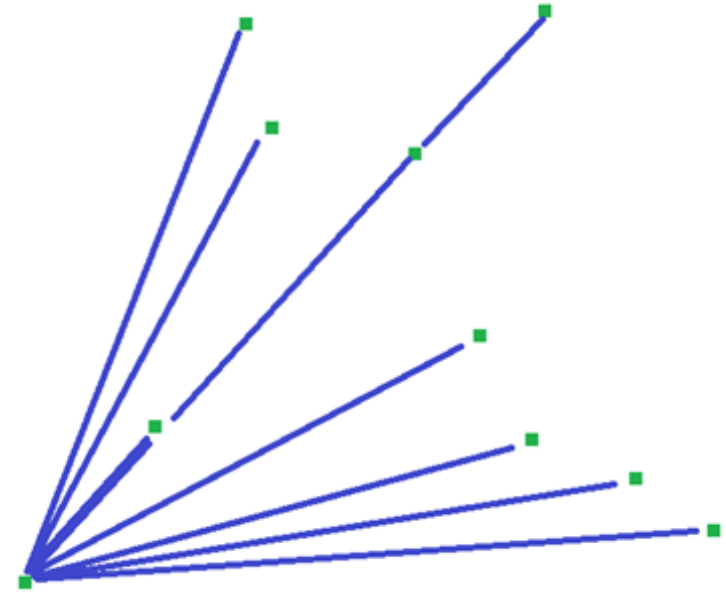
- El procedimiento base es el mismo, se genera una población inicial de 100 luciérnagas, seleccionamos las 50 mejores, aplicamos BL e iteramos por todo el conjunto haciendo comprobaciones y moviendo una de las luciérnagas de cada par generado.
- Al terminar estas 2450 comprobaciones aumentamos el contador de las iteraciones realizadas (no confundir con evaluaciones) y volvemos a aplicar BL y mover las luciérnagas.
- Cada 3 iteraciones (equivalente a unas 3673 evaluaciones) se comprueba cual es la distancia entre las 10 mejores luciérnagas. Para seguir la explicación más fácilmente, supongamos que, de la misma población de antes, las 10 mejores son las pintadas en verde.

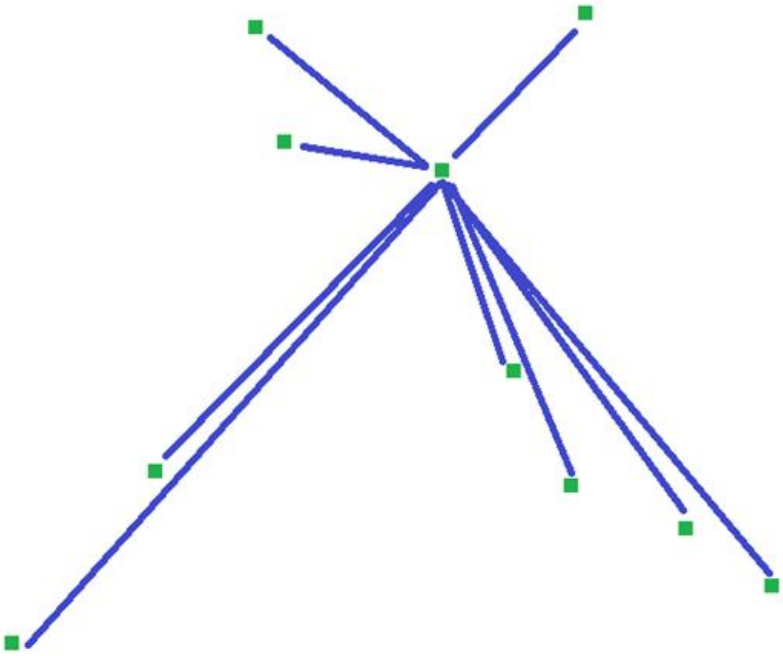


Para que sea más visual, dejaremos solamente las luciérnagas implicadas en la explicación.



- Ahora se comprueba la distancia entre las mejores.

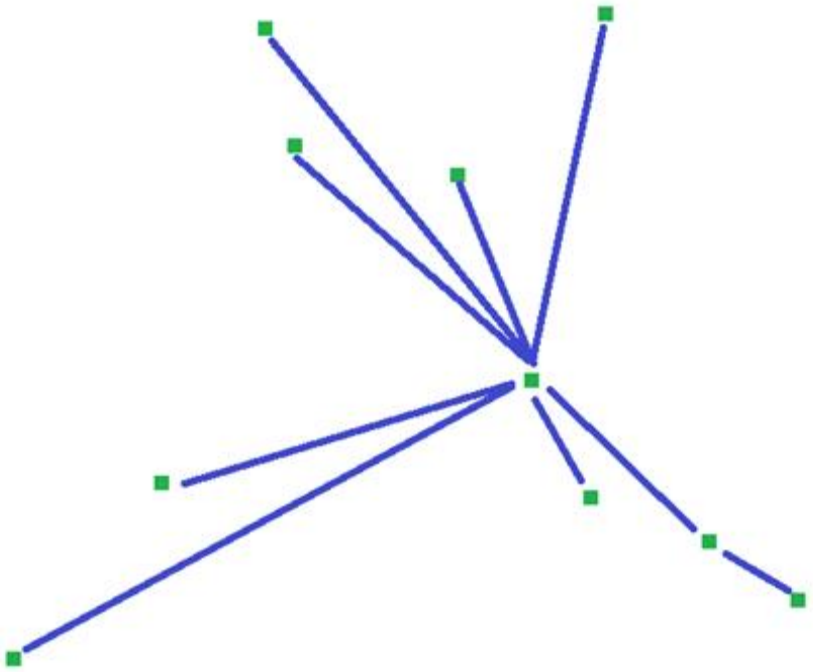




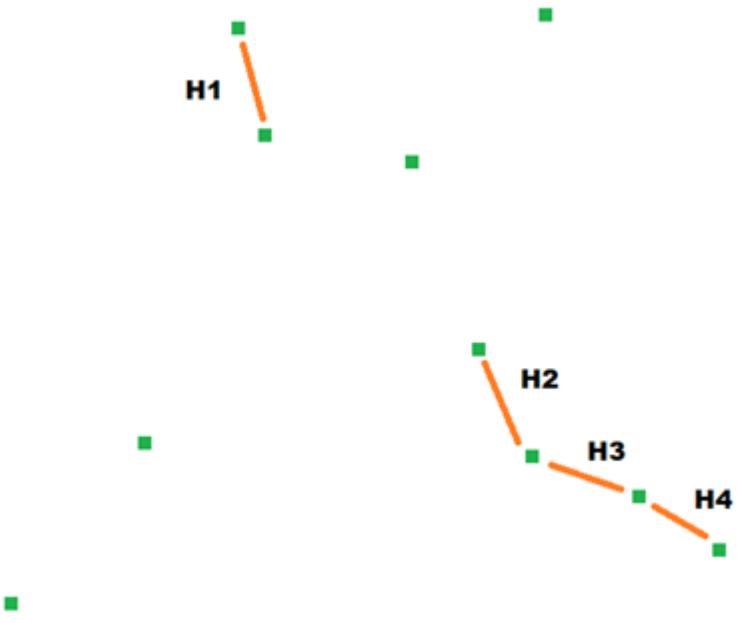
.

.

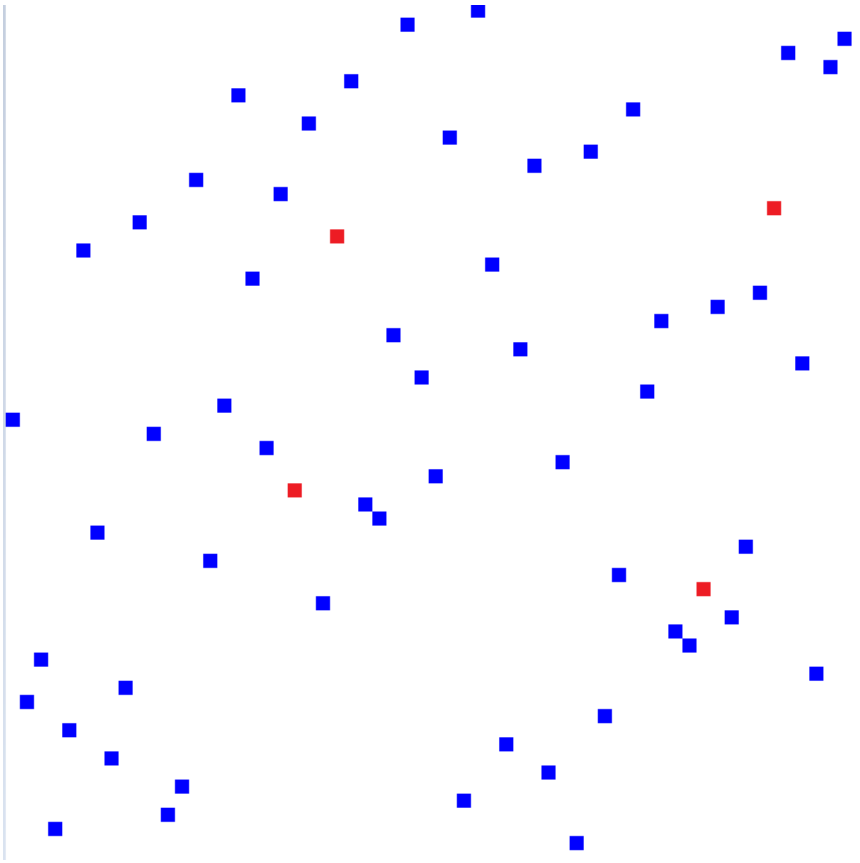
.



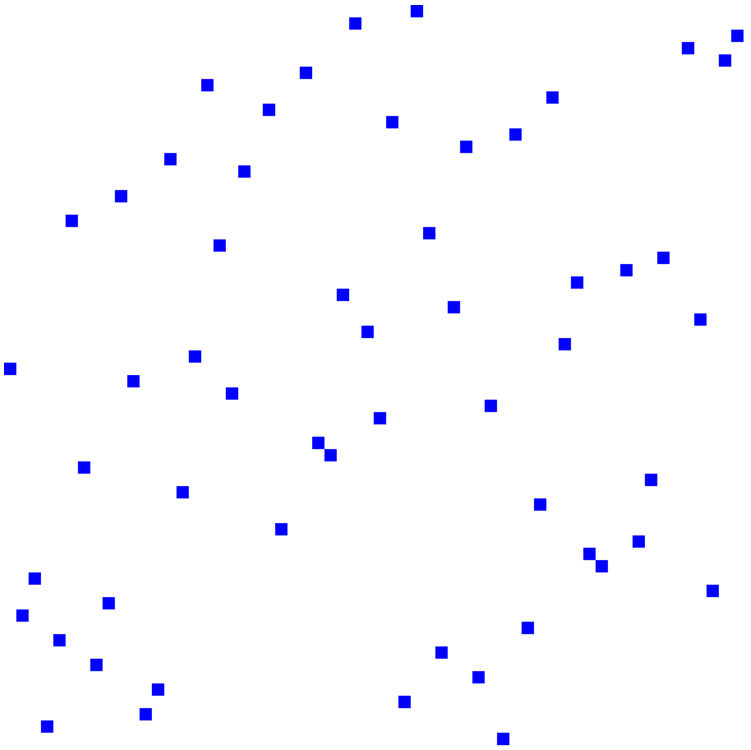
- Para aquellas parejas de luciérnagas que estén muy cerca, se genera un hijo. “Muy cerca” es algo bastante relativo, por lo que para nosotros será que la distancia entre dos de estas sea inferior a 10*dim. Eso es porque cuanto mayor es dim, más distancia suele haber entre cada elemento asociada a la generación de las soluciones aleatorias. Para ejemplo, podemos considerar como “muy cercanas” las siguientes:

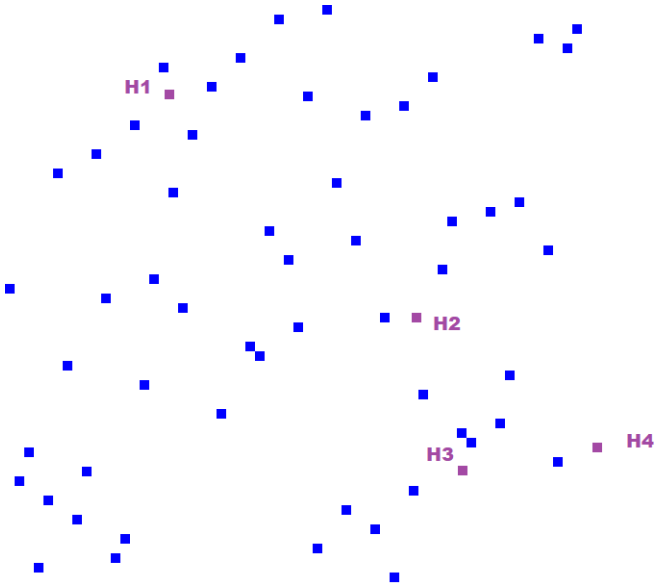


- De esta forma generaremos 4 hijos que se irá añadiendo a una población auxiliar. Tras esto, ingresamos los hijos en la población sustituyendo a los peores y, si se han generado muchos hijos, se introducen únicamente 10 de estos. Así la población irá evolucionando hacia mejores soluciones respetando la diversidad. Localizamos los 4 peores elementos en este caso:



Los eliminamos del conjunto y los sustituimos por las hijas generadas





- Cuando la reproducción ha terminado, el programa sigue su ejecución regular aplicando BL y moviendo las luciérnagas hacia las mejores, pero después de 3 iteraciones completas volverá a generar hijos.

Una ventaja de esta mejora es que permite la diversidad. Si desde la población inicial se han generado varias luciérnagas muy buenas, pero que no están cercanas unas a otras, cuando las mejores se junten, las hijas estarán en puntos intermedios (aunque más cerca del mejor de los padres) y no todas alrededor del mismo elemento bueno.

El pseudocodigo de la parte de la reproducción queda de la siguiente manera:

algoritmo FFA_mejorado

variables como parámetros

tipo entero dim

variables locales

tipo entero constante poblacion_inicial <- 50, cantidad_excesiva_elementos <- 100, evaluaciones_bl_maximas <- 100

tipo double constante delta <- 0.4

vector tipo Luciernaga población, población_auxiliar

tipo entero evaluaciones <- cantidad_excesiva_elementos, contador <- 0

inicio

3. poblacion <- crea_poblacion(poblacion_inicial, cantidad_excesiva_elementos, dim, generador aleat.)
4. Mientras evaluaciones sea menor que 10000 x dim
    a. Si contador es multiplo de 3
        i. Ordenamos poblacion
        ii. Para cada i desde 0 hasta 9
            1. Para cada j desde i+1 hasta 10
                a. (double) distancia <- 0
                b. Para cada el3 desde 0 hasta dim
                    i. distancia <- distancia + (el1.solucion(el3) - el2.solucion(el3))^2
                c. distancia <- raíz cuadrada(distancia)
                d. Si distancia es menor que 10xdim
                    i. Reproducción(población(i), población(j), población_auxiliar)
        iii. Para cada elemento de la poblacion_auxiliar o los 10 primeros si hay mas
            1. Sustituir elemento del final de poblacion por uno de poblacion_auxiliar
    b. Para cada luciérnaga Lu de la población
        i. Aplicar BL(Lu.solucion, Lu.fitness, delta, evaluaciones_bl_maximas, -100, 100, generador)
    c. Mover_luciernaga(población, dim, evaluaciones)

algoritmo reproduccion

variables como parámetros

tipo Luciernaga elem1, elem2

vector tipo Luciernaga &poblacion

variables locales

tipo Luciernaga hijo

tipo entero numero_cambios <- 0.2 x elem1.solucion.tamaño

inicio

1. Si elem1.fitness es menor que elem2.fitness
    a. hijo <- elem1
    b. Para cada i desde 0 hasta numero_cambios
        i. hijo[i] <- elem2[i]
2. Si elem2.fitness es menor que elem1.fitness
    a. hijo <- elem2
    b. Para cada i desde 0 hasta numero_cambios
        i. hijo[i] <- elem1[i]
3. hijo.fitness <- función_fitness(hijo.solucion)
4. Añadir el elemento en la poblacion

En ningún momento se comprueba que los hijos generados sean mejores que los elementos que se están sustituyendo. Esto es porque se simula una evolución real en la que los mejores elementos perduran y los peores van desapareciendo y, siendo hijos de elementos buenos, se cuenta con que pueden ser buenos también.

No siempre es así, pero con el factor de aleatoriedad, podemos encontrar soluciones que nos ayuden a escapar de mínimos locales.

Análisis de resultados

El análisis de resultados estará basado en la observación y posible explicación de los resultados obtenidos de la web Select your Benchmark (tacolab.org) al comparar con otros algoritmos conocidos. Para cada parte del programa se han hecho las evaluaciones completas, ejecutando el original, el que incluye hibridación y el mejorado con las dimensiones 10, 30 y 50 por separado. Tras esto, la elección de algoritmos para la comparación ha salido de la recomendación del profesor de escoger al menos DE (Differential evolution) y PSO (particle swarm optimization) y el tipo de informe seleccionado es 'Ranking Comparison', ya que el de la media solamente saca tablas individuales y nos sería costoso reunir toda la información independiente. Con el seleccionado considero que es suficiente para obtener la información que nos interesa, que es el rendimiento del programa, este comparado con él mismo mejorado y ambos comparados con otros algoritmos.

Primero analizaremos el código original para todas las dimensiones posibles.

Básico con dimensión = 10

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,766667	2,766667	2,733333	2,233333	1,466667	1,433333	1,4	1,4	1,433333	1,5	1,5	1,5	1,5	1,5
PSO	1,833333	1,766667	1,733333	1,9	2,2	2,066667	2,066667	2,033333	1,933333	1,866667	1,866667	1,9	1,9	1,866667
Sheet1	1,4	1,466667	1,533333	1,866667	2,333333	2,5	2,533333	2,566667	2,633333	2,633333	2,633333	2,6	2,6	2,633333

Básico con dimensión = 30

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,733333	2,733333	2,6	1,633333	1,233333	1,166667	1,1	1,066667	1,066667	1,066667	1,1	1,1	1,1	1,1
PSO	1,866667	1,833333	1,866667	2,4	2,333333	2,333333	2,2	2,133333	2,1	2,066667	2,066667	2,066667	2,033333	2
Sheet1	1,4	1,433333	1,533333	1,966667	2,433333	2,5	2,7	2,8	2,833333	2,866667	2,833333	2,833333	2,866667	2,9

Básico con dimensión = 50

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,733333	2,666667	2,433333	1,666667	1,266667	1,1	1,166667	1,166667	1,166667	1,166667	1,1	1,1	1,1	1,1
PSO	1,9	1,8	1,866667	2,166667	2,3	2,333333	2,133333	2,066667	2,033333	2	2	2	2	2
Sheet1	1,366667	1,533333	1,7	2,166667	2,433333	2,566667	2,7	2,766667	2,8	2,833333	2,9	2,9	2,9	2,9

Como vemos, cuanto menor es el valor de la matriz, mejor es ese algoritmo para ese porcentaje de iteraciones. Para dimensión 10, el algoritmo original funciona mejor que los otros dos cuando se realizan un 5% de las iteraciones o menos, pero, a partir de ahí, su comportamiento es bastante malo y no vuelve a ser el mejor en ningún momento. Obviamente, el porcentaje para el que nos interesa que nuestro algoritmo sea el mejor es el de la última columna que se corresponde con la ejecución de todo el programa. De hecho, desde el momento en el que deja de tener el mejor comportamiento pasa a ser el peor de los 3 cuando, según ciertas especificaciones de la clase de prácticas, deberíamos llegar a conseguir mejores resultados que PSO.

Al aumentar la dimensión, lo que aumentamos es la cantidad de elementos que tiene el vector de cada luciérnaga de la población y, por tanto, este aumento supone trabajar con más datos. Cuando esto ocurre, DE mejora su resultado, dejando en claro que, para mayores cantidades de datos a manejar, mejores soluciones encuentra; mientras que los otros dos algoritmos empeoran con el aumento de dimensión. Entre la dimensión 30 y 50 no se nota ninguna diferencia abultada en ninguno de los 3 algoritmos, por lo que ambos pueden tener un umbral a partir del cual mejorar es bastante complejo.

Híbrido con dimensión = 10

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,766667	2,7	2,666667	2,233333	1,566667	1,5	1,533333	1,5	1,533333	1,6	1,633333	1,633333	1,633333	1,633333
PSO	1,8	1,733333	1,733333	1,8	2,333333	2,466667	2,533333	2,566667	2,533333	2,533333	2,533333	2,5	2,5	2,466667
Sheet1	1,433333	1,566667	1,6	1,966667	2,1	2,033333	1,933333	1,933333	1,933333	1,866667	1,833333	1,866667	1,866667	1,9

Híbrido con dimensión = 30

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,7	2,6	2,633333	1,866667	1,666667	1,533333	1,466667	1,433333	1,3	1,3	1,333333	1,333333	1,333333	1,333333
PSO	1,633333	1,5	1,866667	2,566667	2,733333	2,8	2,8	2,766667	2,766667	2,733333	2,666667	2,633333	2,666667	2,666667
Sheet1	1,666667	1,9	1,5	1,566667	1,6	1,666667	1,733333	1,8	1,933333	1,966667	2	2,033333	2	2

Híbrido con dimensión = 50

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,5	2,766667	2,633333	2,1	1,7	1,533333	1,6	1,566667	1,533333	1,5	1,466667	1,4	1,4	1,366667
PSO	1,466667	1,9	2	2,566667	2,733333	2,833333	2,766667	2,766667	2,766667	2,7	2,733333	2,733333	2,733333	2,733333
Sheet1	2,033333	1,333333	1,366667	1,333333	1,566667	1,633333	1,633333	1,666667	1,7	1,8	1,8	1,866667	1,866667	1,9

Una vez que aplicamos la BL la cosa cambia ligeramente. El uso de la BL que hacemos está muy limitado en el sentido de la cantidad de evaluaciones que le permitimos gastar. Podría existir una relación que facilite encontrar el equilibrio entre las evaluaciones que renta gastar en BL y las que debemos dejar al algoritmo original para avanzar por su cuenta, pero una aplicación de esta bajo consideraciones propias ya ha obtenido mejores resultados. Para cualquiera de las dimensiones vemos que se obtiene mejor resultado para cualquiera de las mismas probadas sin la hibridación, pero veámoslo con más detalle. Con una dimensión de 10, es el mejor de los tres hasta alcanzar el 5% de las evaluaciones, donde deja de serlo. En este caso, vemos que dura menos siendo el mejor, pero a partir de este número no se sitúa como el peor, cosa que pasaba con el código original. Ahora hemos conseguido que cuando deja de ser el mejor, al menos los valores que obtiene son mejores que los que obtiene PSO, que dentro de lo que cabe era un comportamiento que esperábamos conseguir.

Si nos fijamos en la dimensión 30 curiosamente no empieza siendo el mejor como en el resto de casos anteriores, sino que es PSO quien obtiene resultados más bajo, pero cuando supera el 2% de las evaluaciones nuestro algoritmo se sitúa como el mejor. Nuevamente deja de serlo antes de llegar a la última columna, pero como novedad se mantiene correcto hasta el 10% con este incluido. A partir de este pasa a ser peor que DE, pero mejor que PSO e incluso mejor que la hibridación en dimensión 10 (hasta el 50% donde se tornan los papeles y el BL con dimensión 10 trabaja mejor). Ya con dimensión 50 vemos un comportamiento muy similar al de 30, donde PSO para pocas evaluaciones es bueno, para los siguientes porcentajes es superado por nuestro algoritmo y desde el 10% en delante DE trabaja mejor. Nuevamente se mantiene como segundo mejor algoritmo hasta el final de las pruebas.

En cuanto a la comparación entre el algoritmo original y el híbrido, vemos cómo en las primeras columnas es mejor el que no tiene Búsqueda Local, pero puede ser debido a que, al no hacer todas las iteraciones, estas las haya gastado únicamente en hacer BL y no le permita avanzar mucho. Lo importante es cómo trabajan para la totalidad del problema y, fijándonos en eso, demostramos que la hibridación efectivamente mejora el rendimiento para todas las dimensiones.

Mejorado con dimensión = 10

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,766667	2,7	2,666667	2,233333	1,533333	1,466667	1,5	1,533333	1,5	1,566667	1,566667	1,566667	1,566667	1,6
PSO	1,8	1,733333	1,733333	1,8	2,433333	2,566667	2,533333	2,5	2,5	2,466667	2,5	2,466667	2,466667	2,466667
Sheet1	1,433333	1,566667	1,6	1,966667	2,033333	1,966667	1,966667	1,966667	2	1,966667	1,933333	1,966667	1,966667	1,933333

Mejorado con dimensión = 30

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,7	2,633333	2,6	1,833333	1,566667	1,433333	1,333333	1,333333	1,3	1,3	1,333333	1,3	1,266667	1,3
PSO	1,633333	1,433333	1,766667	2,5	2,7	2,766667	2,866667	2,833333	2,866667	2,866667	2,766667	2,733333	2,733333	2,766667
Sheet1	1,666667	1,933333	1,633333	1,666667	1,733333	1,8	1,8	1,833333	1,833333	1,833333	1,9	1,966667	2	1,933333

Mejorado con dimensión = 50

	1	2	3	5	10	20	30	40	50	60	70	80	90	100
DE	2,5	2,733333	2,633333	2,1	1,833333	1,633333	1,6	1,533333	1,533333	1,533333	1,466667	1,4	1,366667	1,333333
PSO	1,466667	1,733333	2,033333	2,566667	2,766667	2,866667	2,766667	2,766667	2,766667	2,766667	2,8	2,8	2,8	2,766667
Sheet1	2,033333	1,533333	1,333333	1,333333	1,4	1,5	1,633333	1,7	1,7	1,7	1,733333	1,8	1,833333	1,9

Centrando la vista en nuestra mejora, podemos ver cómo tiene un comportamiento similar a lo ya observado. Mejores resultados con porcentajes bajos (alcanzando el 20% con dimensión 50) y peores conforme vamos realizando mayor parte del problema. En comparación con la BL, que es lo que nos interesa conseguir mejorar, con dimensión 10 se obtienen peores resultados, con dimensión 30 los mejoramos y con dimensión 50 obtenemos exactamente lo mismo. Como además cuentan con un factor aleatorio en la población de la que parten y el tratamiento de los datos, una mejora tan insignificante no deja en claro nada, podríamos considerar que los resultados son simplemente similares. Es posible que con más tiempo de investigación podríamos encontrar un equilibrio de evaluaciones para BL y parámetros para la reproducción que hagan de la mejora una mejora real, pero hasta entonces puede ser considerada una alternativa.

Conclusion

El mal comportamiento original del programa puede ser consecuencia de lo lento que avanza entre las soluciones hacia las mejores. Otra cosa no muy prometedora es el movimiento aleatorio de la mejor, pues este le puede hacer empeorar en la mayoría de los casos y nos proporciona simple fugas ínfimamente probables hacia soluciones mejores.

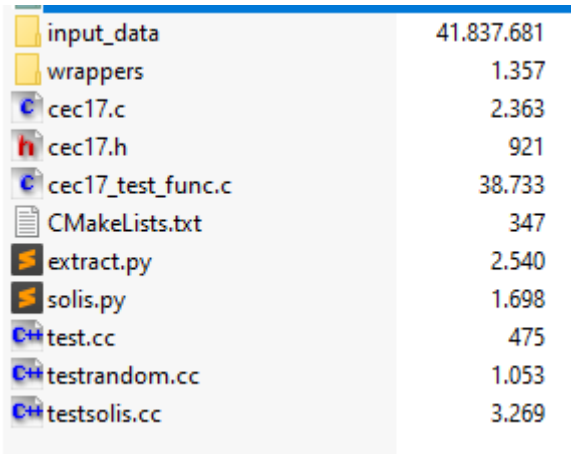
Independientemente del resultado, la mejora no debe ser descartada del todo porque no ha sido sometida a la más exhaustiva de las investigaciones. Considero que, haciendo un buen uso de parámetros y medidas, obtener buenos resultados a partir de otros que ya lo son y eliminar los menos prometedores debe estar conducido a mejorar la solución.

En cuanto al interés en la práctica alternativa, me siento gratamente satisfecho con el hecho de haber investigado y profundizado en el mundo de las metaheurísticas basadas en elementos naturales y, sobre todo, en el de las luciérnagas. Era un mundo que conocía superficialmente y, aunque aún le falta bastante camino por mejorar en cuanto a respaldo matemático, es interesante a nivel didáctico contar con ejemplos de la vida real en un ámbito en el que todo es tan abstracto.

Manual de uso

Para hacer uso de este programa necesitamos tener descargado el directorio *code* que se encuentra en repositorio de Daniel Molina https://github.com/dmolina/cec2017real.

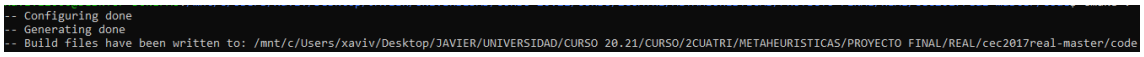
Una vez nos encontremos en una carpeta con este contenido:



Abrimos una terminal de Ubuntu o Debian y nos movemos a este directorio. Escribimos el comando

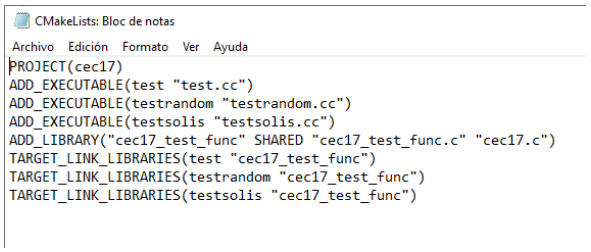
$ cmake .

y veremos algo como:

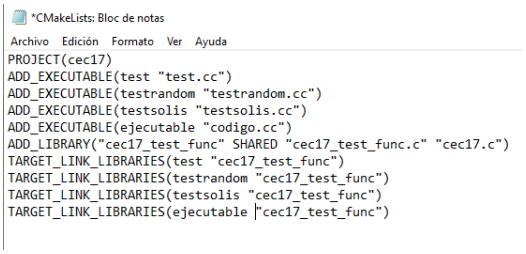


Ahora abrimos el fichero *CMakeLists.txt* y ponemos nuestro .cc y el nombre del ejecutable que generará. Además, linkamos como es debido:

Original

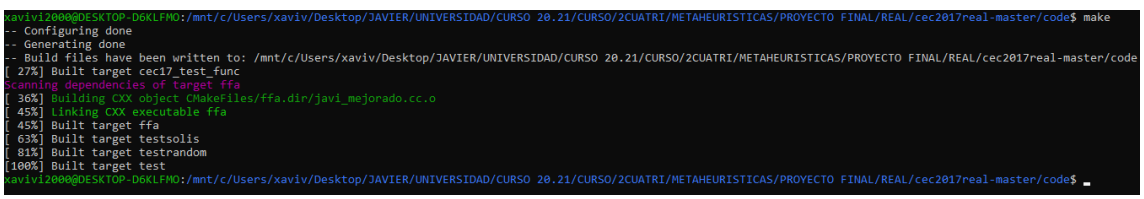


Modificado



El resto de ejecutables generados a partir de códigos que ya venían en el directorio no son necesarios, pero los podemos dejar si queremos que no influyen.

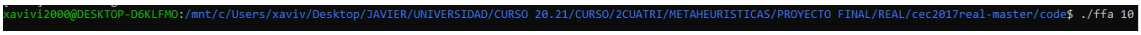
Ahora podemos hacer el make:



Por último, para ejecutarlo solo necesitamos poner:

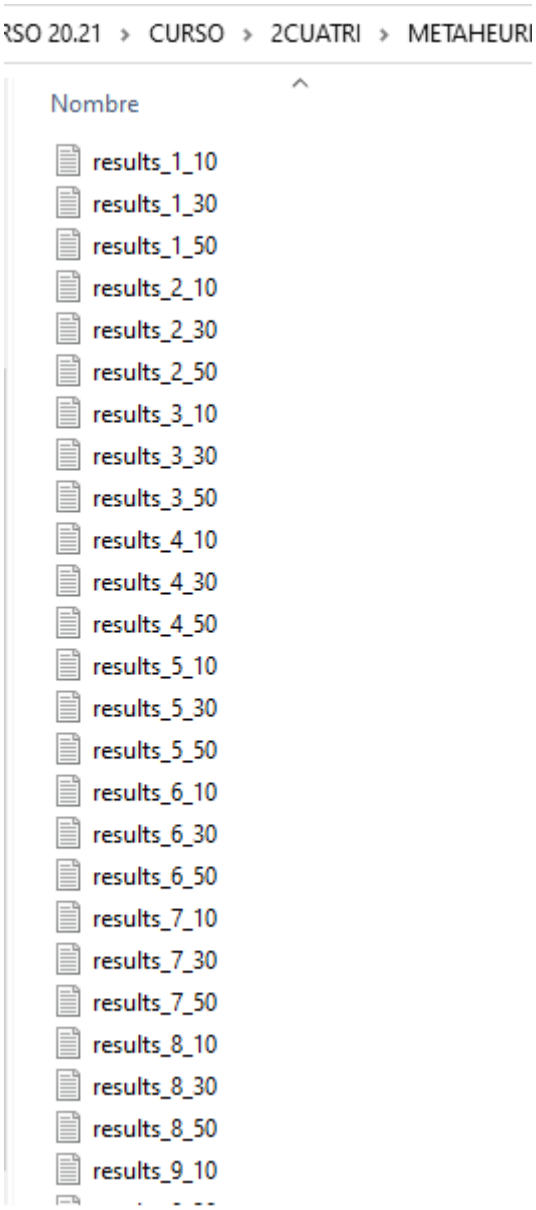
$ ./nombre_ejecutable [DIM]

Por ejemplo:



En el código debe venir especificado el nombre de la carpeta en la que se registrarán los datos. Esto se hace con el cec17_init("nombre_directorio", id_funcion, dim) y debe existir en el mismo directorio en el que estás trabajando un directorio llamado "results_nombre_directorio".

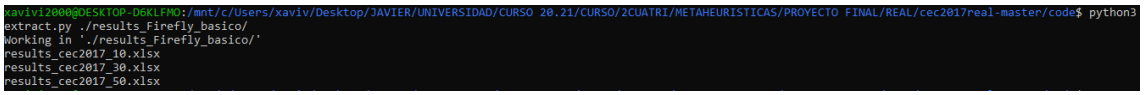
Una vez ejecutemos el programa para todas las dim que queramos, el contenido se guarda en este directorio en ficheros separados por dimensión y número de ejecución de las 30 que hace cada vez. El contenido de nuestra carpeta de resultados debería verse tal que:



Y desde la terminal podremos hacer uso del comando:

$ python3 extract.py ./results_nombre_directorio/

Para generar un Excel con las soluciones obtenidas para cada dim y que será necesario para las comparaciones en tacolab.



Bibliografía

Pseudocodigo general y críticas: Firefly algorithm - Wikipedia

Resumen e introduccion: Yang, Xin-She. (2010). Firefly Algorithm, Stochastic Test Functions and Design Optimisation. International Journal of Bio-inspired Computation. 2. 10.1504/IJBIC.2010.032124. ((PDF) Firefly Algorithm, Stochastic Test Functions and Design Optimisation (researchgate.net))

Definición de algoritmo determinista: Algoritmo determinista - Wikipedia, la enciclopedia libre

Ventajas: Firefly algorithm (slideshare.net)

Parámetros recomendados: Performance evaluation of firefly algorithm with variation in sorting for non-linear benchmark problems (scitation.org)