UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Instituto de Investigaciones en Mateáticas aplicadas y Sistemas

Algoritmo Bio-insipirado Concurrente

Integrantes

Hernández Lozano Juan Pablo
Mosqueda García Raúl Isaid
Ruiz Puga Ingrid Pamela
Veleros Vega Luis Alfonso

Maestro

Oscár Esquivel

Programación Concurrente

Índice

1. Resumen **3**

2. Introducción **3**

3. Marco teórico **4**
3.1. Algoritmos genéticos 4
3.1.1. Paralelismo 5
3.2. Algoritmos meméticos 5
3.2.1. Concurrencia y paralelismo 7
3.3. Cuckoo Search 7
3.3.1. Concurrencia 8
3.4. Problema de las Ocho Reinas 8

4. Selección de algoritmo **9**
4.1. Algorítmo de la colonia artificial de abejas 10
4.2. Meta-heurística 11
4.3. Concurrencia y paralelismo 11

5. Implementación **11**
5.1. Algorítmo 12

6. Documentación de código **13**

7. Análisis de eficiencia **20**
7.1. Resultados 20
7.2. Estadísticas del algoritmo 21
7.3. Speed up 22

1. Resumen

En este proyecto se plantea realizar el análisis de varios algorítmos evolutivos y ver para cuales de estos es posible realizar una implementación con cómputo concurrente o paralelo, se busca encontrar de los algorítmos elegidos cuales son los mas adaptables a la concurrencia y paralelismo.

Al concluir este proyecto se desea tener un concepto mas estructurado de los algoritmos evolutivos y bio-inspirados , así como de los tipos de los mismos y/o vertientes, dedicaremos una sección a los algoritmos genéticos. También se realizará la comparación y/o análisis de diferentes tipos de algoritmos para ciertos problemas. Al final se presentará una conclusión retomando conceptos aprendidos y la implementación del algoritmo seleccionado.

2. Introducción

Muchos problemas de optimización que aparecen en los ámbitos de las ingenierías son muy difíciles de solucionar por medio de técnicas tradicionales, por lo que a menudo se aplican algoritmos evolutivos, inspirados en la naturaleza, que recogen un conjunto de modelos basados en la evolución de los seres vivos.

Los algoritmos evolutivos trabajan con una población de individuos, que representan soluciones candidatas a un problema. Esta población se somete a ciertas transformaciones y después a un proceso de selección, que favorece a los mejores. Cada ciclo de transformación y selección constituye una generación, de forma que después de cierto número de generaciones se espera que el mejor individuo de la población esté cerca de la solución buscada. Los algoritmos evolutivos combinan la búsqueda aleatoria, dada por las transformaciones de la población, con una búsqueda dirigida dada por la selección.

Los algoritmos bio-inspirados son algoritmos no determinísticos, que a menudo presentan, implícitamente, una estructura paralela (múltiples agentes), y son adaptativos (utilizan realimentación con el entorno para modificar el modelo y los parámetros).

Dada una población que posee un numero finito y constante de individuos, el proceso de selección natural o selección del más fuerte , ejercido por el entorno y las circunstancias que lo rodean.

- Los algoritmos bioinspirados están basados en una población , esto es ,son capaces de procesar simultáneamente una colección completa de individuos. la recombinación
- La mayoría de los algoritmos bioinspirados utilizan la recombinación para mezclar la información de varias soluciones candidatas para obtener una nueva , que se espera sea la mas apta
- Estos algoritmos son estocásticos

Los elementos de un algoritmo bio-inspirado son:

1. Inicialización
2. Población
3. Representación
4. Función de evaluación
5. Mecanismos de selección o supervivencia del más fuerte
6. Operadores de variación : recombinación y mutacón
7. Reemplazo o eliminación del mas débil
8. Condición de terminación

3. Marco teórico

En el siguiente marco teórico se presentan investigaciones sobre distintos tipos de algoritmos evolutivos, para los cuales se desarrolla la metodología para realizar el algoritmo, ejemplos de aplicación y una idea del pseudocódigo de cada uno de los algoritmos.

3.1. Algoritmos genéticos

Los algoritmos genéticos constituyen una técnica poderosa de búsqueda y optimización con un comportamiento altamente paralelo, inspirado en el principio darwiniano de selección natural y reproducción genética. En este principio de selección de los individuos más aptos, tienen mayor longevidad y por tanto mayor probabilidad de reproducción. Los individuos descendientes de estos individuos tienen una mayor posibilidad de transmitir sus códigos genéticos a las próximas generaciones.

Actualmente los algoritmos genéticos han cobrado gran importancia por su potencial como una técnica importante para la solución de problemas complejos, siendo aplicados constantemente en la ingeniería. Las aplicaciones de los algoritmos genéticos han sido muy conocidas en áreas como diseño de circuitos, cálculo de estrategias de mercado, reconocimiento de patrones, acústica, ingeniería aeroespacial, astronomía y astrofísica, química, juegos, programación y secuenciación de operaciones, contabilidad lineal de procesos, programación de rutas, interpolación de superficies, tecnología de grupos, facilidad en diseño y localización, transporte de materiales y muchos otros problemas que involucran de alguna manera procesos de optimización.

Los elementos de un algoritmo genético son:

1. **Inicialización del problema a optimizar:** Los algoritmos genéticos tienen su campo de aplicación importante en problemas de optimización complejos, donde se tiene diferentes parámetros o conjuntos de variables que deben ser combinadas para su solución. También se enmarcan en este campo problemas con muchas restricciones y problemas con espacios de búsqueda muy grandes.

2. **Representación para la solución del problema**: Para la solución de un problema en particular es importante definir una estructura del cromosoma de acuerdo con el espacio de búsqueda. En el caso de los algoritmos genéticos la representación más utilizada es la representación binaria, debido a su facilidad de manipulación por los operadores genéticos, su fácil transformación en números enteros o reales, además de la facilidad que da para la demostración de teoremas.

3. **Decodificación del cromosoma:** Para determinar el número real que la cadena binaria de caracteres representa. Es importante tener en cuenta que la cadena no representa un número real, sino que este número binario etiqueta un número dentro del intervalo inicialmente fijado.

4. **Evaluación de un individuo:** Nos muestra el valor de aptitud de cada uno de los individuos. Esta aptitud viene dada por una función que es la unión entre el mundo natural y el problema a resolver matemáticamente. Esta función de aptitud es particular para cada problema particular a resolver y representa para un algoritmo genético lo que el medio ambiente representa para los humanos.

5. **Operadores Evolutivos: Selección** Un algoritmo genético puede utilizar muchas técnicas diferentes para seleccionar a los individuos que deben copiarse hacia la siguiente generación, pero abajo se listan algunos de los más comunes. Algunos de estos métodos son mutuamente exclusivos, pero otros pueden utilizarse en combinación, algo que se hace a menudo. Las diferentes variantes en la operación de selección son: Selección elitista, selección proporcional a la aptitud, selección por rueda de ruleta, selección escalada, selección por torneo, selección por rango, selección por estado estacionario,etc.

6. **Operadores Genéticos: Cruce y Mutación** Una vez que la selección ha elegido a los individuos aptos, éstos deben ser alterados aleatoriamente con la esperanza de mejorar su aptitud para la siguiente generación. Existen dos estrategias básicas para llevar esto a cabo. La primera y más sencilla se llama mutación. Al igual que una mutación en los seres vivos cambia un gen por otro, una mutación en un algoritmo genético también causa pequeñas alteraciones en puntos concretos del código de un individuo. El segundo método se llama cruce, e implica elegir a dos individuos para que intercambien segmentos de su código, produciendo una "descendencia" artificial cuyos individuos son combinaciones de sus padres.

3.1.1. Paralelismo

El primer y más importante punto es que los algoritmos genéticos son intrínsecamente paralelos. La mayoría de los otros algoritmos son en serie y sólo pueden explorar el espacio de soluciones hacia una solución en una dirección al mismo tiempo, y si la solución que descubren resulta subóptima, no se puede hacer otra cosa que abandonar todo el trabajo hecho y empezar de nuevo. Sin embargo, ya que los algoritmos genéticos tienen descendencia múltiple, pueden explorar el espacio de soluciones en múltiples direcciones a la vez. Si un camino resulta ser un callejón sin salida, pueden eliminarlo fácilmente y continuar el trabajo en avenidas más prometedoras, dándoles una mayor probabilidad en cada ejecución de encontrar la solución.

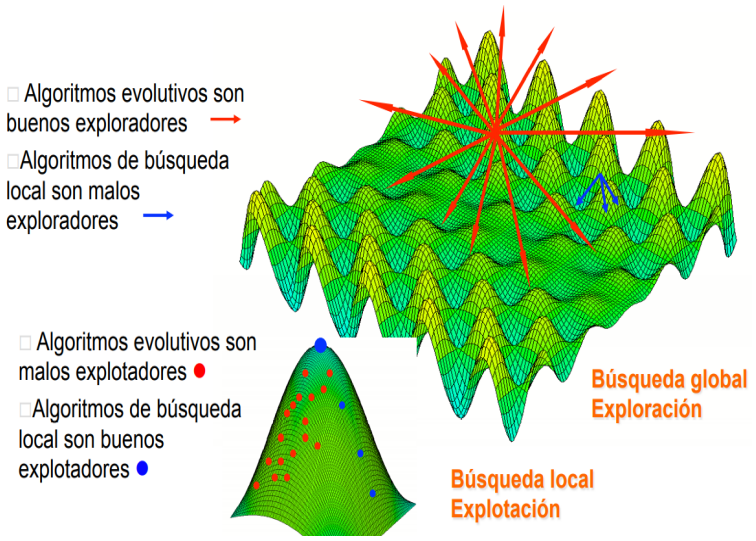
Otra ventaja del paralelismo es que, al evaluar la aptitud de una cadena particular, un algoritmo genético sondea al mismo tiempo cada uno de los espacios a los que pertenece dicha cadena. Tras muchas evaluaciones, iría obteniendo un valor cada vez más preciso de la aptitud media de cada uno de estos espacios, cada uno de los cuales contiene muchos miembros. Por tanto, un algoritmo genético que evalúe explícitamente un número pequeño de individuos está evaluando implícitamente un grupo de individuos mucho más grande. De la misma manera, el algoritmo genético puede dirigirse hacia el espacio con los individuos más aptos y encontrar el mejor de ese grupo. En el contexto de los algoritmos evolutivos, esto se conoce como teorema del esquema, y es la ventaja principal de los algoritmos genéticos sobre otros métodos de resolución de problemas.

3.2. Algoritmos meméticos

Algoritmo basado en la evolución de poblaciones que para realizar búsqueda heurística intenta utilizar todo el conocimiento sobre el problema (usualmente conocimiento en términos de algoritmos específicos de buśqueda local para el problema).

Los MA son mantienen un conjunto de soluciones candidatas para el problema considerado. De acuerdo a la jerga utilizada en los algortimos evolutivos, a cada una de estas soluciones tentativas se les conoce como individuos.Una de las premisas de los MA es buscar mejoras individuales de las soluciones en cada uno de los agentes junto con procesos de cooperación.

En el área de Investigación de Operaciones, y en general en los métodos de optimización, los MA han aportado ideas y técnicas revolucionarias, directamente en problemas como el scheduling o secuenciación de trabajos, diseño de redes de distribución y centros de distribución, diseño de rutas, y en general a en problemas de optimización discreta y continua muy comunes en la administración de operaciones El diseño de los algoritmos memeticos fue realizado con el propósito de que este fuera adecuado para ejecutarse tanto de manera secuencial como en paralelo.



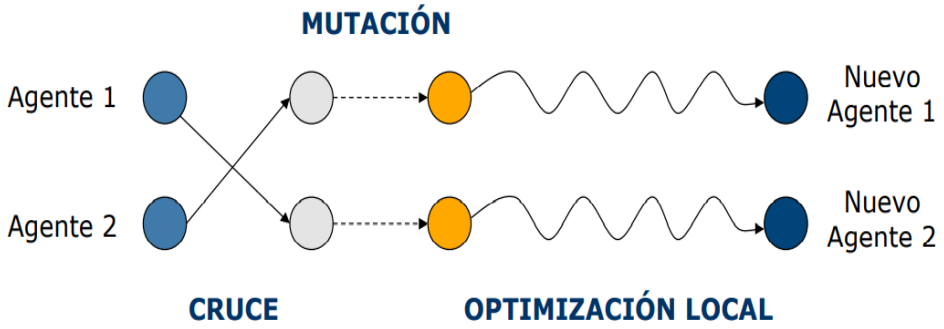
Para ver la amplia cobertura que han tenido los MA resolviendo esta clase de problemas, se muestra a continuación una lista con las aplicaciones de mayor importancia:

- Problema del viajante de comercio
- Problemas de particionado de grafos
- Partición de números
- Conjunto Independiente de cardinalidad máxima
- Coloreado de grafos
- Recubrimiento de conjuntos
- Planificación de tareas en una máquina con tiempos de ”set-upz fechas de entrega
- Planificación de tareas en varias máquinas
- Problemas de mochila multidimensional
- Asignación cuadrática
- Programación entera no-lineal

En los Algoritmos Meméticos se utiliza el término de agentes en lugar de individuos ya que se consideran una extensión de los segundos. Tanto la selección como la actualización (reemplazo), son procesos puramente competitivos. La reproducción es la encargada de crear nuevos agentes (cooperación). Aunque puede aplicarse una gran variedad de operadores de reproducción, existen básicamente dos: Recombinación y Mutación.

- **Recombinación**: Realiza el proceso de cooperación. Crea nuevos agentes utilizando principalmente la información extraída de los agentes recombinados. Se suele hablar de combinación inteligente de información.

- **Mutación**: Permite incluir información externa creando nuevos agentes mediante modificación parcial del agente mutado.



El pseudocodigo del algoritmo se describe como:

```
Entrada: Instancia I de un problema P
Salida: La solucion
Inicio
    Genera poblacion inicial
        Para j de 1:popsize hacer
            Sea ind GenerarSoluconHeuristica I
            sea pop(j) MejoraLocal (ind, I)
            Finpara
            Repetir bucle generacional
        Seleccion
            Sea criadores seleccionarDePoblacion (pop)
        Reproduccion segmentada
            sea auxpop(0) pop
            para j de 1:op hacer
                sea auxpop(j) AplicarOperados op(j)
                auxpop([j,i], I)
                finpara
                Sea newpop auxpop(op)
            Reemplazo
                sea pop ActualizarProblacion (pop, newpop)
            Comprobar convergencia
                    si Convergencia (pop) entonces
                    sea pop  RefrescarPoblacion (pop, I)
                    finsi
                    hasta CriterioTerminacion (pop, I)
                devolver Mejor (pop, I)
```

3.2.1. Concurrencia y paralelismo

Hay que destacar una de las características que favorecen el uso de algoritmos memeticos, pues la creciente disponibilidad de sistemas de computación concurrente, generalmente basada en clusters permite la posibilidad de paralelizar o realizar este algoritmo de forma concurrente debido a que sus características se prestan para ello. Pues los algoritmos memeticos se adaptan muy bien a la paralelización.

3.3. Cuckoo Search

El algoritmo Cuckoo Search es un algoritmo evolutivo usado extensivamente para resolver problemas de optimización. Es muy efectivo para resolver este tipo de problemas pues hace uso de caminatas aleatorias a través de Vuelos de Lévy para abarcar y mantener el balance en todo el espacio de soluciones. Este algoritmo está basado en el comportamiento "parasitorio"del pájaro cuckoo al poner sus huevos. El pájaro cuckoo evita la creación de sus propios nidos por lo cual busca los nidos de otras especies de aves para usar como huéspedes para sus propios huevos, a costa de los huevos del huésped a menos que este descubra el engaño y se deshaga de los huevos del cuckoo o migre a un nido nuevo.

Podemos decir que los huevos del pájaro cuckoo representan nuevas soluciones a un sistema ya establecido, por lo que este algoritmo busca optimizar las situaciones donde estas soluciones funcionen mejor y puedan

reemplazar a las anteriormente obtenidas (huevos del huésped). El algoritmo se resume en los siguientes pasos:

- Cada cuckoo deposita un huevo en un nido seleccionado al azar.
- Los mejores nidos con los mejores huevos sobrevivirán y formaran la siguiente generación.
- Existe una cantidad fija de nidos disponibles y cada huevo tiene una probabilidad $P_a \epsilon [0, 1]$. De ser descubierto, en cuyo caso la madre abandona el nido y crea uno nuevo.

Antes de explorar más a fondo y de manera computacional este algoritmo es importante comprender el concepto de Vuelos de Lévy. En la naturaleza, pájaros e insectos buscan su comida en trayectorias rectas las cuales abruptamente pueden presentar un cambio de dirección. Este tipo de trayectorias con giros aleatorios pueden ser descritos mediante los Vuelos de Lévy, cuya longitud de desplazamiento se extrae de una distribución de Lévy que tiene una varianza y una media infinitas y están dados por la siguiente ecuación:

$$x_i^{t+1} = x_i^t + \alpha Levy(s, \lambda) \tag{1}$$

donde x_i^{t+1} es la siguiente posición, x_i^t es la posición actual, α es un factor de escalamiento del tamaño de los pasos, s es el tamaño de los pasos y λ es un factor de la distribución de Lévy.

El pseudocódigo que describe este algoritmo:

```
1
2  Inicio
3      Funcion objetivo f(x), x=(x_1,...x_d)
4      Generar poblacion inicial de n nidos x_i (i=1,2,...,n)
5  While(t<MaximaGeneracion) o (Criterio de finalizar)
6       Obtener nuevo cuckoo mediante vuelo de levy
7       Evaluar su calidad / fitness F_i
8  Elegir nuevo nido entre n (ejemplo j) aleatoriamente
9  Si (F_i >F_j)
10      Remplazar j por la nueva solucion
11 Fin
12      Una fraccion (P_a) de los peores nidos son abandonados
13      Y nuevos nidos son construidos
14      Mantener mejores soluciones
15      Ordenar las soluciones y encontrar la mejor
16 Fin mientras
17 Fin
```

Observamos las características del computo evolutivo en la optimización de las soluciones pues basados en las características propias y del entorno al que son expuestos los huevos pueden perecer, dar paso a la siguiente generación y mutar sus características para para incrementar sus posibilidades de incubación. Por otra parte la naturaleza bio-inspirada de este algoritmo es obvia.

3.3.1. Concurrencia

Este algoritmo (o una variante del mismo) es un buen candidato para ser desarrollado a través del computo concurrente pues si se tienen muchos pájaros cuckoo (y una cantidad suficiente y aún más grande de posibles huéspedes) la concurrencia ayudaría en la simulación de los diferentes vuelos de Lévy y procesos de incubación los cuales en principio son independientes pero es posible que algún cuckoo quisiese anidar donde otro ya lo ha hecho.

3.4. Problema de las Ocho Reinas

El problema de las n-reinas es uno de los problemas computacionales más conocidos en la actualidad . Este problema trata de buscar en un tablero de ajedrez de minesion n , en que posición de dicho tablero habría que colocar cada una de las n reinas , de tal manera , que una vez que estén colocadas , no se amenacen entre ellas . Este problema , es una generalización del problema del problema de las 8-reinas , en el que hay que

colocar ocho reinas en un tablero de dimensiones 8x8 de manera que no se amenacen entre sí . la inteligencia artificial clásica , resuelve este problema , utilizando una forma constructiva e incremental : en primer lugar se coloca la primera reina , a continuación , la siguiente reina comprobando que no se amenacen entre sí así de forma consecuente un enfoque evolutivo , resuelve el problema utilizando un planteamiento diferente . para poder resolver este mismo problema utilizando un algoritmo evolutivo , se definirá en el probelma , los diferentes elementos que componen un algoritmo bioinspirado

- INICIALIZACIÓN : La inicialización del mismo puede consistir en la generación automática de n posiciones , que serán las posiciones en las que se colocaran de inicio las n reinas , de tal manera que a partir de dicha colocación , comience a ejecutarse el algoritmo.
- POBLACIÓN : Para el problema de las n-reinas , la población en este caso se puede definir como un total de cien individuos , de manera que se tendrán en cuenta cien posibles soluciones , débilmente codificadas y sobre las cuales se llevará a cabo el proceso de selección.
- REPRESENTACIÓN :Una configuración del tablero , esto es , una solución posible dentro del espacio de soluciones o espacio fenotípico , se puede codificar como un vector de ocho posiciones , donde cada posición se refiere dentro de una fila a la columna donde esta situada la reina , siendo la combinación p1=[1,23,4,5,6,7,8] la configuración en la que las ocho reinas , en este caso , se colocan en la diagonal principal.
- FUNCION FITNESS :La funcion fitness es una función que medirá la calidad o la aptitud de las diferentes soluciones .Se puede establecer una función fitness en la que para cada solución , la función de evaluación represente el número de reinas que se amenazan entre sí , De este modo , el objetivo del algoritmo será minimizar dicha función para encontrar una solución cuyo valor fitness sea cero.

  $A = (i, j)/i - R(i) = j - R(j) \wedge i \neq j \vee i = R(i) \vee i \neq j$
- MECANISMOS DE SELECCION O SUPERVIVENCIA :Como mecanismo de selección de lso progenitores , se ha definido que se elegirán los dos mejores , de entre cinco seleccionados aleatoriamente , por lo que tal y como se describió anteriormente , este procedimiento es estocástico y aunque lo normal es que se seleccionen los más aptos un individuo con menos capacitación de adaptación también podrá ser elegido para perpetuar la especie.
- OPERADORES DE VARIACION:Para este ejemplo en concreto . ha de definirse el resultado de la combinación y la mutación en los individuos . En este caso , se ha definido que la probabilidad de combinación es 1 y esta consistirá en cortar y pegar parte del material genético a los progenitores , que se seleccionará aleatoriamente , en el genotipo de los progenitores . Por su parte , la probabilidad de mutación se define como 0.8 y consistirá en el intercambio de valores en la representación de la solución.
- REMPLAZO O ELIMINACION DEL MAS DEBIL: Dado que el tamaño de la población es constante , la descendencia producida durante el proceso de recombinación , hace necesaria la eliminación de dos individuos de los n=2 por los que esta formada la población una vez que se d=genera descendencia . El mecanismo de eliminación del mas débil para este ejemplo , radica en eliminar aquellas soluciones menos óptimas , esto es , aquellas que tienen un mayor número de reinas que se amenazan entre sí.
- CONDICIONDE TERMINACION:La condición de terminación se fija de tal manera que se asegure que el algoritmo terminará en este caso se define una condición de terminación que el algoritmo pare una vez que se haya encontrado la solución o bien se hayan producido 100000 evaluaciones fitness a lo largo del proceso .

  Se plantea el uso de métodos en paralelo o concurrente al tener varias reinas e irlas acomodando por pares

4. Selección de algoritmo

Se seleccionó el siguiente algoritmo para el cual haremos una descripción a profundidad y una explicación sobre la aplicación del algoritmo en una colonia de abejas para encontrar fuentes de alimento que utilicen

distintos tipos de abejas que se definirán a continuación. También se analizará brevemente que tan factible es la concurrencia en este algoritmo y se explicará a detalle cada componente que el algoritmo necesita.

Cabe destacar que este algoritmo se seleccionó debido a su clara aplicación e implemencación en concurrencia, además que la ejemplificación de este tipo de algoritmo resulta muy atractivo pues al realizar un algoritmo que pueda explicar o intentar replicar un fenómeno de la naturaleza tan aparentemente impredecible nos ha resultado fascinante. Sin mencionar que este tipo de aplicación lo podríamos ejemplificar también para otros escenarios como compradores, etc.

En contraste con el resto de los algoritmos investigados, la colonia de abejas nos permite ejemplificar lo concurrencia por grupos de abejas (lo cual detallaremos más adelante), pero a diferencia de los algoritmos memeticos por ejemplo era aquel algoritmo que nos permitía hacer una buena búsqueda local, sin embargo para la aplicación de la colonia de abejas una búsqueda local resultaría contraproducente al reducir nuestro espacio de búsqueda llegaría un punto en el que las fuentes de alimento son muy pocas. Por lo tanto un algoritmo memetico no nos sirve para el tipo de exploración que buscamos.

Una de las ventajas de este algoritmo a diferencia del resto es el número reducido de parámetros que necesita: número de soluciones (fuentes de alimento), el número de abejas empleadas y observadoras, el número total de ciclos (iteraciones) del algoritmo y el número de ciclos que una solución puede permanecer sin ser mejorada antes de ser reemplazada.

4.1. Algorítmo de la colonia artificial de abejas

El algoritmo de la colonia de abejas es un algoritmo de enjambre basado en la meta-heurística introducida por el Dr. Dervis Karaboga en 2005 para optimizar problemas numéricos. Fue inspirado por el inteligente comportamiento de búsqueda de las abejas de míel.

El modelo consta de tres componentes esenciales: abejas recolectoras empleadas, desempleadas y fuentes de alimentos. Los dos primeros componentes, abejas recolectoras empleadas y desempleadas buscan fuentes de alimentos, que es el tercer componente, cerca de su colmena. El modelo también define dos modos principales de comportamiento que son necesarios para la auto-organización y la inteligencia colectiva: el reclutamiento de recolectores de alimentos a fuentes de alimentos que resulta en una retroalimentación positiva y el abandono de fuentes pobres por parte de los recolectores de alimentos, lo que provoca una retroalimentación negativa.

EN ABC (Artificial Bee Colony) una colonia de abejas recolectoras artificiales (agentes) busca fuentes de alimentos artificiales buenas (buenas soluciones para un problema determinado). Para aplicar ABC, el problema de optimización considerado se convierte primero en el problema de encontrar el mejor vector de parámetros que minimice una función objetivo. Luego, las abejas artificiales descubren aleatoriamente una población de vectores de solución inicial y luego los mejoran iterativamente empleando las estrategias: avanzar hacia mejores soluciones por medio de un mecanismo de búsqueda de vecinos mientras abandonan las soluciones pobres.

El algorítmo se emplea para resolver problemas de optimización de la forma:

$$minZ = f(\vec{x}), \vec{x} = (x_1, x_2, \ldots, x_i, \ldots, x_{n-1}, x_n) \in \mathbb{R}^n$$

donde

$$l_i \leq x_i \leq u_i, in = 1, \ldots, n$$

$$\text{s.a. } g_j(\vec{x}) \leq 0, \text{ para } j = 1, \ldots, p$$
$$h_j(\vec{x}) = 0 \text{ para } j = p+1, \ldots, q$$

$f(\vec{x})$ es definido en un espacio de búsqueda S el cual es un rectángulo n-dimensional en $\mathbb{R}^n (S \in \mathbb{R}^n)$ donde el dominio de la variables esta limitado por las cotas inferior y superior.

4.2. Meta-heurística

En ABC, la colonia de abejas artificiales contiene tres grupos de abejas: abejas empleadas asociadas con fuentes de alimento específicas, abejas espectadoras que observan la danza de las abejas empleadas dentro de la colmena para elegir una fuente de alimento y abejas exploradoras que buscan fuentes de alimento al azar. Tanto los espectadores como los exploradores también se denominan abejas desempleadas.

Inicialmente, todas las posiciones de las fuentes de alimento son descubiertas por las abejas exploradoras. A partir de entonces, el néctar de las fuentes alimenticias es explotado por abejas empleadas y abejas espectadoras, y esta explotación continua finalmente hará que se agoten.

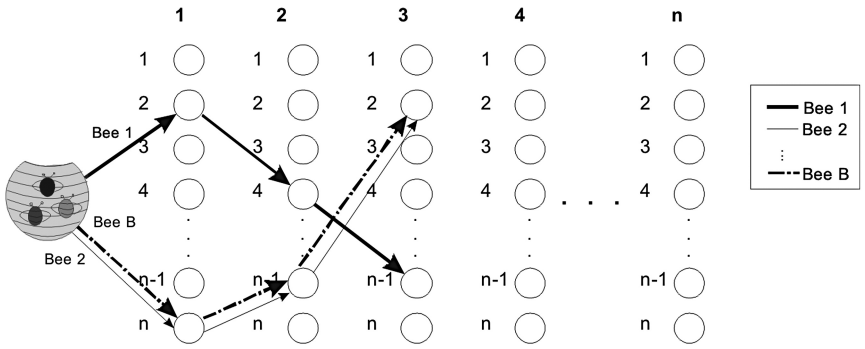
Entonces, la abeja empleada que estaba explotando la fuente de alimento agotada se convierte nuevamente en una abeja exploradora en busca de más fuentes de alimento. En otras palabras, la abeja empleada cuya fuente de alimento se ha agotado se convierte en una abeja exploradora. En ABC, la posición de una fuente de alimento representa una posible solución al problema y la cantidad de néctar de una fuente de alimento corresponde a la calidad (aptitud) de la solución asociada. El número de abejas empleadas es igual al número de fuentes de alimento (soluciones) ya que cada abeja empleada está asociada con una y solo una fuente de alimento.

El esquema general para este algorítmo es el siguiente:

```
Fase de Inicializacion

REPETIR
    Fase de abejas empleadas
    Fase de abejas espectadoras
    Fase de abejas exploradoras
    Memorizar la mejor solucion hasta el momento
HASTA(Ciclo = #Numero maximo de ciclos o tiempo maximo de CPU)
```

Donde de forma grafica el algoritmo se ve represetnado de la siguiente forma:



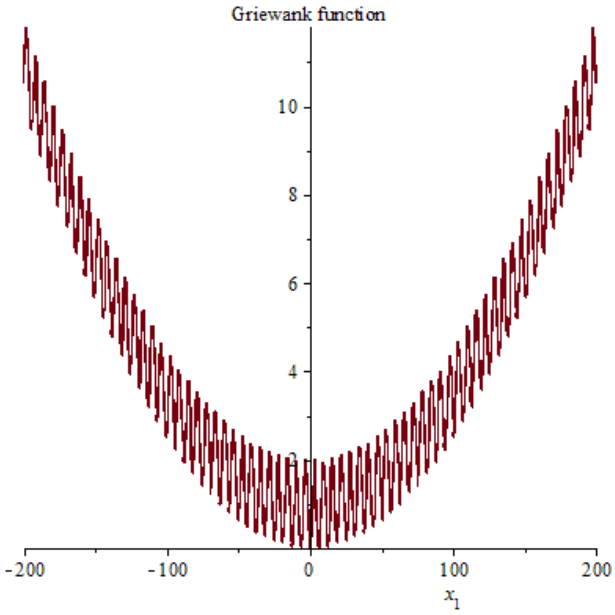
4.3. Concurrencia y paralelismo

Dada la naturaleza del algortítmo, existen varias combinaciones potenciales para hacer una implementación en paralelo. La mas destacable es en principal que se realizan múltiples búsquedas en el espacio de estados, este proceso puede ser implementado de forma en que las búsquedas se hagan en grupos de abejas de búsqueda, y estos grupos de búsquedas pueden ser realizados en paralelo de forma que cada grupo sea llevado acabo en cada procesador del equipo, por este motivo este algoritmo es una buena propuesta para realizar una implementación en paralelo.

5. Implementación

De los algorítmos pressentados en la sección anterior, el que se decidió implementar fue el algorítmo de Artificial Bee Colony (ABC), pues su desarrollo con concurrencia es visible, y tiene aplicaciones vastante diversas (en general es aplicable la mayoría de los problemas de optimización).

Figura 1: Ejemplo de la función Griewhank de primer Orden



En este caso se realizara la implementación del algoritmo para encontrar los mínimos locales de la función no convexa "griewank":

Para realizar la implementación hace falta desarrollar el algorítmo.

5.1. Algorítmo

Recordando de la sección anterior, el esquema general para este algorítmo es el siguiente:

```
Fase de Inicializacion

REPETIR
    Fase de abejas empleadas
    Fase de abejas espectadoras
    Fase de abejas exploradoras
    Memorizar la mejor solucion hasta el momento
HASTA(Ciclo = #Numero maximo de ciclos o tiempo maximo de CPU)
```

Donde cada fase se explica a continuación:

- **Fase de inicialización.** Todos los véctores de la población de las fuentes de alimento $\vec{x_m}$'s, son inicializados ($m : 1, \ldots, SN$: tamaño de la ploblación) por las abejas de búsqueda y se definen los parámetros de control. Dado que cada fuente de alimento $\vec{x_m}$ es una solución al problema de optimización, cada vector $\vec{x_m}$ tiene n variables, $(x_{mi}, i = 1, \ldots, n)$ los cuales son los que serán optimizados para minimizar la función de costo. La siguiente definición podría ser utilizada para inicializar el algorítmo:

$$x_{mi} = l_i + rand(0,1) * (u_i - l_i)$$

  donde l_i y u_i son las fronteras del parámetro x_{im}.

- **Abejas Empleadas.** Las abejas empleadas buscan por nuevas fuentes de alimento ($\vec{v_m}$) teniendo mas néctar dentro de los vecinos de a fuente de alimentos ($\vec{x_m}$) en su memoria. Estas encuentran en una fuente de alimento y evalúan su probabilidad (fitness). Se puede determinar la fuente de alimento vecina $\vec{v_m}$ con la siguiente ecuación:

$$v_{mi} = x_{mi} + \phi_{mi}(x_{mi} - x_{ki})$$

donde $\vec{x_k}$ es una fuente de alimento elegida de manera aleatoria, i es un índice elegido aleatoriamente y ϕ_{mi} es un numero aleatorio en $[-a, a]$. Después de producir nuevas fuentes de alimento $\vec{v_m}$, su función de 'fitness' es calculada y una selección greedy es aplicada entre $\vec{v_m}$ y $\vec{x_m}$. La función de fitness $fit_m(\vec{x_m})$ puede ser calculada para problemas de minimización como la siguiente formula:

$$fit_m(\vec{x_m}) = \begin{cases} \frac{1}{1=f_m(\vec{x_m})} & \text{si } f_m(\vec{x_m}) \geq 0 \\ 1 + abs(f_m(\vec{x_m}) & \text{si } f_m(\vec{x_m}) < 0 \end{cases}$$

donde $f_m(\vec{x_m})$ es el valor de la función objetivo en la solución $\vec{x_m}$.

- **Abejas Espectadoras.** Las abejas desempleadas consisten en 2 grupos de abejas: las abejas espectadoras y las abejas exploradoras. Las abejas empleadoras comparte la información de la fuente de alimento con las abejas espectadoras esperando en la colmena y después las abejas observadoras probabilisticamente escogen sus fuentes de alimentos dependiendo de esta información. En ABC las abejas buscadoras escogen una fuente de alimento dependiendo de las probabilidades calculadas utilizando los valores de fitness dados por las abejas empleadoras. Con este propósito una técnica de selección basada en el fitness puede ser empleada.

  El valor de probabilidad p_m donde cada $\vec{x_m}$ es elegida por una abeja exploradora puede ser calculada utilizando la siguiente expresión:

  $$P_m = \frac{fit_m(\vec{x_m})}{\Sigma_{m=1}^{SN} fit_m(\vec{x_m})}$$

  la cual corresponde a la ponderación de la función de fitness para el vector $\vec{x_m}$. Después de que la fuente de alimento $\vec{x_m}$ para una abeja espectadora haya sido elegida probabilisticamente, una fuente vecina $\vec{v_m}$ es determinada utilizando su ecuación de determinación y su valor de fitness es calculado. Igual que en la fase de las abejas empleadas, una selección greedy es aplicada entre $\vec{v_m}$ y $\vec{x_m}$. Por lo tanto, mas abejas observadoras son atraídas a fuentes mas abundantes y aparece un comportamiento de retroalimentación positiva.

- **Abejas Exploradoras.** Las abejas desempleadas que eligen sus fuentes de alimento al azar se denominan exploradoras. Las abejas empleadas cuyas soluciones no pueden mejorarse mediante un número predeterminado de ensayos, especificadas por el usuario del algoritmo ABC y denominadas aquí "límite.º çriterios de abandono", se convierten en exploradoras y sus soluciones son abandonadas. Luego, los exploradores convertidos comienzan a buscar nuevas soluciones, al azar. Por ejemplo, si la solución $\vec{x_m}$ ha sido abandonada, la nueva solucion descubierta por la exploradora $\vec{x_m}$ puede ser definida por la ecuacion $x_{mi} = l_i + rand(0,1) * (u_i - l_i)$. Por lo tanto, las fuentes que son inicialmente pobres o se han vuelto pobres por la explotación son abandonadas y surge retroalimentación negativa que balancea la positiva.

6. Documentación de código

Config.py

Este archivo se encarga de establecer los parámetros a utilizar en la resolución del problema. Para esto se importa el módulo `configparser`, el cual proporciona la clase `ConfigParser` que implementa un lenguaje de configuración básico que proporciona una estructura similar a la que se encuentra en los archivos INI de Microsoft Windows con la finalidad de escribir programas en Python que los usuarios finales pueden personalizar fácilmente. Observamos que las parámetros de configuración se establecen directamente en el documento `ABC.ini`.

En el programa empezamos por describir la clase `Config` la cual lee el archivo de configuraciones e implementa un objeto de la clase por cada parámetro establecido.

```
class Config:

    def __init__(_self, argv):
            config = configparser.ConfigParser()
            config.read(os.path.dirname(os.path.abspath(__file__))+'/ABC.ini')
            #####SETTINGS FILE######
            _self.OBJECTIVE_FUNCTION = _self.objFunctionSelector.get(config['DEFAULT']['ObjectiveFunction'], "Error")
            _self.NUMBER_OF_POPULATION = int(config['DEFAULT']['NumberOfPopulation'])
            _self.MAXIMUM_EVALUATION = int(config['DEFAULT']['MaximumEvaluation'])
            _self.LIMIT = int(config['DEFAULT']['Limit'])
            _self.FOOD_NUMBER = int(_self.NUMBER_OF_POPULATION / 2)
            _self.DIMENSION = int(config['DEFAULT']['Dimension'])
            _self.UPPER_BOUND = float(config['DEFAULT']['UpperBound'])
            _self.LOWER_BOUND = float(config['DEFAULT']['LowerBound'])
            _self.MINIMIZE = bool(config['DEFAULT']['Minimize'])
            _self.RUN_TIME = int(config['DEFAULT']['RunTime'])
            _self.SHOW_PROGRESS = bool(config['REPORT']['ShowProgress']=='True')
            _self.PRINT_PARAMETERS = bool(config['REPORT']['PrintParameters']=='True')
            _self.RUN_INFO = bool(config['REPORT']['RunInfo']=='True')
            _self.RUN_INFO_COMMANDLINE = bool(config['REPORT']['CommandLine']=='True')
            _self.SAVE_RESULTS = bool(config['REPORT']['SaveResults']=='True')
            _self.RESULT_REPORT_FILE_NAME = config['REPORT']['ResultReportFileName']
            _self.PARAMETER_REPORT_FILE_NAME = config['REPORT']['ParameterReportFileName']
            _self.RESULT_BY_CYCLE_FOLDER = config['REPORT']['ResultByCycleFolder']
            _self.OUTPUTS_FOLDER_NAME = str(config['REPORT']['OutputsFolderName'])
            _self.RANDOM_SEED = config['SEED']['RandomSeed'] == 'True'
            _self.SEED = int(config['SEED']['Seed'])
```

Los diferentes parámetros pueden ser consultados por los usuarios a través de las claves adecuadas:

```
try:
                opts, args = getopt.getopt(argv, 'hn:m:t:d:l:u:r:o:',
                                           ['help', 'np=', 'max_eval=','trial=', 'dim=', 'lower_bound=', 'upper_bound=', 'runtime=',
                                            'obj_fun=','output_folder=','file_name=','param_name=','res_cycle_folder=','show_functions'])
            except getopt.GetoptError:
                print('Usage: ABCAlgorithm.py -h or --help')
                sys.exit(2)
```

Y mediante los comandos correctos estos parámetros pueden ser modificados manualmente:

```
                elif opt in ('-n', '--np'):
                    _self.NUMBER_OF_POPULATION = int(arg)
                elif opt in ('-m', '--max_eval'):
                    _self.MAXIMUM_EVALUATION = int(arg)
                elif opt in ('-d', '--dim'):
                    _self.DIMENSION = int(arg)
                elif opt in ('-t', '--trial'):
                    _self.LIMIT = int(arg)
                elif opt in ('-l', '--lower_bound'):
                    _self.LOWER_BOUND = float(arg)
                elif opt in ('-u', '--upper_bound'):
                    _self.UPPER_BOUND = float(arg)
                elif opt in ('-r', '--runtime'):
                    _self.RUN_TIME = int(arg)
                elif opt in ('-o', '--obj_fun'):
                    _self.OBJECTIVE_FUNCTION = _self.objFunctionSelector.get(arg, "sphere")
                elif opt in ('--output_folder'):
                    _self.OUTPUTS_FOLDER_NAME = arg
                elif opt in ('--param_name'):
                    _self.PARAMETER_REPORT_FILE_NAME = arg
                elif opt in ('--file_name'):
                    _self.RESULT_REPORT_FILE_NAME = arg
                elif opt in ('--res_cycle_folder'):
                    _self.RESULT_BY_CYCLE_FOLDER = arg
```

También se incluye la opción de consulta de las funciones implementadas de `deap.benchmarks`. En la siguiente sección se hablará de dicho módulo y su importancia en el programa.

*ABC.py

La primero a lo que hay que prestar atención a esta sección son las librerías importadas de las cuales destacan 2. Por una parte notamos que se importa `progressbar` la cual gestiona el progreso actual mediante el formato de una barra de progreso de texto para mostrar el progreso de una operación de larga duración, lo que proporciona una señal visual de que el procesamiento está en marcha.

Por otra parte notamos el módulo `deap.benchmarks` sobre el cuál recae el peso matemático de la resolución al algoritmo. `DEAP` es un marco de cálculo evolutivo para la creación rápida de prototipos y la prueba de ideas. Busca hacer que los algoritmos sean explícitos y las estructuras de datos transparentes. Funciona en perfecta armonía con mecanismos de paralelización como multiprocesamiento.

En el código trabajamos sobre la clase `ABC`, cuyos objetos: `foods, f, fitness, trial, prob, solution, globalParams` los recibe como arreglos de `numpy` del archivo de configuraciones anteriormente descrito. También se implementan nuevos objetos iniciados en cero: `globalTime, evalCount, cycle, experimentID` y una lista vacía `globalOpts`.

Notamos que en el archivo `.ini` donde establecemos la configuraciones podemos hacer que la barra de progreso no se visualice. Sin embargo no podremos salir de un proceso hasta que la función `calculate_function` haya terminado de ejecutarse. Dicha función también manda un error si los límites tienen sentido.

```
    def calculate_function(_self, sol):
        try:
            if (_self.conf.SHOW_PROGRESS):
                _self.progressbar.update(_self.evalCount)
            return _self.conf.OBJECTIVE_FUNCTION(sol)

        except ValueError as err:
            print(
                "An exception occured: Upper and Lower Bounds might be wrong. (" + str(err)
    + " in calculate_function)")
            sys.exit()
```

Posteriormente implementamos la función de fitness para encontrar los mínimos y utilizada por las abejas empleadas:

$$fit_m(\vec{x_m}) = \begin{cases} \frac{1}{1=f_m(\vec{x_m})} & \text{si } f_m(\vec{x_m}) \geq 0 \\ 1 + abs(f_m(\vec{x_m}) & \text{si } f_m(\vec{x_m}) < 0 \end{cases}$$

donde $f_m(\vec{x_m})$ es el valor de la función objetivo en la solución $\vec{x_m}$.

```
    def calculate_fitness(_self, fun):
        _self.increase_eval()
        if fun >= 0:
            result = 1 / (fun + 1)
        else:
            result = 1 + abs(fun)
        return result
```

Esta función itera sobre el parámetro `increase_eval` el cuál aumenta su valor en 1 por iteración hasta alcanzar la condición de alto establecida en los parámetro:

```
    def increase_eval(_self):
        _self.evalCount += 1

    def stopping_condition(_self):
        status = bool(_self.evalCount >= _self.conf.MAXIMUM_EVALUATION)
        if(_self.conf.SHOW_PROGRESS):
          if(status == True and not( _self.progressbar._finished )):
               _self.progressbar.finish()
        return status
```

Posteriormente tenemos la función `memorize_best_source`, la cual tiene el parámetro boleano `MINIMIZE` el cuál indica si estamos buscando un mínimo o un máximo. En cualquier caso, si la función de fitness ha encontrado un punto mínimo (o en caso de que `MINIMIZE = False`, un máximo) se encargará de guardar este punto como el nuevo punto fuente.

```
    def memorize_best_source(_self):
        for i in range(_self.conf.FOOD_NUMBER):
            if (_self.f[i] < _self.globalOpt and _self.conf.MINIMIZE == True) or (_self.f[i]
     >= _self.globalOpt and _self.conf.MINIMIZE == False):
                _self.globalOpt = np.copy(_self.f[i])
                _self.globalParams = np.copy(_self.foods[i][:])
```

Finalmente antes de implementar a las abejas tenemos las funciones de inicialización donde todos los véctores de la población de las fuentes de alimento $\vec{x_m}$'s, son inicializados ($m : 1, \ldots, SN$: tamaño de la ploblación) por las abejas de búsqueda y se definen los parámetros de control. Dado que cada fuente de alimento $\vec{x_m}$ es una solución al problema de optimización, cada vector $\vec{x_m}$ tiene n variables, $(x_{mi}, i = 1, \ldots, n)$ los cuales son los que serán optimizados para minimizar la función de costo.

```
    def init(_self, index):
        if (not (_self.stopping_condition())):
            for i in range(_self.conf.DIMENSION):
                _self.foods[index][i] = random.random() * (_self.conf.UPPER_BOUND - _self.
    conf.LOWER_BOUND) + _self.conf.LOWER_BOUND
            _self.solution = np.copy(_self.foods[index][:])
            _self.f[index] = _self.calculate_function(_self.solution)[0]
            _self.fitness[index] = _self.calculate_fitness(_self.f[index])
            _self.trial[index] = 0

    def initial(_self):
        for i in range(_self.conf.FOOD_NUMBER):
            _self.init(i)
        _self.globalOpt = np.copy(_self.f[0])
        _self.globalParams = np.copy(_self.foods[0][:])
```

Ahora pasamos a la implementación de las abejas, empezando por las abejas empleadas. Mientras los parámetros de búsqueda: haber encontrado una fuente suficientemente buena de comida y un número máximo de iteraciones no se hayan alcanzado. Estas abejas buscan por nuevas fuentes de alimento ($\vec{v_m}$) teniendo mas néctar dentro de los vecinos de a fuente de alimentos ($\vec{x_m}$) en su memoria. Si las soluciones de los vecinos rebasan los límites establecidos entonces estos límites se actualizan y se pregunta por el fitness de los nuevos puntos.

```
    def send_employed_bees(_self):
        i = 0
        while (i < _self.conf.FOOD_NUMBER) and (not (_self.stopping_condition())):
            r = random.random()
            _self.param2change = (int)(r * _self.conf.DIMENSION)

            r = random.random()
            _self.neighbour = (int)(r * _self.conf.FOOD_NUMBER)
            while _self.neighbour == i:
                r = random.random()
                _self.neighbour = (int)(r * _self.conf.FOOD_NUMBER)
            _self.solution = np.copy(_self.foods[i][:])

            r = random.random()
            _self.solution[_self.param2change] = _self.foods[i][_self.param2change] + (
                        _self.foods[i][_self.param2change] - _self.foods[_self.neighbour][
    _self.param2change]) * (
                                                                r - 0.5) * 2

            if _self.solution[_self.param2change] < _self.conf.LOWER_BOUND:
                _self.solution[_self.param2change] = _self.conf.LOWER_BOUND
            if _self.solution[_self.param2change] > _self.conf.UPPER_BOUND:
                _self.solution[_self.param2change] = _self.conf.UPPER_BOUND
            _self.ObjValSol = _self.calculate_function(_self.solution)[0]
            _self.FitnessSol = _self.calculate_fitness(_self.ObjValSol)
            if (_self.FitnessSol > _self.fitness[i] and _self.conf.MINIMIZE == True) or (
    _self.FitnessSol <= _self.fitness[i] and _self.conf.MINIMIZE == False):
                _self.trial[i] = 0
                _self.foods[i][:] = np.copy(_self.solution)
                _self.f[i] = _self.ObjValSol
                _self.fitness[i] = _self.FitnessSol
```

```
            else:
                _self.trial[i] = _self.trial[i] + 1
            i += 1
```

Las abejas empleadoras comparte la información de la fuente de alimento con las abejas espectadoras esperando en la colmena y despuées las abejas observadoras probabilísticamente escogen sus fuentes de alimentos dependiendo de esta información, donde la probabilidad está dada por:

$$P_m = 0.9 * \frac{fit_m(\vec{x_m})}{max\{\Sigma_{m=1}^{SN} fit_m(\vec{x_m})\}} + 0.1 \tag{2}$$

```
    def calculate_probabilities(_self):
        maxfit = np.copy(max(_self.fitness))
        for i in range(_self.conf.FOOD_NUMBER):
            _self.prob[i] = (0.9 * (_self.fitness[i] / maxfit)) + 0.1

    def send_onlooker_bees(_self):
        i = 0
        t = 0
        while (t < _self.conf.FOOD_NUMBER) and (not (_self.stopping_condition())):
            r = random.random()
            if ((r < _self.prob[i] and _self.conf.MINIMIZE == True) or (r > _self.prob[i]
    and _self.conf.MINIMIZE == False)):
                t+=1
                r = random.random()
                _self.param2change = (int)(r * _self.conf.DIMENSION)
                r = random.random()
                _self.neighbour = (int)(r * _self.conf.FOOD_NUMBER)
                while _self.neighbour == i:
                    r = random.random()
                    _self.neighbour = (int)(r * _self.conf.FOOD_NUMBER)
                _self.solution = np.copy(_self.foods[i][:])

                r = random.random()
                _self.solution[_self.param2change] = _self.foods[i][_self.param2change] + (
                            _self.foods[i][_self.param2change] - _self.foods[_self.neighbour
    ][_self.param2change]) * (
                                                                    r - 0.5) * 2
                if _self.solution[_self.param2change] < _self.conf.LOWER_BOUND:
                    _self.solution[_self.param2change] = _self.conf.LOWER_BOUND
                if _self.solution[_self.param2change] > _self.conf.UPPER_BOUND:
                    _self.solution[_self.param2change] = _self.conf.UPPER_BOUND

                _self.ObjValSol = _self.calculate_function(_self.solution)[0]
                _self.FitnessSol = _self.calculate_fitness(_self.ObjValSol)
                if (_self.FitnessSol > _self.fitness[i] and _self.conf.MINIMIZE == True) or
    (_self.FitnessSol <= _self.fitness[i] and _self.conf.MINIMIZE == False):
                    _self.trial[i] = 0
                    _self.foods[i][:] = np.copy(_self.solution)
                    _self.f[i] = _self.ObjValSol
                    _self.fitness[i] = _self.FitnessSol
                else:
                    _self.trial[i] = _self.trial[i] + 1
            i += 1
            i = i % _self.conf.FOOD_NUMBER
```

Después de que la fuente de alimento $\vec{x_m}$ para una abeja espectadora haya sido elegida probabilisticamente, una fuente vecina $\vec{v_m}$ es determinada utilizando su ecuación de determinación y su valor de fitness es calculado. Igual que en la fase de las abejas empleadas, una selección greedy es aplicada entre $\vec{v_m}$ y $\vec{x_m}$. Por lo tanto, mas abejas observadoras son atraídas a fuentes mas abundantes y aparece un comportamiento de retroalimentación positiva.

Finalmente declaramos a las abejas scout las cuales evalúan si se ha obtenido una mejor fuente de alimentos, es decir aquellos puntos cuyas soluciones no pueden mejorarse mediante un número predeterminado

de ensayos, especificadas por el usuario del algoritmo ABC y denominadas aquí "límite". Se incrementa el iterador de ciclos y se etiqueta al proceso actual:

```
    def send_scout_bees(_self):
        if np.amax(_self.trial) >= _self.conf.LIMIT:
            _self.init(_self.trial.argmax(axis = 0))

    def increase_cycle(_self):
        _self.globalOpts.append(_self.globalOpt)
        _self.cycle += 1
    def setExperimentID(_self,run,t):
        _self.experimentID = t+"-"+str(run)
```

Reporter.py

Esta sección no forma parte de la solución ya que se encarga de imprimir los resultados en los formatos correspondientes. Mediante la clase `Reporter`, este programa obedece los parámetro establecido en el archivo de configuraciones para:

- Imprimir los parámetros establecidos mediante la función:

```
print_parameters(_self)
```

  Establecida inicialmente como `False`

- La impresión en la línea de comandos de los resultados de la búsqueda realizada por el algoritmo:

```
run_info(_self)
command_line_print(_self)
```

- Guardar los resultados en archivos CSV establecidos:

```
save_results(_self)
```

ABCAlgorithm.py

Esta sección importa todos los programas anteriores y contiene a la función `main(argv)` que inicializa la implementación de la *colonia artificial de abejas*. Podemos apreciar que esta función es la encargada de iniciar el parámetro que lleva cuenta del tiempo de ejecución. Declara un objeto de la clase ABC e implementa las diferentes funciones de este programa como métodos para resolver la búsqueda por soluciones fuente.

```
def main(argv):

    abcConf = Config.Config(argv)
    abcList = list()
    expT = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S").replace(" ","").replace(":"
    ,"")
    for run in range(abcConf.RUN_TIME):

        abc = ABC.ABC(abcConf)
        abc.setExperimentID(run,expT)
        start_time = time.time() * 1000
        abc.initial()
        abc.memorize_best_source()
        while(not(abc.stopping_condition())):
            abc.send_employed_bees()
            abc.calculate_probabilities()
            abc.send_onlooker_bees()
            abc.memorize_best_source()
            abc.send_scout_bees()
            abc.increase_cycle()

        abc.globalTime = time.time() * 1000 - start_time
        abcList.append(abc)
    Reporter(abcList)
```

Implementación de concurrencia

La implementación de concurrencia se realiza sobre la función `main` antes descrita. Podemos entender que de forma concurrente estamos resolviendo varios problemas de optimización con parámetros iniciales distintos para distintos archivos `ABC.ini` y no hacer concurrencia sobre las abejas mismas, pues como se ha visto anteriormente del programa `ABC.py` las soluciones para un problema evolucionan conforme a iteraciones sobre un ciclo y son fuertemente dependientes de las soluciones anteriores.

```
if name == 'main':
    start_time = time.time()

    processes = []
    for i in range(n):
        p = multiprocessing.Process(target=main(sys.argv[1:]), args=())
        processes.append(p)
        p.start()

    for process in processes:
        process.join()

    print('That took {} seconds'.format(time.time() - start_time))
```

7. Análisis de eficiencia

7.1. Resultados

Para el Análisis de eficiencia primero se obtiene el tiempo de ejecución de 20 iteraciones, de las cuales primero se realizaran 20 iteraciones realizadas de manera secuencial, y las próximas 20 se realizaran de forma concurrente con el modulo de multiprocessing como se menciono anteriormente.

Para esto, hay que recordar como se imprimen los resultados de una ejecución. Con los siguientes valores en la configuración:

```
#Population Number
NumberOfPopulation = 50
#Maximum Evaluation
MaximumEvaluation = 500000
Limit = 1500
LowerBound = -600
UpperBound = 600
RunTime = 5
Dimension = 5
ObjectiveFunction = griewank
Minimize = True
```

Se tienen los siguientes resultados:

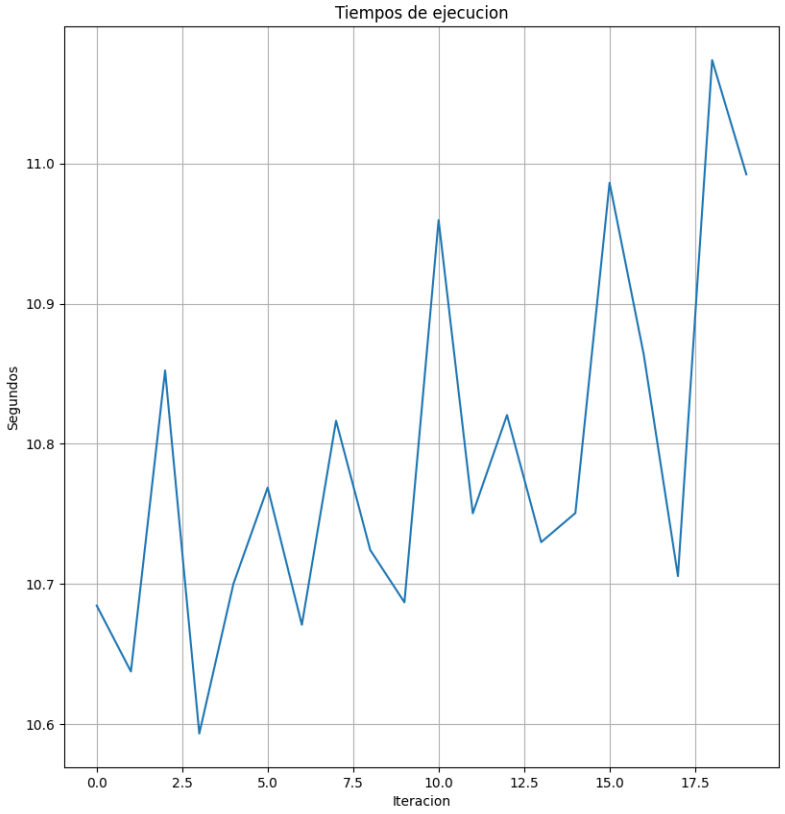
```
isaid@pop-os:~$ /usr/bin/python3 "/home/isaid/Downloads/ABCPython-master/ABCAlgorithm (Concurrent implementation).py"
100% (500000 of 500000) |##########################################################################################
100% (500000 of 500000) |##########################################################################################
100% (500000 of 500000) |##########################################################################################
100% (500000 of 500000) |##########################################################################################
100% (500000 of 500000) |##########################################################################################
2021-02-09143613-0 . run
Global Param[ 1 ]  -4.583884677755993e-09
Global Param[ 2 ]  -1.4109940500917464e-08
Global Param[ 3 ]  7.638149813218032e-09
Global Param[ 4 ]  -1.7606911970650763e-10
Global Param[ 5 ]  -1.5754052875097247e-08
2021-02-09143613-1 . run
Global Param[ 1 ]  -2.3823739332676553e-09
Global Param[ 2 ]  5.364847507302587e-10
Global Param[ 3 ]  1.3976171233048592e-08
Global Param[ 4 ]  -1.2562430097491694e-08
Global Param[ 5 ]  -3.6184859907597994e-09
2021-02-09143613-2 . run
Global Param[ 1 ]  -3.276107416216777e-09
Global Param[ 2 ]  2.000879376447065e-09
Global Param[ 3 ]  -1.710276451352946e-08
Global Param[ 4 ]  -3.2170684004258295e-09
Global Param[ 5 ]  2.228636687160438e-08
2021-02-09143613-3 . run
Global Param[ 1 ]  1.5242390636226872e-09
Global Param[ 2 ]  -1.1079769436596249e-08
Global Param[ 3 ]  -1.4957073453118521e-09
Global Param[ 4 ]  6.0472438801515145e-09
Global Param[ 5 ]  -1.9625384772876214e-08
2021-02-09143613-4 . run
Global Param[ 1 ]  9.747445750839903e-09
Global Param[ 2 ]  -7.125146515218007e-09
Global Param[ 3 ]  1.6047738867129838e-09
Global Param[ 4 ]  1.0868615399230311e-08
Global Param[ 5 ]  3.40210950731585e-09
2021-02-09143613-0 run:  0.0  Cycle:  9997  Time:  10246.051513671875
2021-02-09143613-1 run:  0.0  Cycle:  9998  Time:  10200.060302734375
2021-02-09143613-2 run:  0.0  Cycle:  9997  Time:  10203.029541015625
2021-02-09143613-3 run:  0.0  Cycle:  9997  Time:  10426.9931640625
2021-02-09143613-4 run:  0.0  Cycle:  9997  Time:  10221.40234375
Mean:  0.0  Std:  0.0  Median:  0.0
```

Donde primero se muestra el progreso de cada iteración conforme se van realizando, posteriormente se muestran los parámetros que minimizan la función objetivo para cada iteración y finalmente el tiempo de ejecución en ms de cada iteración.

7.2. Estadísticas del algoritmo

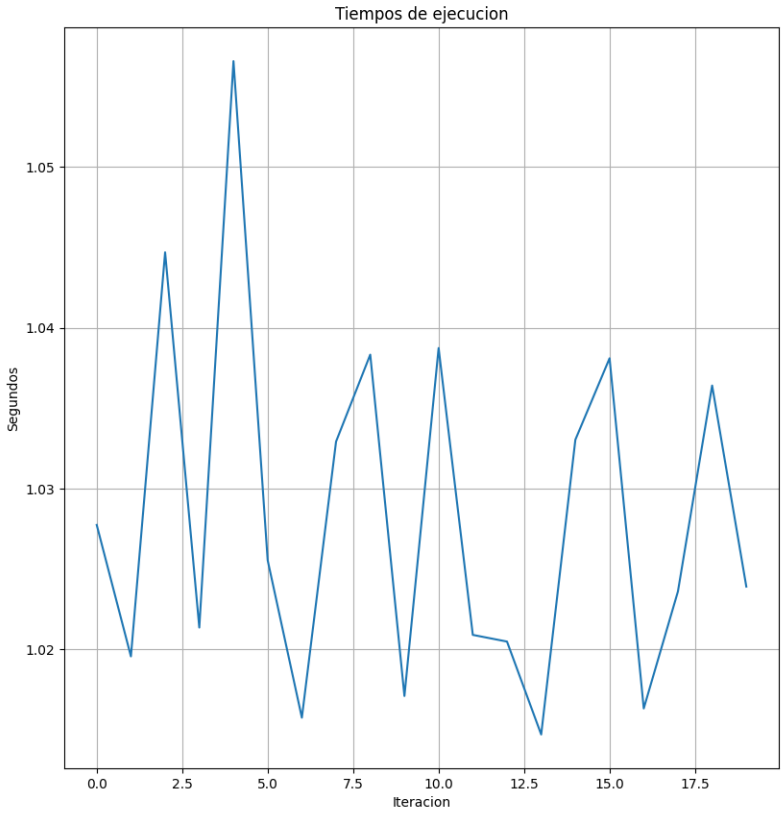
Para realizar las pruebas de concurrencia, nos limitamos a obtener tan solo los tiempos de ejecución de cada iteración.

Para estas pruebas se consideraron ejecuciones con 50 abejas y la función a optimizar en $\mathbb{R}^5$. Para 20 iteraciones en el modelo secuencial se tuvieron los siguientes tiempos de ejecución:



Donde los tiempos de ejecución para cada iteración se encuentran dentro del intervalo $[10.6, 11.1]$ segundos para una exploración (consistiendo de 50 abejas), y un tiempo de ejecución total de 250.85083746910095 segundos para las 20 iteraciones.

Posteriormente, para 20 iteraciones con el modelo implementado de manera concurrente, se tienen los siguientes tiempos de ejecución:



Donde se destaca que el tiempo ejecución individual de las iteraciones reduce considerablemente, donde ahora los tiempos de ejecución se encuentran en el intervalo $[1.1, 1.6]$, reduciendo el tiempo de ejecución en potencia de 10, tomando 20.910200834274292 segundos en realizar las 20 ejecuciones.

7.3. Speed up

Teniendo en cuenta la definición del índice de rendimiento conocido como speedup (o aceleración), el cual se calcula de la siguiente manera:

$$Speedup = \frac{tiempo\_secuencial}{tiempo\_concurrente}$$

El valor máximo del speed up en teoría es igual a la cantidad de procesadores, razón por la cual los resultados suelen compararse con la recta y = x. Si el speed up obtenido es superior al máximo teórico, suelen denominarse como speed up superlineal.

Con esto considerado, se tomo el promedio de la realización de 10 ejecuciones de 5 iteraciones en ambas implementaciones, y se obtuvieron los siguientes tiempos:

- Tiempo secuencial: 59.568 s.
- Tiempo Concurrente: 5.481 s.

Con esto se tiene que:

$$speed\_up = \frac{59.568}{5.481} = 10.86$$

En lo que respecta a la escalabilidad del modelo, hay que recordar que una búsqueda local resultaría contraproducente al reducir nuestro espacio de búsqueda llegaría un punto en el que las fuentes de alimento son muy pocas, además de que este algoritmo solamente requiere la imputación de la dimensión en espacio de soluciones, un tiempo máximo de evaluaciones y un numero de abejas.

De los parámetros definidos anteriormente, los que afectan la ejecución del algoritmo son el espacio de estados y el numero máximo de evaluaciones (intuitivamente pues este es el parámetro de paro). Por lo tanto, la escalabilidad del numero máximo de evaluaciones es lineal, donde $tiempo = \frac{MaximumEvaluation}{1000}$. Donde existe un cambio considerable es en la dimensión del espacio de soluciones. El crecimiento en tiempo a partir del numero de dimensiones parece tener un comportamiento polinomial pues se tuvieron los siguientes tiempos para las respectivas dimensiones:

1. $d = 1 \Rightarrow t \approx 8s$
2. $d = 10 \Rightarrow t \approx 14s$
3. $d = 100 \Rightarrow t \approx 65s$

Sin embargo, dados los requerimientos de la optimización requerida, se considera que difícilmente se buscaran obtener optimizaciones de funciones en dimensiones mayores a potencias de 100, por esto y lo mencionado anteriormente se concluye que la implementación es considerablemente escalable.

Referencias

[1] UNIVERSIDAD DISTRITAL FRANCISCO JOSE DE CALDAS, (YURI CRISTIAN BERNAL PEÑA) Algoritmos memeticos,2017

[2] M. Marelli. An adaptive Cuckoo search algorithm for optimisation, https://doi.org/10.1016/j.aci.2017.09.0011

[3] https://en.wikipedia.org/wiki/Cuckoo_search

[4] Joshi, A.S. Kulkarni, Omkar Kakandikar, Ganesh Nandedkar, Vilas. (2017). Cuckoo Search Optimization- A Review. Materials Today: Proceedings. 4. 7262-7269. 10.1016/j.matpr.2017.07.055.

[5] Dervis Karaboga (2010), Scholarpedia, 5(3):6915. doi:10.4249/scholarpedia.6915

[6] https://abc.erciyes.edu.tr/

[7] https://abcolony.github.io

[8] https://en.wikipedia.org/wiki/Griewank_function

[9] https://software.intel.com/content/www/us/en/develop/articles/predicting-and-measuring-parallel-performance.html#:~:text=Simply%20stated%2C%20speedup%20is%20the,6720%2F126.7%20%3D%2053.038).