UNIVERSIDAD DE GRANADA

METAHEAURÍSTICAS

Práctica Opcional
Algoritmo de Real Coding

Algoritmo de iones

Ignacio Cordón Castillo, 25352973G
nachocordon@correo.ugr.es

4° Doble Grado Matemáticas Informática
Grupo Prácticas Viernes

15 de julio de 2016

Índice

1. Introducción

Como sabemos, se trata de mejorar una metaheurística para minimizar las 20 primeras funciones de codificación real de la competición CEC2014, usando:

- Dimensión D valuada a 10 y 30.
- Número de evaluaciones de cada función en una ejecucion del algoritmo: 100000 en dimensión 10, 300000 en dimensión 30.
- 25 ejecuciones por función.
- Espacio de búsqueda de soluciones: $[-100, 100]^D$

Se compararán las medias obtenidas con estas premisas de ejecución para cada función. Además, debemos tener en cuenta que el mínimo de la función $i-$ésima es $100 \cdot i$. Cuando obtengamos los resultados, computarermos el error respecto a dicho mínimo que hemos obtenido, de manera que buscaremos que los resultados en media para cada una de las funciones se aproximen lo máximo posible a 0.

2. Estado del arte

La mayoría de técnicas para resolver problemas de optimización de funciones (las que nos ocupan son no lineales y muchas no diferenciables) son algoritmos poblaciones, un tipo de algoritmos estocásticos en los que tenemos una serie de soluciones a las que, aplicados una serie de operadores, van convergiendo a una posible solución del problema. Este tipo de algoritmos pueden ser divididos en tres subcategorías:

- *Algoritmos bioinspirados*.

  Basados mayormente en el modelado del comportamiento de enjambres o manadas de animales en la naturaleza. En este sentido, destaca PSO, algoritmo que imita el movimiento de las aves, o ACO (*Ant Colony Optimization*). Recientemente han surgido toda una serie de algoritmos bioinspirados: *Bat Algorithm*, *Firefly Algorithm*, *Cuckoo Search*, *Grey Wolf Optimizatier*
- *Algoritmos genéticos*. Los primeros estudios versaban sobre algoritmos genéticos con codificación real donde los investigadores se centraron durante mucho tiempo en encontrar nuevos operadores de cruce fructíferos. Fueron Rechenberg y Schwefel quienes propusieron en 1964 las estrategias de evolución, que se basaban en la auto-adaptabilidad de los parámetros. Nikolaus Hansen obtuvo en 1996 muy buenos resultados a este respecto modelando una adaptación mediante matrices de covarianza a través de su algoritmo CMAES.

  Posteriormente surgió la evolución diferencial, propuesta por Storm en 1997. Se trataba de un proceso estocástico por el que llevando una población de tuplas n-dimensionales, se iban mutando y recombinando para dar lugar a nuevas soluciones en un espacio de búsqueda. A este respecto, tenemos por ejemplo SaDE o JADE, algoritmos de Differential Evolution que auto-adaptan los parámetros a los espacios de búsqueda para generar soluciones. Históricamente, la potencia de los

algoritmos de Differential Evolution ha sido altísimo. De hecho una variante de este algoritmo, L-SHADE, resultó vencedora del reto completo (30 funciones, no 20) al que nos enfrentamos.

En este campo también sobresalen la *Programación Genética* y el *Biogeography-based Optimizer*

- *Algoritmos inspirados en la física*: la inspiración de estos algoritmos surge de leyes o fenómenos físicos. La diferencia respecto a los algoritmos bioinspirados es que la recombinación de soluciones se modela siguiendo ciertas leyes físicas, o inspirándose en ellas. Podemos citar dentro de esta subcategoría:

  - *Algoritmo de Optimización Magnética*: simula una serie de fuerzas electromagnéticas para desplazar las posiciones de las posibles soluciones en el espacio de búsqueda, ejerciendo las mejores soluciones mayor atracción hacia ellas.
  - *Algoritmo de Búsqueda Gravitacional*: se consideran una serie de masas que se atraen, siguiendo la ley Newtoniana del movimiento, en base a fuerzas modeladas a partir del valor que cada masa (o posible solución) produce en la función de *fitness*.
  - *Algoritmo de iones*: el algoritmo que nos ocupa. Se modelan las soluciones en el espacio a través de posciones que pueden tener carga positiva o negativa. Cargas de distinto signo se atraen, mientras que cargas de signo opuesto se repelen.

Al respecto del algoritmo que hemos escogido, el *algoritmo de iones* (IMO) sólo existe un trabajo publicado por Behzad Javidy, Abdolreza Hatamlou y Seyedali Mirjalili, publicado a 19 de Marzo de 2015. Los autores tratan de modelar de forma simplificada el comportamiento de los iones en el espacio y lo comparan con algoritmos genéticos, con Differential Evolution, con PSO y con algoritmos de Colonia de Hormigas sobre 10 funciones test, obteniendo mejores medias y mejores desviaciones típicas que los citados algoritmos para unos parámetros prefijados.

3. Descripción del algoritmo

El algoritmo se basa en el movimiento de los iones(soluciones) en el espacio. Se inicia una población de iones (igual número de cationes que de aniones) a soluciones aleatorias en el espacio. Se modelan dos fases en el movimiento de los iones:

- **Fase líquida**

  Para cada iteración en la que nos encontremos en fase líquida, efectuamos lo descrito a continuación.

  Se encuentra el mejor catión C_{best} de entre la población, en términos de *fitness*.

  Para el anión $i-$ésimo, A_i, y su componente $j-$ésima, se modela una fuerza de atracción:

$$F_{i,j} = \frac{1}{1 + e^{d_{i,j}}} \qquad d_{i,j} = \frac{-0.1}{|A_i[j] - C_{best}[j]|}$$

y se actualiza la componente del anión a:

$$A_i[j] = A_i[j] + F_{i,j} \cdot (C_{best}[j] - A_i[j])$$

Se efectúa un procedimiento análogo con los cationes, acercándolos al mejor anión.

La idea es buena, dado que si tenemos un catión (resp.anión) bueno, se modifican los aniones para que las componentes se empiecen a parecer a los de ese catión, con mayor grado de convergencia en las componentes más cercanas, y menor en las más lejanas.

Además, la convergencia puntos en el espacio debería estar garantizada, puesto que la sucesión $\left\{\frac{1}{2^n}\right\}$ converge a 0, y se tiene $\frac{1}{2} \leq F_{i,j} \leq 1$. Pasaremos a discutir esto en la siguiente sección.

- **Fase sólida**: Se modela una fase en la que una vez alcanzada la convergencia a una solución, podamos movernos a otros puntos del espacio.

  Efectuamos el siguiente proceso, que es una modificación del propuesto por los autores, puesto que en el proceso descrito en el artículo científico original se presentan algunas imprecisiones:

```
if  fitness(best(cations)) >= fitness(worst(cations)) and
    fitness(best(anions)) >= fitness(worst(anions))

    if (rand < prob_restart){
        random_restart(anions)
    }
    else{
        for i in {1,...length(anions)}{
            if (rand(0,1) > 0.5)
                anion[i] = anion[i] + rand(-1,1)*(best_old_cation)
            else
                anion[i] = anion[i] + rand(-1,1)*(best_cation)
        }
    }
    if (rand < prob_restart){
        random_restart(cations)
    }
    else{
        for i in {1,...length(cations)}{
            if (rand(0,1) > 0.5)
                cation[i] = cation[i] + rand(-1,1)*(best_old_anion)
            else
                cation[i] = cation[i] + rand(-1,1)*(best_anion)
        }
    }
```

La probabilidad de reinicio de soluciones propuesta por los autores en su *paper* original es del 0.05, aunque hemos hecho un pequeño ajuste de los parámetros del algoritmo, a fin de optimizarlos.

Cuando se reinician tanto los cationes como los aniones, se hace por supuesto a tuplas en el espacio de búsqueda: $[-100, 100]^D$

`best_old_cation` y `best_old_anion` representan los mejores iones de cada clase en la anterior ejecución del algoritmo, mientras `best_cation` y `best_anion` representan los mejores iones de cada clase en la actual ejecución del algoritmo. Cuando se recombinan los actuales iones, cationes con los mejores cationes, iones respectivamente, el objetivo es explorar otras zonas del espacio de búsqueda. La eficacia de este aspecto será una de las cuestiones a debatir en las mejoras del algoritmo.

Cuando no se reinician las soluciones, sino que se recombinan usando los mejores aniones y cationes de esta o anteriores ejecuciones, se observó que las soluciones no tenían porqué quedarse en el espacio de búsqueda, así que se implementó un sencillo procedimiento de normalización de soluciones, devolviéndolas a los límites de nuestro dominio:

```
for i in {1,...length(cations)} {
    for j in {1,...length(cation[i]}{
        if (cation[i][j] < -100) cation[i][j] = -100
        elsif (cation[i][j] > 100) cation[i][j] = 100
}

for i in {1,...length(anions)} {
    for j in {1,...length(anion[i]}{
        if (anion[i][j] < -100) anion[i][j] = -100
        elsif (anion[i][j] > 100) anion[i][j] = 100
    }
}
```

Asimismo, la condición que modelaban los autores originalmente para entrar en esta fase sólida era:

```
fitness(best(cations)) >= fitness(worst(cations))/2 and
fitness(best(anions)) >= fitness(worst(anions))/2
```

Pero usar dicha división entre 2 parecía algo ilógico. Ésta constituye una de las grandes imprecisiones que se comentaban anteriormente. Por ello se suprimió dicha división por 2. Más aún, ni siquiera sin esa división por 2 parecemos asegurar que abandonemos la fase líquida en algún momento. Esta será una de las cuestiones a debatir en las mejoras del algoritmo.

3.1. Equilibrio Diversidad Convergencia

Parece existir un problema con la convergencia a puntos del espacio, puesto que podríamos tener dos puntos del espacio que fueran óptimos locales en una cierta zona muy cercana, el mejor anión situado en uno de ellos, y el mejor catión en otro. Podríamos tener a todos los aniones excepto el mejor, en una zona lo suficientemente cercana al mejor catión; y a todos los cationes, excepto el mejor, en una zona lo suficientemente cercana al mejor anión. Se tendría que en cada iteración, todos los aniones se intercambian con los cationes, por la cercanía de las partículas, y todos los cationes se intercambian con los aniones, y nunca llegaríamos a tener la convergencia a un único punto.

Matemáticamente, sea f la función a minimizar en un espacio $[-100, 100]^D$, y sean x, y en dicho espacio. Supongamos:

$$\exists B(x, \delta_1), B(y, \delta_2)$$

verificándose que

$$B(x, \delta_1) \bigcap B(x, \delta_2) = \varnothing$$

f tiene un mínimo local en $B(x, \delta_1)$ y otro en $B(y, \delta_1)$

se cumple que $C_{best} \in B(x, \delta_1)$, $A_{best} \in B(x, \delta_2)$, y además $A_i \in B(x, \delta_1) \forall A_i \neq A_{best}$ y $C_i \in B(x, \delta_2) \forall C_i \neq C_{best}$, con $F_{i,j} \approx 1$. Podríamos entonces no tener convergencia nunca a un único punto, o incluso que fuese tan lenta que estuviésemos perdiendo un montón de evaluaciones con soluciones de *fitness* similar.

Es más, supongamos en lo que sigue que tenemos convergencia a un único punto. Aún así, el mecanismo para entrar a fase líquida no parece del todo certero, porque nunca podemos asegurar que se vaya cumplir que:

```
fitness(best(cations)) >= fitness(worst(cations))/2 and
fitness(best(anions)) >= fitness(worst(anions))/2
```

a pesar de haber alcanzado un óptimo local. Aún así, si este mecanismo fuese del todo certero, deberíamos buscar otro método para introducir mayor diversidad en la población de iones.

3.2. Comparación con las otras metaheaurísticas

Hemos obtenido los siguientes resultados, tras la implementación del algoritmo con las variaciones descritas, pero filedigno a lo expuesto por sus autores:

Cuadro 1: Resultados del *Algoritmo de iones*

	Dimensión 10	**Dimensión 30**
f1	19084554.669857	78738913.351420
f2	16536.801547	13697676.211767
f3	21065.221623	77800.939852
f4	58.764875	246.343748
f5	20.170120	20.468675
f6	7.539682	34.790689
f7	0.861159	1.102618
f8	28.934028	129.837295
f9	26.123286	145.730468
f10	1157.285877	4219.909911
f11	1144.876264	4881.285910
f12	0.661762	1.304201
f13	0.456650	0.440387
f14	0.635537	0.260939
f15	4.164066	51.414376
f16	3.390798	12.366670
f17	236940.300926	3390125.155789
f18	6522.070266	110104.005591
f19	8.886340	32.516194
f20	13892.000148	53412.013643

4. Propuestas de mejora del algoritmo

- **Optimización de parámetros**.

  A este efecto, hemos probado, para las 5 primeras funciones (debido a la falta de tiempo), los siguientes valores de tamaño de población de iones: $\{10, 20, 30, 40, 50\}$ y los siguientes valores de probabilidad de reinicio: $\{0.05, 0.1, 0.15, 0.2\}$

  obteniendo los mejores resultados con un tamaño poblacional de 50 iones, y una probabilidad de reinicio del 0.1, que corresponden a los resultados presentados en el punto número 3

- **Hibridación con búsqueda local**.

  Buscamos que el algoritmo tal cual nos proporcione la diversidad y la búsqueda local el factor de convergencia.

  Se programó una búsqueda local que no funcionaba comedidamente, que fundamentalmente, realizaba las siguientes operaciones:

```
applyLocalSearch(best_solution){
    current = best_solution

    for i in {1,2,...evals_ls} {
        direccion = [random(0,1) for i in {1...dimension}]
        norma = || direccion ||

        for j in {1...dimension} {
```

```
            current[j] += random(0, epsilon)*(direccion[j] / norma);
        }

        normalize(current);
        current.updateFitness();

        if (current.getFitness() <= best_solution.getFitness()){
            best_solution = current
        }
        else{
            current = best_solution
        }
        i++;
    }
}
```

donde se escogía un `epsilon = 0.25` y `evals_ls = 10*dimension`, es decir, se sumaba un vector de norma euclídea menor o igual a epsilon al vector mejor solución, y si encontrábamos mejora, aplicábamos el mismo procedimiento al vector mejorado, hasta agotar las $10 * D$ iteraciones.

Posteriormente, probamos a hibridar con *SolisWets*, *CMAES* y *Simplex*, búsquedas locales programadas en el software Realea de Daniel Molina distribuido bajo licencia GPL. La estrategia para hibridar fue introducir una búsqueda local en fase sólida sobre la mejor solución encontrada hasta el momento. Empleamos $10 \cdot D$ iteraciones en *Solis Wets*, en *Simplex* y en *CMAES*.

Los resultados obtenidos, han sido:

- **Solis Wets**

Cuadro 2: Resultados del *Algoritmo de iones + Solis Wets*

	D=10, IMO	D=10, IMO + SW	D=30, IMO	D=30, IMO + SW
f1	19084554.669857	363696.039099	78738913.351420	455502.264533
f2	16536.801547	1313.954155	13697676.211767	13127.874215
f3	21065.221623	18688.493110	77800.939852	71748.162238
f4	58.764875	24.202097	246.343748	64.979294
f5	20.170120	19.999690	20.468675	19.999692
f6	7.539682	7.402224	34.790689	34.666686
f7	0.861159	0.353630	1.102618	0.011314
f8	28.934028	26.187262	129.837295	124.369383
f9	26.123286	25.470889	145.730468	158.078285
f10	1157.285877	948.631408	4219.909911	3889.044913
f11	1144.876264	1066.564468	4881.285910	4295.850074
f12	0.661762	0.496767	1.304201	1.094158
f13	0.456650	0.311175	0.440387	0.432896
f14	0.635537	0.382150	0.260939	0.257654
f15	4.164066	3.857350	51.414376	59.799756
f16	3.390798	3.449191	12.366670	12.554226
f17	236940.300926	60438.604437	3390125.155789	46233.218822
f18	6522.070266	7802.026855	110104.005591	1599.150327
f19	8.886340	7.871310	32.516194	38.234215
f20	13892.000148	7092.191695	53412.013643	31560.507434

- **CMAES**

Cuadro 3: Resultados del *Algoritmo de iones + CMAES*

	D=10, IMO	D=10, IMO + CMAES	D=30, IMO	D=30, IMO + CMAES
f1	19084554.669857	7787409.210024	78738913.351420	603080.766525
f2	16536.801547	1622.048245	13697676.211767	9668.463875
f3	21065.221623	17145.712244	77800.939852	80051.795107
f4	58.764875	23.531004	246.343748	27.503332
f5	20.170120	19.999604	20.468675	19.999750
f6	7.539682	7.650586	34.790689	34.910613
f7	0.861159	0.783753	1.102618	0.007092
f8	28.934028	27.460799	129.837295	122.737655
f9	26.123286	22.127844	145.730468	151.949401
f10	1157.285877	1087.485644	4219.909911	3899.973468
f11	1144.876264	1070.067969	4881.285910	4053.327961
f12	0.661762	0.568838	1.304201	0.787965
f13	0.456650	0.310108	0.440387	0.387936
f14	0.635537	0.454480	0.260939	0.245177
f15	4.164066	3.668623	51.414376	57.776390
f16	3.390798	3.375496	12.366670	12.408639
f17	236940.300926	191239.202539	3390125.155789	53082.455802
f18	6522.070266	7052.486699	110104.005591	2039.587266
f19	8.886340	7.586558	32.516194	40.452787
f20	13892.000148	9850.321392	53412.013643	34683.954916

- **Simplex**

Cuadro 4: Resultados del *Algoritmo de iones + Simplex*

	D=10, IMO	D=10, IMO + Simplex	D=30, IMO	D=30, IMO + Simplex
f1	19084554.669857	1504465.672234	78738913.351420	4346086.095596
f2	16536.801547	12474.120134	13697676.211767	2561077.417506
f3	21065.221623	7833.146464	77800.939852	49242.768978
f4	58.764875	28.881786	246.343748	89.359609
f5	20.170120	20.074512	20.468675	20.095649
f6	7.539682	6.171463	34.790689	30.754850
f7	0.861159	0.571411	1.102618	0.062850
f8	28.934028	16.717241	129.837295	67.033718
f9	26.123286	25.795814	145.730468	150.276753
f10	1157.285877	685.986171	4219.909911	2185.469491
f11	1144.876264	1030.179614	4881.285910	3892.796287
f12	0.661762	0.431555	1.304201	0.501540
f13	0.456650	0.315380	0.440387	0.466117
f14	0.635537	0.393064	0.260939	0.264987
f15	4.164066	3.016509	51.414376	30.881912
f16	3.390798	3.327450	12.366670	12.234189
f17	236940.300926	209208.552698	3390125.155789	841071.802922
f18	6522.070266	7622.195964	110104.005591	1837.062837
f19	8.886340	4.769108	32.516194	19.797393
f20	13892.000148	5046.809383	53412.013643	41976.014409

- **Modelado de fuerzas de atracción y repulsión completas**:

  Hemos tratado de modelar todas las fuerzas de atracción-repulsión (no tan solo las de atracción hacia el mejor catión o anión, como proponían sus autores). Asimismo, las fuerzas que establecían los autores originales se basaban en la distancia en $\mathbb{R}$ de las proyecciones de cada coordenada de las soluciones al mejor catión/anión. Se ha intentado modelar estas fuerzas por medio de la calidad del fitness de la función a minimizar.

  Explicaremos el algoritmo de movimiento de los iones en el espacio con aniones. Para los cationes, el procedimiento es análogo, intercambiando el papel de aniones y cationes en lo que se pasa a explicar a continuación:

  Las fuerzas de repulsión entre aniones, se modelan, para un anión A_i dado como:

$$F_{i,j} = \frac{1}{2} \cdot \left\{ 1 - \frac{1}{1 + e^{d_{i,j}}} \right\} \qquad d_{i,j} = f(A_j) - f(A_i)$$

  Eso implica que si dos aniones tienen *fit* muy similar, apenas se producirá repulsión, mientras que si dos aniones tienen *fit* distintos, la fuerza de repulsión será de $\frac{1}{2}$, actualizándose las componentes de A_i a:

$$A_i = A_i + \sum_{j=1, j \neq i}^{j=|Aniones|} F_{i,j} \cdot (A_i - A_j)_{i=1...D}$$

Las fuerzas de atracción de los aniones al anión dado A_i se modelan como:

$$F_{i,j} = \frac{1}{2} \cdot \left\{ 1 - \frac{1}{1 + e^{d_{i,j}}} \right\} \qquad d_{i,j} = f(C_j) - f(A_i)$$

De esta forma, la fuerza de atracción es muy alta cuando el catión tiene un *fit* mucho mejor(menor) que el anión, y es más pequeña conforme el anión tenga un *fit* mucho más alto que el catión. y se actualiza A_i como sigue:

$$A_i = A_i + \sum_{j=1}^{j=|Cationes|} F_{i,j} \cdot (C_j - A_i)_{i=1...D}$$

Esta funcionalidad se ha implementado en:

`void updateLocations(vector<Solution>&anions, vector<Solution>cations)`

en el fichero `metaheauristic.cc`. El resto del algoritmo permanece igual que lo descrito hasta el momento, con una búsqueda local de tipo *Solis Wets* de $10 \cdot D$ iteraciones para la mejor solución encontrada hasta el momento en la fase sólida.

También hemos intentado modelar el mecanismo de reinicio de soluciones (`void redistribute(vector<Solution>&ions, vector<Solution>&bests)`), intentando mejorar la diversidad del mismo. LLevamos dos listas, una con los mejores aniones/cationes encontrados hasta el momento (cada vez que calculamos el mejor anión o catión). Introducimos también una pequeña mutación con probabilidad de 0.001 que hace que una componente de una posible solución se modifique a un número aleatorio entre -100 y 100 con dicha probabilidad. El procedimiento seguido en la nueva función de reinicio de soluciones ha sido, fundamentalmente:

```
def redistribute (ions, list_bests){
    for ion in ions{
        if rand(0,1) < prob_restart
            ion[j] = [random(-100,100) for j in {1...length(ion)}]
        else{
            if !list_bests.empty(){
                ions = lists_bests.pop()
            }
            else{
                ion[j] = [random(-100,100) for j in {1...length(ion)}]
            }
            for j in {1...length(ion)}{
                if(rand(0,1) < prob_mutation)
                    ion[j] = random(-100,100)
            }
        }
    }
}
```

Cuadro 5: Resultados del *Algoritmo de iones mejorado + Simplex*

	D=10, IMO	D=10, IMO mejorado	D=30, IMO	D=30, IMO mejorado
f1	19084554.669857	26329.979483	78738913.351420	230907.346453
f2	16536.801547	3721.020604	13697676.211767	9338.377990
f3	21065.221623	18588.818078	77800.939852	73035.251038
f4	58.764875	26.436828	246.343748	19.987042
f5	20.170120	20.121005	20.468675	20.274226
f6	7.539682	8.503934	34.790689	39.073894
f7	0.861159	1.562235	1.102618	0.009251
f8	28.934028	37.319324	129.837295	293.089630
f9	26.123286	49.547829	145.730468	330.339667
f10	1157.285877	970.595306	4219.909911	5467.684508
f11	1144.876264	1054.685361	4881.285910	5191.974227
f12	0.661762	0.739751	1.304201	1.809880
f13	0.456650	0.484668	0.440387	0.498159
f14	0.635537	0.490654	0.260939	0.356620
f15	4.164066	26.022467	51.414376	182.177060
f16	3.390798	3.541966	12.366670	12.956010
f17	236940.300926	5905.531778	3390125.155789	16890.938744
f18	6522.070266	9420.863246	110104.005591	8978.058073
f19	8.886340	8.106570	32.516194	105.267242
f20	13892.000148	4783.046252	53412.013643	31203.001882

5. Empleo del software programado

Para compilar el software proporcionado, basta hacer, en el directorio `src`, lo siguiente:

```
Cmake .
make
./main
```

Cuando ejecutamos el programa, obtenemos para dimensión 10 y 30, la evaluación que producen el algoritmo de iones. Si se quiere modificar algún parámetro, debe hacerse en el fichero `./src/aux.cpp`.

- `num_ejecuciones`. Fijado a 25.
- `population_size`. Tamaño de la población total de iones en cada iteración. La mitad serán aniones y la otra mitad cationes.
- `lbound` y `ubound`. Son los límites del espacio $[lbound, ubound]^D$.
- `prob_restart`. Probabilidad de reinicio de aniones y cationes. Por defecto a 0.1.
- `type_ls`. Puede ser "sw", "cmaes" o "simplex". Tipo de búsqueda local a emplear.

Si se ha modificado algún parámetro, habrá que hacer:

```
make
./main
```

6. Conclusiones aprendidas