Práctica alternativa examen

Propuesta original de metaheurística

Alejandro Palencia Blanco

DNI: 77177568X

alepalenc@correo.ugr.es

Grupo y horario de prácticas: Jueves 17:30-19:30

Índice

1. Descripción de la metaheurística

1.1. Idea que la inspira

Mi propuesta de metaheurística se inspira en el proceso de colonización de territorios inexplorados junto con la invasión que ejercen las civilizaciones más poderosas sobre otras más débiles. He pensado que esta idea puede ser prometedora porque, al incentivar la exploración de nuevas zonas del espacio de búsqueda, favorecemos la posibilidad de encontrar mejores soluciones o al menos con un mayor potencial para ser explotadas. Además, la invasión permite eliminar aquellas soluciones que son poco prometedoras por tener cerca suya a otra con un mayor fitness.

El algoritmo parte de una población con un tamaño prefijado en función de la dimensión generada aleatoriamente. En cada iteración del algoritmo, se aumenta el tamaño de la población mediante colonización, luego se desarrollan las mejores soluciones mediante la búsqueda local Solis Wets y, por último, se eliminan soluciones de la población mediante invasión. Esto se repite hasta que el tamaño de la población llegue a ser muy pequeño, momento en el cual se pasa a desarrollar las mejores soluciones encontradas usando las evaluaciones restantes.

Me he visto obligado a usar Solis Wets en la primera versión de la metaheurística porque sin ella, la explotación de las soluciones era demasiado escasa, lo que provocaba que los tiempos de ejecución fuesen demasiado altos y que no se llegase a encontrar soluciones de una calidad aceptable.

1.2. Estructura de datos que guarda una solución

En mi implementación, una solución es un `struct` de `C++` llamado `Solution` con dos campos:

- `v`: vector de tipo `double` que almacena una solución.
- `fitness`: *fitness* de la solución, inicialmente a -1.

Además, `Solution` dispone de un constructor que recibe como parámetro un entero y fija la dimensión de `v` a ese entero.

1.3. Colonización

El operador de colonización añade a la población el número de hijos especificado en el parámetro num_childs. He implementado dos formas distintas de generar hijos:

- Generación hacia el centro del espacio de búsqueda: Se seleccionan dos padres aletoriamente y se toma como hijo el punto medio del segmento que determinan.
- Generación hacia los límites del espacio de búsqueda: Se selecciona un padre aletoriamente y se genera un hijo cuyas coordenadas son las del padre multiplicadas por una constante mayor que 1 (controlando que el hijo se quede dentro del espacio de búsqueda).

El operador genera el mismo número de hijos mediante ambos métodos de generación, es decir, se generan $num\_childs/2$ hijos mediante cada método.

Algorithm 1: colonization

Data: population : población de soluciones
num_childs : número de hijos a generar
lower : cota inferior del espacio de búsqueda
upper : cota superior del espacio de búsqueda

begin
  $dim \leftarrow |population[0].v|$
  $random\_shuffle(population)$
  for $i \in \{0, \ldots, num\_childs/2 - 1\}$ **do**
    $child \leftarrow new\ Solution(dim)$
    for $j \in \{0, \ldots, dim - 1\}$ **do**
      $child.v[j] \leftarrow (population[2i].v[j] + population[2i + 1].v[j])/2$
    end
    $child.fitness \leftarrow cec17\_fitness(child.v)$
    $population \leftarrow population \cup \{child\}$
  end
  for $i \in \{num\_childs, \ldots, (3 \cdot num\_childs)/2 - 1\}$ **do**
    $child \leftarrow new\ Solution(dim)$
    $t \leftarrow random\ value\ from\ U(0.1, 1)$
    for $j \in \{0, \ldots, dim - 1\}$ **do**
      if $population[i].v[j] \geq 0$ **then**
        $child.v[j] \leftarrow population[i].v[j] + t \cdot (upper - population[i].v[j])$
      else
        $child.v[j] \leftarrow population[i].v[j] + t \cdot (lower - population[i].v[j])$
      end
    end
    $child.fitness \leftarrow cec17\_fitness(child.v)$
    $population \leftarrow population \cup \{child\}$
  end
end

1.4. Invasión

Para el operador de invasión, he implementado la función *removeSolutions*, la cual elimina un número concreto de soluciones de la población siempre verificando que su tamaño nunca sea inferior a s. Esto se lleva a cabo creando el conjunto $sols\_to\_remove$ que contiene las posiciones de las soluciones que posteriormente serán eliminadas.

Para cada par de soluciones cuya distancia sea menor que un umbral especificado, añadimos al conjunto $sols\_to\_remove$ aquella que tenga peor fitness. Se siguen añadiendo soluciones a este conjunto mientras que no se hayan comprobado todas las posibles parejas de soluciones y se cumpla la restricción $|population| - |sols\_to\_remove| > s$.

Algorithm 2: removeSolutions

```
Data: population : población de soluciones
      s : tamaño mínimo de la población al eliminar las soluciones
      threshold : umbral que determina la distancia bajo la que se da una invasión
begin
    sols_to_remove ← {}
    for i ∈ {0, ..., |population| − 1} and |sols_to_remove| < |population| − s do
        for j ∈ {i + 1, ..., |population| − 1} and |sols_to_remove| < |population| − s do
            if distance(population[i], population[j]) < threshold then
                if population[i].fitness < population[j].fitness then
                    sols_to_remove ← sols_to_remove ∪ {j}
                else
                    if population[i].fitness > population[j].fitness then
                        sols_to_remove ← sols_to_remove ∪ {i}
                    end
                end
            end
        end
    end
    for i ∈ sols_to_remove do
        population ← population − {population[i]}
    end
end
```

El operador de invasión elimina de la población un número de hijos concreto para que, al finalizar este operador, el tamaño de la población sea el especificado en el parámetro s. Para ello, primero se calcula la distancia media entre todas las soluciones (usamos la función *distance* para calcular la distancia euclídea entre dos soluciones).

A continuación, llamamos a *removeSolutions* para que elimine soluciones que se encuentren a una distancia menor que $mean\_distance/16$ de otra con mejor fitness. Procedemos análogamente aumentando progresivamente la distancia umbral hasta que la población alcance el tamaño s deseado. Los siguientes umbrales son $mean\_distance/8$, $mean\_distance/4$, $mean\_distance/2$ y 10^7 (este último equivale a que no exista un umbral).

Algorithm 3: invasion

```
Data: population : población de soluciones
      s : tamaño de la población al finalizar la invasión
begin
    mean_distance ← 0
    for i ∈ {0, ..., |population| − 1} do
        for j ∈ {i + 1, ..., |population| − 1} do
            mean_distance ← distance(population[i], population[j])
        end
    end
    mean_distance ← (2 · mean_distance)/(|population| · (|population| − 1))
    random_shuffle(population)
    removeSolutions(population, s, mean_distance/16)
    removeSolutions(population, s, mean_distance/8)
    removeSolutions(population, s, mean_distance/4)
    removeSolutions(population, s, mean_distance/2)
    removeSolutions(population, s, 10^7)
end
```

1.5. Desarrollo

El operador de desarrollo simplemente aplica la búsqueda local Solis Wets a las $sols\_to\_develop$ soluciones con mejor fitness. Todas ellas disponen del mismo número de evaluaciones para ser desarrolladas mediante Solis Wets, especificado en el parámetro $evals\_per\_sol$.

Algorithm 4: development

```
Data: population : población de soluciones
      evals_per_sol : evaluaciones que se aplican en cada búsqueda local
      sols_to_develop : soluciones sobre las que aplica búsqueda local
      lower : cota inferior del espacio de búsqueda
      upper : cota superior del espacio de búsqueda
begin
    if sols_to_develop > |population| then
        sols_to_develop ← |population|
    end
    findBestSolutions(population, sols_to_develop)
    for i ∈ {0, ..., sols_to_develop − 1} do
        soliswets(population[i].v, population[i].fitness, 0.2, evals_per_sol, lower, upper)
    end
end
```

Para encontrar las $sols\_to\_develop$ mejores soluciones, he implementado la función $findBestSolutions$, que coloca las mejores soluciones en las primeras posiciones de la población ordenadas de mejor a peor.

Algorithm 5: findBestSolutions

```
Data: population : población de soluciones
      best_sols_to_find : número de mejores soluciones a encontrar
begin
    for i ∈ {0, ..., best_sols_to_find − 1} do
        k ← i
        for j ∈ {i + 1, ..., |population| − 1} do
            if population[k].fitness > population[j].fitness then
                k = j
            end
        end
        swap(population[i], population[k])
    end
end
```

1.6. Primera versión de la metaheurística

La metaheurística empieza inicializando las siguientes constantes:

- s_init: Tamaño inicial de la población generada aleatoriamente. Este tamaño depende de la dimensión (2000 para dimensión 10, 2500 para 30 y 3000 para 50).
- $fraction$: Fracción de la población actual que se corresponde con el número de hijos generados en esa iteración. Es decir, en cada iteración se generan $p = ceil(s/fraction)$, siendo s el tamaño de la población actual. Si p es impar, se le suma 1 para que sea par. Este valor depende de la dimensión (10 para dimensión 10, 12 para 30 y 14 para 50).
- $development\_evals$: Número de evaluaciones especificadas en Solis Wets cada vez que es aplicado, fijado a 500.

A continuación, genera una población de tamaño s_init aleatoriamente e inicializa $evals = s\_init$ (número de evaluaciones realizadas hasta el momento).

Luego, aplica los operadores de colonización, desarrollo e invasión mientras que queden más de s_init evaluaciones por realizar y el tamaño de la población sea superior a 20 (los valores de las constantes están ajustados para que el bucle finalice por esta segunda condición). Al finalizar cada iteración, el tamaño de la población se reduce a $s_{i+1} = s_i - p$, siendo $p = ceil(s_i/fraction)$.

Al salir del bucle, ya se deberían haber alcanzado soluciones razonablemente prometedoras. Por ello, aplicamos Solis Wets sobre las 5 mejores soluciones hasta que el número de evaluaciones restantes sea inferior a s_init.

Por último, desarrollamos la mejor solución hasta el momento con las evaluaciones restantes y devolvemos su fitness.

Algorithm 6: miMH1

Data: max_evals : número máximo de evaluaciones
dim : dimensión de las soluciones
lower : cota inferior del espacio de búsqueda
upper : cota superior del espacio de búsqueda
Result: fitness : fitness de la mejor solución encontrada
begin
$s\_init \leftarrow 1750 + 25 \cdot dim$
$s \leftarrow s\_init$
$fraction \leftarrow 9 + dim/10$
$development\_evals \leftarrow 500$
$evals \leftarrow 0$
for $i \in \{0, \ldots, s\_init - 1\}$ **do**
$random\_sol \leftarrow new\ Solution(dim)$
for $j \in \{0, \ldots, dim - 1\}$ **do**
$random\_sol.v[j] \leftarrow random\ value\ from\ U(lower, upper)$
end
$random\_sol.fitness \leftarrow cec17\_fitness(random\_sol.v)$
$population \leftarrow population \cup \{random\_sol\}$
end
$evals \leftarrow evals + s\_init$
while $evals < max\_evals - s\_init\ and\ s > 20$ **do**
$p \leftarrow ceil(s/fraction)$
$p \leftarrow p + p\,\%2$
$colonization(population, p, lower, upper)$
$evals \leftarrow evals + p$
$sols\_to\_develop = floor((s\_init - p)/development\_evals)$
$development(population, development\_evals, sols\_to\_develop, lower, upper)$
$evals \leftarrow evals + development\_evals \cdot sols\_to\_develop$
$s \leftarrow s - p$
$invasion(population, s)$
end
$sols\_to\_develop \leftarrow 5$
$findBestSolutions(population, sols\_to\_develop)$
while $evals < max\_evals - s\_init$ **do**
for $i \in \{0, \ldots, sols\_to\_develop\}$ **do**
$soliswets(population[i].v, population[i].fitness, 0.2, development\_evals, lower, upper, random)$
end
$evals \leftarrow evals + development\_evals \cdot sols\_to\_develop$
end
$findBestSolution(population, 1)$
$soliswets(population[0].v, population[0].fitness, 0.2, max\_evals - evals, lower, upper)$
$fitness \leftarrow population[0].fitness$
end

2. Análisis de resultados

He decidido comparar mi metaheurística con dos de los algoritmos vistos en la asignatura basados en poblaciones con variables reales: DE (Differential Evolution) y PSO (Particle Swarm Optimization). Los resultados se muestran en las tablas 1, 2 y 3.

Podemos ver que para todas las dimensiones DE obtiene los mejores resultados en la mayor parte de las funciones. La primera versión de mi metaheurística gana en un número reducido de casos y parece que es ligeramente peor en dimensión 30. Sin embargo, podemos apreciar que, en la mayoría de las funciones, mi metaheurística obtiene errores de órdenes muy similares a los obtenidos por DE y PSO. El único caso excepcional lo encontramos en la función 2, en la que mi metaheurística obtiene un error muchísimo más alto que DE o PSO para todas las dimensiones.

	DE	PSO	miMH1
F01	**0.000e+00**	5.255e+07	7.350e+01
F02	**0.000e+00**	1.000e+00	1.677e+10
F03	**0.000e+00**	1.989e+03	0.000e+00
F04	**1.105e-04**	4.684e+01	6.191e-01
F05	1.151e+02	**3.212e+01**	3.276e+01
F06	3.460e+01	**1.001e+01**	3.136e+01
F07	**3.848e+01**	4.275e+01	1.012e+02
F08	2.983e+01	**2.203e+01**	3.154e+01
F09	1.938e+02	**5.686e+01**	4.117e+02
F10	**3.597e+02**	1.077e+03	8.187e+02
F11	**1.942e-02**	3.843e+01	3.722e+01
F12	**4.931e+00**	2.517e+06	3.791e+04
F13	**5.988e+00**	8.409e+03	3.770e+03
F14	**5.240e-02**	9.993e+01	3.108e+02
F15	**6.060e-02**	2.066e+03	6.829e+03
F16	4.561e+02	**1.415e+02**	1.946e+02
F17	**2.350e+01**	6.497e+01	1.021e+02
F18	**3.630e-02**	1.484e+04	4.166e+03
F19	**5.192e-03**	3.220e+03	5.557e+03
F20	3.837e+02	**8.444e+01**	1.328e+02
F21	1.889e+02	1.320e+02	**1.016e+02**
F22	1.005e+02	**7.740e+01**	8.612e+01
F23	8.098e+02	**3.304e+02**	3.370e+02
F24	**1.000e+02**	1.810e+02	1.119e+02
F25	4.040e+02	4.478e+02	**1.109e+02**
F26	2.706e+02	3.729e+02	**2.310e+02**
F27	**3.897e+02**	4.134e+02	4.160e+02
F28	3.517e+02	4.698e+02	**3.172e+02**
F29	**2.375e+02**	3.193e+02	3.960e+02
F30	**8.051e+04**	6.352e+05	1.186e+05
Best	**18**	8	4

Cuadro 1: Comparación con DE y PSO (dimensión 10)

	DE	PSO	miMH1
F01	4.909e+04	4.175e+09	**4.989e+02**
F02	1.309e+19	**1.000e+00**	9.658e+43
F03	**3.481e+03**	5.453e+04	8.693e+04
F04	**8.430e+01**	1.183e+03	9.984e+01
F05	**2.015e+02**	2.170e+02	2.429e+02
F06	**6.320e+00**	3.694e+01	5.751e+01
F07	**2.334e+02**	3.596e+02	9.142e+02
F08	1.894e+02	**1.745e+02**	1.810e+02
F09	**6.530e+01**	2.842e+03	4.630e+03
F10	**3.764e+03**	6.938e+03	4.155e+03
F11	**7.958e+01**	1.207e+03	1.660e+02
F12	**3.258e+05**	3.586e+08	6.965e+05
F13	**1.536e+02**	4.508e+07	3.916e+04
F14	**7.100e+01**	3.059e+05	3.153e+04
F15	**6.256e+01**	2.737e+05	2.314e+04
F16	**1.319e+03**	1.568e+03	1.361e+03
F17	4.809e+02	**4.725e+02**	8.911e+02
F18	**6.122e+01**	2.170e+06	1.567e+05
F19	**3.572e+01**	1.260e+06	2.453e+05
F20	**2.751e+02**	4.621e+02	6.051e+02
F21	**3.255e+02**	4.113e+02	4.073e+02
F22	**1.002e+02**	1.027e+03	4.157e+03
F23	**5.346e+02**	6.404e+02	8.480e+02
F24	**6.059e+02**	7.095e+02	8.730e+02
F25	**3.870e+02**	6.864e+02	4.006e+02
F26	**4.038e+02**	3.369e+03	3.204e+03
F27	**4.925e+02**	8.068e+02	8.191e+02
F28	3.943e+02	1.105e+03	**3.570e+02**
F29	**1.025e+03**	1.409e+03	2.027e+03
F30	**3.657e+03**	1.359e+07	9.363e+05
Best	**25**	3	2

Cuadro 2: Comparación con DE y PSO (dimensión 30)

	DE	PSO	miMH1
F01	2.039e+08	1.823e+10	**8.952e+02**
F02	5.158e+52	**1.000e+00**	5.109e+78
F03	**7.597e+04**	1.008e+05	8.220e+04
F04	2.322e+02	4.091e+03	**1.735e+02**
F05	**3.978e+02**	4.328e+02	4.350e+02
F06	**1.180e+01**	5.450e+01	6.226e+01
F07	**4.781e+02**	8.197e+02	1.808e+03
F08	**4.070e+02**	4.077e+02	4.188e+02
F09	**3.756e+03**	1.620e+04	1.210e+04
F10	1.127e+04	1.295e+04	**7.482e+03**
F11	**2.173e+02**	3.151e+03	2.296e+02
F12	1.382e+08	6.062e+09	**4.995e+06**
F13	**2.726e+04**	5.262e+08	4.100e+04
F14	**1.683e+02**	1.558e+06	2.417e+04
F15	**5.090e+02**	5.790e+06	1.717e+04
F16	3.047e+03	2.745e+03	**2.001e+03**
F17	1.827e+03	**1.652e+03**	2.277e+03
F18	**2.420e+04**	1.203e+07	3.432e+05
F19	**1.322e+02**	1.336e+07	6.647e+04
F20	**9.384e+02**	1.282e+03	1.285e+03
F21	**6.177e+02**	6.655e+02	7.473e+02
F22	**1.707e+02**	1.258e+04	7.635e+03
F23	**8.435e+02**	1.098e+03	1.537e+03
F24	**8.850e+02**	1.206e+03	1.497e+03
F25	**5.667e+02**	3.069e+03	5.773e+02
F26	**5.745e+02**	7.890e+03	1.023e+04
F27	**5.765e+02**	1.744e+03	2.055e+03
F28	**5.013e+02**	3.468e+03	5.444e+02
F29	**2.218e+03**	3.566e+03	4.525e+03
F30	**2.386e+06**	3.374e+08	2.094e+07
Best	**23**	2	5

Cuadro 3: Comparación con DE y PSO (dimensión 50)

3. Hibridación

Como ya hemos aplicado la búsqueda local Solis Wets en la primera versión de la metaheurística, probaré a hibridar mi metaheurística con DE. Además de que DE es la metaheurística que obtiene los mejores resultados en la gran mayoría de los casos, he pensado que esta hibridación puede ser una buena idea porque es posible la población obtenida mediante mi metaheurística sea un buen punto de partida para aplicar DE posteriormente.

El algoritmo DE que he implementado toma en cada iteración un valor de cr (ratio de cruce) en una distribución uniforme en el intervalo $[0.4, 0.6]$ y un valor de f (factor de escalado de la mutación) en una distribución uniforme en el intervalo $[0.3, 0.7]$.

Algorithm 7: differentialEvolution

Data: population : población de soluciones
iterations : número de iteraciones que se aplican
begin
  for $iter \in \{0, \dots, iterations - 1\}$ **do**
    $cr \leftarrow random\ value\ from\ U(0.4, 0.6)$
    $f \leftarrow random\ value\ from\ U(0.3, 0.7)$
    for $i \in \{0, \dots, |population| - 1\}$ **do**
      $child \leftarrow newSolution(dim)$
      $parents_0 \leftarrow random\ value\ from\ U(0, |population| - 1)$
      $parents_1 \leftarrow random\ value\ from\ U(0, |population| - 1)$
      $parents_2 \leftarrow random\ value\ from\ U(0, |population| - 1)$
      for $k \in \{0, \dots, dim - 1\}$ **do**
        $t \leftarrow random\ value\ from\ U(0.0, 1.0)$
        if $t < cr$ **then**
          $child.v[k] \leftarrow$
          $population[parents_0].v[k] + f \cdot (population[parents_1].v[k] - population[parents_2].v[k])$
        else
          $child.v[k] \leftarrow population[i].v[k]$
        end
      end
      $child.fitness \leftarrow cec17\_fitness(child.v)$
      if $child.fitness < population[i].fitness$ **then**
        $population[i] \leftarrow child$
      end
    end
  end
end

A partir de esta función, he diseñado la segunda versión de mi metaheurística, en la que solo cambian dos cosas:

- La parte relativa a la explotación de las 5 mejores soluciones mediante Solis Wets se sustituye por la hibridación con DE. Esta consiste en aplicar 20 iteraciones de DE hasta que el número de evaluaciones restantes sea menor que $20 \cdot |population|$.
- El criterio de parada de la parte de colonización-invasión cambia de $s > 20$ a $s > s\_init/3$. Esto lo hago con el objetivo de que la población que reciba inicialmente DE tenga un tamaño considerable (un tercio del tamaño inicial).

El pseudocódigo de la parte modificada sería el siguiente:

Algorithm 8: miMH2

Data: max_evals : número máximo de evaluaciones
dim : dimensión de las soluciones
lower : cota inferior del espacio de búsqueda
upper : cota superior del espacio de búsqueda
Result: fitness : fitness de la mejor solución encontrada
begin
⋮
$iters\_de \leftarrow 20$
while $evals < max\_evals - iters\_de \cdot |population|$ **do**
$differentialEvolution(population, iters\_de)$
$evals \leftarrow iters\_de \cdot |population|$
end
$findBestSolution(population, 1)$
$soliswets(population[0].v, population[0].fitness, 0.2, max\_evals - evals, lower, upper)$
$fitness \leftarrow population[0].fitness$
end

3.1. Análisis de resultados

Los resultados se muestran en las tablas 4, 5 y 6.

Podemos ver que la segunda versión consigue mejorar de forma considerable a la primera, sobre todo en las dimensiones 10 y 50, en las que casi consigue igualar en casos ganadores a DE. De nuevo, los órdenes de magnitud de los errores vuelven a ser muy similares con respecto a DE y PSO, exceptuando el caso de la función 2.

Me resulta difícil encontrar la razón por la que el número de casos ganadores de mi metaheurística en dimensión 30 es significativamente menor que en las otras, pero creo que puede ser debido a que los valores de algunos parámetros (tamaño de la población inicial, número de hijos generados) no son los más adecuados.

	DE	PSO	miMH2
F01	**0.000e+00**	5.255e+07	1.337e+03
F02	**0.000e+00**	1.000e+00	1.310e+07
F03	**0.000e+00**	1.989e+03	1.065e-02
F04	**1.105e-04**	4.684e+01	1.919e+00
F05	1.151e+02	3.212e+01	**1.321e+01**
F06	3.460e+01	1.001e+01	**3.207e-01**
F07	3.848e+01	4.275e+01	**2.369e+01**
F08	2.983e+01	2.203e+01	**1.313e+01**
F09	1.938e+02	5.686e+01	**1.353e+00**
F10	**3.597e+02**	1.077e+03	8.547e+02
F11	**1.942e-02**	3.843e+01	9.118e+00
F12	**4.931e+00**	2.517e+06	8.642e+04
F13	**5.988e+00**	8.409e+03	8.810e+02
F14	**5.240e-02**	9.993e+01	2.859e+01
F15	**6.060e-02**	2.066e+03	5.223e+01
F16	4.561e+02	1.415e+02	**4.385e+01**
F17	**2.350e+01**	6.497e+01	4.370e+01
F18	**3.630e-02**	1.484e+04	1.293e+03
F19	**5.192e-03**	3.220e+03	7.550e+00
F20	3.837e+02	8.444e+01	**3.061e+01**
F21	1.889e+02	1.320e+02	**1.040e+02**
F22	1.005e+02	**7.740e+01**	8.883e+01
F23	8.098e+02	**3.304e+02**	3.337e+02
F24	**1.000e+02**	1.810e+02	2.501e+02
F25	4.040e+02	4.478e+02	**3.745e+02**
F26	**2.706e+02**	3.729e+02	2.899e+02
F27	3.897e+02	4.134e+02	**3.824e+02**
F28	3.517e+02	4.698e+02	**3.504e+02**
F29	**2.375e+02**	3.193e+02	3.279e+02
F30	8.051e+04	6.352e+05	**2.641e+03**
Best	**16**	2	12

Cuadro 4: Comparación con DE y PSO (dimensión 10)

	DE	PSO	miMH2
F01	4.909e+04	4.175e+09	**5.825e+03**
F02	1.309e+19	**1.000e+00**	1.394e+38
F03	**3.481e+03**	5.453e+04	1.218e+05
F04	8.430e+01	1.183e+03	**4.264e+01**
F05	2.015e+02	2.170e+02	**1.769e+02**
F06	6.320e+00	3.694e+01	**1.727e+00**
F07	2.334e+02	3.596e+02	**8.713e+01**
F08	1.894e+02	1.745e+02	**1.283e+02**
F09	**6.530e+01**	2.842e+03	1.769e+02
F10	**3.764e+03**	6.938e+03	4.420e+03
F11	**7.958e+01**	1.207e+03	2.132e+02
F12	**3.258e+05**	3.586e+08	2.894e+06
F13	**1.536e+02**	4.508e+07	6.182e+04
F14	**7.100e+01**	3.059e+05	3.036e+04
F15	**6.256e+01**	2.737e+05	3.165e+04
F16	**1.319e+03**	1.568e+03	1.588e+03
F17	4.809e+02	**4.725e+02**	6.673e+02
F18	**6.122e+01**	2.170e+06	6.939e+05
F19	**3.572e+01**	1.260e+06	5.885e+04
F20	**2.751e+02**	4.621e+02	5.665e+02
F21	**3.255e+02**	4.113e+02	3.765e+02
F22	**1.002e+02**	1.027e+03	4.231e+03
F23	**5.346e+02**	6.404e+02	5.643e+02
F24	**6.059e+02**	7.095e+02	6.423e+02
F25	**3.870e+02**	6.864e+02	4.078e+02
F26	**4.038e+02**	3.369e+03	2.652e+03
F27	**4.925e+02**	8.068e+02	5.000e+02
F28	**3.943e+02**	1.105e+03	4.566e+02
F29	**1.025e+03**	1.409e+03	1.691e+03
F30	**3.657e+03**	1.359e+07	1.088e+05
Best	**22**	2	6

Cuadro 5: Comparación con DE y PSO (dimensión 30)

	DE	PSO	miMH2
F01	2.039e+08	1.823e+10	**5.389e+03**
F02	5.158e+52	**1.000e+00**	3.936e+73
F03	**7.597e+04**	1.008e+05	2.318e+05
F04	2.322e+02	4.091e+03	**2.136e+02**
F05	**3.978e+02**	4.328e+02	4.060e+02
F06	1.180e+01	5.450e+01	**6.145e+00**
F07	4.781e+02	8.197e+02	**1.789e+02**
F08	4.070e+02	4.077e+02	**3.243e+02**
F09	3.756e+03	1.620e+04	**2.451e+03**
F10	1.127e+04	1.295e+04	**7.638e+03**
F11	**2.173e+02**	3.151e+03	3.636e+02
F12	1.382e+08	6.062e+09	**3.422e+07**
F13	**2.726e+04**	5.262e+08	6.134e+04
F14	**1.683e+02**	1.558e+06	4.075e+05
F15	**5.090e+02**	5.790e+06	3.314e+04
F16	3.047e+03	**2.745e+03**	2.769e+03
F17	1.827e+03	**1.652e+03**	2.275e+03
F18	**2.420e+04**	1.203e+07	1.501e+06
F19	**1.322e+02**	1.336e+07	1.939e+05
F20	**9.384e+02**	1.282e+03	1.332e+03
F21	6.177e+02	6.655e+02	**5.440e+02**
F22	**1.707e+02**	1.258e+04	8.434e+03
F23	8.435e+02	1.098e+03	**8.406e+02**
F24	**8.850e+02**	1.206e+03	9.649e+02
F25	**5.667e+02**	3.069e+03	5.842e+02
F26	**5.745e+02**	7.890e+03	4.597e+03
F27	5.765e+02	1.744e+03	**5.000e+02**
F28	5.013e+02	3.468e+03	**5.000e+02**
F29	**2.218e+03**	3.566e+03	4.060e+03
F30	**2.386e+06**	3.374e+08	5.538e+07
Best	**15**	3	12

Cuadro 6: Comparación con DE y PSO (dimensión 50)