METAHEURÍSTICAS

PRÁCTICA FINAL (EXAMEN TEORIA): ELECCIÓN DE UNA METAHEURÍSTICA NO ESTUDIADA EN CLASE A PARTIR DEL ARTÍCULO FACILITADO POR EL PROFESOR

(SFHA) - SHEEP FLOCK HEREDITY ALGORITHM

Curso 2019-2020

Tercer curso del Grado en Ingeniería Informática

Iván Cortón da Silva 3ºCSI
15426623-V
ivancorton99@correo.ugr.es
Grupo MH2: jueves de 17.30h a 19.30h

ÍNDICE:

PRÁCTICA FINAL – ESTUDIO DEL MODELO DE EVOLUCION NATURAL DE LA OVEJA EN EL REBAÑO PARA EL PROBLEMA DE LA MÁXIMA DIVERSIDAD

Con el fin de llevar a cabo el correcto desarrollo de la práctica final de la asignatura, se nos pide el estudio de una metaheurística original, siendo en mi caso el *Sheep Flock Heredity Model,* en concreto, aplicado a la resolución del problema de la máxima diversidad (MDP) descrito en la parte de prácticas de la asignatura.

Para ello, a continuación, expondré las adaptaciones de dichas técnicas al problema elegido, en el que finalmente realizaremos una comparación de los resultados obtenidos con las estimaciones convenientes para el valor de los óptimos de una serie de casos del problema.

El problema de la máxima diversidad, está caracterizado por ser un problema de optimización combinatoria que consiste en seleccionar un subconjunto M de m elementos (|M| = m) de un conjunto inicial N de n elementos (con n > m) de forma que se podría formular como sigue:

$$\text{Maximizar} \quad z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Sujeto a} \quad \sum_{i=1}^{n} x_i = m$$

$$x_i = \{0,1\}, \quad i = 1, \ldots, n.$$

donde:

- X es un vector solución al problema binario que indica los m elementos seleccionados.
- d_{ij} es la distancia existente entre los elementos i-ésimos y j-ésimos.

Por otro lado, respecto a los casos a poner a prueba dentro de cada tipo de enfoque del problema (AGG, AGE, AM), se utilizarán un total de **30 casos** seleccionados de varios de los conjuntos de instancias disponibles en la MDPLIB (http://www.optsicom.es/mdp/) y en la plataforma Prado.

Entre estos **30 casos**, encontramos 10 pertenecientes al grupo **GKD** con **distancias Euclideas** relativamente pequeñas, con n=500 y m=50 (GKD-c_11_n500_m50 a GKDc_20_n500_m50). Un segundo grupo **MDG** caracterizado por utilizar **distancias reales** en [0,1000], con n=500 y m=50 (MDG-b_1_n500_m50 a MDG-b_10_n500_m50); y por último otras 10 del grupo **MDG** con **distancias de números enteros** entre 0 y 10 con n=2000 y m=200 (MDG-a_31_n2000_m200 a MDGa_40_n2000_m200).

Enfoque Sheep Flock Heredity Model (SFHM)

En primer lugar, trataremos brevemente de explicar en qué consiste un algoritmo genético, para posteriormente especificar qué caso de algoritmo genético estamos tratando de implementar y su estructura y representación. Los algoritmos genéticos se enmarcan dentro de los algoritmos evolutivos, que incluyen también las estrategias evolutivas, la programación evolutiva y la programación genética. Estos algoritmos están caracterizados por hacer evolucionar una población de individuos sometiéndola a acciones aleatorias similares a las que actúan en la evolución biológica (mutaciones y recombinaciones genéticas), así como una selección de acuerdo con algún criterio (que en nuestro caso será similar al de los algoritmos genéticos generacionales), en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados (mediante la aplicación de elitismo en la segunda versión del algoritmo que explicaremos más adelante).

El *Sheep Flock Heredity Algorithm* fue diseñado inicialmente por Nara et al (1999) y fue aplicado a problemas de planificación. Este algoritmo se basó en la evolución natural de la oveja en el rebaño.

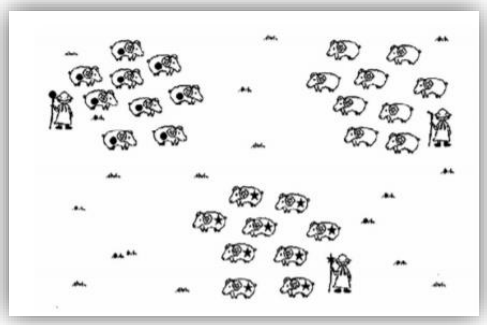


Figura 1. Rebaños de ovejas en un campo.

Normalmente, las ovejas en cada rebaño viven bajo el control de los pastores. Por tanto, podríamos suponer que la herencia genética únicamente tendrá lugar dentro de este rebaño. Algunas de las características especiales de un rebaño se desarrollan solo dentro del mismo por herencia, y la oveja con las características capaces de maximizar el valor de la función objetivo que se reproduce con mayor frecuencia que las demás.

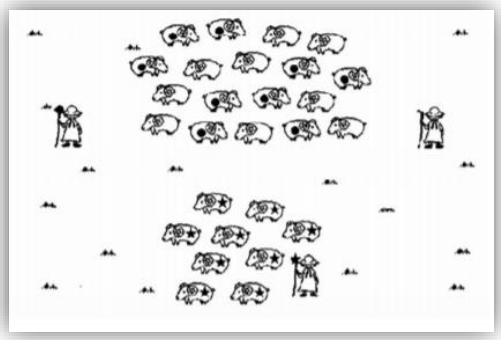


Figura 2. Mezcla de dos de los rebaños.

Supongamos la situación en la que tenemos dos rebaños, en el que uno está mezclado y el otro no, como podemos ver en la *Figura 2*. Las características de las ovejas en el rebaño vecino obviamente serán inherentes a las ovejas en el rebaño mezclado.

Tras realizar la mezcla y reproducción de las ovejas, los pastores son los encargados de separar el rebaño correspondiente para volver a la estructura de tres rebaños que teníamos inicialmente como en la *Figura 1*. Sin embargo, los pastores no pueden distinguir las ovejas que tenían inicialmente ya que la apariencia de las mismas son la misma, lo que hace inevitable la existencia de un cruce o mezcla con las ovejas pertenecientes al rebaño con el que se mezclaron al principio generando así una mayor diversidad de opciones como podemos comprobar en la *Figura 3.*

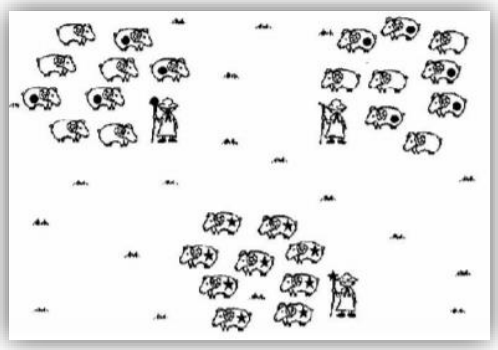
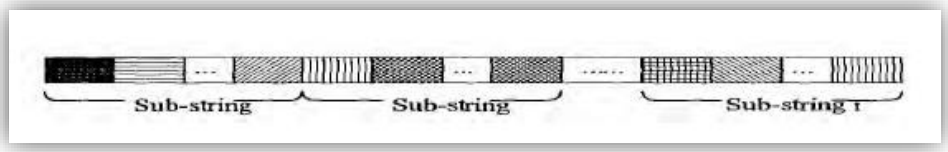


Figura 3. Nuevos rebaños de ovejas.

Obviemos que un cromosoma de un AG puede dividirse en varias subcadenas/subcromosomas de misma longitud (Nara, 1996), quedándonos con una estructura similar a la siguiente:



El fenómeno de la evolución natural de la oveja en el rebaño, se corresponde con las operaciones genéticas sobre cadenas como las de la imagen. Para este tipo de cromosomas, podemos definir los siguientes tipos de operaciones genéticas a llevar a cabo, como son:

- ***Operaciones genéticas entre cromosomas***

- ***Operaciones genéticas entre sub-cromosomas***

En la estructura de cromosomas del algoritmo basado en el modelo de herencia de las ovejas en el rebaño, se introdujeron operaciones genéticas tales como son el cruce y la mutación genética en dos ámbitos diferentes, tanto a nivel de cromosoma como de sub-cromosoma con el fin de mejorar la diversidad.

Para facilitar el entendimiento de los símiles que usaremos para la explicación de las operaciones a tener en cuenta y evitar confusiones, podemos observar la siguiente tabla:

Evolución natural	**Operaciones genéticas**
Rebaño	Cromosoma
Oveja	Sub-cromosoma
Mezcla y separación	Cruce a nivel de cromosoma
Herencia en el rebaño	Cruce a nivel de sub-cromosoma

Una vez llegados a este punto, trataré de explicar las operaciones necesarias para la implementación de dicho algoritmo, aunque primeramente tendré que diferenciar claramente entre los ámbitos de operaciones entre cromosomas y sub-cromosomas:

- **Operaciones entre cromosomas**: Como la mayor parte de los algoritmos genéticos clásicos ya estudiados durante la asignatura, realizaremos una **operación de cruce de un solo punto** *(Single Point Crossover)*, que consiste básicamente en la elección aleatoria de un punto a partir del que se cruzarán dos de nuestros cromosomas, sustituyendo estos nuevos cromosomas generados por el cruce, a aquellos anteriores al cruce. Por otra parte, también encontraremos una operación de mutación genética (**mutación inversa** en el algoritmo original), que en nuestro caso tuvimos que sustituirla por una **mutación de un solo punto** (*Point Mutation*) para poder adaptarlo correctamente al problema de la máxima diversidad, ya que la mutación inversa únicamente resultaría útil si el orden de los elementos de nuestra lista solución alterara el valor de evaluar cada cromosoma bajo la función objetivo, factor que no influye en nuestro caso. *Probabilidad de cruce y mutación: 0.6*m/2 con m igual al número de soluciones al problema; y 0.01 * numGenes, siendo numGenes el producto entre n y m, con n igual al tamaño del problema, respectivamente.*

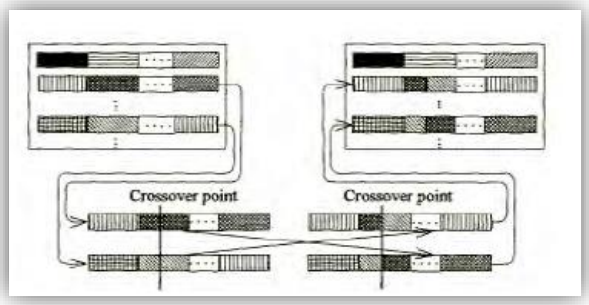


Figura 4. Cruce Simple a nivel de cromosoma.

- **Operaciones entre sub-cromosomas**: Como comentábamos anteriormente a nivel de cromosoma, entre sub-cromosomas también se realizarán operaciones a nivel genético que serán, además, de similar aplicación a las ya utilizadas. Análogamente a las operaciones entre cromosomas, tuve ciertos problemas a la hora de implementar la mutación a nivel de sub-cromosoma, ya que de igual forma que a los cromosomas, el orden no altera el valor del sub-cromosoma, por lo que tuve que utilizar un tipo de **mutación de un solo punto,** junto a la operación de **cruce de un solo punto.**

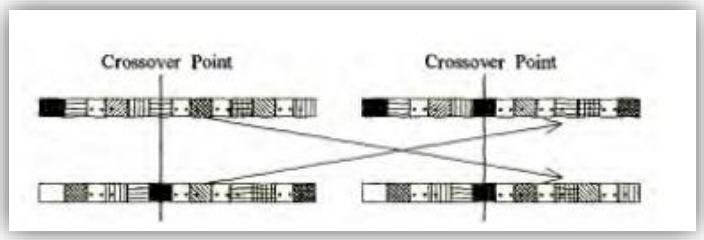


Figura 5. Cruce Simple a nivel de sub-cromosoma.

Representación de la solución:

Utilizaremos un tipo de representación entera, mediante una lista de tamaño m de listas de la forma Sel = (x1, … , x_m) en el que las posiciones de los elementos en Sel serán los índices de los elementos de nuestra solución. Llamaremos a cada lista Sel entorno y al conjunto de listas población.

Función objetivo:

$$\text{Maximizar} \quad z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\text{Sujeto a} \quad \sum_{i=1}^{n} x_i = m$$

$$x_i = \{0, 1\}, \quad i = 1, \ldots, n.$$

Nuestro objetivo es maximizar el coste de nuestra población de manera en que debemos conseguir que la distancia desde cada i a cada j distinto de i dentro de cada cromosoma, sea la máxima posible, atendiendo a que la sumatoria de elementos seleccionados en nuestra solución (cromosoma) sea exactamente m, que variará en función de los datos del problema (50, 200…).

Ciclo de evolución:

```
Inicio
      1. Inicializar listaSol(actual)
      2. Evaluar listaSol(actual)
      Mientras (no lleguemos al límite de generaciones)
            // Nivel de sub-cromosoma
            3. listaPadres = listaSol
            4. padre_i <- listaPadres[i]
            5. hijo_i <- lista de enteros
            6. listaIntermedia <- SinglePointCrossover(padre_i, hijo_i)
            7. Reparación (hijo_i)
            8. Recuperación(listaIntermedia)
            9. Mutacion(listaIntermedia)
            // Nivel de cromosoma
            10. listaPadres2 = listaIntermedia
            11. padre1 <- listaPadres2[i]
            12. padre2 <- listaPadres2[i+1]
            13. Eliminar listaPadres2[i] y listaPadres2[i+1]
            14. Vaciar listaIntermedia
            15. listaIntermedia <- SinglePointCrossover(padre1, padre2, hijo_i)
            16. Reparación (hijo1, hijo2)
            17. Recuperación(listaIntermedia)
            18. Mutacion(listaIntermedia)
            19. CalculoCosteFinal (listaIntermedia)
            20. ListaSol = listaIntermedia
      endwhile
Fin
```

donde:

- listaSol es una lista de cromosomas inicial solución al problema en representación binaria.
- listaPadres es un conjunto de cromosomas obtenido tras la selección de todo el rebaño.
- ListaIntermedia es una lista de cromosomas obtenido tras realizar cruces y mutaciones a la población de padres.

Inicialización de la solución:

```
Inicio(Sol, m)
       Desde 0
       1. Generar i aleatoriamente entre (0, n)
          Si !estaDentro(i, Sol)
             2. Sol[cont]=i
          Sino
             Mientras estaDentro(i, Sol)
             3. Generar nuevamente i
             endwhile
             4. Sol[cont]=i
          Endif
          5. cont++
       Hasta m
Fin
```

Función objetivo:

```
CalculoCosteFinal(Sol, vecDistancias)
       Desde i = listaSol[0]
              Desde j = i
                     1. CosteFinal += vecDistancias[*i][*j]
              Hasta final de listaSol
       Hasta final de listaSol
Fin
```

donde:

- Sol es un cromosoma solución al problema de representación entera.
- VecDistancias es un vector con todas las distancias del conjunto de datos actual.

Función barajar:

```
Inicio(lista)
       1. Transformar lista en vector
       2. Aplicación método std::random_shuffle
       3. Reemplazar datos de vector a lista.
Fin
```

Operadores de Cruce:

Cruce de un solo punto (nivel sub-cromosoma):

```
SinglePointCrossover (padre1, hijo1, hijo2)
       1. Generamos subLista1 y subLista2
       2. PuntoCorteMitad <- m/2
       3. PuntoCorte1 <- aleatorio(0,puntoCorteMitad)
       4. PuntoCorte2 <- aleatorio(puntoCorteMitad, m)
       5. SubLista1 <- sub-cromosoma desde inicio padre1 hasta
          puntoCorteMitad
       6. SubLista2 <- sub-cromosoma desde puntoCorteMitad hasta fin
          padre1
       7. SinglePointCrossover(subLista1, subLista2, hijo1,
          puntoCorteMitad)
Fin
```

Cruce de un solo punto (nivel cromosoma):

```
SinglePointCrossover(padre1, padre2, hijo1, tam)
       1. Generamos hijoParcial1
       2. HijoParcial1 <- primera mitad del padre1 + segunda mitad del
          padre2 (ambos de tamaño tam)
Fin
```

Operador de reparación:

```
Reparacion (hijo1, vecDistancias, aporteHijo1)
       1. NumGenes = Sumatoria de 0 a n del número de unos en hijo1.
       Si numGenes == 0
       Si numGenes > m
              2. Ordenar ascendentemente hijo1 en función de los aportes en
              aporteHijo1.
              3. Sesgar hijo1 a m.
              4. Recalcular el cromosoma hijo1 y sus aportes en aporteHijo1
       Si numGenes < m
              5. Calcular aportes de los genes no seleccionados a los
              seleccionados de hijo1.
              6. Ordenar ascendentemente y añadir los h mejores, siendo
              h = m-numGenes.
Fin
```

Estructura de cruce (Esquema general):

```
Cruce
       1. Generar numCruces = 0.6 * m/2
       Desde 0 hasta numCruces
              2. Generar padre1 <- primer elemento en poblacionPadres
              3. Generar padre2 <- siguiente elemento en poblaciónPadres
              4. Generar hijo1, hijo2 y aporteHijo
              5. Cruce a realizar (Cromosoma o sub-cromosoma)
              Si es Sub-cromosoma
                     6. SinglePointCrossover (padre1, hijo1)
                     7. Reparacion(hijo1, vecDistancias, aporteHijo)
                     8. PoblacionIntermedia <- poblacionIntermedia + hijo1
              Si es cromosoma
                     9. SinglePointCrossover (padre1, padre2, hijo1)
                     10. Reparación (hijo1, vecDistancias, aporteHijo)
                     11. PoblacionIntermedia <- poblacionIntermedia + hijo1
                     12. SinglePointCrossover (padre1, padre2, hijo2)
                     13. Reparación (hijo2)
                     14. PoblacionIntermedia <- poblacionIntermedia + hijo2
       Endwhile
       Mientras tamPoblacionIntermedia < m
              15. Recuperación (listaIntermedia, copiaPadre)
       endwhile
Fin
```

Operador de recuperación:

```
Recuperacion (listaIntermedia, copiaPadres)
      1. Generar padre1 <- primer elemento en poblacionPadres
      2. Generar padre2 <- siguiente elemento en poblaciónPadres
      3. Generar hijo1 e hijo2
            4. SinglePointCrossover (padre1, padre2, hijo1)
            5. Reparación (hijo1)
            6. PoblacionIntermedia <- poblacionIntermedia + hijo1
            7. SinglePointCrossover (padre1, padre2, hijo2)
            8. Reparación (hijo2)
            9. PoblacionIntermedia <- poblacionIntermedia + hijo2
      endwhile
Fin
```

Operador de mutación:

```
Mutacion (listaIntermedia)
      1. NumGenesTotales = n*m;
      2. Generar numEsperadoMutaciones = prob.mutacion * numGenesTotales;
      Mientras numEsperadoMutaciones > 0
            3. Generar i aleatoriamente entre (0, m)
            4. Generar Geni entre (0,n)
            5. Generar Genj entre (0,n)
            6. Cromosoma = Seleccionar el listaIntermedia[i].
            Mientras valor en Geni igual a valor en Genj
                  7. Generar Genj
            Endwhile
            8. Intercambiar(Cromosoma[Geni], Cromosoma[Genj])
      endwhile
Fin
```

Estructura de reemplazamiento (utilizada en Improved Sheep Flock Heredity Model):

```
Reemplazamiento
      1. Ordenar cromosomas por aporte en listaSol(Inicial)
      2. mejorSolActual = Seleccionar cromosoma con mejor aporte en listaSol
      3. maxAporteActual = Seleccionar aporte del mejor cromosoma en listaSol.
      4. ComprobarReemplazamiento (mejorSolActual, maxValorSolActual,
         listaIntermedia, listaAportesIntermedia)
      5. ListaSol = listaIntermedia
Fin
```

Operador de comprobación de reemplazamiento (utilizada en Improved Sheep Flock Heredity Model):

```
ComprobarReemplazamiento (mejorSolActual, maxValorSolActual, listaIntermedia,
listaAportesIntermedia)
      1. Ordenar cromosomas por aporte en listaIntermedia
      Si maxValorSolActual > mejorValorlistaIntermedia
            2. Añadimos mejorSolActual a listaIntermedia
            3. Volvemos a ordenar por aportes en ListaIntermedia
            4. Sesgamos a m
            5. Recalculamos listaAportesIntermedia
Fin
```

Enfoque Improved Sheep Flock Heredity Algorithm

Una vez comprendido el funcionamiento del algoritmo que tenía como objetivo de estudio y todas las funciones implementadas y usadas de anteriores implementaciones de algoritmos genéticos, trataré de explicar aquellas mejoras realizadas sobre dicho algoritmo.

En primer lugar, tuve la indecisión de si comenzar aplicando operaciones primeramente a nivel de cromosoma antes que de subcromosoma, y viceversa. Tras varias pruebas de diferentes casos de problemas, con diferentes semillas aleatorias, pude comprobar que efectivamente, la aplicación de estas operaciones a nuestra población a nivel de subcromosoma seguida de una serie de aplicaciones a nivel de cromosoma daba resultados más satisfactorios que en el caso opuesto. Dejando a un lado esta pequeña indecisión, posteriormente me decidí a utilizar la técnica de elitismo ya implementada en anteriores prácticas de la asignatura, la cual tenía en cuenta el valor de *fitness* de nuestra población, para tratar de mantener aquellas "ovejas" que mejor valor tuvieran para promover la diversidad y la generación de soluciones de calidad al mismo tiempo.

Los resultados fueron algo satisfactorios, aunque no suficientes, por lo que traté de realizar un algoritmo memético a partir del *Sheep Flock Heredity Algorithm*, mediante la hibridación de este con la técnica de *Local Search*, técnica que sabemos que aporta gran explotación a las soluciones en un ámbito local, produciendo soluciones aún mucho más satisfactorias en algo más de tiempo, eso sí, pero con una notoria diferencia de calidad respecto al modelo original del algoritmo del rebaño de ovejas.

En cuanto a lo referente a la búsqueda local, me gustaría aclarar primeramente que el número total de iteraciones del algoritmo genético SFHA utilizado para la obtención de resultados y la toma de pruebas con objeto de corrección han sido un total de 15, mientras que el número de iteraciones máximas utilizadas para la LS han sido un máximo de 400. El ínfimo número de operaciones del SFHA se debe a todas las operaciones necesarias para realizar una iteración, lo que produce un consumo de tiempo de ejecución de alrededor de 40/50 segundos por ejecución del algoritmo por cada caso contemplado. Aun así, y como veremos a continuación, las soluciones obtenidas por ambos algoritmos son bastante satisfactorias y de calidad en un tiempo razonable en comparación con los demás algoritmos estudiados en la asignatura.

Esquema de búsqueda BL:

```
BusquedaLocal(vecDistancias, LIMITE, listaSol, listaCandidatos)
     1. Generar paramos <- false
     2. Cambio <- 0
     while (paramos y cambio < limite)
        4. Generar vector aportes(m)
        5. i<-0, j<-0
        Desde it = listaSol-inicio hasta listaSol-final
              6. j = i
              Desde jt = it hasta listaSol-final
                    7. Aportes[i] += vecDistancias[it][jt]
                    8. Aportes[j] += vecDistancias[it][jt]
              Endfor
       Endfor
        9. ordenación de lista de seleccionados y aportes
       10. Generar iterador = aportes-inicio
       11. Generar Mejora <- false
       while (iterador ¡= aportes-final y ¡Mejora)
              12. barajar(listaCandidatos)
              13. Desde i = listaCandidatos-inicio hasta listaCandidatos-final
                    14. quitar candidato a salir de listaSol y buscar mejor candidato
                    en listaCandidatos.
                    Si encontramos candidato en listaCandidatos, mejor que el extraido
                    de listaSol
                          15. Reemplazamos el candidato en listaSol
                          16. Mejora <- true
                    endif
              endfor
              Si noHayMejora
                    17. comprobar siguiente elemento en Sel
                    18. iterador <- iterador++
              endif

       endwhile
              19. cambio <- cambio + 1
    endwhile
 Fin
```

Esquema de búsqueda por ISFHA:

```
Inicio
       Mientras cambio < limite
       1. Ejecución SFHA
       2. Generar contador <- 0 para contabilizar número de BL realizadas
       Mientras contador < m y cambio%7==0 y cambio!=0
              4. Ejecución BL (vecDistancias, Limite=400, listaSol, listaCandidatos)
              6. Actualizar listaAportes de listaSol
              7. contador <- contador+1
       Endwhile
       8. cambio <- cambio + 1
Fin
```

Comparación entre enfoques

Algoritmo	Desviación	Tiempo
Greedy	1,37	203,33
BL	1,04	3,10
AGG-uniform	13,48	60,88
AGG-posicion	12,83	55,75
AGE-uniforme	11,24	53,28
AGE-posicion	12,03	53,42
AM – (10, 1)	3,01	65,31
AM – (10, 0.1)	3,52	61,53
AM – (10, 0.1mej)	3,36	61,39
ES	0,99	32,36
BMB	0,45	0,81
ILS	0,58	0,65
ILS-ES	0,66	119
SFHA	**1,22**	**46,00**
ISFHA	**0,48**	**57,95**

En vista a los resultados, podemos comprobar que claramente merece la pena la implementación de un algoritmo memético, que, aunque consuma algo más de tiempo que el original, produzca resultados bastante diferentes a los producidos por el algoritmo original, siendo los nuevos de mayor calidad y diversidad que los del original.

Durante el desarrollo de esta práctica final he disfrutado bastante tratando de documentarme realizando un pequeño estudio de cómo se genera la diversidad entre los rebaños de ovejas, cómo los pastores realizan esta labor de un modo supervisado y bajo control y sobretodo, cómo podría adaptar dicho comportamiento a prácticamente la mayoría de problemas tipo MDP.

BIBLIOGRAFÍA:

- Guías para la implementación del ISFHA:
https://www.researchgate.net/publication/267232870_An_improved_sheep_flock_heredity_algorithm_for_job_shop_scheduling_and_flow_shop_scheduling_problems

https://shodhganga.inflibnet.ac.in/bitstream/10603/17691/11/11_chapter%206.pdf

- Apoyo al entendimiento de problemas multiobjetivo:

https://pradogrado1920.ugr.es/pluginfile.php/395066/mod_resource/content/11/Sem06-MHs-Multiobjetivo-MHs-19-20.pdf

https://www.researchgate.net/publication/220313106_Multi-stage_genetic_programming_a_new_strategy_to_nonlinear_system_modeling

- Listado de posibles problemas a estudiar:

https://arxiv.org/pdf/2002.08136.pdf

- Wikipedia:

https://es.wikipedia.org/wiki/Algoritmo_mem%C3%A9tico

https://es.wikipedia.org/wiki/Algoritmo_gen%C3%A9tico

- Youtube:

https://www.youtube.com/watch?v=t1LI2DqMGEU

(Este video resultó de gran ayuda para comprender la cría de ovejas)