UNIVERSIDAD DE GRANADA

Metaheurísticas

Práctica 4

Algoritmo Poblacional Cooperativo

Nombre: Francisco Javier Bolívar Expósito

DNI: ---

Email: ---

Grupo de prácticas: ---

Curso 2019-2020

Índice

1. Descripción de la Metaherística Original y justificación

El **Algoritmo Poblacional Cooperativo** es una metaheurística original que se clasifica dentro de la categoría de los '**Social Human Behavior Based Algorithms**' al estar inspirada en un concepto social humano, el **comunismo**. En vez de simular un comportamiento competitivo como muchos otros algoritmos de poblaciones se pretende **simular cooperación** entre la población.

El funcionamiento inicial proviene de un aforismo que resume de forma general los principios de una sociedad comunista, *'De cada cual según sus capacidades, a cada cual según sus necesidades'*, del cuál se obtiene la idea de que cada individuo de una población **aporten sus capacidades** (las partes de la solución que más contribuyan a la función objetivo) y **cubran sus necesidades** sustituyendo las partes con menor contribución por las aportaciones de otras soluciones si estas proporcionan una mejora.

Para **aportar diversidad** se había planteado re-inicializar las soluciones que no aportaran nada a la sociedad **simulando el deber de trabajar**, lo que no tenía mucho sentido ya que entonces se conservarían siempre las mejores soluciones y se reinicializarían en bucle aquellas peores que podrían servir para aportar diversidad.

Sin embargo, tal y como se recomienda en el paper https://arxiv.org/abs/2002.08136, se ha intentado centrar el enfoque de la investigación **en la mejora del comportamiento** y las propiedades del algoritmo, independientemente de si las modificaciones o hibridaciones con componentes de otros algoritmos para potenciarlo nos lleven a un algoritmo que realmente **no sea una metáfora** clara de ninguna inspiración.

Partiendo de la idea de copia de las mejores partes de la solución se ha intentando mejorar el comportamiento del algoritmo, tras un proceso de prueba y error, el algoritmo obtenido posee las siguientes componentes.

Inicialización

Simulando una población de personas crearemos un conjunto de soluciones.

Cada solución será inicializada de forma aleatoria asegurándonos de obtener una solución válida y guardaremos el cálculo de cuánto contribuye cada elemento de la solución a la función objetivo.

Explotación: Copia de mejores a peores

El componente original es la **copia** de los elementos que **más contribuyen a la solución** sustituyendo a aquellos que menos lo hacen si así se mejora una solución. El pseudocódigo sería el siguiente:

```
mejores = p% elementos que más contribuyen ordenados de mayor a menor cont.
peores = p% elementos que menos contribuyen ordenados de menor a mayor cont.

Para cada mejor en mejores && evals != max_evals:
   Para cada peor en peores && evals != max_evals:
        Si copiar Mejor en Peor mantiene una solución valida:
           sol_vec = copiar(mejor, peor)
           nueva_contribución = Contribución de mejor a la solución

           Si nueva_contribución > contribución_peor:
                sol = sol_vec
                Actualizamos las contribuciones a la solución y eliminamos peor de peores
```

Las comprobaciones y cálculos necesarios son similares a los de la búsqueda local primero el mejor, pero al generar los vecinos con elementos que ya sabemos que funcionan bien en otras soluciones tenemos una **mayor probabilidad de generar una mejora**. De esta forma con un menor de evaluaciones la idea es conseguir **converger más rápidamente** aprovechando mejor el tiempo de ejecución. Realmente tiene un efecto similar a como debido al cruce en los algoritmos genéticos se conservan buenas componentes, pero de una manera más directa (lo que también limita el campo de aplicación de la metaheurística).

Explotación: Búsqueda Local Limitada

Tras esta copia de mejores en peores, se consideró añadir simplemente una **búsqueda local** para tras aprovechar un cierto número de intercambios eficientes seguir mejorando las soluciones de la población con elementos que tal vez no se encuentran en esta y así terminar de llegar al óptimo local.

Sin embargo, tras implementar la BL, ¿Por qué no se observaba ninguna diferencia significativa entre usar o no la componente de copia?

Aunque se redujera el número de evaluaciones necesarias para llegar a un óptimo local, al tener la búsqueda local un **número fijo de evaluaciones máximas** si el vecindario a explorar es muy grande las gastaría todas sin aportar ninguna mejora, y por lo tanto, **sin aprovechar la ventaja** de la componente de copia.

Por lo tanto se añadió un **criterio de parada** a la búsqueda local usada en las prácticas anteriores, un **número límite de evaluaciones** seguidas **sin mejora**. De esta forma la BL solo continua gastando evaluaciones de la función objetivo si mejora antes del límite y se corta antes al combinarse con la componente de copia.

Exploración: Re-inicialización o Mutación

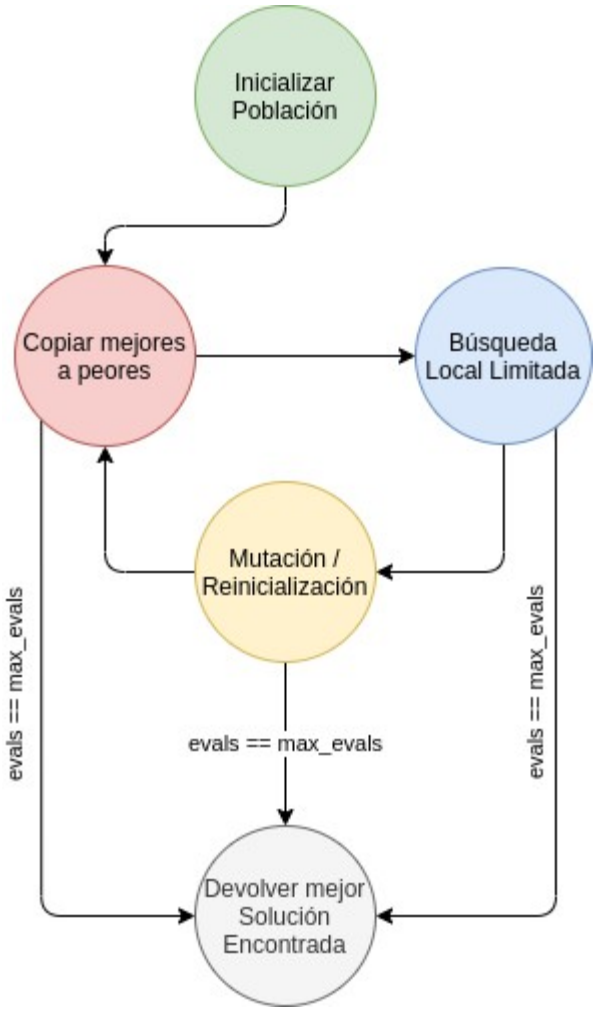
Se espera que de esta forma se consiga una convergencia de las soluciones rápida, pero para mantener un **equilibrio** explotación / exploración es necesario **aportar una diversidad fuerte**, ya que la copia al compartir características reduce la diversidad.

Se modifican un porcentaje de las soluciones **empezando por las mejores**, para evitar quedarnos en la población con soluciones que no puedan mejorar más en óptimos locales y que reduzcan la diversidad de la población al **esparcirse sus elementos** a otras soluciones. Si se modificaran las peores sería muy complicado ganar diversidad. Antes de introducir la componente de diversidad el equilibrio explotación / exploración era muy pobre y se obtenían resultados muy pobres.

Se han probado dos opciones simples para aportar diversidad, **re-inicializar las soluciones** de forma aleatoria o **mutarlas fuertemente**. En el aportado de análisis de resultados veremos cual de estas alternativas proporciona los mejores resultados.

Estructura del Algoritmo

Finalmente, con las componentes indicadas la estructura del algoritmo sería la siguiente:



Como ventajas el algoritmo planteado tiene una **estructura bastante simple**, lo que permitirá analizar más fácilmente su funcionamiento y adaptarlo a cada problema, además de posiblemente un tiempo de ejecución reducido.

También presenta varias oportunidades para su **ejecución en paralelo**, ya sea ejecutando en paralelo las búsquedas locales de cada solución en una iteración (ya que más tarde veremos que la mayor parte del tiempo de ejecución se dedica a la búsqueda local) o teniendo varias poblaciones independientes que cada cierto periodo compartan información (estas posibilidades sin embargo no se han llegado a explorar en la implementación).

La principal desventaja es la dificultad de **encontrar una manera de calcular la contribución** de partes de la solución a esta, limitando su uso a problemas específicos.

Tras obtener resultados experimentales **veremos si su rendimiento es bueno** para el caso del MDP.

2. Descripción del Problema al que se ha adaptado la MH

El problema de la máxima diversidad (**MDP**) consiste en **seleccionar** un **subconjunto** de *m* elementos de un conjunto más grande de *n* de forma que la **diversidad** entre los elementos escogidos se **maximice**.

Para calcular la diversidad consideraremos la variante del problema ***MaxSum***, en la que la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados.

Por lo tanto el problema consistirá en elegir *m* elementos del conjunto inicial de *n* elementos de forma que la suma de las distancias entre ellos sea máxima, siendo la definición matemática:

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

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

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

Xi contiene un 1 si el elemento "i" se ha seleccionado y un 0 si no

MDP es de la clase de problemas **NP-Duros** de optimización combinatoria, por lo que **no es factible** encontrar la solución óptima en un tiempo razonable. Por ello en esta práctica se intenta desarrollar y analizar una metaheurística para encontrar soluciones buenas de forma eficiente, aunque estas no sean las óptimas.

3. Aplicación del algoritmo al problema

3.1 Esquema de representación de soluciones

Se ha usado el mismo esquema de representación de soluciones que en la P1, que usa números enteros por ser más cómodo y eficiente que la representación binaria.

Siendo n = número de elementos en el conjunto inicial, m = número de elementos a seleccionar, la solución se representa como un vector de números enteros que:

- Contiene exactamente m elementos
- No contiene elementos repetidos,
- Cada elemento es un entero perteneciente al intervalo [0, n).

Por lo tanto tenemos un vector solución que contiene los índices de los elementos seleccionados.

Para un problema con n = 10 elementos en el conjunto inicial y m = 5 elementos a seleccionar un vector que represente una posible solución sería por ejemplo: Sol = {2, 4, 7, 8, 9}

3.2. Función objetivo y contribución de cada Elemento

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

La función objetivo evalúa una solución para MDP siguiendo el modelo MaxSum, teniendo como entrada de la solución un vector con la representación decimal usada en la P1.

El objetivo de los algoritmos es maximizar esta función objetivo, que consiste en calcular la suma de distancias entre pares de elementos seleccionados.

Sin embargo, podemos calcularla de otra manera equivalente, primero se calcula la **contribución** de cada elemento seleccionado a la solución, siendo esta la suma de distancias entre este elemento y el resto de elementos de la solución.

$$d_i = \sum_{s_j \in Sel} d_{ij} = d(s_i, Sel)$$

Después se suman las contribuciones de todos los elementos y se dividen entre dos.

Como el APC necesita la contribución de cada elemento a la solución de esta forma podemos saber con los mismos cálculos si una parte contribuye más que otra o si una solución es mejor que otra eficientemente. Como vemos para este problema el **APC es fácilmente adaptable**.

El calculo de las contribuciones se representa en el siguiente pseudocódigo:

calcularContribuciones(solucion, dist):
1. contribuciones = Array de tamaño longitud de la solución

Desde i=0 a tamaño de solucion:
2. contribuciones[i] = Suma de la distancias entre solucion$_i$ y el resto de elem de la sol

3. Devolver contribuciones

3.3. Generación de la Población Inicial

Para generar la población tendremos un parámetro de su cantidad de soluciones.

inicializarSolucion():
1. Sol := elegir m números aleatorios diferentes de [0, n)
2. Devolver Sol

inicializarPoblacion(dist, n_people):

1. population = Matriz n_people x m donde se guardan las soluciones
2. contributions = Matriz n_people x m donde se guradan las contribuciones de cada elemento

Desde i = 0 a n_people:

3. population[i] = inicializarSolucion()
4. contributions[i] = calcularContribuciones(population[i], dist)

5, Devolver population y contributions

3.4. Copia de mejores en peores

De la definición general solo hay que concretar la comprobación de que sustituir un elemento de los peores por uno de los mejores proporciona una solución válida y como se actualizan las contribuciones a la función objetivo. Será necesario elegir el parámetro del porcentaje de soluciones a copiar.

copyBestWorst(dist, population, contributions, p, max_evals):

```
mejores = p% indices de los elementos que más contribuyen ordenados de mayor a menor cont.
peores = p% indices de los elementos que menos contribuyen ordenados de menor a mayor cont.
row_peores = lista que indica a que solución pertenece cada elemento de 'peores'
evals = 0

Para cada mejor en mejores && evals != max_evals:
    Desde i=0 hasta longitud de peores && evals != max_evals:
        Si population[mejor] no está en population[row_peores[i]]:
            sol_vec = population[row_peores[i]]
            sol_vec[peores[i]] = population[mejor]
            evals += 1
            nueva_contribución = Suma de distancias de mejor a cada elemento de sol_vec

            Si nueva_contribución > contributions[peores[i]]:
                contributions[peores[i]] -= Dist de el elem peor al resto de population[row_peores[i]]
                contributions[peores[i]] += Dist de el elem mejor al resto sol_vec
                population[row_peores[i]] = sol_vec

                Eliminar row_peores[i] y peores[i]

Devolver population, contributions, evals
```

3.5 BL

La búsqueda local se conserva igual que en las prácticas anteriores, pero añadiendo el límite de evaluaciones sin mejora.

```
listaDeCandidatos(Sol):
    Elementos = conjunto [0, filas de la matriz de distancias)
    Candidatos = Eliminar de 'Elementos' aquellos valores que estén en 'Sol'
    Devolver Candidatos
```

```
generarVecino(i, elemento_no_seleccionado):
    sol_veci = sol
    sol_vecina[i] = elemento_no_seleccionado
    Devolver sol_vecina
```

```
busquedaLocal(dist, sol, contributions, max_evals, stop_limit):
    evals = 0
    Mientras True
        evals_no_change = 0
        Para cada i en índices de contributions de menor a mayor mientras se encuentre mejora
            Para cada elem en listaDeCandidatos(Sol)
                sol_veci = generarVecino(i, elem)
                Si contribución vecino (suma de distancias de elem a cada elemento de sol_veci) > Z[i]
                    Z = Z  – suma de distancias de cada elemento de sol_veci a sol[i]
                    Z = Z + suma de distancias de cada elemento de sol_veci a elem
                    Z[i] = contribución vecino
                    sol = sol_veci
                Si evals == max_eval o evals_no_change == stop_limit
                    Devolver evals
```

3.6. Mutación Fuerte

Finalmente el último operador a definir será la mutación fuerte de una solución. Consiste en cambiar un porcentaje de elementos (a elegir como parámetro) de la solución por otros que no estén en ella.

```
mutarSolucion(sol, n, p):
    no_seleccionados = Conjunto de 0 a N diferencia con sol

    i = elegir 0.1 * m números aleatorios diferentes de [0, longitud de sol)
    j = elegir 0.1 * m números aleatorios diferentes de [0, longitud de no_seleccionados)

    sol[i] = sol[j]
```

3.7. APC

El pseudocódigo del algoritmo implementado será el siguiente, pudiendo usar tanto el operador mutarSolucion como el operador inicializarSolucion. Se presupone que se pasan a cada operador los parámetros elegidos.

```
apc():
   inicializarPoblacion()
   evals = filas de fitness

   Mientras evals < max_evals
         evals += copyBestWorst()

         Para un porcentaje de sol en las mejores soluciones:
               evals += lsbf(sol, max_evals=max_evals – evals)

         Para un porcentaje de sol en las mejores soluciones:
               sol = mutarSolucion(sol) ó inicializarSolucion()
               Recalculamos fitness para sol y lo contamos en evals

   Devolver mejor solución enontrada hasta el momento
```

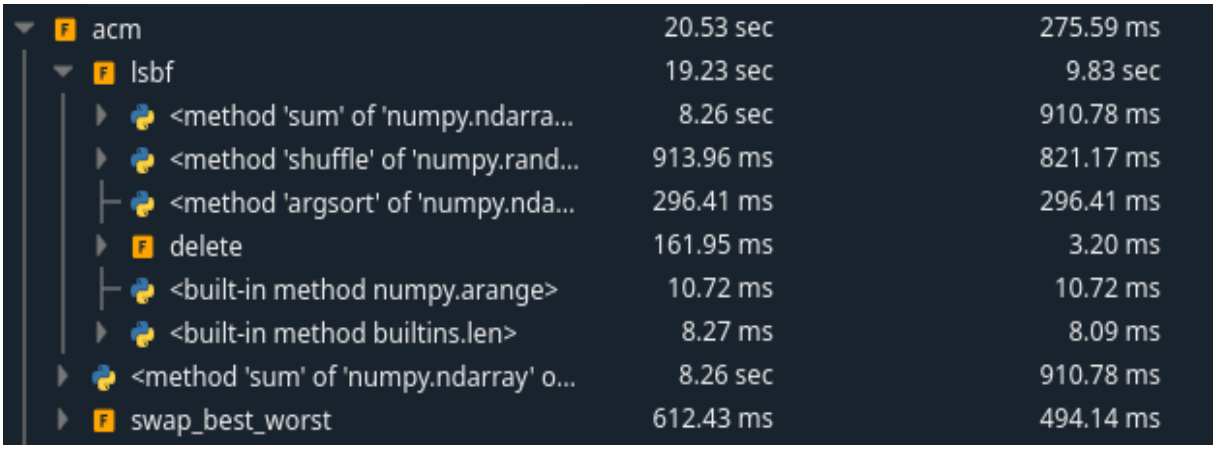
4. Implementación de la práctica y manual de usuario

La implementación de la práctica se ha realizado en **Python**, usando como interprete para las pruebas Python 3.8.1.

Se han utilizado 5 módulos, **time** para medir los tiempos de ejecución, **os** para recorrer el directorio de datos, **numpy** para realizar la mayoría de operaciones, **pandas** para guardar los resultados y generar las hojas de cálculo y **scikit-learn** para probar múltiples parámetros, siendo estos tres últimos los no incluidos en Python.

Se ha utilizado un '**profiler**' para detectar las partes del código que más tiempo de ejecución ocupaban y optimizarlas.

La componente de copia requería mucho tiempo de ejecución, por comprobar si el mejor ya se encontraba en la solución donde se iba a sustituir un peor, este problema se ha mitigado guardando un set por cada solución de la que se tendrá que hacer esta comprobación. También aumentaba su tiempo por problemas al indicar que elementos peores ya se habían intercambiado al usar arrays, que se ha solucionado fácilmente con una implementación con varias listas. Tras optimizar esta parte del código **prácticamente no consume tiempo** excepto por los cálculos de la función objetivo.



La conclusión obtenida es que el mayor tiempo de ejecución **se dedica a la búsqueda local**, y en general en el cálculo de las contribuciones y la función objetivo.

Se ha intentado conseguir un buen tiempo de ejecución, para reducir lo máximo posible el tiempo de cálculo de la función objetivo y la generación de números aleatorios se ha intentando realizar siempre las **operaciones de forma vectorizada** con los arrays de Numpy, **evitando copias innecesarias** de estos vectores optimizando el acceso a memoria y generando los números aleatorios con Numpy en grandes cantidades.

Para usar el programa es necesario instalar NumPy y Pandas, y scikit-learn solo si se quiere hacer un estudio de parámetros. Se incluye un archivo **requirements.txt** que se puede usar para instalar las versiónes usadas, utilizando “pip” con ‘pip install -r requirements.txt’.

Todo está contenido en un único script **mdp_apc.py** que cuelga del directorio **FUENTES.**

El programa proporciona como salida los resultados para los mejores parámetros encontrados para cada versión del algoritmo en un archivo csv. Simplemente será necesario ejecutar el script y se guardaran los resultados para todos los casos de ejecución contenidos en el directorio **BIN** en el directorio raíz de la práctica. También se encuentra programada una prueba de distintos parámetros de los algoritmos, como es ajeno al uso del algoritmo y requiere un gran tiempo de ejecución se ha comentado esta parte del código.

5. Análisis de los resultados

Para todos los algoritmos se ha usado la semilla 510 y 100.000 evaluaciones de la función objetivo.

Para cada parámetro se han probado 2 o 3 posibilidades que parecían razonables intuitivamente, calculando la desviación media para cada una de las combinaciones resultantes. Se muestran a continuación las 10 combinaciones de parámetros que mejor desviación media han proporcionado y las 10 combinaciones que peor desviación media.

p_swap	p_mut_pop	p_ls	n_people	ls_th	desv. media
0.3	1.0	0.4	5	1000	0.49386313213517724
0.3	0.4	0.2	5	1000	0.5014858242869745
0.3	0.4	0.2	25	1000	0.5346564833075985
0.3	0.8	0.4	25	1000	0.535917705440211
0.3	0.8	0.2	25	1000	0.5449482974053134
0.3	0.4	0.4	5	1000	0.5450030352142629
0.1	1.0	0.4	25	1000	0.5454682251122563
0.1	0.8	0.4	25	1000	0.5459983826873349
0.1	0.4	0.4	5	1000	0.5471716425183535
0.1	0.4	0.4	25	1000	0.5493334510088318

APC con Reinicialización - 10 mejores

0.1	1.0	0.4	25	10000	0.6834374746590713
0.1	0.4	0.4	25	10000	0.6834374746590713
0.1	0.8	0.4	25	10000	0.6834374746590713
0.1	0.8	0.4	5	10000	0.683903167116459
0.3	1.0	0.2	25	10000	0.6842939459778358
0.1	1.0	0.2	25	10000	0.6916251355174662
0.1	0.8	0.2	25	10000	0.7024057841219894
0.1	0.4	0.2	5	10000	0.71781752524985
0.1	1.0	0.2	5	10000	0.7267550534044644
0.1	0.8	0.2	5	10000	0.7554767846517585

APC con Reinicialización - 10 peores

p_swap	p_mut_sol	p_mut_pop	p_ls	n_people	ls_th	desv. media
0.1	0.3	0.8	0.2	5	1000	0.25402102544643146
0.3	0.3	0.8	0.2	5	1000	0.3046128303594439
0.1	0.3	1.0	0.2	5	1000	0.3222439244586471
0.1	0.3	1.0	0.4	5	1000	0.3366086288002084
0.1	0.3	0.4	0.2	5	1000	0.3439211546451439
0.3	0.3	1.0	0.2	5	1000	0.3460335329497538
0.1	0.3	0.4	0.4	5	1000	0.3498838803069856
0.3	0.3	0.4	0.4	5	1000	0.35073758217773693
0.1	0.3	0.8	0.4	5	1000	0.37022017387567646
0.3	0.3	1.0	0.4	5	1000	0.38754435073552146

APC con Mutación - 10 mejores

0.3	0.3	0.8	0.4	25	10000	0.6760420113556759
0.3	0.6	1.0	0.4	25	10000	0.6760420113556759
0.3	0.3	0.4	0.4	25	10000	0.6760420113556759
0.3	0.6	0.4	0.4	25	10000	0.6760420113556759
0.1	0.3	0.8	0.4	25	10000	0.6834374746590713
0.1	0.6	0.8	0.4	25	10000	0.6834374746590713
0.1	0.3	0.4	0.4	25	10000	0.6834374746590713
0.1	0.3	1.0	0.4	25	10000	0.6834374746590713
0.1	0.6	1.0	0.4	25	10000	0.6834374746590713
0.1	0.6	0.4	0.4	25	10000	0.6834374746590713

APC con Mutación - 10 peores

Vemos como el **corte temprano** de la búsqueda local favorece al algoritmo como se pretendía, ya que en ambas versiones del algoritmo un **límite más duro** de 1000 evaluaciones sin mejora para cortar **proporciona los mejores resultados**. Puede deberse tanto a aprovechar mejor la rapidez en la convergencia que aporta la componente de copia como ha aumentar la frecuencia de modificación de las soluciones, aportando mayor diversidad.

En la versión de **re-inicialización** se observa que una proporción de copia del 30% de la población proporciona los mejores resultados, cosa que no parece ser tan relevante en la mutación. Al reinicializar completamente las soluciones es más probable caer más lejos del óptimo que con las mutaciones aplicadas, por lo que este extra de intensificación puede ayudar en este caso.

En la versión de mutación se observa que los mejores resultados se consiguen en **poblaciones pequeñas** de tan solo 5 soluciones, puede ser porque esta versión con su similitud a las búsquedas por trayectorias consigue suficiente diversidad con este número reducido. Claramente una mutación fuerte del 30% aporta muy buenos resultados, pero del 60% resulta demasiado fuerte.

Se esperaba ver una mayor diferencia entre modificar el 80% de la población o el 100%, ya que mutar el 100% **no deja intacta ninguna solución** de las explotadas en la iteración anterior de la que se podrían copiar características. Sin embargo al mutar solo el 30% de la solución parece no tener tanta relevancia como se esperaba (aunque los 2 mejores resultados siguen siendo modificando el 80% de la población).

La versión con **mutación** consigue unos **resultados mejores** que la de reinicialización, sus peores son menos malos y sus mejores son más buenos, con una diferencia lo suficientemente significativa como para concluir que está versión es mejor.

Podría ser interesante investigar otras componentes que aporten diversidad. Se considero el cruce de los algoritmos genéticos, pero la copia de características ya conserva parte de las soluciones y el cruce después de un tiempo de ejecución llevaría a una diversidad muy pequeña en la población como en los algoritmos meméticos, por esta razón no se decidió implementarlo, aunque tal vez combinando el cruce con otras estrategias de diversificación podría haber dado resultados positivos pero añadiendo una complejidad bastante mayor. Investigar la hibridación con componentes de CHC que si aportan una alta diversidad podría dar buenos resultados.

Tras estos pequeños comentarios sobre los parámetros y las dos versiones del algoritmo se procede a mostrar los resultados para cada caso de las dos versiones del algoritmo con los mejores parámetros y comparaciones con otros algoritmos implementados en la asignatura, todos utilizando siempre 100.000 evaluaciones de la función objetivo.

Algoritmo Poblacional Cooperativo – Mutación			
Caso	**Coste Obtenido**	**Desv**	**Tiempo**
GKD-c_11_n500_m50	19587,12018	0,00	0,55992436
GKD-c_12_n500_m50	19360,22393	0,00	0,5472858
GKD-c_13_n500_m50	19366,69852	0,00	0,57022882
GKD-c_14_n500_m50	19458,56466	0,00	0,56702209
GKD-c_15_n500_m50	19422,1464	0,00	0,56954384
GKD-c_16_n500_m50	19680,20477	0,00	0,56529546
GKD-c_17_n500_m50	19331,38841	0,00	0,5716064
GKD-c_18_n500_m50	19461,3946	0,00	0,57317019
GKD-c_19_n500_m50	19477,32577	0,00	0,55873322
GKD-c_20_n500_m50	19604,84356	0,00	0,5608952
MDG-b_1_n500_m50	777890,27	0,02	0,55518985
MDG-b_2_n500_m50	779963,54	0,00	0,5518415
MDG-b_3_n500_m50	774256,41	0,32	0,56641173
MDG-b_4_n500_m50	774284,6	0,14	0,53820705
MDG-b_5_n500_m50	773236,45	0,31	0,53969979
MDG-b_6_n500_m50	774909,14	0,03	0,54512715
MDG-b_7_n500_m50	775564,88	0,21	0,53473687
MDG-b_8_n500_m50	779168,62	0,00	0,55023479
MDG-b_9_n500_m50	772348,09	0,32	0,5414598
MDG-b_10_n500_m50	773966,97	0,13	0,54110074
MDG-a_31_n2000_m200	113440	0,61	0,78299785
MDG-a_32_n2000_m200	113512	0,51	0,71738744
MDG-a_33_n2000_m200	113641	0,42	0,75717759
MDG-a_34_n2000_m200	113338	0,76	0,75634098
MDG-a_35_n2000_m200	113317	0,76	0,75610328
MDG-a_36_n2000_m200	113689	0,49	0,78806853
MDG-a_37_n2000_m200	113330	0,77	0,74461126
MDG-a_38_n2000_m200	113782	0,52	0,73512053
MDG-a_39_n2000_m200	113309	0,78	0,75198364
MDG-a_40_n2000_m200	113605	0,51	0,73424077
		0,254021025	0,621058218

Algoritmo Poblacional Cooperativo – Reinicialización			
Caso	**Coste Obtenido**	**Desv**	**Tiempo**
GKD-c_11_n500_m50	19587,12018	0,00	0,56175947189331
GKD-c_12_n500_m50	19360,22393	0,00	0,565002918243408
GKD-c_13_n500_m50	19366,69852	0,00	0,571977615356445
GKD-c_14_n500_m50	19458,56466	0,00	0,562175989151001
GKD-c_15_n500_m50	19422,1464	0,00	0,555885791778564
GKD-c_16_n500_m50	19680,20477	0,00	0,567338705062866
GKD-c_17_n500_m50	19331,38841	0,00	0,583657503128052
GKD-c_18_n500_m50	19461,3946	0,00	0,567113637924194
GKD-c_19_n500_m50	19477,32577	0,00	0,560760021209717
GKD-c_20_n500_m50	19604,84356	0,00	0,583845376968384
MDG-b_1_n500_m50	772255,95	0,74	0,564834594726563
MDG-b_2_n500_m50	776550,08	0,44	0,55630898475647
MDG-b_3_n500_m50	771284,78	0,71	0,543460369110107
MDG-b_4_n500_m50	770039,81	0,69	0,545430183410645
MDG-b_5_n500_m50	774093,45	0,20	0,54151725769043
MDG-b_6_n500_m50	772260,48	0,37	0,539308547973633
MDG-b_7_n500_m50	774913,07	0,30	0,541873693466187
MDG-b_8_n500_m50	773724,51	0,70	0,551995038986206
MDG-b_9_n500_m50	770891,29	0,50	0,548358678817749
MDG-b_10_n500_m50	770718,83	0,55	0,536827325820923
MDG-a_31_n2000_m200	113044	0,96	0,786602735519409
MDG-a_32_n2000_m200	113192	0,79	0,821762084960938
MDG-a_33_n2000_m200	113029	0,96	0,799296855926514
MDG-a_34_n2000_m200	113101	0,96	0,776726007461548
MDG-a_35_n2000_m200	113018	1,02	0,759539365768433
MDG-a_36_n2000_m200	113395	0,75	0,818631649017334
MDG-a_37_n2000_m200	113101	0,97	0,758532047271729
MDG-a_38_n2000_m200	113257	0,98	0,733278751373291
MDG-a_39_n2000_m200	112800	1,23	0,740627288818359
MDG-a_40_n2000_m200	113049	1,00	0,780584573745728
		0,49386313	0,630833768844605

Algoritmo	Desv	Tiempo
Greedy	1,3654	0,00356
BL	0,94706	0,2784
Greedy + BL	0,99049	0,25898
AGG_UO	1,11899	30,2204
AGG_UE	0,80961	38,341
AGG_P	2,74647	13,3015
AGE_UO	1,10216	21,0332
AGE_UE	1,0178	25,9948
AGE_P	1,46029	20,4048
AM_10_1_UE	0,92689	1,74056
AM_10_01_UE	0,82735	6,18934
AM_10_01mej_UE	0,78948	2,00355
AM_10_1_UO	1,09182	1,97166
AM_10_01_UO	1,11749	1,98165
AM_10_01mej_UO	0,74543	7,43613
ES	1,32172	0,56407
BMB	0,72664	0,58009
ILS	0,52824	0,51868
ILS-ES	0,85646	1,12506
APC-M	0,25402	0,62106
APC-R	0,49386	0,63083

¡Se obtienen **mejores resultados** que en el **resto de algoritmos** implementados en prácticas! Sobre todo en el APC-Mutación que tiene una desviación media de solo **0.254%** frente al mejor algoritmo de prácticas (ILS con búsqueda local) que tiene **0.528%**.

Además, como hemos visto anteriormente el tiempo de ejecución se dedica casi exclusivamente a la búsqueda local con el cálculo de la función objetivo factorizada, lo que proporciona un **tiempo de ejecución muy reducido**. Siendo un **algoritmo muy simple** obtiene buenos resultados, sin mostrar una especial dificultad o facilidad para ningún tipo concreto del problema (obviamente los casos de mayor tamaño obtienen peores resultados)

Supera tanto en tiempo como en resultados a todos los algoritmos meméticos y supera al mejor algoritmo (ILS-BL) con un tiempo muy similar.

También se ha hecho unas pequeñas pruebas para estudiar **la escalabilidad**, probando el APC-M con 50.000 evaluaciones y con 300.000 evaluaciones. Así podemos observar que el algoritmo conserva un **buen equilibrio** exploración / explotación a **lo largo de las iteraciones**, a diferencia de los meméticos, por lo que al cambiar el número de evaluaciones escala correctamente.

Algoritmo Poblacional Cooperativo – Mutación(50.000 evals)			
Caso	**Coste Obtenido**	**Desv**	**Tiempo**
GKD-c_11_n500_m50	19587,12018	0,00	0,289173603
GKD-c_12_n500_m50	19360,22393	0,00	0,284993887
GKD-c_13_n500_m50	19366,69852	0,00	0,290518999
GKD-c_14_n500_m50	19458,56466	0,00	0,292820692
GKD-c_15_n500_m50	19422,1464	0,00	0,287569284
GKD-c_16_n500_m50	19680,20477	0,00	0,298054218
GKD-c_17_n500_m50	19331,38841	0,00	0,29076314
GKD-c_18_n500_m50	19461,3946	0,00	0,28792429
GKD-c_19_n500_m50	19477,32577	0,00	0,279278278
GKD-c_20_n500_m50	19604,84356	0,00	0,285269976
MDG-b_1_n500_m50	771292,83	0,87	0,273779631
MDG-b_2_n500_m50	779963,54	0,00	0,309068441
MDG-b_3_n500_m50	769359,28	0,95	0,279470921
MDG-b_4_n500_m50	774284,6	0,14	0,283094168
MDG-b_5_n500_m50	773236,45	0,31	0,276847839
MDG-b_6_n500_m50	772896,06	0,29	0,282467127
MDG-b_7_n500_m50	766289,61	1,41	0,288223743
MDG-b_8_n500_m50	775496,75	0,47	0,279997349
MDG-b_9_n500_m50	769699,54	0,66	0,274867773
MDG-b_10_n500_m50	772946,9	0,26	0,280899048
MDG-a_31_n2000_m200	113244	0,78	0,3983953
MDG-a_32_n2000_m200	113220	0,76	0,381247997
MDG-a_33_n2000_m200	113452	0,59	0,395916462
MDG-a_34_n2000_m200	112889	1,15	0,394384861
MDG-a_35_n2000_m200	113045	0,99	0,414612055
MDG-a_36_n2000_m200	113589	0,58	0,406620264
MDG-a_37_n2000_m200	113054	1,01	0,410608053
MDG-a_38_n2000_m200	113608	0,67	0,399129868
MDG-a_39_n2000_m200	112913	1,13	0,406035185
MDG-a_40_n2000_m200	113545	0,57	0,406095743
		0,453402927	0,32427094

Algoritmo Poblacional Cooperativo – Mutación(300.000 evals)			
Caso	**Coste Obtenido**	**Desv**	**Tiempo**
GKD-c_11_n500_m50	19587,12018	0,00	1,702629566
GKD-c_12_n500_m50	19360,22393	0,00	1,652034283
GKD-c_13_n500_m50	19366,69852	0,00	1,712421656
GKD-c_14_n500_m50	19458,56466	0,00	1,754395485
GKD-c_15_n500_m50	19422,1464	0,00	1,719398975
GKD-c_16_n500_m50	19680,20477	0,00	1,778457403
GKD-c_17_n500_m50	19331,38841	0,00	1,68274188
GKD-c_18_n500_m50	19461,3946	0,00	1,733003139
GKD-c_19_n500_m50	19477,32577	0,00	1,705844402
GKD-c_20_n500_m50	19604,84356	0,00	1,701651096
MDG-b_1_n500_m50	778030,57	0,00	1,705351114
MDG-b_2_n500_m50	779963,54	0,00	1,712192059
MDG-b_3_n500_m50	775511,98	0,16	1,741528511
MDG-b_4_n500_m50	774957,58	0,06	1,649224281
MDG-b_5_n500_m50	775610,96	0,00	1,66425705
MDG-b_6_n500_m50	775081,04	0,01	1,659719229
MDG-b_7_n500_m50	777232,88	0,00	1,653734684
MDG-b_8_n500_m50	779168,62	0,00	1,666186571
MDG-b_9_n500_m50	772348,09	0,32	1,643175364
MDG-b_10_n500_m50	774259,92	0,09	1,672258139
MDG-a_31_n2000_m200	113745	0,35	2,283876896
MDG-a_32_n2000_m200	113715	0,33	2,359479189
MDG-a_33_n2000_m200	113914	0,18	2,36187911
MDG-a_34_n2000_m200	113606	0,52	2,339020729
MDG-a_35_n2000_m200	113596	0,51	2,271746397
MDG-a_36_n2000_m200	114181	0,06	2,278388023
MDG-a_37_n2000_m200	113494	0,63	2,311555624
MDG-a_38_n2000_m200	113891	0,43	2,28896904
MDG-a_39_n2000_m200	113466	0,64	2,287474394
MDG-a_40_n2000_m200	113836	0,31	2,282280445
		0,153358944	1,899162491

Con la **mitad de evaluaciones** que el resto de algoritmos y un tiempo más cercano de la búsqueda local que de ningún otro **seguimos obteniendo el mejor resultado**, mostrando la eficiencia del algoritmo. Y si añadimos un mayor número de evaluaciones la solución sigue mejorando, aunque no linealmente.

Finalmente también se ha considerado relevante ver los resultados que proporciona el algoritmo **sin la componente de copia**, ya que esta podría reducir la diversidad y cuando se estaba empezando a implementar la práctica no se observaba que está proporcionara una mejora, si no todo lo contrario.

Algoritmo Poblacional Cooperativo – Mutación Sin Copia			
Caso	**oste Obtenic**	**Desv**	**Tiempo**
GKD-c_11_n500_m50	19587,12018	0,00	0,533421755
GKD-c_12_n500_m50	19360,22393	0,00	0,533498287
GKD-c_13_n500_m50	19366,69852	0,00	0,541678429
GKD-c_14_n500_m50	19458,56466	0,00	0,539567709
GKD-c_15_n500_m50	19422,1464	0,00	0,522727013
GKD-c_16_n500_m50	19680,20477	0,00	0,529358625
GKD-c_17_n500_m50	19331,38841	0,00	0,533056736
GKD-c_18_n500_m50	19461,3946	0,00	0,534056902
GKD-c_19_n500_m50	19477,32577	0,00	0,534234762
GKD-c_20_n500_m50	19604,84356	0,00	0,53540206
MDG-b_1_n500_m50	770678,38	0,94	0,542438269
MDG-b_2_n500_m50	769894,29	1,29	0,543371677
MDG-b_3_n500_m50	774747,09	0,26	0,537825823
MDG-b_4_n500_m50	774821,09	0,07	0,527902126
MDG-b_5_n500_m50	774953,21	0,08	0,522265434
MDG-b_6_n500_m50	774864,58	0,04	0,530363083
MDG-b_7_n500_m50	776405,36	0,11	0,530771494
MDG-b_8_n500_m50	771154,04	1,03	0,530710936
MDG-b_9_n500_m50	773102,53	0,22	0,52880621
MDG-b_10_n500_m50	772573,27	0,31	0,535687923
MDG-a_31_n2000_m200	113504	0,56	0,734916687
MDG-a_32_n2000_m200	113626	0,41	0,757445574
MDG-a_33_n2000_m200	113461	0,58	0,763593674
MDG-a_34_n2000_m200	113389	0,71	0,755260944
MDG-a_35_n2000_m200	113424	0,66	0,757683754
MDG-a_36_n2000_m200	113924	0,29	0,769475937
MDG-a_37_n2000_m200	113370	0,74	0,836158514
MDG-a_38_n2000_m200	113797	0,51	0,742139339
MDG-a_39_n2000_m200	113716	0,42	0,784792662
MDG-a_40_n2000_m200	113566	0,55	0,715050459
		0,32602839368847	0,609455427

Si eliminamos la componente de copia **los resultados empeoran**, por lo que tras implementar el algoritmo completo que potencia sus ventajas y palia sus desventajas **esta resulta útil**. Aunque los resultados sin ella tampoco son malos.

Los resultados de la metaheurística propuesta han sido mucho mejores de lo que esperaba inicialmente e intentar mejorar una metaheurística sin restricciones con los conocimientos aprendidos durante el curso es una experiencia interesante y buena para el aprendizaje al hacerte razonar sobre las consecuencias de cada componente de cada metaheurística.

Me hubiera gustado poder comparar el APC con otras metaheurísticas más avanzadas que se han visto en teoría u otras que se consideren especialmente eficientes. Sin embargo, no he conseguido realizar esta comparación, ya que cuando conseguía acceder a algún paper de investigación sobre este problema la limitación impuesta a las metaheurísticas era tiempo, ya fuera 1 minuto o 10 segundos, haciendo prácticamente imposible la comparación al disponer de máquinas e implementaciones diferentes.

Como conclusiones, no perder el **equilibrio entre intensidad y diversidad** es clave, la **simplicidad de una metaheurística** es una propiedad relevante ya que facilita tanto su mejora por la facilidad de modificación y compresión como un **tiempo de ejecución reducido** si las técnicas para explotar / explorar son simples, la **elección de parámetros** tiene un gran efecto sobre los resultados, y finalmente, **comprender y utilizar las ventajas** de otras metaheurísticas es tremendamente útil.