METAHEURÍSTICAS

PRÁCTICA 2.A: TÉCNICAS DE BÚSQUEDA BASADAS EN POBLACIONES PARA EL PROBLEMA DE LA MÁXIMA DIVERSIDAD

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 2 – TÉCNICAS DE BÚSQUEDA BASADAS EN POBLACIONES PARA EL PROBLEMA DE LA MÁXIMA DIVERSIDAD

Con el fin de llevar a cabo el correcto desarrollo de la segunda práctica de la asignatura, se nos pide el estudio del funcionamiento de *Técnicas de Búsqueda basadas en Poblaciones,* en concreto, en la resolución del problema de la máxima diversidad (MDP) descrito en el Seminario 2.

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 Algoritmos Genéticos

Primeramente, trataremos brevemente de explicar en qué consiste un algoritmo genético, para posteriormente especificar qué casos de algoritmos genéticos estamos tratando de implementar y su estructura y representación. 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 mediante una selección por Torneo que explicaremos más adelante), en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados.

Los algoritmos genéticos se enmarcan dentro de los algoritmos evolutivos, que incluyen también las estrategias evolutivas, la programación evolutiva y la programación genética. Por otra parte, encontramos dos tipos bien diferenciados de algoritmos genéticos:

- **Algoritmos Genéticos Generacionales:** Este tipo de algoritmo genético se caracteriza por seguir la estructura clásica de la evolución genética (selección, cruce, mutación y reemplazamiento), salvo que difiere de otros algoritmos genéticos en la generación de la nueva población. Me explico, los algoritmos Generacionales se caracterizan principalmente por tratar de generar una población completamente nueva de individuos a partir de la solución actual en la que nos encontramos, mediante el uso de un tipo de selección por **Torneo Binario**, que garantiza la inclusión de los mejores conjuntos de soluciones de la población actual. Tras haber seleccionado los candidatos a evolucionar (los que serán nuestros padres a cruzar posteriormente), utilizaremos, en función del tipo de cruce que queramos usar (**Cruce posición ó Cruce Uniforme**), podremos o bien elegir aquellos genes en los que ambos padres coinciden y posteriormente elegir el resto de genes de uno de los padres e incluirlos en el hijo con un orden aleatorio; o bien elegir de igual manera los genes en los que ambos padres coinciden y rellenar nuestra solución con valores aleatorios (o bien provenientes del padre 1 o 2), aunque este segundo operador de cruce requerirá de una **reparación** posteriormente, ya que producirá soluciones que no serán factibles, al contrario que nuestro primer operador de cruce. Ambos operadores se tendrán en cuenta con una probabilidad de 0.6, siendo así, $(0.6 \cdot m/2)$ el número de cruces esperado sobre una población de padres, y siendo m el número de genes que hacen factibles la solución generada. Tras haber producido una población de candidatos hijos factible a partir de cruces de 2 padres seleccionados como ya hemos comentado, tendremos que aplicarles un operador de **mutación** con una probabilidad de 0.01, aunque trataremos de evitar la generación de números aleatorios excesivos prediciendo al igual que en el caso de los cruces, un número de mutaciones “a priori” condicionado por la esperanza matemática

  Nºmutaciones = Probabilidad de mutación * número de Genes en la población

  Por último, realizaremos un **reemplazamiento elitista** que consistirá en la comprobación del aporte del primer mejor cromosoma de nuestra población actual respecto al primer mejor cromosoma de nuestra población nueva, en caso de ser el actual mejor que el nuevo, se incluiría en la nueva población, reordenaríamos nuestros cromosomas por aporte y realizaríamos un sesgo para ajustarlo a m (tamaño de solución factible).

- **Algoritmos Genéticos Estacionarios:** Este tipo de algoritmo genético, es bien diferenciado de los genéticos generacionales, ya que estos solo realizarán una selección por Torneo, obteniendo así una población nueva de 2 individuos (padre1 y padre2), que posteriormente serán cruzados con probabilidad 1, generando 2 descendientes a partir de dichos padres (hijo1 e hijo2). A continuación, les aplicaremos a ambos hijos el operador de **mutación** que consistirá básicamente en seleccionar aleatoriamente dos genes de cada uno de los cromosomas, que deben tener valores distintos (seleccionado y no seleccionado) para poder realizar un intercambio, incluyendo aquel no incluido y descartando aquel incluido en caso de realizarse la mutación, generando de igual forma soluciones plenamente factibles. Por último, evaluaremos mediante la función objetivo el coste de cada uno de los hijos tratando de incluir el mejor de ellos en nuestra población actual, considerando estos dos hijos como la población nueva.

Representación de la solución:

Utilizaremos un tipo de representación binaria, mediante una lista de tamaño m de listas de la forma Sel = (x1, … , x_n) en el que las posiciones de los elementos en Sel serán elementos de nuestra solución, y 0 o 1 el valor correspondiente a no seleccionado o seleccionado, respectivamente. 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)
              3. listaPadres = TorneoBinario(listaSol, aportesSol, numCompetidores)
              4. listaIntermedia = Cruce Uniforme o Cruce Posición (padre1, padre2,
              hijo1, hijo2, m)
              5. Reparación (hijo1, hijo2)
              6. Recuperación(listaIntermedia)
              7. Mutacion(listaIntermedia)
              8. Reemplazamiento (mejorSolActual, maxValorSolActual, listaIntermedia,
              listaAportesIntermedia)
              9. CalculoCosteFinal (listaIntermedia)
              10. 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 por Torneo.
- ListaIntermedia es una lista de cromosomas obtenido tras realizar cruces a la población de padres.
- MejorSolActual es el mejor cromosoma de listaSol(Actual) y maxValorSolActual su aporte.

Inicialización de la solución:

```
Inicio(Sol, m)
       Desde 0
       1. Generar i aleatoriamente entre (0, n)
          Si Sol[i]!=1
             3. Sol[i]=1
          Sino
             Mientras Sol[i]!= 1
             5. Generar nuevamente i
       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 convertida de representación binaria a representación entera.
- VecDistancias es un vector con todas las distancias del conjunto de datos actual.

Operador de selección:

```
TorneoBinario(listaSol, aportesSol, numPadres)
       1. Creamos una listaPadres vacía
       Desde 0 hasta numPadres
              Mientras i!=j
                     2. Generar i aleatoriamente entre (0, m)
                     3. Generar j aleatoriamente entre (0, m)
              Endwhile
              4. Seleccionar candidato1 <- listaSol[i] y aporte1 <-
              aportesSol[i]
              5. Seleccionar candidato2 <- listaSol[j] y aporte2 <-
              aportesSol[j]
              Si aportesSol[i] > aportesSol[j]
                     ListaPadres <- listaPadres + listaSol[i]
       Endfor
Fin
```

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 Posición:

```
Cruce-posicion (padre1, padre2, hijo1, hijo2)
      1. Generamos HijoParcial1, tamaño n y rellenamos con -1
      Desde i = inicio-padre1 hasta fin-padre1
         Si padre1[i] == padre2[i]
            2. HijoParcial1[i] <- valor coincidente en ambos padres
         Sino
            3. RestosPadre1 <- valorEnPadre1
      Endfor
      4. Generamos HijoParcial2 = hijoParcial1
      5. Generamos RestosPadre2 = restosPadre1
      6. Barajar(restosPadre1)
      Desde i = inicio-hijoparcial1 hasta fin-hijoparcial1
         Si HijoParcial[i] == -1
            7. HijoParcial[i] <- restosPadre[i]
      Endfor
      8. Barajar(restosPadre2)
      Desde i = inicio-hijoparcial2 hasta fin-hijoparcial2
         Si HijoParcial2[i] == -1
            9. HijoParcial2[i] <- restosPadre2[i]
      Endfor
      10. Hijo1 = hijoParcial1
      11. Hijo2 = hijoParcial2
Fin
```

Cruce Uniforme:

```
Cruce-uniforme (padre1, padre2, hijo1)
      1. Generamos HijoParcial1, tamaño n y rellenamos con -1
      Desde i = inicio-padre1 hasta fin-padre1
         Si padre1[i] == padre2[i]
            2. HijoParcial1[i] <- valor coincidente en ambos padres
         Sino
            3. RestosPadre1 <- valorEnPadre1
      Endfor
      Desde i = inicio-padre1 hasta fin-padre1
         4. Generamos i aleatoriamente entre (0,100)
         Si i < 100/2
            5. HijoParcial1[i] <- valorEnPadre1
            6. HijoParcial1[i] <- valorEnPadre2
      Endfor
      7. Hijo1 = HijoParcial1
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. Cambiar representación de binaria a entera de hijo1.
              3. Ordenar ascendentemente hijo1 en función de los aportes en
              aporteHijo1.
              4. Sesgar hijo1 a m.
              5. Recalcular el cromosoma hijo1 y sus aportes en aporteHijo1
       Si numGenes < m
              6. Cambiar representación de binaria a entera de hijo1.
              7. Calcular aportes de los genes no seleccionados a los
              seleccionados de hijo1.
              8. 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 e hijo2
              5. Cruce a realizar (Posicion o Uniforme)
              Si es posición
                     6. Cruce-posicion (padre1, padre2, hijo1, hijo2)
                     7. PoblacionIntermedia <- poblacionIntermedia + hijo1
                     8. PoblacionIntermedia <- poblacionIntermedia + hijo2
              Si es uniforme
                     6. Cruce-uniforme (padre1, padre2, hijo1)
                     7. Reparación (hijo1)
                     8. PoblacionIntermedia <- poblacionIntermedia + hijo1
                     9. Cruce-uniforme (padre1, padre2, hijo2)
                     10. Reparación (hijo2)
                     11. PoblacionIntermedia <- poblacionIntermedia + hijo2
       Endwhile
       Mientras tamPoblacionIntermedia < m
              12. 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. Cruce-uniforme (padre1, padre2, hijo1)
            5. Reparación (hijo1)
            6. PoblacionIntermedia <- poblacionIntermedia + hijo1
            7. Cruce-uniforme (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:

```
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:

```
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 Algoritmos Meméticos

A continuación, trataremos de explicar en qué consiste un algoritmo memético, para posteriormente analizar qué casos de algoritmos meméticos estamos tratando de implementar y su estructura y representación. Estos algoritmos están caracterizados por ser técnicas de optimización que se basan en la hibridación sinérgica de conceptos tomados de otras metaheurísticas, como pueden ser las búsquedas basadas en poblaciones mencionadas anteriormente (como los algoritmos genéticos, entre otros algoritmos evolutivos), y la mejora local (como las técnicas de Búsqueda Local estudiadas para llevar a cabo el desarrollo de la primera práctica de la asignatura.

Los algoritmos meméticos son metaheurísticas basadas en poblaciones, lo que significa que mantienen un conjunto de soluciones candidatas para el problema considerado. En nuestro caso, se nos ha encomendado la tarea del diseño e implementación de 3 tipos de algoritmos meméticos, bien diferenciados en los parámetros a tener en cuenta en cada uno de ellos:

1. **AM-(10,1.0): Cada 10 generaciones, se aplica la BL sobre todos los cromosomas de la población.**
   Este tipo de algoritmo memético, se encarga de realizar 10 generaciones del algoritmo genético generacional uniforme y posteriormente un proceso de Búsqueda Local sobre todos los cromosomas de la población actual en la que nos encontremos tras dichas 10 ejecuciones del ciclo evolutivo del AGG.

2. **AM-(10,0.1): Cada 10 generaciones, se aplica la BL sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente con probabilidad pLS igual a 0.1 para cada cromosoma.**
   Este tipo de algoritmo memético, por otro lado, se encarga de realizar 10 generaciones del algoritmo genético generacional uniforme y posteriormente un proceso de Búsqueda Local sobre un subconjunto de cromosomas de la población actual en la que nos encontremos tras dichas 10 ejecuciones del ciclo evolutivo del AGG aunque esta vez, aplicaremos una probabilidad de 0.1 por cromosoma para realizar la BL sobre este o pasar al siguiente.

3. **AM-(10,0.1mej): Cada 10 generaciones, aplicar la BL sobre los 0.1·N mejores cromosomas de la población actual (N es el tamaño de ésta).**
   Por último, este algoritmo realizará una BL cada 10 generaciones del AGG sobre los 0.1 * N mejores primeros cromosomas de la población actual, condicionando aún más la aplicación de la BL sobre el conjunto solución en el que nos encontremos.

En cuanto a los parámetros de los diferentes algoritmos meméticos, creo que deberíamos considerar que el tamaño de la población del AGG será de 10 cromosomas, además de que la probabilidad de realizar un cruce entre dos padres será de 0.7 y que la probabilidad de mutación por gen será de 0.001.

Para llevar a cabo la ejecución de la parte de la BL, tendremos que transformar el tipo de representación de la solución de representación binaria a representación entera, y esta será detenida cuando no se encuentre mejora en todo el entorno del cromosoma en el que estamos tratando de que explote sus prestaciones o bien cuando se hayan evaluado 400 vecinos distintos en la ejecución. Por otro lado, nuestro AM se detendrá tras realizar 100000 evaluaciones de la función objetivo, incluyendo las de la BL.

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 AM-1:

```
Inicio
       Mientras cambio < limite
       1. Ejecución AGG-uniforme
       2. Tranformar representación binaria a entera
       3. Generar contador <- 0 para contabilizar número de BL realizadas
       Mientras contador < m y cambio%10==0 y cambio!=0
              4. Ejecución BL (vecDistancias, Limite=400, listaSol, listaCandidatos)
              5. Transformar representación entera a binaria e incluir en listaSol
              6. Actualizar listaAportes de listaSol
              7. contador <- contador+1
       Endwhile
       8. cambio <- cambio + 1
Fin
```

Esquema de búsqueda por AM-2:

```
Inicio
       Mientras cambio < limite
          4. Ejecución AGG-uniforme
          5. TamSubconjunto = 0.1 * m (5 cromosomas)
          6. Generar inicioNuevoSub aleatoriamente entre (0, m-5)
          7. Generar finalNuevoSub aleatoriamente entre (5, m)
          Mientras finalNuevoSub – inicioNuevoSub ¡= tamSubConjunto
                8. Generar inicioNuevoSub aleatoriamente entre (0, m-5)
                9. Generar finalNuevoSub aleatoriamente entre (5, m)
          Endwhile
         10. Generar Subcadena <- conjunto de cromosomas entre inicioNuevo y finalNuevo
         11. Tranformar representación binaria a entera
         12. Generar contador <- 0 para contabilizar número de BL realizadas
          Mientras contador < m y cambio%10==0 y cambio!=0
                13. Ejecución BL (vecDistancias, Limite=400, subCadena,
          listaCandidatos)
             14. Transformar representación entera a binaria e incluir en listaSol
             15. Actualizar listaAportes de listaSol
             16. contador <- contador+1
          Endwhile
          17. Cambio <- cambio+1
       endwhile
Fin
```

Esquema de búsqueda por AM-3:

```
Inicio
       Mientras cambio < limite
          1. Ejecución AGG-uniforme
          2. Ordenacion listaSol por aportes
          3. TamSubconjunto = 0.1 * m (5 cromosomas)
          4. Generar Cromosomas <- conjunto de mejores 5 cromosomas en listaSol en
          representación entera.
          5. Generar CandidatosCromo <- Conjunto de candidatos enteros no seleccionados por
          cada cromosoma.
          6. Generar contador <- 0 para contabilizar número de BL realizadas
          Mientras contador < tamSubConjunto y cambio%10==0 y cambio!=0
             7. Ejecución BL (vecDistancias, Limite=400, Cromosomas[0], CandidatosCromo[0])
             8. Cromosomas <- siguiente en Cromosomas
             9. CandidatosCromo <- siguiente en CandidatosCromo
             10. Transformar representación entera a binaria e incluir en listaSol
             11. Actualizar listaAportes de listaSol
             12. contador <- contador+1
          Endwhile
       13. Cambio <- cambio+1
       endwhile
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

En vista a los resultados obtenidos durante el desarrollo de ambas prácticas, podemos ver cómo en cuanto a la desviación; nuestro algoritmo Greedy y de Búsqueda Local siguen siendo mejores en promedio en cuanto a desviación del óptimo que los algoritmos de búsqueda basadas en poblaciones, aunque creo que cabe destacar, la manera en que la hibridación de algoritmos genéticos con Búsqueda Local puede resultar muy útil a la hora de tratar de escapar de óptimos locales en los que podría caer un algoritmo genético produciendo incluso, soluciones suficientemente satisfactorias en un tiempo más o menos razonable o en un número de iteraciones mucho menor que en el que lo haría un algoritmo Greedy o de Búsqueda Local por el primer mejor. En cuanto a la optimalidad de los resultados de los algoritmos genéticos, no he podido llegar a comprobar todo su potencial, ya que obtenía tiempos muy poco razonables, por lo que me tuve que limitar al número de generaciones máxima que me permitía obtener resultados “factibles”. En mi opinión, me ha resultado bastante interesante el hecho de poder generar un algoritmo como son los meméticos, que mediante la hibridación con otros tipos de algoritmos de exploración y explotación han sido capaces de obtener buenas soluciones y en tiempos bastante más razonables que aquellos generados por los algoritmos genéticos o los desarrollados durante la primera práctica.

Por otro lado, y en lo referente al tiempo promedio, encontré una serie de problemas para poder realizar ejecuciones en tiempos razonables obteniendo soluciones suficientemente buenas, sobre todo en el algoritmo genético uniforme, y por consiguiente en los algoritmos meméticos que usan la estructura de estos combinados con los de búsqueda local. Los tiempos obtenidos en estos algoritmos de búsqueda basados en poblaciones son bastante parecidos, lo que hace que si nos tuviéramos que decantar por uno, en mi opinión sería el algoritmo memético que realiza, cada 10 iteraciones, una búsqueda local sobre toda la población, aunque pudiera consumir algo más de tiempo, ahora bien, deberíamos de atender además a la relación desviación-tiempo que estemos buscando, ya que si el tiempo de computo fuera extremadamente significativo, deberíamos de decantarnos por la opción que obtiene soluciones algo menos buenas pero en un mejor tiempo, como es el algoritmo memético encargado de ejecutar la Búsqueda local sobre los $0.1 \cdot N$ mejores cromosomas cada 10 generaciones realizadas por los algoritmos genéticos.

BIBLIOGRAFÍA:

- Guión de la práctica:
https://pradogrado1920.ugr.es/pluginfile.php/395025/mod_resource/content/4/Guion%20P2a%20Poblaciones%20MDP%20MHs%202019-20.pdf

- Seminario 3 de la asignatura:
https://pradogrado1920.ugr.es/pluginfile.php/395054/mod_resource/content/7/Sem03-Problemas-Poblaciones-MHs-19-20.pdf

- Tema 3 (parte uno y parte dos):

https://pradogrado1920.ugr.es/pluginfile.php/394949/mod_resource/content/8/Tema03-Metaheur%C3%ADsticas%20basadas%20en%20poblaciones%20-%20Parte%20I%20-%2019-20.pdf

https://pradogrado1920.ugr.es/pluginfile.php/394950/mod_resource/content/9/Tema03-Metaheur%C3%ADsticas%20basadas%20en%20poblaciones-variables%20reales%20-%20Parte%20II%20-19-20.pdf

- Wikipedia

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

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